Arch Linux Tutorial

Ginger Gengar

12^{th} July 2022

Contents

0.1 Rationale

Below explains why this tutorial exists. This tutorial is only valid for Arch Linux that was personally configured.

0.1.1 Why Use a Computer?

I cannot certainly speak for all users, I use a computer to help me do abstract work. Mathematics is the backbone of most major engineering fields. A lot of things in mathematics can be solved using analytical hand-written solutions, but many real-life problems cannot be solved that way. They must be solved numerically, by manual iteration. This is fine if your problem is small, but once your problem gets large enough, it becomes impossible to do by hand and instead, you have to turn to computers.

What kind of mathematical problems am I referring to? I am referring to ordinary differential equations, large matrix equations, tensor operation and even experimental symbollic computation. Why are these problems important? Solving these mathematical problems allows us to construct bridges, skyscrapers, train networks, cars, ships, aircrafts, rocketry, and most modern technologies we have today. Technology's backbone is theoretical engineering and theoretical engineering's backbone is mathematics, and mathematics can then be solved using general-purpose computers. So computers in this regard represent the last step of computing answers to. Computers are NOT A SUBSTITUTE for HUMAN THOUGHT.

What other things can be done by tinkering with a computer? Automation of manufacturing, moving actuators, flying aircrafts and driving cars. Lower level electronics that can marginally be called computers also perform alot of vital tasks in multiple industries, though I am not an expert in this regard, I deal more with the theoretical computations so far. There are many other applications of computer beyond what I have discussed, but the above are already very good reasons to learn computing.

0.1.2 Why Use Linux?

One of the most astounding things I still remember to this day switching from Windows to Linux (Ubuntu) was the existence of package managers. For some reason my machine did not come with mingw, which made it incapable of compiling `C`. I spent almost 2 days trying to install mingw navigating through old sites, and being concerned of accidentally installing malware. Eventually I failed to install mingw. In Linux (Ubuntu) though, the `C` compiler `gcc` was pre-installed, and even if it wasn't, a simple command to the package manager solves all of 2 days worth of problems in just 2 minutes, not even.

I also attempted to install a linear algebra library called `armadillo`. I also wanted to try out another linear algebra library `eigen`. I had to go through so many hoops installing all of the dependencies manually for `armadillo` such as `BLAS`, `SuperLU`, and others. With a linux system, a simple package manager command installs everything perfectly. So the point is, the package manager in Linux is absolutely necessary for any real programming work and is one of many great reasons to use Linux.

It didn't mean much when I first started Linux, but Linux is largely customizable. The text editor, kernel version, window manager, compositor, shell, and many other utilities can be configured and customized to the user's liking. This makes the Linux operating system an incredibly personal and efficient machine. Think about the Linux operating system as a lego piece whose parts can be defined by the user themselves and chosen. This also means that no 2 Linux operating system are exactly alike because each user has their own configurations and choice of programs.

The customizability of Linux also permits a variety of workflows, including `cli`-based workflows. `cli` stands for command line interface. This is a way to use the computer that abandons the mouse altogether and does everything from just the keyboard. It is faster and easier to do many tasks that

way. In this tutorial, `cli` would be used extensively. The last reason I have is an ideological one, which is that the Linux operating system is part of a free operating system. Free here as define by the `GNU` does not necessarily mean free of cost (although Linux is free of cost) but instead, free here means the users are free to examine the system in great detail, learn and expand it on their own. This is important because it gives the users the reigns to how computing is done. Users can verify no malware is pre-installed on the machine, and that the computer performs up to user expectations. Users are also not dependent on corporates which is good.

0.1.3 Why Arch Linux?

A Linux distribution just means a version of Linux. Arch Linux is a Linux distribution, which means that Arch Linux is a version of the Linux operating system. To be clear, all Arch Linux systems are part of the Linux family, but not all Linux systems is Arch Linux. There are a few reasons why Arch Linux is a good Linux operating system:

Arch Linux supports both binary installs and building from source. Compiled programs are translated from ther text form into machine code, which are 1's and 0's in human-unreadable form. The act of translating text format into machine code is called compilation. Binary installation is downloading those raw machine code and putting them in the appropriate directory inside the system. Building from source or source installation is downloading the human-readable text form, and then performing the compilation process on the user's machine before installing the resulting binary in the user's machine. Building from source always takes longer than binary install, but has the potential of resulting in a higher performing installed program. Arch Linux supports both types of installs, which makes it very flexible and can install a very large variety of programs.

