

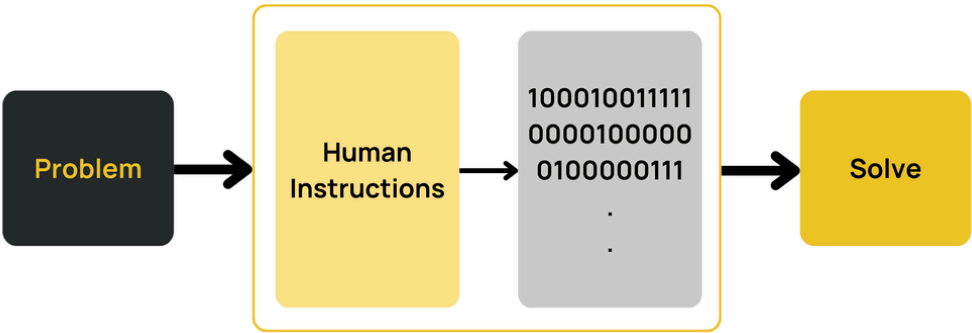
Scripting Lesson 1: Shell Scripting Basics

Topic / Session number	Scripting / Lesson 1
Objectives / Goals	<div>1. Introduction To Programming<div>a. What Is Programming? b. Computational Thinking</div></div> <div>2. What Is Scripting and Why Learn It?</div> <div>3. Shell Script File Structure</div> <div>4. Variables</div> <div>5. Running Shell Script</div> <div>6. Conditionals<div>a. Multiple Conditions b. Nested If Statements</div></div> <div>7. Com</div> <div>8. Quot _</div> <div>9. Exit Status and Special Variables in Shell</div> <div>10. Different Types of Shells</div> <div>11. Practice Questions</div>

★ Objective 1: Introduction To Programming [↗](#)

What Is Programming? [↗](#)

Programming is the process of writing instructions that a computer can understand and execute. These instructions tell the computer what to do in a step-by-step manner. The goal of programming is to solve problems or perform tasks by getting a computer to follow these instructions. It is like teaching a computer how to do something using a special "language" that both humans and computers can understand.



Analogy: Programming is like giving instructions to a robot. ↗

Imagine you have a robot at home, and you want it to make you a sandwich. You need to give the robot clear, step-by-step instructions, like:

- **Pick up the bread.**
- **Spread peanut butter on one slice of bread.**
- **Put jelly on the other slice.**
- **Press the two slices together.**
- **Cut the sandwich in half.**

If you don't give clear instructions, the robot may not understand what you want it to do. Programming is similar. You give instructions to the computer, and if your instructions (code) are correct, the computer will do exactly what you ask.

Key Points: ↗

- **Programming is like writing a recipe:** Just as a recipe tells you how to prepare a dish step by step, programming tells the computer how to solve a problem step by step.
- **Programming Languages:** Just like humans speak different languages, computers use programming languages like Python, Java, or C++.
- **Computer as a Tool:** Think of a computer as a tool that helps you achieve tasks. Programming is how you give the computer the specific instructions on how to carry out those tasks.

Computational Thinking ↗

Computational thinking is the process of thinking logically about how to solve a problem in a way that a computer can understand and perform. It's a way of breaking down a problem into smaller, manageable parts and creating step-by-step instructions to solve them. It's not just for programming; computational thinking can help with everyday problem solving.



Analogy: Solving a Puzzle [↗](#)

Imagine you're trying to complete a jigsaw puzzle. The puzzle has many pieces, and your goal is to put them together to form a picture. Here's how computational thinking might work for this:

1. **Decompose:** Break the puzzle into smaller sections. You might first focus on the edges, then work on the colors or shapes of pieces in the middle.
2. **Pattern Recognition:** Look for patterns in the puzzle pieces, like matching colors or shapes. This will help you know which pieces go together.
3. **Abstraction:** Ignore unnecessary details. You don't need to focus on the picture on each individual piece. You just need to focus on the colors and shapes that fit.
4. **Algorithm Design:** Decide the steps you'll take to solve the puzzle. For example, you might first find all the edge pieces, then work on the sections with similar colors.

