Overview

Bash Scripts & Syntax

There are two approaches to start building sharable, reproducible bash commands.

1. **One-Liners**
    - One-liners are single commands or groups of commands written as one line. These are usually command pipelines or Bash commands separated by semicolons.
        - One-liners can wrap around in the terminal, but have no line breaks to separate the script over numerous lines and can be copied and pasted to the terminal.
        - Many developers retain a list of handy one-liners in a document for quick reference.

Here's an example of a one-liner that reports disk consumption by file type. Copy and paste this line into the terminal and press `return` to see it in action.

```
find . -type f -empty -prune -o -type f -printf "%s\t" -exec
file --brief --mime-type '{}' \; | awk 'BEGIN
{printf("%12s\t%12s\n","bytes","type")} {type=$2; a[type]+=$1}
END {for (i in a) printf("%12u\t%12s\n", a[i], i)|"sort -nr"}'
```

2. **Bash scripts**
    - Bash scripts are text files that contain one or more bash commands. A bash script can be run two ways:

    1. We can use the `bash` command and the script name to tell Bash to run the file's contents. We usually use the `.sh` filename extension for Bash scripts.

    2. We can create an **executable** script that can be run like a program.
        - We add a **shebang** line at the front of the script, to tell the shell which program to use when the script is run.
        - This line starts with `#!` followed by the path to the program running the script.
        - `#!/usr/bin/env bash`

Let's create our first bash script.

1. Create a new blank script called `newbash` by typing `nano newbash` in the terminal and pressing return.

2. Add the shebang line to `newbash`.

- `#!/usr/bin/env bash`

On the next lines, we will write our commands.
3. Type `echo "This is my first Bash script."`

3. Press `return` twice to enter a blank line.

4. To enter a comment using `#`, type `# This is my comment`.

> info
>
> ## ## Blank Lines & Comments
>
> We can add blank lines between commands or sections to make the script more readable.
>
> We can also use comments to leave notes, labels, or instructions for yourself or other programmers. These are ignored when a script is run.

5. Save this script with `^ + O` (control + O) and press `return`.

6. Exit nano with `^ + X`(control + X).

We're going to make this file an executable script, so we'll leave the file extension `.sh` off the end as we save.

Creating an Executable

To make this script executable, we can use the `chmod +x` command along with the filename in the terminal. The name of our file is `newbash` (without the `.sh` extension).

```
chmod +x newbash
```

We can now run this program in the terminal with `./` as long as we're working in the same directory where the file is located.

```
./newbash
```

If we put this file somewhere the shell can find it, we can use the script's name like a command. So, if a system has Bash installed, the script can execute, but many of the user's Bash settings won't be read when the script runs.

Bash scripts are useful in that they help us avoid typos, speed up work, and allow us to share Bash code with other people and computers.

Checkpoint

Convert the `newbash` script into an executable file.

echo

The `echo` command **prints information to standard output** by default. The results of `echo` can also be directed to different locations.

To use it, we type `echo` along with our desired output.

```
echo This is my output.
```

With `echo`, we can mix strings, variables, substitutions, and expansions into a single output. This gives us options for both programmatic and static output.

```
size=small; drinkChoice=Sprite
echo Can I have a $size fry and a $drinkChoice, please?
```

If we use characters that confuse Bash, we'll often get an error.

```
echo I (sometimes) prefer cola.
```

In our example above, the parenthesis confuse Bash because they normally indicate a specific construct within Bash. In this case, we want Bash to ignore what it already knows about these characters.

We'll have to use a **character escape** `\` before each character we want Bash to print without trying to interpret it.

```
echo I \(sometimes\) prefer cola.
```

Instead of using character escapes, we can also change Bash's behavior by using **quotations**.

