HyperionDev

Shell Scripting

Task

Visit our website

Introduction

As you know, the command line is useful for several tasks, including: installation, version control, automation, remote access, and advanced control.

In this lesson, we will introduce how the command line is used in writing scripts to automate tasks. For instance, if you are responsible for setting up multiple machines at your company with the same software, you may want to automate software installations using script files.

If you would like to learn more about common commands or command line basics, please refer to the additional reading: **Installation, Sharing, and Collaboration**.

Script files

As you advance in your skills as a developer, you may at times find that there are certain commands that you continually use. Instead of retyping these commands into the command line repeatedly, you can create a script file that contains these sets of commands. Scripts are plain text files that can be executed as needed, automating tasks and saving time. Often such files will be executed periodically, e.g., daily, weekly, monthly, etc.

We will look at variables and conditional statements in this task, but scripting languages have many more capabilities.

In Windows, we can create PowerShell script files, and in macOS and Linux systems we can create shell scripts. To learn about writing shell scripts appropriate to your operating system, read either the **Windows PowerShell scripts** section if you are using Windows, or the **Unix shell scripts** section if you are using a Linux or macOS operating system.

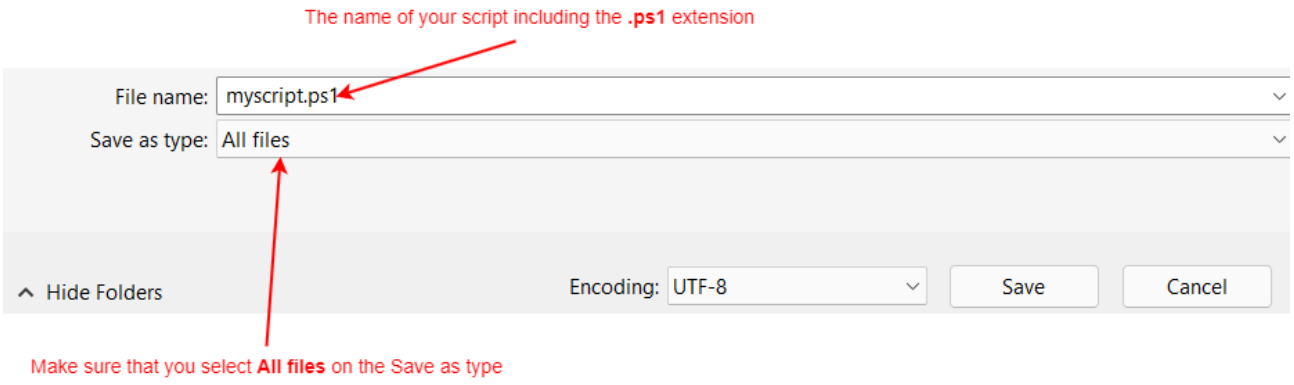
Windows PowerShell scripts

Creating and using PowerShell scripts

To create a PowerShell script file, you need to:

1. Open a plain text editor like Notepad.
2. Navigate to “File > Save As”, and input the desired name for your script file.

3. Append the **.ps1** extension, for example, "**myscript.ps1**", and make sure that you change "Save as type" to "All files". This signifies that the file contains PowerShell script code.



4. Write the set of commands in your file and save it. For example, to print "Hello, World!" to the terminal, write the following in your script file:

```
Write-Output "Hello, World!"
```

To execute a PowerShell script, you can use one of the following methods:

1. Type `.\<scriptname.ps1>` into the terminal and hit enter if you are in the same directory as the script file. For example:

```
.\myscript.ps1
```

2. You can change the directory by using the `cd` command and then execute the script using the command above. For example:

```
cd "C:\Users\user\Desktop\hyperion"
```

3. If you are not in the current directory, you need to use the call operator (`&`) and specify the full path to the script. For example:

```
& "C:\Users\user\Desktop\hyperion\myscript.ps1"
```

By default, PowerShell restricts script execution for security reasons, so you may need to change the execution policy to run your script. Open PowerShell as an administrator and use the following command to set the execution policy:

```
Set-ExecutionPolicy RemoteSigned
```

This command allows locally created scripts to run but requires that a trusted publisher sign scripts downloaded from the Internet.