Key Points: [↗](#)

- **Decomposition:** Breaking a large problem into smaller, easier-to-manage parts.
 - **Example:** Solving a big math problem by breaking it into smaller steps.
- **Pattern Recognition:** Identifying similarities or repeating elements that can simplify the problem-solving process.
 - **Example:** Recognizing that certain pieces of a puzzle fit together based on their shape or color.
- **Abstraction:** Focusing on the essential details and ignoring irrelevant ones. This simplifies the problem and makes it easier to solve.
 - **Example:** Ignoring the actual picture on the puzzle pieces and just focusing on the shape.
- **Algorithm Design:** Creating a step-by-step process to solve the problem efficiently.

- **Example:** Deciding on the best order to solve the puzzle, starting with the edges and then the middle.

Computational thinking is important because it helps you approach complex problems in a structured way that can be understood by both humans and computers. It is the foundation for learning how to program and solve problems using technology.

Summary

- **Programming** is like giving instructions to a computer to make it do something. Just like you tell a robot how to make a sandwich, you tell a computer how to solve problems using a programming language.
- **Computational thinking** helps you break down complex problems into smaller, manageable pieces. It's like solving a jigsaw puzzle, where you first break it into sections, look for patterns, and then follow a set of steps to complete the puzzle.

These skills are essential for anyone learning to work with computers, whether for programming or solving everyday problems!

★ Objective 2: What Is Scripting and Why learn it?

Scripting refers to writing a series of commands that automate tasks on a computer. Instead of manually typing commands into the terminal every time, you can **automate** those actions with a script.

Analogy: Imagine you are following a recipe to bake a cake. If you had to perform each step (measuring ingredients, mixing, baking, etc.) every time, it would take a lot of time. Instead, you can write the recipe down (a script), and follow it automatically each time you want to bake the cake.

Why Learn Shell Scripting?

To understand why scripting is useful, let's start with an analogy and then dive into its importance from a **DevOps engineer's** perspective.

Analogy: Imagine you're in a kitchen, preparing a big meal. You have a recipe book that lists all the steps, such as chopping vegetables, boiling pasta, or frying onions. These are the instructions you follow to get everything ready.

Manual Way: If you had to perform these tasks every time without a recipe, you'd have to remember every step and repeat it over and over again. It would take a lot of time and you could make mistakes. You'd also be very tired by the time you're done!

Using a Recipe (Scripting): But if you had a recipe book, all the steps would be written out for you, and you could just follow the instructions. Even better, you could give the recipe to someone else and they could follow it exactly the same way you did. The recipe is like a script - it automates the process and makes sure everything is done correctly every time, without you having to do it manually.

In the **DevOps world**, **scripting is like that recipe** that automates the process of managing and maintaining infrastructure, software, and systems. It makes sure tasks are performed **consistently** and **efficiently** without human errors.

DevOps engineers are professionals who work to improve the development, deployment, and maintenance of software systems. They bridge the gap between software development (**dev**) and IT operations (**ops**). One of the main tasks of a DevOps engineer is to automate processes to make them faster, more reliable, and scalable.

Here's how **learning shell scripting** is **crucial for DevOps engineers**:

1. Automating Repetitive Tasks:

A DevOps engineer's job often involves a lot of repetitive tasks, such as deploying software updates, configuring servers, or setting up cloud infrastructure. Instead of doing each task manually, which takes time and might introduce errors, DevOps engineers use scripts to **automate** them.

Example: Imagine you need to install updates on 100 servers. Doing this manually means logging into each server, typing commands, and waiting for each one to finish. But with a **script**, you can automate this entire process, running the same command on all servers at once.

Analogy: It's like having a robot chef in the kitchen that follows the recipe without making mistakes and does the same thing every time, allowing you to focus on more important tasks.

2. Saving Time and Reducing Errors:

Every task that is automated through a script saves time. **Time is a valuable resource** in DevOps, where engineers are working to deploy software quickly, fix issues faster, and scale systems without delay.

