1. Writing Your First Shell Script

1.1 Introduction to Shell Scripting

Hello! Welcome to your first step into the world of **shell scripting**. In this lesson, you will learn how to write a basic shell script. Shell scripts are powerful tools used to automate tasks on Unix-like systems.

A **shell script** is a file that contains a series of commands. Think of it as a way to tell your computer to perform multiple operations sequentially. By learning shell scripting, you can save time on routine tasks and even handle complex operations that would otherwise require a lot of manual effort.

Before diving into specifics, it's worth noting that bash (Bourne Again Shell) is one of the most popular shells used for scripting. Other popular alternatives include `zsh`, `sh`, and `ksh`, but for this course, we will focus on bash.

1.2 Creating and Running a Shell Script

Let's begin by creating a simple shell script that prints "Hello, World!" to the screen. This classic example serves as a foundation, helping you understand the script's structure.

Here's what the script looks like:

Bash

Copy to clipboard

```bash
#!/bin/bash

# Simple script to print "Hello, World!"
echo "Hello, World!"
```

Suppose this script is located in a file called `hello_world.sh`. Before running the script, you need to make it executable. To do this, navigate to the directory where the script is saved and run:

Plain text

Copy to clipboard

```
chmod +x hello_world.sh
```

`chmod` is the command used to change file modes or access permissions, and `+x` adds the execute permission to the file.

Finally, the command to run the script from the terminal is:

Plain text

Copy to clipboard

```
./hello_world.sh
```

Running the script above results in "Hello, World!" being output by the console. In the CodeSignal IDE, clicking the "Run" and "Submit" buttons automatically calls the `./solution.sh` command. This is how your solution files are run by Cosmo!

Now let's look at each line of the script in more detail.

1.3 Understanding The Shebang

The shebang (`#!/bin/bash`) is essential in shell scripts as it defines the interpreter to be used when running the script. It is the absolute path to the interpreter's executable file. While we use `/bin/bash` here, you could use `/bin/sh` for more basic scripts or `/bin/zsh` for enhanced scripting functionalities. Using the appropriate interpreter ensures your script runs even on different systems.

Adding Comments

Comments in bash are preceded by a `#`. Adding comments in your script is good practice. It helps you and others understand the purpose and functionality of your script. Comments can be added anywhere in the script and are ignored during execution.

For example:

Bash

Copy to clipboard

```bash
# This is a comment that explains the purpose of the script
echo "Hello, World!"  # This comment explains what this line does
```

The echo command

`echo` is a built-in command in bash and other shell environments that is used to display a line of text or a variable's value to the terminal. For example, `echo "Hello, World!"` will print `Hello, World!` to the screen. It is often used in scripts to provide informational messages, display the values of variables, or to output results to the terminal.

1.4 Summary and Next Steps

Congratulations on writing your first shell script! You've learned how to create a basic shell script, understand the shebang, add comments, and use `echo` to print text. These are the first steps toward mastering shell scripting.

Next, you will dive into more complex scripts that use variables, control structures, and functions. Get ready to create powerful automation scripts that can simplify your workflow and increase productivity.

Let's head to the practice section and reinforce your new skills with some hands-on exercises! Happy scripting!

2. Using Variables in Shell Scripts

2.1 Introduction to Using Variables in Shell Scripts

Welcome back to the world of shell scripting. In the previous lesson, you crafted your very first shell script that printed "Hello, World!" to the screen. Today, we will take the next step by exploring how to use **variables** in shell scripts. This lesson will introduce you to variable declaration, reassignment, and some basic arithmetic operations.

Using variables can make your scripts more dynamic and flexible, allowing you to store information, reuse values, and perform calculations. Let's get started!

2.2 Creating and Using Variables

Variables in shell scripts are used to store data that can be reused throughout the script. To define a variable in a shell script, you use the syntax:

Plain text

Copy to clipboard
```
variable_name=value
```

An important distinction of bash is there **cannot** be spaces before or after the `=` sign.

To access the value of a variable, use a `$` before the variable name.

