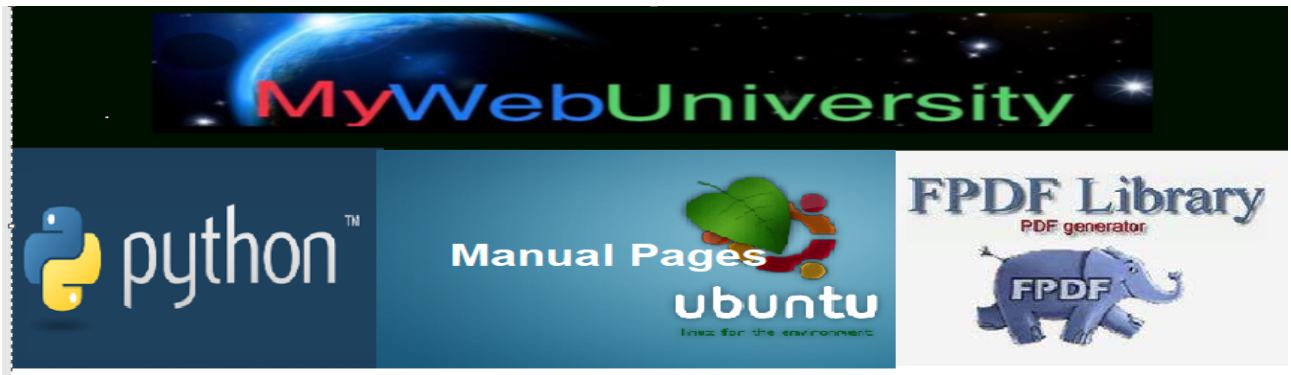

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'fish-for-bash-users.1'

$ man fish-for-bash-users.1

FISH-FOR-BASH-USERS(1) fish-shell FISH-FOR-BASH-USERS(1)

This is to give you a quick overview if you come from bash (or to a lesser extent other shells like zsh or ksh) and want to know how fish differs. Fish is intentionally not POSIX-compatible and as such some of the things you are used to work differently.

Many things are similar - they both fundamentally expand commandlines to execute commands, have pipes, redirections, variables, globs, use command output in various ways. This document is there to quickly show you the differences.

COMMAND SUBSTITUTIONS

Fish spells command substitutions as $(command) or (command), but not `command`.

In addition, it only splits them on newlines instead of $IFS. If you want to split on some? thing else, use string split <>, string split0 <> or string collect <>. If those are used as the last command in a command substitution the splits they create are carried over. So:

```
for i in (find . -print0 | string split0)
```

will correctly handle all possible filenames.

VARIABLES

Fish sets and erases variables with set <> instead of VAR=VAL and a variety of separate builtins like declare and unset and export. set takes options to determine the scope and ex? portedness of a variable:

```
# Define $PAGER *g*lobal and e*x*ported,
# so this is like ``export PAGER=less``
set -gx PAGER less
# Define $alocalvariable only locally,
# like ``local alocalvariable=foo``
```

```
set -l alocalvariable foo
```

or to erase variables:

```
set -e PAGER
```

VAR=VAL statements are available as environment overrides:

```
PAGER=cat git log
```

Fish does not perform word splitting. Once a variable has been set to a value, that value stays as it is, so double-quoting variable expansions isn't the necessity it is in bash. [1]

For instance, here's bash

```
> foo="bar baz"
> printf '"%s"\n' $foo
# will print two lines, because we didn't double-quote
# this is word splitting
"bar"
"baz"
```

And here is fish:

```
> set foo "bar baz"
> printf '"%s"\n' $foo
# foo was set as one element,
# so it will be passed as one element, so this is one line
"bar baz"
```

All variables are "arrays" (we use the term "lists"), and expanding a variable expands to all its elements, with each element as its own argument (like bash's "${var[@]}":

```
> set var "foo bar" banana
> printf %s\n $var
foo bar
banana
```

Specific elements of a list can be selected:

```
echo $list[5..7]
```

The arguments to set are ordinary, so you can also set a variable to the output of a com? mand:

```
# Set lines to all the lines in file, one element per line
set lines (cat file)
```

or a mixture of literal values and output:

```
> set numbers 1 2 3 (seq 5 8) 9
> printf '%s\n' $numbers
1
2
3
5
6
7
8
9
```

A = is unnecessary and unhelpful with set - set foo = bar will set the variable "foo" to two values: "=" and "bar". set foo=bar will print an error.

See Shell variables <#variables> for more.

[1] zsh also does not perform word splitting by default (the SH_WORD_SPLIT option controls this)

WILDCARDS (GLOBS)

Fish only supports the * and ** glob (and the deprecated ? glob) as syntax. If a glob doesn't match it fails the command (like with bash's failglob) unless the command is for, set or count or the glob is used with an environment override (VAR=* command), in which case it expands to nothing (like with bash's nullglob option).

