Bash Basics

What is Bash?

Bash is a *shell*. Think of this as the interface between you and all of the applications and services you might want to run on your computer. A shell *is a program* that is running and mediating your interaction with other programs on your computer: it is responsible for coordinating input and output, launching programs, etc. In fact, the original versions of Windows were called "graphical shells"—they let you run applications using a graphical user interface instead of the command line.

Unix-like systems (including Linux and MacOS) have *command-line* shells that let you interact with your computer using a command-line interface: typing commands and getting responses in text. One of the most popular command-line shells is called **bash**, and it is available on virtually all Unix-like systems—and is the default command line shell on most Linux installations as well as MacOS.

Shells also provide *scripting languages* that let you coordinate multiple actions to perform a single task. For example, you could run one program, and, based on the result of that program, run a second program multiple times. Or you could run one program that generates a list of all the files in a directory, and then take the output of that program to run a second program on each of the files in the directory. We will go over some of the basic things you can do with bash here—there are many resources online that cover the full features

Command completion

Bash lets you "complete" commands without having to type them all the way out. You can hit [tab] to try to finish a command, and bash will look for commands (or filenames) that start with the partial text you have typed to try and complete the command. For example, suppose you have a directory with the following contents:

```
milind@scholar-fe06:~/bash-lect $ ls
bar  baz  cmdline.sh  foo  input.py  outerr.py  sample.sh  testvar.sh
```

The command `cat` will list the contents of a file:

```
milind@scholar-fe06:~/bash-lect $ cat foo
This is a sample text file
```

But if I type a command partially, then hit tab:

```
milind@scholar-fe06:~/bash-lect $ cat f[tab]
```

Then bash will complete the command for me. In this case, there is only one file that starts with 'f', so I get the complete command that I expect:

```
milind@scholar-fe06:~/bash-lect $ cat foo
```

But if there is more than one option to complete the command:

```
milind@scholar-fe06:~/bash-lect $ cat b[tab]
```

Then bash will fill in what it can—in this case, there is more than one file starting with a ‘b’, but both start with ‘ba’:

```
milind@scholar-fe06:~/bash-lect $ cat ba
```

And if I hit tab a second time, bash will show me the possible options for files that start with ‘ba’:

```
milind@scholar-fe06:~/bash-lect $ cat ba[tab]
bar  baz
milind@scholar-fe06:~/bash-lect $ cat ba
```

And I can then continue typing.

Command history

Bash also remembers the commands that I have executed in the past, letting me access them if I want to re-execute them, or modify them. Simply hitting [up] at the command line will step backwards through your command history, and hitting [down] will then step back forward through your history. You can also print out the history of commands you have executed in the past:

```
milind@scholar-fe06:~/bash-lect $ history

…
  349  ls
  350  cat foo
  351  history
milind@scholar-fe06:~/bash-lect $
```

You can then execute a particular command from your history by typing “`![n]`” where n is the number of the command:

```
milind@scholar-fe06:~/bash-lect $ !350
cat foo
This is a sample text file
```

You can also execute the last command by typing `!!`

```
milind@scholar-fe06:~/bash-lect $ !!
cat foo
This is a sample text file
```

There are a lot of other things you can do with bash history — executing a command but modified slightly, re-executing the last time you used a specific command, etc., but we will not cover them here.

Chaining together multiple commands

You can also use bash to chain together multiple commands:

```
milind@scholar-fe06:~/bash-lect $ cat foo; cat bar
This is a sample text file

This is a different text file
```

Or:

```
milind@scholar-fe06:~/bash-lect $ cat foo; cat bar; cat baz
This is a sample text file

This is a different text file

This is a third text file that lets us see how tab completion works
```

Chaining together commands with semicolons is nice. But what happens if one of those commands has an error? For example, if you try to `cat` a file that doesn't exist:

```
milind@scholar-fe06:~/bash-lect $ cat nope; cat foo
cat: nope: No such file or directory
This is a sample text file
```