Arch Linux is also very minimal. Minimal means that the base Arch Linux system has very few things installed in it. This means that users are free to build the system from the ground up, specifying the browser, terminal, window manager, compositor, status bar, text editor and other details themselves. A minimal system IS NOT IDEAL for users who believe an operating system should work out of the box. In a minimal system, one has to specify everything themselves, which is where the customization power of Linux is best revealed. Users who want to trade customization power for convenience should just try another distribution like Ubuntu, Linux Mint, Manjaro, etc. A non-minimal system has drawbacks when the user attempts to customize it because then the user has to uninstall unnecessary pre-installed packages, and then figure out the relationship between different packages and how to break their link and so on, which is more complicated than just configuring the system on your own. By configuring the system on your own, at least the user knows the different relationship bewteen different softwares they themselves have installed.

0.2 Program Configurations

In this section, we will learn how to configure basic programs.

0.2.1 Basic Terminal Usage

Starting programs, writing, and almost anything can be done through the terminal. To open a new terminal press:

```
super + enter
```

The super key is the command key, or the windows key on a typical keyboard. A new terminal instance should appear on pressing the keybinding above. To quit the terminal instances, press:

```
ctrl + shift + w
```

These keybindings have not been changed from the default in kitty. Kitty has tabs and windows. Windows are inside a tab. To open a new tab, press:

```
ctrl + shift + t
```

To navigate to the tab on the left

```
ctrl + shift + left
```

To navigate to the tab on the right:

```
ctrl + shift + right
```

Within a single tab, it is possible to open multiple windows. To do this, press:

```
ctrl + shift + enter
```

To navigate to the left window,

```
ctrl + shift + [
```

To navigate to the right window,

```
ctrl + shift + ]
```

The keybinding `ctrl + shift + w` can close windows one at a time and tabs as well, try experimenting with it. To open a new tab between existing tabs, press:

```
ctrl + t
```

This should cover the basics of terminal navigation.

0.2.2 Basic Window Manager Usage

Without anything else open, user resides in the first workspace. To go to the second workspace, press:

```
super + 2
```

To go the third workspace,

```
super + 3
```

Entering a different number will go to that workspace. To go back to the original workspace, press:

```
super + 1
```

Try opening 2 terminals by pressing `super + enter` twice for the two terminals. Try opening multiple tabs on one of them by pressing `ctrl + shift + t` successively for one of the terminals. To navigate between the terminal instances press:

```
super + left
```

Alternatively, one can press:

```
super + right
```

To make the terminal instances tile differently, One can navigate to either one of the terminal instances and press:

```
super + shift + left
```

One can experiment with the following commands for different ways to tile the terminals:

```
super + shift + up
super + shift + down
super + shift + right
```

One can think of the `super + shift` as the keybinding to "hold and drag" a particular terminal instance. The same logic can be applied to put terminal instances and other programs in different windows. Try navigating to one of the terminal instances and then pressing:

```
super + shift + 2
```

Pressing different numbers will move the terminal instances or other programs to different workspaces. For example, to move a terminal instance to the third workspace, navigate to the terminal instance and press:

```
super + shift + 3
```

The window manager also can force quit a window. To kill a particular window, navigate to it and then press:

```
super + shift + q
```

This should cover the basics of how the window manager works.

0.2.3 Basic Core Utilities

To make a new file use type in the following command in a terminal instance:

```
touch Pokemon.txt
```

`touch` is one of the core utilities in the Linux system, and the command above just means create the file named `Pokemon.txt` inside the current directory. Changing `Pokemon.txt` with another name would change the name of the file you are creating. For example, to create another file named `Digimon.dat` one can simply just type inside the terminal:

```
touch Digimon.dat
```

One difference coming from a `Windows` operating system is that file name extensions like `.txt`, `.dat`, `.exe` in windows tell the operating system what to do with those files. In Linux, file name extensions are more like suggestions for the content inside. The operating system will not care what the extension of those files are. So in a Linux system, naming a file `Pokemon.txt` and `Pokemon.dat` will not change how the computer treats the files. Now to see the list of files and folders in the current directory, invoke the `list` command by typing in the terminal:

```
ls
```

If you do that, then you will see the names of the files we have created in the previous prompts. The way to delete files is to invoke the `remove` command. To delete the file `Pokemon.txt` that is in the current directory, type in:

```
rm Pokemon.txt
```

The `rm` command has to be used with great caution. Deleting a file or directory using the `rm` command IS PERMANENT. Now it is time to create directories. To make a new directory named `PokeDex` invoke the command in the terminal:

```
mkdir PokeDex
```

Using the `list` command will give you a list of all the files and directories in a directory. There are a few ways to delete a directory. The `remove directory` command `rmdir` works if the directory is empty and has nothing in it. The `rm` command with a flag works even if the directory is not empty. To delete the directory `PokeDex`, one can invoke the command:

```
rmdir PokeDex
```

Alternatively, one can invoke the `rm` command to also delete the directory by saying:

```
rm -r PokeDex
```

