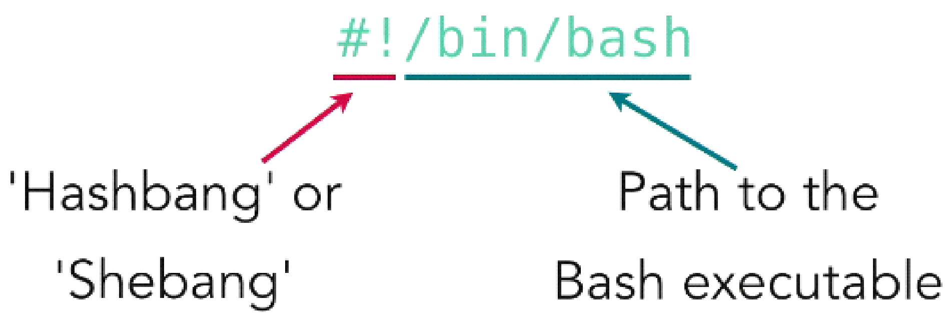
Introduction

Sunday, December 11, 2016 8:50 PM

Bash Syntax

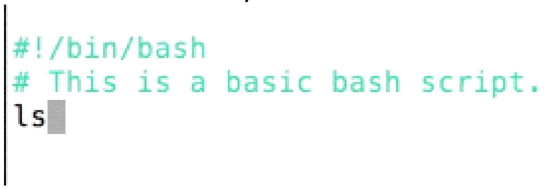
- Every file starts with ***interpreter directive*** Which tells interpreter that it is a bash file once it is made executable.
- Interpreter directive is Also called as Hashbang or shebang



- Usually executable path is /bin/bash

Creating and Running Bash Scripts

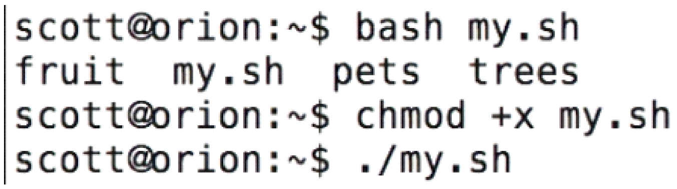
- Let we have a file my.sh like this



```
#!/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-



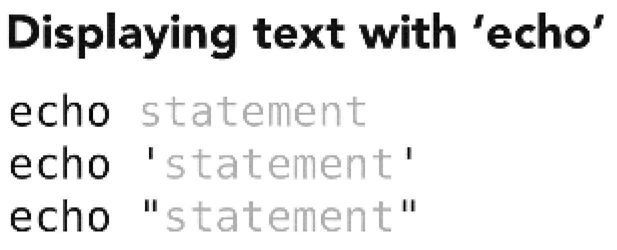
```
scott@orion:~$ bash my.sh
fruit  my.sh  pets  trees
scott@orion:~$ chmod +x my.sh
scott@orion:~$ ./my.sh
```

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



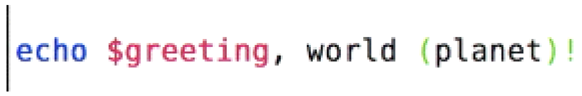
Displaying text with 'echo'

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

Without quotation -

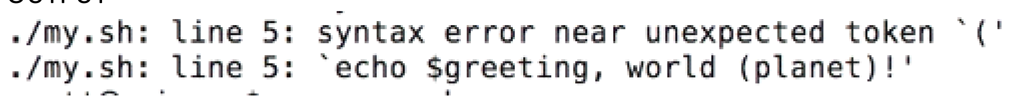
- we have to skip each and every special symbol we encounter.
  example- in following statement we get error as paranthesis are special symbols so we have to escape them.

STATEMENT 1



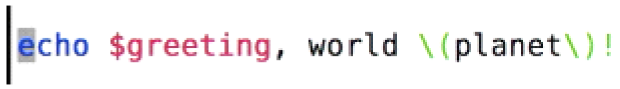
```
echo $greeting, world (planet)!
```

OUTPUT-



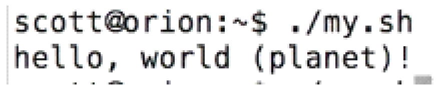
```
./my.sh: line 5: syntax error near unexpected token `('
./my.sh: line 5: `echo $greeting, world (planet)!'
```

STATEMENT 2



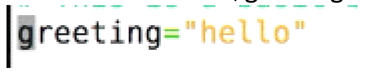
```
echo $greeting, world \(planet\)!
```

OUTPUT-



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

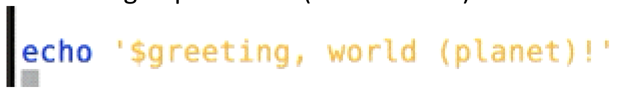
Note- the value of $greeting comes from following variable



```
greeting="hello"
```

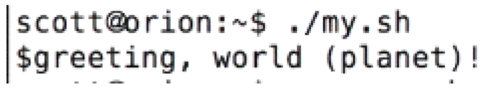
Single quotation-

- 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)!'
```

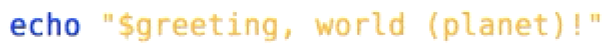
OUTPUT-



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

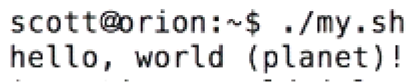
Double quotation-

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



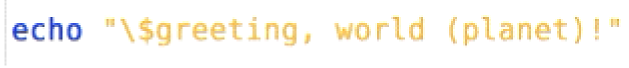
```
echo "$greeting, world (planet)!"
```

OUTPUT-



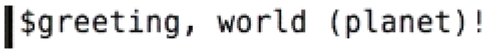
```
scott@orion:~$ ./my.sh
hello, world (planet)!
```

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



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

OUTPUT-

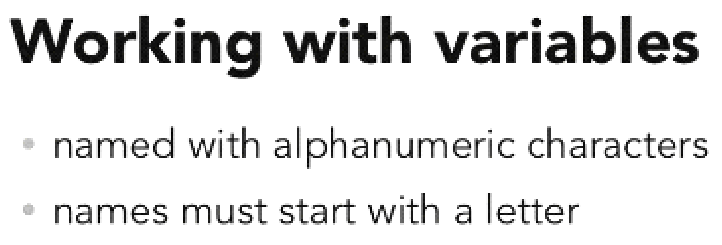


```
$greeting, world (planet)!
```

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

Variables

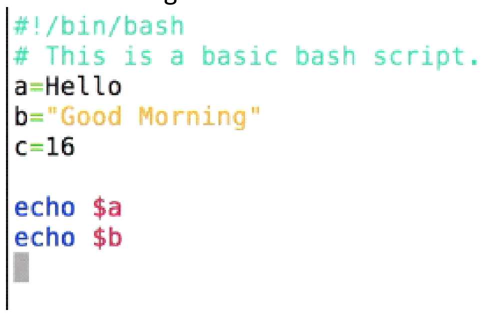
Monday, December 12, 2016 1:16 AM



Working with variables

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

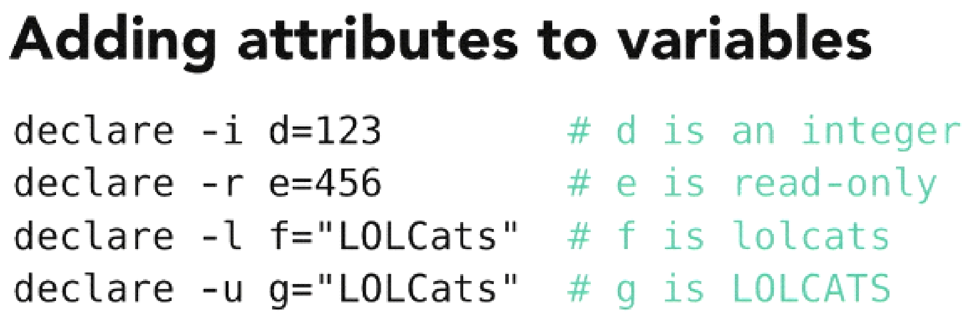
- While defining variable there should not be any space between variable, '=' and value otherwise we will get error.



```
#!/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`

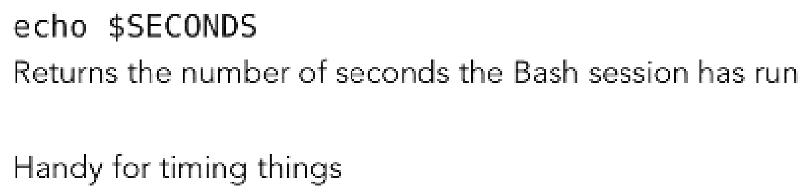
BASH_VERSION variable

```
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

RANDOM variable

echo $RANDOM

Return a random number.

Special variables

- **'*' -** all files/directories in pwd
- '$*' or '$@'- 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**

