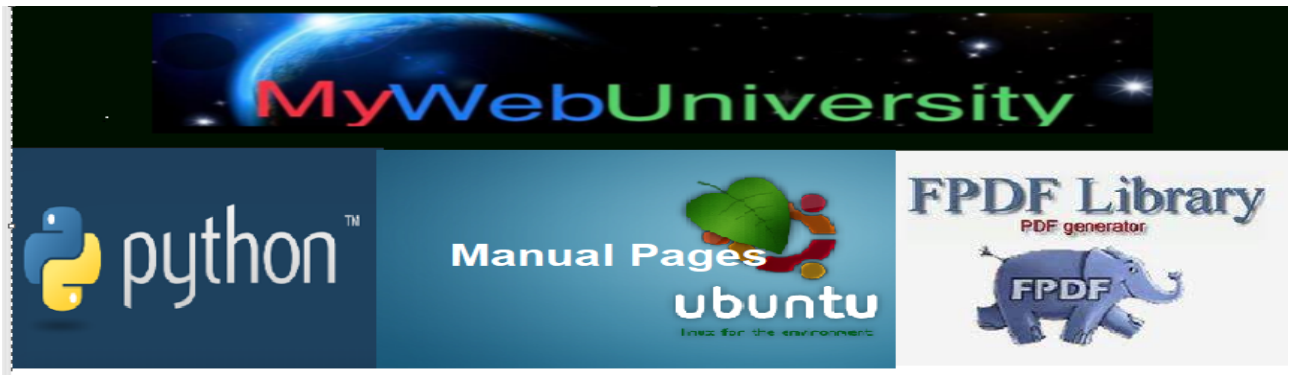

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'fish-interactive.1'

$ man fish-interactive.1

FISH-INTERACTIVE(1) fish-shell FISH-INTERACTIVE(1)

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.

Fish is used by giving commands in the fish language, see The Fish Language <> for information on that.

HELP

Fish has an extensive help system. Use the help <> command to obtain help on a specific sub? ject or command. For instance, writing help syntax displays the syntax section <#syntax> of this documentation.

Fish also has man pages for its commands, and translates the help pages to man pages. 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 ob? tain help on the fg <> builtin, either type fg -h or help fg.

The main page can be viewed via help index (or just help) or man fish-doc. The tutorial can be viewed with help tutorial or man fish-tutorial.

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 (?) or ctrl-f. To accept the first suggested word, press alt-right (?) or alt-f. If the autosugges? tion is not what you want, 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 direc?
tory hierarchies.

If you don't like autosuggestions, you can disable them by setting $fish_autosuggestion_en?
abled to 0:

```
set -g fish_autosuggestion_enabled 0
```

TAB COMPLETION

Tab completion is a time saving feature of any modern shell. When you type tab, fish tries to guess the rest of the word under the cursor. If it finds exactly one possibility, it in?
serts it. If it finds more, it inserts the longest unambiguous part and then opens a menu (the "pager") that you can navigate to find what you're looking for.

The pager can be navigated with the arrow keys, pageup / pagedown, tab or shift-tab. Press?
ing ctrl-s (the pager-toggle-search binding - / in vi mode) opens up a search menu that you can use to filter the list.

Fish provides some general purpose completions, like for commands, variable names, usernames or files.

It also provides a large number of program specific scripted completions. Most of these com?
pletions are simple options like the -l option for ls, but a lot are more advanced. For ex?
ample:

? man and whatis show the installed manual pages as completions.

? make uses targets in the Makefile in the current directory as completions.

? mount uses mount points specified in fstab as completions.

? apt, rpm and yum show installed or installable packages

You can also write your own completions or install some you got from someone else. For that, see Writing your own completions <>.

Completion scripts are loaded on demand, like functions are <#syntax-function-autoloading>. The difference is the $fish_complete_path list <#variables-lists> is used instead of $fish_function_path. Typically you can drop new completions in ~/.config/fish/comple?
tions/name-of-command.fish and fish will find them automatically.

SYNTAX HIGHLIGHTING

Fish interprets the command line as it is typed and uses syntax highlighting to provide feedback. The most important feedback is the detection of potential errors. By default, er?
rors 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

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

Fish also provides pre-made color themes you can pick with fish_config <>. Running just fish_config opens a browser interface, or you can use fish_config theme in the terminal.