In the above example, `-r` is a flag. We pass the `-r` flag to `rm` which means remove recursively all directories and files inside the directory `PokeDex`. After covering creation and destruction of directories, it is now important to be able to navigate between different directories. Create the directory again by invoking `mkdir PokeDex`. To navigate inside the directory `PokeDex`, say:

```
cd  PokeDex
```

Here the `cd` command stands for change directory. To go back to the directory that contains `PokeDex`, say:

```
cd ..
```

The command above would go back to where you once were. If you are confused about where you are in a linux system, invoke the command `pwd`. Type into the terminal:

```
pwd
```

The command above stands for print working directory. The command above would print where you are inside the linux system. Now you might be wondering how can I rename, move files in Linux? It is possible with the `mv` command. Suppose we are at the home directory and you only have a single file `Pokemon.txt` and you would like to rename this file into `Digimon.txt`, then you can simply invoke the following command if you are inside the same directory as `Pokemon.txt`:

```
mv Pokemon.txt Digimon.txt
```

Remember that file name extensions in linux do not mean anything, so if you wish, there is no need to even keep the `.txt` extension in the name `Digimon.txt` . Supposing you have a directory named `PokeDex`, by creating it using one the commands earlier and you wish to move the file `Digimon.txt` into the directory `PokeDex`. You can do this by invoking the command:

```
mv Digimon.txt PokeDex
```

You can use `cd` and `ls` to verify this has been done correctly. `mv` is a versatile command, it also works for renaming and moving directories into inside other directories. For example you have 3 directories `Games` and `Work` and `General`. You could move `Games` and `Work` to `General` by invoking the command:

```
mv Games Work General/
```

The `cp` command stands for `copy`. This command can be used to copy directories and files. Here is an example:

```
cp something1 AnotherThing2
```

Here `something1` can be a file, a directory, it can be anything, and `AnotherThing2` is the name of the copy. To make your life easier, just press `tab` to autocomplete the commands in the shell. Another useful command is:

```
clear
```

This command clears the terminal of all the previus commands. There is a special directory that is meant to be user-specific. That is the home directory To go to the home directory, one can say,

```
cd ~
```

with the `~` character. Invoking the command,

```
cd
```

also just immediately jumps to the home directory.

0.2.4 Basic Text Editing

My specific setup uses `neovim`, which is based off the famous text editor `vim`. `vim` can be a little intimidating to work with initially. There is an in-built vim tutorial. To see the in-built vim tutorial, key in to the terminal

```
vimtutor
```

There one can use the `up down` arrow keys initially. The tutorial uses a vim-like keybinding. To exit the tutorial, type in the following in the vim/Neovim session,

```
:q
```

This is also information that can be found after reading the tutorial. The tutorial does not take too long. Reading the tutorial is highly recommended. Suppose you have some text file, which you have downloaded, or created using the `touch` command, then to open a file named `Helios.txt` one can key into the terminal:

```
nvim Helios.txt
```

in the same directory where the file `Helios.txt` resides. `nvim` is the name of the text editor, which is Neovim. `Helios.txt` is the name of the text file you wish to edit. Additional arguments may be passed to `nvim` or the classical vim editor `vim` to specify additional behaviour. For example, the terminal command,

```
nvim -Ro Helios.txt
```

would make `Neovim` open the file `Helios.txt` in a Read Only format, so you are unable to make accidental changes to the file by default. Here I am going to discuss just the basics of editing a file, and saving a file. For full information about vim, just use the `vimtutor` command. This document will only cover the bare basics of using vim/Neovim, more advanced techniques will not be covered here.

Once you have opened a file, using the command prescribed above, Neovim/vim by default enters normal mode, which means you cannot start writing yet. To be able to edit the file, press `esc` in the vim/neovim session to enter normal mode, and then press, `i` inside the vim/Neovim session to enter "Insert" mode. After that, one can start editing the file. Even after you have done all the edits, your changes are not yet saved into the file. To save the changes into the file, firstly press `esc` to enter normal mode again, and then press `:w` to write the changes into the file. Pressing `esc` allows you to enter normal mode. Once in normal mode, pressing `u` undos the latest changes you've made, meanwhile `ctrl + r` redos the changes you've made.

Once you are satisfied with the edits, you can quit by first entering normal mode by pressing `esc` inside the vim session, and then quitting by pressing `:q`. Remember that you should save changes to the file before quitting (vim will usually throw a warning if you attempt to quit before saving changes). When quitting, one can also do something like this from normal mode: `:wq`. The command `:wq` is a shortcut for Write and Quit. This can be an easier faster way of quitting vim. DO NOT CONSIDER TEXT EDITING A TRIVIAL TASK. TEXT EDITING IS ONE OF THE MOST IMPORTANT SKILLS FOR INTERACTING WITH THE COMPUTER.

