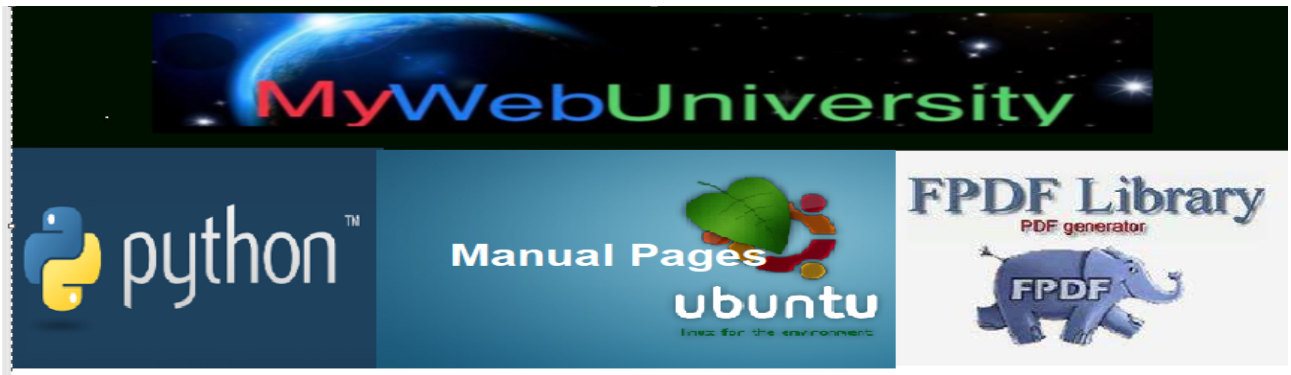

Full credit is given to the above companies including the OS that this PDF file was generated!

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

$ man fish-tutorial.1

FISH-TUTORIAL(1) fish-shell FISH-TUTORIAL(1)

WHY FISH?

Fish is a fully-equipped command line shell (like bash or zsh) that is smart and user-friendly. Fish supports powerful features like syntax highlighting, autosuggestions, and tab completions that just work, with nothing to learn or configure.

If you want to make your command line more productive, more useful, and more fun, without learning a bunch of arcane syntax and configuration options, then fish might be just what you're looking for!

GETTING STARTED

Once installed, just type in fish into your current shell to try it out!

You will be greeted by the standard fish prompt, which means you are all set up and can start using fish:

```
> fish
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
you@hostname ~>
```

This prompt that you see above is the fish default prompt: it shows your username, hostname, and working directory. You can customize it, see how to change your prompt <#prompt>.

From now on, we'll pretend your prompt is just a > to save space.

LEARNING FISH

This tutorial assumes a basic understanding of command line shells and Unix commands, and that you have a working copy of fish.

If you have a strong understanding of other shells, and want to know what fish does differ?

ently, search for the magic phrase unlike other shells, which is used to call out important differences.

Or, if you want a quick overview over the differences to other shells like Bash, see Fish For Bash Users <>.

For the full, detailed description of how to use fish interactively, see Interactive Use <>.

For a comprehensive description of fish's scripting language, see The Fish Language <>.

RUNNING COMMANDS

Fish runs commands like other shells: you type a command, followed by its arguments. Spaces are separators:

```
> echo hello world
hello world
```

This runs the command echo with the arguments hello and world. In this case that's the same as one argument hello world, but in many cases it's not. If you need to pass an argument that includes a space, you can escape <#escapes> with a backslash, or quote <#quotes> it us? ing single or double quotes:

```
> mkdir My\ Files
# Makes a directory called "My Files", with a space in the name
> cp ~/Some\ File 'My Files'
# Copies a file called "Some File" in the home directory to "My Files"
> ls "My Files"
Some File
```

GETTING HELP

Run help to open fish's help in a web browser, and man with the page (like fish-language) to open it in a man page. You can also ask for help with a specific command, for example, help set to open in a web browser, or man set to see it in the terminal.

```
> man set
set - handle shell variables
  Synopsis...
```

To open this section, use help getting-help.

