Shell Scripting

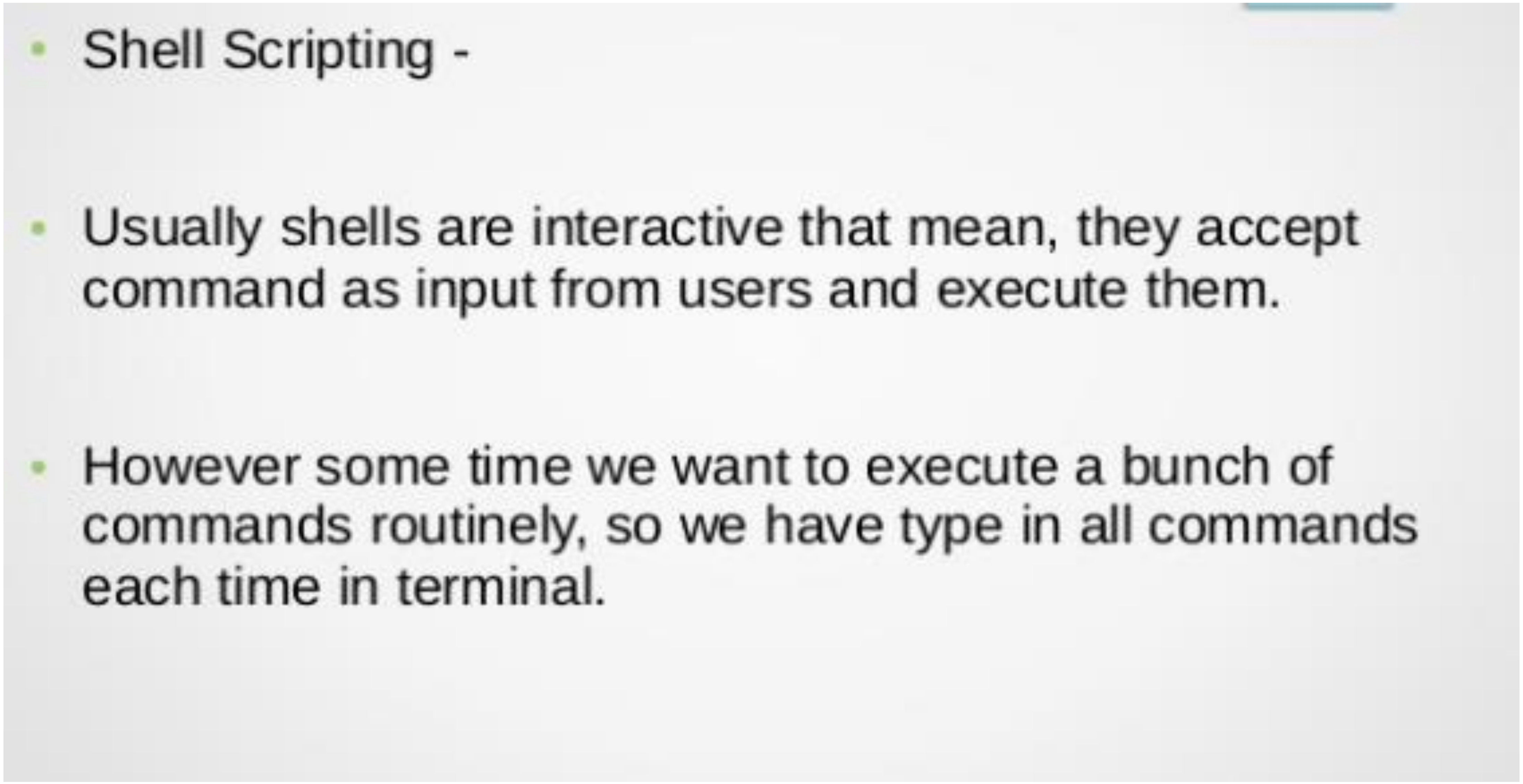
What is a shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- Full programming language
- A program like any other
  - This is why there are so many shells

- A shell is special user program which provide an interface to user to use operating system services.
- Shell accept human readable commands from user and convert them into something which kernel can understand.
- It is a command language interpreter that execute commands read from input devices such as keyboards or from files.
- The shell gets started when the user logs in or start the terminal.