Maybe that's what you want. But what if you want to be smarter. For example, suppose the two commands you are trying to run are: "compile this program" and "execute this program." If compiling the program fails (say, because you have a syntax error in your code), you don't want to then run the program that does not exist — that will just cause another error.

```
milind@scholar-fe06:~/bash-lect $ gcc test.c ; ./a.out
test.c:4:27: error: expected ';' after expression
   printf("Hello, World!")
                          ^
                          ;
1 error generated.
zsh: no such file or directory: ./a.out
```

You only want the second command to run *if the first command succeeds*. We can do this by using a logical *and* connective, instead of a semi-colon:

```
milind@scholar-fe06:~/bash-lect $ gcc test.c && ./a.out
test.c:4:27: error: expected ';' after expression
   printf("Hello, World!")
                          ^
                          ;
1 error generated.
```

Note that now, `a.out` never gets executed, because the first command fails. However, if the first command *succeeds*, the second command will correctly execute:

```
milind@scholar-fe06:~/bash-lect $ gcc test.c && ./a.out
Hello, World!
```

The way && works is that it keeps executing commands left-to-right until it encounters a command that fails. (In class, we explained why it works this way — it's a *short circuit* operator. As soon as a command fails, that means it is impossible for a logical-and to evaluate to true, so bash stops trying. As long as commands are succeeding, it is still possible for logical-and to evaluate to true, so bash keeps going.)

So what if you want the opposite behavior? Execute a command only if the previous command *failed* (maybe you want to run a command that does some error handling if the first command fails). You can do this with the || operator:

```
milind@scholar-fe06:~/bash-lect $ cat nope || cat foo
cat: nope: No such file or directory
This is a sample text file
```

But if the first command succeeds, then bash will *not* execute the second command:

```
milind@scholar-fe06:~/bash-lect $ cat bar || cat foo
This is a different text file
```

Shell scripts

One simple thing you can do with bash is put together multiple commands into a single file, and treat that as a new command that you can invoke. This file is called a shell script. Think of this as being able to create custom commands. We will look at some simple examples where you are using shell scripts to just run multiple commands one after another. But you can also use shell scripts to do more complicated things (run loops, for example).

Here is a sample shell script:

```
milind@scholar-fe06:~/bash-lect $ cat sample.sh
#!/usr/bin/env bash

cat foo
cat bar
```

Ignoring the line at the top, all we are doing is writing down a sequence of commands to execute (in this case, printing out the contents of two files). But what is that line at the top? When you run a shell script, we need to know what *interpreter* to use to execute the script. In this case, we want to use bash itself! To do that, we write `#!/usr/bin/env bash`. This lets the operating system know that the commands that are coming next in the script should be executed using bash (i.e., they are bash commands). If we wanted to use python instead, we might write: `#!/usr/bin/env python`

As an aside, you might often see the first line in a script written a little differently (`#!/usr/bin/bash`). This is an alternate way of invoking bash as the interpreter, but relies on bash being in a particular location on the machine, which may not always be the case.

Once you create a shell script, the next thing you have to do is set the *permissions* on the file so that the operating system knows that it can be executed. To do this, we use the command chmod:

```
milind@scholar-fe06:~/bash-lect $ chmod u+x sample.sh
```

Which says "change the permissions on sample.sh to add (+) the executable (x) flag for the user (u)." You could instead change the permissions so that *anyone* can execute the script, not just you:

```
milind@scholar-fe06:~/bash-lect $ chmod a+x sample.sh
```

(Note that you can also use chmod to make a file readable/writeable by you/anyone, etc., but we won't go into the details of that here).

Now when you run sample, the script executes both commands:

```
milind@scholar-fe06:~/bash-lect $ ./sample.sh
This is a sample text file

This is a different text file
```

Variables

Bash, recall, is a full-on scripting language, even though you can also use it interactively as a command-line shell. What you're really doing, in that case, is running bash commands one at a time, instead of putting them together into a whole program (interestingly, you can run other scripting languages—including Python!—in a similar manner). Like any good programming language, bash lets you define *variables*, names that correspond to values:

```
milind@scholar-fe06:~/bash-lect $ MYVAR=foo
milind@scholar-fe06:~/bash-lect $ echo $MYVAR
foo
```

That first line defines a variable named `MYVAR`, and sets it equal to `foo`. Note that it is important that there are not spaces between `MYVAR` and =. By convention, variables in bash are in capital letters, but they do not have to be.

