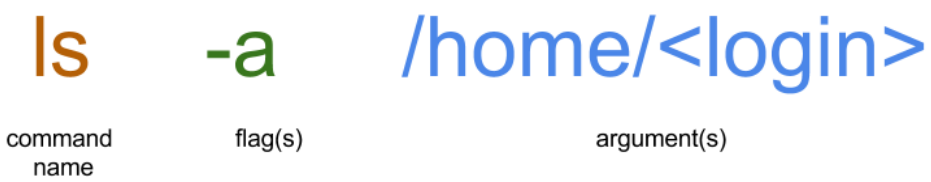
Introduction to Bash - 2015

1 What is Bash?

Bash, or the Bourne Again SHell, is a Unix shell which allows users to type commands which are then processed and sent to the operating system. This shell is very similar to the one used in Apple computers, so if you have a Mac, you can most likely run these commands on the terminal on your personal computer as well.

Commands generally have three parts: the name of the program being run, flags which change how the program executes or displays information, and arguments which can specify path of data to work on, where to put a result, or alter the operation of the program.

For example, in the following command:



we can see all of the parts of a command in action. The first thing to note is that all the parts of a command are separated by spaces. The shell splits your command into tokens that it can understand, almost like how we split a sentence into words. `ls` is the name of the command, `-a` is a flag, and `/home/<login>` is an argument. In this example, `<login>` would be replaced with your CS department login, for example `/home/aisha`. Flags are generally prefixed by one or two dashes, and single-character flags can generally be combined (for example, `-a -l` becomes `-al`). The `/home/<login>` argument is a file path, which is where we want to run the `ls` command. So, this command runs `ls` on the folder `/home/<login>` with the `-a` flag set, which will display the contents of that folder and include all hidden files. We'll learn more about `ls` later.

In the next few steps we'll give you an overview of some useful commands as well as how to use them. We'll also go over a brief explanation of the file system.

For some of you, parts of this may be review, but you will be expected to know and correctly use these commands in CS16. Additionally, this lab goes through essential setup for the upcoming **Introduction to Python**, so please pay close attention and execute all of the commands provided.

2 Running your first commands

First, start by opening your terminal. You should see something that looks like:

```
cslab1a /home/aisha $
```

This is called your command line **prompt**. Any commands that you run will go after the `$`. You should notice that the first part of this prompt consists of the name of the computer you are on, in the Sunlab, this is one of `cslab[1-9][a-h]`. The second part of this prompt shows you which directory you are currently in.

You can also find your current directory by running `pwd`, which stands for "print working directory". Run this command now.

Your terminal should look something like this (but with your login instead of "aisha"):

```
cslab1a /home/aisha $ pwd
/home/aisha
cslab1a /home/aisha $
```

Congratulations! We've learned our first shell command!

3 More about your file system

In order to explore your file system, you'll need to know one command in addition to `pwd`. This command is `cd`, or **c**hange **d**irectory. `cd` will switch your "working directory" (or current directory) to whatever you give it as an argument. If you give it no arguments, it will take you to your home directory. Keep in mind, this command can take either *relative* or *absolute* paths. An absolute path starts with a slash and is the same regardless of your current directory (ex: `/home/aisha/course`). A relative path does not begin with a slash and describes a location relative to your current directory (ex: if your working directory is `/home/aisha`, the relative path `course/cs015` would evalute to `/home/aisha/course/cs015`).

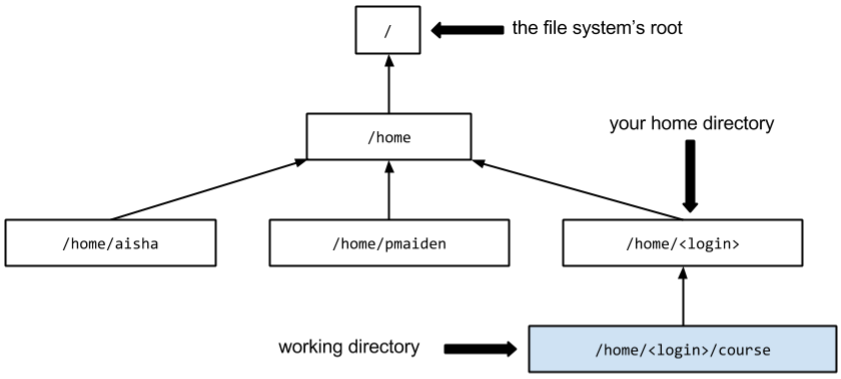
You can think of the file system almost like the containment diagrams you made in CS15. Your topmost directory is `/` (called the "root"). Each other directory has a "parent" (a directory which contains it) and any number of "children" (directories which it contains). Each directory on the path is succeeded by a slash.

Lastly, you will often notice that people type commands such as `cd ..` or `cd ~`. Single and double dots are used to signify the current directory and parent directory respectively and $\sim$ represents your home directory (`/home/<login>`). Note that some people's paths may show up as `/gpfs/main/home/<login>` instead of `/home/<login>`. If yours looks like this,

just ignore the first two parts of the path! We will explore this some in the next example.