Let's take a look at an example:
Bash
Copy to clipboard
```bash
#!/bin/bash

greeting="Hello"
name="World"
echo "$greeting, $name!"   # Prints: Hello, World!
```

- `greeting="Hello"`: This line creates a variable named `greeting` and sets its value to "Hello."
- `name="World"`: Another variable named `name` is created and assigned the value "World."
- `echo "$greeting, $name!"`: This command prints the variables `greeting` and `name` with a comma and exclamation mark. The output will be: `Hello, World!`

Variables are essential for making your scripts more modular and easier to maintain. The values can be changed without modifying the entire script.

2.3 Variable Reassignment

Variables in shell scripts can be reassigned new values, making them adaptable to different scenarios. Let's see how variable reassignment works:
Bash
Copy to clipboard
```bash
#!/bin/bash

greeting="Hello"
hello=$greeting
echo $hello   # Prints: Hello
```

- `hello=$greeting`: This line assigns the value of the variable `greeting` to the variable `hello`. The `$` is required to access the value stored in the `greeting` variable
- `echo $hello`: This prints the value of `hello`, which is now "Hello."

With variable reassignment, you can easily propagate changes throughout your script by modifying a single value.

2.4 Defining and Using Integers

Shell scripts also allow you use arithmetic operations such as `+`, `-`, `*`, `/`, and `%`. Variables used in arithmetic operations must be preceded by `$`. The `$(())` syntax is used to evaluate the arithmetic expression within the parentheses. Let's take a look:
Bash

Copy to clipboard

```bash
#!/bin/bash

# Defining integers
num1=2
num2=5

# Performing addition
sum=$(($num1 + $num2))
echo $sum  # Prints: 7
```

- `num1=2` and `num2=5`: These lines create two integer variables, `num1` and `num2`.
- `sum=$(($num1 + $num2))`: This line access `num1` and `num2` using `$`. The arithmetic expression is inside `$(())` to correctly evaluate the expression.

2.5 Common Arithmetic Pitfalls

When performing arithmetic operations in shell scripts, it's easy to run into some common pitfalls. Let's take a look at a couple of examples that illustrate mistakes to avoid:

Bash

Copy to clipboard

```bash
#!/bin/bash

num1=2
num2=5

x=num1+num2
echo $x  # Prints: "num1+num2"

y=$num1+$num2
echo $y  # Prints: "2+5"
```

- `x=num1+num2`: This line attempts to set the value of `x` as the sum of `num1` and `num2`, but it actually assigns the string "num1+num2" to `x`. This is incorrect because the variable names are treated as a plain strings.
- `y=$num1+$num2`: This line tries to concatenate the values of `num1` and `num2`. Instead of adding the numbers, it results in the string "2+5". This happens because when using the `$` syntax without the `$(())`, the shell does not perform arithmetic operations.

To perform arithmetic operations correctly, always use the `$(())` syntax,

2.6 Correctly Naming Variables

When naming variables in shell scripts, it's important to follow certain rules to ensure your script runs correctly. Here are the key rules for proper variable naming:

- Variable names must begin with a letter (a-z, A-Z) or an underscore (`_`).
- After the first character, variable names can include letters, numbers (0-9), and underscores, but no other special characters.
- Variable names cannot contain spaces or special characters like `!`, `@`, `#`, `$`, `%`, `^`, `&`, `*`, `(`, `)`, `-`, `+`, `=`, etc.

- Variable names are case-sensitive. For example, `Var` and `var` are considered different variables.
- Do not use reserved words or keywords (such as `if`, `then`, `else`, `fi`, `for`, `while`, etc.) as variable names, as they have special meanings in shell scripting.

Proper variable naming is crucial to avoid errors and improve code readability. Let’s see some correct and incorrect ways to name variables:

Bash

Copy to clipboard

```bash
#!/bin/bash

# Correctly naming variables
var=1
var1=1
_var_1=1

# Incorrectly naming variables (commented out to avoid errors)
# 1Var=1
# 123=1
# !var=1
# @var=1
# var*1=1
```

2.7 Summary and Next Steps

Excellent job! Today, you've learned how to create and use variables, reassign variable values, define and use integers, and understand proper variable naming conventions in shell scripts. This knowledge is fundamental for writing dynamic and flexible scripts.

