# Table of Contents

## Overview 5

### Read the Manual 6
- man 6
- GNU info 8
- Documentation from Installed Packages 8
- Ask the Program You’re Trying to Use for Help 8

### Learn How to Find and Install Packages 9
- `sudo apt-get update` 10
- `sudo apt-get upgrade` 10
- `apt-cache` 11
- `sudo apt-get install [package]` 11
- `dpkg` 12

### Use a Terminal Multiplexer 13
- Get you a tmux 13
- Persistent Multi-Terminal Sessions & Easy Multitasking 14
- Keybindings 15

### Use History, Tab Completion, and Line Editing 16
- History 16
- Tab Completion 17
- Line Editing 17

### Use Pattern Matching 18

### Use Wildcards in Bash 20

### Find Text with grep(1) 21
- Further Reading 24

### Find Files with find(1) 24
- Further Reading 25

### Watch Processes 25
- `ps(1)` 25
- `/proc` 26
- `top(1)` and `htop(1)` 27

### Customize your .bashrc 28
- Write Aliases For Commands You Use Frequently 28
- Customize Your Prompt 29
- Keep More History 29

### Write Shell Scripts 30
- A Sample Script 32
- Further Reading 33

### Write Shell Utilities in Other Languages 33
- Python: stars.py 34
• Perl: stars.pl
• Node.js: stars.js

Concluding Remarks
Overview

This guide is a followup to An Illustrated Guide to Shell Magic: Standard I/O & Redirection (https://adafruit.it/Cf7). It's about some of the other ways that Bash and its relatives provide for composable tools and general problem-solving.

It's far from comprehensive. Rather, it's a survey of some techniques and ideas that will let you spend less time typing and more time just doing stuff on a Raspberry Pi or similar machine.

We'll touch on documentation, software installation, persistent terminal sessions, keybindings, pattern matching, process monitoring, aliases, basic scripting, and more.
Unix-like systems have nearly always come with a manual, and Linux is no exception. On a Raspbian machine, there are at least a couple of manuals available at all times.

It's natural to reach for Google as soon as you run into a technical problem, but sometimes the fastest route to a solution is the locally installed manuals. (And sometimes you just don't have an internet connection.)

**man**

Curious about a given topic? If you know the name of a command, you can use `man` (short for "manual"). For example, `man grep` will tell you more than you ever wanted to know about the `grep` command. You can search by keyword with `man -k keyword`. You can also search with `apropos keyword`, which for some reason I have always found easier to remember.

There's an art to reading most man pages. They're not always the friendliest documents, at first glance, but once you have the basics down, they often pack a lot of knowledge into a small space. Here's how the man page for `grep` usually displays, with a few bits highlighted:
### A - GREP(1)
Name of the man page in all caps, followed by the section of the manual in parentheses. Most things you look up will be in section 1, for general programs, but **there are 7 more**.

### B - grep, egrep, fgrep, etc.
Various alternate names under which grep can be invoked, followed by a brief description of the command.

### C - SYNOPSIS
One or more summaries of how the command is invoked. Optional things are usually inside square brackets, alternatives are denoted with a |, and so on.

### D - DESCRIPTION
A longer prose description of the command and its uses.

### E - OPTIONS
A breakdown of available options which change the behavior of the command.
| F - --version | A lot of options have both a short form (-V) and a long form (--version). A lot of the time (but not always!), the long version uses two dashes and the short version uses one. |

You'll often find other info - the authors of a given program, notes about its history or how it differs from other versions of the same utility, known bugs, pointers to related programs, and so forth.

**GNU info**

The GNU project, responsible for the versions of the core utilities most commonly used on Linux systems, has its own documentation system, called `info`. It can be browsed by topic just by invoking `info` or searched with `info -k keyword`. If you already know the name of a node, you can just say `info node`. For example, the documentation on info itself can be reached with `info info`.

**Documentation from Installed Packages**

Debian, the distribution Raspbian is based on, installs a lot of documentation in `/usr/share/doc`. If you look there, you'll find a directory for most installed packages, often containing a brief README and sometimes complete with example configuration files and scripts. These can be a real life-saver.

