19 Examples of Variables in Shell Script

Similar to every programming language, Bash also offers the concept of **Variables**. Variables in **shell scripting** are containers for storing necessary information. They specify memory locations in the system via characters or numeric or alphanumeric values. Values stored in these locations are later accessed and manipulated by referring to their Variable names. In shell scripting, reference to a variable is done by combining a variable name with the **dollar sign ($)** i.e. **$VARIABLE_NAME**.

Rules for Variables in Shell Scripting

Variables in **Shell Script** need to follow a set of rules. Otherwise, you may face runtime errors. These rules cover structures of accessing values stored in a variable, naming conventions of variables, defining the type of a variable, etc. Follow the list below to learn more about the rules of variables in **Shell scripting**.

- Use the equal sign (=) to assign values to variable names.
- Variable names are case sensitive i.e. ‘A’ and ‘a’ are different.
- To refer to a variable use the **dollar sign ($)** i.e. **$VARIABLE_NAME**.
- While updating/changing the variable values use only the variable name with the assignment operator(=) i.e. **VARIABLE_NAME= NEW_VALUE**.
- No need to define variable type while declaring variables.
- Enclose multiple words or string values within **Single Quote (' ')** to consider all characters as input.

Syntax for Variables in Shell Scripting

```
VARIABLE_NAME=VALUE
```

Shell Script Examples Using Variables

Variables are one of the primary concepts in **Shell Scripting**. Therefore, examples using variables are generally very basic. However, the application of variables cannot be avoided in the advanced scripts as well. For your convenience, I have listed the basic shell script examples below. Follow them to learn more about the applications of variables in **Shell Scripting**.

Example 1: Defining Variables in a Bash Script

In Bash Script, declare a variable by assigning(=) value to its reference. Furthermore, print the assigned values using **echo $(VARIABLE_NAME)**.

Code:

```
#!/bin/bash
# Declaration of variables
name=Tom
age=12
# Displaying variables
echo $name $age
```

Output:

```
Tom 12
```

Example 2: Read, Store and Display Input Variable

You can take user input with the **read** command and store it in a variable. Next, use **echo $(VARIABLE_NAME)** to print the user input.

Code:

```
#!/bin/bash
echo "Enter a number:"
```

```
read num
echo "The number is: $num"
```

Output:

```
Enter a number:
12
The number is: 12
```

Example 3: Reading User Input Variable with Prompt Message

The **read** command used with option **-p** allows you to prompt a message along with taking user input. You can use **echo $(VARIABLE_NAME)** to display the user input on the screen.

Code:

```
#!/bin/bash
read -p "Enter a number:" num
echo "The number is: $num"
```

Output:

```
Enter a number: 12
The number is: 12
```

Example 4: Concatenating Multiple Variables

You can concatenate multiple variables and store it into a single variable by enclosing them with a **double quotation (“ ”)**.

Code:

```
#!/bin/bash
# Declaration of variables
name='My name is Tom.'
age='My age is 12.'
# Concatenation
info="${name} ${age}"
echo "Result: $info"
```

Output:

```
Result: My name is Tom. My age is 12.
```

Example 5: Passing Values to Variables as Command Line Arguments

For passing values as command line arguments, you have to run the script along the values in a sequence. Later access these values using the **$** and input sequence number.

Code:

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

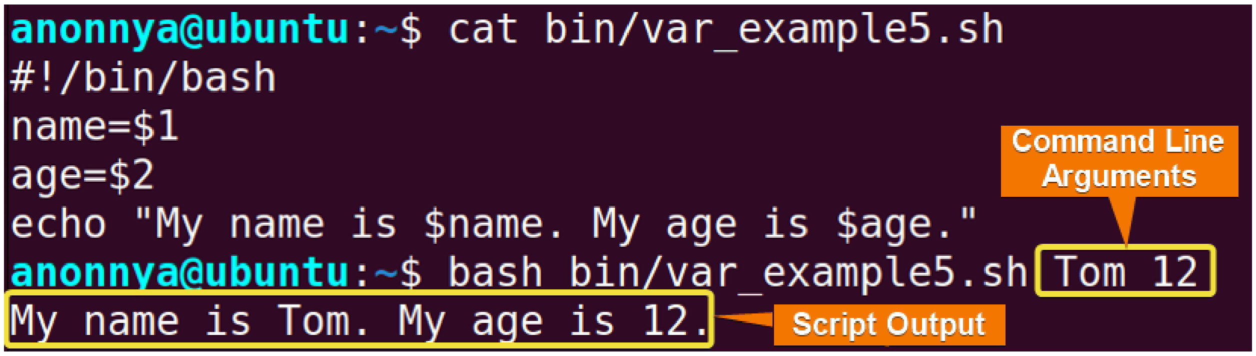
```
name=$1
age=$2
echo "My name is $name. My age is $age."
```

Syntax to run the Script:

```
bash bin/var_example5.sh Tom 12
```

Output:

```
My name is Tom. My age is 12.
```



```
anonnya@ubuntu:~$ cat bin/var_example5.sh
#!/bin/bash
name=$1
age=$2
echo "My name is $name. My age is $age."
anonnya@ubuntu:~$ bash bin/var_example5.sh Tom 12
My name is Tom. My age is 12.
```

Example 6: Deleting a Variable

You can delete a variable using the "**unset command**". After deleting the variable, you can still access the variable but it will not contain any value. Therefore, printing the variable will output nothing.

Code:

```
#!/bin/bash
name="LinuxSimply"
echo "Name before deletion: $name"
unset name
echo "Name after deletion: $name"
```

Output:

```
Name before deletion: LinuxSimply
Name after deletion:
```

Example 7: Getting the Length of a Variable

To get the length of a variable use the syntax "**${#VARIABLE_NAME}**". It will return the number of characters present in the variable.

Code:

```
#!/bin/bash
var="Linuxsimply"
```

```
length=${#var}
echo "The length of the variable is: $length"
```

Output:

```
The length of the variable is: 11
```

Example 8: Checking If a Variable is Empty or Not

Check if a variable is empty or not using the **if-else statement**. Use the “**-z**” option to check if the length of the variable is 0. If the length is 0 then the variable is regarded as empty. Otherwise, it contains some value.

Code:

```
#!/bin/bash
var=""
if [ -z "$var" ]; then
    echo "The variable is empty."
else
    echo "The variable is not empty."
fi
```

Output:

```
The variable is empty.
```

Example 9: Print Environment Variable using Bash Script

You can store an Environment Variable in a regular manner and print it using **${!..}** syntax.

Code:

```
#!/bin/bash
read -p "Enter an Environment Variable name:" var
echo "Environment:${!var}"
```

Output:

```
Enter an Environment Variable name:
HOME
Environment:/home/anonnya
```

Example 10: Changing Internal Field Separator(IFS)/Delimiter Variable

You can modify the default Internal Field Separator of bash by accessing the **IFS** variable. By changing the **IFS** you will be able to access values separated by your desired delimiter. After this task again restore the original **IFS** to avoid any error.

Code:

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

```
#store default IFS
old_IFS= $IFS
IFS=,
read val1 val2 val3 <<< "5,60,70"
echo 1st value: $val1
echo 2nd value: $val2
echo 3rd value: $val3
#restore default IFS
IFS= $old_IFS;
```

Output:

```
1st value: 5
2nd value: 60
3rd value: 70
```

Example 11: Taking Input Variable as Password

In bash, you can utilize the **read** command for taking **password-type** inputs. Application of the **read** with **-sp** option hides the input characters when you type them**.**

Code:

```
#!/bin/bash
read -sp "Enter your password: " pass
echo -e "\nYour password is: $pass"
```

Output:

```
Enter your password:
Your password is: linuxsimply
```

Example 12: Using Variables in Command Substitutions

Variables can be used to store command outputs. Later you can modify this command output using the variable. Below you can see that the output of the **ls command** has been stored in a variable with the **$(command)** syntax.