By mastering variables, you can already see how much more powerful your scripts can become. The next step is to dive into the practice problems to reinforce your understanding and apply what you've learned. Get ready to tackle the challenges ahead and enhance your scripting skills!

3. Comparison Operators in Shell Scripting

3.1 Introduction to Comparison Operators

Welcome to another exciting lesson in your shell scripting journey. In our previous lesson, we delved into using variables in shell scripts — learning how to store data that can be reused throughout a script. Today, we will build on that foundation by exploring **Comparison Operators** in shell scripting.

Comparison operators are essential tools for making decisions within your scripts. They allow you to compare values and execute different actions based on the results of these comparisons. This lesson will cover numeric and string comparisons and help you understand how to incorporate these comparisons into your scripts.

Let's dive in!

3.2 Explanation of Exit Status

Before diving into comparison operators, let's learn about exit statuses. An exit status is a numeric value returned by a command or a script to indicate its execution result. In the context of shell scripting, every command executed returns an exit status, which can be captured and used to determine the success or failure of that command.

An exit status of `0` indicates that the command or script was successful. A non-zero exit status (usually `1`) indicates that the command or script failed.

In shell scripting, comparisons return an exit status to indicate the result of the comparison:

- `0`: Indicates the comparison is true.
- `1`: Indicates the comparison is false.

The `echo $?` command in shell scripting is used to print the exit status of the last executed command.

3.3 Numeric Comparisons

Numeric comparisons are used to evaluate the relationship between two numbers. In shell scripting, we use the `[]` syntax with specific operators to perform these comparisons. There must be a single space at the beginning and end of the condition. Here are the most common numeric comparison operators:

- `-eq`: Equal to
- `-ne`: Not equal to
- `-gt`: Greater than
- `-lt`: Less than
- `-ge`: Greater than or equal to
- `-le`: Less than or equal to

Let's see an example:
Bash
Copy to clipboard

```bash
#!/bin/bash

# Variables
num1=10
num2=20

# Equal to
[ $num1 -eq $num2 ]
echo $?   # 1 (false)

# Not equal
[ $num1 -ne $num2 ]
echo $?   # 0 (true)

# Greater than
```

```
[ $num1 -gt $num2 ]
echo $?  # 1 (false)

# Less than
[ $num1 -lt $num2 ]
echo $?  # 0 (true)

# Greater than or equal to
[ $num1 -ge $num2 ]
echo $?  # 1 (false)

# Less than or equal to
[ $num1 -le $num2 ]
echo $?  # 0 (true)
```

- `num1=10` and `num2=20` defines two integer variables
- For each comparison (`-eq`, `-ne`, `-gt`, `-lt`, `-ge`, `-le`), the result is checked, and the exit status is printed using `echo $?`.
- `echo $?` prints `0` for true and `1` for false.

Understanding these operators lets you incorporate numeric conditions into your scripts effectively.

3.4 String Comparisons

Just like numbers, string comparisons help you evaluate relationships between text values. Here are the most common string comparison operators:

- `==` or `=`: Equal to
- `!=`: Not equal to
- `-z`: String is null (has a length of zero)
- `-n`: String is not null (has a length greater than zero)

Let's explore an example:
Bash
Copy to clipboard

```
#!/bin/bash

# Variables
str1="Hello"
str2="World"
str3="Hello"

# Equal to with ==
[ "$str1" == "$str2" ]
echo $?  # 1 (false)

# Equal to with =
[ "$str1" = "$str3" ]
echo $?  # 0 (true)

# Not equal to
```

```
[ "$str1" != "$str2" ]
echo $?   # 0 (true)

# Null
[ -z "$str1" ]
echo $?   # 1 (false)

# Not null
[ -n "$str1" ]
echo $?   # 0 (true)
```

- `str1="Hello"`, `str2="World"`, and `str3="Hello"` Define three string variables.
- Various string comparisons are performed, and the result is checked using `echo $?`.
- The exit status indicates whether the comparison is true or false.

