Scripting Lesson 5: Shell Scripting Essentials

Course	DevOps Engineering
Topic / Session number	Scripting / Lesson 6
Objectives / Goals	1. Error Handling 2. Functions - Reusable Code 3. Arrays 4. Local vs Global Variables 5. Practice Questions

---

★ Objective 1: Error Handling

Handling Errors with Exit Status:

```bash
#!/bin/bash
command_that_might_fail
if [ $? -ne 0 ]; then
  echo "There was an error!"
else
  echo "Success!"
fi

```

Using `trap` to Handle Errors:

```bash
#!/bin/bash
trap 'echo "An error occurred."' ERR

```

- The `trap` command executes a specified action when a particular signal (like an error) is encountered.

★ Objective 2: Functions - Reusable code

In shell scripting, functions are reusable blocks of code that you can call to perform specific tasks. Functions help make your script more organized and efficient. In this guide, I'll walk you through the basics of defining functions, passing values to them, and returning values from them in a simple and beginner-friendly way.

Defining a Function

A function in shell scripting is defined by the following syntax:

```bash
function function_name {
    # Function code here
```

```
}

```

Alternatively, you can also define a function like this:

```
function_name() {
    # Function code here
}

```

Example:

```
#!/bin/bash
greet() {
  echo "Hello, $1!"
}
greet "Alice"
greet "Bob"

```

> **Analogy**: Functions are like a recipe. Once you define the recipe (function), you can use it as many times as you want to bake a cake (run the function).

Passing Arguments to a Function

When calling a function, you can pass arguments to it. These arguments can be accessed within the function using special variables:

- `$1` refers to the first argument.
- `$2` refers to the second argument, and so on.
- `$@` refers to all arguments.

Returning Values from a Function

Shell functions don't have a built-in way to return complex values like in some other languages. However, you can:

- Use the `echo` command to output the value, and capture that output when calling the function.
- Use the `return` statement to return an exit status (an integer).

Here's a complete example:

Example 1: Passing and Returning Values Using Echo (For Beginners)

This script defines a function that takes two numbers, adds them, and returns the result using `echo`.

```
#!/bin/bash

# Define the function to add two numbers
add_numbers() {
    # Arguments are passed as $1, $2, etc.
    result=$(( $1 + $2 ))
    echo $result  # Echo the result, so we can capture it when calling the function
}

# Main script
echo "Enter the first number: "
read num1

```

```
echo "Enter the second number: "
read num2

# Call the function and store the result
sum=$(add_numbers $num1 $num2)

# Display the result
echo "The sum of $num1 and $num2 is: $sum"

```

Explanation:

1. `add_numbers` is a function that takes two arguments, adds them, and outputs the result using `echo`.
2. We read two numbers from the user using `read`.
3. The result of the function is captured using `$(add_numbers $num1 $num2)` and stored in the variable `sum`.
4. Finally, we print the sum.

Example 2: Passing Arguments and Using Return Codes

In this script, we'll use the `return` statement to set an exit status (not a real value, but an integer). This is useful for simple status codes.

```
#!/bin/bash

# Define a function that checks if the number is positive
check_positive() {
    if [[ $1 -gt 0 ]]; then
        return 0  # Return 0 if the number is positive
    else
        return 1  # Return 1 if the number is not positive
    fi
}

# Main script
echo "Enter a number: "
read num

# Call the function
check_positive $num

# Check the exit status using $?
if [[ $? -eq 0 ]]; then
    echo "$num is a positive number."
else
    echo "$num is not a positive number."
fi

```

Explanation:

1. The function `check_positive` takes one argument (a number) and checks if it's greater than zero.
2. It returns `0` (success) if the number is positive, or `1` (failure) if not.
3. We use `$?` to check the return code of the function and print the appropriate message.

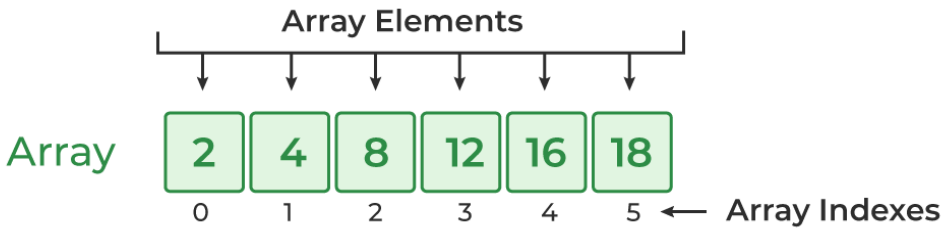
Summary:

- **Passing Arguments:** You can pass values to a function, which are accessed inside the function using `$1`, `$2`, etc.
- **Returning Values:**

- **Using `echo`:** You can return values by printing them, and capture that output.
- **Using `return`:** This is used for returning an exit status (integer), commonly used for error handling.

