Introduction¶

This is the documentation for *fish*, the **f**riendly **i**nteractive **sh**ell.

A shell is a program which helps you operate your computer by starting other programs. fish offers a command-line interface focused on usability and interactive use.

Unlike other shells, fish does not follow the POSIX standard, but still roughly belongs to the same family.

Some of the special features of fish are:

- **Extensive UI**: syntax highlighting, autosuggestions, tab completion and selection lists that can be navigated and filtered.
- **No configuration needed**: fish is designed to be ready to use immediately, without requiring extensive configuration.
- **Easy scripting**: new functions can be added on the fly. The syntax is easy to learn and use.

This bit of the documentation is a quick guide on how to get going. If you are new to this, see the tutorial.

Installation and Start¶

This section is on how to install, uninstall, start and exit a fish shell and on how to make fish the default shell:

- Installation: How to install fish
- Starting and Exiting How to start and exit a fish shell
- Executing Bash: How to execute bash commands in fish
- Default Shell: How to switch to fish as the default shell Uninstalling: How to
- uninstall fish

Installation¶

Up-to-date instructions for installing the latest version of fish are on the fish homepage.

To install the development version of fish see the instructions at the project's GitHub page.

Starting and Exiting¶

Once fish has been installed, open a terminal. If fish is not the default shell:

- Enter `fish` to start a fish shell:

```
> fish
```

- Enter `exit` to exit a fish shell:

```
> exit
```

Executing Bash¶

If fish is your default shell and you want to copy commands from the internet that are written in a different shell language, bash for example, you can proceed in the following way:

Consider, that `bash` is also a command. With `man bash` you can see that there are two ways to do this:

- `bash` has a switch `-c` to read from a string:

  `> bash -c SomeBashCommand` or `bash` without a switch, opens a bash shell that you can

use and `exit` afterwards.

Default Shell¶

You can make fish your default shell by adding fish's executable in two places: - add `/usr/local/bin/fish` to `/etc/shells` - change your default shell with `chsh -s` to `/usr/local/bin/fish`

For for detailed instructions see Switching to fish.

Uninstalling¶

For uninstalling fish: see FAQ: Uninstalling fish.

Shebang Line¶

Since scripts for shell commands can be written in many different languages, they need to carry information about what interpreter is needed to execute them: For this they are expected to have a first line, the shebang line, which names an executable for this purpose:

Example:

A scripts written in `bash` it would need a first line like this:

```
#!/bin/bash
```

This line tells the shell to execute the file with the bash interpreter, that is located at the path `/bin/bash`.

For a script, written in another language, just replace the interpreter `/bin/bash` with the language interpreter of that other language (for example `/bin/python` for a `python` script) This line is only needed when scripts are executed without specifying the interpreter. For functions inside fish or when executing a script with `` `fish /path/to/script `` they aren't required (but don't hurt either!).

Syntax overview¶

Shells like fish are used by giving them commands. Every `fish` command follows the same basic syntax.

A command is executed by writing the name of the command followed by any arguments.

Example: `echo`

```
hello world
```

This calls the `echo` command. `echo` is a command which will write its arguments to the screen. In the example above, the output will be 'hello world'. Everything in fish is done with commands. There are commands for performing a set of commands multiple times, commands for assigning variables, commands for treating a group of commands as a single command, etc.. And every single command follows the same basic syntax.

If you want to find out more about the echo command used above, read the manual page for the echo command by writing: `man echo`

`man` is a command for displaying a manual page on a given topic. The man command takes the name of the manual page to display as an argument. There are manual pages for almost every command on most computers. There are also manual pages for many other things, such as system libraries and important files.

Every program on your computer can be used as a command in `fish`. If the program file is located in one of the directories in the PATH, you can just use the name of the program to use it. Otherwise the whole filename, including the directory (like `/home/me/code/checkers/checkers` or `../checkers`) has to be used. Here is a list of some useful commands:

- `cd`, change the current directory `ls`, list
- files and directories
- `man`, display a manual page on the screen `mv`,
- move (rename) files `cp`, copy files
- `open`, open files with the default application associated with each filetype `less`, list the
- contents of files
-

Commands and parameters are separated by the space character ' '. Every command ends with either a newline (i.e. by pressing the return key) or a semicolon ';'. More than one command can be written on the same line by separating them with semicolons.

A switch is a very common special type of argument. Switches almost always start with one or more hyphens '-' and alter the way a command operates. For example, the '`ls`' command usually lists all the files and directories in the current working directory, but by using the '`-l`' switch, the behavior of '`ls`' is changed to not only display the filename, but also the size, permissions, owner and modification time of each file.

Switches differ between commands and are documented in the manual page for each command. Some switches are common to most command though, for example '`--help`' will usually display a help text, '`-i`' will often turn on interactive prompting before taking action, while '`-f`' will turn it off.

Some common words¶

This is a short explanation of some of the commonly used words in fish.

- **argument** a parameter given to a command

- **builtin** a command that is implemented in the shell. Builtins are commands that are so closely tied to the shell that it is impossible to implement them as external commands.
- **command** a program that the shell can run. In another sense also specifically an external command (i.e. neither a function or builtin).
- **function** a block of commands that can be called as if they were a single command. By using functions, it is possible to string together multiple smaller commands into one more advanced command.
- **job** a running pipeline or command
- **pipeline** a set of commands stringed together so that the output of one command is the input of the next command
- **redirection** an operation that changes one of the input/output streams associated with a job
  switch a special flag sent as an argument to a command that will alter the behavior of the command.
- A switch almost always begins with one or two hyphens.

Quotes¶

Sometimes features such as parameter expansion and character escapes get in the way.
When that happens, the user can write a parameter within quotes, either `'` (single quote) or `"` (double quote). There is one important difference between single quoted and double quoted strings: When using double quoted string, variable expansion still takes place. Other than that, no other kind of expansion (including brace expansion and parameter expansion) will take place, the parameter may contain spaces, and escape sequences are ignored.

The only backslash escape accepted within single quotes is `\'`, which escapes a single quote and `\\`, which escapes the backslash symbol. The only backslash escapes accepted within double quotes are `\"`, which escapes a double quote, `\$`, which escapes a dollar character, `\` followed by a newline, which deletes the backslash and the newline, and lastly `\\`, which escapes the backslash symbol. Single quotes have no special meaning within double quotes and vice versa.

Example: `rm "cumbersome filename.txt"`

Will remove the file 'cumbersome filename.txt', while `rm cumbersome filename.txt` would remove the two files 'cumbersome' and 'filename.txt'.

Escaping characters¶

Some characters can not be written directly on the command line. For these characters, so called escape sequences are provided. These are:

- \a represents the alert character
- \b represents the backspace character \e
- represents the escape character
- \f represents the form feed character
- \n represents a newline character
- \r represents the carriage return character \t
- represents the tab character \v represents the
- vertical tab character
- \ escapes the space character
- \$ escapes the dollar character
- \\ escapes the backslash character

- * escapes the star character
- \? escapes the question mark character (this is not necessary if the *qmark-noglob* feature flag is enabled) \~ escapes the tilde character
- \# escapes the hash character
- \(escapes the left parenthesis character
- \) escapes the right parenthesis character \{
- escapes the left curly bracket character
- \} escapes the right curly bracket character
- \[escapes the left bracket character
- \] escapes the right bracket character \\<
- escapes the less than character
- \\> escapes the more than character
- \^ escapes the circumflex character
- \& escapes the ampersand character
- \| escapes the vertical bar character
- \; escapes the semicolon character
- \" escapes the quote character
- \' escapes the apostrophe character
- \xHH, where *HH* is a hexadecimal number, represents the ascii character with the specified value. For example, \x9 is the tab character.
- \XHH, where *HH* is a hexadecimal number, represents a byte of data with the specified value. If you are using a multibyte encoding, this can be used to enter invalid strings.
  Only use this if you know what you are doing.
- \ooo, where *ooo* is an octal number, represents the ascii character with the specified value. For example, \011 is the tab character.
- \uXXXX, where *XXXX* is a hexadecimal number, represents the 16-bit Unicode character with the specified value. For example, \u9 is the tab character.
- \UXXXXXXXX, where *XXXXXXXX* is a hexadecimal number, represents the 32-bit Unicode character with the specified value. For example, \U9 is the tab character.
- \cX, where *X* is a letter of the alphabet, represents the control sequence generated by pressing the control key and the specified letter. For example, \ci is the tab character

Input/Output Redirection¶

Most programs use three input/output [1] streams, each represented by a number called a file descriptor (FD). These are:

- Standard input, FD 0, for reading, defaults to reading from the keyboard.
- Standard output, FD 1, for writing, defaults to writing to the screen.
- Standard error, FD 2, for writing errors and warnings, defaults to writing to the screen.

Any file descriptor can be directed to a different output than its default through a mechanism called a redirection.

An example of a file redirection is `echo hello > output.txt`, which directs the output of the echo command to the file output.txt.

