Advanced Unix Shell

```
$ echo "Data Sciences Institute"
```


Navigate Files / Directories

Files

Knowing the different types of files available helps us better understand how to navigate and manipulate them.

- Regular files are text files with readable characters.
- Executable files are programs that are invoked as commands.
- Shell scripts are readable executable files, in contrast to bash, which is a non-human-readable executable

Directories

Directories, similar to folders, contain files and subdirectories, forming a hierarchical structure.

- We can think of the structure of directories as a tree with the top of the tree being the root.
- All files can be named and found in relation to the root by listing the directory names in order from the root, separated by slashes, followed by the file's name.

Paths

Using `cd` and `pwd`, let's explore how to use absolute and relative pathnames.

```
$ cd
$ pwd
```

```
$ cd Desktop
$ pwd
```

```
$ cd 15
$ pwd
```

Let's now try move through some directories to get comfortable. Try out lots of different paths depending on the file structures of your computer. Try getting into different directories from different parent directories. The tilde notation ~ in the examples below refer to our home directory.

```
$ cd ~/Desktop
$ pwd
```

```
$ cd ~/Desktop/dir1
$ pwd
```

Options and Arguments

Options and arguments are used to write commands that can make changes to our system. The syntax is:

```
$ command --option argument
```

Options can also be combined, a topic we'll briefly touch on now and explore in more detail later.

There are two ways to write an `--option`:

1. Short option: one dash followed by a single character
2. Long option: two dashes followed by a word

   Some examples:

   `-a` or `--all`
   `-d` or `--directory`
   `-r` or `--reverse`

Let's try these lines of code and see what happens:

```
$ ls -l
```

```
$ ls -lt
```

```
$ ls -lt --reverse
```

- `-l` long format
- `-t` modification time
- `-reverse` reverse the sort order

Notice how the `-lt` command combines multiple options.

Wildcards

Wildcards allow us to quickly specify groups of filenames based on character patterns.

Let's look at a few examples below:

- `*` > matches any character
- `?` > matches any single character
- `[characters]` -> matches any character that is in the set
- `[!'characters]` -> matches any character that is not in the set

Some other helpful character wildcards are:

- `[:digit:]` > matches any numeral
- `[:lower]` ~ matches any lowercase letter
- `[:upper:]` > matches any uppercase letter

Let's try a few in our terminal:

```
$ ls
```

```
$ ls *
```

```
$ ls ax.txt
```

```
$ ls [abc]x
```

```
$ ls [[:upper:]]x
```

```
$ ls [![:digit:]]x
```

Input / Output

Standard Input/Output

Each program uses standard input, output, and error channels. We can think of the standard input default as coming from the keyboard and if we think of everything as a file, a command such as `ls` will result in a file called `standard output` and the status message to a file called `standard error`. By default, both are linked to the screen and not saved to a disk file.

Input/Output Redirection

Input/Output redirection allows us to change where the input comes from and where the output goes to, such as storing the output of a command into a file. We can do this using the redirection operator `>`.

```
$ ls -l /usr/bin > ls-output.txt
```

Here we have redirected the output of `ls --l /usr/bin` to a `.txt` file called *ls-output.txt*.

We can now see the details of that file and if it worked:

```
$ ls -l ls-output.txt
```

By examining the details, we can see that the file was created and is a fairly large text file, indicating that content was written to it.

If we specify a directory that does not exist, we receive the Standard error:

```
$ ls -l /bin/usr > ls--output.txt
```

Why was the standard error not written to the `.txt` file?

What happened to our *ls-output.txt* file?

Although the standard error was not written to the `.txt` file, the destination file is always written from the beginning, therefore, the redirection began to write the file and once noticed there was an error, stopped, resulting in an empty file.

So how do we append rather than rewrite? By using the redirection operator `>>`.

```
$ ls -l /usr/bin >> ls--output. txt
```

If we want to redirect the standard error, we need to use the redirection operator `2>`

```
$ ls -l /bin/usr 2> ls--error.txt
```

If we want to redirect both the standard output and standard error to one file, we have two options.

1. Use `2>&1` at the end of the command.

```
$ ls -l /bin/usr > ls-output.txt 2>&1
```

2. Use `&>` inplace of `>`

```
$ ls -l /bin/usr & 1ls-output. txt
```

cat

`cat` takes one or more files and copies them to standard output. Using the previously created `ls-output.txt`, we can see how this is done:

```
$ cat ls-output.txt
```

We can also use it to join files togther. Let's say I have two files, `file1` and `file2` and I want to combine them into a file called `file3`:

```
$ cat filel file2 > file3
```

Now the contents of file1 and file2 should be combined.

