Bash if elif else Statement: A Comprehensive Tutorial

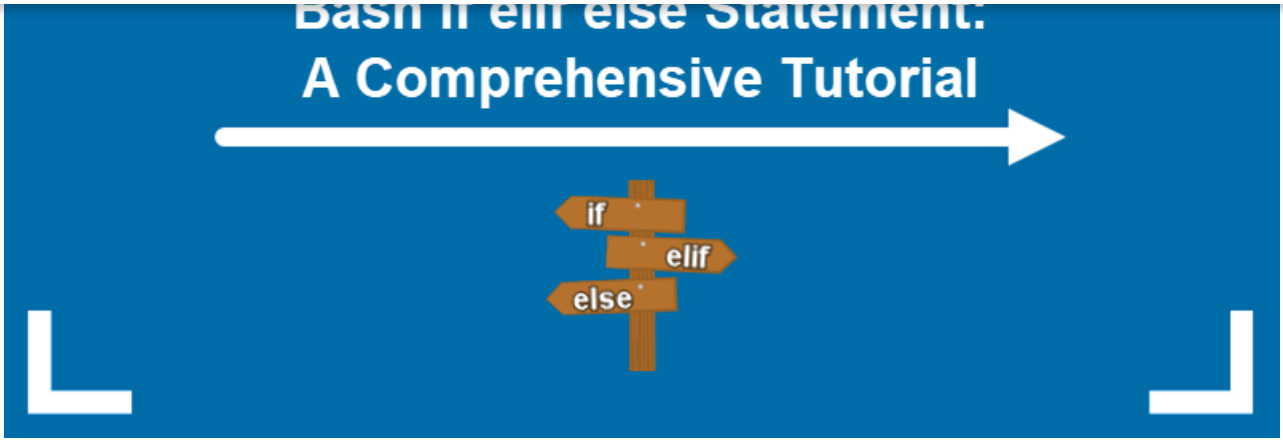
October 21, 2021

BASH LINUX

Introduction

Bash scripts help automate tasks on your machine. The **`if elif else`** statement in bash scripts allows creating conditional cases and responses to specific code results. The **`if`** conditional helps automate a decision-making process during a program.

This article explains what the `if elif else` statement is and shows the syntax through various examples.



Prerequisites

- A machine with Linux OS.
- Access to the command line/terminal.
- Access to a text editor like Vi/Vim.

What is the Bash if Statement?

In programming, the `if` statement is a conditional expression. However, the command tested in the `if` statement evaluates based on the **exit status**. Therefore:

- If the command completes **successfully**, the exit status is **0**.
- If the statement throws an **error**, the exit status is any number between **1** and **255**.

The zero for success and any non-zero value for failure seems counterintuitive. In most other programming languages, zero represents false, and one (or greater) represents true. However, in bash scripting, the UNIX convention returns the exit status instead of a truth value, and the two should not be confused.

Test a sample error command (1 greater than 100) in the terminal by running:

```
test 1 -gt 100
```

Check the exit status using the echo command:

```
echo $?
```

The test returns an exit code **1**, indicating the expression failed.

Similarly, check a sample command that evaluates successfully (1000 greater than 100) in the terminal with:

```
test 1000 -gt 100
```

Print the exit status to confirm the command was successful:

```
echo $?
```

```
kb@phoenixNAP:~$ test 1000 -gt 100
kb@phoenixNAP:~$ echo $?
0
kb@phoenixNAP:~$
```

The test returns an exit code **0**, showing the command completed without an error.

Bash if Statement Example

Follow the instructions below to create an example bash script using an **if** statement.

1. Open the terminal (**CTRL+ALT+T**) and create an example script to test how the bash **if** statement works:

```
vi test_script.sh
```

2. In the script, add the following lines:

```
echo -n "Please enter a whole number: "
read VAR
echo Your number is $VAR
if test $VAR -gt 100
then
        echo "It's greater than 100"
```

```
1 echo -n "Please enter a whole number: "
2 read VAR
3 echo Your number is $VAR
4 if test $VAR -gt 100
5 then
6         echo "It's greater than 100"
7 fi
8 echo Bye!
```

Each line in the script does the following:

- **Lines 1-3** provide instructions to enter a number through the console. The number is read into a variable called **VAR** and printed.
- **Line 4** starts the **`if`** statement and checks the exit status for the command right after (**`$VAR -gt 100`**).
- **Lines 5-6** signals the start of commands to execute only if the statement in line 4 completes successfully (with an exit status 0), meaning we entered a number greater than 100.
- **Line 7** signals the end of the **`if`** statement.
- **Line 8** is outside of the statement and runs as expected, regardless of the **`if`** outcome.