Scripting allows DevOps engineers to **reduce human errors**, which can often happen when doing tasks manually. For instance, when setting up servers or deploying applications, typing errors can lead to problems. A script performs the task exactly as written, preventing mistakes.

Analogy: If you're following a recipe, the recipe won't forget to add an ingredient or put the dish in the oven at the wrong temperature. The script makes sure everything is done correctly every time.

3. Managing Multiple Systems Efficiently:

In modern software environments, there are often hundreds or thousands of systems (servers, virtual machines, containers, etc.). Managing all these systems manually is **impossible**. DevOps engineers use scripts to control many systems at once, execute commands on them in batches, and ensure consistency across them all.

Analogy: Imagine cooking meals for hundreds of people. You can't do it by hand for each person, but you could use machines to speed up cooking and ensure everyone gets the same quality meal at the same time. Scripts are like those machines – they help scale the process for hundreds or thousands of systems.

4. Configuring and Monitoring Systems:

DevOps engineers need to **configure** servers, databases, networks, and other systems. Instead of doing it manually, they write scripts to **set up** and **monitor** systems automatically. For example, scripts can be used to set up a new server, configure a firewall, or check if a website is running smoothly.

Analogy: It's like having an automated kitchen assistant who checks if all the appliances are working properly, and if not, sends you an alert to fix them.

5. Continuous Integration/Continuous Deployment (CI/CD):

In modern software development, **CI/CD pipelines** are used to **automatically test, build, and deploy** applications. Scripts are essential in setting up and managing these pipelines. They help ensure that every time new code is written, it's automatically tested and deployed without manual intervention.

Analogy: Think of this as an automated kitchen assembly line. Each time a new dish is ready, it goes through different stages – testing, cooking, plating – and is automatically sent out for delivery, without anyone having to do it by hand.

6. Working with Cloud Infrastructure: [🔗](#)

Many DevOps engineers work with cloud platforms like AWS, Azure, or Google Cloud. These platforms provide APIs that allow engineers to manage infrastructure using scripts. You can **spin up servers**, **configure networking**, and even create **virtual environments** using shell scripts.

Analogy: It's like controlling a smart kitchen where you can adjust the temperature of the oven, turn on the lights, or start cooking remotely using your phone or computer. With scripting, you can manage your cloud infrastructure easily without being physically present at each server.

Summary [🔗](#)

Scripting is an essential skill for a **DevOps engineer** because it allows them to:

- **Automate repetitive tasks**, saving time.
- **Ensure consistency** and **reduce human errors**.
- **Scale infrastructure management** for hundreds or thousands of systems.
- **Monitor and configure systems** efficiently.
- **Set up CI/CD pipelines** that automate software delivery.

By learning scripting, DevOps engineers can focus on high-level tasks while letting the scripts handle the repetitive and error-prone tasks. It's like giving them a **smart assistant** in the IT world that does all the routine work, allowing them to **work faster**, **scale easily**, and **ensure reliability**.

★ Objective 3: Shell Script File Structure [🔗](#)

A **shell script** is simply a text file that contains a sequence of commands for the shell to execute.

Basic Structure of a Shell Script: [🔗](#)

1. **Shebang (#!):** The first line of the script begins with `#!/bin/bash`, known as the "shebang". This tells the system which interpreter to use (in this case, the `bash` shell).

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

2. **Comments:** Lines starting with `#` are comments, which are ignored by the script but help in explaining the code.

```
1 # This is a comment
```

3. **Commands:** After the shebang and any comments, you can write the actual commands that you want the script to execute.

```
1 echo "Hello, World!"
```

Example Shell Script: [🔗](#)

```
1 #!/bin/bash
2 # This is my first script
3 echo "Hello, World!"
4
```

Analogy: Writing a shell script is like writing a set of instructions on how to perform a task. For example, if you were to write a grocery list, each line would be a command to fetch a particular item.

How to Create Variables [↗](#)

Variables are used to store values that can be used later in the script. In shell scripting, you can declare a variable simply by assigning a value to it.

Syntax: [↗](#)

```
1 variable_name=value
```

- **No spaces** around the `=` sign.
- By default, shell variables are **global** within the script.
- To access the value of a variable, use the **dollar sign** (`$`) before the variable name.

