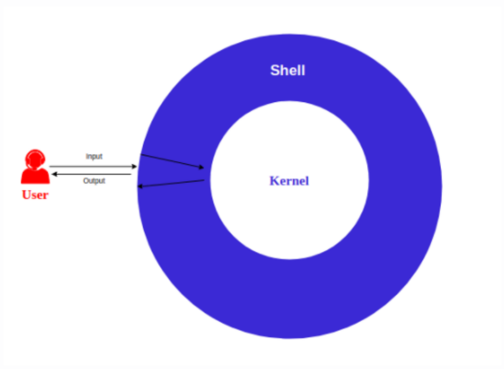
Shell Scripting

Shell

At its base, a **shell** is simply a **macro processor** that **executes commands**. The term macro processor means functionality where text and symbols are expanded to create larger expressions

A Unix shell is both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of gnu utilities. The programming language features allow these utilities to be combined. Files containing commands can be created, and become commands themselves.



Shell and Kernel

> Q: What is Shell?
> A: The shell is the **command interpreter** in an operating system such as Unix or GNU/Linux, it is a program that executes other programs. It provides a computer user an interface to the Unix/GNU Linux system so that the user can run different commands or utilities/tools with some input data.
>
> — - Interview Q&A

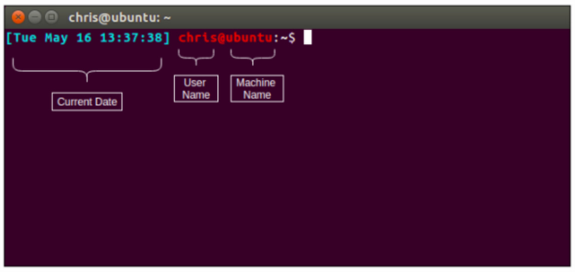
Bash

Bash is the **shell**, or command language interpreter, for the GNU operating system. The name is an acronym for the '**B**ourne-**A**gain **SH**ell'. First used by Stephen Bourne, the author of the direct ancestor of the current Unix shell sh.

While the GNU operating system provides other shells, including a version of csh, Bash is the **default shell**. Like other gnu software, Bash is quite portable. It currently runs on nearly every version of Unix and a few other operating systems – independently-supported ports exist for ms-dos, os/2, and Windows platforms.

The Bash prompt

The Bash prompt can do much more than displaying such simple information as your user name, the name of your machine and some indication about the present working directory. We can add other information such as the current date and time, number of connected users etc.



```
chris@ubuntu: ~
[Tue May 16 13:37:38] chris@ubuntu:~$
```

Bash Prompt

We will some play on Bash prompt. But before change prompts properties we will save our current prompt in another environment variable:

```
[walter@onemachine jerry]$ MYPROMPT=$PS1
[walter@onemachine jerry]$ echo $MYPROMPT
 => [\u@\h \W]\$
[walter@onemachine jerry]$
```

Linux Terminal 201: Customize The Shell Prompt - HakTip 148
Watch later
Share
148
HAK TIP
SHURIKENS FOR I.T. NINJAS
Linux Terminal 201:
Customize The Shell Prompt

https://youtu.be/_kSCpNqKJbM

Some Examples

In order to understand these prompts and the escape sequences used, we refer to the Bash Info or man pages.

- export PS1="[\t \j] "
  - Displays time of day and number of running jobs
- export PS1="[\d][\u@\h \w] : "
  - Displays date, user name, host name and current working directory. Note that \W displays only base names of the present working directory.
- export PS1="{!} "
  - Displays history number for each command.
- export PS1="[\033[1;35m]\u@\h[\033[0m] "
  - Displays user@host in pink.
- export PS1="[\033[7;34m]\u@\h \w [\033[0m] "
  - White characters on a blue background.
- export PS1="[\033[3;35m]\u@\h \w [\033[0m]\a"
  - Pink prompt in a lighter font that alerts you when your commands have finished.

If you want, prompts can execute shell scripts and behave differently under different conditions. You can even have the prompt play a tune every time you issue a command.

Shell Scripts

A shell script is a text file containing shell commands. Scripting allows for an automatic commands execution that would otherwise be executed interactively one-by-one.