Let's revisit our command substitution example.

```
echo "The kernel is $(uname -r)"
```

There are **three** things we have to be careful of when working with quotations:
1. **Double quotes**

- By wrapping text in double quotes, **Bash will attempt to comprehend substitutions, expansions, evaluations, variables, etc.** We generally use these when we mix literal text with other values.

```
echo "The (kernel) is $(uname -r)"
```

2. **Single quotes**
    - Often known as strong quotations, these denote there is content to be treated literally inside the quotes, so **Bash won't try to interpret it**.

```
echo 'The kernel is $(uname -r)'
```

3. **No quotes**
    - In this situation, **Bash will interpret the command as each part is encountered.**

```
echo The kernel is $(uname -r)
```

Using `echo` ends a line of text with a new line character by default. If we use `echo` with no text after, it will return an empty line.

```
echo; echo "I left some space for you!"; echo
```

The `-n` tag **removes the new line** from the `echo` command, letting us use several `echo` commands to build a single line of text. This can be useful for writing complex commands in a script.

```
echo -n "Out of the sea, "; echo -n "wish I could be "; echo
"part of that world."
```

Checkpoint

Match each term with its description.

Variables

Variables let us save, change, and use values by calling them by name. Bash variables are a kind of parameter expansion where we don't have to use braces.

.guides/img/variable

Variables, sometimes called **parameters**, are named using letter and number characters. We add a value to that name with an equal sign = and the value we want to store.

```
status=Passing
```

▼ **Note**
There should be **no space** on either side of the equal sign, otherwise we'll get an error. Variable names are also **case sensitive**. It's common to use lowercase variables as to not confuse them with environmental variables, which are almost always uppercase.

Explore the Bash script to the left, `variable.sh`, to explore variable declarations.

Click the button below to run the script.

Change the script to include your own variables and run the script to examine the output.

We can use `declare -p` in the terminal to get a list of all of the active variables in a session. Click the button below to see this in action.

This command gives us all of the active environmental variables as well as any user-declared variables.

Checkpoint

Modify the script `variable.sh` to include 3 variables:

- `firstName`
- `lastName`
- `myAge`

Set the variables to any value you like. In the script, include lines to display the variables using the `echo` command. The result of these commands should display the following:

```
The user's full name is firstName lastName.
They are myAge years old.
```

Fill in the variable names using the values of the variables.

Numbers and Arithmetic

Scripts often require doing some kind of math. In Bash, we can work with integers via arithmetic expansion and arithmetic evaluation. It's important to note that Bash only works with integer numbers, not decimals, floating points, or fractions.

Arithmetic expansion `$((...))` allows us to do math with literal numbers or variables, resulting in either something we can display or a variable.

There are six basic math operators in Bash. Let's try them out with parameter expansion in the terminal.

1. Addition `+`

```
echo $(( 7+2 ))
```

2. Subtraction `-`

```
echo $(( 72-24 ))
```

3. Multiplication `*`

```
echo $(( 12*29 ))
```

4. Division `/`

```
echo $(( 98/2 ))
```

5. Modulo `%`

```
echo $(( 10%3 ))
```

6. Exponentiation `**`

```
echo $(( 3**3 ))
```

We can also use expressions that are nested inside parenthesis for complex expressions.

```
echo $(( (9-5) + 23 * (12/3) - 13 ))
```

Arithmetic evaluation `((...))` changes the value of an existing variable. This is written without a dollar sign `$`.

Let's set a variable, `marbles` to the value `5`.

```
marbles=5
echo $marbles
```

We can increment and decrement the variable by 1 using `++` and `--`

```
((marbles++))
echo $marbles
```

```
((marbles--))
echo $marbles
```

We can add 5 to this value with the `+=` notation, indicating that the value should be equal to itself plus some other value.

```
((marbles+=5))
echo $marbles
```

This also works for subtraction.

```
((marbles-=3))
echo $marbles
```

Bash can only work with integer values. Calculations with decimal results return incorrect values.

Let's declare two variables as integers with the `-i` flag.

```
declare -i a=2
declare -i b=9
```

If we divide these numbers, bash will return only the integer portion of the result.

```
echo $(( a/b ))
```

The `$RANDOM` variable returns a pseudo-random number $0 - 32767$. Click the button below to generate random numbers with `$RANDOM`

We can create random integers within a smaller range, like 20, using the modulo `%` operator. It's common to add 1 to these operations to ensure the result doesn't return a 0 value.

```
echo $(( 1 + $RANDOM % 20 ))
```

Click the button below to generate random numbers between 1 - 20.

