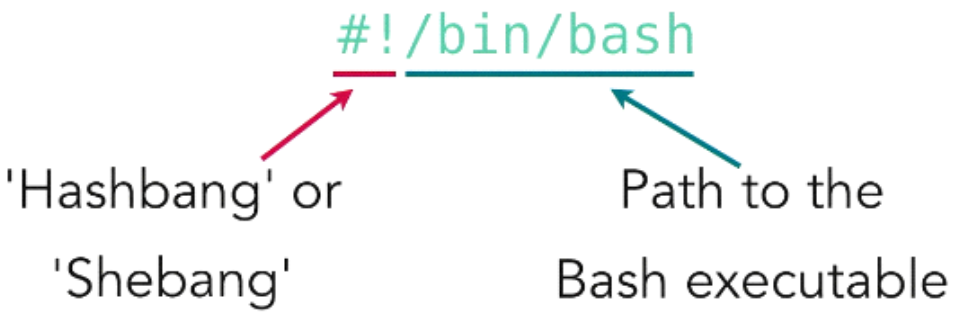
Introduction

Sunday, December 11, 2016 8:50 PM

Bash Syntax

- Every file starts with ***interpreter directive*** Which tells interpreter that it is a bash file once it is made executable.
- Interpreter directive is Also called as Hashbang or shebang



- Usually executable path is /bin/bash

Creating and Running Bash Scripts

- Let we have a file my.sh like this

```
#!/bin/bash
# This is a basic bash script.
ls
```

- Now if we want to execute it we can use

```
$sh my.sh
```

Or

```
$bash my.sh
```

- But if we want to make it executable we have to use 'chmod' like

```
$ chmod +x my.sh
```

- After that we can use simply name of file to execute it.

FULL EXAMPLE-

```
scott@orion:~$ bash my.sh
fruit  my.sh  pets  trees
scott@orion:~$ chmod +x my.sh
scott@orion:~$ ./my.sh
```

http://Bash-hackers.org is great resource for bash .

echo command

Sunday, December 11, 2016 9:24 PM

- Echo can be used to print some information on screen
- There are three possible conditioning of quotation with echo
  1- without quotation 2- single quote 3- double quote

Displaying text with 'echo'

```
echo statement
echo 'statement'
echo "statement"
```

Without quotation -

- we have to skip each and every special symbol we encounter.
  example- in following statement we get error as paranthesis are special symbols so we have to escape them.

STATEMENT 1

```
echo $greeting, world (planet)!
```

OUTPUT-

```
./my.sh: line 5: syntax error near unexpected token `('
./my.sh: line 5: `echo $greeting, world (planet)!'
```

STATEMENT 2

```
echo $greeting, world \(planet\)!
```

OUTPUT-

```
scott@orion:~$ ./my.sh
hello, world (planet)!
```

Note- the value of $greeting comes from following variable

```
greeting="hello"
```

Single quotation-

- In single quotation nothing inside quote interpreted, so variables comes literally same as they are inside single quotations.(same as PHP)

```
echo '$greeting, world (planet)!'
```

OUTPUT-

```
scott@orion:~$ ./my.sh
$greeting, world (planet)!
```

Double quotation-

- It is best way as it also evaluates variable inside and also we not need to escape any special symbol. (same as PHP)

```
echo "$greeting, world (planet)!"
```

OUTPUT-

```
scott@orion:~$ ./my.sh
hello, world (planet)!
```

- If we do not want to evaluate some variable then we can escape that dollar($) using backspace(\)

```
echo "\$greeting, world (planet)!"
```

OUTPUT-

```
$greeting, world (planet)!
```

- For simple new line we can use simple empty 'echo' as echo automatically adds new line.

Variables

Monday, December 12, 2016 1:16 AM

Working with variables

- named with alphanumeric characters
- names must start with a letter

- While defining variable there should not be any space between variable, '=' and value otherwise we will get error.

```
#!/bin/bash
# This is a basic bash script.
a=Hello
b="Good Morning"
c=16

echo $a
echo $b
```

Ofcourse this is valid
d=" this is a valid variable declaration";
(because value is starting just after '=' and there is no space between them)

- We use $ while using variable value(accessing alue of variable

Adding attributes to variables

```
declare -i d=123        # d is an integer
declare -r e=456        # e is read-only
declare -l f="LOLCats"  # f is lolcats
declare -u g="LOLCats"  # g is LOLCATS
```

- 'I' indicates that variable is integer
- 'r' indicates that variable will be constant(integer or string)
- 'l' means string will be stored as lower case(give any value convert automatically)
- 'u' means string will be stored as upper case(" ----------"----------")

Built-in variables

- There are some built-in variable on both linux and mac machines.