Most Commonly Used Shells

`/bin/csh`	C shell
`/bin/tcsh`	Enhanced C Shell
`/bin/sh`	The Bourne Shell / POSIX shell
`/bin/ksh`	Korn shell
`/bin/bash`	Korn shell clone, from GNU

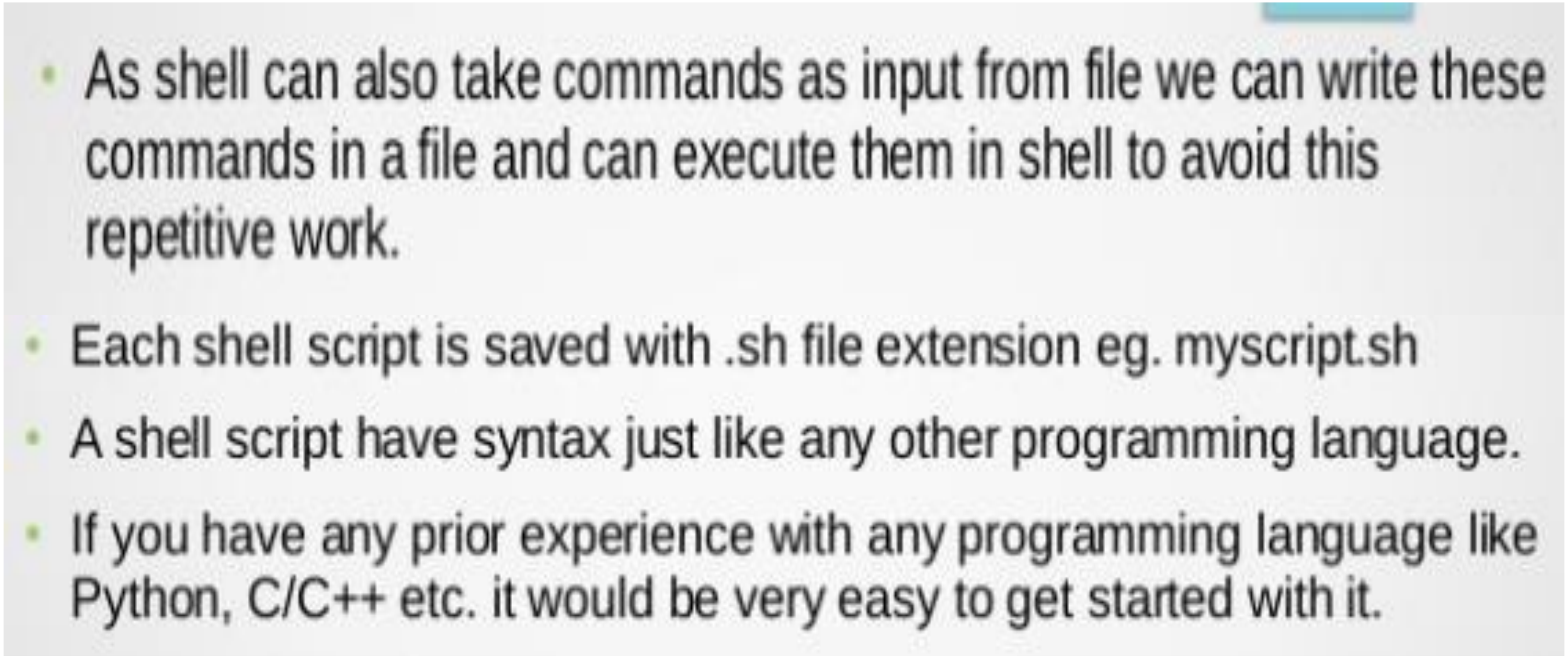


- Shell Scripting -

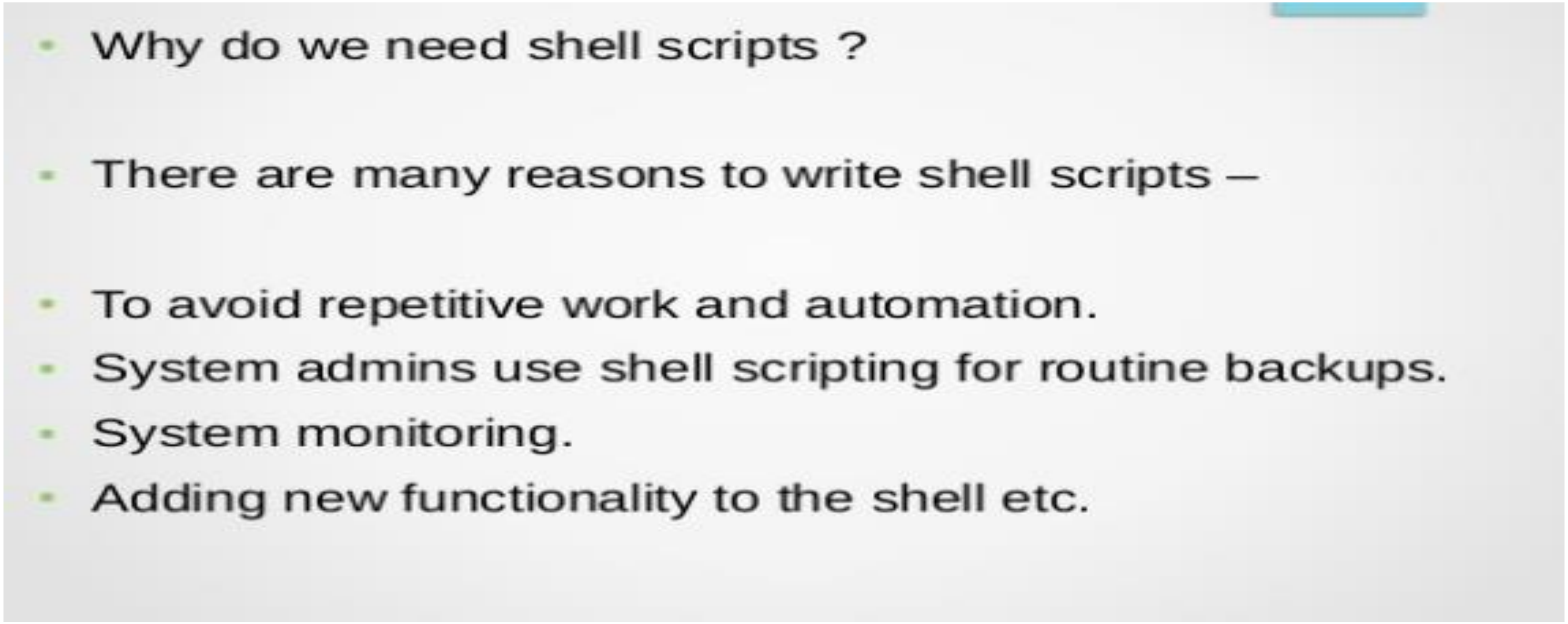
- Usually shells are interactive that mean, they accept command as input from users and execute them.

- However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

- The shell is, after all, a real programming language, complete with variables, control structures, and so forth.
- No matter how complicated a script gets, it is still just a list of commands executed sequentially.



- As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work.
- Each shell script is saved with .sh file extension eg. myscript.sh
- A shell script have syntax just like any other programming language.
- If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.


- Why do we need shell scripts ?
- There are many reasons to write shell scripts –
- To avoid repetitive work and automation.
- System admins use shell scripting for routine backups.
- System monitoring.
- Adding new functionality to the shell etc.

Shell Scripts

- A shell script is a regular text file that contains shell or commands
  - Before running it, it must have execute permission:
    - **`chmod +x`** ***`filename`***
- A script can be invoked as:
  - **`./filename.sh`**

Shell Scripts