These comparisons help you manipulate and evaluate text in your scripts.

3.5 Quotes in String Comparisons

Using quotes in string comparisons like `[ "$str1" == "$str2" ]` is essential for several important reasons:

Handling Empty Strings

Quotes help manage cases where variables might be empty or undefined. Without quotes, an empty variable can lead to syntax errors:

Bash
Copy to clipboard

```
#!/bin/bash

str1=""
[ $str1 == "Hello" ]   # [ == "Hello" ]
```

In this case, the shell interprets `str1` as an empty value, leading to an incomplete comparison like `[ == "Hello" ]`, which is not valid syntax.

Preserving Whitespaces

Quotes ensure that any whitespaces within string values are preserved and treated correctly. If you compare strings without quotes, the shell may treat each whitespace-separated part of the string as a distinct argument.

Bash
Copy to clipboard

```
#!/bin/bash

str1="Hello World"
[ $str1 == "Hello World" ]   # [ Hello == Hello World ]
```

Here, `str1` is split into two arguments: Hello and World. The comparison [Hello == Hello World] is not valid syntax and will lead to an error.

3.6 Summary and Next Steps

Great job! In today's lesson, you learned how to perform numeric and string comparisons in shell scripts. You explored different operators and saw how these comparisons can make your scripts more powerful and versatile.

Armed with this knowledge, you're ready to dive into practice problems to reinforce your learning and hone your scripting skills. The upcoming exercises will challenge you to apply these comparison operators in various scenarios, helping you gain a deeper understanding.

Let's jump into the practice section and start applying what you've learned. Happy scripting!

4. Control Structures and Logical Operators in Shell Scripting

4.1 Introduction to Control Structures and Logical Operators

In this lesson, we will delve into an essential aspect of shell scripting — **Control Structures and Logical Operators**. Control structures such as `if`, `elif`, and `else` allow you to make decisions within your scripts, enabling them to respond intelligently based on different conditions. Additionally, logical operators like `&&` (AND) and `||` (OR) help you combine multiple conditions to create more complex logic.

Let's get started!

4.2 Syntax of if/elif/else Statements

The `if/elif/else` control structure allows you to perform different actions based on various conditions. Here’s the syntax:

Bash

Copy to clipboard

```bash
if [ condition1 ]
then
    # Commands to execute if condition1 is true
elif [ condition2 ]
then
    # Commands to execute if condition1 is false and condition2 is true
else
    # Commands to execute if both condition1 and condition2 are false
fi
```

Each condition must be enclosed in `[]`. After the condition, the `then` keyword specifies the code to execute if the condition evaluates to `0` (true).

All control structures must start with an `if` statement. The `elif` and `else` statements are optional. Each `elif` condition is checked until one of the conditions evaluates to true. If none of the `if` or `elif` conditions are true, the `else` statement is executed.

The control structure must end with `fi`.

4.2.1 Using If-Else Statements

Let's take a look at an example to determine if a number is positive, negative, or neither.

Bash

Copy to clipboard

```bash
#!/bin/bash
# Script to demonstrate if-else control structure
number=5

if [ $number -gt 0 ]
then
    echo "The number is positive"   # Prints: "The number is positive"
elif [ $number -lt 0 ]
then
    echo "The number is negative"   # This won't execute
else
    echo "The number is zero"   # This won't execute
fi
```

- `number=5` initializes the variable `number` with the value 5.
- The `if` statement checks if `number` is greater than 0.
- If true, it prints "The number is positive".
- If the first condition is false, the `elif` statement checks if `number` is less than 0.
- If the `elif` condition is also false, the `else` statement executes, printing "The number is zero".

4.3 Introducing Logical Operators

Logical operators allow you to combine multiple conditions in your control structures. The `&&` (AND) operator ensures both conditions must be true, while the `||` (OR) operator requires at least one condition to be true.