HOME variable

`echo $HOME`
Returns user home directory

Mac: `/Users/scott`
Linux: `/home/scott`

- Here scott is username(like ashish)

HOST variable

`echo $HOSTNAME`
Returns system name

Mac: `scott.local`
Linux: `orion`

PWD variable

`echo $PWD`
Returns current directory

MACHTYPE variable

`echo $MACHTYPE`
Returns machine type

Mac: `x86_64-apple-darwin12`
Linux: `x86_64-pc-linux-gnu`

BASH_VERSION variable

`echo $BASH_VERSION`
Returns version of Bash

Mac: `3.2.48(1)-release`
Linux: `4.2.25(1)-release`

SECONDS variable

```
echo $SECONDS
```
Returns the number of seconds the Bash session has run

Handy for timing things

Special variables

- **'*'** - all files/directories in pwd
- '$*'- all positional parameters in following bash or argument in function
- '$#' - number of positional parameters in bash file or argument in function
- '$?' - represent output of last command

Working with Integer Maths

Monday, December 12, 2016 6:51 PM

- **Note- Bash math only works with integers , for floating point maths we can use a program 'bc'**

- For working with Integers we need to wrap that expression in double **parantheses**

```
(( expression ))
```

NOTE- double paranthesis tells bash that this is an arithmetic expression otherwise it may think it as string.

- If we wanted to get the output in some variable then we have to use command substitution.(the expression calculated as a command)

- `val=$(( expression ))`

Arithmetic operations

Operation	Operator
Exponentiation	`$a ** $b`
Multiplication	`$a * $b`
Division	`$a / $b`
Modulo	`$a % $b`
Addition	`$a + $b`
Subtraction	`$a - $b`

- Bash supports 6 basic arithmetic operations.
  Example-

```
#!/bin/bash
# This is a basic bash script.
d=2
e=$((d+2))
echo $e
```

Output-

```
scott@orion:~$ ./my.sh
4
```

- We can also use increment and decrement operator(as C both post and pre), and we can also use combination assignments(like +=,/= etc)

```
d=2
e=$((d+2))
echo $e
((e++))
echo $e
((e--))
echo $e
echo
((e+=5))
echo $e
((e*=3))
echo $e
((e/=3))
echo $e
((e-=5))
echo $e
```

- As each of these commands are bash command we can use **command substitution**
  To get the output of any of these.

```
d=2;
e=$((d+2))
echo $e;
c=$((e+=2))
echo $e;
echo $c;
```

OUTPUT-

```
ashish_patel@my_pc MINGW32 /f/my_folder/Google Drive/bash
$ ./first.sh
4
6
6
```

- If we remove (( )) from expression command will not be treated as arithmetic expression and we might get absurd result.
  - Like in the following example we get string concatnation instead of addition
    -
```
d=2;
e=$d+2;
f=d+2;
echo $d;
echo $e;
echo $f;
```
    -
```
$ ./first.sh
2
2+2
d+2
```

Bc command

Monday, December 12, 2016 7:49 PM

- The bc command evaluates expressions and has syntax similar to the c programming language.

- The bc command supports the following features.
  - Arithmetic operators
  - Increment and decrement operators
  - Assignment operators
  - Comparision or Relational Operators
  - Logical or Boolean operators
  - Math Functions
  - Conditional statements
  - Iterative statements
  - Functions

  From <http://www.unixmantra.com/2013/05/bc-unix-calculator.html>

- The great thing about this is that it works exactly like **C programming language** and we can do integer , float calculation and also can implement iterative statements functions etc.

- the **echo** statement is used to provide the expressions to the bc command.
- For creating a new variable inside echo we use simple name but if we want to use previous we use $ with it.

```
#!/bin/bash
#this is  simple bash file

read x
echo "$x"|bc;

echo "p=10;p"|bc;

# variable made inside echo can't be used outside
echo "some value $p";
```

- **Note that nothing will be printed after 'some value' as $p does not belong to global scope;**

Arithmetic operator Examples:

1. Finding Sum of Two expressions

```
$ echo "2+5" | bc
7
```

2. Difference of Two numbers

```
$ echo "10-4" | bc
6
```

3. Multiplying two numbers

```
$ echo "3*8" | bc
24
```

4. Dividing two numbers

When you divide two numbers, the bc command Ignores the decimal part and returns only the integral part as the output. See the below examples

```
$ echo "2/3" | bc
0
$ echo "5/4" | bc
1
```

- Simple bc gives integer output so if we wanted float we can use -l with bc

```
echo "22/7"|bc -l
```

OUTPUT-

```
3.14285714285714285714
```

- There is another way of doing this 'using' scale function

Use the scale function to specify the number of decimal digits that the bc command should return.

```
$ echo "scale=2;2/3" | bc
.66
```

5. Finding the remainder using modulus operator

```
$ echo "6%4" | bc
2
```

6. Using exponent operator

```
$ echo "10^2" | bc
100
```

Note- bash uses ** for power, bc uses ^.