We can also use `cat` to add to a `.txt` file.

```
$ cat > new_cat.txt
```

Now we can type the desired text into the file. To finish, use `CTRL-D` to exit.

What is the difference between `$ cat > new_cat.txt` and `$ cat >> new_cat.txt` ?

Finally, we can redirect the standard input from the keyboard to the file *new_cat.txt*

```
$ cat < new_cat.txt
```

This is almost identitcal to just typing *$ cat new_cat.txt* but we can see later how it could be more useful.

Pipes / Filters

Pipelines read data from standard output and send it to standard input using the pipe operator `|` . This means the standard output of one command can be piped into the standard input of another.

Several commands put together in a pipeline are often referred to as filters. Filters take an input, change it and then output it.

Commands

Let's learn a few more commands that will help us further understand pipelines and filters. We'll learn:

- extract columns from output `cut`
- sort lines of text `sort`
- report or omit repeated lines `uniq`
- print lines matching a patter `grep`
- search directories and subdirectories for files `find`
- ouput the first part of a file `head`
- output the last part of afile `tail`

cut

Let's look at a `csv` to see how we Can initially see our data. Because it's a `csv`, each line is separated by a comma. Let's first read that file using `cat`:

```
$ cat parking_data.csv
```

We'll see a lot of text, so let's make some sense of it using cut.

To use cut, I need to pass a couple options:

1. `-d` which cuts the text based on what follows. For example, `-d:` will cut based on colons or `-d" "` will cut based on a space.
2. `-f`, which extracts a particular field based on what follows. For example, `-f1` will take the first field or `-f2` will take the second field and so on.

In this example, I'm taking the file *parking_data* and cutting it based on colons and then only extracting the first field.

```
$ cut -d, -f1 < parking_data.csv
```

What happens if I add another `-f` option? What does this do?

```
$ cut -d, -f1 -f2 < parking_data.csv
```

How would I specify more than three fields?

sort

How can we make our previous example more readable?

One answer is to use the sort feature. We can pipe this with the cut feature:

```
$ cut -d, -f1 < parking_data.csv | sort
```

uniq

Additionally, I can make the above command even more readable by removing any duplicates with `uniq`

```
$ cut -d, -f1 < parking_data.csv | sort | uniq
```

grep

`grep` is a powerful tool for finding patterns in text files. The syntax is:

```
$ grep pattern [file...]
```

In our case, we're going to use it with our previous example and pipe it with other commands:

```
$ cut -d, -f1 parking_data.csv | sort | uniq | grep FIRE
```

The results are all patterns of FIRE in the text file.

find

Another useful use for `grep` is to find files in directories. `grep` is nicely combined with `find` for this feature.

```
$ find ~/Desktop/dir1 | grep cat
```

Here we're searching in the directory *dir1* with the pattern *cat*. This would be helpful if we wanted to know if there were any files with the word cat in the filename.

head | tail

We can also extract the first and last part of files using `head` and `tail`. We can also add the option `-n` followed by a number to extract a certain number of lines.

```
$ head -n 5 ls-output.txt
```

```
$ tail -n 5 ls-output.txt
```

`head` and `tail` can also be used in pipelines:

```
$ cut -d, -f1 < parking_data.csv | sort | uniq | head --n 5
```

```
$ cut -d, -f1 < parking_data.csv | sort | unig | tail -n 5
```

Expansions

Expansion uses special characters to expand upon something before the shell processes it. We have learned a few expansions so far such as the tilde `~` and wildcards `*`. We've also seen some character wildcards `[characters]`.

Expansions are another feature that help us when we're manipulating and working with files and directories.

Other examples of expansions are:

- arithmetic expansion
- brace expansion

Arithmetic Expansion

Arithmetic expansion basically makes the shell a calculator. The syntax is:

```
$((expression))
```

For example:

```
$ echo $((2 + 2))
```

Arithmetic expressions can be nested:

```
$ echo $(($((2 + 2)) * 3))
```

Just for reference, here is a list of the arithmetic operators:

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Integer division
`**`	Exponentiation

Brace Expansion

Brace expansions allow us to create multiple text strings from a pattern containing braces. Here are a few examples:

```
$ echo Test-{A,B,C}-Example
```

```
$ echo Number_{1..5}
```

```
$ echo {Z..A}
```

Brace expansions can also be nested:

```
$ echo a{A{1,2},B{3,4}}b
```

We can use brace expansion to help make multiple directories using `mkdir`.

```
$ mkdir dir-{1..3}
```

This command makes 3 directories named *dir-1*, *dir-2*, and *dir-3*

Quoting / Backslashing

Quoting suppresses unwanted expansions. We can use double quotes, single quotes or backslashes:

- Double quotes force special characters to lose their meaning and are treated as ordinary characters except for `*` , `\` , and `'` .
- Single quotes suppress all expansion
- Backslashes are used to escape single characters

Many times there will be file names or directories that are named with spaces. In this case, we'll need to use double quotes so that the shell can read it. Using `touch` we can create a text file named something separated with two words:

```
$ touch "two words.txt"
```

We can then see the details of the file we just created:

```
$ ls -l "two words.txt"
```

If we want to rename the text, we would do as follows:

```
$ mv "two words.txt" two_words.txt
```

Let's see what these three examples do in shell:

```
$ echo '2 * 3 > 5 is an equation’
```

```
$ echo '2 * 3 > 5' is an equation
```

```
$ echo 2 \x 3 \> 5 is an equation
```

Command Line Editing

Getting familiar with command line editing can save you time. Bash uses a library called Redline to use command line editing

There are many shortcuts and you don’t have to memorize them all, just use the ones that you feel are best. There are even more shortcuts that you can read about in the textbooks!

Command Character

Command	Description
CTRL-B	Move one character backwards
CTRL-F	Move one character forwards
DEL	Delete one character backwards
CTRL-D	Delete one character at cursor location

Word Commands

Command	Description
ESC-B	Move one word backwards
ESC-F	Move one word forwards
ESC-DEL	Delete one word backwards
ESC-D	Delete one word forwards
CTRL-Y	Undo

Line Commands

Command Description

Command	Description
CTRL-A	Move to the beginning of the line
CTRL-E	Move to the end of the line
CTRL-K	Delete text from the cursor to the end of the line
CTRL-U	Delete text from the cursor to the beginning of the line

History Line Commands

Command	Description
CTRL-P	Move to the previous line in your history of commands
CTRL-N	Move to the next line in your history of commands
`!!`	Repeat the last command
`!number`	Repeat history list item number
`!string`	Repeat last history item starting with string
`!?string`	Repeat last history item containing string

Completion Command

Completion commands autocomplete your command if it exists by hitting `tab`. If it does not exist, the command will not be able to complete.

If multiple exist, the command will also not be able to complete because it will not know which one to choose.

For example, let's say we have two files called `file1` and `file2`. If would not be able to use autocomplete because the shell will not know which to choose until the last character.

If we have two files, one called `foot.txt` and one called `file.txt`. This command would not be able to autocomplete:

```
$ ls f
```

But this one will:

```
$ ls fil
```

Shell Scripts

Shell Scripts

Shell scripts allow us to combine several commands into one file, rather than one by one on the command line.

The shell will read the script just as if you were to write the command on the command line.

Most things that can be done in the shell script can be done on the command line and vice versa.

Writing Shell Scripts

There are three important considerations when writing the shell script

1. **Write a script:** scripts are ordinary text files. You can use a text editor that will provide syntax highlighting (color coding elements of the script). It can help find errors but writing in TextEdit is possible.
2. **Make a shell script executable:** set the script permissions to allow it to be executed
3. **Put the shell script somewhere the shell can find it:** the shell script automatically searches certain directories for executable files when no explicit pathname is specified.

Set Up

Open either TextEdit or your text editor of choice. Some popular programs are Sublime Text, Vim, Atom, and Notepad++.
If you want to see the syntax highlighting, you might have to save your script as a `.sh` file.
Without doing this, your file will just look like a regular `.txt` file.
Once you open your text editor and save it, we can begin our first script!

Script File Format

We must first tell the shell the name of the interpreter that should be used to execute the script. This is marked by using a shebang: `#!`
Throughout the script, you can and **should** use `#` to make comments. Comments make your code more readable and can help you understand your code when you come back to it.

```
#!/bin/bash
# this is our first comment
echo "This is our first script!"
```

Here we can see we've told the shell to use `/bin/bash` using the shebang `#!`
We've also added a comment using `#`
And finally, something quite familiar, we have our first line of script using `echo`

A Note on Commenting

Commenting is important not just so you can understand your own work, but also so others can understand your work in collaborative projects. It also helps make your code reproducible.

Comments can be inline:

```
echo "Hello World" #this is an inline comment
```

or as comment blocks:

```
#this is a comment block
echo "Hello World"
```

Executable File Permission

In order to execute our file, we have to add file permissions: `chmod` helps make our script executable, `775` is used to make scripts that everyone can execute, `700` is used to make scripts that only the owner can execute

Here, `chmod` is combined with `775` so that everyone can execute the script:

```
$ ls -l first_script.sh
```

```
$ chmod 775 first_script.sh
```

Script File Location

In order to run our script, we have to call it using `./` in front of the script filename (`./Script`).
File location is important to run your script. If just `Script` was written, the shell would not be able to find the script and try to read it as a command, outputting `command not found`.
Running `echo $PATH` helps us see what directories are being searched for the script.