- To read standard input from a file, write `<SOURCE_FILE`
- To write standard output to a file, write `>DESTINATION` To
- write standard error to a file, write `2>DESTINATION` [2]
- To append standard output to a file, write `>>DESTINATION_FILE`
- To append standard error to a file, write `2>>DESTINATION_FILE`

To not overwrite ("clobber") an existing file, write `>?DESTINATION` or `2>?DESTINATION` (this is also known as the "noclobber" redirection)

`DESTINATION` can be one of the following:

- A filename. The output will be written to the specified file.
- An ampersand (`&`) followed by the number of another file descriptor. The output will be written to that file descriptor instead.
- An ampersand followed by a minus sign (`&-`). The file descriptor will be closed.

As a convenience, the redirection `&>` can be used to direct both stdout and stderr to the same file.

Example:

To redirect both standard output and standard error to the file 'all_output.txt', you can write `echo Hello &> all_output.txt`, which is a convenience for `echo Hello > all_output.txt 2>&1`. Any file descriptor can be redirected in an arbitrary way by prefixing the redirection with the file descriptor.

- To redirect input of FD N, write `N<DESTINATION`
- To redirect output of FD N, write `N>DESTINATION`
- To append the output of FD N to a file, write `N>>DESTINATION_FILE`

Example: `echo Hello 2>output.stderr` writes the standard error (file descriptor 2) of the target program to `output.stderr`.

[1]Also shortened as "I/O" or "IO".

[2]Previous versions of fish also allowed spelling this as `^DESTINATION`, but that made another character special so it was deprecated and will be removed in future.

Piping¶

The user can string together multiple commands into a *pipeline*. This means that the standard output of one command will be read in as standard input into the next command. This is done by separating the commands by the pipe character '`|`'. For example `cat foo.txt | head` will call the `cat` program with the parameter 'foo.txt', which will print the contents of the file 'foo.txt'. The contents of foo.txt will then be filtered through

the program 'head', which will pass on the first ten lines of the file to the screen. For more information on how to combine commands through pipes, read the manual pages of the commands you want to use using the `man` command. If you want to find out more about the `cat` program, type `man cat`.

Pipes usually connect file descriptor 1 (standard output) of the first process to file descriptor 0 (standard input) of the second process. It is possible to use a different output file descriptor by prepending the desired FD number and then output redirect symbol to the pipe. For example: `make fish 2>| less` will attempt to build the fish program, and any errors will be shown using the less pager.

As a convenience, the pipe `&|` may be used to redirect both stdout and stderr to the same process. (Note this is different from bash, which uses `|&`).

Background jobs¶

When you start a job in `fish`, `fish` itself will pause, and give control of the terminal to the program just started. Sometimes, you want to continue using the commandline, and have the job run in the background. To create a background job, append an & (ampersand) to your command. This will tell fish to run the job in the background. Background jobs are very useful when running programs that have a graphical user interface.

Example: `emacs &` will start the emacs text editor in the

background.

Job control¶

Most programs allow you to suspend the program's execution and return control to `fish` by pressing `Control+Z` (also referred to as `^Z`). Once back at the `fish` commandline, you can start other programs and do anything you want. If you then want you can go back to the suspended command by using the fg (foreground) command.

If you instead want to put a suspended job into the background, use the bg command.

To get a listing of all currently started jobs, use the jobs command. These listed jobs can be removed with the disown command.

Functions¶

Functions are programs written in the fish syntax. They group together one or more commands and their arguments using a single name. It can also be used to start a specific command with additional arguments.

For example, the following is a function definition that calls the command `ls` with the argument '`-l`' to print a detailed listing of the contents of the current directory:

```
function ll
    ls -l $argv
end
```

The first line tells fish that a function by the name of `ll` is to be defined. To use it, simply write `ll` on the commandline. The second line tells fish that the command `ls -l $argv` should be called when `ll` is invoked. '`$argv`' is a list variable, which always contains all arguments sent to the function. In the example above, these are simply passed on to the `ls` command. For more information on functions, see the documentation for the function builtin.

Defining aliases¶

One of the most common uses for functions is to slightly alter the behavior of an already existing command. For example, one might want to redefine the `ls` command to display colors. The switch for turning on colors on GNU systems is '`--color=auto`'. An alias, or wrapper, around `ls` might look like this:

```
function ls
    command ls --color=auto $argv
end
```

There are a few important things that need to be noted about aliases:

- Always take care to add the `$argv` variable to the list of parameters to the wrapped command. This makes sure that if the user specifies any additional parameters to the function, they are passed on to the underlying command.
- If the alias has the same name as the aliased command, you need to prefix the call to the program with `command` to tell fish that the function should not call itself, but rather a command with the same name. If you forget to do so, the function would call itself until the end of time. Usually fish is smart enough to figure this out and will refrain from doing so (which is hopefully in your interest).
- Autoloading isn't applicable to aliases. Since, by definition, the function is created at the time the alias command is executed. You cannot autoload aliases.

To easily create a function of this form, you can use the alias command.

Autoloading functions¶

Functions can be defined on the commandline or in a configuration file, but they can also be automatically loaded. This has some advantages:

- An autoloaded function becomes available automatically to all running shells. If the function
- definition is changed, all running shells will automatically reload the altered version. Startup time and memory usage is improved, etc.
-

When fish needs to load a function, it searches through any directories in the list variable `$fish_function_path` for a file with a name consisting of the name of the function plus the suffix '`.fish`' and loads the first it finds.

By default `$fish_function_path` contains the following:

- A directory for end-users to keep their own functions, usually `~/.config/fish/functions` (controlled by the `XDG_CONFIG_HOME` environment variable).
- A directory for systems administrators to install functions for all users on the system,
- usually `/etc/fish/functions` (really `$__fish_sysconfdir/functions`).
  Directories for third-party software vendors to ship their own functions for their software. Fish searches the directories in the `XDG_DATA_DIRS` environment variable for a `fish/vendor_functions.d` directory; if this variable is not defined, the default is usually to search `/usr/share/fish/vendor_functions.d` and

  `/usr/local/share/fish/vendor_functions.d`.
- The functions shipped with fish, usually installed in `/usr/share/fish/functions` (really

  `$__fish_data_dir/functions`).

These paths are controlled by parameters set at build, install, or run time, and may vary from the defaults listed above.

This wide search may be confusing. If you are unsure, your functions probably belong in `~/.config/fish/functions`.

It is very important that function definition files only contain the definition for the specified function and nothing else. Otherwise, it is possible that autoloading a function files requires that the function already be loaded, which creates a circular dependency.

Autoloading also won't work for event handlers, since fish cannot know that a function is supposed to be executed when an event occurs when it hasn't yet loaded the function. See the event handlers section for more information.

Autoloading is not applicable to functions created by the `alias` command. For functions simple enough that you prefer to use the `alias` command to define them you'll need to put those commands in your `~/.config/fish/config.fish` script or some other script run when the shell starts.

If you are developing another program, you may wish to install functions which are available for all users of the fish shell on a system. They can be installed to the "vendor" functions directory. As this path may vary from system to system, the `pkgconfig` framework should be used to discover this path with the output of `pkg-config --variable functionsdir fish`. Your installation system should support a custom path to override the pkgconfig path, as other distributors may need to alter it easily.

Conditional execution of code and flow control¶

There are four fish builtins that let you execute commands only if a specific criterion is met. These builtins are if, switch, and and or.

The `switch` command is used to execute one of possibly many blocks of commands depending on the value of a string. See the documentation for switch for more information.

The other conditionals use the exit status of a command to decide if a command or a block of commands should be executed. See their documentation for more information.

Parameter expansion (Globbing)¶

When an argument for a program is given on the commandline, it undergoes the process of parameter expansion before it is sent on to the command. Parameter expansion is a powerful mechanism that allows you to expand the parameter in various ways, including performing wildcard matching on files, inserting the value of a shell variable into the parameter or even using the output of another command as a parameter list.

Wildcards¶

If a star (`*`) or a question mark (`?`) is present in the parameter, `fish` attempts to match the given parameter to any files in such a way that:

- `*` can match any string of characters not containing '/'. This includes matching an empty string.
  `**` matches any string of characters. This includes matching an empty string. The matched string may
- include the `/` character; that is, it recurses into subdirectories. Note that augmenting this wildcard

with other strings will not match files in the current working directory (`$PWD`) if you separate the strings

with a slash ("/"). This is unlike other shells such as zsh. For example, `**\/*.fish` in zsh will match `.fish` files in the PWD but in fish will only match such files in a subdirectory. In fish you should type `**.fish` to match files in the PWD as well as subdirectories.

- `?` can match any single character except '/'. This is deprecated and can be disabled via the *qmark-noglob* feature flag, so *?* will just be an ordinary character.

Other shells, such as zsh, provide a rich glob syntax for restricting the files matched by globs. For example, `**(.)`, to only match regular files. Fish prefers to defer such features to programs, such as `find`, rather than reinventing the wheel. Thus, if you want to limit the wildcard expansion to just regular files the fish approach is to define and use a function. For example,

```
function ff --description 'Like ** but only returns plain files.'
    # This also ignores .git directories.     find . \( -
name .git -type d -prune \) -o -type f | \          sed -
n -e '/^\.\/\.git$/n' -e 's/^\.\///p' end
```

You would then use it in place of `**` like this, `my_prog (ff)`, to pass only regular files in or below $PWD to `my_prog`.

Wildcard matches are sorted case insensitively. When sorting matches containing numbers, consecutive digits are considered to be one element, so that the strings '1' '5' and '12' would be sorted in the order given.