3. Save and close Vim:

```
:wq
```

4. Next, make the file executable:

```
chmod +x test_script.sh
```

5. Lastly, run the script with:

```
./test_script.sh
```

```
It's greater than 100
Bye!
kb@phoenixNAP:~$
```

The script outputs a different message based on the entered number. Run the script multiple times and test for other numbers to confirm the behavior.

Bash if Statement Syntax

The basic syntax for a bash **if** statement is:

```
if <commands>
then
        <commands>
fi
```

Each keyword has a specific function:

- **if** signals the statement's beginning. The command right after is the one in which the exit status check applies.
- **then** executes the commands only if the previous review completes successfully.
- **fi** closes the if statement.

Enclosing the test command in different brackets results in different execution methods for the **if** statement. The table below provides a short description as well as a use case for each bracket type.

Syntax	What it is	When to use
if (<commands>)	Subshell executed in a subprocess.	When the commands affect the current shell or environment. The changes do not remain when the subshell completes.
if ((<commands>))	Bash extension.	Use for arithmetic operations and C-style variable manipulation.
if [<commands>]	POSIX builtin, alias for **test <commands>**.	Comparing numbers and testing whether a file exists.

Below are example bash scripts that use each bracket type with a more in-depth explanation.

Single-Parentheses Syntax

Using single parentheses in bash scripting creates a subshell. When combined with the **if** statement, the subprocess finishes before continuing the program. The **if** analyzes the exit status and acts accordingly.

The bash **if** statement with single parentheses syntax looks like the following:

```
if ( <commands> )
then
        <commands>
fi
```

Try the example below to see how the sub-process behaves together with the **if** statement:

1. Create the script using Vim:

```
vi single_parentheses.sh
```

2. Add the following lines of code to the script:

```
outer_variable=Defined
echo Before if:
echo inner_variable = $inner_variable
echo outer_variable = $outer_variable
if (
        echo Inside subshell:
        inner_variable=Defined
        echo inner_variable = $inner_variable
```

```
        echo After then:
        echo inner_variable = $iner_variable
        echo outer_variable = $outer_variable
fi
echo After fi:
echo inner_variable = $inner_variable
echo outer_variable = $outer_variable
```

```
1 outer_variable=Defined
2 echo Before if:
3 echo inner_variable = $inner_variable
4 echo outer_variable = $outer_variable
5 if (
6         echo Inside subshell:
7         inner_variable=Defined
8         echo inner_variable = $inner_variable
9         outer_variable=Changed
10        echo outer_variable = $outer_variable
11 )
12 then
13        echo After then:
14        echo inner_variable = $iner_variable
15        echo outer_variable = $outer_variable
16 fi
17 echo After fi:
18 echo inner_variable = $inner_variable
19 echo outer_variable = $outer_variable
```

The program does the following:

- **Line 1** creates a variable called **`outer_variable`** in which we store a string **`Defined`**.
- **Lines 2-4** print the variables to the console. At this moment, **`outer_variable`** has a string stored in it, while **`inner_variable`** is blank.
- **Line 5** starts the **`if`** statement and a sub-process, delimited by single parentheses.
- **Line 6-11** store a string inside the **`inner_variable`** and change the **`outer_variable`** to a different string. Both values print to the console, and the sub-process ends with an exit code. In this case, the sub-process ends successfully with an exit code **`0`**.
- **Line 12-16** execute after the sub-process and print the variable values. However, the values change back to what they were before the **`if`** statement. The sub-process only stores the values locally and not globally.
- **Lines 16-19** run after the commands in the **`then`** clause. The values remain unchanged outside the statement.