★ Objective 3: Arrays

Arrays in shell scripting are used to store multiple values under a single variable name. In bash, arrays are zero-indexed.



Syntax:

```
array_name[index]=value

```

Examples:

- **One-dimensional Array**:

```
# Declare an array
fruits=("Apple" "Banana" "Cherry")

# Access array elements
echo ${fruits[0]}  # Output: Apple
echo ${fruits[1]}  # Output: Banana

```

- **Accessing All Array Elements:**

```
# Loop through array
for fruit in "${fruits[@]}"; do
    echo $fruit
done
# Output: Apple, Banana, Cherry (one per line)

```

- **Array Length**: To get the length of the array, you can use the following syntax:

```
echo ${#fruits[@]}  # Output: 3 (number of elements in the array)

```

- **Associative Arrays (Bash 4.0 and above)**: An associative array stores key-value pairs (similar to dictionaries in Python).

```
declare -A person
person[name]="Alice"
person[age]=25
person[city]="New York"

# Access values by key
echo ${person[name]}  # Output: Alice
```

```
echo ${person[age]}   # Output: 25

```

⭐ Objective 4: Local vs Global Variables

Variables in shell scripting can be either **global** or **local**. The scope of a variable determines where it can be accessed in the script.

Global Variables:

- These are the default type of variable in shell scripts.
- A **global variable** is accessible throughout the script, including inside functions.

Local Variables:

- These are variables that are accessible only **within a function**.
- To declare a **local variable**, you use the `local` keyword inside a function.
- A **local variable** cannot be accessed outside the function where it's declared.

Example of Global Variables

```
#!/bin/bash

# Global variable
greeting="Hello, World!"

# Function that uses global variable
say_hello() {
    echo $greeting
}

say_hello  # Output: Hello, World!

```

In the example above, the variable `greeting` is accessible both inside and outside the function `say_hello`.

Example of Local Variables

```
#!/bin/bash

# Global variable
greeting="Hello, World!"

# Function with a local variable
say_hello() {
    local greeting="Hello, Alice!"  # Local variable
    echo $greeting  # Output: Hello, Alice!
}

# Call the function
say_hello

# Global variable is still accessible outside the function
echo $greeting  # Output: Hello, World!

```

Explanation:

- The `greeting` variable inside the `say_hello` function is **local** and only accessible within that function.
- The global variable `greeting` remains unchanged outside the function and retains its value.

Example with Both Global and Local Variables

```bash
#!/bin/bash

# Global variable
username="John"

# Function to display username
display_username() {
    local username="Alice"  # Local variable with same name
    echo "Hello, $username!"  # Output: Hello, Alice!
}

# Calling the function
display_username

# Global variable is not affected
echo "Global username: $username"  # Output: Global username: John

```

In this example, the local variable `username` inside the function doesn't affect the global `username` outside the function.

Important Points

1. **Global Variable**: A variable declared outside of any function is accessible throughout the script, including inside functions.
2. **Local Variable**: A variable declared inside a function using `local` is only accessible within that function.
3. If you don't use the `local` keyword inside a function, the variable will be treated as **global** and could potentially overwrite a global variable.

When to Use Local vs Global Variables

- **Use Global Variables** when:
  - The variable needs to be accessed by multiple functions or in multiple places throughout the script.
  - You want to share data across different parts of your script.
- **Use Local Variables** when:
  - The variable is only needed inside a function.
  - You want to avoid modifying a global variable accidentally.
  - You want to prevent namespace conflicts and keep your script clean and predictable.

Conclusion

- **Global Variables** are visible across the entire script, including inside functions.
- **Local Variables** are confined to the scope of the function they are declared in and help prevent unintended changes to global variables.
- It’s always good practice to use **local variables** inside functions to ensure better code modularity and reduce potential bugs related to variable conflicts.

★ Objective 5: Practice Questions

1. Write a script that automates file backup. Ask the user for a directory to back up and copy it to a backup location with the current date.
2. Write a script that monitors system disk usage and sends an alert if disk usage exceeds 90%.