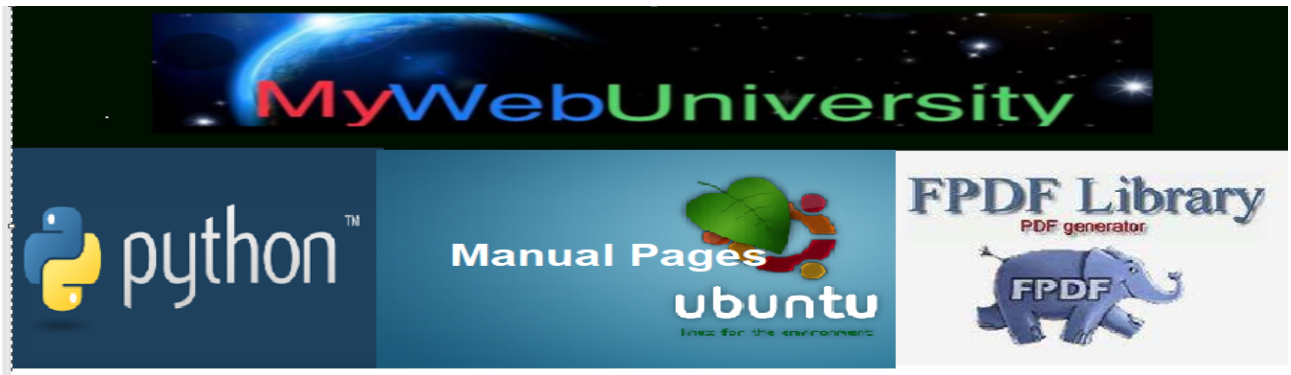

Full credit is given to the above companies including the OS that this PDF file was generated!

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

$ man fish-prompt-tutorial.1

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

Warning:

This document uses formatting to show what a prompt would look like. If you are viewing this in the man page, you probably want to switch to looking at the html version instead. Run help custom-prompt to view it in a web browser.

Fish ships a number of prompts that you can view with the fish_config <> command, and many users have shared their prompts online.

However, you can also write your own, or adjust an existing prompt. This is a good way to get used to fish's scripting language <>.

Unlike other shells, fish's prompt is built by running a function - fish_prompt <>. Or, more specifically, three functions:

? fish_prompt <>, which is the main prompt function

? fish_right_prompt <>, which is shown on the right side of the terminal.

? fish_mode_prompt <>, which is shown if vi mode <#vi-mode> is used.

These functions are run, and whatever they print is displayed as the prompt (minus one trailing newline).

Here, we will just be writing a simple fish_prompt.

OUR FIRST PROMPT

Let's look at a very simple example:

```
function fish_prompt
    echo $PWD '>'
end
```

This prints the current working directory (PWD <#envvar-PWD>) and a > symbol to show where

the prompt ends. The > is quoted <#quotes> because otherwise it would signify a redirection <#redirects>.

Because we've used echo <>, it adds spaces between the two so it ends up looking like (assuming _ is your cursor):

```
/home/tutorial >_
```

FORMATTING

echo adds spaces between its arguments. If you don't want those, you can use string join <> like this:

```
function fish_prompt
    string join '' -- $PWD '>'
end
```

The -- indicates to string that no options can come after it, in case we extend this with something that can start with a -.

There are other ways to remove the space, including echo -s and printf <>.

ADDING COLOR

This prompt is functional, but a bit boring. We could add some color.

Fortunately, fish offers the set_color <> command, so you can do:

```
echo (set_color red)foo
```

set_color can also handle RGB colors like set_color 23b455, and other formatting options including bold and italics.

So, taking our previous prompt and adding some color:

```
function fish_prompt
    string join '' -- (set_color green) $PWD (set_color normal) '>'
end
```

A "normal" color tells the terminal to go back to its normal formatting options.

set_color works by producing an escape sequence, which is a special piece of text that terminals interpret as instructions - for example, to change color. So set_color red produces the same effect as:

```
echo \e\[31m
```

Although you can write your own escape sequences by hand, it's much easier to use set_color.

