Overview

Learners will be able to...

- **Create a script that makes conditional decisions using if statements.**
- **Create a script that makes conditional decisions using case statements.**
- **Repeat a set of commands using the following types of loops:**
  - **For Loops**
  - **While Loops**
  - **Until Loops**
- **Read a file line by line using the `read` keyword.**
- **Read a file line by line using loops.**
- **Reuse a set of commands within a function.**

info

Make Sure You Know

This assignment assumes the learner has a basic understanding of Bash concepts and working in the terminal.

Limitations

This assignment is an overview of important scripting concepts that learners should be familiar with before attempting upcoming assignments.

If Statements

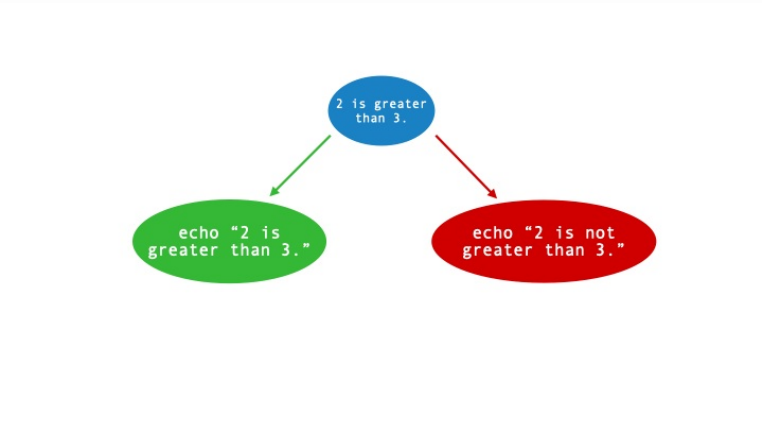
Using control structures in scripts allows us to control and change how script commands are executed based on conditions we specify.

An **IF statement** is a control that runs code based on an expression's truth value.

In Bash, IF is followed by an expression returning a true or false value. Next, write the keyword 'then'. Followed by the script lines to run if the condition is true.

```
if ...
then
  ...
fi
```

We can either end the if statement with `fi` or continue with an **else statement** that has a script to run if the condition is false.



.guides/img/if

The expression itself is a test.

There are many types of tests, including **extended tests** (using double brackets) and **arithmetic evaluations** (using double parenthesis). The **IF** statement can also use commands since they always return with a zero or non-zero status, interpreted as true or false.

Let's see how this works in a script.

```
declare -i a=3

if [[$a -gt 4 ]]; then
        echo "$a is greater than 4."
else
        echo "$a is not greater than 4."
fi
```

Let's look at each piece of this statement

- We start by declaring an integer, `a = 3`.
- Then we begin the IF statement with `if`
- The IF statement checks the condition of the variable. Here, we used the extended test indicated by the double brackets `[[]]` to test if `a > 4`.
- We uses the `then` keyword to represent the beginning of our commands that will run if the condition is true
- We, then, used the `then` keyword to start the commands that will run if the condition is false.
- We complete the statement with `fi`.

If the condition is true, **then** our terminal will display `$a is greater than 4.`

If it is false, **then** it will display, `$a is not greater than 4.`

Use the script to the left to explore the code segment below.

```
declare -i a=3

if [[ $a -gt 4 ]]; then
        echo "$a is greater than 4."
else
        echo "$a is not greater than 4."
fi
```

Click the button below to run the script in the terminal.

By running this script, we conduct a test to see if our variable is greater than or less than four. Using the else statement, we were able to call different echos based on the input variable.

Because we are comparing numbers, we could also use an arithmetic test indicated by two parenthesis.

```
declare -i a=3

if [[ $a -gt 4 ]]; then
        echo "$a is greater than 4."
else
        echo "$a is not greater than 4."
fi
```

The results would be the same.

This means there are different ways getting the same result. These can be chosen based on readability or for consistency within a script.

If a condition doesn't match the first expression, we can check it again using an **ELIF statement** (**else if**).

An **ELIF statement** can be placed between the `IF` and `ELSE` statements. When present the variable will be tested against both conditions. Only if both conditions fail will we receive the else condition.

This allows us to build decision trees based on a series of conditions rather than just a single yes or no.

We can also read if statements at the command prompt with colons between each command, but this becomes difficult to edit and think about, so consider moving your commands into a script. This is great for using branched logic in your Bash scripts.