If we want to run our script without `./`, we can create a `/bin` for our script, move our script into the bin folder and then run it. It's important to note that we have to make this bin in our home directory. If we made it on our Desktop, the script would still not be found.

```
$ mkdir bin
$ mv first_script.sh bin
$ first_script.sh
```

In this block of code, we're making the bin folder using `mkdir`, moving the script into the bin with `mv` and then running the script without `./`.

Good Locations for Scripts

For personal use, a good place to put your script is `/bin`.
For everyone's access, it's better to put scripts in `/usr/local/bin`.

Shell Functions

Functions

Functions are a good way to break down code into smaller, more manageable chunks. Each chunk can represent a task.

For example, let's say your entire process is make pasta. It can be broken down into:

1. Prepare vegetables
2. Make sauce
3. Cook pasta
4. Serve

Each of these steps can be expanded further into subprocesses. Cook pasta can be:

1. Fill pot with water
2. Boil water
3. Measure pasta
4. Add pasta to boiling water
5. Cook for 8-12 minutes
6. Strain

Functions have two syntactic forms:

```
function name {
    commands
    return
}
```

```
name () {
    commands
    return
}
```

`name` is the name of the function

`commands` are the commands contained in the function

Let's write our first function:

```
#!/bin/bash
function funct {
    echo "Step 2"
    return
}

---
#program starts here

echo "Step 1"
funct
Echo “Step 3"
```

What do you think this function will output?

Let's save and run this function in our terminal to see what happens.

Here's a good time to recap how to save, grant permissions, and run the script.

- `chmod` - permissions command
- `775` - grant permissions to everyone
- `700` - grant permissions to yourself
- `/bin` - where to save permissions

Variables

Global Variables

Let's make our script more complex with some variables. We can first define variables directly through the terminal.

```
$ foo="Something cool"
$ echo $foo
```

Notice how in order to call the variable we need to add `$` before the variable. The quotes are not necessary if the value of the variable doesn't include spaces when defining it. If we did not include the quotes here, we would receive an error.

Now let's add some global variables to our script:

```
#!/bin/bash

step="Step 2"

function funct {
    echo $step
    return
}

#program starts here

echo, step yl”
funct
echo "Step 3"
```

What do we think will be the output in this example?

Local Variables

Local variables are variables that are contained within the function. Because they're contained, they can have names that already exist in the shell globally or within other shell functions.

```
#!/bin/bash

foo=@ # global variable foo

funct_1 () {
    local foo # variable foo local to funct_1
    foo=1

    echo "funct_1: foo = $foo"
}

funct_2 () {
    local foo # variable foo local to funct_2
    foo=2
    echo "funct_2: foo = $foo"
}

echo "global: foo = $foo"
```

What would happen if we removed `local` ?

```
#!/bin/bash

foo=@ # global variable foo
funct_1 () {
    foo=1
    echo "funct_1: foo = $foo"
}

funct_2 () {
    foo=2
    echo "funct_2: foo = $foo"
}

echo "global: foo = $foo"
funct_1
echo "global: foo = $foo"
funct_2
echo "global: foo $foo"
```

Parameters

Positional Parameters

Positional parameters are built-in parameters that allow our programs to get access to the contents of the command line.
This is extremely valuable when we are creating scripts and then want to pass a parameter through the script from the command line.

If our code has more than 9 positional parameters, you need to enclose the positional parameter in curly brackets `${10}`

Let's create a script to see how this works:

```
#!/bin/bash

echo "
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
```

In the example, you may notice that we haven't given `$0` any specific value.
Let's try to run the script a couple ways through the command line to see what this means:

1. Run the script with arguments `a` `b` `c` `d`.
2. Run the script with any arguments of your choice.

What do we notice?

$* and $@

`$*` —> Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands into a double-quoted string containing all of the positional parameters, each separated by the first character of the IFS shell variable (by default a space character).

`$@` —> Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word surrounded by double quotes.