- When a script is run, the **kernel** determines which shell it is written for by examining the first line of the script
    - If 1st line starts with **`#!`*`pathname-of-shell`***, then it invokes *pathname* and sends the script as an argument to be interpreted
    - If **`#!`** is not specified, uses default shell
    - If a shebang is not specified   the script will be parsed by whatever the default interpreter is used by that Shell. For example, the default interpreter for bash is bash and for zsh is sh.
    - The sequence of characters (#!) is called shebang and is used to tell the operating system which interpreter to use to parse the rest of the file.

    - Shebang Interpreter Directive

Simple Example

```
#!/bin/bash

echo Hello World
```

Scripting vs. C Programming

- Advantages of shell scripts
  - Easy to work with other programs
  - Easy to work with files
  - Easy to work with strings

- Disadvantages of shell scripts
  - Slower
  - Not well suited for data structures

- echo prints the rest of the line to the screen (standard output).

- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

- Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix shell variables will have their names in UPPERCASE.

- The following examples are valid variable names −

- _ALI
- TOKEN_A
- VAR_1
- VAR_2

- Following are the examples of invalid variable names −

- 2_VAR
- -VARIABLE
- VAR1-VAR2
- VAR_A!
- The reason you cannot use other characters such as !, *, or - is that these characters have a special meaning for the shell.