Let's examine how they work:

4.3.1 Using the && Operator

The `&&` (AND) operator is used to combine multiple conditions in a control structure such that all the conditions must be true for the block of commands to execute. For example:

Bash

Copy to clipboard

```bash
#!/bin/bash
# Variables
num1=10
num2=20
str1="Hello"
str2="World"

# Using && (AND) Operator
if [ $num1 -lt $num2 ] && [ "$str1" != "$str2" ]
then
    echo "num1 is less than num2 AND str1 is not equal to str2"   #
Prints: "num1 is less than num2 AND str1 is not equal to str2"
else
```

```
    echo "Condition is false"   # This won't execute
fi
```

- `num1` is 10, `num2` is 20, `str1` is "Hello", and `str2` is "World".
- The `if` statement checks if `num1` is less than `num2` **AND** `str1` is not equal to `str2`.
- Both conditions are true, so the script prints: `"num1 is less than num2 AND str1 is not equal to str2"`.

4.3.2 Using the || Operator

The `||` (OR) operator is used to combine multiple conditions in a control structure such that at least one of the conditions must be true for the block of commands to execute. For example:

Bash
Copy to clipboard

```bash
#!/bin/bash

# Variables
num1=10
num2=20

# Using || (OR) Operator
if [ $num1 -gt 5 ] || [ $num2 -lt 5 ]
then
    echo "num1 is greater than 5 OR num2 is less than 5"   # Prints:
"num1 is greater than 5 OR num2 is less than 5"
else
    echo "Neither condition is true"   # This won't execute
fi
```

- The `if` statement checks if `num1` is greater than 5 **OR** `num2` is less than 5.
- Since `num1` is indeed greater than 5, the script prints: `"num1 is greater than 5 OR num2 is less than 5"`.

4.4 Summary and Next Steps

Excellent work! In this lesson, you explored the fundamental concepts of control structures and logical operators in shell scripting. You learned about `if-else` statements and how to combine conditions using `&&` and `||` operators.

These constructs are powerful tools that enable you to write more intelligent and responsive scripts. Understanding and applying these concepts will greatly enhance your ability to automate tasks and solve real-world problems.

Now, it's time to put your newfound knowledge to the test! Move on to the practice sections where you will apply what you've learned through hands-on exercises. Happy scripting!

5. Arrays and Looping Constructs in Shell Scripting

5.1 Introduction to Arrays and Looping Constructs in Shell Scripting

Hello! In this lesson, we will explore the powerful concepts of **arrays** and **looping constructs** in shell scripting. These are essential tools that will help you manage and manipulate data efficiently within your scripts. By the end of this lesson, you'll be able to handle arrays, iterate over their elements, and utilize different looping constructs to automate repetitive tasks.

Arrays allow you to store multiple items in a single variable, making it easier to handle lists of data. Looping constructs such as `for` and `while` loops enable you to execute a block of code multiple times, adding flexibility and power to your scripts.

Let's start our journey into arrays and looping constructs!

5.1.1 Syntax for Creating Arrays

In shell scripting, creating an array involves declaring a variable and assigning it multiple values enclosed in parentheses. Each value is separated by a space. Here's the general syntax:

Bash

Copy to clipboard

```
#!/bin/bash
computers=("Dell" "HP" "Lenovo")
```

You can add elements to an array using `+=`. For example:

Bash

Copy to clipboard

```
#!/bin/bash
computers=("Dell" "HP" "Lenovo")
computers+=("Mac")
```

Now the array also includes the value "Mac".

5.2 Array Length and Printing

Often you need to find the length of an array or print out all of its contents. You can access the length of an array using `${#array_name[@]}`. To print the contents of an array use `${array_name[@]}`. Let's take a look:

Bash

Copy to clipboard

```
#!/bin/bash
computers=("Dell" "HP" "Lenovo")
computers+=("Mac")
echo "Number of computers: ${#computers[@]}"
echo "All computers: ${computers[@]}"
```

- `${#computers[@]}` accesses the number of elements using the `${#array_name[@]}` syntax
- `${computers[@]}` accesses all the elements of the array using the `${array_name[@]}` syntax.

