Part One

Scripting in the Bourne-Again Shell (bash)

What is a shell?

- A shell is a program that interprets user input, line-by-line, for the machine
- It allows the user to access all programs available on the computer

Part 1: Outline

1. file expansion and wildcards
2. variables, redirection, and substitution
3. shellscript
4. getting arguments from terminal
5. for loops and if statements

File Expansion and wildcards

File Expansion (1)

Filename expansion. When bash sees

```
$ ls *.txt
```

It expands the command to, for example:

```
$ ls a.txt b.txt
```

This is the simplest type of file expansion

File expansion (2)

A fairly complete list of expansion terms

```
*         Zero or more characters
?         One of any character
\         Escape following special character
[xyz]     Matches any of the enclosed characters
{a,b,c}   Matches any of the enclosed strings
```

File expansion: examples

```
$ ls *.txt              # all ending with .txt
$ ls *.{docx,doc}       # e.g. .doc or .docx
$ ls f[1-3].txt         # f1.txt f2.txt f3.txt
$ ls *.doc?             # e.g. .doc or .docx
# The horror, the horror
$ ls ??_[0-9]*.{fa,faa,fna,gff}
```

Escaping special chars

```
$ rm Harry Potter.pdf  # two files
$ rm Harry\ Potter.pdf # escape space
$ rm \*.txt      # removes ‘*.txt’
$ rm ‘*.txt’     # or just quote it
```

Do It Yourself (DIY)

- Spend a few minutes trying **?**, **[ ]**, and **{ }** in your shell
- Try listing specific sets of files on your filesystem
- Do you run into any interesting errors?

Using wildcards with `rm` can be dangerous!

Variables

Variables

Declaring variables:

```
$ x=5
$ echo $x
$ y='path/to/some/file'
$ head $y
```

No spaces allowed in declaration!

Combining strings

```
$ x=’cats and dogs’
$ y=’ and a pony’
$ echo $x$y
cats and dogs and a pony
```

Double and single quotes

```
$ x='Alice'
$ y='Bob'
$ echo "$x sent $y a message"
Alice sent Bob a message
$ echo '$x sent $y a message'
$x sent $y a message
```

Space in Bash is special

```
$ x=cats and dogs                    # WRONG
bash: and: command not found
$ x='cats and dogs'                  # RIGHT
```

Spaces are separators in Bash

Bash interpreted ‘and’ as a command

Anonymous variables $()

Evaluate a command, retrieving output as a variable

```
# The two lines below have the same output
$ head *.txt
$ head $(ls *.txt)
```

DIY (2)

Define and echo a few variables

Combine them into new variables

Try a few anonymous variables

Experiment, see what breaks

```
$ echo ‘todays date is: ‘$(date)
```

Piping and Redirection

Redirection

Given A and B are programs and f is a file

```
A | B    Pipe STDOUT from A to STDIN of B
A > f    Overwrite f with A’s STDOUT
A >> f   Append A’s STDOUT to end of f
```

Redirection Examples

```
# redirect output of head into a file
head a.txt > b.txt
# you can do the same thing like this
cat a.txt | head > b.txt
# append the 100th line to b.txt
head -100 a.txt | tail -1 >> b.txt
```

Shellscripting

What are shellscripts

Anything you type into your terminal, can be pasted into a file and executed

The code in the shellscript is read line-by-line by the bash interpreter, just like the lines you type into your terminal

Hello World in shellscript

```
#!/bin/bash
echo ‘hello world’
```

- Write the above two lines into a file
- Make it executable (**`chmod`** `755 hw.sh`)
- Call it (`./hw.sh`)

Hashbang (#!)

You need to tell the system what program should interpret your script

Syntax:

#! /bin/bash

#! /usr/bin/python

Calling a script (example)

```
$ cat myscript.sh
  #!/bin/bash
  ls *
$ chmod 755 myscript.sh  # make executable
$ ./myscript.sh  # execute! (why ‘./’?)
```

Sending stuff to the script

All the words that come after a command are the command's arguments, e.g.

```
$ rm -f a.txt b.txt c.txt
  $0 $1   $2    $3    $4
```

Your shellscript can also take arguments

Getting arguments

```
$ cat myscript.sh
#!/bin/bash
echo “$2 $1 $3”

$ ./myscript.sh 10 20 30
20 10 30
```

Arguments must be space-separated

DIY (3)

Modify your Hello World script to print some arguments

Try putting some other commands in the script

For-loops and if statements

For-loop demo

```
for f in *.fa
do
   echo “processing $f”
   blastp -query $f -db mydb > $f’.output’
done
```

Bash for-loops: Syntax

```
for x in <list>; do
    <code>
done
```

For-loop example (1)

```
#!/bin/bash
for x in 1 2 3; do
   echo $x
done

for x in 1 2 3; do echo $x; done
```

For loop example (2)

```
# For each file, write its head to new file
$ for x in *.txt; do head > $x'.head'; done
```

For-loops are particularly useful when you have many input files *and* many output files

DIY (4)

Test a for-loop in your shellscript

Input checking

It is good practice to check whether the input to a script is valid, this is done with **if-else** statements

If-statement syntax

```
if [[ <condition> ]]
then
    <code>
fi
```

Dying gracefully

```
# If myfile.txt isn't readable stop the script
if [[ ! -r myfile.txt ]]; then
   exit
fi
```

`!` means NOT

`-r` tests readability of the file

Useful Tests

```
-r file is readable
-d directory exists
-z test is a variable is empty
```

Conclusion

If you are working with lots of input and output files, calling lots of programs, and making sophisticated pipelines, shellscripts are wonderful.

But don’t code deep logic and algorithms in it.