- Assignment operator can be used same to C,

Assignment Operator Examples:

Assignment operators are used to assign a value to the variable. The following example shows how to use the assignment operators:

Assigns 10 to the variable and prints the value on the terminal.

```
$ echo "var=10;var" | bc
```

Increment the value of the variable by 5

```
$ echo "var=10; var+=5;var | bc
15
```

The lists of assignment operators supported are:

```
var = value   : Assign the value to the variable
var += value : similar to var = var + value
var -= value  : similar to var = var - value
var *= value  : similar to var = var * value
var /= value   : similar to var = var / value
var ^= value  : similar to var = var ^ value
var %= value : similar to var = var % value
```

- **Note that increment and decrement operators will output the variable but +=1, -=1 not.**

```
1
2 echo "without decremnet"
3 echo "var=10;var-=1;"|bc
4
5 echo "with decrement"
6 echo "var=10;var--;"|bc
7
```

OUTPUT-

```
without decremnet
with decrement
10
```

Relational Operators Examples:

- All relational operators of C language can be used in bc, returns 0 or 1

```
$ echo "10 > 5" | bc
1

$ echo "1 == 2" | bc
0
```

Logical Operator Examples:

'

```
$ echo "4 && 10" | bc
1
$ echo "0 || 0" | bc
0
```

Math Functions:

The built-in math functions supported are:

s (x) : The sine of x, x is in radians.
c (x) : The cosine of x, x is in radians.
a (x) : The arctangent of x, arctangent returns radians.
l (x) : The natural logarithm of x.
e (x) : The exponential function of raising e to the value x.
j (n,x): The bessel function of integer order n of x.
sqrt(x): Square root of the number x.

In addition to the math functions, the following functions are also supported.

length(x) : returns the number of digits in x
read() : Reads the number from the standard input.

- We will never use read() of bc as it may cause some problems, so use bash's 'read'

Conditional Statements-

- We can use C like if-else with 'bc' ,
- Python like print can be used to print things

The following example shows show to use the if condition

```
$ echo 'if(1 == 2) print "true" else print "false"' | bc
false
```

Iterative Statements:

- Loops can also be used just like C .

The following examples prints numbers from 1 to 10 using the for and while loops

```
$ echo "for(i=1;i<=10;i++) {i;}" | bc
$ echo "i=1; while(i<=10) { i; i+=1}" | bc
```

- For further details see
  http://www.unixmantra.com/2013/05/bc-unix-calculator.html

Comparing Value

Tuesday, December 13, 2016 12:40 AM

- For comparing we use double brackets [[ ]](for math we have use double parenthesis.
- Sometimes we will be able to use [](simple test) , but we will always use extended text ([[ ]]), because it contains some extending features like regular expression matching etc.
- By default all values are strings and string comparison will be done lexicographically(as C++/Java)

```
[[ expression ]]

        1: FALSE
        0: TRUE
```

(remember space between expression and bracket, also between operator and variable/value)

- In bash returns are opposite to C, 1 for false and 0 for true.

Comparison operations

Operation	Operator
Less than	[[$a < $b]]
Greater than	[[$a > $b]]
Less than or equal to	[[$a <= $b]]
Greater than or equal to	[[$a >= $b]]
Equal	[[$a == $b]]
Not equal	[[$a != $b]]

- We know, $? Represents result of previous command

```
#!/bin/bash
# This is a basic bash script.
[[ "cat" == "cat" ]]
echo $?

[[ "cat" = "dog" ]]
echo $?
```

OUTPUT-first true second false

```
0
1
```

- **Note- any = or == can be used with strings to compare ,because by [[ ]] we able to know that we are going to compare.**

- By default BASH compares values as string , so this will give following output

```
[[ 20 > 100 ]]
echo $?
```

OUTPUT-

```
0
```

- Output is 0(true) because 20 is lexicographically greater than 100 and bash by default take every value as a string value.
  so for comparing as numbers we have to use special comparison operators.

Comparison operators for numbers-

Operation	Operator
Less than	[[$a -lt $b]]
Greater than	[[$a -gt $b]]
Less than or equal to	[[$a -le $b]]
Greater than or equal to	[[$a -ge $b]]
Equal	[[$a -eq $b]]
Not equal	[[$a -ne $b]]

- Operations are same as strings but operators are different.
  To learn - all can be formed by first letter of their operations & all of two characters.
- Now if we used this operation then

```
[[ 20 -gt 100 ]]
echo $?
```

OUTPUT-

```
1
scott@orion:~$
```

Alternate way for comparing numbers

- We can also compare numbers using (( )) operators, in them we can use simple operators like(<,>,=) etc.
- Remember 0 is true and 1 is false;

```
root@kali:~/Desktop# (( 20 < 100 )); echo $?
0
root@kali:~/Desktop# (( 20 > 100 )); echo $?
```

```
root@kali:~/Desktop# (( 20 < 100 )); echo $?
0
root@kali:~/Desktop# (( 20 > 100 )); echo $?
1
root@kali:~/Desktop# ((100 == 100 )); echo $?
0
```

```
root@kali:~/Desktop# ((100 <= 100 && 20 > 40 )); echo $?
1
root@kali:~/Desktop# ((100 <= 100 || 20<40 )); echo $?
0
```

Logic operations

Operation	Operator
Logical AND	[[$a && $b]]
Logical OR	[[$a || $b]]
Logical NOT	[[! $a]]

- Null String (empty string) value can be checked by following operations.