For example, to disable nearly all coloring:

```
fish_config theme choose None
```

Or, to see all themes, right in your terminal:

```
fish_config theme show
```

Syntax highlighting variables

The colors used by fish for syntax highlighting can be configured by changing the values of various variables. The value of these variables can be one of the colors accepted by the set_color <> command. Options accepted by set_color like --background=, --bold, --dim, --italics, --reverse, --underline and --underline-color= are also accepted.

Example: to make errors highlighted and red, use:

```
set fish_color_error red --bold
```

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

Variable	Meaning
fish_color_normal	default color
fish_color_command	commands like echo
fish_color_keyword	keywords like if - this falls back on the command color if unset

?   ? quoted text like "abc"   ?

?   fish_color_quote   ?   ?

??

?   ? IO redirections like >/dev/null   ?

?   fish_color_redirection   ?   ?

??

?   ? process separators like ; and &   ?

?   fish_color_end   ?   ?

??

?   ? syntax errors   ?

?   fish_color_error   ?   ?

??

?   ? ordinary command parameters   ?

?   fish_color_param   ?   ?

??

?   ? parameters that are filenames (if ?

?   fish_color_valid_path   ? the file exists)   ?

??

?   ? options starting with "-", up to ?

?   fish_color_option   ? the first "--" parameter   ?

??

?   ? comments like '# important'   ?

?   fish_color_comment   ?   ?

??

?   ? selected text in vi visual mode   ?

?   fish_color_selection   ?   ?

??

?   ? parameter   expansion   operators ?

?   fish_color_operator   ? like * and ~   ?

??

?   ? character escapes like \n and ?

?   fish_color_escape   ? \x70   ?

??

? ? autosuggestions (the proposed ?

? fish_color_autosuggestion ? rest of a command) ?

??

? ? the current working directory in ?

? fish_color_cwd ? the default prompt ?

??

? ? the current working directory in ?

? fish_color_cwd_root ? the default prompt for the root ?

? ? user ?

??

? ? the username in the default ?

? fish_color_user ? prompt ?

??

? ? the hostname in the default ?

? fish_color_host ? prompt ?

??

? ? the hostname in the default ?

? fish_color_host_remote ? prompt for remote sessions (like ?

? ? ssh) ?

??

? ? the last command's nonzero exit ?

? fish_color_status ? code in the default prompt ?

??

? ? the '^C' indicator on a canceled ?

? fish_color_cancel ? command ?

??

? ? history search matches and se? ?

? fish_color_search_match ? lected pager items (background ?

? ? only) ?

??

? ? the current position in the his? ?

? fish_color_history_current ? tory for commands like dirh and ?

? ? cdh ?

??

If a variable isn't set or is empty, fish usually tries $fish_color_normal, except for:

? $fish_color_keyword, where it tries $fish_color_command first.

? $fish_color_option, where it tries $fish_color_param first.

? For $fish_color_valid_path, if that doesn't have a color, but only modifiers, it adds those to the color that would otherwise be used, like $fish_color_param. But if valid paths have a color, it uses that and adds in modifiers from the other color.

Pager color variables

fish will sometimes present a list of choices in a table, called the pager.

Example: to set the background of each pager row, use:

```
set fish_pager_color_background --background=white
```

To have black text on alternating white and gray backgrounds:

```
set fish_pager_color_prefix black
set fish_pager_color_completion black
set fish_pager_color_description black
set fish_pager_color_background --background=white
set fish_pager_color_secondary_background --background=brwhite
```

Variables affecting the pager colors:

??
? Variable ? Meaning ?
??
? ? the progress bar at the bottom ?
? fish_pager_color_progress ? left corner ?
??
? ? the background color of a line ?
? fish_pager_color_back? ? ?
? ground ? ?
??
? ? the prefix string, i.e. the ?
? fish_pager_color_prefix ? string that is to be completed ?
??
? ? the completion itself, i.e. the ?
? fish_pager_color_comple? ? proposed rest of the string ?

? tion ? ?

??

? ? the completion description ?

? fish_pager_color_descrip? ? ?

? tion ? ?

??

? ? background of the selected com? ?

? fish_pager_color_se? ? pletion ?

? lected_background ? ?

??

? ? prefix of the selected completion ?

? fish_pager_color_se? ? ?

? lected_prefix ? ?

??

? ? suffix of the selected completion ?

? fish_pager_color_se? ? ?

? lected_completion ? ?

??

? ? description of the selected com? ?

? fish_pager_color_se? ? pletion ?

? lected_description ? ?

??

? ? background of every second unse? ?

? fish_pager_color_sec? ? lected completion ?

? ondary_background ? ?

??

? ? prefix of every second unselected ?