0.2.5 Dotfiles & Config Files

The terminal that is configured in this system is `kitty`. `kitty` by default does not look like it does on the machine. The colors have been changed, and the tab bars have also been changed. The font sizes have been changed and so on and so forth. The keybindings for the window manager like `super + shift + 2` to move something to workspace 2 are also modified from their original versions. How can we tell the program user-specific preferences?

The way users can specify their preferences are by writing files inside the `.config` directory in the home directory. All directories that start with a `.` are commonly known as "dotfiles". Directories that start with a `.` are assumed to be system details that the user does not need to know about. So, a simple naive `ls` command will not show all of the "dotfiles". "dotfiles" are also known as "hidden files". Additional flags can be passed to `ls` to allow it to see the "dotfiles". Navigate to the home directory by invoking the command:

```
cd ~
```

Once there, invoke the `list` command with the flags `-a`. This will specify `list` to show all files inside the directory, including the hidden ones:

```
ls -a
```

This may look a bit ugly, so pass the additional parameter `-l` to make it look like a list:

```
ls -l -a
```

This might take a long time to type, so you can make it shorter by instead invoking:

```
ls -la
```

This is a shorter notation for `ls -l -a`. Once you have performed any of the `list` commands above, you can see the "hidden files" such as `.bashrc` and a `.config`. Navigating to the `.config` directory and then invoking the command `ls` would show all of the different programs. Navigating to one of the program directory would reveal another file. For example, navigate to the `kitty` directory. Inside that directory, there is a file `kitty.conf` and `colors.conf`. `kitty.conf` is where the user specific configurations for the `kitty` terminal resides. Changing the contents of the file `kitty.conf` with a text editor from the previous part changes the behaviour of the `kitty` terminal. The same treatment can be applied to different programs to personalize their behaviours.

What to put inside the config file of each program is all program dependent and I would not reiterate the different options here. For more details for each program, visit the individual program pages and read the appropriate documentation regarding customizing the program.

0.3 Reading & Writing: LaTeX

LaTeXis a powerful typesetting system for communicating technical ideas effectively. It is beautiful, elegant, and comprehensive. Here is a list of a few things LaTeXcan do when writing documents:

- Write Mathematical equations quickly and beautifully: This includes matrix equations, tensor notations, and many more
- Write chemical equations: This is relevant for the chemists out there
- Write clean tables: This is an important aspect of data visualization
- Include code blocks and listings: This comes with fully customizable color and syntax highlighting
- Draw out complex circuit diagrams: THis is for the electrical computer engineers who need to communicate circuitry across
- Command line based, so LaTeXis scriptable and automatic documentation can be generated

This document is not meant to be a fully-fledged LaTeXtutorial. I have another guide for that, discussing in greater depth how to use the LaTeXsystem more effectively. This document is just to show how LaTeXfits to my particular Arch linux setup.

Writing in LaTeXis relatively straightforward. One can just write a file containing LaTeXcode and then call the LaTeXcompiler to generate a PDF document. This PDF document can then be viewed using some PDF reader such as zathura. One needs to know how to use a PDF reader first to examine the output of the LaTeXcompiler. So it is advised to read this section sequentially.

0.3.1 Reading PDF

The PDF reader for my particular linux setup is zathura. Suppose you have a PDF document named `Ruler.pdf` in some directory. To view the PDF document, navigate a terminal instance to the same directory `Ruler.pdf` resides in, and then invoke the command below:

```
zathura Ruler.pdf
```

This will open a new zathura instance, showing the contents of `Ruler.pdf` . Here are a few default keybindings on how to do a basic usage of zathura:

- `ctrl + r` toggles dark mode and light mode. This might be useful for working at night
- `r` simply rotates the document clockwise, this is useful for viewing landscape documents without tilting your head.
- `d` toggles viewing 2 pages at the same time or just 1 page at the same time
- `gg` goes to the beginning of the document
- `shift + g` goes to the end of the document
- `=` puts the document to fit the screen
- `shitf + plus` is used to zoom into the document
- `minus` is used to zoom out of the document
- `ctrl + m` is used to open another instance of the same exact PDF document

It is important to know that zathura is just one PDF reader you can choose out of many. There are other PDF readers out there such as Okular, Evince, XPDF, GNU GV, Mupdf, firefox, and more. I have just made the choice of choosing zathura because of its vim-like keybindings and configurability, but each user can choose for themselves, a PDF reader they like. To demonstrate alternate PDF viewers, try opening the document not with zathura, but with firefox this time. Just like before navigate to the directory `Ruler.pdf` resides in, and then try invoking the command

```
firefox Ruler.pdf
```

This should open a firefox instace showing the contents of `Ruler.pdf` . Yes, `firefox`, the browser, has an in-built PDF reader which makes `firefox` capable of displaying PDF documents, so firefox can be used as a PDF reader if you like.