Defining Variables

- Variables are defined as follows −
- variable_name=variable_value

For example −

NAME=“Amity Class"

- The above example defines the variable NAME and assigns the value “Amity Class" to it. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.
- Shell enables you to store any value you want in a variable. For example −
- VAR1=“Amity Class"
- VAR2=100

Accessing Values

- To access the value stored in a variable, prefix its name with the dollar sign ($)

- For example, the following script will access the value of defined variable NAME and print it on STDOUT −

```
#!/bin/bash
NAME=“Amity Class"
echo $NAME
```

- The above script will produce the following value −

```
Amity Class
```

Read-only Variables

- Shell provides a way to mark variables as read-only by using the read-only command.
- After a variable is marked read-only, its value cannot be changed.
- For example, the following script generates an error while trying to change the value of NAME −

```
NAME=“Amity Class"
readonly NAME
NAME=“AI Class"
```

- The above script will generate error

$

Shell Variables

- Linux shell scripting are using two types of variables : **System Defined Variables** & **User Defined Variables**

- a shell script doesn’t require you to **declare a type** for your variables
- The value of a variable can be returned/used by adding a prefix $

To set:

```
    name=value
```

To Read: **`$var`**

Variable Example

```
#!/bin/bash

MESSAGE="Hello World"
echo $MESSAGE
```

You can accept input from the keyboard and assign an input value to a user defined shell variable using read command.

The following script uses he **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

- read multiple variables
- read a b c
- read -p "Prompt" variable1 variable2
- Where,
- **-p "Prompt"** : Display prompt to user without a newline.
- **variable1** : The first input (word) is assigned to the variable1.
- **variable2** : The second input (word) is assigned to the variable2.

Example

```
#! /bin/bash
 read -p "Enter your name : "  name
 "Hi, $name. Let us be friends!"
```

Save and close the file. Run it as follows:

```
chmod +x filename.sh
./filename.sh
```

- *Example*
- #!/bin/bash

```
# read three numbers and assigned them to 3 vars
read -p "Enter number one : " n1
read -p "Enter number two : " n2
read -p "Enter number three : " n3
```

- # display back 3 numbers - punched by user.

```
echo "Number1 - $n1"
echo "Number2 - $n2"
echo "Number3 - $n3"
```

- Handling Passwords

The -s option causes input coming from a terminal do not be displayed on the screen.

```
#!/bin/bash
read -s -p "Enter Password  : " my_password
echo
echo "Your password - $my_password"
```

Environmental Variables

NAME	MEANING
$HOME	Absolute pathname of your home directory
$PATH	A list of directories to search for
$USER	Your login name
$SHELL	Absolute pathname of login shell
$TERM	Type of your terminal
$PS1	Prompt

- To group several strings together as one argument it is necessary to use double quotes: “
- For example:
- v=Hello World
- echo $v
- Hello
- v="Hello World"
- echo $v
- Hello World

Command Substitution

- Used to turn the output of a command into a string
- Used to create arguments or variables
- Command is placed with backquote`   ` to capture the output of command

```
$ date
Wed Sep 25 14:40:56 EDT 2001
$ NOW=`date`
```

Operators

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Arithmetic

- shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs

- Use external command **/bin/expr**
- **expr expression**
  - Evaluates expression and sends the result to standard output.

```
expr 4 + 3 (space around operator)
```

```
expr 4 "*" 12
```