? fish_pager_color_sec? ? completion ?

? ondary_prefix ? ?

??

? ? suffix of every second unselected ?

? fish_pager_color_sec? ? completion ?

? ondary_completion ? ?

??

? ? description of every second unse? ?

? fish_pager_color_sec? ? lected completion ?

? ondary_description ? ?

??

When the secondary or selected variables aren't set or are empty, the normal variables are used, except for $fish_pager_color_selected_background, where the background of $fish_color_search_match is tried first.

ABBREVIATIONS

To avoid needless typing, a frequently-run command like git checkout can be abbreviated to gco using the abbr <> command.

```
abbr -a gco git checkout
```

After entering gco and pressing space or enter, a gco in command position will turn into git checkout in the command line. If you want to use a literal gco sometimes, use ctrl-space [1].

Abbreviations are a lot more powerful than just replacing literal strings. For example you can make going up a number of directories easier with this:

```
function multicd
    echo cd (string repeat -n (math (string length -- $argv[1]) - 1) ../)
end
abbr --add dotdot --regex '^\.\.+$' --function multicd
```

Now, .. transforms to cd ../, while ... turns into cd ../../ and .... expands to cd ../../../.

The advantage over aliases is that you can see the actual command before using it, add to it or change it, and the actual command will be stored in history.

[1] Any binding that executes the expand-abbr or execute bind function <> will expand ab? breviations. By default ctrl-space is bound to just inserting a space.

PROGRAMMABLE PROMPT

When it is fish's turn to ask for input (like after it started or the command ended), it will show a prompt. Often this looks something like:

```
you@hostname ~>
```

This prompt is determined by running 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. For vi mode, the output of fish_mode_prompt <> will be prepended on the left.

If fish_transient_prompt <#envvar-fish_transient_prompt> is set to 1, fish will redraw the prompt with a --final-rendering argument before running a commandline, allowing you to change it before pushing it to the scrollback.

Fish ships with a few prompts which you can see with fish_config <>. If you run just fish_config it will open a web interface [2] where you'll be shown the prompts and can pick which one you want. fish_config prompt show will show you the prompts right in your termi? nal.

For example fish_config prompt choose disco will temporarily select the "disco" prompt. If you like it and decide to keep it, run fish_config prompt save.

You can also change these functions yourself by running funced fish_prompt and funcsave fish_prompt once you are happy with the result (or fish_right_prompt if you want to change that).

[2] The web interface runs purely locally on your computer and requires python to be in? stalled.

CONFIGURABLE GREETING

When it is started interactively, fish tries to run the fish_greeting <> function. The de? fault fish_greeting prints a simple message. You can change its text by changing the $fish_greeting variable, for instance using a universal variable <#variables-universal>:

```
set -U fish_greeting
```

or you can set it globally <#variables-scope> in config.fish <#configuration>:

```
set -g fish_greeting 'Hey, stranger!'
```

or you can script it by changing the function:

```
function fish_greeting
    random choice "Hello!" "Hi" "G'day" "Howdy"
end
```

save this in config.fish or a function file <#syntax-function-autoloading>. You can also use funced <> and funcsave <> to edit it easily.

PROGRAMMABLE TITLE

Most terminals allow setting the text displayed in the titlebar of the terminal window. Fish does this by running the fish_title <> function. It is executed before and after a command and the output is used as a titlebar message.

The status current-command <> 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_title <> function is called. The first argument will contain the most recently executed foreground command as a string.

The default title shows the hostname if connected via ssh, the currently running command (unless it is fish) and the current working directory. All of this is shortened to not make the tab too wide.

Examples:

To show the last command and working directory in the title:

```
function fish_title
    # `prompt_pwd` shortens the title. This helps prevent tabs from becoming very wide.
    echo $argv[1] (prompt_pwd)
    pwd
end
```

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.

Like bash and other shells, fish includes two sets of keyboard shortcuts (or key bindings): one inspired by the Emacs text editor, and one by the vi text editor. The default editing mode is Emacs. You can switch to vi mode by running 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 the fish_key_bindings variable to its name. For example:

```
function fish_hybrid_key_bindings --description \
"Vi-style bindings that inherit emacs-style bindings in all modes"
    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 fish_hybrid_key_bindings
```

While the key bindings included with fish include many of the shortcuts popular from the respective text editors, they are not a complete implementation. They include a shortcut to open the current command line in your preferred editor (alt-e by default) if you need the

full power of your editor.

Shared bindings

Some bindings are common across Emacs and vi mode, because they aren't text editing bind? ings, 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. tab is the same as ctrl-i.