File names beginning with a dot are not considered when wildcarding unless a dot is specifically given as the first character of the file name.

Examples:

- `a*` matches any files beginning with an 'a' in the current directory.
- `???` matches any file in the current directory whose name is exactly three characters long.
  `**` matches any files and directories in the current directory and all of its subdirectories.
-

Note that for most commands, if any wildcard fails to expand, the command is not executed, $status is set to nonzero, and a warning is printed. This behavior is consistent with setting `shopt -s failglob` in bash. There are exactly 4 exceptions, namely set, overriding variables in overrides, count and for. Their globs are permitted to expand to zero arguments, as with `shopt -s nullglob` in bash. Examples:

```
ls *.foo
# Lists the .foo files, or warns if there aren't any.

set foos *.foo if count
$foos >/dev/null
    ls $foos end # Lists the
.foo files, if any.
```

Command substitution¶

The output of a series of commands can be used as the parameters to another command. If a parameter contains a set of parenthesis, the text enclosed by the parenthesis will be interpreted as a list of commands. On expansion, this list is executed, and substituted by the output. If the output is more than one line long, each line will be expanded to a new parameter. Setting `IFS` to the empty string will disable line splitting.

If the output is piped to string split or string split0 as the last step, those splits are used as they appear and no additional splitting on newlines takes place.

The exit status of the last run command substitution is available in the status variable if the substitution occurs in the context of a `set` command.

Only part of the output can be used, see index range expansion for details.

Fish has a default limit of 100 MiB on the amount of data a command substitution can output. If the limit is exceeded the entire command, not just the substitution, is failed and `$status` is set to 122. You can modify the limit by setting the `fish_read_limit` variable at any time including in the environment before fish starts running. If you set it to zero then no limit is imposed. This is a safety mechanism to keep the shell from consuming too much memory if a command outputs an unreasonable amount of data, typically your operating system also has a limit, and it's often much lower. Note that this limit also affects how much data the `read` command will process.

Examples:

```
echo (basename image.jpg .jpg).png
# Outputs 'image.png'.

for i in *.jpg; convert $i (basename $i .jpg).png; end
# Convert all JPEG files in the current directory to the
# PNG format using the 'convert' program.

begin; set -l IFS; set data (cat data.txt); end
# Set the ``data`` variable to the contents of 'data.txt'
# without splitting it into a list.

set data (cat data | string split0)
# Set ``$data`` to the contents of data, splitting on NUL-bytes.
```

Brace expansion¶

A comma separated list of characters enclosed in curly braces will be expanded so each element of the list becomes a new parameter. This is useful to save on typing, and to separate a variable name from surrounding text.

Examples:

```
> echo input.{c,h,txt} input.c
input.h input.txt

> mv *.{c,h} src/
# Moves all files with the suffix '.c' or '.h' to the subdirectory src.

> cp file{,.bak}
# Make a copy of `file` at `file.bak`.

> set -l dogs hot cool cute
> echo {$dogs}dog hotdog
cooldog cutedog
```

If two braces do not contain a "," or a variable expansion, they will not be expanded in this manner:

```
> echo foo-{} foo-{}
> git reset --hard HEAD@{2}
# passes "HEAD@{2}" to git
> echo {{a,b}}
{a} {b} # because the inner brace pair is expanded, but the outer isn't.
```

If after expansion there is nothing between the braces, the argument will be removed (see the cartesian product section):

```
> echo foo-{$undefinedvar}
# Output is an empty line, just like a bare `echo`.
```

If there is nothing between a brace and a comma or two commas, it's interpreted as an empty element:

```
> echo {,,/usr}/bin /bin
/bin /usr/bin
```

To use a "," as an element, quote or escape it.

Variable expansion¶

A dollar sign followed by a string of characters is expanded into the value of the shell variable with the same name. For more on shell variables, read the Shell variables section.

Examples:

```
echo $HOME
# Prints the home directory of the current user.
```

Undefined

and empty variables expand to nothing:

```
echo $nonexistentvariable
# Prints no output.
```

To separate a variable name from text encase the variable within double-quotes or braces:

```
echo The plural of $WORD is "$WORD"s
# Prints "The plural of cat is cats" when $WORD is set to cat.
echo The plural of $WORD is {$WORD}s
# ditto
```

Note that without the quotes or braces, fish will try to expand a variable called `$WORDs`, which may not exist.

The latter syntax `{$WORD}` works by exploiting brace expansion.

In these cases, the expansion eliminates the string, as a result of the implicit cartesian product. If, in the example above, $WORD is undefined or an empty list, the "s" is not printed.

However, it is printed if $WORD is the empty string (like after `set WORD ""`).

Unlike all the other expansions, variable expansion also happens in double quoted strings. Inside double quotes (`"these"`), variables will always expand to exactly one argument. If they are empty or undefined, it will result in an empty string. If they have one element, they'll expand to that element. If they have more than that, the elements will be joined with spaces [3].

Outside of double quotes, variables will expand to as many arguments as they have elements. That means an empty list will expand to nothing, a variable with one element will expand to that element, and a variable with multiple elements will expand to each of those elements separately.

When two unquoted expansions directly follow each other, you need to watch out for expansions that expand to nothing. This includes undefined variables and empty lists, but also command substitutions with no output. See the cartesian product section for more information.

The `$` symbol can also be used multiple times, as a kind of "dereference" operator (the `*` in C or C++), like in the following code:

```
set foo a b c
set a 10; set b 20; set c 30
for i in (seq (count
$$foo))      echo $$foo[$i]
end
 # Output
is: # 10
# 20
# 30
```

When using this feature together with list brackets, the brackets will always match the innermost `$` dereference. Thus, `$$foo[5]` will always mean the fifth element of the `foo` variable should be dereferenced, not the fifth element of the doubly dereferenced variable `foo`. The latter can instead be expressed as `$$foo[1][5]`.

[3]Unlike bash or zsh, which will join with the first character of $IFS (which usually is space).

Cartesian Products¶

Lists adjacent to other lists or strings are expanded as cartesian products:

Examples:

```
>_ echo {good,bad}" apples" good
apples bad apples


>_ set -l a x y z
>_ set -l b 1 2 3


>_ echo $a$b
x1 y1 z1 x2 y2 z2 x3 y3 z3


>_ echo $a"-"$b
x-1 y-1 z-1 x-2 y-2 z-2 x-3 y-3 z-3


>_ echo {x,y,z}$b x1 y1 z1
x2 y2 z2 x3 y3 z3


>_ echo {$b}word
1word 2word 3word


>_ echo {$c}word
# Output is an empty line
```

Be careful when you try to use braces to separate variable names from text. The problem shown above can be avoided by wrapping the variable in double quotes instead of braces

(`echo "$c"word`).

This also happens after command substitution. Therefore strings might be eliminated. This can be avoided by making the inner command return a trailing newline. E.g.

```
>_ echo (printf '%s' '')banana # the printf prints literally nothing
>_ echo (printf '%s\n' '')banana # the printf prints just a newline, so the command
substitu banana
```

```
# After command substitution, the previous line looks like:
```

```
>_ echo ""banana
```

Examples:

```
>_ set b 1 2 3 >_
echo (echo x)$b x1
x2 x3
```

Index range expansion¶

Sometimes it's necessary to access only some of the elements of a list, or some of the lines a command substitution outputs. Both allow this by providing a set of indices in square brackets. Sequences of elements can be written with the range operator '`..`'. A range '`a..b`' ('a' and 'b' being integers) is expanded into a sequence of indices '`a a+1 a+2 ... b`' or '`a a-1 a-2 ... b`' depending on which of 'a' or 'b' is higher. Negative range limits are calculated from the end of the list. If an index is too large or small it's silently clamped to one or the size of the list as appropriate.

If the end is smaller than the start, or the start is larger than the end, range expansion will go in reverse. This is unless exactly one of the given indices is negative, so the direction doesn't change if the list has fewer elements than expected.

Some examples:

```
echo (seq 10)[1 2 3]
# Prints: 1 2 3

# Limit the command substitution output
echo (seq 10)[2..5] # Uses elements
from 2 to 5
# Output is: 2 3 4 5

# Use overlapping ranges:
echo (seq 10)[2..5 1..3]
# Takes elements from 2 to 5 and then elements from 1 to 3
# Output is: 2 3 4 5 1 2 3

# Reverse output
echo (seq 10)[-1..1]
# Uses elements from the last output line to
# the first one in reverse direction
# Output is: 10 9 8 7 6 5 4 3 2 1

# The command substitution has only one line,
# so these will result in empty output:
echo (echo one)[2..-1]
echo (echo one)[-3..1]
```

The same works when setting or expanding variables:

```
# Reverse path variable
set PATH $PATH[-1..1]
# or
set PATH[-1..1] $PATH

# Use only n last items of the PATH
set n -3 echo $PATH[$n..-1]
```

Variables can be used as indices for expansion of variables, like so:

```
set index 2 set letters a b c d
echo $letters[$index] # returns
'b'
```

However using variables as indices for command substitution is currently not supported, so:

```
echo (seq 5)[$index] # This won't work
 set sequence (seq 5) # It needs to be written on two lines like
this. echo $sequence[$index] # returns '2'
```

When using indirect variable expansion with multiple *$* (`$$name`), you have to give all indices up to the variable you want to slice:

```
> set -l list 1 2 3 4 5
> set -l name list
> echo $$name[1]
1 2 3 4 5
> echo $$name[1..-1][1..3] # or $$name[1][1..3], since $name only has one element.
1 2 3
```

Home directory expansion¶

The ~ (tilde) character at the beginning of a parameter, followed by a username, is expanded into the home directory of the specified user. A lone ~, or a ~ followed by a slash, is expanded into the home directory of the process owner.