This only works for fish's own documentation for itself and its built-in commands (the "builtins"). For any other commands on your system, they should provide their own documenta? tion, often in the man system. For example man ls should tell you about your computer's ls command.

SYNTAX HIGHLIGHTING

You'll quickly notice that fish performs syntax highlighting as you type. Invalid commands are colored red by default:

```
> /bin/mkd
```

A command may be invalid because it does not exist, or refers to a file that you cannot exe? cute. When the command becomes valid, it is shown in a different color:

```
> /bin/mkdir
```

Valid file paths are underlined as you type them:

```
> cat ~/somefi
```

This tells you that there exists a file that starts with somefi, which is useful feedback as you type.

These colors, and many more, can be changed by running fish_config, or by modifying color variables <#variables-color> directly.

For example, if you want to disable (almost) all coloring:

```
fish_config theme choose None
```

This picks the "none" theme. To see all themes:

```
fish_config theme show
```

Just running fish_config will open up a browser interface that allows you to pick from the available themes.

AUTOSUGGESTIONS

As you type fish will suggest commands to the right of the cursor, in gray. For example:

```
> /bin/hostname
```

It knows about paths and options:

```
> grep --ignore-case
```

And history too. Type a command once, and you can re-summon it by just typing a few letters:

```
> rsync -avze ssh . myname@somelonghost.com:/some/long/path/doo/dee/doo/dee/doo
```

To accept the autosuggestion, hit right (?) or ctrl-f. To accept a single word of the auto? suggestion, alt-right (?). If the autosuggestion is not what you want, just ignore it.

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 COMPLETIONS

A rich set of tab completions work "out of the box".

Press tab and fish will attempt to complete the command, argument, or path:

> /pritab => /private/

If there's more than one possibility, it will list them:

> ~/stuff/stab

~/stuff/script.sh (command) ~/stuff/sources/ (directory)

Hit tab again to cycle through the possibilities. The part in parentheses there (that "com?mand" and "directory") is the completion description. It's just a short hint to explain what kind of argument it is.

fish can also complete many commands, like git branches:

> git merge prtab => git merge prompt_designer

> git checkout btab

builtin_list_io_merge (Branch) builtin_set_color (Branch) busted_events (Tag)

Try hitting tab and see what fish can do!

VARIABLES

Like other shells, a dollar sign followed by a variable name is replaced with the value of that variable:

> echo My home directory is $HOME

My home directory is /home/tutorial

This is known as variable substitution, and it also happens in double quotes, but not single quotes:

> echo "My current directory is $PWD"

My current directory is /home/tutorial

> echo 'My current directory is $PWD'

My current directory is $PWD

Unlike other shells, fish has an ordinary command to set variables: set, which takes a vari?able name, and then its value.

> set name 'Mister Noodle'

> echo $name

Mister Noodle

(Notice the quotes: without them, Mister and Noodle would have been separate arguments, and $name would have been made into a list of two elements.)

Unlike other shells, variables are not further split after substitution:

> mkdir $name

```
> ls
Mister Noodle
```

In bash, this would have created two directories "Mister" and "Noodle". In fish, it created only one: the variable had the value "Mister Noodle", so that is the argument that was passed to mkdir, spaces and all.

You can erase (or "delete") a variable with -e or --erase

```
> set -e MyVariable
> env | grep MyVariable
(no output)
```

For more, see Variable expansion <#expand-variable>.

EXPORTS (SHELL VARIABLES)

Sometimes you need to have a variable available to an external command, often as a setting. For example many programs like git or man read the $PAGER variable to figure out your pre? ferred pager (the program that lets you scroll text). Other variables used like this include $BROWSER, $LANG (to configure your language) and $PATH. You'll note these are written in ALLCAPS, but that's just a convention.

To give a variable to an external command, it needs to be "exported". This is done with a flag to set, either --export or just -x.

```
> set -x MyVariable SomeValue
> env | grep MyVariable
MyVariable=SomeValue
```

It can also be unexported with --unexport or -u.