Examples [↗](#)

String Variable:

```
1 name="Alice"
2 echo $name # Output: Alice
3
```

Numeric Variable:

```
1 age=25
2 echo $age # Output: 25
3
```

String and Numeric Variable: [↗](#)

```
1 #!/bin/bash
2 name="Alice" # Create a variable 'name' with value 'Alice'
3 age=25       # Create a variable 'age' with value 25
4 echo "Hello, $name! You are $age years old."
5
```

Special Characters or Spaces in Variables:

To include special characters or spaces in a variable, enclose the value in **quotes**.

```
1 greeting="Hello, World!"
2 echo $greeting # Output: Hello, World!
3
```

Reading Input Into Variables: [↗](#)

You can also ask the user for input and store it in a variable using `read`:

```
1 #!/bin/bash
2 echo "Enter your name: "
3 read name
4 echo "Hello, $name!"
5
```

Analogy: A variable is like a storage box where you can store a piece of information (like your name or age) for later use in your script.

★ Objective 5: Running Shell Script [↗](#)

To run a shell script:

1. **Make the script executable:** Use the `chmod` command to make the script executable.

```
1 chmod +x scriptname.sh
```

2. **Run the script:** Use `./` to execute the script.

```
1 ./scriptname.sh
```

★ Objective 6: Conditionals [↗](#)

Conditionals allow the script to make decisions based on certain conditions (similar to `if-else` statements in other programming languages).

Syntax of an If-Else Statement: [↗](#)

```
1 #!/bin/bash
2 if [ $1 -gt 10 ]; then
3     echo "Number is greater than 10"
4 elif [ $1 -eq 10 ]; then
5     echo "Number is equal to 10"
6 else
7     echo "Number is less than 10"
8 fi
9
```

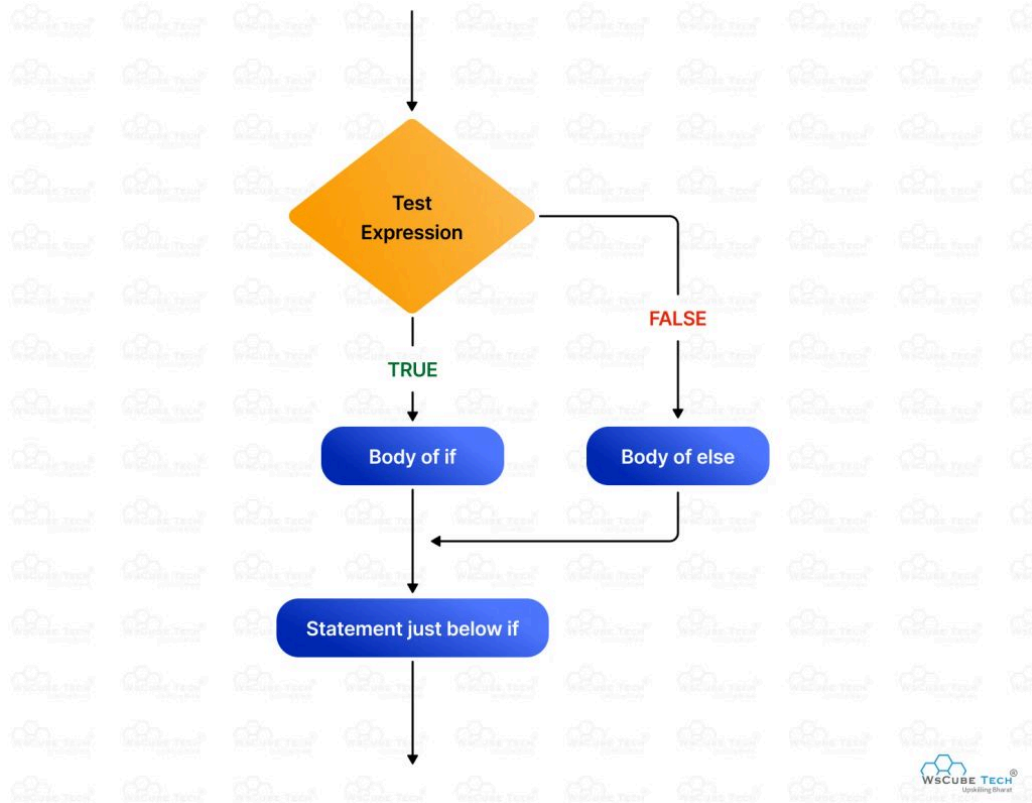
- `$1` is a **positional parameter**, meaning the first argument passed to the script.
- `elif`: The `elif` (else if) statement allows you to check multiple conditions in sequence. If the first `if` condition is false, the script checks the condition in `elif`.