Checkpoint

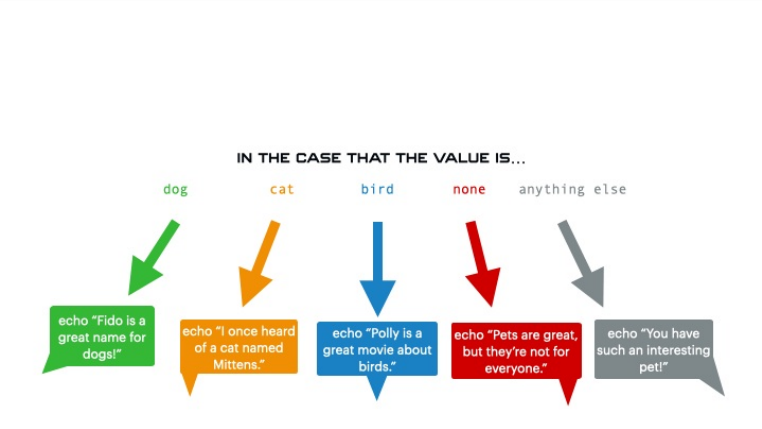
Fill in the blanks to complete the statements

Case Statement

The **case statement** is a control structure that allows you to write code that will run when a certain criteria is met.

The **case statement** is commonly used to simplify complex conditionals when there are multiple options. Using the **case statement** instead of nested if statements will make your bash scripts more readable and maintainable.

A case statement **must be given a list of values that could match our input** and scenarios to run. Once the input in received, **Bash runs the scenario that matches the input value**.



.guides/img/case

To write a case statement:
- Start with the term `case`, the **variable** you're testing, and the word `in`.
- Add your **test condition** on the next line.
- The value to test is followed by **two semi-colons** to tell Bash that we're done with that condition.

```
#!/bin/bash

echo -n "Enter the name of a country: "
read COUNTRY

echo -n "The official language of $COUNTRY is "

case $COUNTRY in

  Lithuania)
    echo -n "Lithuanian"
    ;;

  Romania | Moldova)
    echo -n "Romanian"
    ;;

  Italy | "San Marino" | Switzerland | "Vatican City")
    echo -n "Italian"
    ;;

  *)
    echo -n "unknown"
    ;;
esac
```

This statement checks is the country entered by the user and responds with the commands nested within that particular criteria.

If nothing matches the criteria, there's an asterisk for that.

You in the statement with the `esac` keyword. This is `case` reversed.

Case statements are important for menus and user input. They allow us to work with various circumstances more elegantly than a sequence of if statements. However, you may use if statements to replace case statements and vice versa. It all depends on your Bash script structure.

Checkpoint

Challenge 1

Use `nano` to create a script called `music.sh`.

- Include the Bash shebang line at the top: `#!/bin/bash`
- Write a command that presents the user with the prompt `Enter major or minor:`
- Use the `read` command to save the user's response in a variable called `scaleType`.
- Write an **if statement** to test the user's response in the `scaleType` variable.
  - If the `scaleType` is `major`
    - Then display the response `Major scales sound bright and hopeful`
  - Else, if the `scaleType` variable is `minor`
    - Display the response `Minor scales sound sad and mysterious`
  - If the user enters anything else
    - Display `I'm sorry, I don't understand`
- Write a command that presents the user with the prompt `What is your favorite genre of music?`
- Use the `read` command to save the user's response in a variable called `genre`.
- Write a case statement to display the following results:
  - If the user enters `pop`, display the message `You might enjoy Ariana Grande`
  - If the user enters `classical`, display the message `You might enjoy Vivaldi`
  - If the user enters `hip hop`, display the message `You might enjoy Drake`
  - If the user enters `dance`, display the message `You might enjoy UMEK`
  - If the user enters `country`, display the message `You might enjoy Jason Aldean`
  - If the user enters any other value, display the message `Great choice!`

Use the `Try It!` button to test your script in the terminal.

When your script works as described above submit your script below.