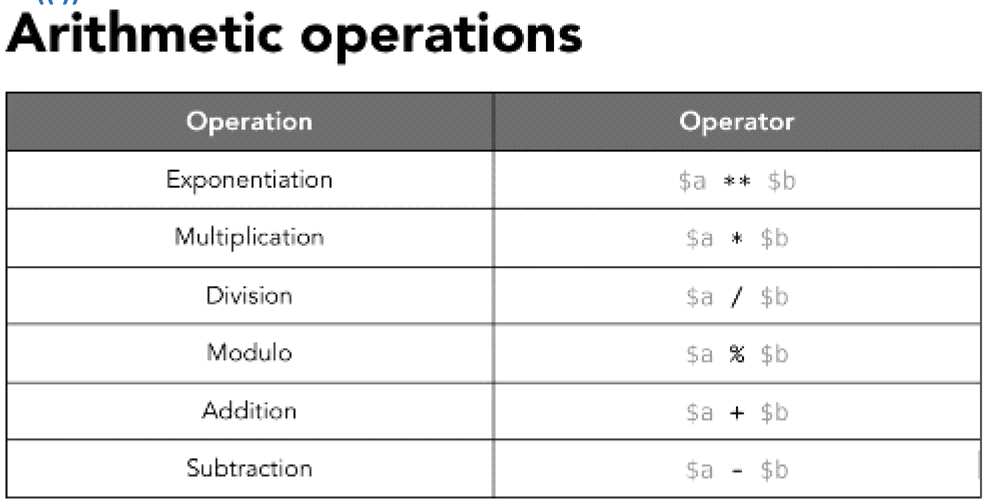
```
(( expression ))
```

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 ))
  ```

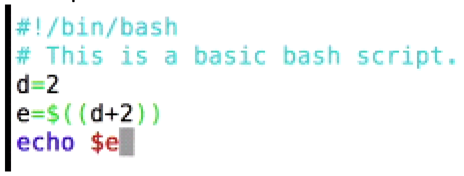
- **An important thing to note here is that we do not need to use '$' infront of a variable inside (( )).**



Arithmetic operations

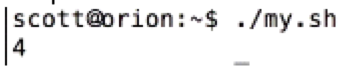
Operation	Operator
Exponentiation	`$a ** $b`
Multiplication	`$a * $b`
Division	`$a / $b`
Modulo	`$a % $b`
Addition	`$a + $b`
Subtraction	`$a - $b`

- Bash supports 6 basic arithmetic operations.
  Example-



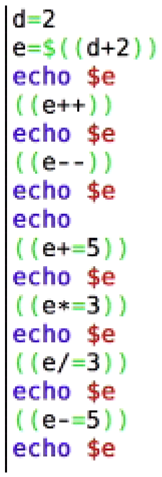
```
#!/bin/bash
# This is a basic bash script.
d=2
e=$((d+2))
echo $e
```

Output-



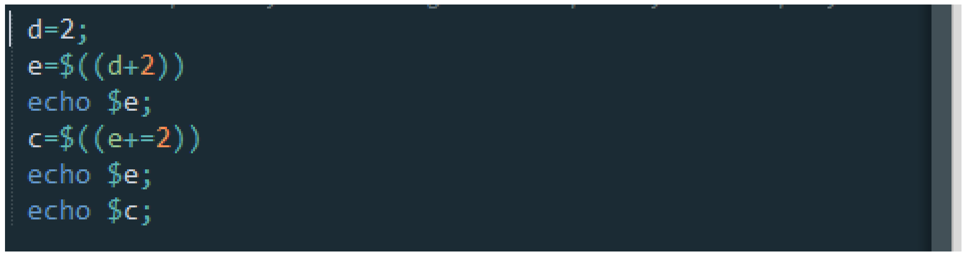
```
scott@orion:~$ ./my.sh
4
```

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



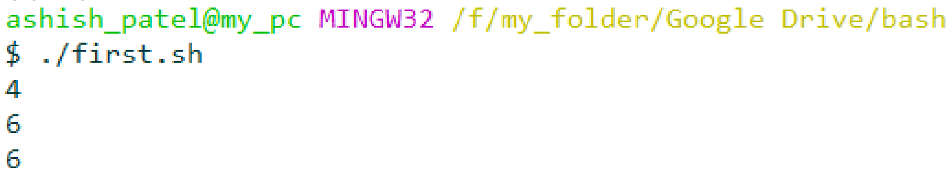
```
d=2
e=$((d+2))
echo $e
((e++))
echo $e
((e--))
echo $e
echo
((e+=5))
echo $e
((e*=3))
echo $e
((e/=3))
echo $e
((e-=5))
echo $e
```

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

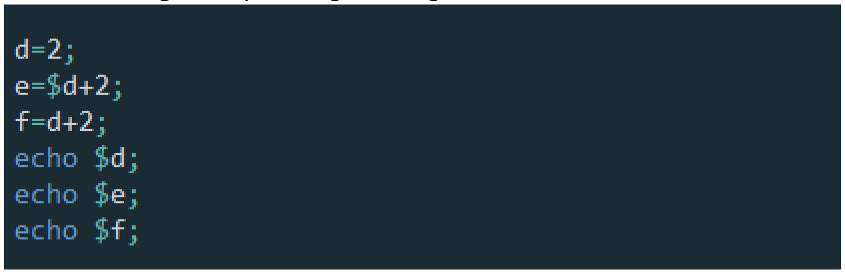


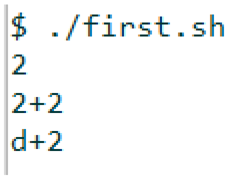
```
d=2;
e=$((d+2))
echo $e;
c=$((e+=2))
echo $e;
echo $c;
```

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

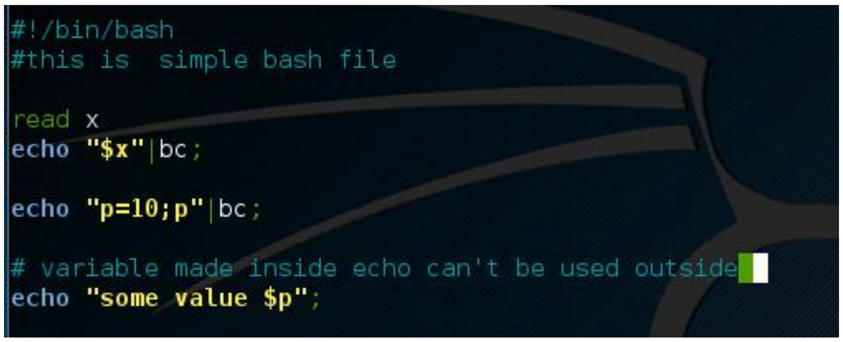
- The bc command evaluates expressions and has syntax similar to the c programming language.

- The bc command supports the following features.
    - Arithmetic operators
    - Increment and decrement operators
    - Assignment operators
    - Comparison 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.

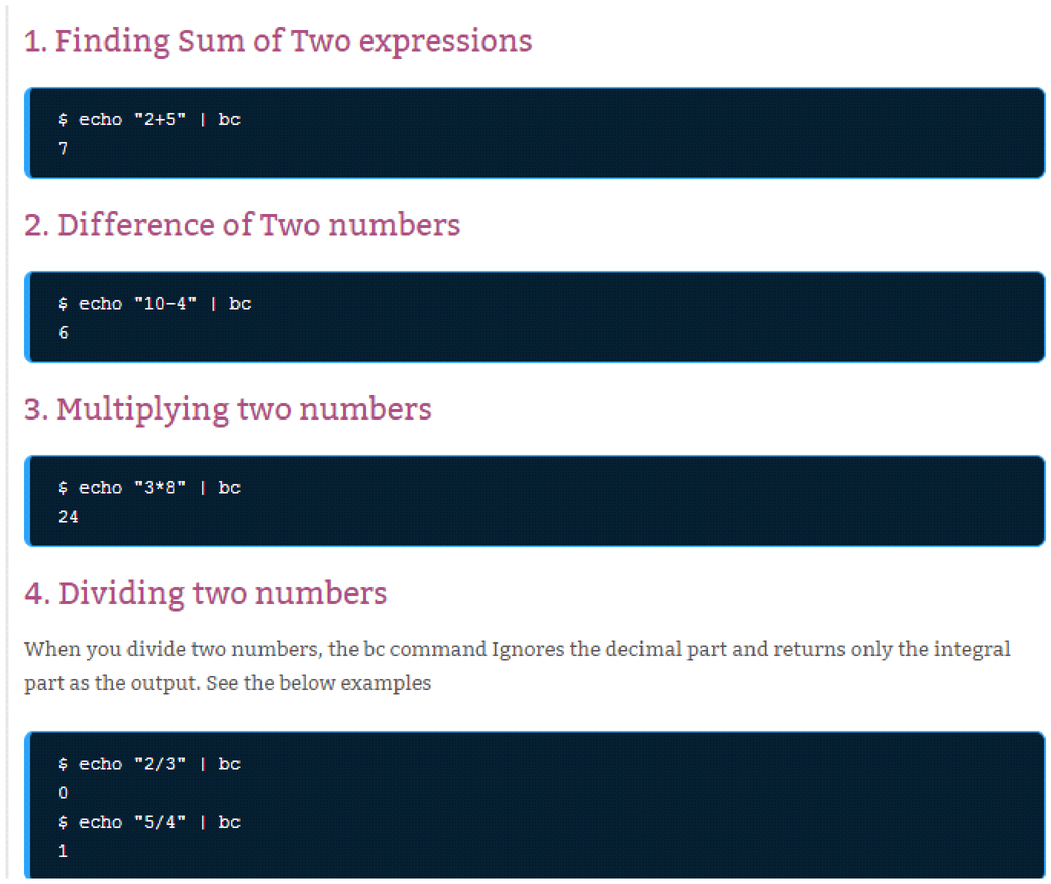
- the **echo** statement is used to provide the expressions to the bc command.
- For creating a new variable inside echo we use simple name but if we want to use previous we use $ with it.



```
#!/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

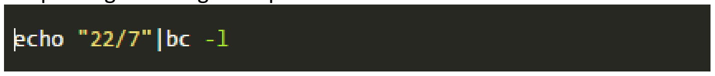
```
$ echo "3*8" | bc
24
```

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

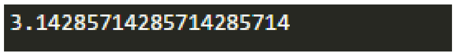
```
$ echo "2/3" | bc
0
$ echo "5/4" | bc
1
```