This script prints out

Plain text

Copy to clipboard

```
Number of computers: 4
All computers: Dell HP Lenovo Mac
```

5.2.1 Array Indexing

Array indexing is the method of accessing individual elements in an array using their position, known as the index. Similar to other coding languages, the first element is at index 0. To access an element of an array use `${array_name[index_number]}`. Let's take a look:

Bash

Copy to clipboard

```bash
#!/bin/bash
computers=("Dell" "HP" "Lenovo")
echo "The first computer is: ${computers[0]}"
echo "The second computer is: ${computers[1]}"
echo "The third computer is: ${computers[2]}"
```

The output of this script is:

Plain text

Copy to clipboard

```
"The first computer is: Dell"
"The second computer is: HP"
"The third computer is: Lenovo"
```

With this understanding of arrays, let's shift our focus to loops.

5.3 Using While Loops

The `while` loop in shell scripting allows you to execute a block of code repeatedly as long as a given condition is true. It's useful for cases where the number of iterations is not known beforehand and depends on certain conditions. The syntax structure of a `while` loop is:

Bash

Copy to clipboard

```bash
#!/bin/bash
while [ condition ]
do
    command1
    command2
    ...
done
```

- `condition` is a test expression that is evaluated before each iteration. The loop continues as long as this condition is true.
- The `do...done` block contains the commands to be executed as long as the condition is true.

Let's look at an example:

Bash

Copy to clipboard

```bash
#!/bin/bash
```

```bash
counter=1

# Start the while loop
while [ $counter -le 3 ]
do
    echo "Counter: $counter"
    # Increment the counter
    ((counter++))
done
```

- `counter=1` creates a variable named `counter` is initialized to 1. This variable will control the number of iterations in the loop.
- `while [ $counter -le 3 ]`: The `while` loop starts with the condition `[ $counter -le 3 ]`. The loop will continue to execute as long as the value of `counter` is less than or equal to (`-le`) 3.
- `do` marks the start of the block of commands to run for each iteration
- `echo "Counter: $counter"`: Inside the loop, the value of the `counter` variable is printed
- `((counter++))`: This command increments the `counter` variable by 1.
- `done`: This marks the end of the `while` loop. The loop then checks the condition `[ $counter -le 3 ]` again. If it is true, the loop executes again; otherwise, it stops.

The output of the script is:

Plain text

Copy to clipboard

```
Counter: 1
Counter: 2
Counter: 3
```

It is important to properly increment the `counter` variable so the condition eventually becomes false. If the `counter` variable is not updated appropriately, the `while` loop will run indefinitely.

5.4 Using For Loops

The `for` loop in shell scripting allows you to iterate until a condition is met. It's commonly used to automate repetitive tasks by executing a block of code multiple times. The syntax of a `for` loop is:

Bash

Copy to clipboard

```bash
#!/bin/bash
for ((initialization; condition; update))
do
  commands
done
```

- `initialization`: This sets the starting value for the loop control variable. It is executed only once at the beginning of the loop.
- `condition`: This is a boolean expression. The loop continues to execute as long as this condition is true. It is evaluated before each iteration.
- `update`: This operation updates the loop control variable after each iteration of the loop.

- `do`: This keyword indicates the beginning of the block of commands that will be executed in each iteration of the loop.
- `commands`: These are the commands or code that will be executed each time the loop iterates.
- `done`: This keyword marks the end of the loop block. After executing the commands, control goes back to the `update` step and the condition is checked again. If the condition is true, the loop runs again; otherwise, it terminates.

Here's a simple example to illustrate the syntax:

Bash

Copy to clipboard

```bash
#!/bin/bash
for ((x=1; x<=3; x++))
do
  echo "Iteration $x"
done
```

- The loop control variable `x` is initialized to 1, and the loop continues as long as `x` is less than or equal to 3. The variable `x` is incremented by 1 after each iteration.
- `echo "Iteration $x"`: This command prints the current value of `x` during each iteration.
- `done`: This marks the end of the loop. The loop will repeat until the condition `x<=3` is no longer true.