))

info

Note

If you need to do extensive math for a task, Bash may not be the appropriate scripting tool to use.

Checkpoint

Declare two variables, `num1` and `num2`, and perform the following operations in the terminal.

1. **`num1` plus `num2`**
2. **`num1` subtract `num2`**
3. **`num1` multiplied by `num2`**
4. **`num1` divided by `num2`**
5. **Increase `num1` by 7 using `+=`**
6. **Decrease `num2` by 15 using `-=`**
7. **`num1` modulo `num2`**
8. **`num1` to the `num2` power**

Formatting Output

The echo built-in `-e` allows you to use escaped characters or sequences to format the output.

Special characters like tab, new line, and the bell can be represented by escape sequences. They can also make the text and the terminal look different.

Let's make column headers with tabs, `\t`, between them.

```
echo -e "Student\t\tArea of Study"; echo -e "Rodney\t\tSystems
Administration"; echo -e "Arya\t\tDefense Against White Walkers"
```

Using the new line character, `\n` makes the terminal show the following characters on a new line on the screen.

```
echo -e "Winter\nIs\nComing"
```

Both of these are useful for arranging data in a user-friendly format.

`printf`

The `printf` built-in allows us to use placeholders while composing strings and formatting values, making our script and output more organized.

It's possible to use `echo` and `printf` to create the same output two different ways.

When using `echo`, we mix our expansions in with the string.

```
echo "The answers are: $(( 8 - 5)) and $(( 15 / 3 ))"
```

When using `printf` use a placeholder inside the string. At the end of the string, we append the code that represents those values in the same order as the placeholders appear in the string.

```
printf "The answers are: %d and %d\n" $(( 8 - 5)) $(( 15 / 3 ))
```

There are several placeholders that represent many different data types. The table below represents just a few options.

Placeholder	Data Type
`%d`	Decimal
`%s`	String
`%c`	ASCII Character
`%i`	Integer
`%Y`	Year
`%m`	Month
`%d`	Day
`%H`	Hour
`%M`	Minute
`%s`	Second

As you can see from the available data types above, we can also use `printf` to display and format dates and times.

```
printf "%(%Y-%m-%d %H:%M:%S)T\n"
```

Use the reference sheet to the left to explore different formatting options for `printf`.

Checkpoint

Match each command with its expected output.

Challenge 1

Using the script to the left, `systemReport.sh`, create a script to print a system report about your system.

- Use the `echo` command to print `System Report for _____`
  - Fill in the blank using the variable holding the host name.

The following system information should be presented as a tab-separated table including 2 tab characters in each line:

- A tab in front of the string
- A tab between the label and value
  #

  info

Hint

When using the `printf` command, the `%s` placeholder can be used to represent string values, such as system variables.

1. Use the `printf` command to print the following line.

- `Report Date: _____`
  - Fill in the blank using placeholders to format the date with the following string.
  - `"%(%Y-%m-%d)T"`

2. Use the `printf` command to print the following line.

- `Bash Version: _____`
- Fill in the blank using the variable holding the current version of Bash.

3. Use the `printf` command to print the following line.

- `Kernel Release: _____`
- Fill in the blank using command substitution and the `uname -r` command

4. Use the `printf` command to print the following line.

- `Available Storage: _____`
- Fill in the blank using the `df -h` command to print the amount of free disk space available.

5. Use the `printf` command to print the following line.

- `Available Memory: _____`
- Fill in the blank using the `free -h` command to display the amount of free memory available.

6. Use the `printf` command to print the following line.

- `Files in Directory: _____`
- Fill in the blank using command substitution and the `ls` command piped into the `wc -l` command.

Click the button below to test your script.