3. Save the script and close the editor:

```
chmod +x single_parentheses.sh
```

5. Lastly, run the example to test the results:

```
./single_parentheses.sh
```

```
kb@phoenixNAP:~$ ./single_parentheses.sh
Before if:
inner_variable =
outer_variable = Defined
Inside subshell:
inner_variable = Defined
outer_variable = Changed
After then:
inner_variable =
outer_variable = Defined
After fi:
inner_variable =
outer_variable = Defined
kb@phoenixNAP:~$
```

The output prints the variable states as the program progresses.

Double-Parentheses Syntax

The double-parentheses syntax for a bash **`if`** statement is:

```
if (( <commands> ))
then
        <commands>
fi
```

The double parentheses construct in bash allows:

- **Arithmetic evaluation**. Defining a variable as **`a=$(( 1+1 ))`** calculates the equation and sets **a** to 2.
- **C-style variable manipulation.** For example, incrementing variables with **`(( a++ ))`**.

Note: Double parentheses are analogous to most other programming languages, where zero is false and one is true.

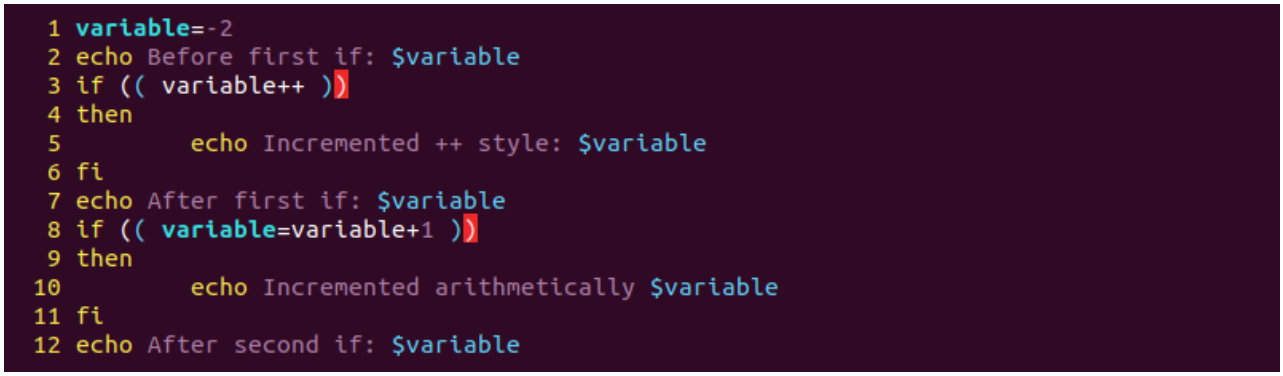
Try the following example to see how double parentheses work:

1. Create a bash script in the terminal:

```
vi double_parentheses.sh
```

2. Add the following code to *double_parentheses.sh*:

```
variable=-2
echo Before first if: $variable
if (( variable++ ))
then
        echo Incremented ++ style: $variable
fi
echo After first if: $variable
if (( variable=variable+1 ))
then
        echo Incremented arithmetically $variable
fi
echo After second if: $variable
```



Each line number in the script does the following:

a message if the variable is not zero.

3. Save the script and close Vim:

```
:wq
```

4. Change script permissions to executable:

```
chmod +x double_parentheses.sh
```

5. Run the script to see the results:

```
./double_parentheses.sh
```

```
kb@phoenixNAP:~$ ./double_parentheses.sh
Before first if: -2
Incremented ++ style: -1
After first if: -1
After second if: 0
kb@phoenixNAP:~$
```

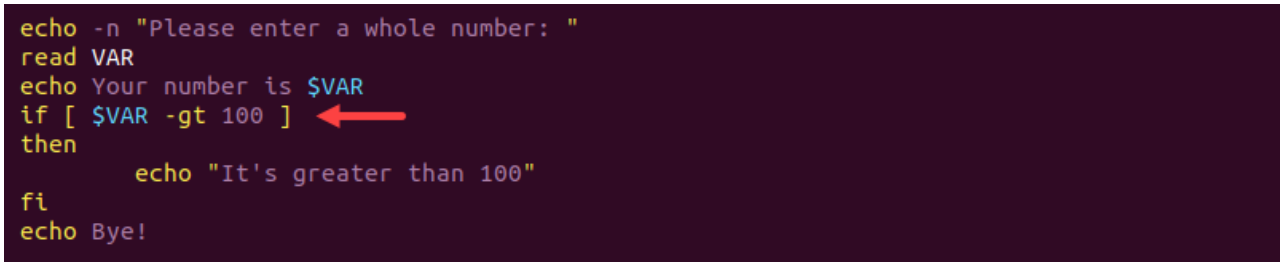
Single-Bracket Syntax

The single bracket is another name for the **`test`** command and a standard POSIX utility available for all shells. The basic syntax is:

```
if [ <commands> ]
then
        <commands>
fi
```