The output of the script is:

Plain text

Copy to clipboard

```
Iteration 1
Iteration 2
Iteration 3
```

Using "for in" loops

The `for in` loop in shell scripting allows you to iterate over a sequence of values or elements. The syntax of a `for` loop is:

Bash

Copy to clipboard

```bash
#!/bin/bash
for variable in list
do
    command1
    command2
    ...
done
```

- `variable` is a loop control variable that takes the value of each element in the list one by one.
- `list` can be a sequence of numbers, strings, or an array.
- The `do...done` block contains the commands to be executed in each iteration.

Let's see a concrete example looping over a range of numbers. A range is defined as `{start..end}`. Unlike some coding languages, `end` is **inclusive**. Let's take a look:

Bash

Copy to clipboard

```bash
#!/bin/bash
# Script to demonstrate for loop
for i in {1..5}
do
  echo "Iteration $i"
done
```

- `for i in {1..5}`: The loop control variable `i` takes values from 1 to 5.
- `do ... done`: This specifies the block of code to run for each iteration
- `echo "Iteration $i"`: Prints the current iteration number.

The output of the above script is:
Plain text
Copy to clipboard

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

5.5 Looping Through Arrays

Combining arrays and loops, you can iterate over array elements by combining `for in` with `"${array_name[@]}"`. Here's an example:
Bash
Copy to clipboard

```bash
#!/bin/bash
# Looping through array
computers=("Dell" "HP" "Lenovo")

for computer in "${computers[@]}"
do
    echo "$computer"
done
```

- `for computer in "${computers[@]}"`: The `for` loop starts with the control variable `computer`.
- `do ... done`: The code inside this block is run during each iteration.
- `echo "$computer"`: The loop prints the current value of the `$computer` variable from the `computers` array

The output of the script is:
Plain text
Copy to clipboard

```
Dell
HP
Lenovo
```

5.6 Summary and Next Steps

Great job! In this lesson, you've learned the fundamentals of arrays and looping constructs in shell scripting. You now know how to declare and access arrays, use `for` loops and `while` loops, and iterate over array elements.

These skills are crucial for automating tasks and handling complex data in your scripts. Arrays allow you to manage multiple items efficiently, while loops enable you to perform repetitive tasks seamlessly.

Now, it's time to apply what you've learned! Proceed to the practice section to reinforce your understanding with hands-on exercises. Get ready to enhance your scripting skills further. Happy coding!

6. Functions in Shell Scripts

Welcome to this lesson about **functions in shell scripts**, a powerful concept that helps you organize your code efficiently. Functions allow you to group commands into reusable blocks, making your scripts cleaner and easier to manage. By the end of this lesson, you'll have a solid grasp of creating and using functions in your shell scripts.

Let's dive in!

6.1 Introduction to Functions

Functions in shell scripts are similar to those in other programming languages. They help you encapsulate code, making it modular, reusable, and easier to debug. Think of a function as a mini-script within your main script that you can call whenever needed.

6.2 Defining and Calling Functions

The basic syntax for defining a function in shell scripts is straightforward. Here's how you can define and call a simple function:

Bash

Copy to clipboard

```bash
#!/bin/bash
# Defining a function
function_name() {
  # Commands to be executed
}

# Calling the function
function_name
```

Let's start by defining a simple function:

Bash

Copy to clipboard

```bash
#!/bin/bash
# Defining a function
greet() {
  echo "Hello"
}

greet
```

- `greet() { ... }`: This block defines a function named `greet`.
- `echo "Hello"`: This command inside the function prints "Hello"
- `greet`: This line calls the function `greet` resulting in the output "Hello".