Let's take a look at this code piece by piece:

```
print_params () {
    echo "\\$1 = $1"
    echo "\\$2 = $2"
    echo "\\$3 = $3"
    echo "\\$4 = $4"
}
pass_params () {
    echo -e "\\n" '$* :'3 print_params $x
    echo -e "\\n" '"$x" :'s print_params '$x"
    echo -e "\\n" '$@ :'; print_params $@
    echo -e "\\n" '"$@" :'; print_params '$@"'
}
pass_params "word" "words with spaces"
```

1. Here we have two functions: `print_params ()` and `pass_params ()`. `pass_params ()` calls on the function `print_params ()` within its function.
2. In the first function, `echo` is printing the line inside the double quotes. The `\` infront of `$1` escapes the `$`, thus losing its meaning, as we learned earlier.

```
print_params () {
    echo "\\$1 = $1"
    echo "\\$2 = $2"
    echo "\\$3 = $3"
    echo "\\$4 = $4"
}
```

3. In the second function, `echo` again is printing the line inside the single quotes. `"\n"` is adding a tab at the beginning of the line for readability. It is then calling on the first function (`print_params ()`) with the argument `$*` . The second echo is calling the first function but with the argument `$*` in double quotes. This is repeated for `$@`

```
pass_params () {
    echo -e "\\n" '$x :'; print_params $x
    echo -e "\\n" '"$x" :'s print_params '$x"
    echo -e "\\n" '$@ :'; print_params $@
    echo -e "\\n" '"$@" :'; print_params '$@"'
}
```

4. In the final part of the code, we're calling on the `pass_params ()` function and passing two arguments: `"word"` and `"words with spaces"`.

```
pass_params "word" "words with spaces"
```

Let's see what happens's when we run the script in terminal. Remember, we don't have to pass any arguments in the command line because we have done so in our script.

Let's take a look at another example. In this example we'll get a greater understanding of variables and positional parameters:

```
function afunc {
    echo in function: $0 $1 $2
    var1="in function"
    echo var1: $var1
}
var1="outside function"

echo var1: $var1
echo $0: $1 $2
afunc funcarg1 funcarg2
echo var1: $var1
echo $0: $1 $2
```

Let's break it down again:

1. In our first function called `afunc` , using `echo` we will print `in function:` and pass 3 positional parameters. We will then define the variable `var1` and call it `"in function"` and print it using echo again.

```
function afunc {
    echo in function: $@ $1 $2
    var1="in function"
    echo var1: $var1
}
```

2. Outside of the function, we'll create another variable also named `var1` and give it the value of `"outside function"`

```
var1="outside function"
```

3. We'll then add the program.
    a) `echo` , we'll print `var1`
    b) Print 3 positional parameters
    c) Call the function with two arguments
    d) Print `var1` again
    e) Print 3 positional parameters again

```
echo var1: $var1
echo $0: $1 $2
afunc funcarg1 funcarg2
echo var1: $var1
echo $0: $1 $2
```

Let's run it in our terminal without any additional arguments and see what the output is.

- Why did `echo $@: $1 $2` only output one argument?
- Why did `var1` change the third time to `inside function` rather than `outside function` ?

Now let's change and add a few things to see what happens:

- In our terminal, what happens if we pass two arguments by entering `ascript.sh arg1 arg2` with `ascript.sh` being the name of our script and `arg1 arg2` being two random arguments?
- What happens if we add `local` to our function?

Parameter Expansion

Let's discuss the difference between `$a` and `${a}`

`$a` on its own is fine, but when placed next to another string, it can confuse the shell. For example:

- `$a_file` the shell will try to expand a variable named `a_file` rather than `a`
- `${a}_file` the shell will now try to expand the variable `a`

This can help us be more flexible when navigating and manipulating files and directories.

Let's look at the code below to see how this helps us:

```
$ filename="myfile"
$ touch $filename
$ mv $filename ${filename}1
```

This block of code creates a file based on our defined variable and then renames it with the same variable but with an additional component.

Parameter expansion also help us if our variables are unset (i.e. do not exist) or are empty. Let's take a look at a couple examples in the next few slides.

1. `${parameter:=x}` If parameter is unset or empty, expansion results in the value of `x` and the value of `x` is assigned to the parameter. If it's not empty, it results in the value of the parameter

```
$ foo=
$ echo ${foo:="Something else"}
$ echo $foo
$ foo=bar
$ echo ${foo:="something else"}
$ echo $foo
```

Through this sequence of commands we can see that when `$foo` is empty, `:-` fills the variable with `"something else"`. Once we define the variable, `:-` results in our defined variable.

2. `${parameter:=x}` If parameter is unset or empty, expansion results in the value of x and the value of x is assigned to the parameter. If it's not empty, it results in the value of the parameter

```
$ foo=
$ echo ${foo:="something else"}
$ echo $foo
$ foo=bar
$ echo ${foo:="Something else"}
$ echo $foo
```

We can see that when `$foo` is empty, `:=` assigns the variable with `"something else"`. If we define the variable again, `:-` results in our second defined variable.

