**Shell programming on Linux**

1. **Create and execute Shell file**

1.1. Open a Text Editor: You can use any text editor to create a shell script. Common editors include `nano`, `vim`, or `gedit`. For example, to use `nano`, open your terminal and type:

nano hello.sh

This will open a new, empty file named `hello.sh` in the `nano` text editor.

1.2. Write the Shell Script: In the text editor, type the following lines of code:

#!/bin/bash

# This is a simple shell script that prints "Hello, World!" to the terminal.

echo "Hello, World!"

The `#!/bin/bash` line is called a "shebang" and tells the system that this script should be interpreted using the Bash shell.

1.3. Save the Script: In `nano`, you can save the file by pressing `Ctrl` + `O`, then press `Enter`. To exit, press `Ctrl` + `X`.

1.4. Make the Script Executable: Before you can run the script, you need to make it executable. In your terminal, use the `chmod` command:

chmod +x hello.sh

This command grants execute permission to the `hello.sh` script.

1.5. Execute the Script: Now, you can run the script by entering its name preceded by `./`:

./hello.sh

You should see the output "Hello, World!" printed to the terminal.

1. **Variables, read user input, pass argument**

Here's a script that demonstrates these concepts:

#!/bin/bash

# Define a variable

name=""

# Read user input

echo "What is your name?"

read name

# Print a greeting

echo "Hello, $name!"

# Check if an argument was provided

if [ $# -eq 0 ]; then

echo "No arguments provided."

else

echo "You provided $# argument(s):"

# Loop through and print each argument

for arg in "$@"; do

echo "- $arg"

done

fi

Save this script as, for example, greet.sh, and follow the same steps as before to make it executable:

chmod +x greet.sh

You can run it with arguments like this:

./greet.sh John Doe Alice

This will produce output like:

What is your name?

[Enter your name]

Hello, [Your Name]!

You provided 3 argument(s):

- John

- Doe

- Alice

The script demonstrates the use of a variable (name), reading user input with read, and checking for and printing command-line arguments ($# for the number of arguments and "$@" to loop through them).

1. **Array**

3.1. Declaration and Initialization:

You can declare an array and initialize it in several ways. Here's an example of declaring an array and populating it:

# Declare an empty array my\_array=() # Initialize an array with values my\_array=("apple" "banana" "cherry")

3.2. Accessing Array Elements:

You can access individual elements of an array using their indices. Array indices start at 0.

# Access the first element echo ${my\_array[0]} # Output: apple # Access the second element echo ${my\_array[1]} # Output: banana

3.3. Array Length:

To find the length (number of elements) of an array, you can use the ${#array\_name[@]} syntax.

# Get the length of the array length=${#my\_array[@]} echo "Array length: $length" # Output: Array length: 3

3.4. Adding Elements to an Array:

You can append elements to an array using the += operator.

my\_array+=("date")

3.5. Iterating Through an Array:

You can loop through the elements of an array using a for loop.

# Loop through and print each element for element in "${my\_array[@]}"; do echo $element done

3.6. Associative Arrays:

In addition to indexed arrays, Bash also supports associative arrays, where you can use strings as keys.

# Declare an associative array declare -A fruits # Initialize the associative array fruits["apple"]="red" fruits["banana"]="yellow" # Access values using keys echo "Color of apple: ${fruits["apple"]}" # Output: Color of apple: red

1. **Loop**

4.1. for Loop:

A for loop is used when you know how many times you want to iterate. It's typically used to iterate over a list of items or a range of values.

Here's a simple example of a for loop that iterates over a list of numbers and prints each number:

for num in 1 2 3 4 5 do echo "Number: $num" done

You can also use a for loop to iterate over the elements of an array:

fruits=("apple" "banana" "cherry") for fruit in "${fruits[@]}" do echo "Fruit: $fruit" done

4.2. while Loop:

A while loop is used when you want to continue executing a set of commands as long as a specified condition is true.

Here's an example of a while loop that counts from 1 to 5:

count=1 while [ $count -le 5 ] do echo "Count: $count" ((count++)) done

The loop will continue as long as the condition [ $count -le 5 ] (count is less than or equal to 5) is true.

4.3. until Loop:

An until loop is similar to a while loop, but it continues executing a set of commands until a specified condition becomes true.

Here's an example of an until loop that counts from 1 until it reaches 5:

count=1 until [ $count -gt 5 ] do echo "Count: $count" ((count++)) done

The loop will continue until the condition [ $count -gt 5 ] (count is greater than 5) becomes true.

You can combine loops with conditional statements (if statements) and other control structures to create more complex scripts.

1. **Function**

Defining a Function:

You define a function using the function keyword followed by the function name and a pair of parentheses. The function body is enclosed within curly braces {}.

bashCopy code

function my\_function { # Function code goes here echo "This is my function" }

Alternatively, you can omit the function keyword and define a function like this:

bashCopy code

my\_function() { # Function code goes here echo "This is my function" }

Calling a Function:

To call a function, simply use its name followed by a pair of parentheses.

bashCopy code

my\_function

Passing Arguments to a Function:

You can pass arguments to a Bash function just like you would with a script. Inside the function, you can access these arguments using special variables like $1, $2, and so on.

bashCopy code

function greet { echo "Hello, $1!" } greet "Alice" # Calls the function with the argument "Alice" greet "Bob" # Calls the function with the argument "Bob"

Returning Values:

Bash functions can return values using the return statement. The returned value can be stored in a variable when calling the function.

bashCopy code

function add { result=$(( $1 + $2 )) return $result } # Call the function and store the result in a variable sum=$(add 5 3) echo "Sum is $sum" # Output: Sum is 8

Local Variables:

By default, variables declared inside a function are local to that function and do not affect variables with the same name outside the function. This is known as variable scoping.

bashCopy code

my\_function() { local var="Inside Function" echo "Inside Function: $var" } var="Outside Function" my\_function echo "Outside Function: $var"

This code will output:

javascriptCopy code

Inside Function: Inside Function Outside Function: Outside Function

Functions are an essential part of Bash scripting, allowing you to modularize your code and make it more maintainable. They are particularly useful when you need to perform a specific task multiple times within your script.