? left (?) and right (?) 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, right (?) accepts the autosuggestion.

? enter executes the current commandline or inserts a newline if it's not complete yet (e.g. a) or end is missing).

? alt-enter inserts a newline at the cursor position. This is useful to add a line to a com? mandline that's already complete.

? alt-left (?) and alt-right (?) move the cursor left or right by one argument (or one word on macOS). If the command line is empty, they move forward/backward in the directory his? tory. If the cursor is already at the end of the line, and an autosuggestion is avail? able, alt-right (?) (or alt-f) accepts the first argument (or word on macOS) in the sug? gestion.

? ctrl-left (?) and ctrl-right (?) move the cursor left or right by one word. These accept one word of the autosuggestion - the part they'd move over.

? shift-left (?) and shift-right (?) move the cursor one word left or right, without stop? ping on punctuation. These accept one big word of the autosuggestion.

? up (?) and down (?) (or ctrl-p and ctrl-n for emacs aficionados) search the command his? tory for the previous/next command containing the string that was specified on the comman? dline 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 con? taining 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.

? ctrl-c interrupts/kills whatever is running (SIGINT).

? ctrl-d deletes one character to the right of the cursor. If the command line is empty, ctrl-d will exit fish.

? ctrl-u removes contents from the beginning of line to the cursor (moving it to the kill? ring).

? ctrl-l pushes any text above the prompt to the terminal's scrollback, then clears and re? paints the screen.

? ctrl-w removes the previous path component (everything up to the previous "/", ":" or "@") (moving it to the Copy and paste (Kill Ring)).

? ctrl-x copies the current buffer to the system's clipboard, ctrl-v inserts the clipboard contents. (see fish_clipboard_copy <> and fish_clipboard_paste <>)

? alt-d moves the next word to the Copy and paste (Kill Ring).

? ctrl-delete moves the next word (or next argument on macOS) to the Copy and paste (Kill Ring).

? alt-d lists the directory history if the command line is empty.

? alt-delete moves the next argument (or word on macOS) to the Copy and paste (Kill Ring).

? shift-delete removes the current history item or autosuggestion from the command history.

? 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-o opens the file at the cursor in a pager. If the cursor is in command position and the command is a script, it will instead open that script in your editor. The editor is chosen from the first available of the $VISUAL or $EDITOR variables.

? 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. If you set the PAGER variable, its value is used instead of less.

? alt-w prints a short description of the command under the cursor.

? alt-e edits 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. If the commandline is empty, prepend sudo to the last commandline. If sudo is not installed, various similar commands are tried: doas, please, and run0.

? ctrl-space Inserts a space without expanding an abbreviation. For vi mode, this only ap? plies to insert-mode.

To enable emacs mode, use fish_default_key_bindings <>. This is also the default.

? home or ctrl-a moves the cursor to the beginning of the line.

? end or ctrl-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 ctrl-e accepts the autosuggestion.

? ctrl-b, ctrl-f move the cursor one character left or right or accept the autosuggestion just like the left (?) and right (?) shared bindings (which are available as well).

? alt-b, alt-f move the cursor one word left or right, or accept one word of the autosugges? tion. If the command line is empty, moves forward/backward in the directory history in? stead.

? ctrl-n, ctrl-p move the cursor up/down or through history, like the up and down arrow shared bindings.

? delete or backspace or ctrl-h removes one character forwards or backwards respectively.

? ctrl-backspace removes one word backwards and alt-backspace removes one argument back? wards. On macOS, it's the other way round.

? alt-< moves to the beginning of the commandline, alt-> moves to the end.

? ctrl-k deletes from the cursor to the end of line (moving it to the Copy and paste (Kill Ring)).

? escape and ctrl-g cancel the current operation. Immediately after an unambiguous comple? tion this undoes it.

? alt-c capitalizes the current word.

? alt-u makes the current word uppercase.

? ctrl-t transposes the last two characters.

? alt-t transposes the last two words.

? ctrl-z, ctrl-_ (ctrl-/ on some terminals) undo the most recent edit of the line.

? alt-/ or ctrl-shift-z reverts the most recent undo.

? ctrl-r opens the history in a pager. This will show history entries matching the search, a few at a time. Pressing ctrl-r again will search older entries, pressing ctrl-s (that oth? erwise toggles pager search) will go to newer entries. The search bar will always be se? lected.

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 ac? tive. escape enters command mode. The commands available in command, insert and visual mode

are described below. Vi mode shares some bindings with Emacs mode.