Combining different expansions¶

All of the above expansions can be combined. If several expansions result in more than one parameter, all possible combinations are created.

When combining multiple parameter expansions, expansions are performed in the following order:

- Command substitutions
- Variable expansions Bracket
- expansion
- Wildcard expansion

Expansions are performed from right to left, nested bracket expansions are performed from the inside and out.

Example:

If the current directory contains the files 'foo' and 'bar', the command `echo a(ls){1,2,3}` will output `abar1 abar2 abar3 afoo1 afoo2 afoo3`.

Shell variable and function names¶

The names given to shell objects like variables and function names are known as "identifiers". Each type of identifier has rules that define what sequences of characters are valid to use.

A variable name cannot be empty. It can contain only letters, digits, and underscores. It may begin and end with any of those characters.

A function name cannot be empty. It may not begin with a hyphen ("-") and may not contain a slash ("/"). All other characters, including a space, are valid.

A bind mode name (e.g., `bind -m abc ...`) is restricted to the rules for valid variable names.

Shell variables¶

Shell variables are named pieces of data, which can be created, deleted and their values changed and used by the user. Variables may optionally be "exported", so that a copy of the variable is available to any subprocesses the shell creates. An exported variable is referred to as an "environment variable".

To set a variable value, use the set command. A variable name can not be empty and can contain only letters, digits, and underscores. It may begin and end with any of those characters.

Example:

To set the variable `smurf_color` to the value `blue`, use the command `set smurf_color blue`.

After a variable has been set, you can use the value of a variable in the shell through variable expansion.

Example: To use the value of the variable `smurf_color`, write `$` (dollar symbol) followed by the name of the variable, like `echo Smurfs are usually $smurf_color`, which would print the result 'Smurfs are usually blue'.

Variable scope¶

There are three kinds of variables in fish: universal, global and local variables.

- Universal variables are shared between all fish sessions a user is running on one computer.
- Global variables are specific to the current fish session, but are not associated with any specific block scope, and will never be erased unless the user explicitly requests it using `set -e`.
- Local variables are specific to the current fish session, and associated with a specific block of commands, and is automatically erased when a specific block goes out of scope. A block of commands is a series of commands that begins with one of the commands `for`, `while` , `if`, `function`, `begin` or `switch`, and ends with the command `end`.

Variables can be explicitly set to be universal with the `-U` or `--universal` switch, global with the `-g` or `--global` switch, or local with the `-l` or `--local` switch. The scoping rules when creating or updating a variable are:

- If a variable is explicitly set to a scope (universal, global or local), that setting will be honored. If a variable of the same name exists in a different scope, that variable will not be changed.
- If a variable is not explicitly set to a scope, but has been previously defined, the variable scope is not changed.
- If a variable is not explicitly set to a scope and has not been defined, the variable will be local to the currently executing function. Note that this is different from using the `-l` or `--local` flag. If one of those flags is used, the variable will be local to the most inner currently executing block, while without these the variable will be local to the function. If no function is executing, the variable will be global.

There may be many variables with the same name, but different scopes. When using a variable, the variable scope will be searched from the inside out, i.e. a local variable will be used rather than a global variable with the same name, a global variable will be used rather than a universal variable with the same name.

Example:

The following code will not output anything:

```
begin
    # This is a nice local scope where all variables will die
set -l pirate 'There be treasure in them thar hills' end
echo $pirate
# This will not output anything, since the pirate was local
```

Overriding variables for a single command¶

If you want to override a variable for a single command, you can use "var=val" statements before the command:

```
# Call git status on another directory (can also be done via `git -C somerepo status`)
GIT_DIR=somerepo git status
```

Note that, unlike other shells, fish will first set the variable and then perform other expansions on the line, so:

`set foo banana foo=gagaga echo $foo # prints gagaga, while in other shells it might print "banana"` Multiple elements can be given in a brace expansion:

```
# Call bash with a reasonable default path.
PATH={/usr,}/{s,}bin bash
```

This syntax is supported since fish 3.1.

More on universal variables¶

Universal variables are variables that are shared between all the users' fish sessions on the computer. Fish stores many of its configuration options as universal variables. This means that in order to change fish settings, all you have to do is change the variable value once, and it will be automatically updated for all sessions, and preserved across computer reboots and login/logout.

To see universal variables in action, start two fish sessions side by side, and issue the following command in one of them `set fish_color_cwd blue`. Since `fish_color_cwd` is a universal variable, the color of the current working directory listing in the prompt will instantly change to blue on both terminals.

Universal variables are stored in the file `.config/fish/fish_variables`. Do not edit this file directly, as your edits may be overwritten. Edit the variables through fish scripts or by using fish interactively instead.

Do not append to universal variables in config.fish, because these variables will then get longer with each new shell instance. Instead, simply set them once at the command line.

Variable scope for functions¶

When calling a function, all current local variables temporarily disappear. This shadowing of the local scope is needed since the variable namespace would become cluttered, making it very easy to accidentally overwrite variables from another function.

For example:

```
function shiver
    set phrase 'Shiver me timbers'
end
function avast
    set --local phrase 'Avast, mateys'     #
Calling the shiver function here can not
    # change any variables in the local scope
    shiver
echo $phrase end
avast

# Outputs "Avast, mateys"
```

Exporting variables¶

Variables in fish can be "exported", so they will be inherited by any commands started by fish. In particular, this is necessary for variables used to configure external commands like $LESS or $GOPATH, but also for variables that contain general system settings like $PATH or $LANGUAGE. If an external command needs to know a variable, it needs to be exported.

Variables can be explicitly set to be exported with the `-x` or `--export` switch, or not exported with the `-u` or `--unexport` switch. The exporting rules when setting a variable are identical to the scoping rules for variables:

- If a variable is explicitly set to either be exported or not exported, that setting will be honored.
- If a variable is not explicitly set to be exported or not exported, but has been previously defined, the previous exporting rule for the variable is kept.
- Otherwise, by default, the variable will not be exported.
- If a variable has local scope and is exported, any function called receives a _copy_ of it, so any changes it makes to the variable disappear once the function returns. Global variables are accessible to functions whether they are exported or not.

As a naming convention, exported variables are in uppercase and unexported variables are in lowercase.

Lists¶

`fish` can store a list (or an "array" if you wish) of multiple strings inside of a variable. To access one element of a list, use the index of the element inside of square brackets, like this: `echo $PATH[3]`

Note that list indices start at 1 in `fish`, not 0, as is more common in other languages. This is because many common Unix tools like `seq` are more suited to such use. An invalid index is silently ignored resulting in no value being substituted (not an empty string).

If you do not use any brackets, all the elements of the list will be written as separate items. This means you can easily iterate over a list using this syntax: `for i in $PATH; echo $i is in the path; end`

To create a variable `smurf`, containing the items `blue` and `small`, simply write: `set smurf`

```
blue small
```

It is also possible to set or erase individual elements of a list:

```
# Set smurf to be a list with the elements 'blue' and 'small'
set smurf blue small

# Change the second element of smurf to 'evil'
set smurf[2] evil

# Erase the first element
set -e smurf[1]

# Output 'evil'
echo $smurf
```

If you specify a negative index when expanding or assigning to a list variable, the index will be calculated from the end of the list. For example, the index -1 means the last index of a list.

A range of indices can be specified, see index range expansion for details.

All lists are one-dimensional and cannot contain other lists, although it is possible to fake nested lists using the dereferencing rules of variable expansion.

When a list is exported as an environment variable, it is either space or colon delimited, depending on whether it is a path variable:

```
set -x smurf blue small set -x
smurf_PATH forest mushroom env |
grep smurf

# smurf=blue small
# smurf_PATH=forest:mushroom
```

`fish` automatically creates lists from all environment variables whose name ends in PATH, by splitting them on colons. Other variables are not automatically split.

PATH variables¶

Path variables are a special kind of variable used to support colon-delimited path lists including PATH, CDPATH, MANPATH, PYTHONPATH, etc. All variables that end in `PATH` (casesensitive) become PATH variables.

PATH variables act as normal lists, except they are are implicitly joined and split on colons.

```
set MYPATH 1 2 3
echo "$MYPATH" #
1:2:3
set MYPATH "$MYPATH:4:5"
echo $MYPATH # 1 2 3 4 5
echo "$MYPATH" #
1:2:3:4:5
```

Variables can be marked or unmarked as PATH variables via the `--path` and `--unpath` options to `set`.

Special variables¶

The user can change the settings of `fish` by changing the values of certain variables.