3. `${parameter:?x}` If parameter is unset or empty, this expansion causes the script to exit with an error, and the contents of `x` are sent to standard error. If parameter is not empty, the expansion results in the value of parameter.

```
$ foo=
$ echo ${foo:?"something else"}
$ echo $?
$ foo=bar
$ echo ${foo:?"something else"}
$ echo $?
```

We can see that when `$foo` is empty, `:?` gives us an error which we can see as `echo $?` outputs `1`. If we define the variable again, `:?` results in the value of our variable.

4. `${parameter:+x}` If parameter is unset or empty, the expansion results in nothing. If parameter is not empty, the value of `x` is substituted for parameter; however, the value of parameter is not changed.

```
$ foo=
$ echo ${foo:+"something else"}
$ echo $foo
$ foo=bar
$ echo ${foo:+"something else"}
$ echo $foo
```

Here, `:+` resulted in an empty output and the value of `$foo` remains empty. If we define the variable, `:+` will still output what we defined, but it will not reassign the variable permanently.

String Operators

String operators are extremely valuable for operations on pathnames. They can help extract parts of pathnames, especially if they follow a pattern. Many pathnames typically follow patterns, such as all extensions are preceded with `.` .
Some character expansions are:

1. `${#parameter}`
2. `${parameter:offset}`
3. `${parameter:offset: Length}`
4. `${#parameter}` expands into the length of the string contained by the parameter.

```
$ foo="Toronto needs more trees"
$ echo "'$foo' is ${#foo} characters long."
```

With the following expansions, we can extract a portion the string contained by the parameter.

2. `${parameter:offset}` will extract characters from *offset* characters to the end of the string. For example, counting from the beginning of the string, the *n* of *needs* is 8 characters from the beginning. Because did not specify an end, `echo` will print from *needs* onwards.

```
$ foo="Toronto needs more trees"
¢ echo ${foo:8}
```

3. `${parameter:offset: length}` will specify the length that we want to extract. This length is counted not from the beginning of the string, but from the offset of the string.

```
$ foo="Toronto needs more trees"
$ echo ${foo:8:5}
```

We can see that from the beginning of the string, *n* is 8 characters in, and from *n*, s of *needs* is the 5th character from *n*. Therefore, our ouput will be *needs*.

Let's now see how to use patterns in our parameter expansions. There are several ways we can achieve this:

1. `${parameter#pattern}`
2. `${parameter##pattern}`
3. `${parameterxpattern}`
4. `${parameterxspattern}`

1. `${parameter#pattern}` removes the shortest leading portion of the string contained in *parameter* defined by the *pattern*.

```
$ foo=/User/name/Desktop/file.txt.zip
$ echo ${foo#/x/}
```

In this example, we've defined foo as a file with an extension.

The expansion matches any (`*`) pattern of `/*/` and returns the shortest leading portion.

2. `${parameter##pattern}` is very similar to the previous expansion except it removes the longest leading portion of the string.

```
$ foo=/User/name/Desktop/file.txt.zip
$ echo ${foo##/x/}
```

Very similar to the previous example, the expansion matches any (`*`) pattern of `/*/` and returns the longest leading portion.

3. `${parameter%spattern}` removes the shortest ending portion of the string rather than the beginning.

```
$ foo=/User/name/Desktop/file.txt.zip
$ echo ${fo00%.x}
```

4. `${parameter%spattern}` removes the longest ending portion of the string.

```
$ foo=/User/name/Desktop/file.txt.zip
$ echo ${fo00%%.\*}
```

What happens if we change our pattern to `#*_` ?

Let's pretend a file named "rachaels_file" and we want to know its extension. How would we do that?

What if our file was name "rachaels file"

We can also use expansions to replace the contents of the parameter with a string based on the pattern.

1. `${parameter/pattern/string}` replaces only the first occurence of pattern.
2. `${parameter//pattern/string}` replaces all occurances.
3. `${parameter/#pattern/string}` requires the match to occur at the beginning of the

```
$ echo ${foo//MP3/mp3}
```

```
$ echo ${foo/#MP3/mp3}
```

```
$ echo ${foo/%MP3/mp3}
```

Can you think of when this might be helpful?

Let's say I have a a named "rachaels cool file". I want to rename them because spaces cause problems in filenames. How would I do this?

Arithmetic Assignment

We have seen assignment before with examples such as `foo=5` . This is a simple assignment but we can also add complexity to this assignment with other operators.

- `$((parameter += x))` assigns the parameter to itself `+` x
- `$((parameter -= x))` assigns the parameter to itself `-` x
- `$((parameter *= x))` assigns the parameter to itself `*` x
- `$((parameter /= x))` assigns the parameter to itself `/` x

We can also increase or decrease our parameters by one.

- `$((parameter++))` increases parameter by one after the parameter is retruned
- `$((parameter--))` decreases the parameter by one after the parameter is returned
- `$((++parameter))` increases parameter by one before the parameter is returned
- `$((--parameter))` decreases parameter by one before the parameter is returned.