This works the other way around as well! If fish is started by something else, it inherits that parents exported variables. So if your terminal emulator starts fish, and it exports $LANG set to en_US.UTF-8, fish will receive that setting. And whatever started your terminal emulator also gave it some variables that it will then pass on unless it specifically de? cides not to. This is how fish usually receives the values for things like $LANG, $PATH and $TERM, without you having to specify them again.

Exported variables can be local or global or universal - "exported" is not a scope <# variables-scope>! Usually you'd make them global via set -gx MyVariable SomeValue.

For more, see Exporting variables <#variables-export>.

LISTS

The set command above used quotes to ensure that Mister Noodle was one argument. If it had

been two arguments, then name would have been a list of length 2. In fact, all variables in fish are really lists, that can contain any number of values, or none at all.

Some variables, like $PWD, only have one value. By convention, we talk about that variable's value, but we really mean its first (and only) value.

Other variables, like $PATH, really do have multiple values. During variable expansion, the variable expands to become multiple arguments:

```
> echo $PATH
/usr/bin /bin /usr/sbin /sbin /usr/local/bin
```

Variables whose name ends in "PATH" are automatically split on colons to become lists. They are joined using colons when exported to subcommands. This is for compatibility with other tools, which expect $PATH to use colons. You can also explicitly add this quirk to a vari? able with set --path, or remove it with set --unpath.

Lists cannot contain other lists: there is no recursion. A variable is a list of strings, full stop.

Get the length of a list with count:

```
> count $PATH
5
```

You can append (or prepend) to a list by setting the list to itself, with some additional arguments. Here we append /usr/local/bin to $PATH:

```
> set PATH $PATH /usr/local/bin
```

You can access individual elements with square brackets. Indexing starts at 1 from the be? ginning, and -1 from the end:

```
> echo $PATH
/usr/bin /bin /usr/sbin /sbin /usr/local/bin
> echo $PATH[1]
/usr/bin
> echo $PATH[-1]
/usr/local/bin
```

You can also access ranges of elements, known as "slices":

```
> echo $PATH[1..2]
/usr/bin /bin
> echo $PATH[-1..2]
/usr/local/bin /sbin /usr/sbin /bin
```

You can iterate over a list (or a slice) with a for loop:

```
for val in $PATH
  echo "entry: $val"
end
# Will print:
# entry: /usr/bin/
# entry: /bin
# entry: /usr/sbin
# entry: /sbin
# entry: /usr/local/bin
```

One particular bit is that you can use lists like Brace expansion <#expand-brace>. If you attach another string to a list, it'll combine every element of the list with the string:

```
> set mydirs /usr/bin /bin
> echo $mydirs/fish # this is just like {/usr/bin,/bin}/fish
/usr/bin/fish /bin/fish
```

This also means that, if the list is empty, there will be no argument:

```
> set empty # no argument
> echo $empty/this_is_gone # prints an empty line
```

If you quote the list, it will be used as one string and so you'll get one argument even if it is empty.

For more, see Lists <#variables-lists>. For more on combining lists with strings (or even other lists), see cartesian products <#cartesian-product> and Variable expansion <#expand-variable>.

WILDCARDS

Fish supports the familiar wildcard *. To list all JPEG files:

```
> ls *.jpg
lena.jpg
meena.jpg
santa maria.jpg
```

You can include multiple wildcards:

```
> ls l*.p*
lena.png
lesson.pdf
```

The recursive wildcard ** searches directories recursively:

> ls /var/**.log

/var/log/system.log

/var/run/sntp.log

If that directory traversal is taking a long time, you can ctrl-c out of it.

For more, see Wildcards <#expand-wildcard>.

PIPES AND REDIRECTIONS

You can pipe between commands with the usual vertical bar:

> echo hello world | wc

1 2 12

stdin and stdout can be redirected via the familiar < and >. stderr is redirected with a 2>.