For Loops

A **For loop** runs code for each value in a list or range. During each loop iteration, a variable within the loop represents the current list item. The example below shows us the syntax for a **for loop**.

```
for i in LIST
do
    COMMANDS
done
```

The for loop has several pieces:
- `for`: This is the keyword to let Bash know a for loop is coming
- `i`: This value becomes a variable representing an item in the list. It can be named anything, but is often related to the list it's referring to.
- `in LIST` This can be read *"within the list or range"*. `LIST` identifies the items that `i` will *"loop"* through, or access item by item.
- `do`: This marks the beginning of the command section. Commands after this line will be executed once for every item `i` in the list `LIST`.
- `COMMAND`: This section includes the commands we'd like to run or perform for each item `i` in the list `LIST`
- `done`: This marks the end of the command section as well as the end of the loop.

Let's look at an example! Copy the code segment below and paste it into the terminal.

```
for i in 1 2 3 4 5
do
    echo $i
done
```

This loop will run 5 times with `i` having a **different value each time**.

For loop visual representation

We can also write a for loop on a single line as a one-liner.

```
for i in 1 2 3 4 5; do echo $i; done
```

Using explicit values separated by spaces is useful for looping through short lists, but if we want to cycle through 100 items, it can get tedious to say the least. We can use **brace expansion** to help with this task.

```
for i in {1..100}; do echo $i; done
```

For loops can also be evaluated arithmetically, by incrementing over the counter variable. During this process:

- The variable `i` starts at 1
- A test is performed to check if `i` is less than or equal to 100
  - If `i` < 100
    - The variable `i` is incremented or increased by 5
  - If `i` >= 100
    - The test fails and the loop stops.

Arrays can be looped through using parameter expansion.

Let's create an array called `favMusicals` and write a for loop that displays each array item's value to the terminal.

```
declare -a favMusicals=('Hamilton' 'The Lion King' 'Grease'
        'West Side Story' 'Rent')

for musical in "${favMusicals[@]}"
do
  echo "I love the musical: $musical"
done
```

Command substitution can also be useful for creating input for our for loop.

```
for i in $(ls)
do
    echo "The file $i is in the working directory"
done
```

This useful example does a few things:
- `for` each item `i`
- `in` the result of the `ls` command (a list of files in the current directory)
- Display the value if the item `$i` within a string

For loops are often used with the output of commands like `find` and `expand`, that give us a list of files that we may not know the path to at the time the script is written.

Checkpoint

While and Until Loops

Loops are controlled structures that repeat a piece of code until it ends. **While loops** and **until loops** are two types of conditional loops. The distinction is small, but depending on your script's goals, one may be better than the other.

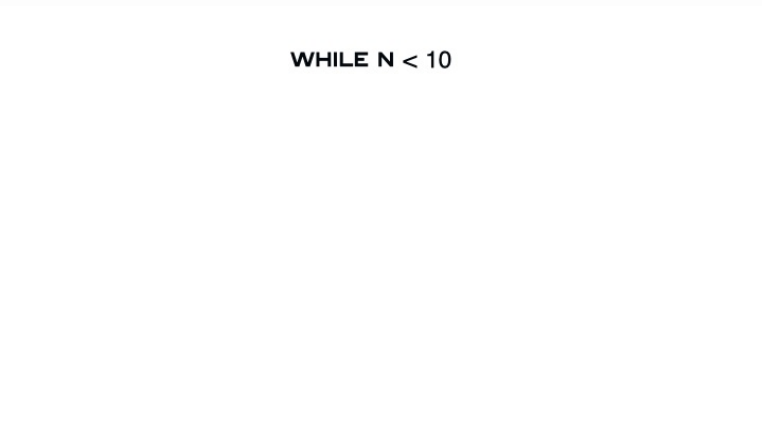
`while`

The **while loop** runs **while the condition is true** and stops when the condition is false. The while keyword precedes the condition that determines whether the loop should continue. This is followed by do and the code to loop over. Then done terminates the construct.

```
while...
do
  ...
done
```

Create an integer variable n and set it to 0. "While n is less than 10", this program will:

- Check if the current value of `n` is less than 10
- While this is true, the program will do the `echo "n is equal to: $n"` to print the current value of `n`
- Increase the value of `n` by 1 with the `n++` increment notation.
- When n is no longer less than 10, the loop will stops