These are very subtle changes so let's see what we mean after and before a parameter is returned:

```
$ foo=1
$ echo $((foo++))
$ echo $foo
```

```
$ foo=1
$ echo $((++foo))
$ echo $foo
```

Command Substitution

So far we've learned how to get values into variables by using assignment statements (`x=5`) and positional parameters (`x=$1`). Another way is command substitution which allows you to use the standard output of the command as if it were a variable.

Let's say we want to assign a variable to the output of a command so that we can apply another command to that output. In this particular case, we want to make a variable equalall files beginning with *t*. We then want to apply a sort command on that variable:

```
$ x=$(find tx)
$ echo $x | sort
```

Although this seems quite simple now, we'll see how this can be extremely powerful when we move into flow control.

Flow Control

Flow control allows programs to "change directions" based on the results from a given input.

Bash supports several constructs:

- `if/else`
- `while` / `until`
- `case`
- `for`

if / else

`if/else` is a conditional statement that chooses whether or not to do something based on a true or false statement.

```
if condition; then
  commands
[elseif condition; then
  commands... ]
[else
  commands ]
fi
```

Here, we've assigned `x` to the value `5`. We've then written an `if/else` statement that asks if `x` is equal to `5` than tell us that `x` equals `5`. Otherwise (`else`), tell us that `x` does not equal `5`.

```
x=5
if [ $x = 5 ]; then
  echo "x equals 5."
else
  echo "x does not equal 5."
fi
```

Let's take a look at a more practical example: we want to know if there are any files in our directory that contain spaces.

```
#!/bin/bash
cd ~/Desktop/dir1
if [[ -n $(find . -type f | grep " ") ]]; then
  echo "A file contains a space"
else
  echo "No files contain a space"
fi
```

First we've changed our working directory to *dir1*:

```
cd ~/Desktop/dir1
```

We then utilized command substitutions that we've just learned by storing the output of files that contain a space. The `-n` option checks if the length of a string is *nonzero*:

```
-n $(find . -type f | grep " ")
```

By wrapping our output in an if statement, we're stating:

1. `if` the value of `$(find . -type f | grep " ")` is nonzero, then print (`echo`) "A file contains a space"
2. Otherwise (`else`), print (`echo`) `"No files contain a space"`

Control Operators

Control operators (`&&` and `||`) allow you to test more than one thing at a time. Their syntax is:

```
if command1 && command2; then
  ...
fi
```

```
if command1 || command2; then
  ...
fi
```

With the `&&` operator, command1 is executed and command2 is executed only if command1 is **successful**.
With the `||` operator, command1 is executed and command2 is executed only if command1 is *unsuccessful*.

Example of `&&`

```
filename=$1
word1=$2
word2=$3

if grep $word1 $filename && grep $word2 $filename; then
    echo "$word1 and $word2 are both in $filename."
fi
```

Using positional parameters that we learned earlier, what do you think will happen if we run the previous code?

- What happens if both words exist?
- What happens if only one word exists?
- What happens if no words exist?

Example of `||`

```
filename=$1
word1=$2
word2=$3

if grep $word1 $filename || grep $word2 $filename; then
  echo "$word1 or $word2 is in $filename."
fi
```

Similarly, what will happen if...

- What happens if both words exist?
- What happens if only one word exists?
- What happens if no words exist?

While

Using the while command, let's discuss looping. Looping allows portions of a program to repeat as long as the condition is false.

This syntax is:

```
while condition; do
    commands
done
```

Let's make a basic while script that displays five numbers in sequential order from 1 to 5 and then tells us when it's finished.

```
#!/bin/bash

# script called while-count.sh

count=1

while [ $count -le 5 ]; do
    echo $count
    count=$((count +1))
done
echo "Finished."
```

Why does the loop end?

While loops are extremely helpful to read lines of a file and then perform some command if a line meets a certain condition. Let's explore how to read lines first:

```
file=file1
while read -r line; do
    echo $line
done < "$file"
```

In this script, we're creating a variable with our file. We're then reading the file until the last line is read. In this example, we're using an input redirection that we learned earlier (`<`), which passes the file into the read command. We've also used `-r` so that any backslashes are escaped.

Because line is acting as a variable, we can also nest another loop if `$file` meets a condition. Let's say we have a file and we want to know every line that has `bananas` in it. How would we combine the while loop with an if statement?

```
while read -r line; do
    if [[ $line == *"bananas"* ]]; then
        echo $line
    fi
done < "$file"
```

Here we're reading the file line by line using the `while` loop. We're then saying `if` our variable, `$line` equals `"banana"` , then print the `$line` .