> grep fish < /etc/shells > ~/output.txt 2> ~/errors.txt

To redirect stdout and stderr into one file, you can use &>:

> make &> make_output.txt

For more, see Input and output redirections <#redirects> and Pipes <#pipes>.

COMMAND SUBSTITUTIONS

Command substitutions use the output of one command as an argument to another. Unlike other shells, fish does not use backticks `` for command substitutions. Instead, it uses parenthe?ses with or without a dollar:

> echo In (pwd), running $(uname)

In /home/tutorial, running FreeBSD

A common idiom is to capture the output of a command in a variable:

> set os (uname)

> echo $os

Linux

Command substitutions without a dollar are not expanded within quotes, so the version with a dollar is simpler:

> touch "testing_$(date +%s).txt"

> ls *.txt

testing_1360099791.txt

Unlike other shells, fish does not split command substitutions on any whitespace (like spaces or tabs), only newlines. Usually this is a big help because unix commands operate on a line-by-line basis. Sometimes it can be an issue with commands like pkg-config that print

what is meant to be multiple arguments on a single line. To split it on spaces too, use string split.

```
> printf '%s\n' (pkg-config --libs gio-2.0)
-lgio-2.0 -lgobject-2.0 -lglib-2.0
> printf '%s\n' (pkg-config --libs gio-2.0 | string split -n " ")
-lgio-2.0
-lgobject-2.0
-lglib-2.0
```

If you need a command substitutions output as one argument, without any splits, use quoted command substitution:

```
> echo "first line
second line" > myfile
> set myfile "$(cat myfile)"
> printf '|%s|' $myfile
|first line
second line|
```

For more, see Command substitution <#expand-command-substitution>.

SEPARATING COMMANDS (SEMICOLON)

Like other shells, fish allows multiple commands either on separate lines or the same line.

To write them on the same line, use the semicolon (";"). That means the following two examples are equivalent:

```
echo fish; echo chips
# or
echo fish
echo chips
```

This is useful interactively to enter multiple commands. In a script it's easier to read if the commands are on separate lines.

EXIT STATUS

When a command exits, it returns a status code as a non-negative integer (that's a whole number >= 0).

Unlike other shells, fish stores the exit status of the last command in $status instead of $?.

```
> false
```

```
> echo $status
1
```

This indicates how the command fared - 0 usually means success, while the others signify kinds of failure. For instance fish's set --query returns the number of variables it queried that weren't set - set --query PATH usually returns 0, set --query arglbargl boogagoogoo usually returns 2.

There is also a $pipestatus list variable for the exit statuses [1] of processes in a pipe.

For more, see The status variable <#variables-status>.

[1] or "stati" if you prefer, or "stat?s" if you've time-travelled from ancient Rome or work as a latin teacher

COMBINERS (AND, OR, NOT)

fish supports the familiar && and || to combine commands, and ! to negate them:

```
> ./configure && make && sudo make install
```

Here, make is only executed if ./configure succeeds (returns 0), and sudo make install is only executed if both ./configure and make succeed.

fish also supports and <>, or <>, and not <>. The first two are job modifiers and have lower precedence. Example usage:

```
> cp file1 file1_bak && cp file2 file2_bak; and echo "Backup successful"; or echo "Backup failed"
Backup failed
```

As mentioned in the section on the semicolon, this can also be written in multiple lines, like so:

```
cp file1 file1_bak && cp file2 file2_bak
and echo "Backup successful"
or echo "Backup failed"
```

CONDITIONALS (IF, ELSE, SWITCH)

Use if <> and else <> to conditionally execute code, based on the exit status of a command.

```
if grep fish /etc/shells
    echo Found fish
else if grep bash /etc/shells
    echo Found bash
else
    echo Got nothing
end
```

To compare strings or numbers or check file properties (whether a file exists or is write?

able and such), use test <>, like

```
  if test "$fish" = "flounder"
      echo FLOUNDER
  end
  # or
  if test "$number" -gt 5
      echo $number is greater than five
  else
      echo $number is five or less
  end
  # or
  # This test is true if the path /etc/hosts exists
  # - it could be a file or directory or symlink (or possibly something else).
  if test -e /etc/hosts
      echo We most likely have a hosts file
  else
      echo We do not have a hosts file
  end
```

Combiners can also be used to make more complex conditions, like

```
  if command -sq fish; and grep fish /etc/shells
      echo fish is installed and configured
  end
```

For even more complex conditions, use begin <> and end <> to group parts of them.