- `PATH`, a list of directories in which to search for commands
- `CDPATH`, a list of directories in which to search for the new directory for the `cd` builtin.
- `LANG`, `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC` and `LC_TIME` set the language option for the shell and subprograms. See the section Locale variables for more information.
- A large number of variable starting with the prefixes `fish_color` and `fish_pager_color`. See Variables for changing highlighting colors for more information.
- `fish_emoji_width` controls the computed width of certain characters, in particular emoji, whose rendered width changed in Unicode 9 and hence varies across terminal emulators. This should be set to 2 if your terminal emulator supports Unicode >= 9 and renders them double-width, and 1 otherwise. Set this only if you see graphical glitching when printing emoji, typically it will be automatically detected.
- `fish_ambiguous_width` controls the computed width of ambiguous-width characters. This should be set to 1 if your terminal emulator renders these characters as single-width (typical), or 2 if double-width.
- `fish_escape_delay_ms` overrides the default timeout of 30ms after seeing an escape character before giving up on matching a key binding. This is explained in the documentation for the bind builtin command. This delay facilitates using escape as a meta key.

`fish_greeting`, the greeting message printed on startup.

- `fish_history`, the current history session name. If set, all subsequent commands within an interactive fish session will be logged to a separate file identified by the value of the variable. If unset, or set to `default`, the default session name "fish" is used. If set to an empty string, history is not saved to disk (but is still available within the interactive session).
- `fish_trace`, if set and not empty, will cause fish to print commands before they execute, similar to *set -x* in bash. The trace is printed to the path given by the --debug-output option to fish (stderr by default). `fish_user_paths`, a list of directories that are prepended to `PATH`. This can be a universal
- variable.
  `umask`, the current file creation mask. The preferred way to change the umask variable is through the
- umask function. An attempt to set umask to an invalid value will always fail.
  `BROWSER`, the user's preferred web browser. If this variable is set, fish will use the specified browser instead of the system default browser to display the fish documentation.
-

`fish` also sends additional information to the user through the values of certain environment variables. The user cannot change the values of most of these variables.

- _, the name of the currently running command (though this is deprecated, and the use of `status current-command` is preferred).
- `argv`, a list of arguments to the shell or function. `argv` is only defined when inside a function call, or if fish was invoked with a list of arguments, like `fish myscript.fish foo bar`. This variable can be
- changed by the user. `history`, a list containing the last commands that were entered.
- `HOME`, the user's home directory. This variable can be changed by the user. `hostname`, the
- machine's hostname.
- `IFS`, the internal field separator that is used for word splitting with the read builtin.
  Setting this to the empty string will also disable line splitting in command substitution.
  This variable can be changed by the user. `PWD`, the
- current working directory.
- `status`, the exit status of the last foreground job to exit. If the job was terminated through a signal, the exit status will be 128 plus the signal number.
- `pipestatus`, a list of exit statuses of all processes that made up the last executed pipe. `USER`, the
- current username. This variable can be changed by the user. `CMD_DURATION`, the runtime of the
- last command in milliseconds. `version`, the version of the currently running fish (also available
- as `FISH_VERSION` for backward compatibility).
  `SHLVL`, the level of nesting of shells
- `COLUMNS` and `LINES`, the current size of the terminal in height and width. These values are only used
- by fish if the operating system does not report the size of the terminal. Both variables must be set in that case otherwise a default of 80x24 will be used. They are updated when the window size changes.

The names of these variables are mostly derived from the csh family of shells and differ from the ones used by Bourne style shells such as bash.

Variables whose name are in uppercase are generally exported to the commands started by fish, while those in lowercase are not exported (`CMD_DURATION` is an exception, for historical reasons). This rule is not enforced by fish, but it is good coding practice to use casing to distinguish between exported and unexported variables. `fish` also uses several variables internally. Such variables are prefixed with the string `__FISH` or `__fish`. These should never be used by the user. Changing their value may break fish.

The status variable¶

Whenever a process exits, an exit status is returned to the program that started it (usually the shell). This exit status is an integer number, which tells the calling application how the execution of the command went.

-

In general, a zero exit status means that the command executed without problem, but a non-zero exit status means there was some form of problem.

Fish stores the exit status of the last process in the last job to exit in the `status` variable.

If `fish` encounters a problem while executing a command, the status variable may also be set to a specific value:

- 0 is generally the exit status of fish commands if they successfully performed therequested operation.
- 1 is generally the exit status of fish commands if they failed to perform the requestedoperation.
  121 is generally the exit status of fish commands if they were supplied with invalid arguments.
- 123 means that the command was not executed because the command name containedinvalid characters.
- 124 means that the command was not executed because none of the wildcards in thecommand produced any matches.
- 125 means that while an executable with the specified name was located, the operatingsystem could not actually execute the command.
- 126 means that while a file with the specified name was located, it was not executable.
  127 means that no function, builtin or command with the given name could be located.
-
- If a process exits through a signal, the exit status will be 128 plus the number of the signal.

Variables for changing highlighting colors¶

The colors used by fish for syntax highlighting can be configured by changing the values of a various variables. The value of these variables can be one of the colors accepted by the set_color command. The `--bold` or `-b` switches accepted by `set_color` are also accepted.

The following variables are available to change the highlighting colors in fish:

- `fish_color_normal`, the default color `fish_color_command`, the color for commands
- `fish_color_quote`, the color for quoted blocks of text `fish_color_redirection`, the color
- for IO redirections `fish_color_end`, the color for process separators like ';' and '&'
- `fish_color_error`, the color used to highlight potential errors `fish_color_param`, the color
- for regular command parameters `fish_color_comment`, the color used for code comments
- `fish_color_match`, the color used to highlight matching parenthesis `fish_color_selection`,
- the color used when selecting text (in vi visual mode) `fish_color_search_match`, used to
- highlight history search matches and the selected pager item (must be a background)
- `fish_color_operator`, the color for parameter expansion operators like '*' and '~'
- `fish_color_escape`, the color used to highlight character escapes like '\n' and '\x70'
- `fish_color_cwd`, the color used for the current working directory in the default prompt
  `fish_color_autosuggestion`, the color used for autosuggestions
- `fish_color_user`, the color used to print the current username in some of fish default prompts
-
- `fish_color_host`, the color used to print the current host system in some of fish default prompts
- `fish_color_host_remote`, the color used to print the current host system in some of fish default
- prompts, if fish is running remotely (via ssh or similar) `fish_color_cancel`, the color for the '^C'
- indicator on a canceled command

Additionally, the following variables are available to change the highlighting in the completion pager:

- `fish_pager_color_progress`, the color of the progress bar at the bottom left corner
- `fish_pager_color_background`, the background color of a line
- `fish_pager_color_prefix`, the color of the prefix string, i.e. the string that is to be completed
- `fish_pager_color_completion`, the color of the completion itself `fish_pager_color_description`,
- the color of the completion description
- `fish_pager_color_secondary_background`, `fish_pager_color_background` of every second unselected completion. Defaults to `fish_pager_color_background`
- `fish_pager_color_secondary_ prefix`, `fish_pager_color_prefix` of every second unselected completion. Defaults to `fish_pager_color_prefix`
- `fish_pager_color_secondary_completion`, `fish_pager_color_completion` of every second unselected completion. Defaults to `fish_pager_color_completion`
- `fish_pager_color_secondary_description`, `fish_pager_color_description` of every second unselected completion. Defaults to `fish_pager_color_description`
- `fish_pager_color_selected_background`, `fish_pager_color_background` of the selected completion. Defaults to `fish_color_search_match fish_pager_color_selected_prefix`,
- `fish_pager_color_prefix` of the selected completion. Defaults to `fish_pager_color_prefix`
- `fish_pager_color_selected_completion`, `fish_pager_color_completion` of the selected completion. Defaults to `fish_pager_color_completion`

`fish_pager_color_selected_description`, `fish_pager_color_description` of the selected completion. Defaults to `fish_pager_color_description` Example:

To make errors highlighted and red, use: `set`

```
fish_color_error red --bold
```

Locale variables¶

The most common way to set the locale to use a command like 'set -x LANG en_GB.utf8', which sets the current locale to be the English language, as used in Great Britain, using the UTF-8 character set. For a list of available locales, use 'locale -a'.

`LANG`, `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC` and `LC_TIME` set the language option for the shell and subprograms. These variables work as follows: `LC_ALL` forces all the aspects of the locale to the specified value. If `LC_ALL` is set, all other locale variables will be ignored. The other `LC_` variables set the specified aspect of the locale information. `LANG` is a fallback value, it will be used if none of the `LC_` variables are specified.

Builtin commands¶

Many other shells have a large library of builtin commands. Most of these commands are also available as standalone commands, but have been implemented in the shell anyway. To avoid code duplication, and to avoid the confusion of subtly differing versions of the same command, `fish` generally only implements builtins for actions which cannot be performed by a regular command.

For a list of all builtins, functions and commands shipped with fish, see the list of commands. The documentation is also available by using the `--help` switch of the command.

Interactive use¶

Fish prides itself on being really nice to use interactively. That's down to a few features we'll explain in the next few sections.

Help¶

`fish` has an extensive help system. Use the `help` command to obtain help on a specific subject or command. For instance, writing `help syntax` displays the syntax section of this documentation. `fish` also has man pages for its commands. For example, `man set` will show the documentation for `set` as a man page.

