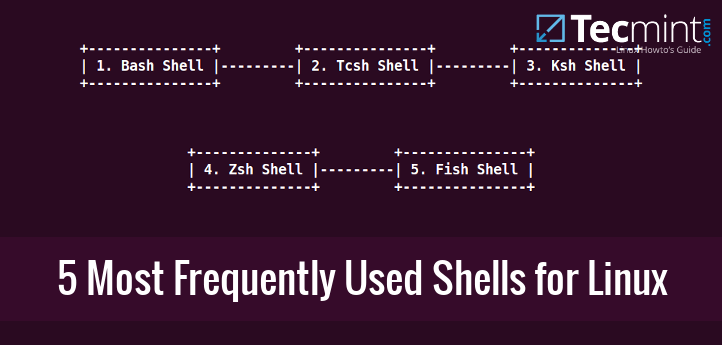
Types of shell



### To change your shell with chsh:

# grep tecmint /etc/passwd

**# chsh --shell /bin/sh tecmint**

# grep tecmint /etc/passwd

Usermod also user to change the shell

1. cat /etc/shells

At the shell prompt, list the available shells on your system with cat /etc/shells.

1. chsh

Enter chsh (for "change shell"). Code Listing 3.5 shows the system response. Some systems prompt for a password, and some don't.

1. /bin/zsh

Type in the path and name of your new shell.

Creating Temporary Aliases

What you need to do is type the word alias then use the name you wish to use to execute a command followed by **"="** sign and quote the command you wish to alias.

The syntax is as follows:

$ alias shortName="your custom command here"

$ unalias alias\_name

$ unalias -a [remove all alias]

#### Creating Permanent Aliases

To keep **aliases** between sessions, you can save them in your user’s shell configuration profile file. This can be:

* Bash – **~/.bashrc**

**Echo command in linux**

**A=10**

**echo $A**

**echo vivek bhardwaj**

**echo -e "Tecmint \bis \ba \bcommunity \bof \bLinux \bNerds"**

**Getting started with Shell Programming**

# Variables in Shell

To process our data/information, data must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).

In Linux (Shell), there are two types of variable:  
(1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.  
(2) **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters.

(1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

You can see system variables by giving command like **$ set**, some of the important System variables are:

|  |  |
| --- | --- |
| **System Variable** | **Meaning** |
| BASH=/bin/bash | Our shell name |
| BASH\_VERSION=1.14.7(1) | Our shell version name |
| COLUMNS=80 | No. of columns for our screen |
| HOME=/home/vivek | Our home directory |
| LINES=25 | No. of columns for our screen |
| LOGNAME=students | students Our logging name |
| OSTYPE=Linux | Our Os type |
| PATH=/usr/bin:/sbin:/bin:/usr/sbin | Our path settings |
| PS1=[\u@\h \W]\$ | Our prompt settings |
| PWD=/home/students/Common | Our current working directory |
| SHELL=/bin/bash | Our shell name |
| USERNAME=vivek | User name who is currently login to this PC |

***NOTE*** that Some of the above settings can be different in your PC/Linux environment. You can print any of the above variables contains as follows:  
$ echo $USERNAME  
$ echo $HOME

Exercise:  
1) If you want to print your home directory location then you give command:  
a)$ echo $HOME

# **How to define User defined variables (UDV)**

To define UDV use following syntax  
*Syntax:*  
variable name=value

'**value**' is assigned to given '**variable name**' and Value must be on right side = sign.

(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g. In following variable declaration there will be no error  
$ no=10  
  
*Example:*$ no=10# this is ok  
To define variable called 'vech' having value Bus  
$ vech=Bus  
To define variable called n having value 10  
$ n=10

# How to print or access value of UDV

**$variablename**

Define variable vech and n as follows:  
$ vech=Bus  
$ n=10  
To print contains of variable 'vech' type  
$ echo $vech

# **More about Quotes**

There are three types of quotes

|  |  |  |
| --- | --- | --- |
| **Quotes** | **Name** | **Meaning** |
| **"** | Double Quotes | "Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $).double quotes will interperate (e.g., variables, backquote, \ escapes) |
| **'** | Single quotes | 'Single quotes' - Single quotes won’t interpolate anything  $a as it is |
| **`** | Back quote | `Back quote` - To execute command  echo vivek `ls` |

# **5.9. Math in Shell Scripts**

Shell script variables are by default treated as strings, **not numbers**, which adds some complexity to doing math in shell script. To keep with script programming paradigm and allow for better math support, languages such Perl or Python would be better suited when math is desired. However, it is possible to do math with shell script. In fact, over the years, multiple facilities have been added to Unix to support working with numbers.

**Note**

As we will see, some of the commands used to facilitate math are a little picky about things like spaces around operators.

## **5.9.1. declare**

You may recall, that when the text book introduced the **declare** statement, it said that it is not always needed. So what do you get by declaring a variable to be an integer? The following example illustrates that a declared integer is not treated as a string.

$ n=6/3

$ echo $n

6/3

$ declare -i n

$ n=6/3

$ echo $n

2

When you do not need the **declare** statement is when you will use a program or built-in command to evaluate a math statement.

## **5.9.2. expr**

An old Unix program that can evaluate math is **expr**. **expr** became popular in the days of the Bourne Shell, which did not support math. With Bash and Korn shell, it is generally not needed. Since it is a command, command substitution is needed. We are still treating the variable as a string. As you can see, it is picky about spaces.

$ z=5

$ z=`expr $z+1` ---- Need spaces around + sign.