**Ask the Program You're Trying to Use for Help**

It's often the case that utilities are written with their own built-in help. For example, `grep` (about which we'll be talking more in a minute) has a `--help` option. This isn't always provided, but it's enough of a convention that it's worth trying if you're stumped.
Commands with a lot of options might print quite a bit more help than fits on your screen. There's a generic solution to the problem of text scrolling out of your terminal: Pipe the command to `less`.

Learn How to Find and Install Packages

The Raspbian archives can be a little out of date, but they contain a huge collection of software packages.

A package is something like the installers you might download on other operating systems - it contains executable software, documentation, configuration files, and so on, along with instructions for where these files should go on the filesystem.
Packages also contain information about their dependencies, the other packages that 
ythey depend on in order to function.

**sudo apt-get update**

You'll want to remember a handful of commands. The first is **apt-get**, which updates 
the list of available packages and installs from it. First, let's make sure we're up-to-
date. Because we're updating system-wide files, we'll need to use **sudo apt-get update**. This will take a while, depending on your network connection, and you'll 
probably get a few screens full of output.

```
pi@raspberrypi ~ $ sudo apt-get update
Get:1 http://mirrordirector.raspbian.org wheezy Release.gpg [490 B]
... Fetched 7,040 kB in 39s (180 kB/s)
Reading package lists... Done
```

**sudo apt-get upgrade**

Next, you can choose to upgrade any packages on the system that have new versions 
available. This is usually a good idea on freshly installed systems or machines you 
haven't used in a while. Again, you'll need to use **sudo**, and you may have to wait a 
while. If there are a lot of packages to be downloaded, you'll be prompted whether to 
continue. Hit "y" as long as you're prepared to wait out the download and installation 
time.

```
pi@raspberrypi ~ $ sudo apt-get upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be upgraded:
  dbus dbus-x11 e2fslibs e2fsprogs krb5-locales libcomerr2 libdbus-1-3 li
  libraspberrypi-dev libraspberrypi-doc libraspberrypi0 libss2 libxml2 nt
  python3-rpi.gpio raspberrypi-bootloader sonic-pi sudo unzip wolfram-eng
31 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 350 MB of archives.
After this operation, 6,007 kB disk space will be freed.
Do you want to continue [Y/n]?
Get:2 http://mirrordirector.raspbian.org/raspbian/ wheezy/main e2fslibs a
Get:3 http://mirrordirector.raspbian.org/raspbian/ wheezy/main e2fsprogs
...```
Next is **apt-cache**, which works with the cached list of packages. It's handy for figuring out what's available. Maybe your Pi has a real shortage of talking bovines - let's see if anything can help us with that, using **apt-cache search cowsay**.

```
pi@raspberrypi ~ $ apt-cache search cowsay
```

In order to inspect the package, you can use **apt-cache show cowsay**. In particular, look for anything under the **Description** line:

```
pi@raspberrypi ~ $ apt-cache show cowsay
Package: cowsay
Version: 3.03+dfsg1-4
Installed-Size: 89
Maintainer: Francois Marier <francois@debian.org>
Architecture: all
Depends: perl
Suggests: filters
Size: 21850
    SHA256: db58abec6da06b0114f8798ce77d6ff6ce7e7de8216c86740d1b3cb017
    SHA1: 21de074e7e203d28302eb29ceb2840ab459cb46
    MD5sum: c6346d68171147118b4fa28d949754b1
Description: configurable talking cow
Homepage: http://www.nog.net/~tony/warez/cowsay.shtml
Description-md5: c312f9ae79aed8150f991fcfa3df1a03
Tag: game::toys, implemented-in::perl, interface::commandline, role::program,
    use::entertaining, works-with::text
Section: games
Priority: optional
Filename: pool/main/c/cowsay/cowsay_3.03+dfsg1-4_all.deb
```

**sudo apt-get install** [package]

So cowsay looks like a promising package - let's install with **sudo apt-get install cowsay** and see what happens.
Success!

**dpkg**

Finally, if you’re working with individual Debian packages (`.deb` files) rather than downloading them from an apt repository, for example when installing a custom kernel ([https://adafruit.it/Cf8](https://adafruit.it/Cf8)), you can use `dpkg`.

To see Info on a package:

```
dpkg -I package.deb
```

To install a package:

```
sudo dpkg -i package.deb
```
Use a Terminal Multiplexer

Get you a tmux

I heard about GNU Screen (https://adafruit.it/eAH) for years, but aside from using it as a serial terminal to talk to an Arduino a handful of times, I never used it much, and I didn’t really get the big idea.

Then one day a friend convinced me to try tmux (https://adafruit.it/eAl), which is a lot like screen but more modern and capable in a number of ways. It took a while to sink in, but now I rarely open a shell anywhere that's not inside a tmux session. Why? Well, you should try it out. First, let's install tmux.

```
    sudo apt-get update
    sudo apt-get install tmux
```

And start it up.
Ok, so now you have your regular terminal, but with a little green status bar at the bottom. So what? It kind of seems like you just opened another terminal inside your terminal (yo dawg), but what does this get you?

### Persistent Multi-Terminal Sessions & Easy Multitasking

Here's the big idea: Your network connection can go away - you can even just close a terminal window whenever you feel like it - and the next time you connect to the remote machine, you just type:

```
 tmux attach
```

...and there's your old shell, plugging away, just the way you left it. You can even connect to the same session from a bunch of different machines at once, and see the same thing on all of those screens.

There are other useful features. You can open up multiple shells and cycle between them, and there's a way to search through the scrollbar in a given buffer that comes in handy all the time.
But the main thing is that you can leave lots of programs running on the Pi and reconnect to them whenever you feel like it, and even switch between different computers or networks for accessing them. You could, for example, start things while you have a monitor and keyboard plugged into the Pi, then reattach to them later over SSH. (Or vice versa.)

**Keybindings**

In order to get much real use out of tmux, you’ll want to remember a handful of keybindings.

<table>
<thead>
<tr>
<th>Keybinding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-b</td>
<td>Prefix for most commands. To send a literal Ctrl-b, type it twice in a row.</td>
</tr>
<tr>
<td>Ctrl-b d</td>
<td>Detach from current tmux session. (Leaves tmux running in the background; you can reconnect at any time with <code>tmux attach</code>.)</td>
</tr>
<tr>
<td>Ctrl-b c</td>
<td>Create a new window.</td>
</tr>
<tr>
<td>Ctrl-b n</td>
<td>Go to next window.</td>
</tr>
<tr>
<td>Ctrl-b p</td>
<td>Go to previous window.</td>
</tr>
<tr>
<td>Ctrl-b ?</td>
<td>Show a list of shortcuts (hit q to quit out of this list).</td>
</tr>
<tr>
<td>Ctrl-b w</td>
<td>Display a list of windows which you can navigate with the arrow keys and switch to with Enter.</td>
</tr>
<tr>
<td>Ctrl-b %</td>
<td>Split the current window into two panes.</td>
</tr>
<tr>
<td>Ctrl-b o</td>
<td>Cycle through open panes.</td>
</tr>
</tbody>
</table>
In order to quit tmux, just type `exit` or press Ctrl-d in the shell like you normally would to logout. Once all open windows are closed, tmux itself will exit.

There's lots more. The tmux man page (https://adafruit.it/eAJ) and this cheatsheet (https://adafruit.it/eAK) are both good references.

### Use History, Tab Completion, and Line Editing

**History**

Bash keeps a record of your previous commands - try using the up and down arrow keys (or Ctrl-p / Ctrl-n) to cycle through history.