Variables in PowerShell

PowerShell has “weakly typed” variables, i.e., it does not require you to explicitly mention different types of objects or variables. To declare a variable in a PowerShell script, you use `$variable_name=value`, and to get its value, you would then type `$variable_name`, as seen in the following code:

```
# Declaring a variable and assigning value to the variable
$fileName = "Scripting is awesome."

# Display the value stored in the variable
Write-Output $fileName
```

If you want to assign user input to a variable, you can use `$variable_name = Read-Host`, for example:

```
# Get user input using variables
$userName = Read-Host "Enter username "

# Notice how we have accessed the data on the variable $userName below and
combined it with a string data set.
Write-Output "Your username is $userName"
```

Conditional statements in PowerShell

Conditional statements in scripts allow you to execute different commands or code blocks based on certain conditions, providing decision-making capabilities within your script. These statements evaluate expressions and execute code blocks based on whether the expressions return true or false. By using conditional statements, you can make your scripts more dynamic and responsive to different scenarios, enhancing their functionality and robustness. We will look at the basic syntax of conditional statements next.

Conditional blocks in PowerShell scripting can take one of four formats:

1. `if` blocks:

```
if (condition){
  statement
}
```

2. `if/else` blocks:

```
if (condition) {
  statement
}
else {
  default
}
```

3. `if/elif/else` blocks:

```
if (condition) {
  statement
}
elseif (condition){
  statement
}
else {
  default
}
```

4. Nested `if/else` blocks:

```
if (condition) {
  statement
}
else {
  if (condition) {
    statement
  }
  else {
    statement
  }
}
```

In the example above, curly brackets `{}` represent the body of the `if` statement, and the condition of the `if` statement is inside the brackets `()`. If the condition in the brackets is true then the statement in the curly brackets will be executed.

Within the condition, we can use comparison operators like:

- `-gt` to represent “greater than”.
- `-lt` to represent “less than”.
- `-eq` to represent “equality”.
- `-le` to represent ‘’less than or equal to”.

You can learn more about additional comparison operators and their uses in **this comprehensive guide**.

You can also combine multiple conditions using the:

- AND operator, `-a`, which requires both conditions to be true, and
- OR operator, `-o`, which requires at least one of the conditions to be true.

Let's look at an example script that checks if a file named **example.txt** exists in the current directory. If it exists, it outputs a message saying the file exists. If not, it creates the file and then outputs a message saying it has been created.

```
# Check if the file exists
if (Test-Path -Path "example.txt") {
    # If the file exists, print a message
    Write-Output "The file 'example.txt' already exists."
} else {
    # If the file does not exist, create it and print a message
    New-Item -ItemType File -Name "example.txt"
    Write-Output "The file 'example.txt' has been created."
}
```

Let's look at another example to demonstrate the use of the `if`, `if-else`, `if-elif-else`, and nested `if-else` statements.

```
# Create a menu and allow the user to make a choice
$menu = Read-Host  "Select an option below by entering a number:`n1 - list
directories and files in the current directory`n2 - Create a new file in the
current directory`n3 - Create a new folder in the current directory.`n"

if ( $menu -eq 1){
    # List all the directories
    ls
}
elseif ($menu -eq 2){
    # Get the name of the file from the user and create the file in the
folder
    $file = Read-Host "Enter the name of the file that you want to create"

    # An if statement is created inside the body of the if statement (nested
if statement)
    # Check if the file exists before creating it
    if (Test-Path -Path $file){
        Write-Output "The file named $file already exists"
    }
    else {
```

```
    # Create a new file
        New-Item  $file
        Write-Output "File with the name $file has been created"
    }
}
elseif ($menu -eq 3){
    # Get the name of the file from the user and create the file in the
folder
    $folder = Read-Host "Enter the name of the folder that you want to
create"

    # An if statement is created inside the body of the if statement (nested
if statement)
    # Check if the folder exists before creating it
    if (Test-Path -Path $folder){
        Write-Output "The folder named $folder already exists"
    }
    else {
        # Create a new folder
        New-Item -Path $folder -ItemType Directory
        Write-Output "Folder with the name $folder has been created"
    }
}
else {
    Write-Output "You have made an invalid choice."
}
```

In the PowerShell program above, the user is presented with a menu and asked to make a choice. If the user enters `1`, the condition `if ($menu -eq 1)` becomes true, and the script lists all files and directories in the current directory using the `ls` command. If `2` is entered, the condition `elseif ($menu -eq 2)` becomes true, prompting the user to enter a file name. Here, the program checks if a file with the given name already exists using the `Test-Path` cmdlet. If the file exists, a message informs the user that it already exists; otherwise, the `New-Item` cmdlet creates the file. This demonstrates a nested `if` statement, where one `if` statement is placed within another to ensure precise logic handling.