0.3.2 Writing LaTeX

Suppose you have a file named `Example.tex` in some directory, then to generate the PDF document, navigate the terminal to the same directory as `Example.tex` and invoke the following terminal command:

```
pdflatex Example.tex
```

If you are unsure what to put in the LaTeXfile initially, just copy paste the content below:

```
\documentclass[a4paper, 12pt]{report}

\begin{document}

This is some text.

\end{document}
```

The LaTeXcode above should produce a simple one page document with the content "This is some text". The small LaTeXcode above can be used to at least learn how to compile LaTeXdocuments manually. Typically though, to make life a little easier, a vim plugin was already installed called `vimtex`. This allows the compilation and viewing of a PDF to be directly handled from just inside neovim. To compile and see the document, just type in the following command while on normal mode inside a vim session opening a `.tex` file.

```
\ll
```

To look at more features of `vimtex`, simply type in the command, inside a vim session,

```
:help vimtex
```

To see any compilation errors, type the following while on normal mode:

```
\le
```

To see a table of content for the document and jump to specific parts of the document, type in the following while on normal mode:

```
\lt
```

By the end of writing and examining your first LaTeXdocument, the workflow of a modular system should be clear. The general goal was to produce a document for communicating mathematical or other technical ideas. To achieve this goal, we used 3 separate programs that only do one thing, but does that thing really well:

1. Neovim: Text editor to write the LaTeXcode, which is the content we want to communicate
2. LaTeXCompiler: This converts the LaTeXcode edited from Neovim into a full-fledged PDF
3. Zathura: This is a PDF reader which allows us to see the final result of the document generated by the LaTeXCompiler

All of the programs shown above work well together and is used to generate the beautiful document we have written. Any of the 3 programs above can be switched for another alternative based on whatever you, as the user want, and all of the 3 programs above are extremely configurable, allowing the programs above to fit to the specific needs of the user. If you wish to construct even more complex tasks, the way to do it usually is not to scrap existing programs for larger more comprehensive programs, but instead to just add on newer small programs to fill the required task. For example, a workflow I have ever written for a single task looked like this:

1. Neovim: Text editor to write the LaTeXcode, C code, Python code, and Bash script
2. LaTeXCompiler: This converts the LaTeXcode edited from Neovim into a full-fledged PDF. By importing a package, LaTeXwas capable of inlining computer code
3. Zathura: This is a PDF reader which allows us to see the final result of the document generated by the LaTeXCompiler
4. GCC: a `C` language compiler that was used to convert `C` code into binary, which was used to run a theoretical simulation
5. python3: a python interpreter which was used to visualize the data generated by the `C` program and also to confirm hand-typed mathematical analytical expressions
6. Bash: a shell interpreter that "glues" the `C` and python programs together, allowing results to be generated automatically by a single command

Each of those programs in itself is relatively small, and does only one thing but does it well. Together, the small programs could be used to construct a massively complex idea from the ground up with less human duplicate work and present it in a beautiful manner.

0.4 Basic Programming

0.4.1 Compiling Programs

Compilation is the process of taking a file with the code written in it and translating the code into machine language. Most compiled languages are going to be compiled in the exact same way. Suppose we have a C program in some directory named `c_program.c`. To run the program, navigate a terminal instante to the same directory and then type in the following comand:

```
gcc c_program.c
```

Use the `ls` command to see the files in the directory before and after compilation. After compilation, there will be a binary named `a.out`. To run this binary in a linux environment, type in:

```
./a.out
```

The output of the program will then appear on the terminal console. For users without significant programming background, copy the following content into the file `c_program.c` :

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
printf("Hello␣World!\n");
return EXIT_SUCCESS;}
```

Compiling and running the program shown above should result in "Hello World!" being printed into the terminal console after running with `./a.out`. More advanced compilation options would be available, but this document is not meant to delve into too much detail on `C` programming.

0.4.2 Running Interpreted Programs

Back in the early days of computing, there was a split in how programs ran on the computer. Two types of programming languages that arose were the compiled programs and interpreted programs. As mentioned before, compiled programs translate all of the human-readable text into machine language all at once and then running the program is just running the translated machine language as we have demonstrated in the previous subsection.

Interpreted programs take on a different approach in running them. Intepreted languages like python typically have an interpreter whose task roughly is to look at the program code and go line by line. Each line, the interpreter translates the program code into machine language and then runs the translated instructions. This means that interpreted languages unlike their compiled counterparts require translation from source code into machine language every single time the program runs instead of just once for the compiled language.

Interpreted programs have an inherent overhead compared to compiled languages which means that interpreted languages are much slower than their compiled counterparts. There are several advantages to having intepreters. Interpreted programs can be interrupted in the middle and new code instructions can be keyed in by the programmer. This makes debugging interpreted programs much easier compared to their compiled counterparts. So the main takeaway of an interpreted language is that it is easier to write and develop in, but it is slower when run on the computer.

Suppose you have some directory with a Python program named `python_prog.py`. To run the program, navigate a terminal instance to the directory where the program Python program resides and run the following command:

```
python python_prog.py
```

This should run the python program. It is much simpler to run interpreted programs compared to compiled programs. Users who are unfamiliar with the python programming language can copy the code into `python_prog.py` just to test they can succesfully run a python program:

```
#!/bin/python

print("Hello␣Snake!")
```

Running the program above should result in "Hello Snake!" being printed into the terminal console. Entering the terminal console and executing code from there is one great advantage of interpreted programs. To enter the console, key in the command in a terminal instance:

```
python -i
```

From there, one can type python code and upon pressing enter, it will be executed. The following is an example:

```
Python 3.10.5 (main, Jun  6 2022, 18:49:26) [GCC 12.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> x = 1
>>> y = 3
>>> x+y
4
>>> print(x)
1
>>> print(y)
3
>>> print(x*x)
1
>>> print(y*y)
9
>>>
```

To quit the python terminal, simply type in:

```
ctrl + d
```

while inside the python console. For debugging, to enter the python terminal console after the program has finished and examine the variables directly, type in the command:

```
python -i python_prog.py
```

This will give the user command over the python console after `python_prog.py` has finished, allowing the programmer to examine the variables and debug the program effectively.

0.5 Shell Scripting & Custom Scripts

0.5.1 Trivial Hello World Program

Try typing a command into the terminal

```
echo "Hello World"
```

This command will print "Hello World" into the terminal console. It doesn't seem very useful in itself but will be used to show how commands typed into the bash terminal can be automated using a script. Now create a file and name it whatever you wish, but for this tutorial, it would be named `Hello_world_Bash`. Copy over the `echo` command into the file but this time add one line at the top like this:

```
#!/bin/bash

echo "Hello␣World"
```

Change the permissions of the file by using the command:

```
chmod + x Hello_world_Bash
```

After this, the file can be executed by running the command,

```
./Hello_world_Bash
```

and the outupt of the command would be,

```
Hello World
```

What have we done here? The process described above is essentially shell scripting. We have a command (but later this can be a set of commands) that we know works and we can execute by typing. However, being lazy, we decided that we wanted to do less typing.

Therefore, we created a file and wrote the command we would otherwise type ourselves into the file. This file has a special line at the top `#!/bin/bash`. This is called a shebang, which is a one-line description how the shell should run or interpret the script. For this particular case, we have specified that this script should be run by a `bash` interpreter located at `/bin/bash`.

After that, we changed the file premissions using the `chmod` command. We gave the file execute permissions which tells the shell of the system that this file is a script and contains executable instructions. This was the step `chmod +x Hello_world_Bash`.

After all the steps above, the bash script has been made and is ready to run. The final command `./Hello_world_Bash` is a command to execute the contents of the script. This bash script though, is only available if the terminal is at the working directory the script resides in. To be able to call the functionality from anywhere, one must let the shell know to look at the script's directory. To do this, one must edit the `PATH` variable in the `.bashrc` script located in the home directory. Without too much detail, one can add the following line in the `.bashrc` script located in the home directory to make the script executable from anywhere within the linux system:

```
PATH="DIRECTORY_THE_SCRIPT_HELLO_WORLD_BASH_RESIDES_IN"
export path
```

Now from anywhere in the linux system, calling `Hello_world_Bash` prints "Hello World!"

0.5.2 Screen Brightness Example

`light` is a program that is used to control the brightness of the computer screen To increase the brightness of the screen by 5 percent, key in the command,

```
light -A 5
```

To decrease the brightness of the screen by 5 percent, key in the command,

```
light -U 5
```

It is really cumbersome to have to keep typing `light -A 5` every time we need the screen a little bit brighter. Therefore, this is the perfect excuse to write a bash script to automate the screen lights. The 2 scripts below can be used to increase brightness and reduce brigthness,

```
#!/bin/bash
#This is for increasing brightness
light -A 5
```

The script to decrease brightness:

```
#!/bin/bash
#This is for decreasing brightness
light -U 5
```

In shell scripting, anything that starts with a `#` symbol is a comment, which will not be read by the interpreter. This is with the exception of the shebang that was written at the very first line of the file, which tells the shell what interpreter to use to execute the file's content. If the files for making the screen brighter is named `MakeBrighter` and the file for making the screen dimmer is named `MakeDarker`, then the following command will add an execute permission to both of those files:

```
chmod +x MakeBrighter
chmod +x MakeDarker
```

Alternatively to operate on both files with just a single command,

```
chmod +x MakeBrighter MakeDarker
```

Now if we are in the same directory, we can invoke the script to make the screen brighter by using the command,

```
./MakeBrighter
```

Alternatively, we can make the screen a bit darker by invoking the command,