When you run your program, your system report should look similar to the output below.

```
System Report for desert-henry
        Report Date:    2022-04-04
        Kernel Release: 5.4.0-1059-aws
        Bash Version:   4.4.20(1)-release
        Available Storage:      4.3G
        Available Memory:       735M
        Files in Directory:     4
```

Comparing Values

Scripts frequently need to compare and test values. **Test** is a built-in in Bash, represented by **single brackets** `[...]`.

You can check the Test help pages any time by typing `help test` in the terminal and pressing `return`.Click the button below to take a look at the help pages for the `test` built-in.

Enter the `clear` command in the terminal to give yourself a clean workspace.

Let's test whether the home directory is actually a directory with the following command.

```
[ -d ~ ]
```

info

Remember

The tilde ~ represents the user's home directory.

We can use the `$?` variable to read the result of the comparison.

```
echo $?
```

Booleans

When we do a comparison or test, we receive a return status of `0` for success, or `1` for failure. We can treat the expression like a true or false value. This is useful when using flow control structures like if and while loops to combine tests and comparisons.

In this case, the test returned a `0` for success, so it is true that the home directory is, indeed, a directory.

Now, let's test to see if `variable.sh` is a directory.

```
[ -d variable.sh ]; echo $?
```

This test returned a `1` for failure, so it is not true that `variable.sh` is a directory. This is, of course, because it is a file.

String operators test and compare text string properties like whether they're empty, equal, or otherwise.

Let's test if "green" is the same as "blue".

```
[ "green" = "blue" ]; echo $?
```

This returns a `1` for false. These are not the same.

Now test if "green" is the same as "green".

```
[ "green" = "green" ]; echo $?
```

This returns a `0` for true. Here, "green" is definitely the same as "green".

Rather than using greater than `>`, less than `<`, or equal signs `=` to compare integers, Test uses them to compare texts. For example, a string can be "less than" another if it comes first alphabetically.

Test has it's own unique expressions for comparing numerical values.

Flag	Meaning
`-eq`	Equal
`-ne`	Not Equal
`-lt`	Less Than
`-le`	Less Than or Equal
`-gt`	Greater Than
`-ge`	Greater Than or Equal

These expressions return a binary result.
- `0` - True
- `1` - False

Let's test to see if 7 is equal to 15.

```
[ 7 -eq 15 ]; echo $?
```

This statement returns `1` for False.

We can use an exclamation point `!` to invert the expression. This is like asking for the opposite of the expression requested.

Let's invert the previous expression to test if 7 is NOT equal to 15,.

```
[ ! 7 -eq 15 ]; echo $?
```

This returns `0` for True, 7 is not equal to 15.

Remember to keep space between the comparison values and operator expression. Otherwise, we'll receive an error.

```
[7-eq15]; echo $?
```

Checkpoint

Which of the following command lines would return True (0)?

▼ Hint

If you are stuck, test each command in the terminal.

Extended Test

Similar to test, Bash also supports the **extended test**, represented by **double brackets** `[[...]]`. We get the same features as test, plus a few extras. Extended tests let us to use many expressions within a test and check for more complicated logic.

AND `&&`

We can test to see if multiple statements are true using the logical **AND** `&&` operator. **All of the statements being tested must be true for the extended test to also be true.**

Let's test to see if:
- The home directory, `~`, is a directory ,`-d`

AND `&&`

- The file `/workspace/variable.sh` does exist, `-a`.

```
[[ -d ~ && -a ~/workspace/variable.sh ]]; echo $?
```

Both of these statements are true, so the extended test returns a `0`.

If one of the test statements are false, the entire extended test is false.

```
[[ -d ~ && -a ~/workspace/excalibur.sh ]]; echo $?
```

Even though the first statement for the home directory is true, there is no file `excalibur.sh`, so the second statement is false. This makes the entire extended test return a `1` for false.

OR `||`

We can test to see if at least one of multiple statements are true using the logical **OR** `&&` operator. **Only one of the statements being tested must be true for the extended test to also be true.**

If we test to see if:
- The home directory, `~`, is a directory ,`-d`

OR `||`

- The file `/workspace/excalibur.sh` does exist, `-a`.

```
[[ -d ~ || -a ~/workspace/excalibur.sh ]]; echo $?
```

Since one of these statements is true, the entire extended test will return true, `0`.