Globbing doesn't happen on expanded variables, so:

```
set foo "*"
echo $foo
```

will not match any files.

There are no options to control globbing so it always behaves like that.

The ** glob will match in subdirectories as well. In other shells this often needs to be turned on with an option, like setopt globstar in bash.

Unlike bash, fish will also follow symlinks, and will sort the results in a natural sort, with included numbers compared as numbers. That means it will sort e.g. music tracks correctly even if they have numbers like 1 instead of 01.

See Wildcards <#expand-wildcard> for more.

QUOTING

Fish has two quoting styles: "" and ''. Variables are expanded in double-quotes, nothing is

expanded in single-quotes.

There is no $'', instead the sequences that would transform are transformed when unquoted:

```
> echo a\nb
a
b
```

See Quotes <#quotes> for more.

STRING MANIPULATION

Fish does not have ${foo%bar}, ${foo#bar} and ${foo/bar/baz}. Instead string manipulation is done by the string <> builtin.

For example, to replace "bar" with "baz":

```
> string replace bar baz "bar luhrmann"
baz luhrmann
```

It can also split strings:

```
> string split "," "foo,bar"
foo
bar
```

Match regular expressions as a replacement for grep:

```
> echo bababa | string match -r 'aba$'
aba
```

Pad strings to a given width, with arbitrary characters:

```
> string pad -c x -w 20 "foo"
xxxxxxxxxxxxxxxxxfoo
```

Make strings lower/uppercase:

```
> string lower Foo
foo
> string upper Foo
FOO
```

repeat strings, trim strings, escape strings or print a string's length or width (in termi? nal cells).

SPECIAL VARIABLES

Some bash variables and their closest fish equivalent:

? $*, $@, $1 and so on: $argv

? $?: $status

? $$: $fish_pid

? $#: No variable, instead use count $argv

? $!: $last_pid

? $0: status filename

? $-: Mostly status is-interactive and status is-login

PROCESS SUBSTITUTION

Instead of <(command) fish uses (command | psub). There is no equivalent to >(command).

Note that both of these are bashisms, and most things can easily be expressed without. E.g.

instead of:

```
source (command | psub)
```

Use:

```
command | source
```

as fish's source <> can read from stdin.

HEREDOCS

Fish does not have <<EOF "heredocs". Instead of

```
cat <<EOF
some string
some more string
EOF
```

use:

```
printf %s\n "some string" "some more string"
```

or:

```
echo "some string
some more string"
# or if you want the quotes on separate lines:
echo "\
some string
some more string\
"
```

Quotes are followed across newlines.

What "heredocs" do is:

1. Read/interpret the string, with special rules, up to the terminator. [2]

2. Write the resulting string to a temporary file.

3. Start the command the heredoc is attached to with that file as stdin.

This means it is essentially the same as just reading from a pipe, so:

```
echo "foo" | cat
```

is mostly the same as

```
cat <<EOF
foo
EOF
```

Like with heredocs, the command has to be prepared to read from stdin. Sometimes this re?
quires special options to be used, often giving a filename of - turns it on.

For example:

```
echo "xterm
rxvt-unicode" | pacman --remove -
# is the same as (the `-` makes pacman read arguments from stdin)
pacman --remove xterm rxvt-unicode
```

and could be written in other shells as

```
# This "-" is still necessary - the heredoc is *also* passed over stdin!
pacman --remove - << EOF
xterm
rxvt-unicode
EOF
```

So heredocs really are minor syntactical sugar that introduces a lot of special rules, which is why fish doesn't have them. Pipes are a core concept, and are simpler and compose nicer.

[2] For example, the "EOF" is just a convention, the terminator can be an arbitrary string, something like "THISISTHEEND" also works. And using <<- trims leading tab characters (but not other whitespace), so you can indent the lines, but only with tabs. Substitu? tions (variables, commands) are done on the heredoc by default, but not if the termina? tor is quoted: cat << "EOF".

TEST (TEST, [, [[)

Fish has a POSIX-compatible test or [builtin. There is no [[and test does not accept == as a synonym for =. It can compare floating point numbers, however.

set -q can be used to determine if a variable exists or has a certain number of elements (set -q foo[2]).

ARITHMETIC EXPANSION

Fish does not have $((i+1)) arithmetic expansion, computation is handled by math <>:

```
math $i + 1
```

Unlike bash's arithmetic, it can handle floating point numbers:

```
> math 5 / 2
2.5
```

And also has some functions, like for trigonometry:

```
> math cos 2 x pi
1
```

You can pass arguments to math separately like above or in quotes. Because fish uses () parentheses for command substitutions, quoting is needed if you want to use them in your ex? pression:

```
> math '(5 + 2) * 4'
```

Both * and x are valid ways to spell multiplication, but * needs to be quoted because it looks like a glob.