The second line is a little more interesting. ${variablename} in bash *replaces* the variable with the value of that variable. So `echo $MYVAR` is equivalent to `echo foo`

An important point, here, is that `MYVAR` is a local variable—it is only accessible to the current bash instance. Notably, this means that if you run a shell script, `MYVAR` *will not exist* within that shell script. This is similar to how if you have a local variable in a function in C, it is only accessible within that function. To make a variable global (well, sort of — in reality, this makes the variable accessible until the bash instance exits), you *export* it:

```
milind@scholar-fe06:~/bash-lect $ export MYVAR

milind@scholar-fe06:~/bash-lect $ export NEWVAR=bar
```

Then, within a shell script, you can access the variable just like you would any other way:

```
milind@scholar-fe06:~/bash-lect $ cat testvar.sh
#!/usr/bin/env bash

cat $LOCALVAR

milind@scholar-fe06:~/bash-lect $ export LOCALVAR=baz
milind@scholar-fe06:~/bash-lect $ ./testvar.sh
This is a third text file that lets us see how tab completion works

milind@scholar-fe06:~/bash-let $ export LOCALVAR=bar
milind@scholar-fe06:~/bash-lect $ ./testvar.sh
This is a different text file
```

You can also define new variables based on old ones (bash variables are essentially always strings, though some commands implicitly treat strings as numbers):

```
milind@scholar-fe06:~/bash-lect $ VAR1=foo; echo $VAR1
foo
milind@scholar-fe06:~/bash-lect $ VAR2=$VAR1; echo $VAR2
foo
milind@scholar-fe06:~/bash-lect $ VAR3=${VAR1}.out; echo $VAR3
foo.out
```

Note that in the last example, we put VAR1 in braces. This just guarantees that we are going to access a variable named VAR1. It is not always necessary, but it is good practice.

Redirection

When a program prints output or reads input in Unix-like systems, it always writes to or reads from *file descriptors* that correspond to the destination (or source) of the content being input/output. But the system provides special file descriptors called `stdout` — for printing "regular" output — `stdin` — for reading regular input — and `stderr` — for printing error/warning information. When you invoke the "standard" print command in most programming languages (e.g., `printf` in C or `print` in Python), you are going to print to `stdout`. When you invoke the "standard" input command (e.g., `scanf` in C or `sys.stdin.readline()` in Python), you are reading from `stdin`.

By default, `stdout` prints to the screen, and `stdin` reads from the keyboard input (i.e., the command line):

```
milind@scholar-fe06:~/bash-lect $ cat input.py
#!/usr/bin/env python3

import sys
```

```
inp = sys.stdin.readline()

print("Read this line: " + inp)
milind@scholar-fe06:~/bash-lect $ ./input.py
Hello
Read this line: Hello
```

Here, for clarity, we are highlighting input provided from the keyboard in blue, and output sent by the program to the screen in red.

We can *redirect* these standard file descriptors in bash to send the output not to the screen but to a file, or to accept input not from the command line but from an input file. The special character "`>`" sends stdout to the specified file (creating the file if one doesn't exist, and overwriting the contents of the file if it already exists):

```
milind@scholar-fe06:~/bash-lect $ ./input.py > tmp
Hello
milind@scholar-fe06:~/bash-lect $ cat tmp
Read this line: Hello
```

If you use the special character "`>>`", it redirects stdout to the specified file but *appends* the output to the file (rather than overwriting the existing file).

```
milind@scholar-fe06:~/bash-lect $ ./input.py >> tmp
Hello
milind@scholar-fe06:~/bash-lect $ cat tmp
Read this line: Hello

Read this line: Hello
```