```bash
pi@raspberrypi ~ $ history
```

If you want to see a bunch of history at once, try the `history` command. This can get long, so try piping it to `tail`:

```bash
pi@raspberrypi ~ $ history | tail
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>help history</td>
<td>93</td>
<td>help history</td>
</tr>
<tr>
<td>94</td>
<td>cowthink what?</td>
<td>95</td>
<td>cowthink 'what?'</td>
</tr>
<tr>
<td>96</td>
<td>history</td>
<td>97</td>
<td>man history</td>
</tr>
<tr>
<td>98</td>
<td>help history</td>
<td>99</td>
<td>history</td>
</tr>
<tr>
<td>100</td>
<td>history</td>
<td>head</td>
<td>101</td>
</tr>
</tbody>
</table>

Suppose you remember part of a command, and want to reuse it?

Try typing Ctrl-r, followed by the part of the command you remember. Bash will search the history for matches. You can then press Enter to execute the command over again, or Esc to edit it.
Tab Completion

Often, you know the name of a command or file, but it would be nice not to have to type the whole thing. Try typing the first few letters of a command and hit Tab. Bash will attempt to find a matching command and fill it in for you. If there's more than one command that matches, you may need to hit Tab again, and you'll be presented with a list of possible commands.

Similarly, typing part of a file name and pressing Tab will often complete the name. If more than one file matches what you've already typed, hit Tab a couple of times and you'll get a list of files.

Line Editing

If you need to fix a typo or revise an earlier command you've retrieved from history, Bash provides a set of editing commands.

- Ctrl-b or left arrow key: Move back one character
<table>
<thead>
<tr>
<th>Ctrl-f or right arrow</th>
<th>Move forward one character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-a or Home</td>
<td>Move to beginning of line</td>
</tr>
<tr>
<td>Ctrl-e or End</td>
<td>Move to end of line</td>
</tr>
<tr>
<td>Ctrl-k</td>
<td>Kill (delete/cut) text from cursor position to end of line</td>
</tr>
<tr>
<td>Ctrl-y</td>
<td>Yank (paste) previously killed text</td>
</tr>
</tbody>
</table>

There's more - check out [the manual on Readline Interaction](https://adafru.it/eAL).

### Use Pattern Matching

The people who originally wrote Unix and a lot of the software that runs on it were fond of pattern matching.

You can see this in shells like Bash, in tools like `sed` (a stream editor used for transforming text) and `grep` (a finder of text), and in many of the programming languages that emerged in the 80s and 90s.

Pattern matching is an idea that covers a lot of specific tools. For our purposes, it usually means finding a pattern of characters in lines of text, inside files, or in the names of files themselves.

This is usually done by writing patterns in a little language which describes the text to be matched. Here's a really simple version of one of those:

```
* → any of anything
? → one of anything
```
We'll say that * (often said aloud as "star") means "0 or more of any character", and ? means "1 of any character". Any other characters in the pattern will be treated as literal characters, which means they represent themselves and we're literally looking for that character. A b will mean that we want to match on a b.

Consider the pattern R*. It'll match any string of characters that starts with an R, like Ralph or Rob.

How about the pattern R*b? Well, we know Ralph fits some of the criteria. It starts with R, followed by some number of characters, but is there a b at the end?

Not so much. How about for Rob?
Characters like these are often called wildcards, because they're like cards that can stand in for any other card in games like poker.

# Use Wildcards in Bash

In the shell, you'll often find yourself working with collections of files. For example, you might have lots of text files, Python scripts, or GIFs.

Right now, I have a collection of many image files in one directory, some of which have `.gif` extensions and some of which have `.png` extensions. The former are animated GIFs used for these guides, and the latter are still screenshots saved as PNGs.

That's a lot of files. I've decided I want to put the GIFs in their own folder, but how to move them without typing all of those names?

Well, Bash supports a number of wildcards for working with files, including the basic `*` and `?` just discussed. Let's try the `mv` command ([https://adafru.it/Cf9](https://adafru.it/Cf9)) with a wildcard:

```bash
pi@raspberrypi ~/adafruit_guides $ ls | wc -l
160
pi@raspberrypi ~/adafruit_guides $ mkdir ../adafruit_guide_gifs
pi@raspberrypi ~/adafruit_guides $ mv *.gif ../adafruit_guide_gifs
```
To break this down,

1. `ls | wc -l` pipes (https://adafruit.it/Cfa) the result of `ls` to `wc`, which is a command for counting words and lines. The `-l` option tells it to only return the number of lines. This just tells us the number of files in the current directory.
2. `mkdir ../adafruit_guide_gifs` makes a new directory one level up from the current working directory (remember that `..` is another way of saying "the parent of this directory").
3. In `mv *.gif ../adafruit_guide_gifs`, the wildcard `*.gif` is expanded by Bash to a list of all the files ending in `.gif` in the current directory, and then the command is executed like normal.
4. We use `wc -l` again to show that we've moved 63 files to `adafruit_guide_gifs`.