- Particularly useful with command substitution
  X=`expr $X + 2`
- Example

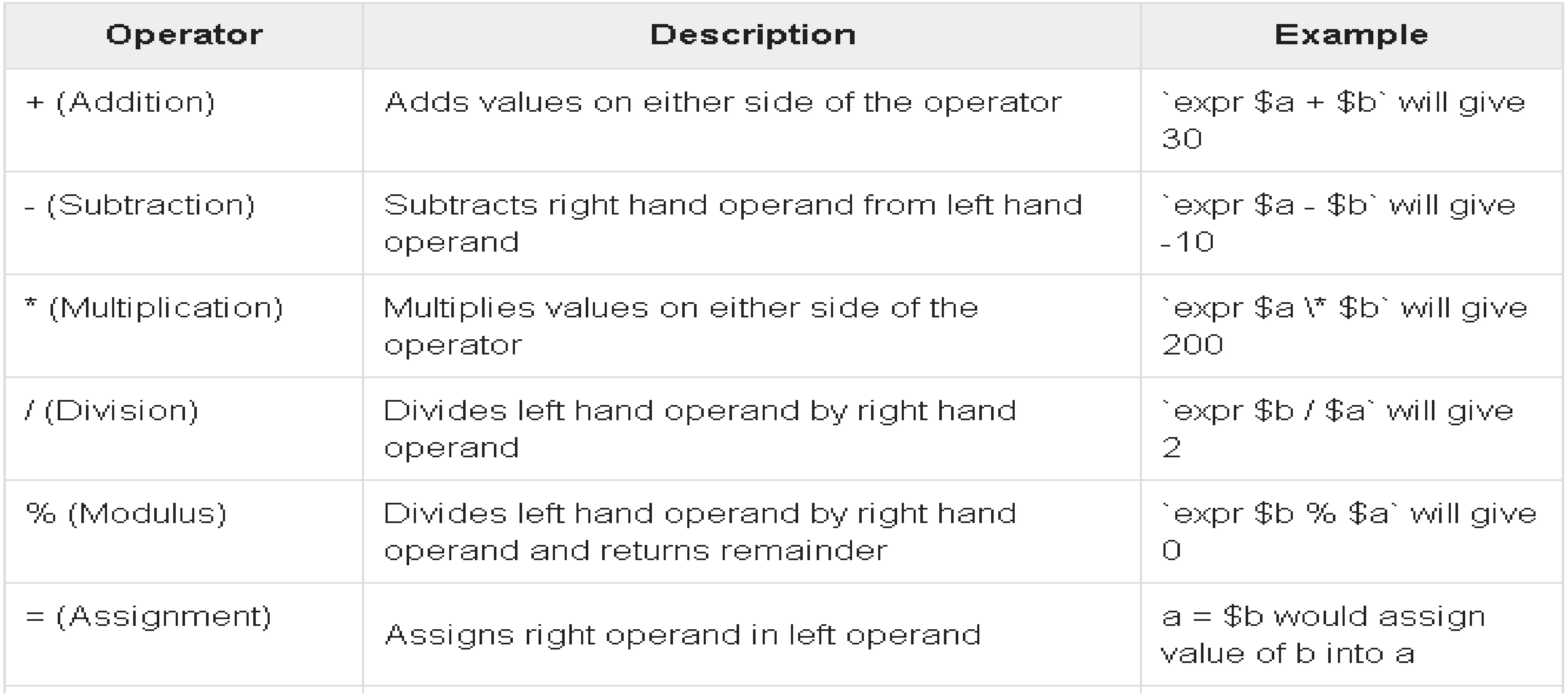
```
#!/bin/bash
val=`expr 2 + 2`
echo "Total value : $val"
```

The following points need to be considered while using expr −

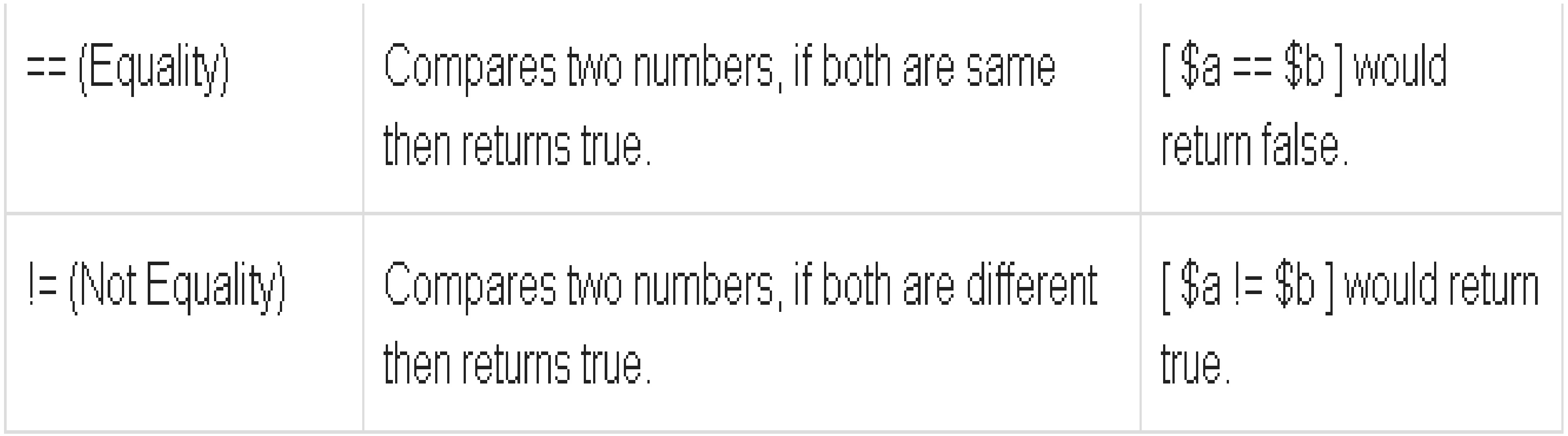
- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ‘ ‘, called the backquote