The special character "`<`" redirects `stdin` to the specified input file (i.e., it is as if the contents of the input file are typed in from the keyboard):

```
milind@scholar-fe06:~/bash-lect $ ./input.py < foo
Read this line: This is a sample text file
```

You can combine both input redirection and output redirection:

```
milind@scholar-fe06:~/bash-lect $ ./input.py < foo > tmp
milind@scholar-fe06:~/bash-lect $ cat tmp
Read this line: This is a sample text file
```

`stderr` is often used by programs to print errors and warnings. This is a *different* file descriptor than `stdout`, so redirecting `stdout` will not redirect `stderr` (deliberately so: that way you will see errors as they happen even if you are redirecting output to a file):

```
milind@scholar-fe06:~/bash-lect $ cat outerr.py
#!/usr/bin/env python3
```

```
import sys

print("This goes to stdout")

print("This goes to stderr", file=sys.stderr)

milind@scholar-fe06:~/bash-lect $ ./outerr.py
This goes to stdout
This goes to stderr
milind@scholar-fe06:~/bash-lect $ ./outerr.py > tmp
This goes to stderr
milind@scholar-fe06:~/bash-lect $ cat tmp
This goes to stdout
```

Here, we are redirecting `stdout` to `tmp`, but letting `stderr` print to the screen (coded in purple for clarity). You can also choose to redirect `stderr` (e.g., if you want to send errors to a log) using "2>":

```
milind@scholar-fe06:~/bash-lect $ ./outerr.py > tmp 2> log
milind@scholar-fe06:~/bash-lect $ cat log
This goes to stderr
```

If you want to redirect both stderr and stdout to the same file, use "&>":

```
milind@scholar-fe06:~/bash-lect $ ./outerr.py &> both
milind@scholar-fe06:~/bash-lect $ cat both
This goes to stderr
This goes to stdout
```

Pipes

Many times, you will want to run one program and send its output to another program. You *could* do this by redirecting the output of one program to a file, then sending that file in as the input to the second program, but bash provides a simpler facility for doing this: *pipes*. A pipe redirects the output of one program to the input of another automatically:

```
milind@scholar-fe06:~/bash-lect $ ./outerr.py | ./input.py
This goes to stderr
Read this line: This goes to stdout
```

Pipes are commonly used to chain together simple utility commands in bash. For example, listing all the files in a directory is easy:

```
milind@scholar-fe06:~/bash-lect $ ls -l
total 64
-rwxr-xr-x 1 milind student 8928 Aug 23 09:42 a.out
-rw-r--r-- 1 milind student   31 Aug 21 08:51 bar
-rw-r--r-- 1 milind student   69 Aug 21 08:52 baz
-rw-r--r-- 1 milind student   40 Aug 23 11:55 both
-rwxr--r-- 1 milind student   38 Aug 21 09:08 cmdline.sh
```

```
-rw-r--r-- 1 milind student   28 Aug 21 08:51 foo
-rwxr--r-- 1 milind student   96 Aug 21 09:41 input.py
-rw-r--r-- 1 milind student   20 Aug 23 11:55 log
-rwxr--r-- 1 milind student  113 Aug 21 09:44 outerr.py
-rwxr--r-- 1 milind student   37 Aug 22 09:51 sample.sh
-rw-r--r-- 1 milind student   50 Aug 23 09:42 test.c
-rwxr--r-- 1 milind student   36 Aug 22 09:54 testvar.sh
-rw-r--r-- 1 milind student   20 Aug 23 11:55 tmp
```

But what if I want to only see the files that have the extension "`.py`"? You can do this by piping the output of `ls` to the command `grep`, which does a string search:

```
milind@scholar-fe06:~/bash-lect $ ls -l | grep py
-rwxr--r-- 1 milind student   96 Aug 21 09:41 input.py
-rwxr--r-- 1 milind student  113 Aug 21 09:44 outerr.py
```

In particular, `grep <pattern> <filename>` will search in the file `<filename>` and `print out any lines that contain` <pattern>.