Since filename expansion of this kind (also known as globbing) is built into Bash, it works with any command that takes a list of filenames.

---

**Find Text with grep(1)**

Using patterns to specify files in the shell is all well and good - it'll save you mountains of keystrokes - but it's also a bit limited. Often, rather than the names of files on the filesystem, you need to work with text produced by other commands, or with the contents of large text files.

It's also often desirable to write patterns that can match more complicated text.

This is where tools like `grep` come in. Its task is to look at some text (in a file or in the standard output from another command) and print the lines that match a pattern called a regular expression.

Regular expressions (also frequently written regex or regexp) are kind of like wildcards on steroids. Instead of a handful of general-purpose magic characters, they offer a rich vocabulary for expressing what characters and how many of them a pattern should match.

Regen are actually a deep, complicated topic, and they come in dozens of different flavors. It can take years to master their use, and they frequently confuse even very
experienced programmers. Despite all that, it's easy enough to learn the basics, and the basics are enough to get quite a bit done.

As a basic example, let's find some dictionary words. `grep` takes a pattern and (optionally) a file to search.

```bash
pi@raspberrypi ~ $ grep '^magi.*$' /usr/share/dict/words
magic
magic's
magical
magically
magician
magician's
magicians
magisterial
magisterially
magistrate
magistrate's
magistrates
```

The pattern here, `^magi.*$`, is passed to `grep` inside of single quotes so that the shell won't expand characters like `*`. It demonstrates a bunch of the basics:

<table>
<thead>
<tr>
<th>^</th>
<th>start of line</th>
</tr>
</thead>
<tbody>
<tr>
<td>magi</td>
<td>the literal characters &quot;magi&quot;</td>
</tr>
<tr>
<td>.</td>
<td>any character</td>
</tr>
<tr>
<td>*</td>
<td>0 or more of the preceding token, <code>.</code> - what's known as a quantifier</td>
</tr>
<tr>
<td>$</td>
<td>end of line</td>
</tr>
</tbody>
</table>
Notice how * works subtly different in a regular expression from a shell pattern. Rather than standing in for any character on its own, it imposes a quantity on the previous token.

The ^ and $ serve to anchor the pattern to the beginning and end of the line, respectively. These aren't always necessary, but it can be very useful to say that a string occurs at one end or the other.

By default, the grep command doesn't treat most characters as magical. To get the full range of its abilities, you'll want to invoke it as egrep or grep -E for Extended grep. With that in place, here are more of the basics:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[123]</td>
<td>one of 1, 2, or 3</td>
</tr>
<tr>
<td>[a-z]</td>
<td>one of the characters a-z</td>
</tr>
<tr>
<td>\w</td>
<td>a &quot;word&quot; character</td>
</tr>
<tr>
<td>[0-9]</td>
<td>one of 0 through 9</td>
</tr>
<tr>
<td>+</td>
<td>one or more of the previous thing</td>
</tr>
<tr>
<td>?</td>
<td>zero or one of the previous thing</td>
</tr>
</tbody>
</table>
In addition to -E, `grep` takes a bunch of other options. It's worth reading the man page, but the handful that seem to come up most often in shell pipelines are as follows:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(foo){1,3}</code></td>
<td>one to three occurrences of <code>foo</code></td>
</tr>
<tr>
<td>`(foo</td>
<td>bar)`</td>
</tr>
<tr>
<td>`(foobar</td>
<td>baz)*`</td>
</tr>
</tbody>
</table>

In addition to -E, `grep` takes a bunch of other options. It's worth reading the man page, but the handful that seem to come up most often in shell pipelines are as follows:

- `grep -i foo` look for `foo` without paying attention to case - will find `F00`, `foo`, `f00`, etc.
- `grep -v foo` invert the search - find lines that don't contain `foo`
- `grep -c foo` print a count of lines matching `foo`
- `grep -l foo *.txt` list the text files that match `foo`

Further Reading

- Mastering Regular Expressions (https://adafruit.it/eAO), by Jeffrey Friedl, is one of the more comprehensive available texts.
- The GNU Grep manual (https://adafruit.it/eAP)

Find Files with find(1)

`find` is a deceptively powerful tool. Invoked without any options, it'll recursively list all of the files under the current directory and any subdirectories.
...that can be useful, but it's also kind of overwhelming. Most often, you'll want to limit
the results by file name or type. For example, to find files starting with "squirrel"
followed by any extension, you can do something like:

```
pi@raspberrypi ~ $ find . -name 'squirrel.*'
./python_games/squirrel.png
./python_games/squirrel.py
```

The `-name` and `-iname` (the case-insensitive version) options take shell patterns. If
you'd rather use more complex regular expressions, there's a `-regex` option.

`find` supports literally dozens of other tests on files, along with actions like deleting
files (dangerous!), executing commands with files as parameters, and printing file info
according to user-supplied formats.

Further Reading

- [A find Tutorial and Primer](https://adafru.it/eAQ), by Daniel Miessler
- [A Unix/Linux “find” Command Tutorial](https://adafru.it/eAR), by Wayne Pollock
- [The GNU Findutils manual](https://adafru.it/eAS)

Watch Processes

Even a simple Linux system is usually doing a lot of things at once.

```
ps(1)
```

`ps` lists processes. If you run it with no options, it'll show you the processes running
under your current login.
The **PID** column lists a process ID. Every running program on the system will have one of these. If a process is stuck otherwise bothersome, you can stop it with `kill [process ID]`.

If you want to see more detail about all of the things, try `ps aux`:

```
pi@raspberrypi ~ $ ps
  PID TTY      TIME CMD
30054 pts/2    00:00:00 bash
30116 pts/2    00:00:00 ps
```

`ps` takes a gazillion different options in a confusing array of different formats, but memorizing one or two of them is usually good enough to give you a window into what's happening.

If you're looking for a specific program, you might try grepping the output of `ps`. Let's say I wanted to know about tmux sessions running on my Raspberry Pi. One approach might be `ps aux | grep tmux`:

```
pi@raspberrypi /proc/16514 $ ps aux | grep tmux
```

Alternatively, you can use `pgrep`, which is specifically for finding process IDs:

```
pi@raspberrypi /proc/16514 $ pgrep -l tmux
16513 tmux
22350 tmux
30247 tmux
```

(The `-l` is for listing process names along with the PID.)

/proc

Under Linux, `ps` works by reading the files in `/proc`, which is a virtual filesystem full of info about running processes and suchlike things. If you know the id of a running process, you can inspect all sorts of info about it like so:
top(1) and htop(1)

`ps` is useful for checking on the state of a running system or writing a shell pipeline, but often what you want is something more like an interactive monitor. Enter `top`:

```
Top - 20:43:19 up 7 days, 57 min, 3 users, load average: 0.01, 0.02, 0.05
Tasks: 80 total, 1 running, 88 sleeping, 0 stopped, 0 zombie
%CPU(s): 0.2 us, 0.1 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
Mem: 943364 total, 489948 used, 453376 free, 116872 buffers
Swap: 102396 total, 0 used, 102396 free, 203564 cached
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM TIME+ COMMAND
22332 pi      20   0  0     0   0 S   0.0   0.0  0:00.00 htop
```

`top` is old school, and it's probably available on just about every Unix system in use. Unfortunately, it's also kind of cryptic and difficult to read at a glance. My advice is to install `htop` instead - it's prettier, friendlier, and has a ton of features for sorting and filtering the process list.

```
sudo apt-get install htop
```

```
Help 2/7/77:54 [htop]
```