There is also a switch <> command:

```
  switch (uname)
  case Linux
      echo Hi Tux!
  case Darwin
      echo Hi Hexley!
  case FreeBSD NetBSD DragonFly
      echo Hi Beastie!
  case '*'
```

```
    echo Hi, stranger!
  end
```

As you see, case <> does not fall through, and can accept multiple arguments or (quoted) wildcards.

For more, see Conditions <#syntax-conditional>.

FUNCTIONS

A fish function is a list of commands, which may optionally take arguments. Unlike other shells, arguments are not passed in "numbered variables" like $1, but instead in a single list $argv. To create a function, use the function <> builtin:

```
  function say_hello
    echo Hello $argv
  end
  say_hello
  # prints: Hello
  say_hello everybody!
  # prints: Hello everybody!
```

Unlike other shells, fish does not have aliases or special prompt syntax. Functions take their place. [2]

You can list the names of all functions with the functions <> builtin (note the plural!).

fish starts out with a number of functions:

```
  > functions
```

N_, abbr, alias, bg, cd, cdh, contains_seq, dirh, dirs, disown, down-or-search, edit_command_buffer, export, fg, fish_add_path, fish_breakpoint_prompt, fish_clipboard_copy, fish_clipboard_paste, fish_config, fish_default_key_bindings, fish_default_mode_prompt, fish_git_prompt, fish_hg_prompt, fish_hybrid_key_bindings, fish_indent, fish_is_root_user, fish_job_summary, fish_key_reader, fish_md5, fish_mode_prompt, fish_npm_helper, fish_opt, fish_print_git_action, fish_print_hg_root, fish_prompt, fish_sigtrap_handler, fish_svn_prompt, fish_title, fish_update_completions, fish_vcs_prompt, fish_vi_cursor, fish_vi_key_bindings, funced, funcsave, grep, help, history, hostname, isatty, kill, la, ll, ls, man, nextd, open, popd, prevd, prompt_hostname, prompt_pwd, psub, pushd, realpath, seq, setenv, suspend, trap, type, umask, up-or-search, vared, wait

You can see the source for any function by passing its name to functions:

```
  > functions ls
  function ls --description 'List contents of directory'
    command ls -G $argv
```

```
end
```

For more, see Functions <#syntax-function>.

[2] There is a function called alias <>, but it's just a shortcut to make functions. fish also provides abbreviations <#abbreviations>, through the abbr <> command.

LOOPS

While loops:

```
while true
    echo "Loop forever"
end
# Prints:
# Loop forever
# Loop forever
# Loop forever
# yes, this really will loop forever. Unless you abort it with ctrl-c.
```

For loops can be used to iterate over a list. For example, a list of files:

```
for file in *.txt
    cp $file $file.bak
end
```

Iterating over a list of numbers can be done with seq:

```
for x in (seq 5)
    touch file_$x.txt
end
```

For more, see Loops and blocks <#syntax-loops-and-blocks>.

PROMPT

Unlike other shells, there is no prompt variable like PS1. To display your prompt, fish executes the fish_prompt <> function and uses its output as the prompt. And if it exists, fish also executes the fish_right_prompt <> function and uses its output as the right prompt.

You can define your own prompt from the command line:

```
> function fish_prompt; echo "New Prompt % "; end
New Prompt % _
```

Then, if you are happy with it, you can save it to disk by typing funcsave fish_prompt. This saves the prompt in ~/.config/fish/functions/fish_prompt.fish. (Or, if you want, you can create that file manually from the start.)