1. Why have we added the wildcard `*` ?
2. What would happen if we didn't include `*` ?

Until

Until loops are similar to while, except unlike while loops that run as long as the condition is true, the until loop will run as long as the condition is **false**

```
until condition; do
    commands
done
```

Let's create a script similar to the while statement: a basic while script that displays five numbers in sequential order from 1 to 5 and then tells us when it's finished.

```
count=1

until [ $count -gt 5 ]; do
    echo $count
    count=$((count +1) )
done
echo "Finished."
```

How is this script different to the while loop?

How might this be useful? Let's say we want to create 3 directories labeled *dir1*, *dir2* and *dir3*:

```
x=1
until [[ $x == 4 ]]; do
    echo "Creating dir$x..."
    mkdir dirs~x
    ((x++) )
done
```

Here we've created a variable `x=1` because we want our first directory to be *dir1*. We're then saying up until `x=4`, make a directory `mkdir` called *dir* plus our variable. We've then added 1 to `x` each iteration using an arithmetic assignment. The `echo` part is just to give us some feedback on what is happening behind the scenes.

for

For our final flow control, we're going to learn a powerful loop called `for`. The syntax is:

```
for variable [in words]; do
    commands
done
```

What we might notice is that this flow uses variables that will increment during the execution of the loop.

How would we use `for` if we wanted to list all files and directories in a folder?

```
for i in $(find x«); do
    echo $i
done
```

The variable `i` becomes all instances of the variable `$(find *)` . For each instance of `i` , we are then printing it. Although this seems quite basic and there more simple ways to list all files and directories (`ls`), this enables us to do many things with the looped variable `i` by nesting other loops.

What other ways can we use for loops?
What other ways can we use for loops within files?

Questions?

- Why do we use `i`?

Next Week: Git and Github

- Please make sure to come with a GitHub account

Additional Material

Exit Status

Commands issue a value to the system when they terminate, which is an integer in the range of O and 255 indicating the success or failure of a command's execution.

Conventionally, zero indicates success and any other value indicates failure.

Let's list a file that we know exists on our desktop:

```
$ ls -d /usr/bin
$ echo $?
```

`-d` is an option that returns the file if it exists and is a directory.

`$?` returns the value of the last executed command. The value being either zero for success or any other number for failure.

If we then list a file that we know does not exist in our desktop and return the value of `$?`, what do we expect to happen?

```
$ ls -d /bin/usr
$ echo $?
```

Exit Command

The `exit` command ina script replaces the return command and accepts a single, optional argument, which becomes the scripts exit status.

When no argument is passed, it defaults to zero. This enables our scripts to indicate an error.

If the script is a function in a larger program, we can use `return` instead of `exit` with a single, optional argument, allowing our function to indicate an error.

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then
    if [ -f "$FILE" ]; then
        echo "$FILE 1sva regular file."
    fi
    if [ -d "$FILE" ]; then
        echo "$FILE is a directory."
    fi
else
    echo "$FILE does not exist"
    exit 1
fi

exit

test_file () {
    # test-file: Evaluate the status of a file

    FILE=~/.bashrc

    if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
        fi
        if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
        fi
    else
        echo "$FILE does not exist"
        return 1
    fi
}
```

`if / else` statements are most frequently used with `test`

`test` performs a variety of checks and comparisons

Its syntax is:

```
`test expression`

or

`[ expression ]`
```

There are many expressions that are used to evaluate the status of files. Some important **File Expressions** include:

Expression	Is True If
-e file	file exists
-d file	file exists and is a directory
-f file	file exists and is a regular file
-r file	file exists and is readable (has readable permissions for the effective user)
-s file	file exists and has a length greater than zero

String Expressions

Expression	Is True If
string	string is not null
-n string	the length of string is greater than zero
-z string	the length of string is zero
string1 == string2	string1 equals string2
string1 != string2	string1 and string2 are not equal

Breaking Out Of A Loop

Bash has two build-in commands that can be used to control program flow inside loops.

- `break` command immediately terminates a loop and resumes with the next statement following the loop
- `continue` command skips the remainder the loop that is not needed (ie. a condition has been met) and resumes with the next iteration of the loop. `continue` allows for a more efficient execution

```
if condition; then
    if condition; then
        commands
        continue
    fi

    if condition; then
        commands
        continue
    fi
else condition; then
    command
fi
```

If the first `if` condition is met, then the second one will be skipped and resumed with the next iteration.

```
if condition; then
    if condition; then
        commands
        continue
    fi
    if condition; then
        commands
        break
    fi
else condition; then
    command
fi

If the second `if` condition is met, then the break immediately
terminates the loop and resumes with the next statement.
```