This process is repeated if the user enters `3`, but in this case, the program requests a folder name. Similar to file creation, `Test-Path` checks if the folder already exists, and if not, `New-Item -ItemType Directory` creates the folder. For any input other than `1`, `2`, or `3`, the `else` block displays an error message informing the user of an invalid choice. The `Read-Host` cmdlet, used to capture user input, and conditionals (`if`, `elseif`, `else`) control the program flow based on the input provided. This ensures smooth script execution and user-friendly feedback.

Copy the code into your terminal to see what happens with each option.

Unix shell scripts

Shebangs

Every file in a Unix file system has a set of permissions. One of these permissions is the execute permission. If you want to make a file executable from the terminal, you will need to give it the execute permission by running the following command to change the mode to executable:

```
$ chmod +x [file name]
```

Now that you have an executable file, you need to give the shell interpreter a heads-up of what to run the file with. This is done in the first line of the file with a special line called a shebang.

If you were writing a Bash script, your shebang would be:

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

When you execute your script, the shell will know that the script must be run with the program you've specified in the shebang, i.e., Bash.

To create a shell script:

1. Open a text editor (e.g., gedit).
2. Add `#!/bin/bash` to the first line of the script file, like so:

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

3. On the following lines enter the instructions that you would usually type into the terminal, one line per instruction. For example, to print "Hello, World!" to the terminal write the following in your script file:

   ```
   #!/bin/bash
   echo "Hello, World!"
   ```

4. Save the file. It is not a requirement, but it's common practice to save your file with a **.sh** extension. To save the file properly, you may need to specify that the file is a plain text file. Do this by selecting "Format > Make plain text".
5. Make this file executable by typing `chmod +x <scriptname.sh>` into the command line. For example:

   ```
   chmod +x myscript.sh
   ```

6. To run the script, type `sh myscript.sh`, where **myscript.sh** is the name of the script file.

To execute a Bash script you can use one of the following methods:

1. Type `./<scriptname.sh>` into the terminal and hit enter if you are in the same directory as the script file. For example:

```
./myscript.sh
```

2. If you are not in the current directory, you need to specify the full path to the script. For example:

```
/home/user/hyperion/myscript.sh
```

Variables in Bash

Bash has "weakly typed" variables, i.e., it does not require you to explicitly mention different types of objects or variables. To declare a variable in a Bash script, you type `variable_name=value`, and to get its value, you would type `$variable_name`, as in the following code example:

```
#!/bin/bash

fact="Linux is awesome!"
echo "Fact:" $fact
```

Output:

```
vin@localhost:~/Documents/bash scripting> ./variables.sh
Fact: Linux is awesome!
vin@localhost:~Documnets/bash scripting>
```

If you want to assign user input to a variable, you can use `read variable_name`:

```
#!/bin/bash

echo "Type in a fact: "
read fact
echo "Fact:" $fact
```

Output:

```
vin@localhost:~/Documents/bash scripting> ./variables.sh
Type in a fact:
Linux is awesome!
Fact: Linux is awesome!
vin@localhost:~Documnets/bash scripting>
```

Conditional statements in Bash

Conditional statements in scripts allow you to execute different commands or code blocks based on certain conditions, providing decision-making capabilities within your script. These statements evaluate expressions and execute code blocks based on whether the expressions return true or false. By using conditional statements, you can make your scripts more dynamic and responsive to different scenarios, enhancing their functionality and robustness. We'll look at the basic syntax of conditional statements next.

Conditional blocks in Bash scripting can take one of four formats:

1. `if` blocks:

```
if [[ condition ]]; then
  statement
fi
```

2. `if/else` blocks:

```
if [[ condition ]]; then
  statement
else
  default
fi
```

3. `if/elif/else` blocks:

```
if [[ condition ]]; then
  statement
elif [[ condition ]]; then
  statement
else
  default
fi
```

4. Nested `if/else` blocks:

```
if [[ condition ]]; then
  statement
else
  if [[ condition ]]; then
    statement
  else
    statement
  fi
fi
```

In these examples, `fi` serves as the ending delimiter for the `if` statement, indicating the conclusion of the `if` block. Recalling the overall structure of an `if` block from above:

```
if [[ condition ]]; then
  statement
fi
```

- `if` initiates the conditional block.
- `then` introduces the statement to be executed if the condition is true.
- `fi` closes the `if` statement.

Within the condition, we can use the following operators:

- `-gt` to represent “greater than”.
- `-lt` to represent “less than”.
- `-eq` to represent “equality”.
- -le to represent ‘’less than or equal to”.

To explore additional comparison operators and their uses, you can learn more in **this comprehensive guide**.