String null value

Operation	Operator
Is null?	[[-z $a]]
Is not null?	[[-n $a]]

Example-

```
a=""
b="cat"
[[ -z $a && -n $b ]]
echo $?
```

This will return true(0) only if a is null and b is not null
OUTPUT-
0

Regular Expression matching

- to check if a string matches some regular expression we use =~ sign

```
a="This is my string!"
if [[ $a =~ [0-9]+ ]]; then
        echo $a is greater than 4!
else
        echo $a is not greater than 4!
fi
```

Rule of thumb -

- For string use [[ ]](only possible way)
- For number use (( )) and if use [[ ]] use special operators(-gt,-eq etc);

Command Substitution

Monday, December 12, 2016 6:51 PM

- Sometimes we wanted to store value of some command in a variable , This could be done by command Substitution.
- Command substitution will suppress output and send it to variable

```
d=$(pwd)
echo $d
```

- If we haven't used $ before 'pwd' command then we have d=pwd which means 'd' store 'pwd' not the result of command 'pwd';
- Example- find ping for example.com

```
#!/bin/bash
# This is a basic bash script.
a=$(ping -c 1 example.com | grep 'bytes from' | cut -d = -f 4)
echo "The ping was $a"
```

- Now the ping value is available to us in form of a variable 'a', which we can use later in our script.
- OUTPUT-

```
scott@orion:~$ ./my.sh
The ping was 1.66 ms
```

Working with strings

Tuesday, December 13, 2016 1:03 AM

- indexing is 0 based in bash(not like mathematica)

Concatenation

- Concatenation of strings can be done by putting string together **without any space** between them.

```
scott@orion:~$ a="hello"
scott@orion:~$ b="world"
scott@orion:~$ c=$a$b
scott@orion:~$ echo $c
helloworld
```

length of string

- Length of string can be found out by #(pound or hash) sign

```
scott@orion:~$ echo ${#a}
5
scott@orion:~$ echo ${#c}
10
```

substring

- We can braces {,} to extract substring from a string
  here c is helloword(from previous example)

  syntax- ${str:start:no_of_char}

- Following will extract substring staring from 3 and to end

```
scott@orion:~$ d=${c:3}
scott@orion:~$ echo $d
loworld
```

- Following will extract starting from 3 and 4 characters

```
scott@orion:~$ e=${c:3:4}
scott@orion:~$ echo $e
lowo
```

- We can also use negative indexes like python and mathematica, **but we have to place a space before - sign**
  - So following will extract last 4 characters

```
scott@orion:~$ echo ${c: -4}
orld
```

- Following will extract start 3 characters from last 4

```
scott@orion:~$ echo ${c: -4:3}
orl
```

replace a part of string

- We can also use sed tool to replace text of input stream of file as in following example

```
fruit="apple banana banana cherry";
#initial value
echo  $fruit;

#replacing all banana with manago
fruit=$(echo $fruit|sed 's/banana/mango/g');

#outputting final value
echo $fruit;
```

OUTPUT-

```
root@kali:~/Desktop# ./ap.sh
apple banana banana cherry
apple mango mango cherry
```

- We can also use bash inbuilt functionality to replace for variables
  Syntax - **${var/old/new}**

```
scott@orion:~$ fruit="apple banana banana cherry"
scott@orion:~$ echo ${fruit/banana/durian}
apple durian banana cherry
```

- But this will only replace first occurance in variable , for replacing all we have to use double slace(//) before search term(telling all replace karo)

```
scott@orion:~$ echo ${fruit//banana/durian}
apple durian durian cherry
```

EXAMPLE-

```
fruit="apple banana banana cherry";
#initial value
echo  $fruit;

#replacing all banana with manago
fruit=${fruit//banana/mango};

#outputting final value
echo $fruit;
```

OUTPUT-

```
root@kali:~/Desktop# ./ap.sh
apple banana banana cherry
apple mango mango cherry
```

Note - for modifying the variable itself we have assign it back to it.

- The extra / used is called modifier and there are also couple of modifiers we can use
  - # - replace only if search term is in the starting of string
  - % - replace only if search term is at the end

```
scott@orion:~$ echo ${fruit/#apple/durian}
durian banana banana cherry
scott@orion:~$ echo ${fruit/#banana/durian}
apple banana banana cherry
scott@orion:~$ echo ${fruit/%cherry/durian}
apple banana banana durian
scott@orion:~$ echo ${fruit/%banana/durian}
apple banana banana cherry
```

- We can use matching(*) with this operation too like
  - In this we are replacing first occurance of anythign starts with 'c'

```
scott@orion:~$ echo ${fruit/c*/durian}
apple banana banana durian
```

Coloring and Styling texts

Tuesday, December 13, 2016 5:48 PM

- There are two ways of styling and coloring text output in bash
  - ASCII escape codes
  - Using 'tput' command

METHOD 1- Ansii escape codes

- For using ascii codes we have to use use -e opetion with echo ,which tells the interpreter that we are going to used ansii escape codes.
- Example - to print something in green color

```
scott@orion:~$ echo -e "\033[34;42mColor Text\033[0m"
Color Text
scott@orion:~$
```

  - **Echo -e** is used to tell that we are going to escape
  - **\033[** is used to tell start of escaping and this ends with m
    simillary **\033[** is used to tell end of escaping that also ends with m
  - Numbers after **\033**[and before **m** tells styling and colors of text inside escaping
- Number after **[** and before **m** corresponds to styling foreground and background colors

syntax- **\033[<style_code>;<forground_code>;<background_code>m**

- If not given the default will be used.
  Note - we have **\033[0m** at last, in that 0 specifies no-style.