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



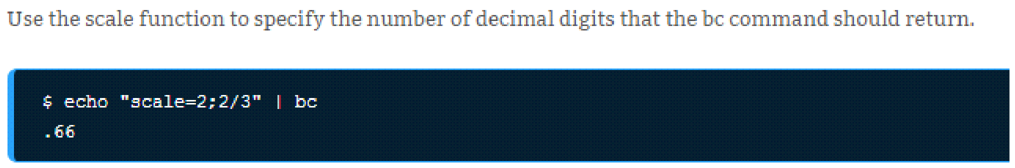
```
echo "22/7"|bc -l
```

OUTPUT-



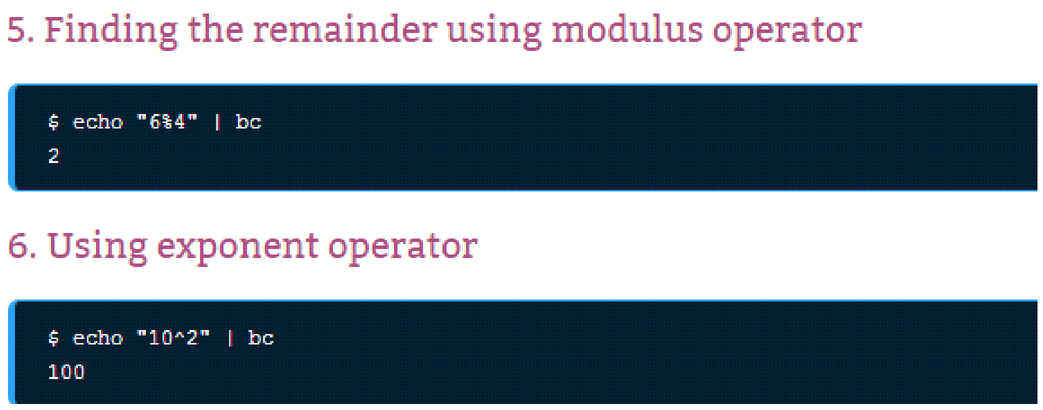
```
3.14285714285714285714
```

- 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

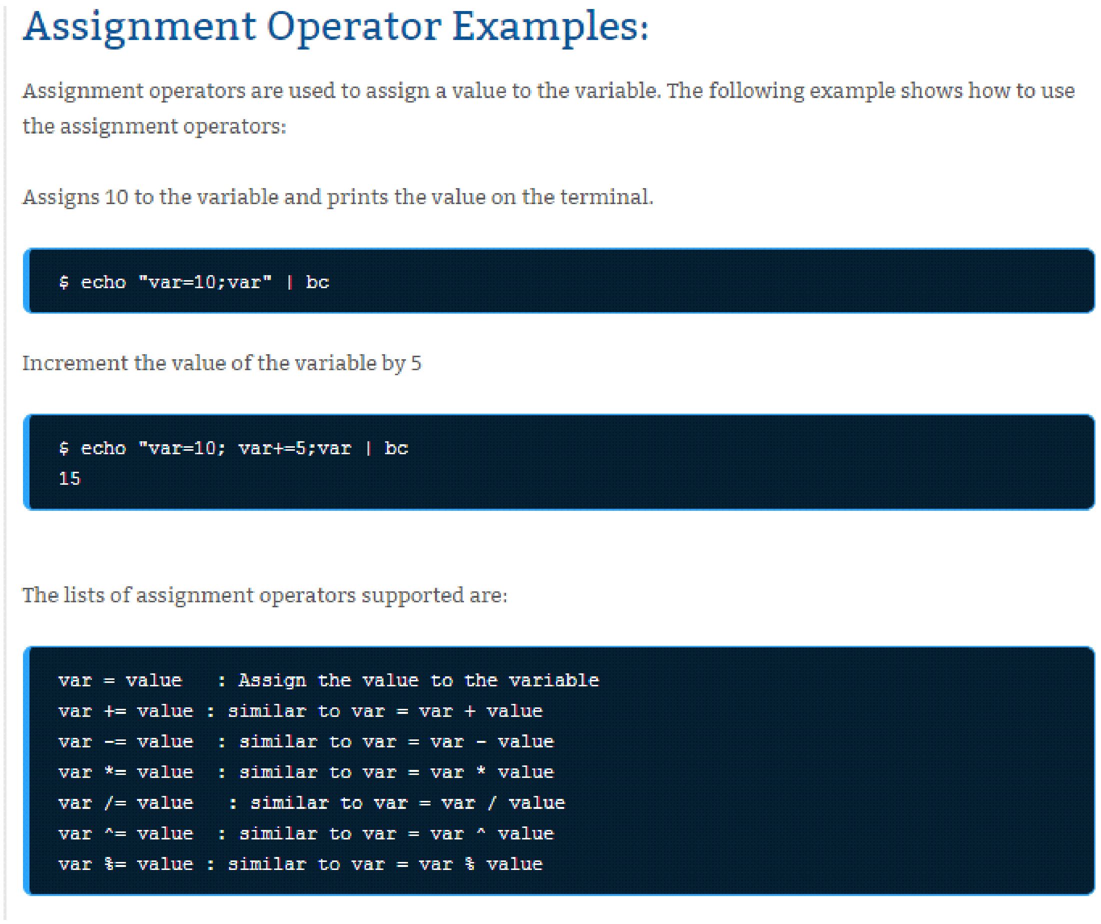
```
$ echo "6%4" | bc
2
```

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
```

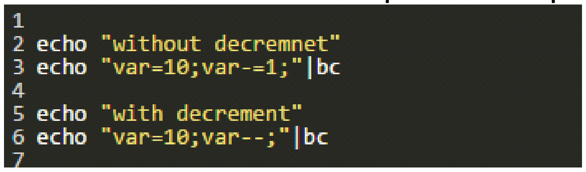
Increment the value of the variable by 5

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

The lists of assignment operators supported are:

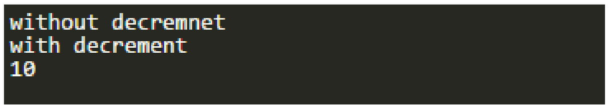
```
var = value   : Assign the value to the variable
var += value : similar to var = var + value
var -= value  : similar to var = var - value
var *= value  : similar to var = var * value
var /= value   : similar to var = var / value
var ^= value  : similar to var = var ^ value
var %= value : similar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

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



```

echo "without decremnet"
echo "var=10;var-=1;"|bc

echo "with decrement"
echo "var=10;var--;"|bc
```

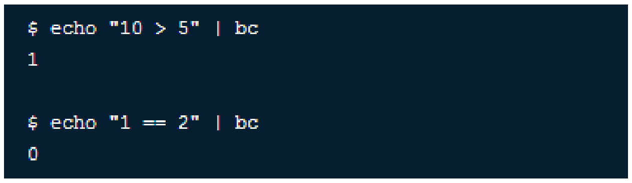
OUTPUT-



```
without decremnet
with decrement
10
```

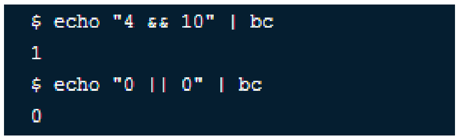
Relational Operators Examples:

- 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.

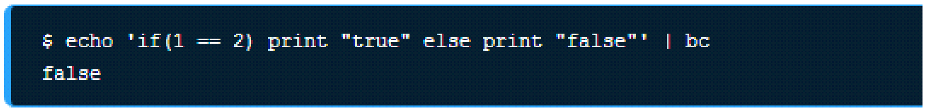
length(x) : returns the number of digits in x
read() : Reads the number from the standard input.

- 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

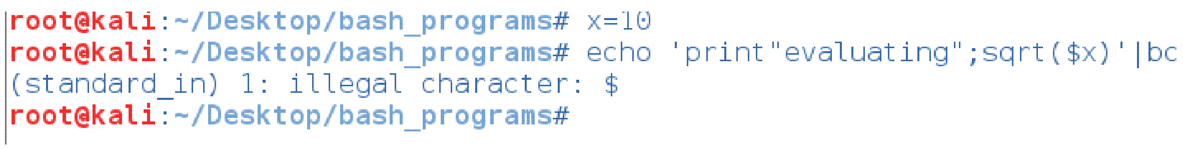
The following example shows show to use the if condition



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

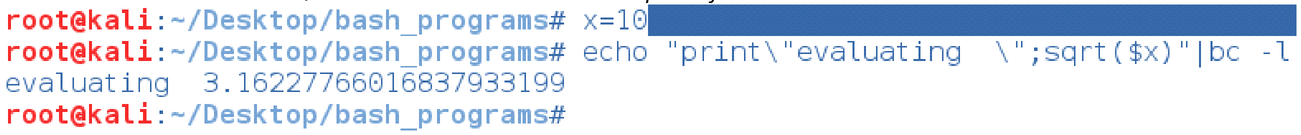
Problem with print and echo working together -

- we have used 'single quote' for 'echo' as it understand both single and double quotes while 'print' only understand 'double quotes' , But now there is a problem, if we have to use variable from global scope in 'bc' command then it can't be evaluated by 'echo' we will get error;
  EXAMPLE - here we want to evaluate x inside single quote which is not possible.



```
root@kali:~/Desktop/bash_programs# x=10
root@kali:~/Desktop/bash_programs# echo 'print"evaluating";sqrt($x)'|bc
(standard_in) 1: illegal character: $
root@kali:~/Desktop/bash_programs#
```

SOLUTION - *ESCAPE DOUBLE QUOTES FOR ECHO and use double quotes for both*



```
root@kali:~/Desktop/bash_programs# x=10
root@kali:~/Desktop/bash_programs# echo "print\"evaluating  \";sqrt($x)"|bc -l
evaluating  3.16227766016837933199
root@kali:~/Desktop/bash_programs#
```

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

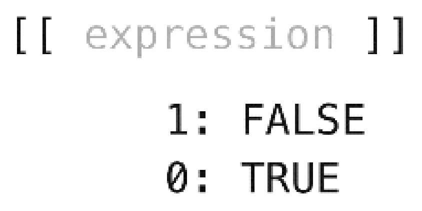
```
$ echo "for(i=1;i<=10;i++) {i;}" | bc
$ echo "i=1; while(i<=10) { i; i+=1}" | bc
```

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

Comparing Value

Tuesday, December 13, 2016 12:40 AM

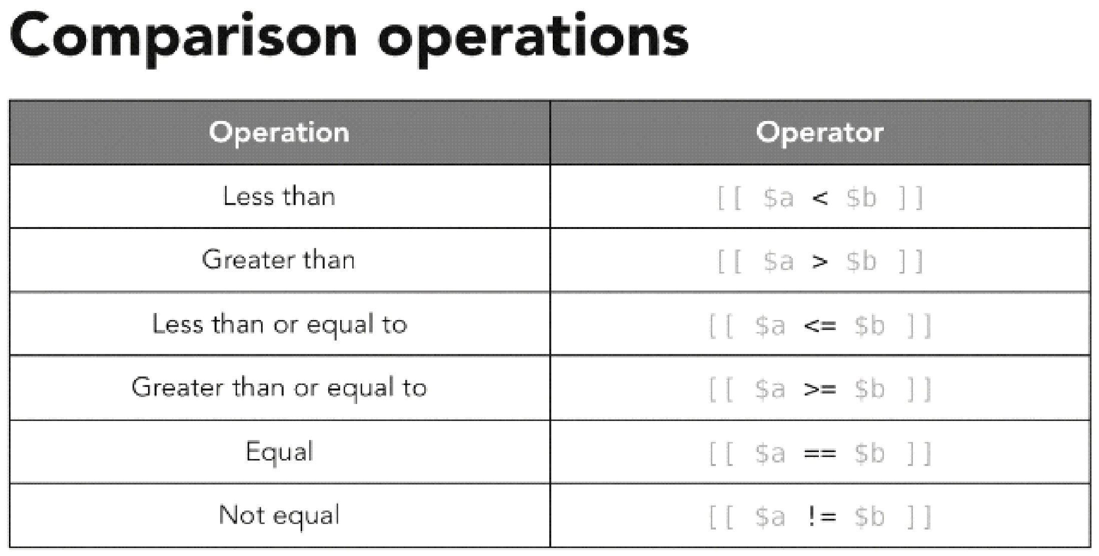
- For comparing we use double brackets [[ ]](for math we have use double parenthesis.
- Sometimes we will be able to use [](simple test) , but we will always use extended text ([[ ]]), because it contains some extending features like regular expression matching etc.
- By default all values are strings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

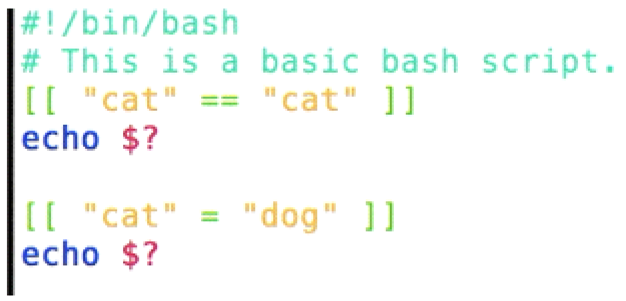
- In bash returns are opposite to C, 1 for false and 0 for true.