You can combine multiple conditions using the:

- AND operator, `-a`, which requires both conditions to be true, and
- OR operator, `-o`, which requires at least one of the conditions to be true.

Let's look at a simple example program written as a Bash script. If we wanted to write a program that allows a user to choose whether they want to list directories and files in the current directories, create a new file, or create a new folder, the program would look as follows:

```
#!/bin/bash

# Create a menu and allow the user to make a choice
echo  -e "Select an option below by entering a number:\n1 - list directories
and files in the current directory\n2 - Create a new file in the current
directory\n3 - Create a new folder in the current directory.\n"
read menu

if [ $menu -eq 1 ]; then
    # List the directories and files in the current directory
    ls
elif [ $menu -eq 2 ]; then
    # Get the name of the file from the user and create the file in the
folder
    echo "Enter the name of the file that you want to create"
    read file

    # An if statement is created inside the body of the if statement (nested
if statement)
    # Check if the file exists before creating it
    if [ -f $file ]; then
        echo "The file named $file already exists"
    else
    # Create a new file
      touch $file
      echo "File with the name $file has been created"
    fi
elif [ $menu -eq 3 ]; then
    # Get the name of the file from the user and create the file in the
folder
    echo "Enter the name of the folder that you want to create"
    read folder

    # An if statement is created inside the body of the if statement (nested
if statement)
    # Check if the folder exists before creating it
    if [ -d $folder ]; then
        echo "The folder named $folder already exists"

    else
        # Create a new folder
```

```
        mkdir $folder
        echo "Folder with the name $folder is created"
    fi
else
    # If the user enters a number that is not 1,2, or 3
    echo "You have made an invalid choice."
fi
```

In the program above we demonstrated the use of **`if-elif-else`**, nested **`if`** statements, and **`if-else`** statements.

Recall that the first line, **`#!/bin/bash`**, is called a shebang, which specifies the path to the Bash interpreter, ensuring that the script runs using Bash.

The script begins by printing the following to the terminal:

```
Select an option below by entering a number:
1 - list directories and files in the current directory
2 - Create a new file in the current directory
3 - Create a new folder in the current directory.
```

The next line, **`read menu`**, waits for user input and assigns the entered value to the variable **`menu`**.

Next, the script checks the value of the variable **`menu`** with an **`if`** statement: **`if [ $menu -eq 1 ]; then`**. This condition evaluates whether the value of the variable **`menu`** is equal to **`1`** (with **`-eq`** meaning "equal to"). If the condition is true, the script will list the directories and files in the current directory to the console.

If the condition is false, the script moves to the next **`elif [ $menu -eq 2 ]`** block. If the value of the variable menu is equal to 2, then the script will request the name of the new file that the user wants to create. The nested **`if-else`** statement will first check if the file that the user wants to create is already created or not, and if the file already exists then the program will display the message: "**`The file named [name of the file] already exists`**". Otherwise, it will create the file in the current directory.

If the condition is false, the script moves to the next **`elif [ $menu -eq 3 ]`** block. If the value of the variable menu is equal to 3, then the script will request the name of the new folder that the user wants to create. The nested **`if-else`** statement will first check if the folder that the user wants to create exists or not, and if the folder already exists then the program will display the message: "**`The folder named [name of the folder] already exists`**". Otherwise, it will create the folder in the current directory.

Any value assigned to the variable menu except 1, 2, or 3 will lead to the **`else`** section of the code, which will display the message: "**`You have made an invalid choice.`**"

Copy the code into your terminal to see for yourself what happens when you choose different options.

Instructions

First, read the accompanying example files, which should help you understand scripting. You may execute the examples relevant to your operating system to see the output. Feel free to also write and run your own examples before doing the practical task to become more comfortable with scripts.

Practical task

Follow these steps:

1. Create a script file called **file_cd** (remember to save this file with a **.ps1** extension if you are using Windows).
    - Inside **file_cd**, insert commands to create three new folders (directories). Name your folders whatever you'd like.
    - Next, insert commands to navigate inside one of the folders you created and create three new folders inside this folder. Also, insert commands to remove two of the folders you created.
2. Create another file called **ifExample** (again, remember to save this file with a **.ps1** extension if you are using Windows).
3. Inside the **ifExample** file do the following:
    - Write an `if` statement to create a new folder named **if_folder** if a folder named **new_folder** already exists.
    - Within the same file, write an `if-else` statement to check whether a folder named **if_folder** exists. If it does, create a new folder named **hyperionDev** otherwise, create a new folder named **new-projects**.[Practical task introduction and context]

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.