Colored text (ANSI)

Color	Foreground	Background
Black	30	40
Red	31	41
Green	32	42
Yellow	33	43
Blue	34	44
Magenta	35	45
Cyan	36	46
White	37	47

- Color code goes from 0 to 7, 3 used for fg and 4 for bg

Color examples (ANSI)

Color	Foreground
White on Black	`echo -e '\033[37;40mWhite on Black\033[0m'`
Black on Red	`echo -e '\033[30;41mBlack on Red\033[0m'`
Green on Black	`echo -e '\033[32;40mGreen on Black\033[0m'`
Red on White	`echo -e '\033[31;47mRed on White\033[0m'`
Blue on Yellow	`echo -e '\033[34;43mBlue on Yellow\033[0m'`

Styled text (ANSI)

Style	Value
No Style	0
Bold	1
Low Intensity	2
Underline	4
Blinking	5
Reverse	7
Invisible	8

Example- to print error message ERROR should be blinking itself?

```
#!/bin/bash
# This is a basic bash script.
echo -e "\033[5;31;40mERROR: \033[0m\033[31;40mSomething went wrong.\033[0m"
```

- Here we open styling and then write 'ERROR' then closed that as other part has other type of styling, then for other part we used styling.
- In the first part we used style 5 which is blinking but we do not need that for second part.

OUTPUT-

- **ERROR** in output is blinking so we are having two following states.

```
scott@orion:~$ ./my.sh
ERROR: Something went wrong.
scott@orion:~$
```

```
scott@orion:~$ ./my.sh
ERROR: Something went wrong.
scott@orion:~$
```

- We can create variables to store starting and ending so we can use them easily.

```
flashred="\033[5;31;40m"
red="\033[31;40m"
none="\033[0m"
echo -e $flashred"ERROR: \033[31;40mSomething went wrong.\033[0m"
```

  - This gives same output as above(flashing ERROR and message)

METHOD 2- using tput utility

- Tput can used to create styled text and it is more easy to use and roburst than ansii code.
- As 'tput' is a command we have to use command substitution.

Styled text (tput)

Style	Command
Foreground	`tput setaf [0-7]`
Background	`tput setab [0-7]`
No Style	`tput sgv0`
Bold	`tput bold`
Low Intensity	`tput dim`
Underline	`tput smul`
Blinking	`tput blink`
Reverse	`tput rev`

- Color codes are same as before not we don't have to use 3 and 4(we use seta+(f or b))

Color	setaf	setab
Black	0	0
Red	1	1
Green	2	2
Yellow	3	3
Blue	4	4
Magenta	5	5
Cyan	6	6
White	7	7

example- doing previous thing with tput.

```
flashred=$(tput setab 0; tput setaf 1; tput blink)
red=$(tput setab 0; tput setaf 1)
none=$(tput sgr0)
echo -e $flashred"ERROR: "$none$red"Something went wrong."$none
```

For more see **$man terminfo**

Date and Cal command

Tuesday, December 13, 2016 8:09 PM

- Date command can be used to get information about date and time

```
scott@orion:~$ date
Thu Oct 17 21:06:18 UTC 2013
```

- We can also use our formatting by specifying + after date

```
scott@orion:~$ date +"%d-%m-%Y"
17-10-2013
scott@orion:~$ date +"%H-%M-%S"
21-07-31
scott@orion:~$ man date
```

(for formatting see **$man date**)

- Cal shows calander (see notes for more)

Printf command

Tuesday, December 13, 2016 7:41 PM

- Printf formates data same as C's printf

- Basic structure of command
  - Printf FORMAT [ARGUMENT]

  ([] means may or may not be)

```
scott@orion:~$ printf "Name:\t%s\nID:\t%04d\n" "Scott" "12"
Name:   Scott
ID:     0012
scott@orion:~$ printf "Name:\t%s\nID:\t%04d\n" "Someone Else" "123"
Name:   Someone Else
ID:     0123
```

- If we wanted to assign output of printf to some variable we can use **-v var_name** before starting outputting, this will tell printf to not print anything but assign it to var_name variable.
  example - **printf -v var "this is output";**

- Ofcourse we can also use command substitution like
  Example- **var=$(printf "this is output");**
  (both will suppress printing to terminal and tell them to assign value to variable instead)

```
#!/bin/bash
# This is a basic bash script.
today=$(date +"%d-%m-%Y")
time=$(date +"%H:%M:%S")
printf -v d "Current User:\t%s\nDate:\t\t%s @ %s\n" $USER $today $time
echo "$d"
```

OUTPUT-

```
scott@orion:~$ ./my.sh
Current User:   scott
Date:           17-10-2013 @ 21:16:55
```

Checkout others at http://wiki.bash-hackers.org/commands/builtin/printf

Reading and Writing to text file

Tuesday, December 13, 2016 9:20 PM

- We can't use bash to read and write from binary files, but we can work with text files.
- Less than(<) and greater than(>) sign are key here
    - '<' used for input(outputting from file's perspective)

'>' sign

- '>' used for outputting to file(input from file's perspective)
- It creates file if not exist and overwrite it if it exits.

