Tutorial

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. - to change this prompt see how to change your prompt - to switch to fish permanently see switch your default shell to fish.

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

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*.

You can include a literal space in an argument with a backslash, or by using single or double quotes:

```
> mkdir My\ Files
> cp ~/Some\ File 'My Files'
> ls "My Files"
Some File
```

Commands can be chained with semicolons.

Getting Help

`fish` has excellent help and man pages. Run `help` to open help in a web browser, and `man` 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...
```

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 execute. When the command becomes valid, it is shown in a different color:

```
> /bin/mkdir
```

`fish` will underline valid file paths 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 variables directly.

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
```

Especially powerful is the recursive wildcard ** which 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 `Control+C` out of it.

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 need to first redirect stdout, and then stderr into stdout:

```
> make > make_output.txt 2>&1
```

Autosuggestions

`fish` suggests commands as you type, and shows the suggestion to the right of the cursor, in gray. For example

> /bin/hostname

It knows about paths and options

> *grep* --*i*gnore-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 arrow) or `Control+F`. To accept a single word of the autosuggestion, `Alt+→` (right arrow). If the autosuggestion is not what you want, just ignore it.

Tab Completions

`fish` comes with a rich set of tab completions, that work "out of the box."

Press `Tab`, and `fish` will attempt to complete the command, argument, or path

> /pri `Tab` => /private/

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

> ~/stuff/s `Tab` ~/stuff/script.sh (Executable, 4.8kB) ~/stuff/sources/ (Directory)

Hit tab again to cycle through the possibilities.

`fish` can also complete many commands, like git branches:

```
> git merge pr :kbd:`Tab` => git merge prompt_designer
> git checkout b :kbd:`Tab`
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 performs variable substitution:

```
> echo My home directory is $HOME
My home directory is /home/tutorial
```

Variable substitution 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 no dedicated *VARIABLE=VALUE* syntax for setting variables. Instead it has an ordinary command: `set`, which takes a variable 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. Other shells use the term "arrays", rather than lists.

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

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

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 *$PAGER* variable to figure out your preferred pager (the program that lets you scroll text). Other variables used like this include *$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". Unlike other shells, `fish` does not have an export command. Instead, a variable is exported via an option 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`.

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 variable 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 beginning, 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
entry: /usr/bin/
entry: /bin
entry: /usr/sbin
entry: /sbin
entry: /usr/local/bin
```

Lists adjacent to other lists or strings are expanded as cartesian products unless quoted (see Variable expansion):

```
> set a 1 2 3
> set 1 a b c
> echo $a$1
1a 2a 3a 1b 2b 3b 1c 2c 3c
> echo $a" banana"
1 banana 2 banana 3 banana
> echo "$a banana"
1 2 3 banana
```

This is similar to Brace expansion.

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 parentheses:

```
> 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 are not expanded within quotes. Instead, you can temporarily close the quotes, add the command substitution, and reopen them, all in the same argument:

```
> 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. This 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 " ")
-lgio-2.0
-lgobject-2.0
-lglib-2.0
```

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
```

Exit Status

When a command exits, it returns a status code as a natural number. 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.

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 --queryPATH` 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.

[1] or "stati" if you prefer, or "statūs" 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.txt file1_bak.txt && cp file2.txt file2_bak.txt ; and echo "Backup succ
Backup failed
```

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

```
cp file1.txt file1_bak.txt && cp file2.txt file2_bak.txt
and echo "Backup successful"
```

```
or echo "Backup failed"
```

Conditionals (If, Else, Switch)

Use `if`, `else 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 writeable 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

if test -e /etc/hosts # is true if the path /etc/hosts exists - it could be a file
    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 grep fish /etc/shells; and command -sq fish
    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
```

Note that `case` does not fall through, and can accept multiple arguments or (quoted) wildcards.

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
Hello
> say_hello everybody!
Hello everybody!
```

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

You can list the names of all functions with the `functions` keyword (note the plural!). `fish` starts out with a number of functions:

```
> functions
alias, cd, delete-or-exit, dirh, dirs, down-or-search, eval, export, fish_command_
```

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
```

Loops

While loops:

```
> while true
    echo "Loop forever"
end
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
```

Prompt

Unlike other shells, there is no prompt variable like PS1. To display your prompt, `fish` executes a function with the name `fish_prompt`, and its output is used as the prompt.

You can define your own prompt:

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

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

```
> function fish_prompt
      set_color purple
      date "+%m/%d/%y"
      set_color FF0
      echo (pwd) '>' (set_color normal)
  end
```

will look like

02/06/13
/home/tutorial >

You can choose among some sample prompts by running `fish_config prompt`. `fish` also supports RPROMPT through `fish_right_prompt`.

$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.

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, 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 modify the `$fish_user_paths` universal variable, which is automatically prepended to `$PATH`. For example, to permanently add `/usr/local/bin` to your `$PATH`, you could write:

```
> set -U fish_user_paths /usr/local/bin $fish_user_paths
```

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. (Note: you should NOT add this line to `config.fish`. If you do, the variable will get longer each time you run fish!)

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 variables.

If you want to organize your configuration, fish also reads commands in .fish files in `~/.config/fish/conf.d/`. See initialization 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 automatically, and $fish_function_path 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
```

Switching to fish?

If you wish to use fish (or any other shell) as your default shell, you need to enter your new shell's executable `/usr/local/bin/fish` in two places: - add `/usr/local/bin/fish` to `/etc/shells` - change your default shell with `chsh -s /usr/local/bin/fish`

You can use the following commands for this:

Add the fish shell `/usr/local/bin/fish` to `/etc/shells` with:

```
> echo /usr/local/bin/fish | sudo tee -a /etc/shells
```

Change your default shell to fish with:

```
> chsh -s /usr/local/bin/fish
```

(To change it back to another shell, just substitute `/usr/local/bin/fish` with `/bin/bash`, `/bin/tcsh` or `/bin/zsh` as appropriate in the steps above.)