The first bash **`if`** example provided in this tutorial (*test_script.sh*) works equally well with the alternative syntax:

```
if [ $VAR -gt 100 ]
then
        echo "It's greater than 100"
fi
echo Bye!
```



Run the script to confirm the output is equivalent. For the complete documentation and details on using bracket syntax, run the man command on the **`test`** utility:

```
man test
```

Double-Bracket Syntax

The double-bracket syntax in bash **`if`** scripts is the best option if portability is not necessary. The double-brackets are superior to single-brackets and include many advanced options. The syntax is:

```
if [[ <commands> ]]
then
        <commands>
fi
```

Try the example below to see how wildcard string matching works in an **`if`** command:

1. Create a shell script file called *double_brackets*:

```
vi double_brackets.sh
```

```
then
        echo Hello $USER
fi
echo Bye!
```

```
if [[ $USER == k* ]]
then
        echo Hello $USER
fi
echo Bye!
```

3. The script checks if the starting letter of the username is **k** and sends a hello message if it is. Save and close the script:

```
:wq
```

4. Make the file executable with chmod:

```
chmod +x double_brackets.sh
```

5. Run the program with:

```
./double_brackets.sh
```

```
kb@phoenixNAP:~$ ./double_brackets.sh
Hello kb
Bye!
kb@phoenixNAP:~$
```

Other Types of Bash Conditional Statements

The **if** statement only performs one conditional check. Modify the **if** with other types of bash conditionals to create complex assessments.

```
if <command>
then
        <commands>
else
        <commands>
fi
```

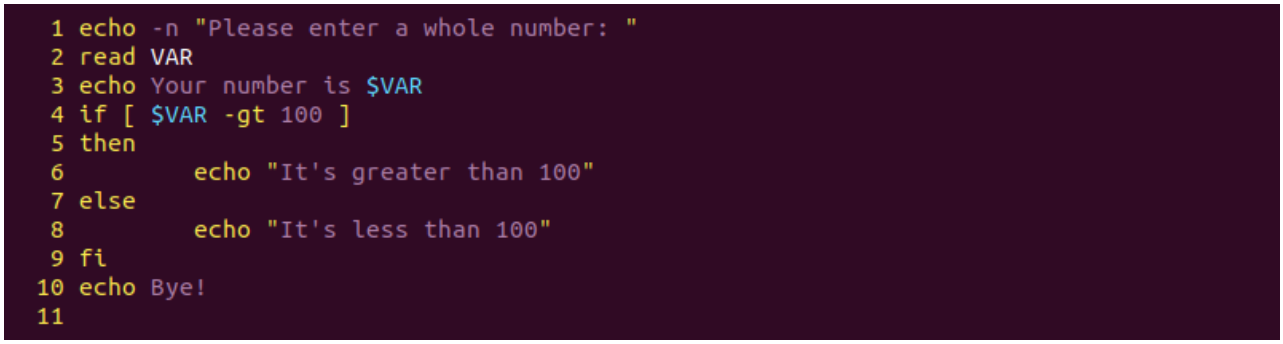
The following example demonstrates how the **`if else`** conditional works:

1. Create a new script using Vim:

```
vi if_else.sh
```

2. Insert the following code into the script:

```
echo -n "Please enter a whole number: "
read VAR
echo Your number is $VAR
if [ $VAR -gt 100 ]
then
        echo "It's greater than 100"
else
        echo "It's less than 100"
fi
echo Bye!
```



message.

- If the number is **less than 100**, the message in the **`else`** clause (**line 8**) prints to the console.