Arith

- Assume variable **a** holds 10 and variable **b** holds 20 then



Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a



== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

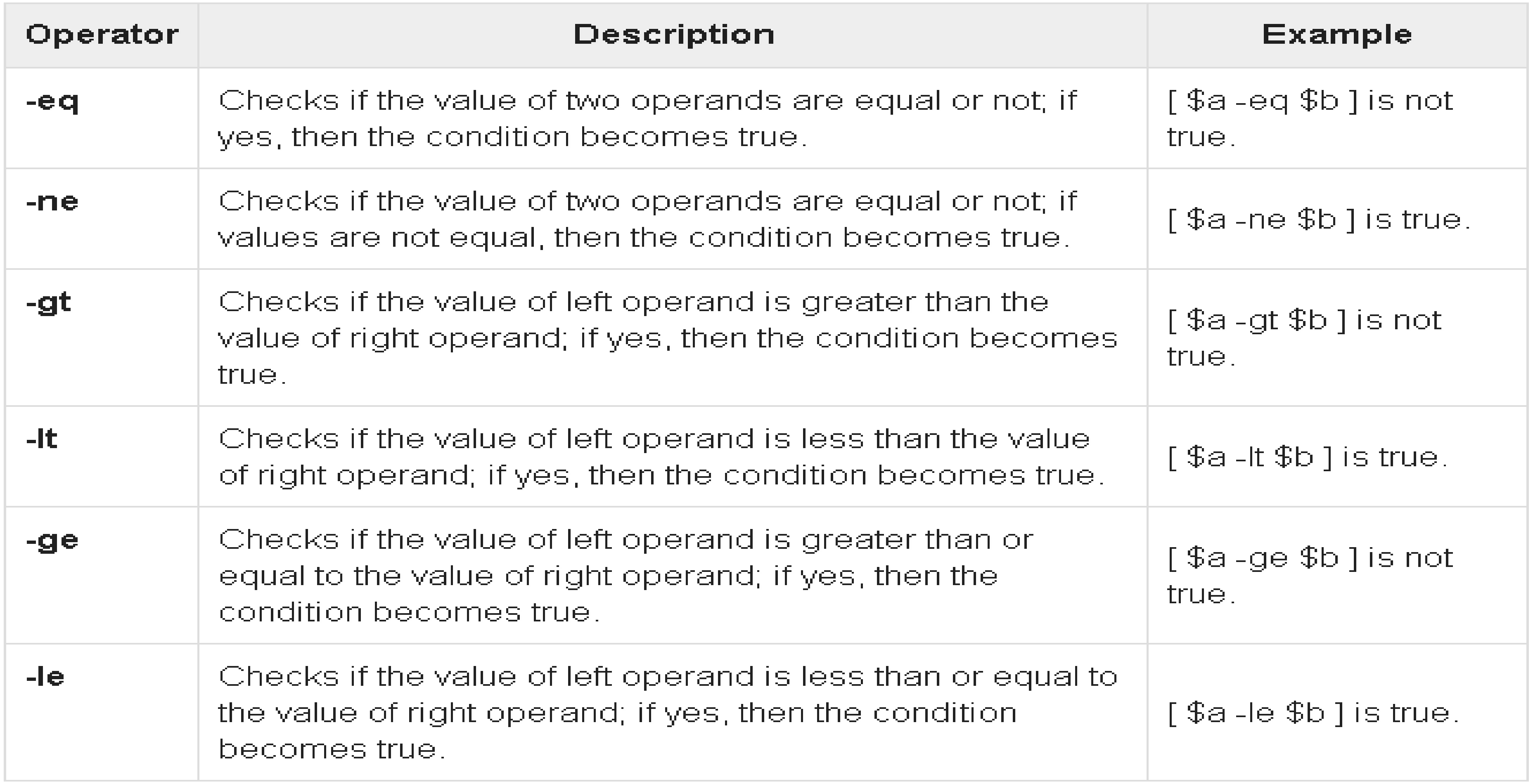
It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example [$a == $b] is correct whereas, [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

```
a=10
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"
```