Help on a specific builtin can also be obtained with the `-h` parameter. For instance, to obtain help on the `fg` builtin, either type `fg -h` or `help fg`.

Autosuggestions¶

fish suggests commands as you type, based on command history, completions, and valid file paths. As you type commands, you will see a suggestion offered after the cursor, in a muted gray color (which can be changed with the `fish_color_autosuggestion` variable). To accept the autosuggestion (replacing the command line contents), press right arrow or `Control+F`. To accept the first suggested word, press `Alt+→`, `Right` or `Alt+F`. If the autosuggestion is not what you want, just ignore it: it won't execute unless you accept it.

Autosuggestions are a powerful way to quickly summon frequently entered commands, by typing the first few characters. They are also an efficient technique for navigating through directory hierarchies.

Tab Completion¶

Tab completion is one of the most time saving features of any modern shell. By tapping the tab key, the user asks `fish` to guess the rest of the command or parameter that the user is currently typing. If `fish` can only find one possible completion, `fish` will write it out. If there is more than one completion, `fish` will write out the longest prefix that all completions have in common. If the completions differ on the first character, a list of all possible completions is printed. The list features descriptions of the completions and if the list doesn't fit the screen, it is scrollable by using the arrow keys, the page up/page down keys, the tab key or the space bar.

If the list is visible, pressing control-S (or the `pager-toggle-search` binding) will allow filtering the list. Shift-tab (or the `complete-and-search` binding) will trigger completion with the search field immediately visible. These are the general purpose tab completions that `fish` provides:

- Completion of commands (builtins, functions and regular programs).
- Completion of shell variable names.

  Completion of usernames for tilde expansion.
- Completion of filenames, even on strings with wildcards such as '`*`' and '`**`'.

`fish` provides a large number of program specific completions. Most of these completions are simple options like the `-l` option for `ls`, but some are more advanced. The latter include:

- The programs `man` and `whatis` show all installed manual pages as completions. The `make`
- program uses all targets in the Makefile in the current directory as completions.
  The `mount` command uses all mount points specified in fstab as completions.
-
- The `ssh` command uses all hosts that are stored in the known_hosts file as completions.
- (See the ssh documentation for more information)
- The `su` command uses all users on the system as completions.
- The `apt-get`, `rpm` and `yum` commands use all installed packages as completions.

Writing your own completions¶

To specify a completion, use the `complete` command. `complete` takes as a parameter the name of the command to specify a completion for. For example, to add a completion for the program `myprog`, one would start the completion command with `complete -c myprog ...` To provide a list of possible completions for myprog, use the `-a` switch. If `myprog` accepts the arguments start and stop, this can be specified as `complete -c myprog -a 'start stop'`. The argument to the `-a` switch is always a single string. At completion time, it will be tokenized on spaces and tabs, and variable expansion, command substitution and other forms of parameter expansion will take place. `fish` has a special syntax to support specifying switches accepted by a command. The switches `-s`, `-l` and `-o` are used to specify a short switch (single character, such as `-l`), a gnu style long switch (such as '`--color`') and an old-style long switch (like '`-shuffle`'), respectively. If the command 'myprog' has an option '-o' which can also be written as '`--output`', and which can take an additional value of either 'yes' or 'no', this can be specified by writing: `complete -c myprog -s o -l output -a "yes no"`

There are also special switches for specifying that a switch requires an argument, to disable filename completion, to create completions that are only available in some combinations, etc.. For a complete description of the various switches accepted by the `complete` command, see the documentation for the complete builtin, or write `complete --help` inside the `fish` shell.

As a more comprehensive example, here's a commented excerpt of the completions for systemd's `timedatectl`:

```
# All subcommands that timedatectl knows - this is useful for later.
set -l commands status set-time set-timezone list-timezones set-local-rtc set-ntp
# Disable file completions for the entire command
# because it does not take files anywhere # Note
that this can be undone by using "-F". #
# File completions also need to be disabled
# if you want to have more control over what files are offered (e.g. just directories, or
ju complete -c timedatectl -f

# This line offers the subcommands
# -"status",
# -"set-timezone",
# -"set-time"
# -"list-timezones" # if no subcommand
has been given so far.
#
# The `-n`/`--condition` option takes script as a string, which it executes.
# If it returns true, the completion is offered.
# Here the condition is the `__fish_seen_subcommands_from` helper function.
# If returns true if any of the given commands is used on the commandline,
# as determined by a simple heuristic.
```

```
# For more complex uses, you can write your own function.
# See e.g. the git completions for an example.
#
complete -c timedatectl -n "not __fish_seen_subcommand_from $commands" -a "status set-time
s
# If the "set-timezone" subcommand is used, # offer the output of
`timedatectl list-timezones` as completions. # Each line of
output is used as a separate candidate, # and anything after a
tab is taken as the description.
# It's often useful to transform command output with `string` into that form.
complete -c timedatectl -n "__fish_seen_subcommand_from set-timezone" -a "(timedatectl list
# Completion candidates can also be described via `-d`,
# which is useful if the description is constant.
# Try to keep these short, because that means the user gets to see more at once.
complete -c timedatectl -n "not __fish_seen_subcommand_from $commands" -a "set-local-rtc" -
d
# We can also limit options to certain subcommands by using conditions.
complete -c timedatectl -n "__fish_seen_subcommand_from set-local-rtc" -l adjust-system-
cloc
# These are simple options that can be used everywhere.
complete -c timedatectl -s h -l help -d 'Print a short help text and exit'
complete -c timedatectl -l version -d 'Print a short version string and exit'
complete -c timedatectl -l no-pager -d 'Do not pipe output into a pager'
```

For examples of how to write your own complex completions, study the completions in `/usr/share/fish/completions`. (The exact path depends on your chosen installation prefix and may be slightly different)

Useful functions for writing completions¶

`fish` ships with several functions that are very useful when writing command specific completions. Most of these functions name begins with the string '`__fish_`'. Such functions are internal to `fish` and their name and interface may change in future fish versions. Still, some of them may be very useful when writing completions. A few of these functions are described here. Be aware that they may be removed or changed in future versions of fish.

Functions beginning with the string `__fish_print_` print a newline separated list of strings. For example, `__fish_print_filesystems` prints a list of all known file systems. Functions beginning with `__fish_complete_` print out a newline separated list of completions with descriptions. The description is separated from the completion by a tab character.

- `__fish_complete_directories STRING DESCRIPTION` performs path completion on STRING, allowing only directories, and giving them the description DESCRIPTION.
- `__fish_complete_path STRING DESCRIPTION` performs path completion on STRING, giving them the description DESCRIPTION.
- `__fish_complete_groups` prints a list of all user groups with the groups members as description.
- `__fish_complete_pids` prints a list of all processes IDs with the command name as description.
- `__fish_complete_suffix SUFFIX` performs file completion allowing only files ending in SUFFIX, with an optional description.
- `__fish_complete_users` prints a list of all users with their full name as description.
- `__fish_print_filesystems` prints a list of all known file systems. Currently, this is a static list, and not dependent on what file systems the host operating system actually understands.
- `__fish_print_hostnames` prints a list of all known hostnames. This functions searches the fstab for nfs servers, ssh for known hosts and checks the `/etc/hosts` file.
- `__fish_print_interfaces` prints a list of all known network interfaces.

`__fish_print_packages` prints a list of all installed packages. This function currently handles Debian, rpm and Gentoo packages.

Where to put completions¶

Completions can be defined on the commandline or in a configuration file, but they can also be automatically loaded. Fish automatically searches through any directories in the list variable `$fish_complete_path`, and any completions defined are automatically loaded when needed. A completion file must have a filename consisting of the name of the command to complete and the suffix '`.fish`'.

By default, Fish searches the following for completions, using the first available file that it finds:

- A directory for end-users to keep their own completions, usually `~/.config/fish/completions` (controlled by the `XDG_CONFIG_HOME` environment variable); A
- directory for systems administrators to install completions for all users on the system, usually `/etc/fish/completions`;
- Directories for third-party software vendors to ship their own completions for their software. Fish searches the directories in the `XDG_DATA_DIRS` environment variable for a `fish/vendor_completions.d` directory; if this variable is not defined, the default is usually to search `/usr/share/fish/vendor_completions.d` and `/usr/local/share/fish/vendor_completions.d`;
- The completions shipped with fish, usually installed in `/usr/share/fish/completions`; and
- Completions automatically generated from the operating system's manuals, usually stored in `~/.local/share/fish/generated_completions`.

These paths are controlled by parameters set at build, install, or run time, and may vary from the defaults listed above.

This wide search may be confusing. If you are unsure, your completions probably belong in `~/.config/fish/completions`.

If you have written new completions for a common Unix command, please consider sharing your work by submitting it via the instructions in Further help and development.

If you are developing another program and would like to ship completions with your program, install them to the "vendor" completions directory. As this path may vary from system to system, the pkgconfig framework should be used to discover this path with the output of `pkg-config --variable completionsdir fish`. Your installation system should support a custom path to override the pkgconfig path, as other distributors may need to alter it easily.

Command line editor¶

The `fish` editor features copy and paste, a searchable history and many editor functions that can be bound to special keyboard shortcuts.