Multiple lines are OK. Colors can be set via set_color <> by passing it named ANSI colors, or hex RGB values:

```
function fish_prompt
    set_color purple
    date "+%m/%d/%y"
    set_color FF0000
    echo (pwd) '>' (set_color normal)
end
```

This prompt would look like:

```
02/06/13
/home/tutorial > _
```

See also Writing your own prompt <>.

You can choose among sample prompts by running fish_config for a web UI or fish_config prompt for a simpler version inside your terminal.

$PATH

$PATH is an environment variable containing the directories that fish searches for commands. Unlike other shells, $PATH is a list, not a colon-delimited string.

Fish takes care to set $PATH to a default, but typically it is just inherited from fish's parent process and is set to a value that makes sense for the system - see Exports.

To prepend /usr/local/bin and /usr/sbin to $PATH, you can write:

```
> set PATH /usr/local/bin /usr/sbin $PATH
```

To remove /usr/local/bin from $PATH, you can write:

```
> set PATH (string match -v /usr/local/bin $PATH)
```

For compatibility with other shells and external commands, $PATH is a path variable <# variables-path>, and so will be joined with colons (not spaces) when you quote it:

```
> echo "$PATH"
/usr/local/sbin:/usr/local/bin:/usr/bin
```

and it will be exported like that, and when fish starts it splits the $PATH it receives into a list on colon.

You can do so directly in config.fish, like you might do in other shells with .profile. See this example.

A faster way is to use the fish_add_path <> function, which adds given directories to the path if they aren't already included. It does this by modifying the $fish_user_paths univer?

sal variable, which is automatically prepended to $PATH. For example, to permanently add /usr/local/bin to your $PATH, you could write:

```
> fish_add_path /usr/local/bin
```

The advantage is that you don't have to go mucking around in files: just run this once at the command line, and it will affect the current session and all future instances too. You can also add this line to config.fish, as it only adds the component if necessary.

Or you can modify $fish_user_paths yourself, but you should be careful not to append to it unconditionally in config.fish, or it will grow longer and longer.

STARTUP (WHERE'S .BASHRC?)

Fish starts by executing commands in ~/.config/fish/config.fish. You can create it if it does not exist.

It is possible to directly create functions and variables in config.fish file, using the commands shown above. For example:

```
> cat ~/.config/fish/config.fish
set -x PATH $PATH /sbin/
function ll
    ls -lh $argv
end
```

However, it is more common and efficient to use autoloading functions and universal vari? ables.

If you want to organize your configuration, fish also reads commands in .fish files in ~/.config/fish/conf.d/. See Configuration Files <#configuration> for the details.

AUTOLOADING FUNCTIONS

When fish encounters a command, it attempts to autoload a function for that command, by looking for a file with the name of that command in ~/.config/fish/functions/.

For example, if you wanted to have a function ll, you would add a text file ll.fish to ~/.config/fish/functions:

```
> cat ~/.config/fish/functions/ll.fish
function ll
    ls -lh $argv
end
```

This is the preferred way to define your prompt as well:

```
> cat ~/.config/fish/functions/fish_prompt.fish
```

```
function fish_prompt
    echo (pwd) "> "
end
```

See the documentation for funced <> and funcsave <> for ways to create these files automati? cally, and $fish_function_path <#syntax-function-autoloading> to control their location.

UNIVERSAL VARIABLES

A universal variable is a variable whose value is shared across all instances of fish, now and in the future ? even after a reboot. You can make a variable universal with set -U:

```
> set -U EDITOR vim
```

Now in another shell:

```
> echo $EDITOR
vim
```

You only need to set universal variables once interactively. There is no need to add them to your config files <#configuration>. For more details, see Universal Variables <# variables-universal>.

READY FOR MORE?

If you want to learn more about fish, there is lots of detailed documentation <#intro>, the official gitter channel <https://gitter.im/fish-shell/fish-shell>, an official mailing list <https://lists.sourceforge.net/lists/listinfo/fish-users>, and the github page <https://github.com/fish-shell/fish-shell/>.

Author

fish-shell developers

Copyright