Example-

```
scott@orion:~$ echo "Some text" > file.txt
scott@orion:~$ cat file.txt
Some text
```

- We can use '>' sign to make a file empty by outputting nothing to it.

```
scott@orion:~$ > file.txt
scott@orion:~$ cat file.txt
```

- We can append to a file (if exist) using **>>** symbol instead of **>.** (it will also create if not exist)

```
scott@orion:~$ echo "Some text" > file.txt
scott@orion:~$ echo "Some more text" >> file.txt
scott@orion:~$ cat file.txt
Some text
Some more text
```

'<' sign

- < symbol is used for taking input from a file(reading a file).(output from file's perspective)
- Example - reading a file and outputting its content

```
scott@orion:~$ cat < file.txt
Some text
Some more text
```

Arrays

Tuesday, December 13, 2016 8:31 PM

- Arrays in bash are make with single parenthesis (that is why for math we are using 2)
- Arrays are 0 based in bash (like c, not like mathematica)

Making arrays

```
#!/bin/bash
# This is a basic bash script.
a=()
b=("apple" "banana" "cherry")
```

- In above example a becomes empty array and b having 3 elements
  note - there is not comma between elements(separated by space)

Accessing elements in array

- We use following syntax for accessing element

```
echo ${b[2]}
```

(This will output cherry)

Setting value and extending Array

- We can set value to array just like C, we do not need to use dollar(same as variable)

- If we want to append a value with array we can do using **+=** operator

```
b+=("mango")
```

  (this will append value to array $b)
- If we use index which do not exist then we are extending array upto that index,
  and if middle index are empty then will be give default value.

```
b[5]="kiwi"
```

(note initially we have 3 elements in $b so index upto 2 was there but now we have 6 element and 6th element is "kiwi")

Displaying value

- We can use index as stated earlier.
- If we haven't used any index then we get first element
- We can use @ operator to access all elements of an array.(null element not shown)

```
echo ${b[@]}
```

  (only those element will come which are not null, so in case of sparse arrays{comes if we have added an element leaving some between empty}
  We not going to see those are in between and empty.)

example-

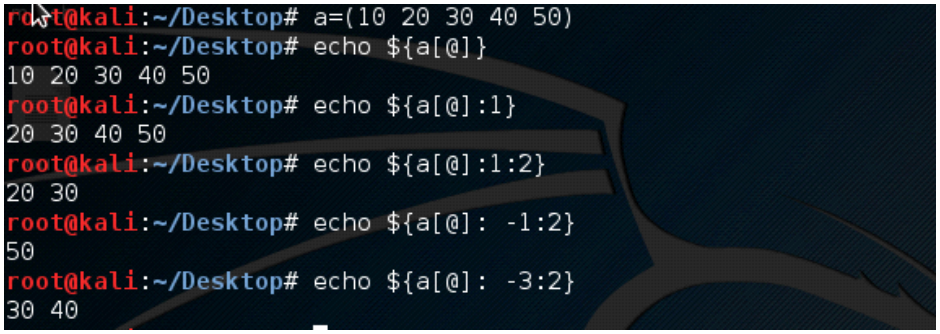
```
#!/bin/bash
#this script shows error msg

a=("ashish" 10 "patel" "value")
echo ${a[@]};
a[10]="hello"
echo ${a[@]};
```

OUTPUT-

```
root@kali:~/Desktop# ./my.sh
ashish 10 patel value
ashish 10 patel value hello
```

- We can also use range (not step) in this as we did in strings
  syntax- **${arr[@]:start:elements}**

  Examples-



- We can use negative as we did in second last , where we asked for 2 element from last 1(which is only 1)

Associative array

- availabe in **bash 4 and above**
- We have to used **declare** keyword while declaring to make associative array
  (recall **declare** also used when we want to specify type of variable)

```
#!/bin/bash
# This is a basic bash script.
declare -A myarray
myarray[color]=blue
myarray["office building"]="HQ West"

echo ${myarray["office building"]} is ${myarray[color]}
```

- In first line we declared an associative array using declare keyword
- And we can insert value as we did in simple arrays
- And now we can access values using keys

OUTPUT-

```
scott@orion:~$ ./my.sh
HQ West is blue
```

Note 1-we can use ${myarray[@]} to output whole array but it will output only value (not keys)
2-we can use Only -A as -a will not work

Some Operations on arrays

- echo ${arr[@]} - output all values of array
- echo ${!arr[@]} - output all keys of arrays
- echo ${#arr[@]} - number of elements in the arrray
- echo ${#arr} or echo ${#arr[0]} - output length of first element
- echo ${#arr[i]}- output length of ith index element

Here document

Saturday, December 17, 2016 11:38 PM

- **here** document in bash lets us to specify input freely for a command.
- '<<' specifies **here** document and input will be feeded into the command until endstring comes
- Obviously we have to use **endstring** such that it will not come in input.
- Input will stop only when "endstring" comes in new line alone.