PROMPTS

Fish does not use the $PS1, $PS2 and so on variables. Instead the prompt is the output of the fish_prompt <> function, plus the fish_mode_prompt <> function if vi mode <#vi-mode> is enabled. The output of the fish_right_prompt <> function is used for the right-sided prompt.

As an example, here's a relatively simple bash prompt:

```
# <$HOSTNAME> <$PWD in blue> <Prompt Sign in Yellow> <Rest in default light white>
PS1='\h\[\e[1;34m\]\w\[\e[m\] \[\e[1;32m\]\$\[\e[m\] '
```

and a rough fish equivalent:

```
function fish_prompt
    set -l prompt_symbol '$'
    fish_is_root_user; and set prompt_symbol '#'
    echo -s (prompt_hostname) \
    (set_color blue) (prompt_pwd) \
    (set_color yellow) $prompt_symbol (set_color normal)
end
```

This shows a few differences:

? Fish provides set_color <> to color text. It can use the 16 named colors and also RGB se? quences (so you could also use set_color 5555FF)

? Instead of introducing specific escapes like \h for the hostname, the prompt is a func?

tion. To achieve the effect of \h, fish provides helper functions like prompt_hostname <>, which prints a shortened version of the hostname.

? Fish offers other helper functions for adding things to the prompt, like fish_vcs_prompt <> for adding a display for common version control systems (git, mercurial, svn), and prompt_pwd <> for showing a shortened $PWD (the user's home directory becomes ~ and any path component is shortened).

The default prompt is reasonably full-featured and its code can be read via type fish_prompt.

Fish does not have $PS2 for continuation lines, instead it leaves the lines indented to show that the commandline isn't complete yet.

BLOCKS AND LOOPS

Fish's blocking constructs look a little different. They all start with a word, end in end and don't have a second starting word:

```
for i in 1 2 3; do
  echo $i
done
# becomes
for i in 1 2 3
  echo $i
end
while true; do
  echo Weeee
done
# becomes
while true
  echo Weeeeeee
end
{
  echo Hello
}
# becomes
begin
  echo Hello
```

```
end
if true; then
  echo Yes I am true
else
  echo "How is true not true?"
fi
# becomes
if true
  echo Yes I am true
else
  echo "How is true not true?"
end
foo() {
  echo foo
}
# becomes
function foo
   echo foo
end
# (bash allows the word "function",
#  but this is an extension)
```

Fish does not have an until. Use while not or while !.

SUBSHELLS

Bash has a feature called "subshells", where it will start another shell process for certain things. That shell will then be independent and e.g. any changes it makes to variables won't be visible in the main shell.

This includes things like:

```
# A list of commands in `()` parentheses
(foo; bar) | baz
# Both sides of a pipe
foo | while read -r bar; do
    # This will not be visible outside of the loop.
    VAR=VAL
```

```
    # This background process will not be, either
    baz &
  done
```

Fish does not currently have subshells. You will have to find a different solution. The iso? lation can usually be achieved by scoping variables (with set -l), but if you really do need to run your code in a new shell environment you can use fish -c 'your code here' to do so explicitly.

() subshells are often confused with {} grouping, which does not use a subshell. When you just need to group, you can use begin; end in fish:

```
  (foo; bar) | baz
  # when it should really have been:
  { foo; bar; } | baz
  # becomes
  begin; foo; bar; end | baz
```

The pipe will be run in the same process, so while read loops can set variables outside:

```
  foo | while read bar
    set -g VAR VAL
    baz &
  end
  echo $VAR # will print VAL
  jobs # will show "baz"
```

Subshells are also frequently confused with command substitutions, which bash writes as `command` or $(command) and fish writes as $(command) or (command). Bash also uses subshells to implement them.

BUILTINS AND OTHER COMMANDS

By now it has become apparent that fish puts much more of a focus on its builtins and exter? nal commands rather than its syntax. So here are some helpful builtins and their rough equivalent in bash:

? string <> - this replaces most of the string transformation (${i%foo} et al) and can also be used instead of grep and sed and such.

? math <> - this replaces $((i + 1)) arithmetic and can also do floats and some simple func? tions (sine and friends).

? argparse <> - this can handle a script's option parsing, for which bash would probably use

getopt (zsh provides zparseopts).

? count <> can be used to count things and therefore replaces $# and can be used instead of wc.

? status <> provides information about the shell status, e.g. if it's interactive or what the current linenumber is. This replaces $- and $BASH_LINENO and other variables.

? seq(1) can be used as a replacement for {1..10} range expansion. If your OS doesn't ship a seq fish includes a replacement function.

OTHER FACILITIES

Bash has set -x or set -o xtrace to print all commands that are being executed. In fish, this would be enabled by setting fish_trace <#envvar-fish_trace>.

Or, if your intention is to profile how long each line of a script takes, you can use fish --profile - see the page for the fish command <>.

Author

fish-shell developers

Copyright