Comparison operations

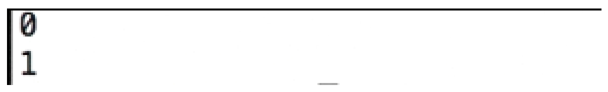
Operation	Operator
Less than	`[[ $a < $b ]]`
Greater than	`[[ $a > $b ]]`
Less than or equal to	`[[ $a <= $b ]]`
Greater than or equal to	`[[ $a >= $b ]]`
Equal	`[[ $a == $b ]]`
Not equal	`[[ $a != $b ]]`

- We know, $? Represents result of previous command



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

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

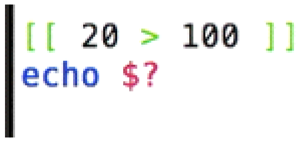
OUTPUT-first true second false



```
0
1
```

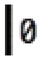
- **Note- any = or == can be used with strings to compare ,because by [[ ]] we able to know that we are going to compare.**

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



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

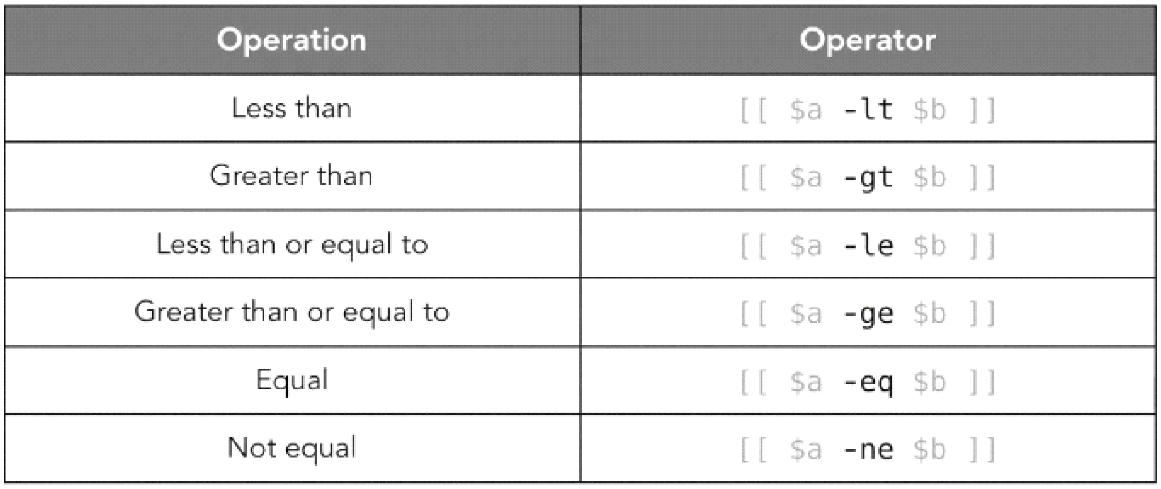
OUTPUT-



```
0
```

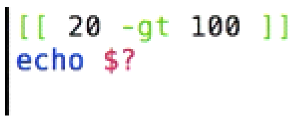
- Output is 0(true) because 20 is lexicographically greater than 100 and bash by default take every value as a string value.
  so for comparing as numbers we have to use special comparison operators.

Comparison operators for numbers-



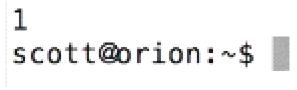
Operation	Operator
Less than	`[[ $a -lt $b ]]`
Greater than	`[[ $a -gt $b ]]`
Less than or equal to	`[[ $a -le $b ]]`
Greater than or equal to	`[[ $a -ge $b ]]`
Equal	`[[ $a -eq $b ]]`
Not equal	`[[ $a -ne $b ]]`

- 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 $?
```

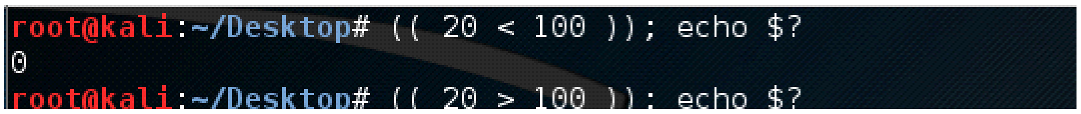
OUTPUT-



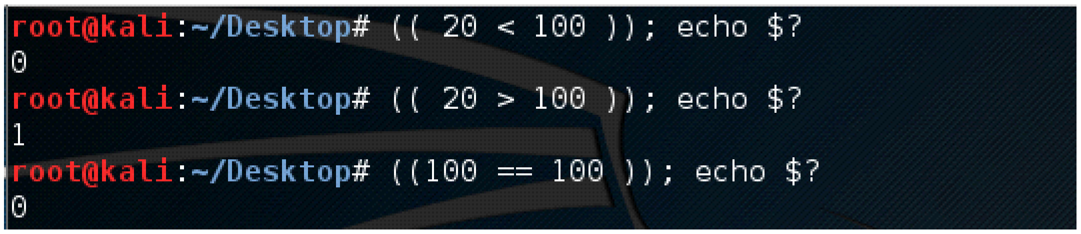
```
1
scott@orion:~$
```

Alternate way for comparing numbers

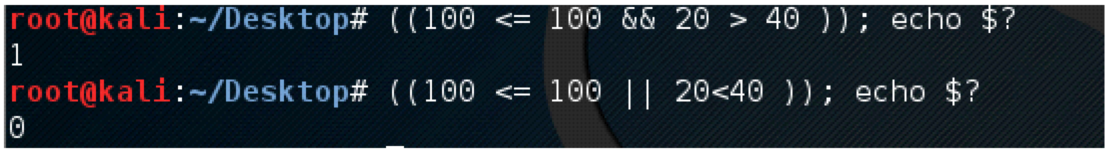
- We can also compare numbers using (( )) operators, in them we can use simple operators like(<,>,=) etc.
- Remember 0 is true and 1 is false;



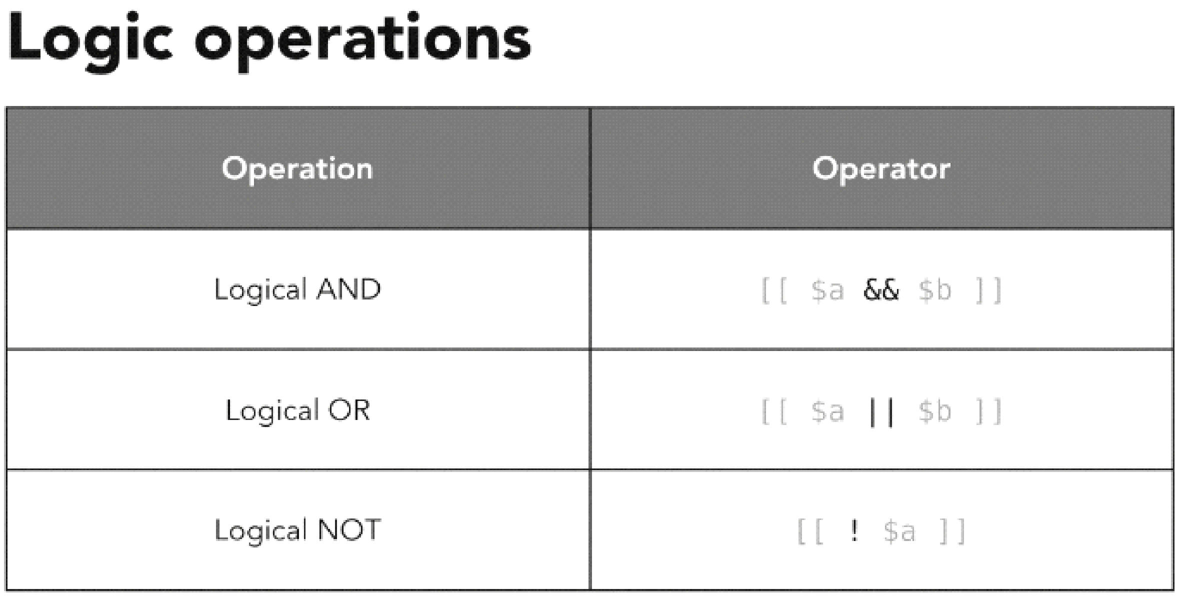
```
root@kali:~/Desktop# (( 20 < 100 )); echo $?
0
root@kali:~/Desktop# (( 20 > 100 )); echo $?
```



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



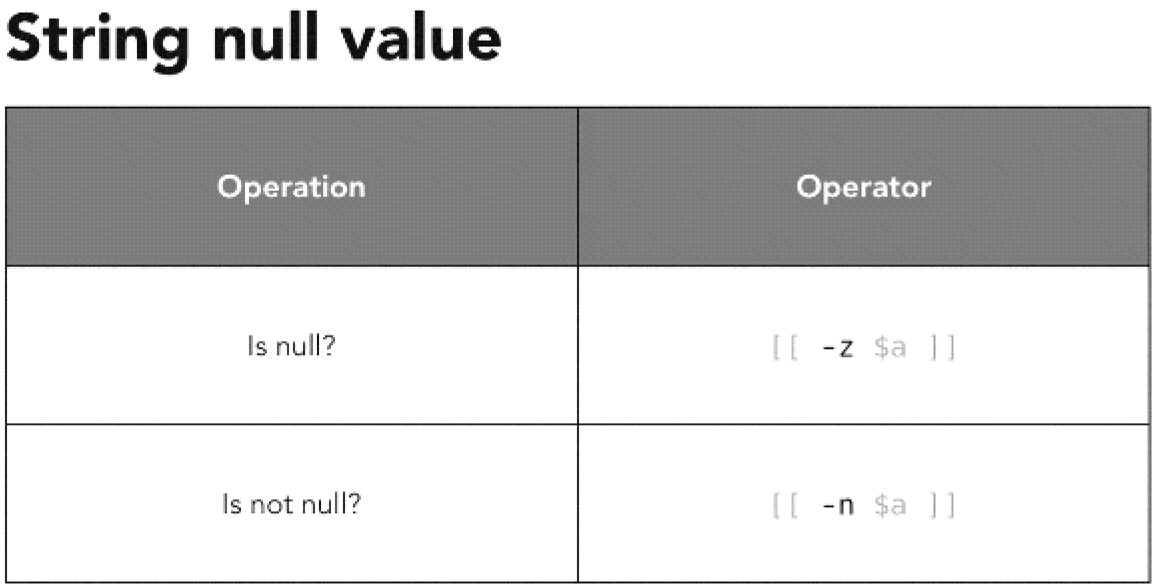
```
root@kali:~/Desktop# ((100 <= 100 && 20 > 40 )); echo $?
1
root@kali:~/Desktop# ((100 <= 100 || 20<40 )); echo $?
0
```



Logic operations

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

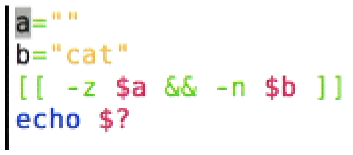
- Null String (empty string) value can be checked by following operations.



String null value

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

(to learn 'isnull' is same as 'isZero' so we have -z)
Example-

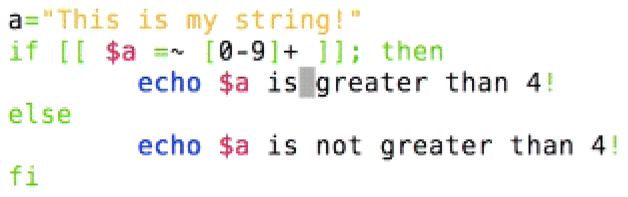


```
a=""
b="cat"
[[ -z $a && -n $b ]]
echo $?
```

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
```

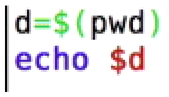
Rule of thumb -

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

Command Substitution

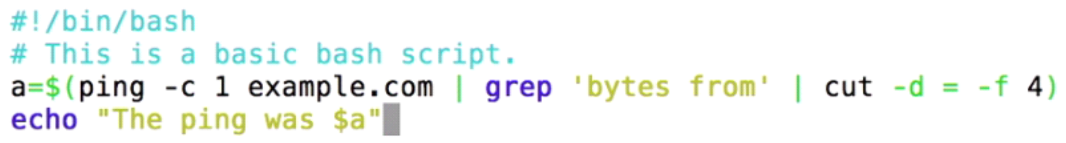
Monday, December 12, 2016 6:51 PM

- 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



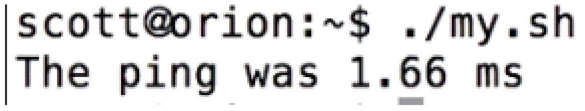
```
d=$(pwd)
echo $d
```

- 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
```

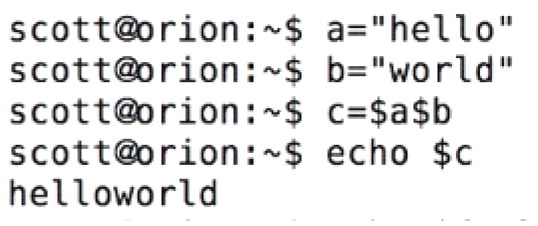
Working with strings

Tuesday, December 13, 2016 1:03 AM

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

Concatenation

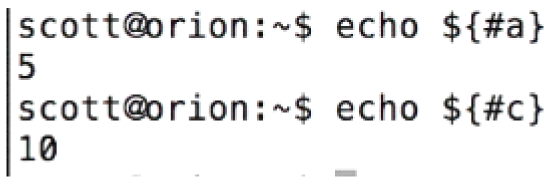
- Concatenation of strings can be done by putting string together **without any space** between them.