Syntax- command << **endstring**

---

input

---

endstring

Example- this example feeds lot of input into command **cat.**

```
#!/bin/bash
# This is a basic bash script.
cat << EndOfText
This is a
multiline
text string
EndOfText
```

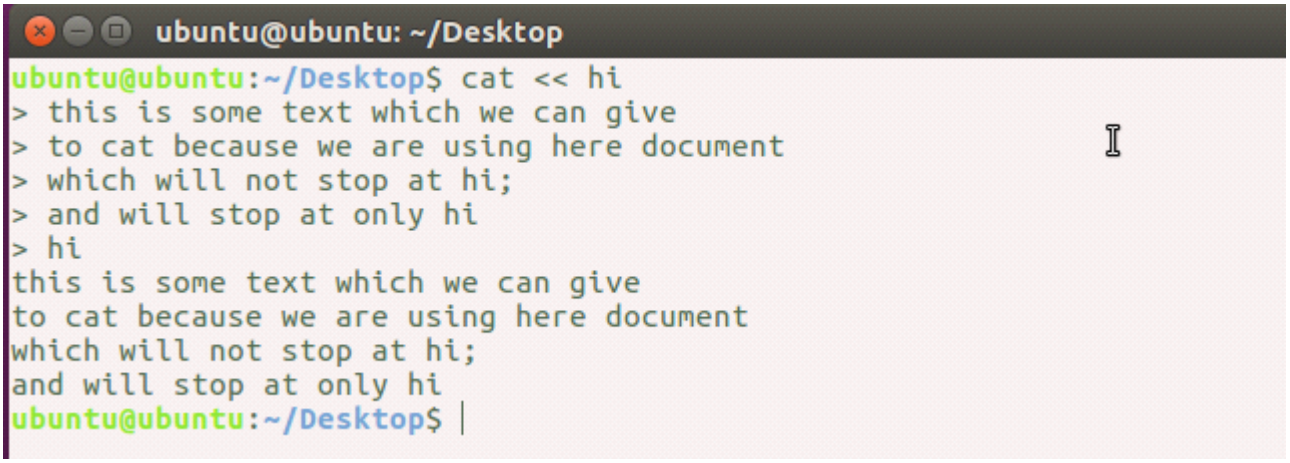
(here all thing between "EndOfText' is feeded to cat command which will output it.)

Output-

```
scott@orion:~$ ./my.sh
This is a
multiline
text string
scott@orion:~$
```

- There is one more interesting option to know in **here** Document , dash(-) option.
- This option will make bash to skip starting tab input so that we can indent input nicely to make it visible properly.(output same as before)

Example- in ubuntu



- if we wanted to out file content to some file say hi.txt we can do as fillows

```
 $cat <<hi > hi.txt
This is input
hi
```

If statements

Sunday, December 18, 2016 12:05 AM

If statement

- If statement executes code based on the truth value of expression.
- We can write 'then' in same line but we have to use ';'

```
if expression;then
```

Or we can us them on other lines

```
if expression
then
  echo “True”
fi
```

- 'then' will be used to start(same as { in c) and 'fi' used as end (same as } in c)

If else statement

- If else represent two opposite statement in which only one executed at a time.

```
if expression
then
  echo “True”
else
  echo “False”
fi
```

If elif statement

- We can use if-elif-else to have multiple cases

```
if expression
then
  echo “True”
elif expression2; then
  echo “ex is False, e2 is True”
fi
```

examples

Sunday, December 18, 2016 12:48 AM

Example 1- to test a variable less than or equal to 4

```
a=5
if [ $a -gt 4 ]; then
        echo $a is greater than 4!
else
        echo $a is not greater than 4!
fi
```

(here we can use simple test([]) but we will always use ([[ ]]) as it is more advance)

Example 2- to check if a string matches some regular expression we use
=~ sign

```
a="This is my string!"
if [[ $a =~ [0-9]+ ]]; then
        echo $a is greater than 4!
else
        echo $a is not greater than 4!
fi
```

while loop

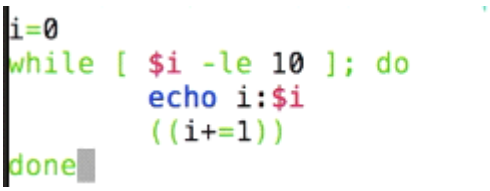
Sunday, December 18, 2016 1:02 AM

- Like if statement it is needed to have some thing so that we could be able to identify starting end ending point loop.
- So here we use '**do**' and '**done**' we have used 'then' and 'fi' in 'if' statements

- Example- printing from 1 to 10 using integer comparison{using(( )) }

```
i=1
while (( $i<=10 ));do
echo $i;
((i++))
done;
```