©Adafruit Industries
Customize your .bashrc

Bash is configurable software, and you can tweak a lot of its behavior by editing a file in your home directory called `.bashrc`. To edit, just:

```bash	nano ~/.bashrc
```

On a stock Raspbian installation, you should see something like the following:

```
# ~/.bashrc: executed by bash() for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples
# If not running interactively, don't do anything
#   (e.g. "PS1" ) $ return
# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth
# append to the history file, don't overwrite it
shopt -s histappend
# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000
# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize
# If set, the pattern "*" used in a pathname expansion context will
```

Have a look around this file. Much of it may seem like gibberish, but the basics are fairly simple. Lines starting with `#` are comments. Variables are set like `FOO=bar`.

Once you've made any changes, you can get them to take effect in your current shell with:

```bash
source ~/.bashrc
```

Write Aliases For Commands You Use Frequently

An alias is just a shortcut you can type in place of another command. You define one like so:

```bash
alias moo="cowsay moo"
```

You can try this at the prompt:
The syntax for adding an alias to `.bashrc` so that it’ll be available on every login is identical - have a look through the stock version of the file and you’ll probably find several.

**Customize Your Prompt**

Your prompt - the bit of text, like `pi@raspberrypi ~ $`, that appears before your cursor in Bash - is actually defined by a variable called `PS1`. You can inspect the current value of this variable like so:

```
pi@raspberrypi ~ $ echo $PS1
\[[\e\033[01;32m\]u@\h\ \[\033\[00m\] \[\033\[01;34m\]\w\[\033\[00m\]
```

Not exactly pretty, is it? It makes use of special escapes like `\u` for user, `\h` for host name, `\w` for working directory, etc. Sequences like `\[\033[01;32m\]` are used for special non-printing characters that the terminal recognizes as instructions to use certain text colors.

Serious prompt customization is outside the scope of this guide, but you can learn plenty by messing around with it. Have a look at ezprompt.net (https://adafru.it/eAT) for a quick way to assemble a custom prompt, and see the Bash Reference Manual's Controlling the Prompt (https://adafru.it/eAU) section for a breakdown of escape sequences.

**Keep More History**

The default `.bashrc` sets history length to 1000. If you're anything like me, your memory needs way more help than that. Look for `HISTSIZE` (the amount of history
kept in memory when Bash is running) and **HISTFILESIZE** (the amount of history kept in the history file on disk) and adjust them to your liking. Mine look like so:

```bash
export HISTSIZE=10000
export HISTFILESIZE=120000
```

You can look at the contents of command line history with `nano $HISTFILE`.

---

## Write Shell Scripts

If, metaphorically speaking, a shell pipeline is kind of like an incantation or minor spell that you come up with on the spot to solve some problem, then scripts are a lot like the pages of a spellbook where you keep the incantations that you’ve found really useful.

If you’re familiar with a general-purpose programming language like Python, C, Ruby, or JavaScript, then maybe you’ve noticed by now that Bash has some familiar features. That’s because the shell is a programming language.

Just about any sequence of shell commands you’d type at the prompt can be stashed in a simple text file and executed as a program.

What’s more, Bash has many of the features you’d expect of a programming language:

- **variables**
  ```bash
  export FOO=bar
  echo $FOO
  ```
Between these constructs and the features provided by the standard utilities, shell scripts can be used to solve all sorts of problems. For example:

- Here's a script [here](https://adafru.it/eAV) adapted from Lady Ada's PiTFT installation guide [here](https://adafru.it/sha) which does the tedious work of configuring a touchscreen for the Pi.
- Here's one for [cross-compiling kernels for the Raspberry Pi](https://adafru.it/eAW).
- [raspi-config](https://adafru.it/eAW) runs the first time a Pi boots and covers a wide range of configuration options.
A Sample Script

Let's assemble a very simple filter script that turns all the non-space characters in its input into little stars. First, open stars.sh in Nano:

```
nano stars.sh
```

Next, type or paste the following lines and hit Ctrl-x to save the file.

```
#!/bin/bash
sed 's/\S/★/g' /dev/stdin
```