Explanation: [↗](#)

- `-gt`: Greater than.
- `-eq`: Equal to.
- The script first checks the condition after `if`. If false, it checks the `elif`. If no conditions are met, the script runs the code under `else`.

Notice that in the **if statement** above we **indented** the commands on **line 3, 5** and **7** that will only run/executed if the preceding statement is evaluated to true. This is referred to as indenting and it is an important part of writing good (easy to read), clean code (in any language, not just shell scripts). The aim is to improve readability and reduce than chance for making mistakes. There aren't any rules regarding indenting in Bash, however, it is highly recommended that you indent your code, especially as your scripts get larger., otherwise, you will find it increasingly difficult to see the structure in your scripts.

Flowchart of the if-else Statement



Multiple Conditions [↗](#)

Sometimes we only want to do something if multiple conditions are met. Other times we would like to perform the action if one of several condition is met. We can accommodate these with **boolean operators**.

- **and** - &&
- **or** - ||

For instance maybe we only want to perform an operation if the file is readable **and** has a size greater than zero.

AND: [↗](#)

```
1 #!/bin/bash
2 # If the first argument (a file path) is readable and not empty
3 if [ -r $1 ] && [ -s $1 ]
4 then
5     echo This file is useful.
6 fi
```

Maybe we would like to perform something slightly different if the user is either bob or andy.

OR: [↗](#)

```
1 #!/bin/bash
2 # If user is either "bob" or "andy"
3 if [ $USER == 'bob' ] || [ $USER == 'andy' ]
4 then
5     ls -alh
6 else
7     ls
```

Nested If Statements [↗](#)

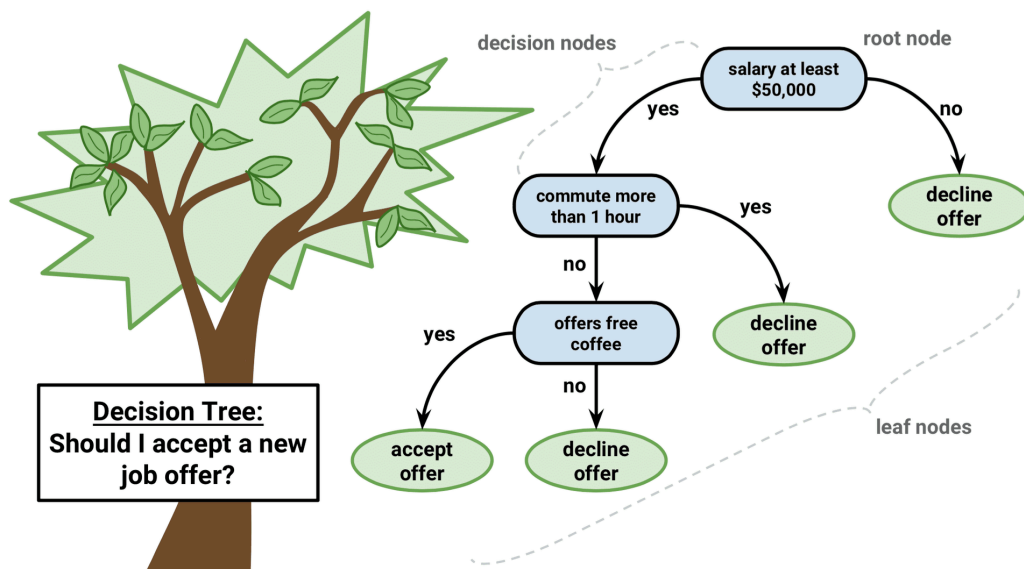
Here is a good example where having a consistent indentation will make it a lot easier to understand any written script. You may have as many **if statements** as necessary inside your script. It is also possible to have an if statement inside of another if statement (Avoid nesting if possible though. It will be discussed in class.). For example, we may want to analyze a number that was passed into our script on the command line which will be held in the `$1` variable:

```
1  #!/bin/bash
2  # Checking whether the number is greater than 100
3  if [ $1 -gt 100 ]
4  then
5      echo Hey that's a large number.
6      # Checking whether the number is divisible by 2 (whether it's even)
7      if [ $1 % 2 == 0 ]
8      then
9          echo And is also an even number.
10     fi
11 fi
```