To enable vi mode, use fish_vi_key_bindings <>. 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
    # Then execute the vi-bindings so they take precedence when there's a conflict.
    # Without --no-erase fish_vi_key_bindings will default to
    # resetting all bindings.
    # The argument specifies the initial mode (insert, "default" or visual).
    fish_vi_key_bindings --no-erase 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 cursors to an underscore
set fish_cursor_replace_one underscore
set fish_cursor_replace underscore
# Set the external cursor to a line. The external cursor appears when a command is started.
```

```
# The cursor shape takes the value of fish_cursor_default when fish_cursor_external is not specified.
set fish_cursor_external line
# 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.

Fish knows the shapes "block", "line" and "underscore", other values will be ignored.

If the cursor shape does not appear to be changing after setting the above variables, it's likely your terminal emulator does not support the capabilities necessary to do this.

Command mode

Command mode is also known as normal mode.

? h moves the cursor left.

? l moves the cursor right.

? k and j 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. In multi-line commands, they move the cursor up and down respectively.

? i enters insert mode at the current cursor position.

? I enters insert mode at the beginning of the line.

? v enters visual mode at the current cursor position.

? a enters insert mode after the current cursor position.

? A enters insert mode at the end of the line.

? o inserts a new line under the current one and enters insert mode

? O (capital-"o") inserts a new line above the current one and enters insert mode

? 0 (zero) moves the cursor to beginning of line (remaining in command mode).

? d,d deletes the current line and moves it to the Copy and paste (Kill Ring).

? D deletes text after the current cursor position and moves it to the Copy and paste (Kill Ring).

? p pastes text from the Copy and paste (Kill Ring).

? u undoes the most recent edit of the command line.

? ctrl-r redoes the most recent edit.

? [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.

? / opens the history in a pager. This will show history entries matching the search, a few at a time. Pressing it again will search older entries, pressing ctrl-s (that otherwise toggles pager search) will go to newer entries. The search bar will always be selected.

? backspace moves the cursor left.

? g,g / G moves the cursor to the beginning/end of the commandline, respectively.

? ~ toggles the case (upper/lower) of the character and moves to the next character.

? g,u lowercases to the end of the word.

? g,U uppercases to the end of the word.

? :,q exits fish.

Insert mode

? escape enters command mode.

? backspace removes one character to the left.

? ctrl-n accepts the autosuggestion.

Visual mode

? left (``?`) and right`(``?`) extend the selection backward/forward by one character.

? h moves the cursor left.

? l moves the cursor right.

? k moves the cursor up.

? j moves the cursor down.

? b and w extend the selection backward/forward by one word.

? d and x move the selection to the Copy and paste (Kill Ring) and enter command mode.

? escape and ctrl-c enter command mode.

? c and s remove the selection and switch to insert mode.

? X moves the entire line to the Copy and paste (Kill Ring), and enters command mode.

? y copies the selection to the Copy and paste (Kill Ring), and enters command mode.

? ~ toggles the case (upper/lower) on the selection, and enters command mode.

? g,u lowercases the selection, and enters command mode.

? g,U uppercases the selection, and enters command mode.

? ",*,y copies the selection to the clipboard, and enters command mode.

Custom bindings

In addition to the standard bindings listed here, you can also define your own with bind <>:

```
# Prints ``^C`` and a new prompt
bind ctrl-c cancel-commandline
```

Put bind statements into config.fish <#configuration> or a function called fish_user_key_bindings.

If you change your mind on a binding and want to go back to fish's default, you can erase it again:

```
bind --erase ctrl-c
```

Fish remembers its preset bindings and so it will take effect again. This saves you from having to remember what it was before and add it again yourself.

If you use vi bindings, note that bind will by default bind keys in command mode. To bind something in insert mode:

```
bind --mode insert ctrl-c 'commandline -r ""'
```

Key sequences

To find out the name of a key, you can use fish_key_reader <>.

```
> fish_key_reader # Press Alt + right-arrow
Press a key:
bind alt-right 'do something'
```

Note that the historical way the terminal encodes keys and sends them to the application (fish, in this case) makes a lot of combinations indistinguishable or unbindable. In the usual encoding, ctrl-i is the same as the tab key, and shift cannot be detected when ctrl is also pressed.

There are more powerful encoding schemes, and fish tries to tell the terminal to turn them on, but there are still many terminals that do not support them. When fish_key_reader prints the same sequence for two different keys, then that is because your terminal sends the same sequence for them, and there isn't anything fish can do about it. It is our hope that these schemes will become more widespread, making input more flexible.