6.3 Passing Arguments to Functions

In shell scripting, you cannot directly pass named parameters to functions in the same way as you might in some other programming languages. Instead, you pass positional parameters. When calling a function, each parameter is assigned a position number, starting at 1. These arguments are then accessed inside the function using `$position_number`. Let's take a look:

Bash

Copy to clipboard

```bash
#!/bin/bash
# Function with 1 input
update_one() {
  echo "Updating $1"
}

# Function with 2 inputs
update_two() {
    echo "Updating $1 and $2"
}

update_one "Photos"
update_two "Photos" "Browser"
```

- `update_one` is a function that takes one input parameter (`$1`). The function prints the string "Updating" followed by the value of the input parameter.
- `update_two` is a function that takes two input parameters (`$1` and `$2`). The function prints the string "Updating" followed by the values of the two input parameters, separated by "and".
- `update_one "Photos"` calls the `update_one` function and the positional argument `$1` is "Photos"
- `update_two "Photos" "Browser"` calls the update_two function. The positional argument `$1` is "Photos" and the positional argument `$2` is "Browser".

The output of this script is:

Plain text

Copy to clipboard

```
Updating Photos
Updating Photos and Browser
```

6.4 Counting Arguments with `$#`

In addition to handling specific arguments, you may want to know the total number of arguments passed to a function. You can do this using the special variable `$#`, which holds the number of positional parameters.

Bash

Copy to clipboard

```
#!/bin/bash
# Function to count arguments
count_args() {
    echo "Number of arguments: $#"
}

count_args "Photos" "Browser" "Documents"
```

- `count_args() { ... }`: This block defines a function named `count_args`.
- `echo "Number of arguments: $#"`: This command prints the total number of arguments passed to the function.
- `count_args "Photos" "Browser" "Documents"` calls the `count_args` function with three arguments, so the output will be `Number of arguments: 3`.

6.5 Iterating Over Arguments Using Arrays and `$@`

In shell scripting, you might find yourself needing to iterate over arguments both as an array and using the `$@` special parameter. You can convert the arguments to an array using `args=("$@")`. You can iterate over the arguments directly using `"$@"`.

Below is an example that demonstrates both methods within a function:

Bash

Copy to clipboard

```
#!/bin/bash
print_args() {
  arr=("$@")

  echo "Using array indices:"
  for x in "${arr[@]}"
   do
    echo "$x"
  done

  echo "Using \$@:"
  for y in "$@"
  do
    echo "$y"
  done
}

print_args "Photos" "Browser" "Documents"
```

- `arr=("$@")`: This command assigns all positional parameters to an array named `arr`.
- `for x in "${arr[@]}"; do`: This loop iterates over each element in the `arr` array, printing each argument.
- `for y in "$@"; do`: This loop iterates directly over the positional parameters using `$@`.

When calling `print_args "Photos" "Browser" "Documents"`, the output will be:

Plain text

Copy to clipboard

```
Using array indices:
Photos
Browser
Documents
Using $@:
Photos
Browser
Documents
```

6.6 Returning Values from Functions

While functions can echo or print output directly, you can also return values using `echo` and capture them using command substitution. For example:

Bash

Copy to clipboard

```bash
#!/bin/bash
# Function to increase a version number
increase_version() {
  ((version += increment))
  echo $version
}

version=2
increment=3
# Capture the return value using command substitution
result=$(increase_version)
echo "Updating from version $version to $result"    # Prints: "Updating
from version 2 to 5"
```

- `increase_version() { ... }`: This block defines a function named `increase_version`.
- `((version += increment))`: This command increments the `version` number by the value of `increment`.
- `echo $version`: This command prints the updated `version` number.
- `result=$(increase_version)`: This line captures the function’s output using command substitution.
- `echo "Updating from version $version to $result"`: This command prints the message with the original and updated version numbers.

6.7 Summary and Next Steps

Excellent work! In this lesson, you’ve learned the essentials of defining and calling functions in shell scripts, passing arguments to functions, and capturing return values. Functions are a crucial tool for writing efficient, modular, and maintainable shell scripts.

Now, it’s your turn to practice what you've learned. Dive into the practice section to reinforce your understanding with hands-on exercises. Happy coding!