3. Save the script and close Vim:

```
:wq
```

4. Make the script executable:

```
chmod +x if_else.sh
```

5. Lastly, run the script multiple times and test for various values:

```
./if_else.sh
```

```
kb@phoenixNAP:~$ ./if_else.sh
Please enter a whole number: 10
Your number is 10
It's less than 100
Bye!
kb@phoenixNAP:~$ ./if_else.sh
Please enter a whole number: 200
Your number is 200
It's greater than 100
Bye!
kb@phoenixNAP:~$
```

if elif Statement

The **`elif`** clause combined with the **`if else`** statement creates multiple conditional checks. The **`if elif`** creates a series of checks with different results. The syntax is:

```
if <command>
then
        <commands>
elif <command>
then
```

To create a script using **`elif`**:

1. Create a shell file named *elif*:

```
vi elif.sh
```

2. In the *elif.sh* file, add the following example code:

```
echo -n "Please enter a whole number: "
read VAR
echo Your number is $VAR
if [ $VAR -gt 100 ]
then
        echo "It's greater than 100"
elif [ $VAR -lt 100 ]
then
        echo "It's less than 100"
else
        echo "It's exactly 100"
fi
echo Bye!
```

```
 1 echo -n "Please enter a whole number: "
 2 read VAR
 3 echo Your number is $VAR
 4 if [ $VAR -gt 100 ]
 5 then
 6         echo "It's greater than 100"
 7 elif [ $VAR -lt 100 ]
 8 then
 9         echo "It's less than 100"
10 else
11         echo "It's exactly 100"
12 fi
13 echo Bye!
14
```

The example adds an **`elif`** check on **line 7** to see if the entered number is less than 100. If the statements in lines 4 and 7 both fail, the program jumps to the else clause.

4. Make the *elif.sh* file executable:

```
chmod +x elif.sh
```

5. Run the script multiple times and check the behavior for different numbers:

```
./elif.sh
```

```
kb@phoenixNAP:~$ ./elif.sh
Please enter a whole number: 1
Your number is 1
It's less than 100
Bye!
kb@phoenixNAP:~$ ./elif.sh
Please enter a whole number: 101
Your number is 101
It's greater than 100
Bye!
kb@phoenixNAP:~$ ./elif.sh
Please enter a whole number: 100
Your number is 100
It's exactly 100
Bye!
kb@phoenixNAP:~$
```

Add multiple **elif** clauses to branch out the statement for further detailed checks. For instances where the **if** and **elif** pattern series grows, the better option is to use a **case** statement.

Nested if Statement

Nested **if** statements add a branch inside the **if**. Specifically, when a command passes the first if check, it goes through a new check to filter the result further. The syntax is:

```
if <commands>
then
        if <commands>
        then
                <commands>
```

The nested if is commonly used to search through multi-dimensional arrays. However, try to avoid having more than two or three nested `if` statements to reduce program complexity. Rethink the code's logic when the nested `if` keeps growing in depth.

Conclusion

Following this tutorial, you should know how to create an `if elif else` statement in a bash script and different syntaxes available. Next, check out how to implement the if statement to check if a file or directory exists in bash.

Milica Dancuk

Milica Dancuk is an aspiring technical writer at phoenixNAP and a data nerd. Her background in Electrical Engineering and Computing and her teaching experience give her a unique set of skills - being able to easily explain complex technical concepts through her content.

Next you should read

DevOps and Development, SysAdmin

Bash wait Command with Examples

September 23, 2021

Learn how to use the wait command through hands-on examples in bash scripts.

READ MORE

SysAdmin, Web Servers

How To Customize Bash Prompt in Linux

May 12, 2020

Follow this article to learn how to make changes to your BASH prompt. The guide shows how to edit the bashrc file, as well as how to modify PS1 variables. Use the information in this tutorial to change...

READ MORE

DevOps and Development, SysAdmin

How To Check If File or Directory Exists in Bash

August 30, 2019

Searching for specific files or directories can

process. This article is going to walk...

READ MORE

DevOps and Development, SysAdmin

Vim Commands Cheat Sheet

July 22, 2020

Vim is an open-source text editor found in most Linux distributions. This article lists all the commonly used Vim commands and includes a downloadable one-page cheat sheet for future reference.

READ MORE