Let's break it down:

- **Line 3** - Perform the following, only if the first command line argument is greater than 100.
- **Line 7** - This is a nested if (meaning, inside of another if statement).
- **Line 9** - Is executed, if and only if, **both** if statements (**line 4** and **8**) are true.

You can nest as many if statements as you like but as a general rule of thumb if you need to nest more than 3 levels deep you should probably need to revisit your logic to reduce nesting.



A decision tree that could be represented by if-elif-else statements

★ Objective 7: Comparison and the “test” Command [↗](#)

The `test` command evaluates expressions to determine if they are true or false. It is commonly used with conditional statements.

Using `test` Command and its Alias `[]` : [↗](#)

```
1 #!/bin/bash
2 if test $1 -gt 10; then
3     echo "The number is greater than 10."
4 else
5     echo "The number is 10 or less."
6 fi
7
```

Alias of `test` : [↗](#)

`[<expression>]` is an alias for `test` . So, the following is also valid:

```
1 #!/bin/bash
2 if [ $1 -gt 10 ]; then
3     echo "The number is greater than 10."
4 else
5     echo "The number is 10 or less."
6 fi
7
```

Comparison Operators: [↗](#)

- **String Comparison:**

- `=` , `==` : Equal
- `!=` : Not equal
- `<` , `>` : Less than, greater than (use within double quotes).

- **Integer Comparison:**

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

- **File Comparison:**

- `-e` : Exists
- `-f` : Is a regular file
- `-d` : Is a directory
- `-r` , `-w` , `-x` : Read, write, execute permissions

★ Objective 8: Quoting [↗](#)

Single quotes and double quotes are both functional in Linux while working with shell scripts or executing commands directly in the terminal but there is a difference between the way the bash shell interprets them.

Single quotes:

Enclosing characters in single quotation marks (`' '`) holds onto the literal value of each character within the quotes. This is convenient when you do not want to use the escape characters to change the way the bash interprets the input string.

Double quotes:

Double quotes (") are similar to single quotes except that it allows the shell to interpret dollar sign (\$), backtick (`), backslash (\) and exclamation mark (!). The characters have special meaning when used with double quotes, and before display, they are evaluated. A double quote may be used within double quotes by preceding it with a backslash.

Example [↗](#)

```
1 test=10
2 echo "$test"
3 echo '$test'
```

```
[mishra@localhost ~]$ test=10
[mishra@localhost ~]$ echo "$test"
10
[mishra@localhost ~]$ echo '$test'
$test
[mishra@localhost ~]$
```

★ Objective 9: Exit Status and Special Variables in Shell [↗](#)

- Every command returns an **exit status**.
 - 0 means success.
 - Non-zero means failure.

Special Variables: [↗](#)

- \$?: The exit status of the last command.

```
1 echo "Hello!"
2 echo $?
3
```

- 0 means successful execution.

- \$0: Name of the script.

```
1 echo "The name of the script is $0."
2
```

- \$1, \$2, ...: Positional parameters (arguments passed to the script).