Now, let's practice using `cd`. Enter the following commands and check your output against the provided example. Take a look at the diagram provided below the commands and try to follow the file system structure as you go. You should end up at the blue highlighted box.

```
cslab1a /home/aisha $ cd ..
cslab1a /home $ cd ~
cslab1a ~ $ cd course
cslab1a ~/course $ cd cs015
cslab1a ~/course/cs015 $ cd ..
cslab1a ~/course $ cd .
cslab1a ~/course $
```



4 Commands for manipulating the file system

You should now be in your course directory. Let's start by making a directory for cs016. To do this, you'll use the program `mkdir`, which takes a single argument – the path of the directory to make. In most cases, you'll be using relative paths, so you will only need the

name of the new directory. Before we do this, let's try out the first command we saw, `ls`. `ls` stands for **l**ist **s**egments, but really what it means is "show me what is in my working directory". Let's try this in our course directory now. You should see a folder for each CS course you've taken, you will most likely see something like this:

```
cslab1a ~/course $ ls
cs015/
cslab1a ~/course $
```

Now let's make our directory:

```
cslab1a ~/course $ mkdir cs016
cslab1a ~/course $
```

We have now made an empty `cs016` directory. Next, we're going to try to give it some children files and/or directories to prepare us for the first lab.

Checkpoint: *Try moving into your newly made cs016 directory. Remember, there are two different ways to do this (using relative and absolute paths). For practice, try moving in both ways.*

Now that we're in our `cs016` directory, let's try making a file. We know already that we use `mkdir` to make a directory. To make a file, we use the command `touch`, which also takes an argument specifying the path (or in most cases the name). Let's call this file `test`. Try making the file and then using `ls` to make sure that we've made it correctly.

```
cslab1a ~/course/cs016 $ touch test
cslab1a ~/course/cs016 $ ls
test
cslab1a ~/course/cs016 $
```

Checkpoint: *Try making a directory called* `pythonIntro` *within your* `cs016` *directory and the move into that directory. We will use this directory in the Introduction to Python lab.*

Now we'll need to copy the stencil for the python lab into this directory. To copy we will use the command `cp` which takes two arguments, first what to copy, and second where to copy it to. You'll notice, we're using `*`, which is a wildcard meaning "all". In this case, using `/course/cs016/asgn/intros/pythonIntro/stencil/*` as our first argument means that we want to copy everything from the `stencil` directory. We use `.` as our second argument because we want to copy everything to our current directory.

```
cslab1a ~/course/cs016/pythonIntro $ ls
cslab1a ~/course/cs016/pythonIntro $ cp /course/cs016/asgn/intros/pythonIntro/stencil/* .
cslab1a ~/course/cs016/pythonIntro $ ls
pieCount.txt pieCounter.py* primePrinter.py* sectionOne.py*
```

We're going to now learn `mv`, which stands for move. It is used not only to move files, but also to rename files. Like `cp`, `mv` takes two arguments, where to move a file or directory from and where to move a file to. Now let's move your file `test` into your `pythonIntro` folder and then rename it `README`. You are going to need to make a `README` file for each of your projects which documents design, bugs, and other things. You can learn more about this in the docs and will also hear about them in your project handouts.

```
cslab1a ~/course/cs016/pythonIntro $ mv ../test .
cslab1a ~/course/cs016/pythonIntro $ mv test README
```

Last but not least, we need to know how to remove any files or directories that we have created but no longer want. **Please DO NOT try any of these until you have read the whole section.** To do this, we use a command called `rm` (or `rmdir` for directories). `rm` and `rmdir` take a single argument - the file/folder to remove. There are two flags that are important with `rm`, `-r` and `-f`. `-r` means recursive, so if you want to remove a directory as well as everything in it, you can use `rm -r <directory>`. Note that you cannot remove a directory that has files/subdirectories in it using `rmdir`, which is why we use `rm -r` instead. You will also notice once you try these that it will ask you for each file whether you're sure you want to remove the file or directory. You will have to type "y" or "yes" in order to give it permission to delete. In order to override this behavior, you can add the `-f` flag to `rm`, for example, `rm -f` or `rm -rf`.

Use this with caution! It is really easy to delete everything using these commands and while sunlab consultants can restore your files from 2 hours previous using something called "snapshots", this can sometimes mean quite a bit of work lost. Let's try this out without the `-f` flag to see how it works:

```
cslab1a ~/course/cs016/pythonIntro $ rm README
rm: remove regular empty file ‘README’? y
cslab1a ~/course/cs016/pythonIntro $ mkdir toDelete
cslab1a ~/course/cs016/pythonIntro $ touch toDelete/emptyFile1
cslab1a ~/course/cs016/pythonIntro $ rm -r toDelete
rm: descend into directory ‘toDelete’? y
rm: remove regular empty file ‘toDelete/emptyFile1’? y
rm: remove directory ‘toDelete’? y
cslab1a ~/course/cs016/pythonIntro $
```

You should now be all ready for the Introductoin to Python. Please run `cs016_bashCheckoff` so we know that you've completed it. If you've set everything up successfully you should get a success message, otherwise, you should go back through the lab and make sure you have completed all parts of the intro. The next few pages include a list of other helpful tips and a detachable bash cheatsheet listing both the commands we have already learned

as well as some other commands that your TAs thought may be helpful.

Tip: If you ever get confused about a command, you can use the man pages (effectively a bash user manual) by typing `man <command>`, where `<command>` is the command you want more information about. For example, `man ls` will tell you about all of the flags you can use to see different sorts of files in a directory.

5 Other helpful tips

- Adding `&` to the end of a command will run it in the background. This will mean that your terminal will still be usable when you're running a program.
  ex: `sublime &`
- If you forget to add an `&` but want to add it afterwards, type *ctrl-z* in your terminal, which will pause your program, and then type `bg` to start running it in the background instead.
- Add `& disown` to the end of a command makes it so when you close your terminal, the program isn't automatically closed.
  ex: `google-chrome & disown`
- If you're not sure where in a path you're going next or what a file name is as you are typing, you can use the tab key to autocomplete or be given a list of possible completions to your command.
- To open a new tab in your terminal, type *ctrl-shift-t*