These logical operators can also be used outside of extended test brackets to run commands under certain conditions.

In the example below, if the extended test is successful, then the command following the `&&` will be run.

```
[[ -a ~/workspace/variable.sh ]] && echo /workspace/variable.sh
does exist.
```

If the extended test fails, Bash will not go on to run the following command.

```
[[ -a ~/workspace/excalibur.sh ]] && echo
/workspace/excalibur.sh does exist.
```

Because commands also return a success or fail statement, these operators can also be used with system commands.

```
cd ~ && echo "You are now in the home directory"
```

These are commonly used to display confirmation messages.

```
ls && echo "I have listed the contents of this directory."
```

We also have access to two built-ins, true and false, that generate a success or failure code.

Click the button below to run: `true && echo "It is known!"`

Click the button below to run: `false && echo "That's pure fiction!"`

With extended test, we can use regular expression matching to try and match a given string. To do this, we:

- Provide an extended test with a string to match
- Type `=~`
- Provide a regular expression to compare the string to.

Let's test to see if the word `caramel` matches the expression `c.*`.

```
[[ "caramel" =~ c.* ]]; echo $?
```

Now, let's try the same test with the word `vanilla` matching the expression `c.*`.

```
[[ "vanilla" =~ c.* ]]; echo $?
```

This does not match, so it returns `1` for false.

> info
>
> ## The `c.*` expression
>
> This is an expression meaning "The character **'c'**, followed by any number of any type of characters".
>
> We'll learn all about these expressions in an upcoming lesson.

Because tests and comparisons are so common in Bash scripting, you should practice them often.

It's also a good idea to use the extended test regularly in your scripts rather than switching between it and the normal test. This helps keep us consistent and allows you to access more features.

If your script needs to be able to work on other shells, you'll want to use normal tests instead.

Checkpoint

Which of the following exoressions correctly uses extended test notation?

Arrays

Arrays let us to store and retrieve multiple pieces of related data quickly. Bash supports two types of arrays:

1. Indexed Arrays

Indexed arrays are used to set or read data based on their position in a list.

An indexed array is **implicitly defined** by enclosing a list of values in parenthesis and giving the set a name. Let's implicitly define an array called `instruments` with the following elements:
- piano
- flute
- guitar
- oboe

```
instruments=("piano" "flute" "guitar" "oboe")
```

We can alternatively use `declare -a` followed by the array name to **explicitly define** it.

```
declare -a instruments=("piano" "flute" "guitar" "oboe")
```

Unlike many other languages, there are **no commas** between items.

To get an element from the array, use the array's zero-based index notation `[..]` and the variable name inside curly braces.

```
echo ${instruments[1]}
```

This returns `flute`, which is in the second place, but because our indices start at `0`, this item is at index `1`.

.guides/img/array

Array values can also be set by index. Let's insert `trumpet` at the $6th$ index in our `instruments` array.

```
instruments[6]="trumpet"
```

Every index in an array doesn't need to be filled in order to work. These empty indexes can be useful to record data at a certain position in an array.

We can use the `+=` operator to append items to the end of an array. This, essentially, appends a piece of an array to an existing array.

```
instruments+=("tuba")
```

If we don't put the appending value in parenthesis, bash will append the value to the array's zero index, overwriting the first item in the list.

Let's take a look at all of the elements in our array using the `@` symbol.

```
echo ${instruments[@]}
```

This returns all of our populated items in the array, but it doesn't give us a good picture of which indices are empty.

Click the button below to loop through the array and display its contents.

We'll look into looping in detail in an upcoming lesson.
2. Associative Arrays

You can create associative arrays to specify two values instead of just one. We can do this by using `declare -A` followed by the array name.

Let's create associative array called `student`, and assign it some values.

```
declare -A student
student[name]=Rodney
student["area of study"]="Systems Administration"
```

To use or access a key with spaces, it **must be surrounded in quotes**.