SHORTENING THE WORKING DIRECTORY

This is fine, but our PWD <#envvar-PWD> can be a bit long, and we are typically only interested in the last few directories. We can shorten this with the prompt_pwd <> helper that

will give us a shortened working directory:

```
function fish_prompt
    string join '' -- (set_color green) (prompt_pwd) (set_color normal) '>'
end
```

prompt_pwd takes options to control how much to shorten. For instance, if we want to display the last two directories, we'd use prompt_pwd --full-length-dirs 2:

```
function fish_prompt
    string join '' -- (set_color green) (prompt_pwd --full-length-dirs 2) (set_color normal) '>'
end
```

With a current directory of "/home/tutorial/Music/Lena Raine/Oneknowing", this would print

```
~/M/Lena Raine/Oneknowing>_
```

STATUS

One important bit of information that every command returns is the status <#variables-status>. This is a whole number from 0 to 255, and usually it is used as an error code - 0 if the command returned successfully, or a number from 1 to 255 if not.

It's useful to display this in your prompt, but showing it when it's 0 seems kind of waste?ful.

First of all, since every command (except for set <>) changes the status, you need to store it for later use as the first thing in your prompt. Use a local variable <#variables-scope> so it will be confined to your prompt function:

```
set -l last_status $status
```

And after that, you can set a string if it is not zero:

```
# Prompt status only if it's not 0
set -l stat
if test $last_status -ne 0
    set stat (set_color red)"[$last_status]"(set_color normal)
end
```

And to print it, we add it to our string join:

```
string join '' -- (set_color green) (prompt_pwd) (set_color normal) $stat '>'
```

If $last_status was 0, $stat is empty, and so it will simply disappear.

So our entire prompt is now:

```
function fish_prompt
    set -l last_status $status
```

```
    # Prompt status only if it's not 0
    set -l stat
    if test $last_status -ne 0
        set stat (set_color red)"[$last_status]"(set_color normal)
    end
    string join '' -- (set_color green) (prompt_pwd) (set_color normal) $stat '>'
end
```

And it looks like:

```
~/M/L/Oneknowing>false
~/M/L/Oneknowing[1]>_
```

after we run false (which returns 1).

TRANSIENT PROMPT

To enable transient prompt functionality, set the fish_transient_prompt <#envvar-fish_transient_prompt> variable to 1:

```
set -g fish_transient_prompt 1
```

With this set, fish re-runs prompt functions with a --final-rendering argument before running a commandline. So you can use it to declutter your old prompts. For example if you want to see only the current directory name when you scroll up:

```
function fish_prompt
    set -l last_status $status
    set -l stat
    set -l pwd
    # Check if it's a transient or final prompt
    if contains -- --final-rendering $argv
        set pwd (path basename $PWD)
    else
        set pwd (prompt_pwd)
        # Prompt status only if it's not 0
        if test $last_status -ne 0
            set stat (set_color red)"[$last_status]"(set_color normal)
        end
    end
    string join '' -- (set_color green) $pwd (set_color normal) $stat '>'
```

```
  end
```

Now running two commands in the same directory could result in this screen:

```
  Oneknowing>false
  ~/M/L/Oneknowing[1]>_
```

SAVE THE PROMPT

Once you are happy with your prompt, you can save it with funcsave fish_prompt (see funcsave - save the definition of a function to the user's autoload directory <>) or write it to ~/.config/fish/functions/fish_prompt.fish yourself.

If you want to edit it again, open that file or use funced fish_prompt (see funced - edit a function interactively <>).

WHERE TO GO FROM HERE?

We have now built a simple but working and usable prompt, but of course more can be done.

?

Fish offers more helper functions:

? prompt_login to describe the user/hostname/container or prompt_hostname to de? scribe just the host

? fish_is_root_user to help with changing the symbol for root.

? fish_vcs_prompt to show version control information (or fish_git_prompt / fish_hg_prompt / fish_svn_prompt to limit it to specific systems)

? You can add a right prompt by changing fish_right_prompt <> or a vi mode prompt by chang? ing fish_mode_prompt <>.

?

Some prompts have interesting or advanced features

? Add the time when the prompt was printed

? Show various integrations like python's venv

? Color the parts differently.

You can look at fish's sample prompts for inspiration. Open up fish_config <>, find one you like and pick it. For example:

```
  fish_config prompt show # <- shows all the sample prompts
  fish_config prompt choose disco # <- this picks the "disco" prompt for this session
  funced fish_prompt # <- opens fish_prompt in your editor, and reloads it once the editor exits
```

Author

fish-shell developers