```
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



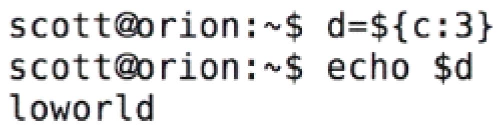
```
scott@orion:~$ echo ${#a}
5
scott@orion:~$ echo ${#c}
10
```

substring

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

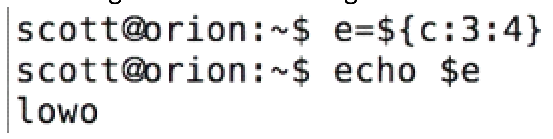
  syntax- ${str:start:no_of_char}

- Following will extract substring staring from 3 and to end



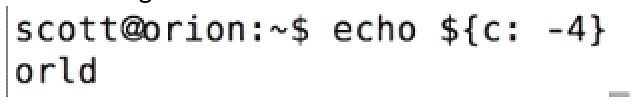
```
scott@orion:~$ d=${c:3}
scott@orion:~$ echo $d
loworld
```

- Following will extract starting from 3 and 4 characters



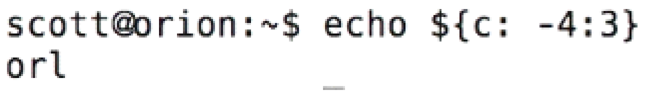
```
scott@orion:~$ e=${c:3:4}
scott@orion:~$ echo $e
lowo
```

- 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
```

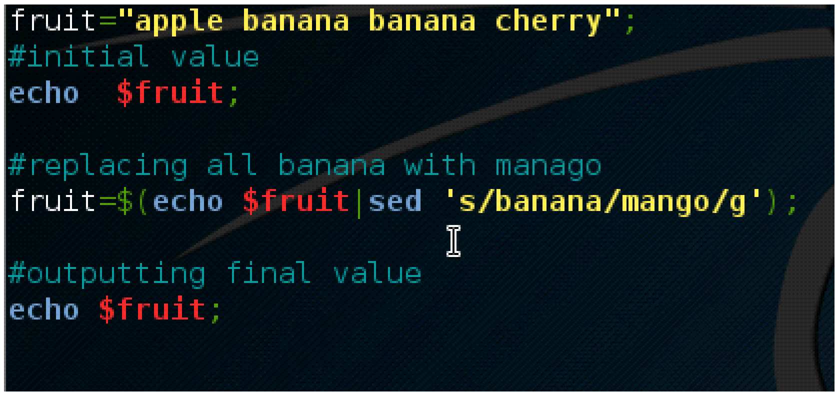
- Following will extract start 3 characters from last 4



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

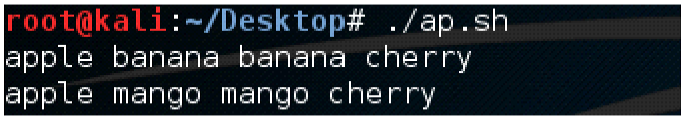
replace a part of string

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



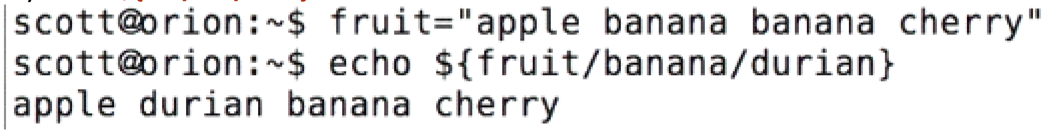
```
fruit="apple banana banana cherry";
#initial value
echo  $fruit;

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

#outputting final value
echo $fruit;
```

OUTPUT-



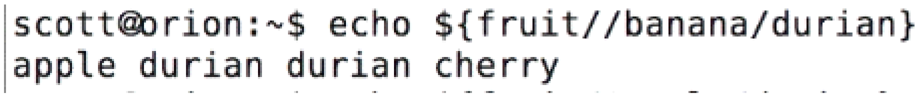
```
root@kali:~/Desktop# ./ap.sh
apple banana banana cherry
apple mango mango cherry
```

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



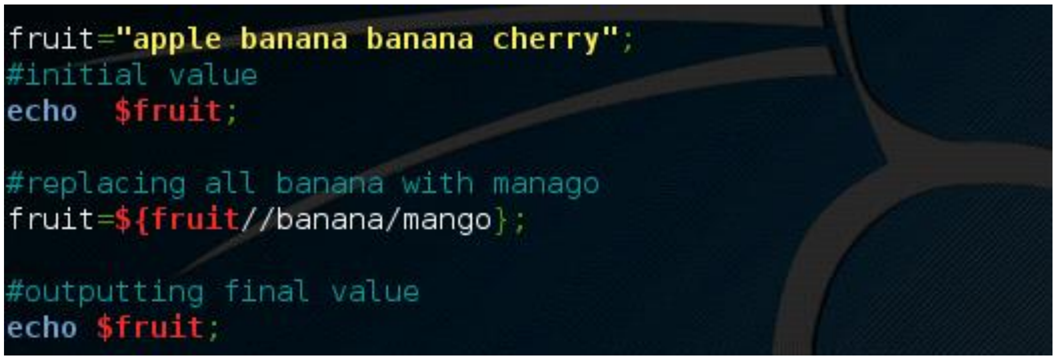
```
scott@orion:~$ fruit="apple banana banana cherry"
scott@orion:~$ echo ${fruit/banana/durian}
apple durian banana cherry
```

- 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)



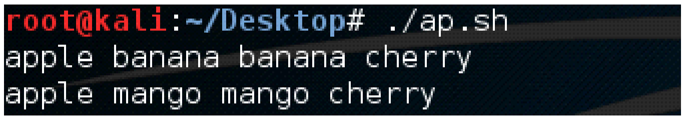
```
scott@orion:~$ echo ${fruit//banana/durian}
apple durian durian cherry
```

EXAMPLE-



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

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

#outputting final value
echo $fruit;
```

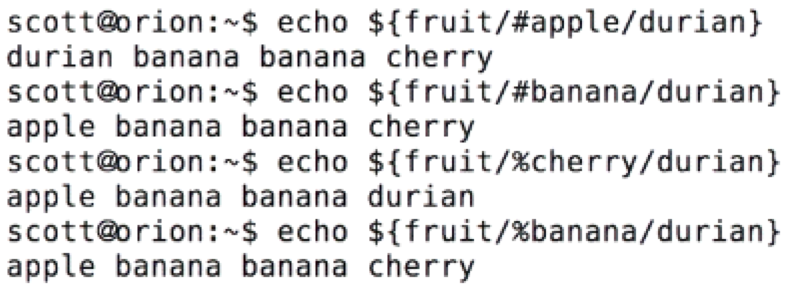
OUTPUT-



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

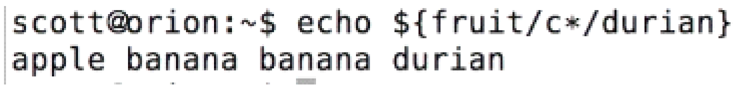
Note - for modifying the variable itself we have assign it back to it.

- 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
```

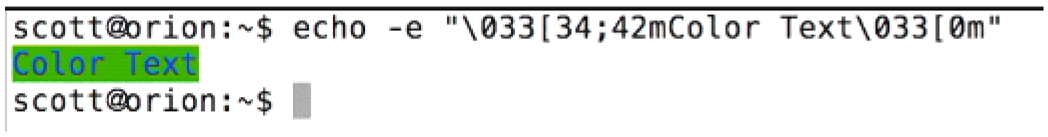
Coloring and Styling texts

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

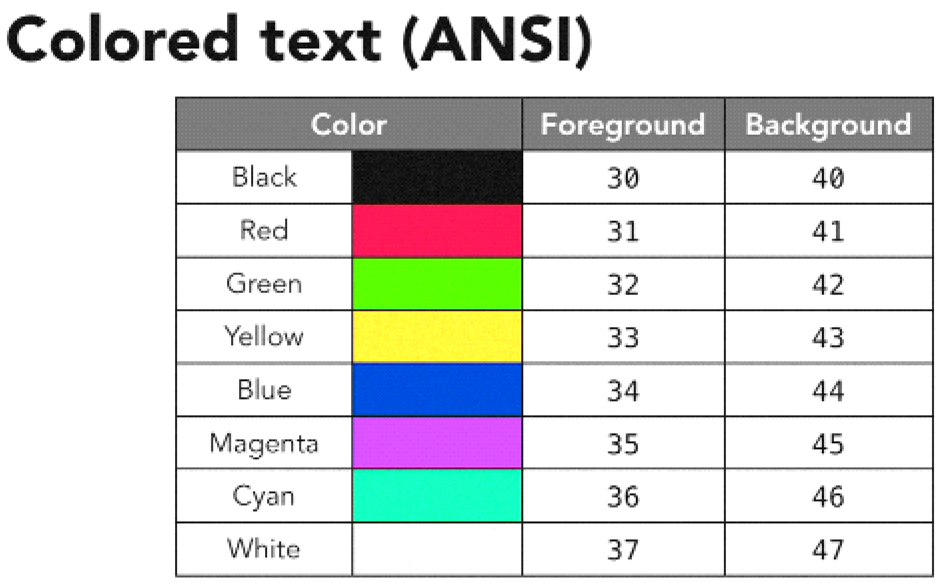
- For using ascii codes we have to use use -e opetion with echo ,which tells the interpreter that we are going to used ansii escape codes.
- Example - to print something in green color