Relational Operators

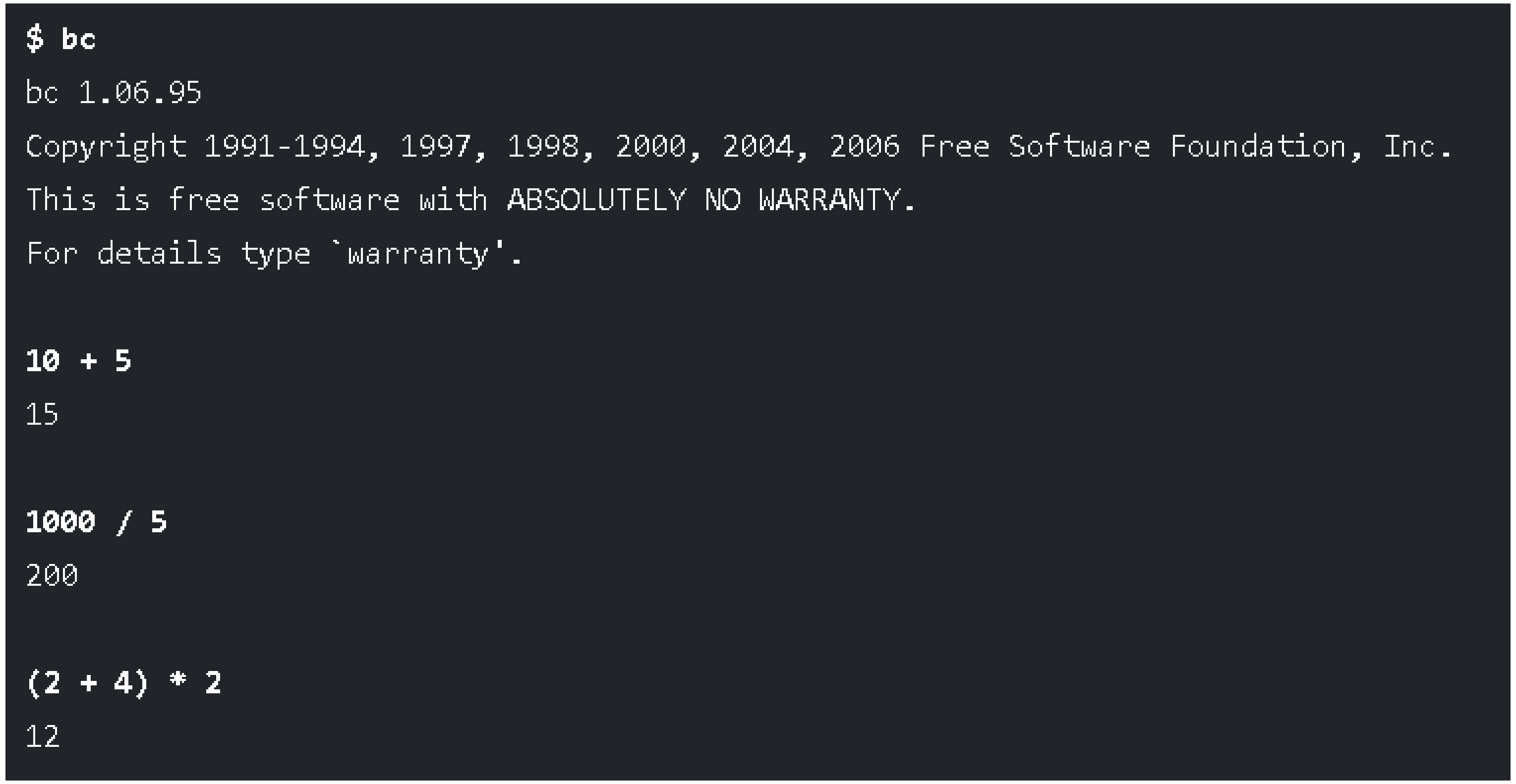
- Assume variable **a** holds 10 and variable **b** holds 20 then



Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

- It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them

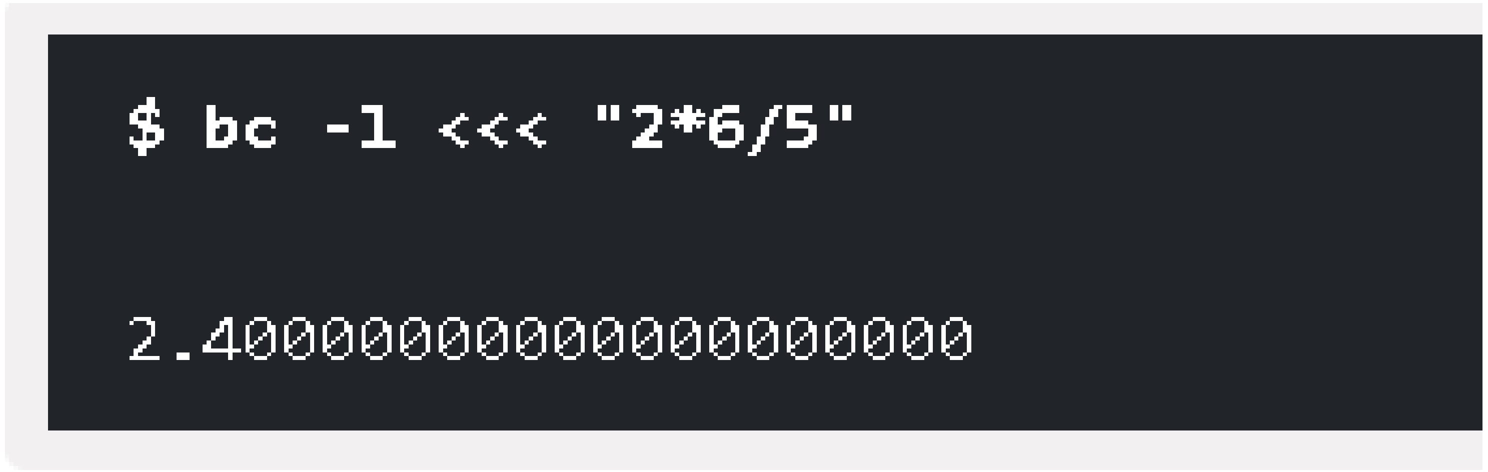
bc

- To open **bc** in interactive mode, type the command **bc** on command prompt and simply start calculating your expressions.
- 

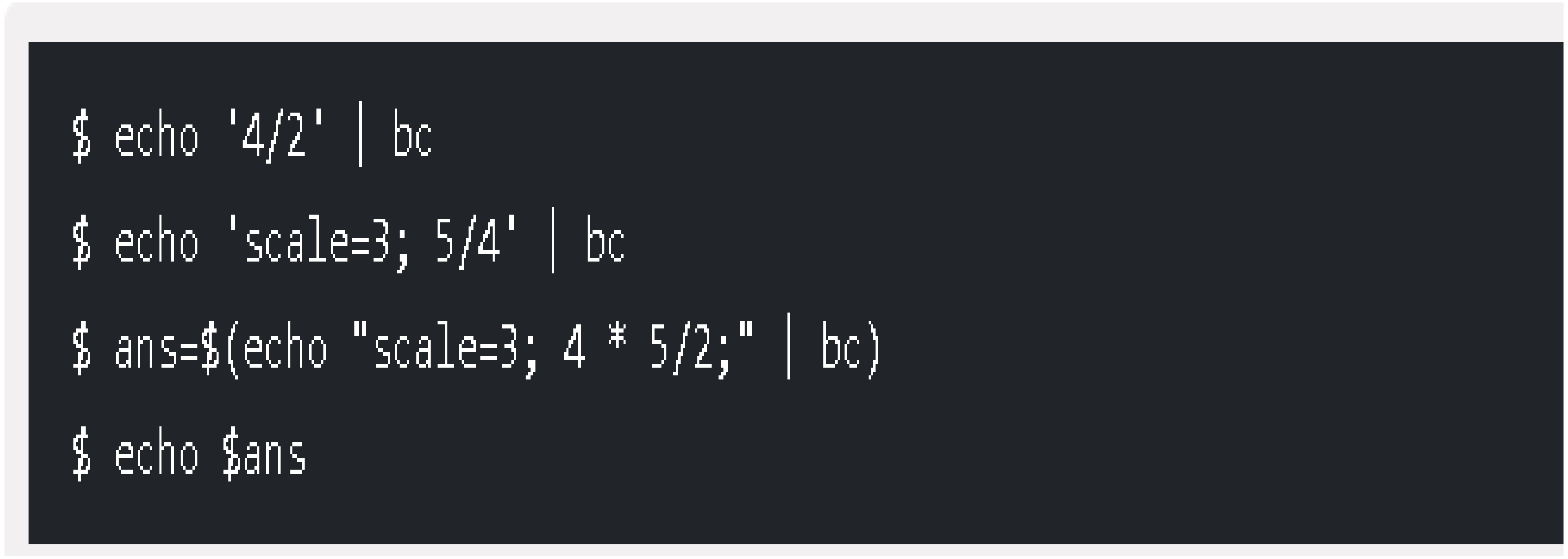
```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.

10 + 5
15

1000 / 5
200

(2 + 4) * 2
12
```

```
scale=3; 5/4
```

You can also use the following command for common shells for instance in bash to pass arguments to **bc** as shown.



```
$ bc -l <<< "2*6/5"

2.40000000000000000000
```



```
$ echo '4/2' | bc
$ echo 'scale=3; 5/4' | bc
$ ans=$(echo "scale=3; 4 * 5/2;" | bc)
$ echo $ans
```