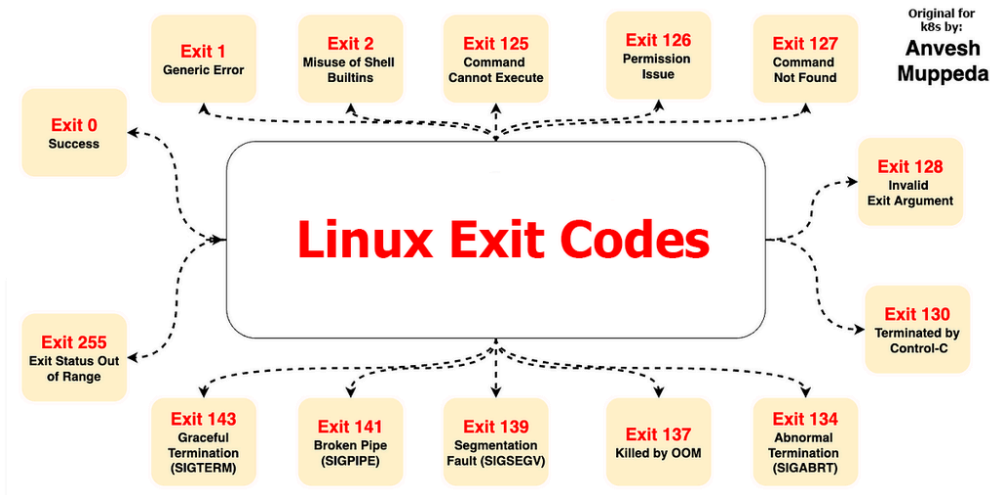
```
1 echo "The first argument is $1."
2 echo "The second argument is $2."
3
```

- \$#: Number of arguments passed to the script.

```
1 echo "Number of arguments: $#"
```

- \$@ and \$*: All arguments passed to the script.

```
1 echo "All arguments: $@"
2
```



★ Objective 10: Different Types of Shells






Bash is the most popular shell. It runs more or less everywhere and is installed on almost every UNIX system. Most of the time, choosing Bash as your scripting language is a safe choice, and it's what we will be using in this module.

Then there's the **Z shell**. Recently, Apple made this the default shell for macOS. In a sense, the Z shell is a modern alternative to Bash with some nice and nifty features. So there's some nice extras, and that makes it nice for scripting. But it's not so useful if you want to share with others because the Z shell is not in the default install for many systems.

Then there's **sh** or **the Bourne shell**. This is the most portable of the three, and you can consider it the standard UNIX shell in some way. This shell has existed longer than both Bash and the Z shell, and it is much more standard, present on basically every UNIX system you will find. It has a lot less features than newer shells, and generally, coding for sh is a bit more complex. You would write scripts for sh if you need to be sure it runs everywhere, including very minimal Linux systems or some other more rare UNIX systems.

TYPES OF SHELL IN LINUX



-  **BASH:** BOURNE AGAIN SHELL IS THE DEFAULT FOR MOST OF THE LINUX DISTRIBUTIONS
-  **KSH:** KORN SHELL IS A HIGH-LEVEL PROGRAMMING LANGUAGE SHELL
-  **CSH:** C SHELL FOLLOWS C LIKE SYNTAX AND PROVIDES SPELLING CORRECTION AND JOB CONTROL
-  **ZSH:** Z SHELL PROVIDES SOME UNIQUE FEATURES SUCH AS FILENAME GENERATION, STARTUP FILES, LOGIN/LOGOUT WATCHING, CLOSING COMMENTS ETC.
-  **FISH:** FRIENDLY INTERACTIVE SHELL PROVIDES SOME SPECIAL FEATURES LIKE WEB-BASED CONFIGURATION, AUTO-SUGGESTIONS, FULLY SCRIPTABLE WITH CLEAN SCRIPTS



RTK

★ Objective 11: Practice Questions [↗](#)

1. Write a script that asks for the user's name and age and prints out a greeting message.
 2. Write a script that checks if a file exists, and prints a message if it does or doesn't.
 3. Modify the script to accept the file name as an argument.
 4. Write a script that checks whether a number is positive, negative, or zero. Use `if`, `elif`, and `else` in the script.
 5. Use `test` or `[]` to write a script that checks if a directory exists. If not, create it.
 6. Write a script that takes a filename as an argument and checks if it is a regular file, directory, or doesn't exist at all.
-