```
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

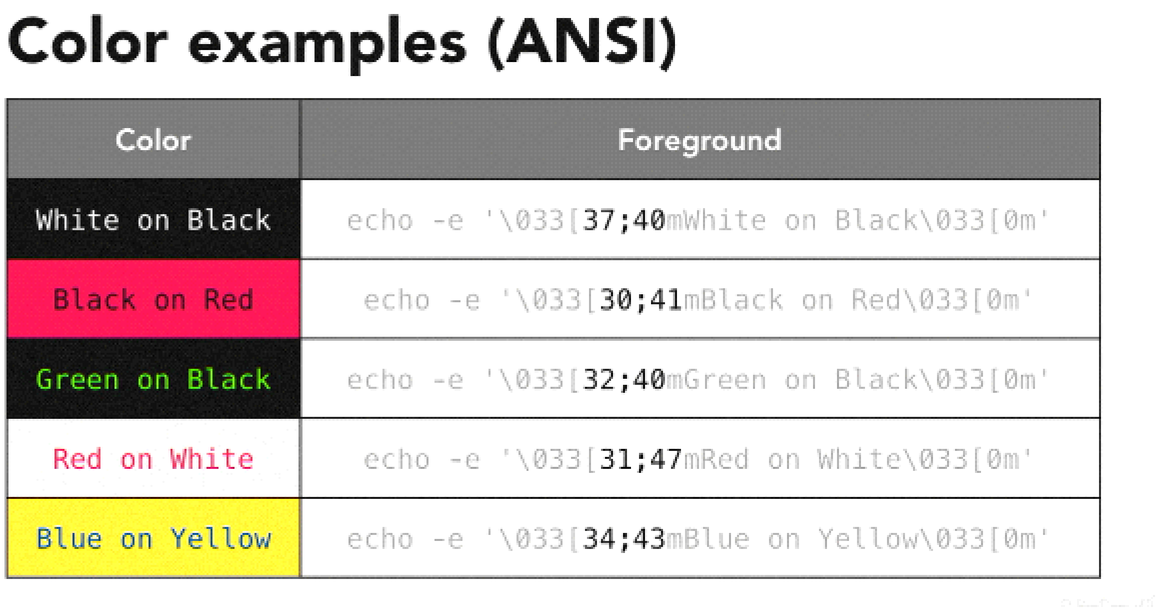
  syntax- **\033[<style_code>;<forground_code>;<background_code>m**
- If not given the default will be used.
  Note - we have **\033[0m** at last, in that 0 specifies no-style.



Colored text (ANSI)

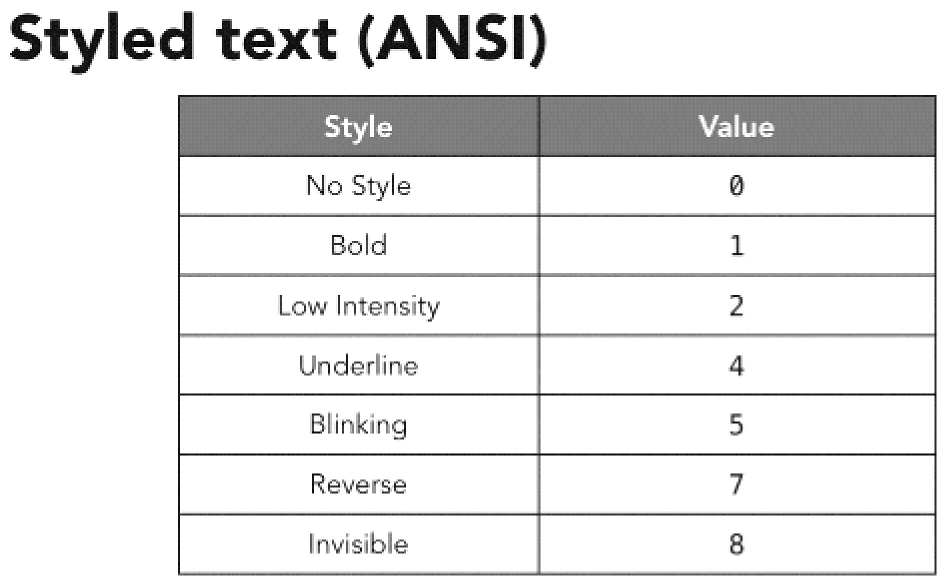
Color		Foreground	Background
Black	⬛	30	40
Red	🟥	31	41
Green	🟩	32	42
Yellow	🟨	33	43
Blue	🟦	34	44
Magenta	🟪	35	45
Cyan	🟦	36	46
White	⬜	37	47

- 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

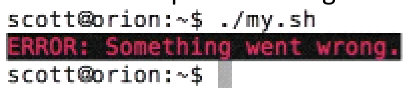
Example- to print error message ERROR should be blinking itself?

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

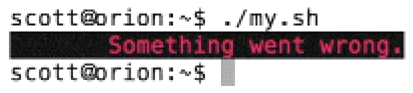
- Here we open styling and then write 'ERROR' then closed that as other part has other type of styling, then for other part we used styling.
- In the first part we used style 5 which is blinking but we do not need that for second part.

OUTPUT-

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

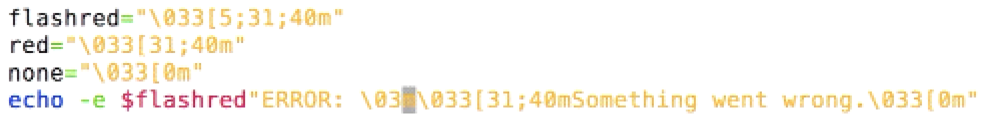


```
scott@orion:~$ ./my.sh
ERROR: Something went wrong.
scott@orion:~$
```



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

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

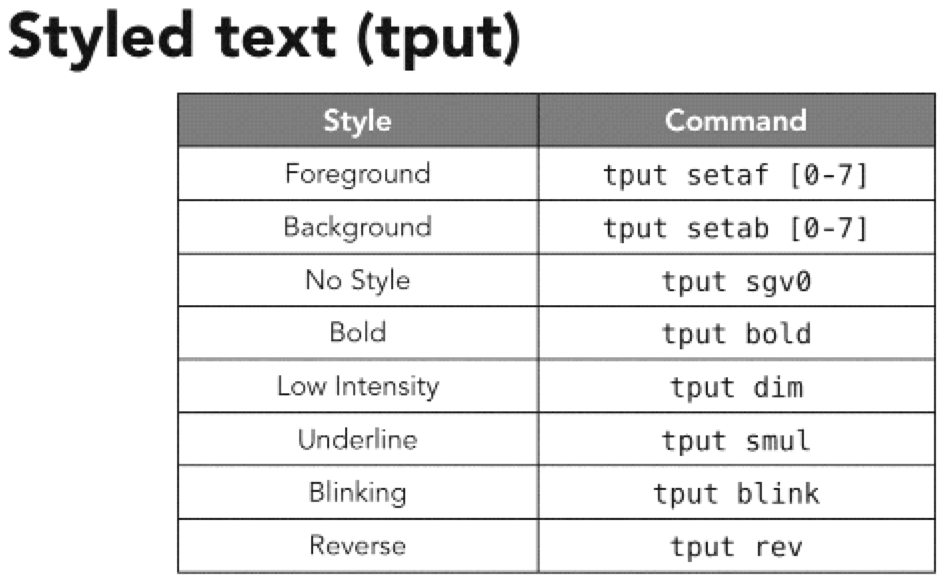


```
flashred="\033[5;31;40m"
red="\033[31;40m"
none="\033[0m"
echo -e $flashred"ERROR: \033\033[31;40mSomething went wrong.\033[0m"
```

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

METHOD 2- using tput utility

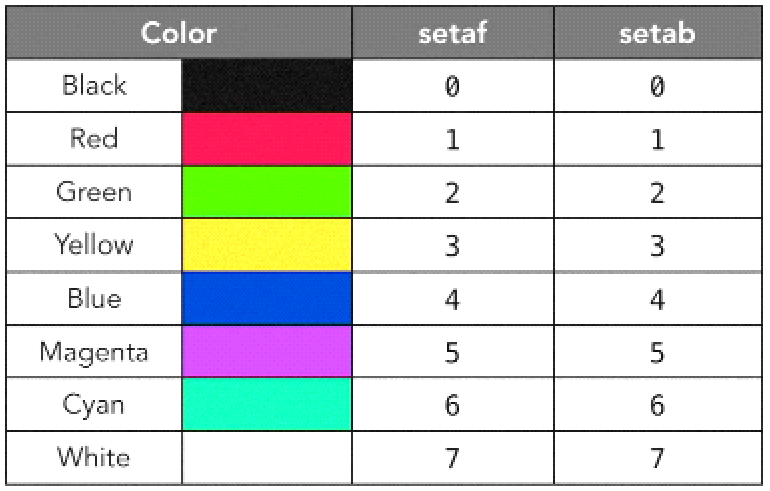
- Tput can used to create styled text and it is more easy to use and roburst than ansii code.
- As 'tput' is a command we have to use command substitution.



Styled text (tput)

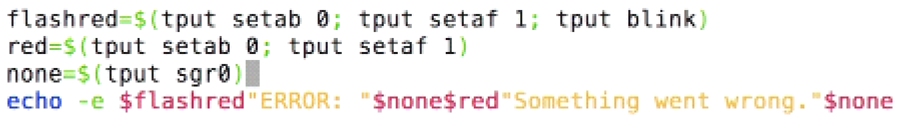
Style	Command
Foreground	`tput setaf [0-7]`
Background	`tput setab [0-7]`
No Style	`tput sgv0`
Bold	`tput bold`
Low Intensity	`tput dim`
Underline	`tput smul`
Blinking	`tput blink`
Reverse	`tput rev`

- 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.



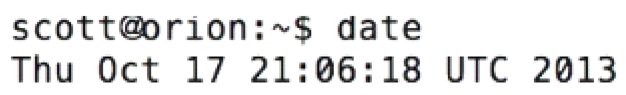
```
flashred=$(tput setab 0; tput setaf 1; tput blink)
red=$(tput setab 0; tput setaf 1)
none=$(tput sgr0)
echo -e $flashred"ERROR: "$none$red"Something went wrong."$none
```

For more see **$man terminfo**

Date and Cal command

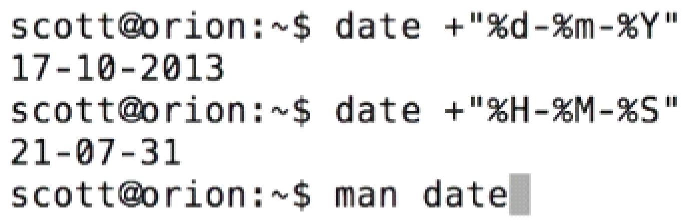
Tuesday, December 13, 2016   8:09 PM

- 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)

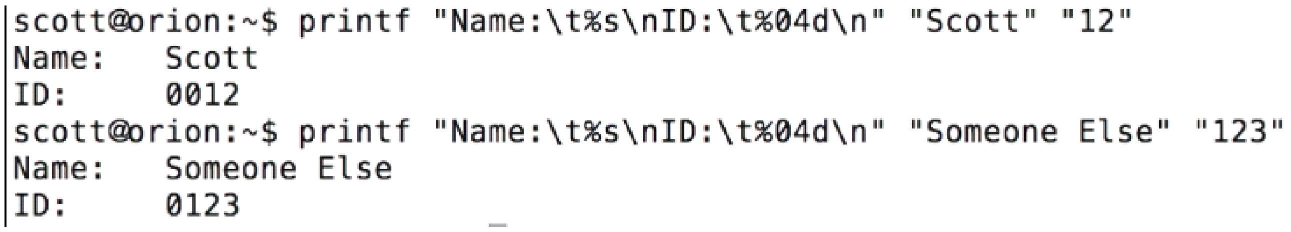
Printf command

Tuesday, December 13, 2016 7:41 PM

- Printf formates data same as C's printf

- Basic structure of command
  - Printf FORMAT [ARGUMENT]