```
./MakeDarker
```

Of course, we would want to be able to invoke the command from anywhere in our system right? So add the directory the `MakeBrighter` and `MakeDarker` script reside in to the `PATH` variable. Add the following to the `.bashrc` file located in the home directory `~`:

```
PATH="DIRECTORY_THE_SCRIPT_MakeBrighter_Resides_In"
PATH="DIRECTORY_THE_SCRIPT_MakeDarker_Resides_In"
export path
```

Note that the `export path` command needs to be written just once in the entire `.bashrc` file. Doing the above would allow the command `MakeBrighter` and `MakeDarker` to be called from wherever you are inside the system. There is an additional step that can be taken which would require configuring the window manager to view the `MakeBrighter` and `MakeDarker` scripts and call those scripts whenever a particular keybinding is pressed. That way, controlling the screen brightness can be done fully through pressing buttons on the keyboard. As a matter of fact, my personally configured system controls screen brightness the exact same way. This is the content of my `MakeBrighter` file:

```
#!/bin/bash

#Author: Hyahoos

#Increase Brightness
light -A 5
```

This is the content of the `MakeDarker` file:

```
#!/bin/bash

#Author: Hyahoos

#Make Screen Darker:
light -U 5
```

This is basically identical in functionality to whatever we have written earlier. This is another way the shell can be used to control system-related things.

0.5.3 Running Consecutive Programs

Suppose we're working on some complex project wherein we need to compile `programA.c` and then collect the resulting output named `data.dat`. The output file is then run into a series of Python post-processing programs, for plotting which would be named `programB.py`. Several symbollic computations also need to be verified along the way which would be handled by several interpreted Python scripts `programC.py` and `programD.py`. These are the likely commands to run when dealing with the terminal:

```
gcc programA.c
./a.out
python programB.py
python ProgramC.py
python programD.py
```

The commands above assume that running the program using `./a.out` would result in an output file being generated as specified from within the `C` program. If the `C` program instead prints out the results to `stdout`, we can instead do something like this:

```
gcc programA.c
./a.out > data.dat
python programB.py
python ProgramC.py
python programD.py
```

This ammendment allows us to specify the name of the file from interacting with the shell. The `>` operator means if there is any data in the file `data.dat` erase it and overwrite it with the new data incoming. Perhaps the Python script `programB.py` accepts a name of a file instead of having the file name hardcoded in. Therefore, we can also try doing this:

```
gcc programA.c
./a.out > data.dat
python programB.py data.dat
python ProgramC.py
python programD.py
```

Now writing all that command can be cumbersome, so a bash script can be written into a file to automate writing all those commands. The bash script may look like this:

```
#!/bin/bash

#Compiling the Simulation Program
echo "Compiling␣the␣Simulation␣Program..."
gcc programA.c

#Running the Simulation
echo "Running␣and␣Collecting␣Simulation␣Data..."
./a.out > data.dat

#Doing some Post-Processing Work
echo "Generating␣Graphs␣and␣Data␣Visualization..."
python programB.py data.dat

#Doing some Symbollic Computation Verification
echo "Doing␣some␣Symbollic␣Computation␣Verification..."
python ProgramC.py
python programD.py
```

Remember that the `#` commands are just comments, meanwhile the `echo` commands just print to the terminal console (`stdout`). The `echo` commands are just used to tell which part of the script the shell is currently at. It is pure cosmetics, the `echo` commands are not that important, the script can run just fine without the `echo` messages.

This is how to automate larger and increasingly complex work using bash scripts. It is important to note that in the above, we were only dealing with a single file for the compiled simulation, which is `programA.c`. In many instances, it may be more convenient to have multiple files for the compiled part of the simulation. Which means that compilation has to handle multiple files. To do this in a more efficient way, build tools such as `Makefile` can be used to automate the compilatio of compiled programs into binary.

`Makefile` is a tool for `C` and `C++` programs, but there are other tools and other programming languages out there. For example, the `Rust` programming language is also compiled and comes shipped fully with its own automated build tools named cargo. So compiling `Rust` multi-file programs could be done using `cargo` instead of interacting natively with the shell `bash`.

0.6 Basic Package Management

Arch linux supports 2 primary ways of building and installing programs. Please remember that compiled programs need to be translated from source code (human-readable text) into machine

instructions (binary). The first most common way of installing a program is called binary install. The second less commonly used way of installing a program is called build from source.

When the user invokes a binary installation, pacman, the Arch Linux package manager fetches binaries from Arch linux servers. These binaries have already been compiled for the target machine architecture (x86-64). After the package manager fetches the binaries, it will install them in the somewhere in `/bin` or `/usr` or other similar locations near the root directory `/`. Since compilation has already been performed by the package maintainers, a binary install is fast, as no compilation is needed. Since the binaries are compiled for a wide variety of machines, the compiled code might not be fully optimised for the target machine. However, this is insignificant in most cases.

The user can also invoke building from source. Building from source typically fetches source code from the AUR (Arch User Repository). Usually building from source is built using an AUR helper, such as yay. When the user invokes a build from source, the AUR helper, in our case yay, would roughly perform the steps shown below:

1. Create a directory inside `/.cache/yay`, which will be used for building from source
2. In the directory, download all the source code
3. After downloading source code, begin compilation process
4. After compilation, the binaries are made into an Arch package
5. The Arch package manager, pacman would be invoked on the Arch package, still located in that directory
6. The Arch package manager, then installs the built package

Of course, the above is a simplified idea of how building from source works. To read the full detail, read the Arch wiki on the ABS (Arch Build System). Those articles would provide the complete picture. Here we will discuss the basics of doing a binary installation. When considering a binary installation, the first thing to determine is whether the program we want is available in the Arch servers. To figure out if such a program named `PROG_NAME` is available in the Arch servers, try the following command:

```
pacman -Ss PROG_NAME
```

Let's understand the command above. The `-Ss` parts are called flags. The flags are used to specify to the package manager `pacman` that we would like to search for a program named `PROG_NAME`. The command above should give a list of programs with similar names as `PROG_NAME`. If the program does not exist, then consider a build from source install (this will be explained later). Otherwise, just proceed with the binary install. To do a binary installation of a program named `PROG_NAME`, use the following command:

```
sudo pacman -Syu PROG_NAME
```

Let's break down the command above. `sudo` is used to give the current user root privileges, without the `sudo` command, the user would not have enough authority to install programs in the system. `pacman` is the name of the package manager. `-Syu` are the flags which are passed to the package manager `pacman`. The `-Syu` flag is used to install the program specified and upgrade any outdated packages in the system. `PROG_NAME` would be the name of the program passed to the package manager. Saying the command above should install the program painlessly.

After installing the program above, it is important to make sure that the installation was succesfull. One can immediately try using the program, but this method is depends on what the program really does. It is also possible to verify that the package manager has installed the program with the command below:

```
pacman -Qe
```

This would give out all the packages installed on the machine. But it is quite unreadable because it writes out everything all at once. To make the list more readable, use the following command:

```
pacman -Qe | less
```

What does `| less` do? The `|` operator is known as the pipe operator. The pipe operator redirects the output of the program `pacman -Qe` into the program `less`. The program `less` then puts whatever `pacman -Qe` produces in a kind of scrollable list. Pipes `|` as well as redirect operators `>` are very important for linux systems. The package named `PROG_NAME` could be uninstalled using the following command:

```
pacman -Rsu PROG_NAME
```

Building from source should be relatively easy as well, because most of the flags used for pacaman applies for the yay AUR helper as well. To search for packages:

```
yay -Ss PROG_NAME
```

To install a specific program,

```
yay PROG_NAME
```

0.7 Learning Program Usage

You might be wondering if there is a way to quickly read the different flags for different programs without having to remember them all. Fortunately, there are a couple of ways to seek help when we forget how to use a particular command. For example, a pretty universal trick is to pass the `-h` flag or the `--help` flag to the program which is the "help" option. Usually a brief description on how to use the program would appear if we use the help flag. For example, if we forget how to use the basic `mv` command, we can read a brief description by typing:

```
mv --help
```

If we forget how to use a file compression program such as gzip, then we can try getting an easy summary of program usage by typing:

```
gzip --help
```

In general, a good idea is to try the following for an unknown program named `PROG_NAME`:

```
PROG_NAME --help
```

If we need more detail on how a particular program is used, we can consult the manual pages. Opening the manual pages is very easy. Suppose we have a program named `PROG_NAME` we can see its manual page by using the command:

```
man PROG_NAME
```

For example, if we want to read the manual page for the package manager, pacman, we can use the command:

```
man pacman
```

Manual pages and help options are a great way to learn about a new program. The purpose of this document is to just teach the bare essentials on getting started with a Linux system. For anyone to be an expert at linux systems, they need to be able to learn by themselves by reading the necessary documentations. These documentations can be found through the manual pages, help options, Arch wiki, Official program page, github, and many other sources.

0.8 Miscellaneous

0.8.1 Keybinding Summary

0.8.2 Further Reading

General Arch Linux

The Arch Linux Wiki contains a lot of technical information. When trying to debug the system, the Arch Linux Wiki is a great rescource:
`https://wiki.archlinux.org/`

More information about the ABS (Arch Build System):
`https://wiki.archlinux.org/title/Arch_Build_System`

Kitty Terminal

This is the official page for the kitty terminal:
`https://sw.kovidgoyal.net/kitty/`
The Github source code for the kitty terminal:
`https://github.com/kovidgoyal/kitty`

0.8.3 Art Gallery