.guides/img/while

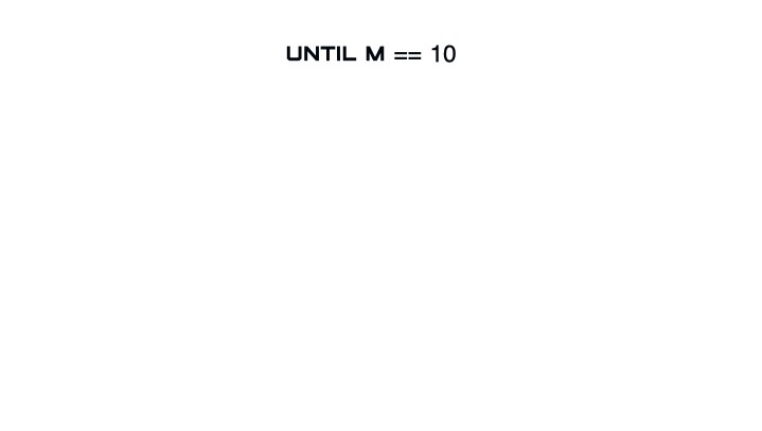
```
nano while

echo "While Loop"

declare -i n=0
while (( n < 10 ))
do
       echo "n is equal to: $n"
       (( n++ ))
done
```

until

The **until loop** runs **until its condition is true** and then quits.



.guides/img/until

```
echo -e "\nUntil Loop"

declare -i m=0
until (( m == 10 ))
do
        echo "m is equal to: $m"
        (( m++ ))
        sleep 1
done
```

`ctrl + o`

These loops are interchangeable if approached properly. Be aware of the possibility of infinite loops, where your condition is either always false, or always true. We can explore an example of an infinite loop below.

Use `nano` to create a new script called `myscript`.

```
nano myscript
```

Include the shebang line `#!/bin/bash` at the top and copy the following loop into the script.

```
until (( m > m ))
```

Save and run this program.

When we run this program, it just keeps looping because, at no point will `m` ever be less than itself. If we ever find ourselves stuck in an infinite loop, we can press `ctrl + c` while our script is running to break out and terminate the program.

Checkpoint

Fill in the Blanks to complete the statement below.

Reading and Writing Text Files

Our bash scripts will write to text files with output redirection using single and double greater than signs.

Using a single greater than sign `>` replaces existing data in the destination file with new data from the operation writing to it.

Double greater than signs `>>` keep existing data while new data is added to the target file.

When we read from files, we can use the input redirection operator `<` to pass a text file as input to the `read` keyword. Then, we can use a while loop to read the text file line by line, allowing us to use each line inside the while loop.

Let's create a script to save some text to a file called `hotpotato.txt` in the `/home/codio/workspace` directory.

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

for line in {1..5}
do
    echo "Hot Potato $line" >
/home/codio/workspace/hotpotato.txt
done
```

Click the `Try It!` button below to run this script.

When we display the contents of the file with `cat`, we see that we only get the line `Hot Potato 5`.

```
cat hotpotato.txt
```

This is because each time the for loop runs, the `>` output redirection overwrites the previous line with the new line, leaving us with only the most recent output. This can be useful for storing and changing various settings.

Remove this file from the terminal with `rm lines.txt` and we'll try again.

Let's revisit our `hotpotato.sh` script and change the operator from `>` to `>>`.

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

for line in {1..5}
do
    echo "Hot Potato $line" >> /home/codio/workspace/lines.txt
done
```

Click the `Try It!` button below to run this script.

Now, when we display the contents of the file with `cat`, we see that a new line has been appended to the existing file for each iteration of the loop. This can be very useful for creating system log files.

We can also use scripts to read lines from text files.

Let's use the input redirection operator `<` to pass `/home/codio/workspace/lines.txt` as input to the `read` keyword.

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

while read textline
do
    echo "The line I'm reading says: $textline"
done < /home/codio/workspace/lines.txt
```

In this example, we use a `while` loop to:
- Check if there is information a line of the file
- While this returns `true`, the loop:
- Saves the line of text in a variable called `textline`
- Echos the statement including the current `textline` variable.
- Moves on to the next line in the file
- When the line is empty, the `while` loop returns `false` and ends.

We can use each line in the file to do simple things like display, or more complicated things like text extraction and text transformation. Some scripts use a regular expression to search through mountains of log files for specific text.

Some programs use a text file as a *"configuration file"* to read and save settings or adjustments for later use by reading and writing to the file.

Checkpoint

Which of the following facilitates redirection that overwrites any existing data in the destination file?

Functions

As you write your script, you may find yourself repeating commands or writing similar commands with only minor variations. Avoiding repetition is good practice, and keeping scripts organized and free of duplicates makes them easier to maintain.

When writing code, we can use **functions** to create a series of commands once and then run it by name whenever we need the command series. This helps us structure our code.

To build a function, we need:
- the name of the function you want to build
- an open and closed parenthesis `()`, and
- braces to surround the function's commands.