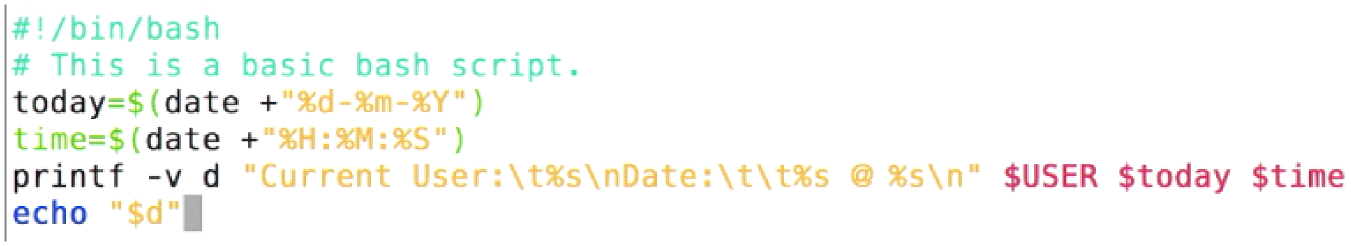
  ([] means may or may not be)



```
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
```

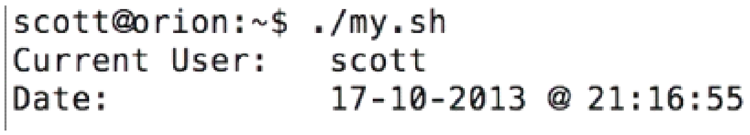
- If we wanted to assign output of printf to some variable we can use **-v var_name** before starting outputting, this will tell printf to not print anything but assign it to var_name variable.
  example - **printf -v var "this is output";**

- 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

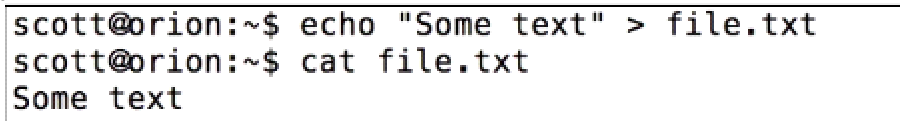
Tuesday, December 13, 2016 9:20 PM

- 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

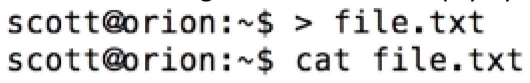
- '>' used for outputting to file(input from file's perspective)
- It creates file if not exist and overwrite it if it exits.

Example-



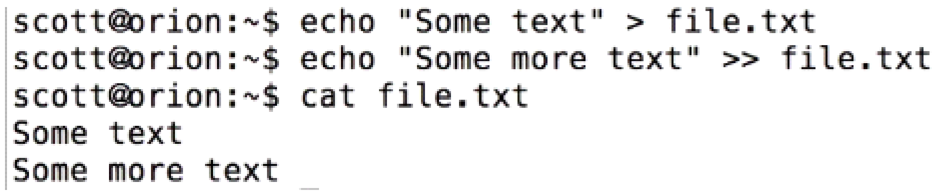
```
scott@orion:~$ echo "Some text" > file.txt
scott@orion:~$ cat file.txt
Some text
```

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



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

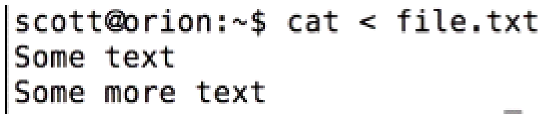
- We can append to a file (if exist) using **>>** symbol instead of **>.** (it will also create if not exist)



```
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")

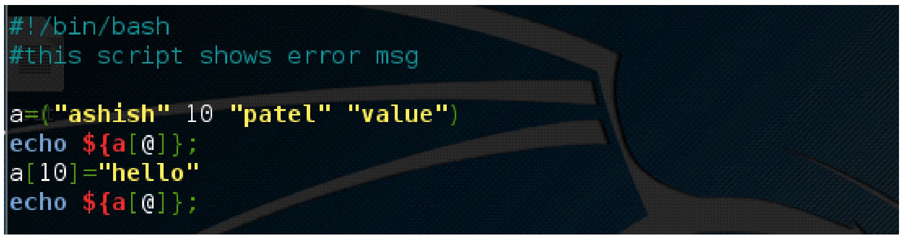
Displaying value

- We can use index as stated earlier.(We can use @ operator to access all elements of an array.(null element not shown)

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

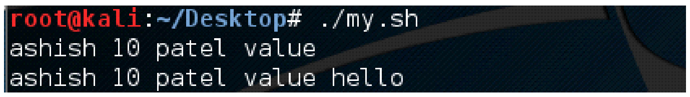
(only those element will come which are not null, so in case of sparse arrays{comes if we have added an element leaving some between empty} We not going to see those are in between and empty.)

example-



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

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

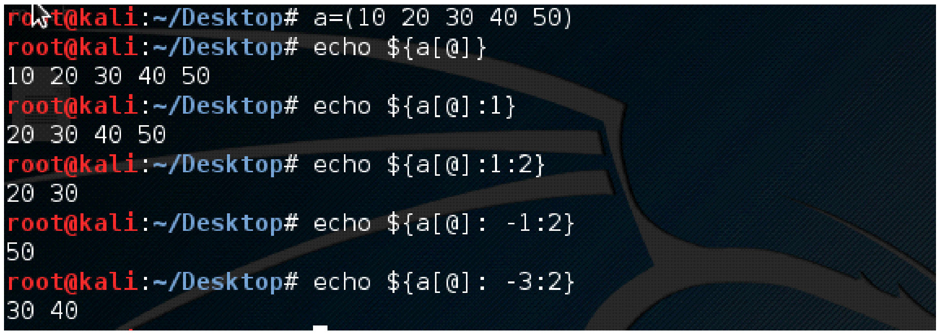
OUTPUT-



```
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-

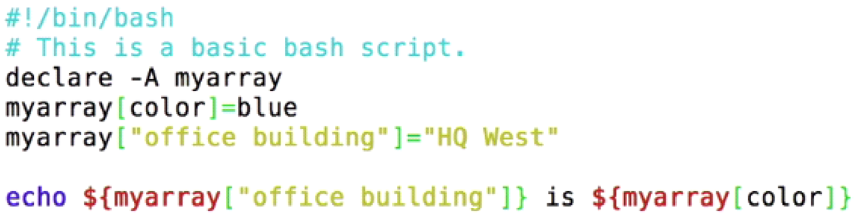


```
root@kali:~/Desktop# a=(10 20 30 40 50)
root@kali:~/Desktop# echo ${a[@]}
10 20 30 40 50
root@kali:~/Desktop# echo ${a[@]:1}
20 30 40 50
root@kali:~/Desktop# echo ${a[@]:1:2}
20 30
root@kali:~/Desktop# echo ${a[@]: -1:2}
50
root@kali:~/Desktop# echo ${a[@]: -3:2}
30 40
```

- 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

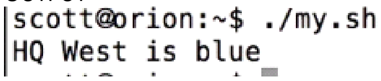
- available in **bash 4 and above**
- We have to used **declare** keyword while declaring to make associative array
  (recall **declare** also used when we want to specify type of variable)



```
#!/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 in 'declare'

Some Operations on arrays

- echo ${arr[@]} - output all values of array
- echo ${!arr[@]} - output all keys of arrays (may not work outside loop in ubuntu)
- echo ${#arr[@]} - number of elements in the array
- echo ${#arr[0]} - output length of first element
- echo ${#arr[i]}- output length of ith index element

NOTE- there may be different behaviours of $arr and ${#arr} in different distributions so use full version
for example - echo $arr;#output first element in kali but full array in ubuntu

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**

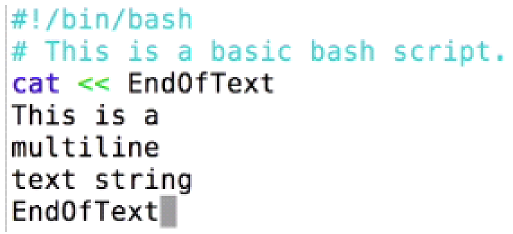
----

input

----

endstring

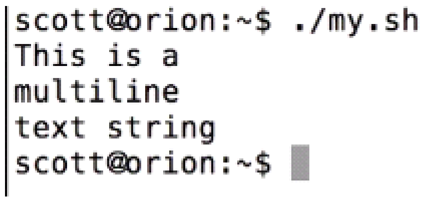
Example- this example feeds lot of input into command **cat.**



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

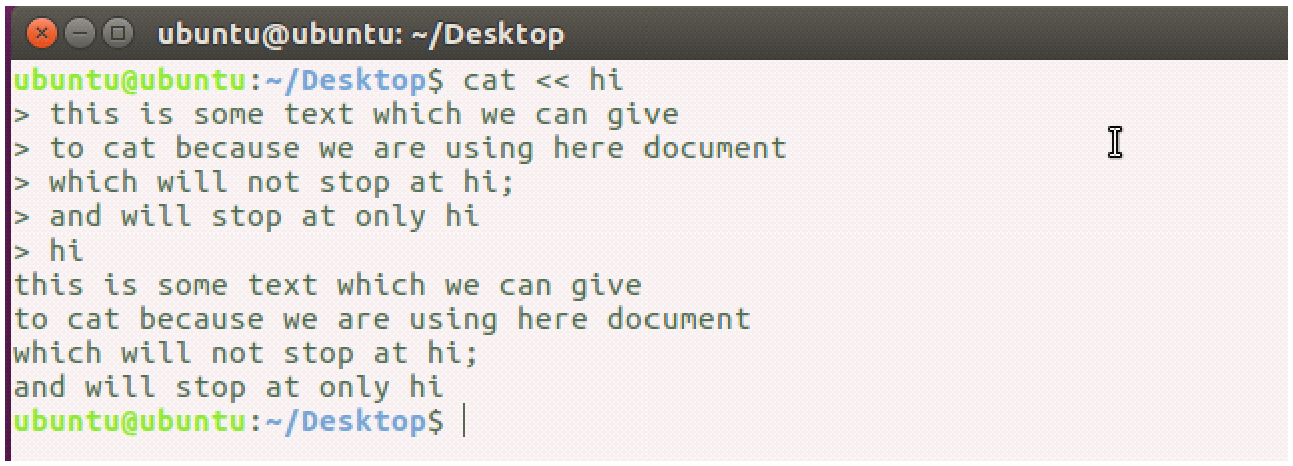
(here all thing between "EndOfText' is feeded to cat command which will output it.)

Output-

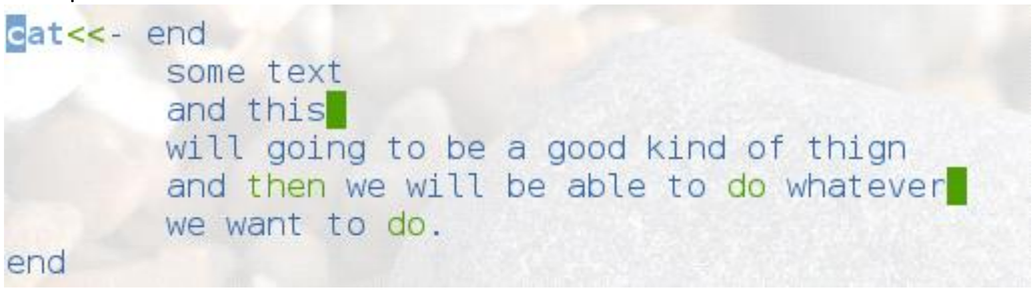