Similar to bash, fish has Emacs and Vi editing modes. The default editing mode is Emacs. You can switch to Vi mode with `fish_vi_key_bindings` and switch back with `fish_default_key_bindings`. You can also make your own key bindings by creating a function and setting $fish_key_bindings to its name. For example:

```
function hybrid_bindings --description "Vi-style bindings that inherit emacs-style
bindings     for mode in default insert visual         fish_default_key_bindings -M $mode
end
    fish_vi_key_bindings --no-erase end
set -g fish_key_bindings
hybrid_bindings
```

Shared bindings¶

Some bindings are shared between emacs- and vi-mode because they aren't text editing bindings or because what Vi/Vim does for a particular key doesn't make sense for a shell.

- `Tab` completes the current token. `Shift, Tab` completes the current token and starts the pager's search mode.
- `Alt+←`, `Left` and `Alt+→`, `Right` move the cursor one word left or right (to the next space or punctuation mark), or moves forward/backward in the directory history if the command line is empty. If the cursor is already at the end of the line, and an autosuggestion is available, `Alt+→`, `Right` (or `Alt+F`) accepts the first word in the suggestion.
- `Shift,←`, `Left` and `Shift,→`, `Right` move the cursor one word left or right, without stopping on punctuation.
- ↑ (Up) and ↓ (Down) (or `Control+P` and `Control+N` for emacs aficionados) search the command history for the previous/next command containing the string that was specified on the commandline before the search was started. If the commandline was empty when the search started, all commands match. See the history section for more information on history searching.
- `Alt+↑`, `Up` and `Alt+↓`, `Down` search the command history for the previous/next token containing the token under the cursor before the search was started. If the commandline was not on a token when the search started, all tokens match. See the history section for more information on history searching.
- `Control+C` cancels the entire line.
- `Control+D` delete one character to the right of the cursor. If the command line is empty, `Control+D` will exit fish.
- `Control+U` moves contents from the beginning of line to the cursor to the killring.
- `Control+L` clears and repaints the screen.
- `Control+W` moves the previous path component (everything up to the previous "/", ":" or "@") to the killring.
- `Control+X` copies the current buffer to the system's clipboard, `Control+V` inserts the clipboard contents.
- `Alt+d` moves the next word to the killring.
- `Alt+h` (or `F1`) shows the manual page for the current command, if one exists. `Alt+l` lists the contents of the current directory, unless the cursor is over a directory argument, in which case the contents of that directory will be listed.
- `Alt+p` adds the string '`| less;`' to the end of the job under the cursor. The result is that the output of the command will be paged.
- `Alt+w` prints a short description of the command under the cursor.
- `Alt+e` edit the current command line in an external editor. The editor is chosen from the first available of the `$VISUAL` or `$EDITOR` variables.
- `Alt+v` Same as `Alt+e`.

`Alt+s` Prepends *sudo* to the current commandline.

Emacs mode commands¶

- `Home` or `Control+A` moves the cursor to the beginning of the line.
- `End` or `Control+E` moves to the end of line. If the cursor is already at the end of the line, and an autosuggestion is available, `End` or `Control+E` accepts the autosuggestion.

← (Left) (or `Control+B`) and → (Right) (or `Control+F`) move the cursor left or right by one character. If the cursor is already at the end of the line, and an autosuggestion is available, the → (Right) key and the `Control+F` combination accept the suggestion. `Delete` and `Backspace` removes one character forwards
- or backwards respectively.
- `Control+K` moves contents from the cursor to the end of line to the killring.
- `Alt+c` capitalizes the current word.
- `Alt+u` makes the current word uppercase. `Control+t`
- transposes the last two characters `Alt+t` transposes
- the last two words

You can change these key bindings using the bind builtin.

Vi mode commands¶

Vi mode allows for the use of Vi-like commands at the prompt. Initially, insert mode is active. `Escape` enters command mode. The commands available in command, insert and visual mode are described below. Vi mode shares some bindings with Emacs mode.

It is also possible to add all emacs-mode bindings to vi-mode by using something like:

```
function fish_user_key_bindings
    # Execute this once per mode that emacs bindings should be used in
fish_default_key_bindings -M insert
    # Without an argument, fish_vi_key_bindings will default to
    # resetting all bindings.
    # The argument specifies the initial mode (insert, "default" or visual).
fish_vi_key_bindings insert end
```

When in vi-mode, the fish_mode_prompt function will display a mode indicator to the left of the prompt. To disable this feature, override it with an empty function. To display the mode elsewhere (like in your right prompt), use the output of the `fish_default_mode_prompt` function.

When a binding switches the mode, it will repaint the mode-prompt if it exists, and the rest of the prompt only if it doesn't. So if you want a mode-indicator in your `fish_prompt`, you need to erase `fish_mode_prompt` e.g. by adding an empty file at *~/.config/fish/functions/fish_mode_prompt.fish*. (Bindings that change the mode are supposed to call the *repaint-mode* bind function, see bind)

The `fish_vi_cursor` function will be used to change the cursor's shape depending on the mode in supported terminals. The following snippet can be used to manually configure cursors after enabling vi-mode:

```
# Emulates vim's cursor shape behavior
# Set the normal and visual mode cursors to a block
set fish_cursor_default block
# Set the insert mode cursor to a line
set fish_cursor_insert line
# Set the replace mode cursor to an underscore
set fish_cursor_replace_one underscore
# The following variable can be used to configure cursor shape in
```

-

```
# visual mode, but due to fish_cursor_default, is redundant here
set fish_cursor_visual block
```

Additionally, `blink` can be added after each of the cursor shape parameters to set a blinking cursor in the specified shape.

Command mode¶

Command mode is also known as normal mode.

- `h` moves the cursor left.`l` moves the cursor
- right.
- `i` enters insert mode at the current cursor
- position. `v` enters visual mode at the current
- cursor position. `a` enters insert mode after
- the current cursor position. `Shift,A` enters
- insert mode at the end of the line.
- `0` (zero) moves the cursor to beginning of line (remaining in command mode).
- `dd` deletes the current line and moves it to the killring.
- `Shift,D` deletes text after the current cursor position and moves it to the killring.
- `p` pastes text from the killring. `u`
- search history backwards.
- `[` and `]` search the command history for the previous/next token containing the token under the cursor before the search was started. See the history section for more information on history searching. `Backspace` moves the cursor left.

Insert mode¶

- `Escape` enters command mode.
- `Backspace` removes one character to the left.

Visual mode¶

- ← (Left) and → (Right) extend the selection backward/forward by one character.
- `b` and `w` extend the selection backward/forward by one word. `d` and `x` move
- the selection to the killring and enter command mode. `Escape` and `Control+C`
- enter command mode.

Copy and paste (Kill Ring)¶

`fish` uses an Emacs-style kill ring for copy and paste functionality. For example, use `Control+K` (*kill-line*) to cut from the current cursor position to the end of the line. The string that is cut (a.k.a. killed in emacs-ese) is inserted into a list of kills, called the kill ring. To paste the latest value from the kill ring (emacs calls this "yanking") use `Control+Y` (the *yank* input function). After pasting, use `Alt+Y` (*yank-pop*) to rotate to the previous kill.

Copy and paste from outside are also supported, both via the `Control+X` / `Control+V` bindings (the *fish_clipboard_copy* and *fish_clipboard_paste* functions [4]) and via the terminal's paste function, for which fish enables "Bracketed Paste Mode", so it can tell a paste from manually entered text. In addition, when pasting inside single quotes, pasted single quotes and backslashes are automatically escaped so that the result can be used as a single token simply by closing the quote after.

[4]These rely on external tools. Currently xsel, xclip, wl-copy/wl-paste and pbcopy/pbpaste are supported.

Searchable history¶

After a command has been entered, it is inserted at the end of a history list. Any duplicate history items are automatically removed. By pressing the up and down keys, the user can search forwards and backwards in the history. If the current command line is not empty

when starting a history search, only the commands containing the string entered into the command line are shown.

By pressing `Alt+↑`, `Up` and `Alt+↓`, `Down`, a history search is also performed, but instead of searching for a complete commandline, each commandline is broken into separate elements just like it would be before execution, and the history is searched for an element matching that under the cursor.

History searches can be aborted by pressing the escape key.

Prefixing the commandline with a space will prevent the entire line from being stored in the history.

The command history is stored in the file `~/.local/share/fish/fish_history` (or `$XDG_DATA_HOME/fish/fish_history` if that variable is set) by default. However, you can set the `fish_history` environment variable to change the name of the history session (resulting in a `<session>_history` file); both before starting the shell and while the shell is running.

See the history command for other manipulations.

Examples:

To search for previous entries containing the word 'make', type `make` in the console and press the up key.

If the commandline reads `cd m`, place the cursor over the `m` character and press `Alt+↑`, `Up` to search for previously typed words containing 'm'.

Multiline editing¶

The fish commandline editor can be used to work on commands that are several lines long. There are three ways to make a command span more than a single line:

- Pressing the `Enter` key while a block of commands is unclosed, such as when one or more block commands such as `for`, `begin` or `if` do not have a corresponding `end` command.
  Pressing `Alt+Enter` instead of pressing the `Enter` key.
- By inserting a backslash (`\`) character before pressing the `Enter` key, escaping the newline.