Code:

```
#!/bin/bash
files=$(ls)
echo -e "List of files in the Current directory:\n$files"
```

Output:

```
List of files in the Current directory:
bin
Desktop
Documents
```

```
Downloads
Music
Pictures
```

Example 13: Writing and Reading Variables to a File

You can write a variable directly to a file using redirection(>) in **Bash**. To read the variable written in file you can source that file to include the file contents in your script and again refer to that variable.

Code:

```bash
#!/bin/bash
echo "var=Linuxsimply" > var_file.txt
source var_file.txt
echo "Hello from $var!"
```

Output:

```
Hello from Linuxsimply!
```

Example 14: Using Variables in for Loops

Variables are used inside a **for loop** as a part of its syntax. The range or condition to run the **for loop** repetitively is defined with the help of variables. In the below example, you can see that the **for loop** uses the range of values “**{1..5}**” and the variable accessing values from the range is “**i**”.

Code:

```bash
#!/bin/bash
for i in {1..5}; do
    echo "Variable: $i"
done
```

Output:

```
Variable: 1
Variable: 2
Variable: 3
Variable: 4
Variable: 5
```

Example 15: Using Variables in while Loops

The **while loop** in bash script can utilize variables to set the range of the loop. You can write a condition using predefined variables inside the **while[ ]** to run the loop.

Code:

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

```
var=1
while [ "$var" -le 5 ]; do
    echo "Variable: $var"
    var=$((var+ 1))
done
```

Output:

```
Variable: 1
Variable: 2
Variable: 3
Variable: 4
Variable: 5
```

Example 16: Using Variables in until Loops

Using variables in **until loop** is similar to using variables in **while loop.** You will need to write a condition using predefined variables inside the **until[ ]** to run the loop.

Code:

```
#!/bin/bash
var=1
until [ $var -eq 6 ]
do
    echo "Variable: $var"
    var=$((var+ 1))
done
```

Output:

```
Variable: 1
Variable: 2
Variable: 3
Variable: 4
Variable: 5
```

Example 17: Accessing Variables from Array

To access the values stored in an **array** you will need to use the index and array name as the variable. The syntax for accessing variables in an array is "**${ARRAY_NAME[INDEX]}**". It will return the value stored in the specified index of an array. You can store this value in another variable or you can directly display it on the terminal.

Code:

```
#!/bin/bash
arr=("mango" "grape" "apple" "cherry" "orange")
echo "First array element: ${arr[0]}"
echo "Last array element: ${arr[4]}"
```
Output:
```
First array element: mango
Last array element: orange
```

Example 18: Passing Variables to Functions

Variables play a vital role when it comes to implementing functions. You can pass a variable or a value to your defined function by simply typing it after the function name. This value is accessed by the function as arguments in the form **$1**, **$2**, **…** etc.

Code:
```
#!/bin/bash
function hello {
    var=$1
    echo "Hello from $var!"
}
hello "Linuxsimply"
```
Output:
```
Hello from Linuxsimply!
```

Example 19: Using Variables in Case Statements

Variables are used as **case statements**. You can take a variable as user input and pass it in the **Case Statement** to compare with some predefined values or variables. A list of statements then executes depending on the comparison.

Code:
```
#!/bin/bash
read -p "Enter a digit: " dig
case $dig in
    [0-9]) echo "It's a Digit!";;
    *) echo "I don't know what it is";;
esac
```
Output:
```
Enter a digit: 9
It's a Digit!
```

Conclusion

In this article, I have presented a list of hands-on examples of the topic **variables** in **Shell Scripting**. It covers all the basic applications of variables that can be useful while creating a

Shell Script. Moreover, I have included the syntax as well as a set of rules for defining variables in **Bash**. Therefore, this blog can be very useful in learning all about variables in Bash Scripting.