We can access items in this array similar to the way we access indexed arrays, but by **using the key string instead of the index number**.

```
echo ${student[name]} is majoring in ${student["area of
study"]}.
```

info

Consider your Bash Version

When you're writing a script, keep in mind that associative arrays only supported with Bash 4 and above

Because associative and indexed arrays only allow one level or layer, **we can't make arrays inside of other arrays in bash.** If you need to represent data in a hierarchical array, you may need to transition from Bash to a more suitable programming language.

Arrays can help your script or hurt it. The same data can be saved as single variables in an indexed or associative array. What you do depends on how you want to model, think about, and manage your program's data.

Checkpoint

Fill in the blanks to complete the statement below.

Discovery 1

Test `[]`

Explore the manual page and try each of the following.

1. Check if the file `systemSummary.txt` exists and echo the result.

▼ **Solution**

The `-e` option returns True if a specified file exists.

```
[ -e systemSummary.txt ]; echo $?
```

2. Check if the file `thisisascript.txt` has been modified since it was last read and echo the results.

▼ **Solution**

The `-N` option returns True if a specified file has been modified since it was last read.

```
[ -N thisisascript.txt ]; echo $?
```

3. Check to see if the file `newbash` is newer than `variable.sh`.

▼ **Solution**

The `-nt` option returns True if **FILE1** is newer than **FILE2** according to their modification date.

```
[ newbash -nt variable.sh ]; echo $?
```

3. Check to see if the string `agglutinogens` comes before `agglutination` alphabetically and return the result.

▼ **Solution**

The `<` option returns True if **STRING1** sorts before **STRING2** lexicographically. This is exactly what we're looking for.

```
[ agglutinogens < agglutination ]; echo $?
```

4. Check to see if the file `systemReport.sh` exists and if it's empty.

▼ **Solution**

The `-e` and `-z` options let us check if a file exists, as well as if it is empty. We can use the `&&o` operator to test these as the same

```
[[ -e systemReport.sh && -z systemReport.sh ]]; echo $?
```

Because the file is not empty, the 2nd test statement fails and the test returns `1` for false.

Discovery 2

Arrays ( )

Explore the help page to the left and try each of the following.

1. Assign an indexed array called `tempo` containing the values below:

- Lento
- Largo
- Adagio
- Andante
- Moderato
- Vivace
- Presto

▼

Solution

We can do this a few ways. We can declare the array first, then populate each index explicitly.

```
declare -a tempo
  tempo[0]=Lento
  tempo[1]=Largo
  tempo[2]=Adagio
  ...
```

We can also assign items to the array using compound assignments.

```
tempo=(Lento, Largo, Adagio, Andante, Moderato, Vivace,
Presto)
```

2. Display all of the elements of `tempo` quoted separately, then display the elements quoted as a single string.

▼

Solution

We can display all elements of the array quoted separately using `@`

```
echo ${tempo[@]}
```

The `*` character displays the elements as a single quote.

```
echo ${tempo[*]}
```

3. Display indices 2 through 5 of the `tempo` array.

▼

Solution

We can print a range of indices using the format below.

```
echo ${tempo[@]:2:5}
```

4. Declare an associative array called `BPM` containing the following string indices and values.

- Lento : 40
- Largo : 45
- Adagio : 55
- Andante : 75
- Moderato : 95
- Vivace : 135
- Presto : 175

▼

Solution

We can use `declare -A` to declare an associative array.

```
declare -A BPM
```

We can populate each index individually...

```
BPM[Lento]=40
  BPM[Largo]=45
  BPM[Adagio]=55
  ...
```

... or we can assign items using compound assignments.

```
BPM=( [Lento]=40 [Largo]=45 [Adagio]=55 [Andante]=75,
[Moderato]=95, [Vivace]=135, [Presto]=175 )
```

5. Add the following string:value pair to the beginning of the `BPM` array.

- Grave : 35

▼

Solution

We can add this item to the beginning of the array using the following syntax.

```
BPM=({[Grave]=35 ${BPM[@]})
```