The fish commandline editor works exactly the same in single line mode and in multiline mode. To move between lines use the left and right arrow keys and other such keyboard shortcuts.

Running multiple programs¶

Normally when `fish` starts a program, this program will be put in the foreground, meaning it will take control of the terminal and `fish` will be stopped until the program finishes. Sometimes this is not desirable. For example, you may wish to start an application with a graphical user interface from the terminal, and then be able to continue using the shell. In such cases, there are several ways in which the user can change fish's behavior.

- By ending a command with the `&` (ampersand) symbol, the user tells `fish` to put the specified command into the background. A background process will be run simultaneous with `fish`. `fish` will retain control of the terminal, so the program will not be able to read from the keyboard.

- By pressing `Control+Z`, the user stops a currently running foreground program and returns control to `fish`. Some programs do not support this feature, or remap it to another key. GNU Emacs uses `Control+X z` to stop running.
- By using the bg and fg builtin commands, the user can send any currently running job into the foreground or background.

Note that functions cannot be started in the background. Functions that are stopped and then restarted in the background using the `bg` command will not execute correctly.

Initialization files¶

On startup, Fish evaluates a number of configuration files, which can be used to control the behavior of the shell. The location of these is controlled by a number of environment variables, and their default or usual location is given below.

Configuration files are evaluated in the following order:

- Configuration shipped with fish, which should not be edited, in `$__fish_data_dir/config.fish` (usually `/usr/share/fish/config.fish`). Configuration
- snippets in files ending in `.fish`, in the directories:
    - `$__fish_config_dir/conf.d` (by default, `~/.config/fish/conf.d/`)
    - `$__fish_sysconf_dir/conf.d` (by default, `/etc/fish/conf.d/`)
    - Directories for third-party software vendors to ship their own configuration snippets for their software. Fish searches the directories in the `XDG_DATA_DIRS` environment variable for a `fish/vendor_conf.d` directory; if this variable is not defined, the default is usually to search `/usr/share/fish/vendor_conf.d` and `/usr/local/share/fish/vendor_conf.d`

  If there are multiple files with the same name in these directories, only the first will be executed. They are executed in order of their filename, sorted (like globs) in a natural order (i.e. "01" sorts before "2").

- System-wide configuration files, where administrators can include initialization that should be run for all users on the system - similar to `/etc/profile` for POSIX-style shells in `$__fish_sysconf_dir` (usually `/etc/fish/config.fish`).

- User initialization, usually in *~/.config/fish/config.fish* (controlled by the `XDG_CONFIG_HOME` environment variable, and accessible as `$__fish_config_dir`).

These paths are controlled by parameters set at build, install, or run time, and may vary from the defaults listed above.

This wide search may be confusing. If you are unsure where to put your own customisations, use *~/.config/fish/config.fish*.

Note that ~/.config/fish/config.fish is sourced _after_ the snippets. This is so users can copy snippets and override some of their behavior.

These files are all executed on the startup of every shell. If you want to run a command only on starting an interactive shell, use the exit status of the command *status --is-interactive* to determine if the shell is

interactive. If you want to run a command only when using a login shell, use *status --is-login* instead. This will speed up the starting of non-interactive or nonlogin shells.

If you are developing another program, you may wish to install configuration which is run for all users of the fish shell on a system. This is discouraged; if not carefully written, they may have side-effects or slow the startup of the shell. Additionally, users of other shells will not benefit from the Fish-specific configuration. However, if they are absolutely required, you may install them to the "vendor" configuration directory. As this path may vary from system to system, the `pkgconfig` framework should be used to discover this path with the output of *pkg-config --variable confdir fish*.

Examples:

If you want to add the directory `~/linux/bin` to your PATH variable when using a login shell, add the following to your *~/.config/fish/config.fish* file:

```
if status --is-login
    set -x PATH $PATH ~/linux/bin
end
```

If you want to run a set of commands when `fish` exits, use an event handler that is triggered by the exit of the shell:

```
function on_exit --on-event fish_exit
echo fish is now exiting end
```

Future feature flags¶

Feature flags are how fish stages changes that might break scripts. Breaking changes are introduced as opt-in, in a few releases they become opt-out, and eventually the old behavior is removed.

You can see the current list of features via `status features`:

```
> status features
stderr-nocaret  on     3.0      ^ no longer redirects stderr
qmark-noglob    off    3.0      ? no longer globs regex-easyesc
off    3.1      string replace -r needs fewer \\'s
```

There are two breaking changes in fish 3.0: caret `^` no longer redirects stderr, and question mark `?` is no longer a glob.

There is one breaking change in fish 3.1: `string replace -r` does a superfluous round of escaping for the replacement, so escaping backslashes would look like `string replace -ra '([ab])' '\\\\\\\$1' a`. This flag removes that if turned on, so `'\\\$1'` is enough.

These changes are off by default. They can be enabled on a per session basis:

```
> fish --features qmark-noglob,stderr-nocaret
```

or opted into globally for a user:

```
> set -U fish_features stderr-nocaret qmark-noglob
```

Features will only be set on startup, so this variable will only take effect if it is universal or exported.

You can also use the version as a group, so `3.0` is equivalent to "stderr-nocaret" and "qmarknoglob".

Prefixing a feature with `no-` turns it off instead. .. _other:

Other features¶

Syntax highlighting¶

`fish` interprets the command line as it is typed and uses syntax highlighting to provide feedback to the user. The most important feedback is the detection of potential errors. By default, errors are marked red.

Detected errors include:

- Non existing commands.
- Reading from or appending to a non existing file.
- Incorrect use of output redirects
- Mismatched parenthesis

When the cursor is over a parenthesis or a quote, `fish` also highlights its matching quote or parenthesis.

To customize the syntax highlighting, you can set the environment variables listed in the Variables for changing highlighting colors section.

Programmable title¶

When using most virtual terminals, it is possible to set the message displayed in the titlebar of the terminal window. This can be done automatically in fish by defining the `fish_title` function. The `fish_title` function is executed before and after a new command is executed or put into the foreground and the output is used as a titlebar message. The *status currentcommand* builtin will always return the name of the job to be put into the foreground (or
'fish' if control is returning to the shell) when the `fish_prompt` function is called. The first argument to fish_title will contain the most recently executed foreground command as a string, starting with fish

2.2. Examples: The default `fish` title is:

```
function fish_title
    echo (status current-command) ' '
pwd end
```

To show the last command in the title:

```
function fish_title
echo $argv[1] end
```

Programmable prompt¶

When fish waits for input, it will display a prompt by evaluating the `fish_prompt` and *fish_right_prompt* functions. The output of the former is displayed on the left and the latter's output on the right side of the terminal. The output of `fish_mode_prompt` will be prepended on the left, though the default function only does this when in vi-mode.

Configurable greeting¶

If a function named `fish_greeting` exists, it will be run when entering interactive mode.
Otherwise, if an environment variable named `fish_greeting` exists, it will be printed.

Private mode¶

fish supports launching in private mode via `fish --private` (or `fish -P` for short). In private mode, old history is not available and any interactive commands you execute will not be appended to the global history file, making it useful both for avoiding inadvertently leaking personal information (e.g. for screencasts) and when dealing with sensitive information to prevent it being persisted to disk. You can query the global variable *fish_private_mode*`(`if set -q fish_private_mode ...`) if you would like to respect the user's wish for privacy and alter the behavior of your own fish scripts.

Event handlers¶

When defining a new function in fish, it is possible to make it into an event handler, i.e. a function that is automatically run when a specific event takes place. Events that can trigger a handler currently are:

- When a signal is delivered
- When a process or job exits
- When the value of a variable is updated
- When the prompt is about to be shown
- When a command lookup fails Example:

To specify a signal handler for the WINCH signal, write:

```
function my_signal_handler --on-signal WINCH
echo Got WINCH signal! end
```

Please note that event handlers only become active when a function is loaded, which means you might need to otherwise source or execute a function instead of relying on autoloading. One approach is to put it into your initialization file.

For more information on how to define new event handlers, see the documentation for the function command.

Debugging fish scripts¶

Fish includes a built in debugging facility. The debugger allows you to stop execution of a script at an arbitrary point. When this happens you are presented with an interactive prompt. At this prompt you can execute any fish command (there are no debug commands as such). For example, you can check or change the value of any variables using `printf` and `set`. As another example, you can run `status print-stack-trace` to see how this breakpoint was reached. To resume normal execution of the script, simply type `exit` or [ctrl-D].

To start a debug session simply run the builtin command `breakpoint` at the point in a function or script where you wish to gain control. Also, the default action of the TRAP signal is to call this builtin. So a running script can be debugged by sending it the TRAP signal with the `kill` command. Once in the debugger, it is easy to insert new breakpoints by using the funced function to edit the definition of a function.

Further help and development¶

If you have a question not answered by this documentation, there are several avenues for help:

- The official mailing list at fish-users@lists.sourceforge.net
- The Internet Relay Chat channel, #fish on `irc.oftc.net` The
- project GitHub page

If you have an improvement for fish, you can submit it via the mailing list or the GitHub page.

Other help pages¶

- Introduction
- Commands
- Design
- Tutorial
- Frequently asked questions
- License

Table of Contents

Table of Contents

Navigation