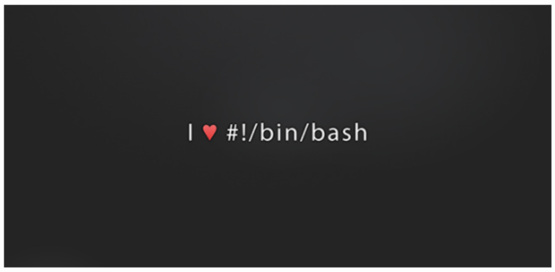
The UNIX shell program interprets user commands, which are either directly entered by the user, or which can be read from a file called the shell script or shell program. Shell scripts are interpreted, not compiled. The shell reads commands from the script line per line and searches for those commands on the system, while a compiler converts a program into a machine-readable form, an executable file – which may then be used in a shell script.

A shell script may be made executable by using the **chmod** command to turn on the **execute** bit. When Bash finds such a file while searching the PATH for a command, it spawns a sub-shell to execute it.

Bash scripts often begin with

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

(assuming that Bash has been installed in /bin), since this ensures that Bash will be used to interpret the script, even if it is executed under another shell.



Bash

Shell types

Just like people know different languages and dialects, your UNIX system will usually offer a variety of shell types:

sh or Bourne Shell: the original shell still used on UNIX systems and in UNIX-related environments. This is the **basic shell**, a small program with few features. While this is not the standard shell, it is still available on every Linux system for compatibility with UNIX programs.

bash or **Bourne Again shell** : the **standard GNU shell**, intuitive and flexible. Probably most advisable for beginning users while being at the same time a powerful tool for the advanced and professional user. On Linux, bash is the standard shell for common users. This shell is a so-called superset of the Bourne shell, a set of add-ons and plug-ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in sh, also work in bash. However, the reverse is not always the case.

csh or **C shell** : the syntax of this shell resembles that of the **C programming language**. Sometimes asked for by programmers.

tcsh or **TENEX C shell** : a superset of the common C shell, enhancing user-friendliness and speed. That is why some also call it the Turbo C shell.

ksh or the **Korn shell**: sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration a nightmare for beginning users.

Shell Types

The file /etc/shells gives an overview of known shells on a Linux system:

```
cat /etc/shells
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

Your default shell is set in the /etc/passwd file.

To switch from one shell to another, just enter the name of the new shell in the active terminal. The system finds the directory where the name occurs using the PATH settings, and since a shell is an executable file (program), the current shell activates it and it gets executed

Finding Bash

The easiest way is to use which command:

```
$ which bash
```

To find the bash run this command from the root dir:

```
$ find ./ name_of_your_bash
```

Other locations to check:

```
ls -l /bin/bash
ls -l /sbin/bash
ls -l /usr/local/bin/bash
ls -l /usr/bin/bash
ls -l /usr/sbin/bash
ls -l /usr/local/sbin/bash
```

To define your script's interpreter as Bash, first locate a full path to its executable binary using which command, prefix it with a shebang #! and insert it as the first line of your script. There are various other techniques on how to define a shell interpreter, but this is a solid start.

Open up you favorite text editor and create file called hello_world.sh. Insert the following lines to a file:

Note: Every bash shell script in starts with shebang:"***#!***" which is not read as a comment. First line is also a place where you put your interpreter which is in this case: /bin/bash.

echo

Here is our first bash shell script example; traditional hello world script:

```
#!/bin/bash
# declare STRING variable
STRING="Hello World"
#print variable on a screen
echo $STRING
```

Navigate to a directory where your hello_world.sh is located and make the file executable:

```
$ chmod +x hello_world.sh
```

Make bash shell script executable. Now you are ready to execute your first bash script:

```
$ ./hello_world.sh
```

```
Hello World!
```

Let's request user's age then print his age:

```
#! /bin/bash
clear
echo "Enter your age"
read  st1
echo "Your age: " $st1
```

```
$ ./your_age.sh
```

```
Enter your age
29 (User will enter an age)
Your age: 29
```

Variables

You can use variables as in any programming language. There are no data types. A variable in bash can contain a number, a character, a string of characters. You have no need to declare a variable, just assigning a value to its reference will create it.

Sample: Writing "James" using variables:

```
#!/bin/bash
NAME=”James”
echo $NAME
echo "$NAME"
echo "${NAME}!"
```

String Quotes

- Enclosing characters in single quotes (' ' ') preserves the literal value of each character within the quotes.
- Enclosing characters in double quotes (' " ') preserves the literal value of all characters within the quotes, with the exception of '$', '`', '\'

```
NAME="John"
echo "Hi $NAME"  #=> Hi John
echo 'Hi $NAME'  #=> Hi $NAME
```

Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated.