In the historical scheme, escape is the same thing as alt in a terminal. To distinguish be? tween pressing escape and then another key, and pressing alt and that key (or an escape se? quence the key sends), fish waits for a certain time after seeing an escape character. This is configurable via the fish_escape_delay_ms <#envvar-fish_escape_delay_ms> variable.

If you want to be able to press escape and then a character and have it count as alt+that character, set it to a higher value, e.g.:

```
set -g fish_escape_delay_ms 100
```

Similarly, to disambiguate other keypresses where you've bound a subsequence and a longer sequence, fish has fish_sequence_key_delay_ms <#envvar-fish_sequence_key_delay_ms>:

```
# This binds the sequence j,k to switch to normal mode in vi mode.
# If you kept it like that, every time you press "j",
# fish would wait for a "k" or other key to disambiguate
bind -M insert -m default j,k cancel repaint-mode
# After setting this, fish only waits 200ms for the "k",
# or decides to treat the "j" as a separate sequence, inserting it.
set -g fish_sequence_key_delay_ms 200
```

Copy and paste (Kill Ring)

Fish uses an Emacs-style kill ring for copy and paste functionality. For example, use ctrl-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 ctrl-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 ctrl-x / ctrl-v bindings (the fish_clipboard_copy and fish_clipboard_paste functions [3]) and via the terminal's paste function, for which fish enables "Bracketed Paste Mode", so it can tell a paste from manu? ally 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 by closing the quote after. Kill ring entries are stored in fish_killring variable.

The commands begin-selection and end-selection (unbound by default; used for selection in vi visual mode) control text selection together with cursor movement commands that extend the current selection. The variable fish_cursor_selection_mode <# envvar-fish_cursor_selection_mode> can be used to configure if that selection should include the character under the cursor (inclusive) or not (exclusive). The default is exclusive, which works well with any cursor shape. For vi mode, and particularly for the block or un? derscore cursor shapes you may prefer inclusive.

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

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 new? line.

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 short? cuts.

Searchable command history

After a command has been executed, it is remembered in the history list. Any duplicate his? tory items are automatically removed. By pressing the up and down keys, you can search for? wards 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 like it would be before execution, and the history is searched for an element matching that under the cursor.

For more complicated searches, you can press ctrl-r to open a pager that allows you to search the history. It shows a limited number of entries in one page, press ctrl-r [4] again to move to the next page and ctrl-s [5] to move to the previous page. You can change the text to refine your search.

History searches are case-insensitive unless the search string contains an uppercase charac? ter. You can stop a search to edit your search string by pressing escape or pagedown.

Prefixing the commandline with a space will prevent the entire line from being stored in the history. It will still be available for recall until the next command is executed, but will not be stored on disk. This is to allow you to fix misspellings and such.

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'.

[4] Or another binding that triggers the history-pager input function. See bind <> for a list.

[5] Or another binding that triggers the pager-toggle-search input function.

PRIVATE MODE

Fish has a private mode, in which command history will not be written to the history file on disk. To enable it, either set $fish_private_mode to a non-empty value, or launch with fish --private (or fish -P for short).

If you launch fish with -P, it both hides old history and prevents writing history to disk.

This is useful to avoid leaking personal information (e.g. for screencasts) or when dealing with sensitive information.

You can query the variable fish_private_mode (if test -n "$fish_private_mode" ...) if you would like to respect the user's wish for privacy and alter the behavior of your own fish scripts.

NAVIGATING DIRECTORIES

Navigating directories is usually done with the cd <> command, but fish offers some advanced features as well.

The current working directory can be displayed with the pwd <> command, or the $PWD special variable <#variables-special>. Usually your prompt already does this.

Directory history

Fish automatically keeps a trail of the recent visited directories with cd <> by storing this history in the dirprev and dirnext variables.

Several commands are provided to interact with this directory history:

? dirh <> prints the history

? cdh <> displays a prompt to quickly navigate the history

? prevd <> moves backward through the history. It is bound to alt-left (?)

? nextd <> moves forward through the history. It is bound to alt-right (?)

Directory stack

Another set of commands, usually also available in other shells like bash, deal with the di?

rectory stack. Stack handling is not automatic and needs explicit calls of the following commands:

? dirs <> prints the stack

? pushd <> adds a directory on top of the stack and makes it the current working directory

? popd <> removes the directory on top of the stack and changes the current working direc? tory

Author

fish-shell developers

Copyright

fish-shell developers