To break this down:

- `#!/bin/bash` is what's known as a shebang or a hashbang. `#!` are special characters that tell the kernel "this file should be run by feeding it to the following program". (You'll also commonly see this written as `#!/usr/bin/env bash`, which is sometimes considered a more portable way to invoke an interpreter.)
- `sed` is the stream editor. It acts as a filter, taking a stream of text in one side and transforming it by running one or more commands.
- `s/\S/★/g` is a command that says, more or less, "substitute strings matching this regular expression (\S) with this replacement string (★), globally". The `\S` will match everything that's not whitespace in its input.
- `\s` (lowercase) is the metacharacter for space characters; think of the uppercase version as inverting this.
• /dev/stdin is a special file that contains the standard input to our script.

Now make the file executable, and give it a try...

```bash
chmod +x ./stars.sh
```

Hit Enter to submit text, and Ctrl-d on a line by itself to end input. You could also, just as easily, pipe the output of some other command to `./stars.sh`.

Further Reading

- Bash Reference Manual (https://adafru.it/eAU)

Write Shell Utilities in Other Languages

We've established that the shell has lots of the features found in general-purpose programming languages. With that out of the way, I'm going to make the opposite point. Remember that fragment I used as an example of conditional logic in Bash?

```bash
# Is it between midnight and 6am?
# (We'll assume that counts as "after midnight".)
if [ "`date '+%H'`" -lt 6 ]; then
  echo "do not feed the mogwai"
fi
```

This is actually... Well, it's not very much fun dealing with this kind of syntax (https://adafru.it/eAY).
The truth is that, while shell scripting is really useful, it's often not nearly as well suited to writing complicated programs as other languages. If you already know some Python, Perl, Ruby, Node.js, etc., you'll often have a better time using those tools than trying to shoehorn too much into the shell.

As a rough guideline, I usually consider rewriting things in another language once a shell script takes up more than one or two screens in my text editor.

Fortunately, working with other languages than Bash doesn't mean you have to lose the benefits of shell pipelines, standard I/O, and interaction with other utilities.

Following are the equivalents of stars.sh in a handful of popular languages. If you use this basic pattern of reading from standard input and writing to standard out, you can write programs that interact seamlessly with the traditional Unix environment, while benefitting from modern high-level language features and libraries.

**Python: stars.py**

```python
#!/usr/bin/env python
# encoding: utf-8
import sys
import re
write = sys.stdout.write
line = sys.stdin.readline()
while line:
    stars = re.sub('\S', '★', line)
    write(stars)
    line = sys.stdin.readline()
```

**Perl: stars.pl**

```perl
#!/usr/bin/env perl
use warnings;
use strict;

while (my $line = &lt;&gt;) {
    $line =~ s/\S//g;
    print $line . "\n";
}
```
Node.js: stars.js

For this one to run, you'll need Node.js. We've got a [tutorial on installing Node.js on the Pi](https://adafru.it/iDR), or you can use the Adafruit Pi Finder ([https://adafru.it/enj](https://adafru.it/enj)) to install [Occidentalis](https://adafru.it/eB0), our growing collection of packages for single-board computers, which includes Node along with other development tools and configuration helpers.

```javascript
#!/usr/bin/env node
process.stdin.resume();
process.stdin.setEncoding('utf8');
function(data) {
  var stars = data.replace(/\S/g, '*');
  process.stdout.write(stars);
}
```

Concluding Remarks

The shell can be a complicated place, but hopefully by now you're starting to get a sense of how the pieces fit together. No one concept defines the Unix environment. Instead, Unix is something like a library of ideas. Many of them are old, dusty ideas,
and most have at least a few sharp edges, but they tend to be in the library because they can be combined with each other to solve new and complicated problems.

This series will continue with specific projects utilizing the Raspberry Pi and other single-board Linux computers. In the meanwhile, if you haven’t, you might want to check out the huge pile of Adafruit guides already available for the Raspberry Pi (http://adafruit.it/dpd) and the Beaglebone (https://adafruit.it/jWC).