```
fn(){
  ...
}
```

.guides/img/function

Functions must be defined before they can be used. Oftentimes, programmers will place all of their functions at the top of their scripts or in another file entirely to keep things in order.

Let's create a new script, `functions`, and define a function called `briefing`.

```
briefing() {
  echo "Good Morning, T'Challa!"
  echo "The weather today will be cloudy."
  echo "The first thing on your schedule is to check your
email."
}

echo "It's time to get ready for your day..."

briefing
```

Save and close your script and run it in the terminal using `bash functions`.

Your output should look like the following:

```
It's time to get ready for your day...
Good Morning, T'Challa!
The weather today will be cloudy.
```

important

Notice!

The echo statement, `It's time to get ready for your day...` arrives first as output. This is because the commands within the function `briefing` are not run until the function is called by name, after the echo statement.

Arguments

There will be times where we'll need to run the code with slightly different information as an input. We can use arguments to pass data to a function.

Let's open our `functions` file and make a few changes.

```
nano functions
```

Let's change the following in our script:

- Replace the name in the `Good Morning` statement with `$1`
- Replace the weather forecast `cloudy` with with `$2`

```
briefing() {
  echo "Good Morning, $1!"  # $1 represents the first variable
passed to the function
  echo "The weather today will be $2."  # $2 is the second
variable passed to the function
}

echo "It's time to get ready for your day..."

briefing Tony sunny. # Here, we call the briefing function,
passing Tony as variable 1 and sunny as variable 2
```

Other special variables can be used inside a function, like `$@` or `$funcname`. A dollar sign `$` indicates all of a function's arguments. This can helpful if you intend to loop through them from your script.

Working with functions often requires a variable that isn't accessible outside the function. Unless otherwise specified, all variables in Bash are **global**. The term **local variable** refers to a variable that **only exists within one function**. We use local variables instead of saved data to avoid issues where the variables are accidentally modified or misused.

Functions help us in organizing our code. They're fairly common in larger scripts, so knowing what they are and how they work is more than useful.

Checkpoint

Drag the code blocks into the area below to create a function called `counter`. The function should:

- Display the message `Hello, _____! Watch me count!
  - The blank will be filled with a function argument
- Run a for loop to display numbers $1 - 10$ each on its own line
- After closing the function, call the function with the argument `Tobias`

Discovery 1

Loops

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

1. 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 )
```

2. Write a `FOR` loop:

- For each item in the array `BPM`
  - Display `The BPM for tempo marking ITEM is VALUE`

▼

Solution

We can use the value inside $ITEM < /code >$ $to display the key or index of the array. We can display the value of that item by using <$ $code >$`{BPM1[$KEY]}`.

```
for KEY in "${!BPM[@]}"; do echo "Key: $KEY"; echo "Value:
${BPM[$KEY]}"; echo; done
```

3. Write a `WHILE` loop:

- Set a variable called `currentTempo` equal to `0`
- While `currentTempo` is less than `40`
  - Display `CURRENTTEMPO BPM is to slow to play`
  - Increase `currentTempo` by `5`

▼

Solution

Using the `while` keyword, we can use brackets `[]` to test a statement for truth and perform a command as long as the test is true.

```
currentTempo=0
while [ $currentTempo -le 40 ]
do
    echo "$currentTempo BPM is too slow to play"
    (( currentTempo+=5 ))
done
```

Discovery 2

Functions

Explore the help page to the upper-left and explore each of the following. Practice using the file to the lower-left, `scratch.sh`. Use the button below to run your practice script.

1. Write a function called `isWorking` that:

- Displays " Now running isWorking function"
- Returns `0`, indicating a successful run

▼

Solution

We can use the following syntax to create a function and place our desired commands inside. Then we can call the function and confirm the return code with `$?`

```
isWorking(){
    echo "Now running isWorking function"
    return 0
    }
```

isWorking
echo $?

2. Modify the `isWorking` function to:

- Accept a string as an argument
- Display `Did someone say ARGUMENT`
- return 0

▼

Solution

We can use the following syntax to create a function and place our desired commands inside. Then we can call the function and confirm the return code with `$?`

```
isWorking(){
    echo "Did someone say $1"
    return 0
    }
```

isWorking Banana
echo $?