- Example- printing from 1 to 10 using simple test{using [] }



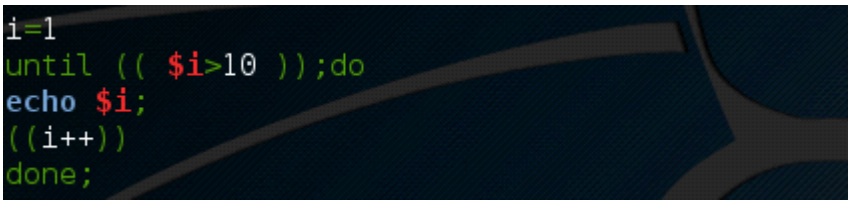
```
i=0
while [ $i -le 10 ]; do
        echo i:$i
        ((i+=1))
done
```

Until loop

Sunday, December 18, 2016 1:14 AM

- Until loop is counterpart to while loop
- It applies condition in reverse order as it does something the **condition not become true.**

- **Example-** printing 1 to 10



```
i=1
until (( $i>10 ));do
echo $i;
((i++))
done;
```

- **Example-** printing until 10 comes

```
j=0
until [ $j -ge 10 ]; do
        echo j:$j
        ((j+=1))
done
```

For loops

Sunday, December 18, 2016 1:18 AM

- For loop is used to traverse according to some specific criteria usually a variable or range.

Looping through some values-

- Example - to iterate through some given value(here 1, 2 and 3)

```
#!/bin/bash
# This is a basic bash script.
for i in 1 2 3
do
        echo $i
done
```

(value is assigned to 'I' one by one and printed)

OUTPUT-

1
2
3

Looping through brace expansion -

- We can use brance expansion here, so all value of brace expansion will come to I
  And we can do whatever we wanted to do with it.

example - print 1 to 100

```
#!/bin/bash
# This is a basic bash script.
for i in {1..100}
do
        echo $i
done
```

- We can also specify interval(as before)
  example - printing odd number between [1, 100]

```
#!/bin/bash
# This is a basic bash script.
for i in {1..100..2}
do
        echo $i
done
```

C style for loop -

- We also 'C' style version of for loop
  Example- printing 1 to 10

```
#!/bin/bash
# This is a basic bash script.
for (( i=1; i<=10; i++ ))
do
        echo $i
done
```

Looping through an array -

- Although we can use 'while' loop also (using I as index), but it is pretty easy to use 'for'

loop with arrays.

- For this, we will use 'in' keyword(as in python) and use '@' for getting whole array.

  Example- printing content of a array variable;

```
arr=("apple" "banana" "cherry")
for i in ${arr[@]}
do
        echo $i
done
```

Looping through an Associative array -

- Looping through associative array is little bit tricky , first we will get all keys using '${!arr[@]}' now these key will be assigned to 'I' one by one which we can use to get real
  element of array.

```
#!/bin/bash
# This is a basic bash script.
declare -A arr
arr["name"]="Scott"
arr["id"]="1234"
for i in "${!arr[@]}"
do
        echo "$i: ${arr[$i]}"
done
```

(we have to use '$i' as key to get the value).

Looping through output of some command

- We can use command substitution to get output of some command in 'I' and then use it
- Accordingly.
- Example- print all files names in present directly using loop.

```
#!/bin/bash
# This is a basic bash script.
for i in $(ls)
do
        echo "$i"
done
```

(as ls returns files names, name of each will be passed to 'I' one by one and then can be printed)
OUTPUT-

```
scott@orion:~$ ./my.sh
auth.log
error.txt
fruit
log.txt
my.sh
otherfolder
pets
success.txt
trees
```

Switch case statement

Sunday, December 18, 2016 2:11 AM

- Switch case is implemented using 'case' statement in bash
- Example -

```
#!/bin/bash
# This is a basic bash script.
a="dog"
case $a in
        cat) echo "Feline";;
        dog|puppy) echo "Canine";;
        *) echo "No match!";;
esac
```

  - Starting done with 'in' and end is done with reverse case ie. 'esac'
  - We can show a option left to ')' parantheses
  - We can end result of some option with double semicolon(;;) as single is reserved for statement ending and if we have multiple statement we have to use it after each statement.
  - Default is given by option '*'
  - We can give same output for one or more option(as we do with multiple case in c) by using pipe operator;

functions

Sunday, December 18, 2016 2:18 AM

- Basic syntax of a function is as follows

```
#!/bin/bash
# This is a basic bash script.
function greet {
        echo "Hi there!"
}
```

- Now we can call the function just by name(as it do not pass any value)

```
echo "And now, a greeting!"
greet
```

OUTPUT-

```
scott@orion:~$ ./my.sh
And now, a greeting!
Hi there!
```

Passing argument to function

- We can pass value to function by just putting value to be passed ahead to function name with space in between.
- In function definition also we do not to take it in variable
  $i will store the first passed value and so on….

- Following example shows a function which do greeting using a argument to funciton.

```
#!/bin/bash
# This is a basic bash script.
function greet {
        echo "Hi, $1"
}

echo "And now, a greeting!"
greet Scott
```

OUTPUT-

```
scott@orion:~$ ./my.sh
And now, a greeting!
Hi, Scott
```