```
echo {A,B}.js
{A,B}    Same as A B
{A,B}.js    Same as A.js B.js
{1..5}   Same as 1 2 3 4 5
```

String Lists

The brace expansion is only performed if the given string list is really a list of strings:

```
$ echo {I,want,my,money,back}
I want my money back

$ echo _{I,want,my,money,back}
_I _want _my _money _back

$ echo {I,want,my,money,back}_
I_ want_ my_ money_ back_

$ echo _{I,want,my,money,back}-
_I- _want- _my- _money- _back-
```

Comments

```
# Single line comment
: '
This is a
multi-line
comment
'
```

Comparisons

Numeric and String Comparisons

Using bash shell comparisons, we can compare strings (words, sentences) or integer numbers whether raw or as variables. Note that [[is actually a command/program that returns either 0 (true) or 1 (false). The following table lists rudimentary comparison operators for both numbers and strings:

Shell comparison example:

```
[ 100 -eq 50 ]; echo $? [ “GNU” = “UNIX” ]; echo $?
```

Command	Operation
-lt	<
-gt	>
-le	<=
-ge	>=
-eq	==
-ne	!=

File Conditions

We can check files' conditions:

```
[[ -e FILE ]]   Exists
[[ -r FILE ]]   Readable
[[ -d FILE ]]   Directory
[[ -w FILE ]]   Writable
```

Conditionals

Conditionals let you decide whether to perform an action or not, this decision is taken by evaluating an expression.

if .. then example:

```
#!/bin/bash
if [ "foo" = "foo" ]; then
  echo expression evaluated as true
fi
```

if .. then...else example:

```
#! /bin/bash
clear
echo "enter a number"
read  st1
if  (( $st1 ==  5  ))
then
  echo "You entered five!"
fi
# -eq  -gt -lt -ge  -le -a -o
if [ $st1 -eq 5  ]
then
   echo equal to five
elif (( $st1 < 5 ))
then
   echo "less than five"
else
 echo "more than five"
fi
```

Loops

for, while and until

In this part, you'll find for, while and until loops.

The for loop is a little bit different from other programming languages. Basically, it lets you iterate over a series of ‘words’ within a string.

The while executes a piece of code if the control expression is true, and only stops when it is false (or an explicit break is found within the executed code.

The until the loop is almost equal to the while loop, except that the code is executed while the control expression evaluates to false

For:

```
#!/bin/bash
for i in $( ls ); do
   echo item: $i
done

for service in S3 EC2 Lambda Glacier CloudFront Kinesis
do
  echo "Amazon Service: $service"
done
```

While:

```
#!/bin/bash
     COUNTER=0
     while [  $COUNTER -lt 10 ]; do
         echo The counter is $COUNTER
         let COUNTER=COUNTER+1
     done

      Until sample
     #!/bin/bash
     COUNTER=20
     until [  $COUNTER -lt 10 ]; do
         echo COUNTER $COUNTER
         let COUNTER-=1
     done
```

Functions

As in almost any programming language, you can use functions to group pieces of code in a more logical way or practice the divine art of recursion.

Declaring a function is just a matter of writing function my_func { my_code }.

Calling a function is just like calling another program, you just write its name.

Functions sample:

```
#!/bin/bash

function quit {
   exit
}

function hello {
   echo Hello!
}

 hello
 quit

 echo foo
```

Case

Each case statement starts with the case keyword followed by the case expression and the in keyword. The statement ends with the esac keyword. You can use multiple patterns separated by the | operator. The) operator terminates a pattern list.

```
case EXPRESSION in

  PATTERN_1)
    STATEMENTS
    ;;

  PATTERN_2)
    STATEMENTS
    ;;

  PATTERN_N)
    STATEMENTS
    ;;

  *)
    STATEMENTS
    ;;
esac
```

Case and Function

```bash
#!/bin/bash
inputs(){
     read -p "Enter first integer: " int1
    read -p "Enter second integer: " int2
}

exitPrompt(){
    read -p "Do you wish to continue? [y]es or [n]o: " ans
    if (( $ans == "n"))
    then
        echo "Exiting the script. Have a nice day!"
        sleep 2
        exit
    else
        continue
    fi
}


while(true)
    do
    clear
    printf "Choose from the following operations: \n"
    printf "[a]Addition\n[b]Subtraction\n[c]Multiplication\n[d]Division\n"
    printf "################################\n"
    read -p "Your choice: " choice

    case $choice in
    [aA])
        inputs
        res=$((int1+int2))
    ;;

    [bB])
        inputs
        res=$((int1-int2))
    ;;

    [cC])
        inputs
        res=$((int1*int2))
    ;;

    [dD])
        inputs
        res=$((int1/int2))
    ;;

    *)
        res=0
        echo "wrong choice!"
    esac

    echo "The result is: " $res
    exitPrompt
done
```