$ echo $z

5+1

$ z=`expr $z + 1`

$ echo $z

6

## **5.9.3. let**

A Bash and Korn shell built-in command for math is **let**. As you can see, it is also a little picky about spaces, but it wants the opposite of what **expr**wanted. **let** also relaxes the normal rule of needing a $ in front of variables to be read.

$ let z=5

$ echo $z

5

$ let z=$z+1

$ echo $z

6

$ let z=$z + 1 # --- Spaces around + sign are bad with let

-bash: let: +: syntax error: operand expected (error token is "+")

$let z=z+1 # --- look Mom, no $ to read a variable.

$echo $z

7

## **5.9.4. BASH Arithmetic**

With the BASH shell, whole arithmetic expressions may be placed inside double parenthesis. This form is more forgiving about spaces.

$ ((e=5))

$ echo $e

5

$ (( e = e + 3 ))

$ echo $e

8

$ (( e=e+4 )) # -- spaces or no spaces, it doesn't matter

$ echo $e

12

|  |  |
| --- | --- |
| **Arithmetic Operators** | |
| + - | Addition, subtration |
| ++ -- | Increment, decrement |
| \* / % | Multiplication, division, remainder |
| \*\* | Exponentiation |

Numerical boolean expressions in [Control Constructs](http://faculty.salina.k-state.edu/tim/unix_sg/bash/control.html#control) are also expressed using double parenthesis.

**if** (( x > y )); then

echo "x > y"

fi

|  |  |
| --- | --- |
| **Logical and Boolean Operators** | |
| <= >= < > | Less than or equal, greater than or equal, less than, greater than |
| == != | Equal, not equal |
| ! | Logical NOT |
| && | Logical AND |
| || | Logical OR |

## **5.9.5. bc**

What if you want to do math with floating point numbers or you have some fairly complicated math to do? Neither form of let, supports floating point numbers. The bc command is needed. But you have to treat the variables as strings.

Here is what happens when we try to do floating point math with the shell:

$let r=3.5

-bash: let: r=3.5: syntax error in expression (error token is ".5")

$(( r = 3.5 ))

-bash: ((: r = 3.5 : syntax error in expression (error token is ".5 ")

**bc**

An arbitrary precision calculator language. **bc** may either be run interactively, or as a shell script command. In interactive mode, type cntrl-d (EOF) to exit.

SYNOPSIS

**bc**

**bc** EXPRESSION

Here are some examples:

$ bc

bc 1.06

Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.

This is free software with ABSOLUTELY NO WARRANTY.

For details type `warranty'.

3 + 2

5

obase=2

12

1100

<cntrl-d>

Remember to type cntrl-d (EOF) to exit from interactive mode.

$r=3.5

$s=`echo "$r + 2.2" | bc`

$echo $s

5.7

$ z = `echo $z + 1 | bc`

-bash: z: command not found

# -- spaces around = sign are bad

(shell thing, not bc)

$ z=`echo "$z + 1" | bc`

$ echo $z

8

$ z=`echo "$z+1" | bc` -- spaces don't matter with bc

$ echo $z

9

c=$((a+b))

c=`expr $a + $((b\*b\*b))`

c=$((a+(b\*b\*b)))

**Note:**expr 20 %3 - Remainder read as 20 mod 3 and remainder is 2.  
expr 10 \\* 3 - Multiplication use \\* and not \* since its wild card.

For the last statement not the following points

(1) First, before expr keyword we used ` (back quote) sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde (~) on PC keyboard OR to the above of TAB key.

(2) Second, expr is also end with ` i.e. back quote.

(3) Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum

(4) Here if you use double quote or single quote, it will NOT work  
For e.g.  
**$ echo "expr 6 + 3"** # It will print expr 6 + 3  
**$ echo 'expr 6 + 3'** # It will print expr 6 + 3

*Example*:  
**$ echo "Today is date"**  
Can't print message with today's date.  
**$ echo "Today is `date`".**  
It will print today's date as, Today is Tue Jan ....,Can you see that the `date` statement uses back quote?

# **The read Statement**

read -p “ prompt\_text”

-s do not echo input coming from a terminal

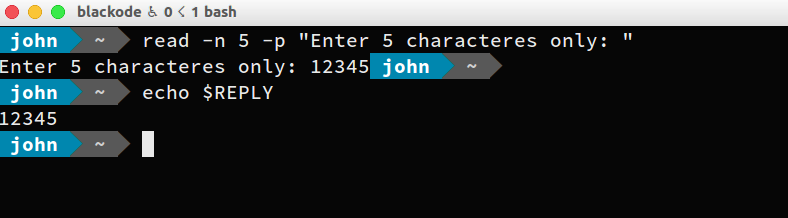
-a array assign the words read to sequential indices of the array

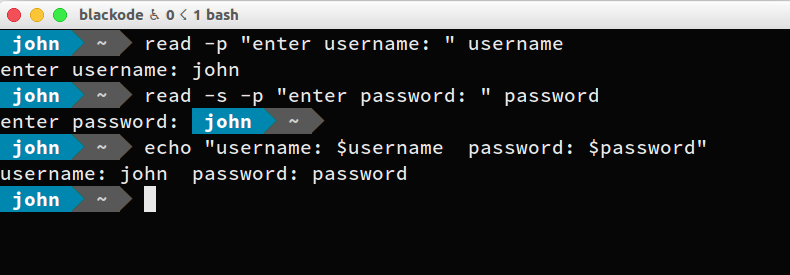
variable ARRAY, starting at zero

-t timeout time out and return failure if a complete line of input is

not read withint TIMEOUT seconds.

Here we read the data along with some hint text .





|  |
| --- |
|  |

# More command on one command line

**$ date;who**

|  |
| --- |
|  |