```
scott@orion:~$ ./my.sh
This is a
multiline
text string
scott@orion:~$
```

Example- in ubuntu



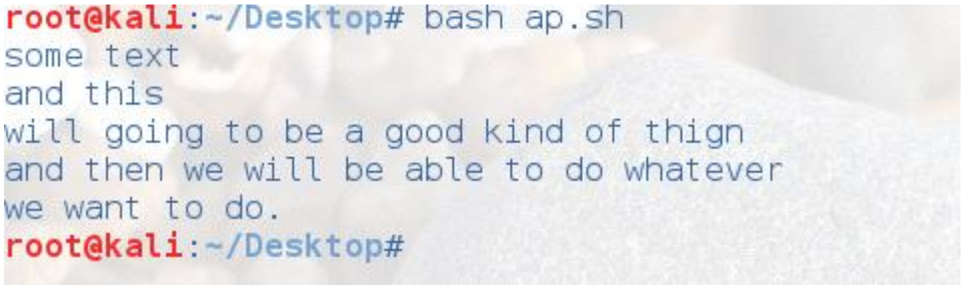
```
ubuntu@ubuntu:~/Desktop$ cat << hi
> this is some text which we can give
> to cat because we are using here document
> which will not stop at hi;
> and will stop at only hi
> hi
this is some text which we can give
to cat because we are using here document
which will not stop at hi;
and will stop at only hi
ubuntu@ubuntu:~/Desktop$
```

- 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-
- 

```
cat<<- end
        some text
        and this
        will going to be a good kind of thign
        and then we will be able to do whatever
        we want to do.
end
```

(dash - should be placed just after <<, so no space In between)

OUTPUT-



```
root@kali:~/Desktop# bash ap.sh
some text
and this
will going to be a good kind of thign
and then we will be able to do whatever
we want to do.
root@kali:~/Desktop#
```

- 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
```

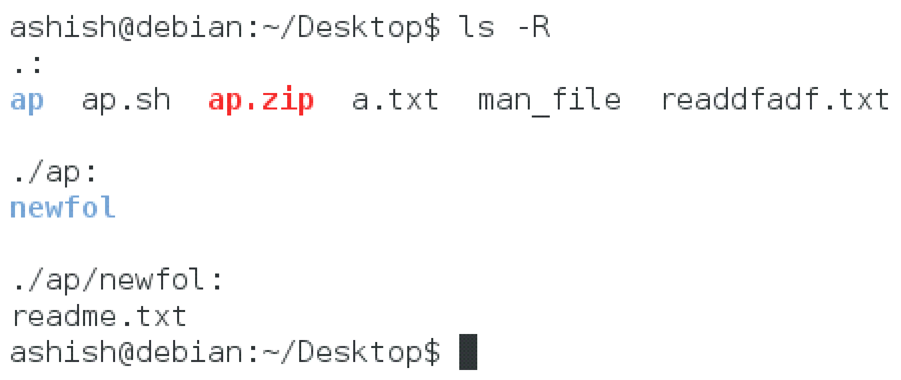
v

File showing

Friday, January 20, 2017 12:15 AM

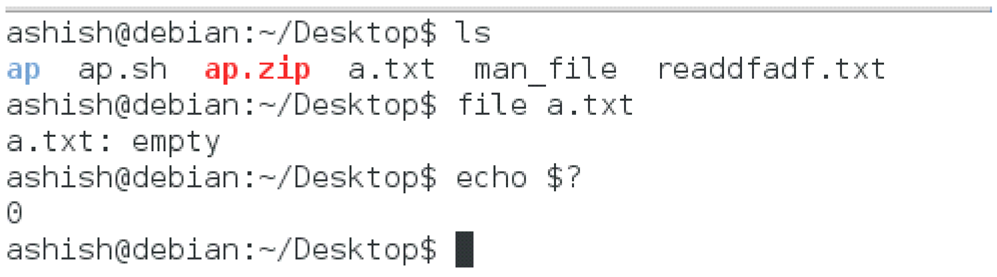
Showing only inside a folder(recursively)-

- We can know all files inside a directory using
  $ls -R (we can also give path to other dirs)



```
ashish@debian:~/Desktop$ ls -R
.:
ap  ap.sh  ap.zip  a.txt  man_file  readdfadf.txt

./ap:
newfol

./ap/newfol:
readme.txt
ashish@debian:~/Desktop$
```

- So directory structure is as follows, '.' has follows
  - Ap->newfol->readme.txt
  - Ap.sh
  - a.txt
  - Man_file
  - Readdfadf.txt



```
ashish@debian:~/Desktop$ ls
ap  ap.sh  ap.zip  a.txt  man_file  readdfadf.txt
ashish@debian:~/Desktop$ file a.txt
a.txt: empty
ashish@debian:~/Desktop$ echo $?
0
ashish@debian:~/Desktop$
```

Find command

Wednesday, February 15, 2017 9:55 PM

http://www.binarytides.com/linux-find-command-examples/

1. List all files in current and sub directories

This command lists out all the files in the current directory as well as the subdirectories in the current directory.

```
$ find
.
./abc.txt
./subdir
./subdir/how.php
./cool.php
```

The command is same as the following

```
$ find .
$ find . -print
```

2. Search specific directory or path

The following command will look for files in the test directory in the current directory. Lists out all files by default.

```
$ find ./test
./test
./test/abc.txt
./test/subdir
./test/subdir/how.php
./test/cool.php
```

The following command searches for files by their name.

```
$ find ./test -name "abc.txt"
./test/abc.txt
```

We can also use wildcards

```
$ find ./test -name "*.php"
./test/subdir/how.php
./test/cool.php
```

Note that all sub directories are searched recursively. So this is a very powerful way to find all files of a given extension.

Trying to search the "/" directory which is the root, would search the entire file system including mounted devices and network storage devices. So be careful. Of course you can press Ctrl + c anytime to stop the command.

Ignore the case

It is often useful to ignore the case when searching for file names. To ignore the case, just use the "iname" option instead of the "name" option.

3. Limit depth of directory traversal

The find command by default travels down the entire directory tree recursively, which is time and resource consuming. However the depth of directory travesal can be specified. For example we don't want to go more than 2 or 3 levels down in the sub directories. This is done using the maxdepth option.

```
$ find ./test -maxdepth 2 -name "*.php"
./test/subdir/how.php
./test/cool.php

$ find ./test -maxdepth 1 -name *.php
./test/cool.php
```

The second example uses maxdepth of 1, which means it will not go lower than 1 level deep, either only in the current directory.

4. Invert match

It is also possible to search for files that do no match a given name or pattern. This is helpful when we know which files to exclude from the search.

```
$ find ./test -not -name "*.php"
./test
./test/abc.txt
./test/subdir
```

So in the above example we found all files that do not have the extension of php, either non-php files. The find command also supports the exclamation mark inplace of not.

```
find ./test ! -name "*.php"
```

5. Combine multiple search criterias

It is possible to use multiple criterias when specifying name and inverting. For example

```
$ find ./test -name 'abc*' ! -name '*.php'
./test/abc.txt
./test/abc
```

The above find command looks for files that begin with abc in their names and do not have a php extension. This is an example of how powerful search expressions can be build with the find command.

```
find -not -name "query_to_avoid"
```

- We can also use **-not** in place of **!** Operator.

OR operator

When using multiple name criterias, the find command would combine them with AND operator, which means that only those files which satisfy all criterias will be matched. However if we need to perform an OR based matching then the find command has the "o" switch.

```
$ find -name '*.php' -o -name '*.txt'
./abc.txt
./subdir/how.php
./abc.php
./cool.php
```

The above command search for files ending in either the php extension or the txt extension.

6. Search only files or only directories

Sometimes we want to find only files or only directories with a given name. Find can do this easily as well.

```
$ find ./test -name abc*
./test/abc.txt
./test/abc

Only files

$ find ./test -type f -name "abc*"
./test/abc.txt

Only directories

$ find ./test -type d -name "abc*"
./test/abc
```

Quite useful and handy!

7. Search multiple directories together

So lets say you want to search inside 2 separate directories. Again, the command is very simple

```
$ find ./test ./dir2 -type f -name "abc*"
./test/abc.txt
./dir2/abcdefg.txt
```

Check, that it listed files from 2 separate directories.

- We can see if we want to search specific filetype (file or directory) we can give using **-type** command.
  - File - "f"
  - Directory- "d"

8. Find hidden files

Hidden files on linux begin with a period. So its easy to mention that in the name criteria and list all hidden files.

```
$ find ~ -type f -name ".*"
```

Paste command

Wednesday, February 15, 2017 10:28 PM

About paste

The **paste** command displays the corresponding lines of multiple files side-by-side.

Description

paste writes lines consisting of the sequentially corresponding lines from each *FILE*, separated by tabs, to the standard output. With no *FILE*, or when *FILE* is a dash ("-"), **paste** reads from standard input.

paste syntax

```
paste [OPTION]... [FILE]...
```

Options

-d, **--delimiters**=*LIST*	reuse characters from *LIST* instead of tabs.
-s, **--serial**	paste one file at a time instead of in parallel.
--help	Display a help message, and exit.
--version	Display version information, and exit.

paste examples

```
paste file1.txt file2.txt
```

This command would display the contents of **file1.txt** and **file2.txt**, side-by-side, with the corresponding lines of each file separated by a tab.

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

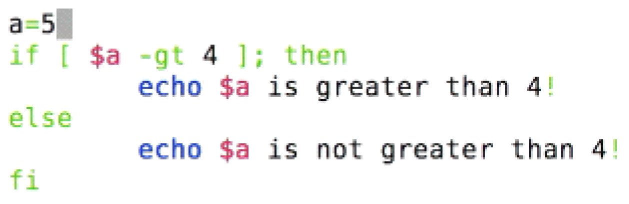
- We can use if-elif-else to have multiple cases

```
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

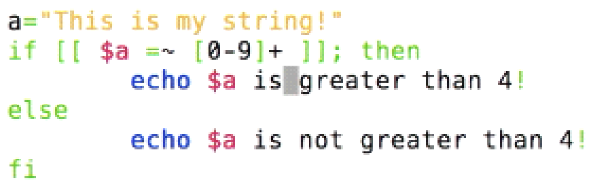


```
a=5
if [ $a -gt 4 ]; then
        echo $a is greater than 4!
else
        echo $a is not greater than 4!
fi
```

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

- NOTE- **there should be a space between if also, no restriction with (( )) comparision for integers.**

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

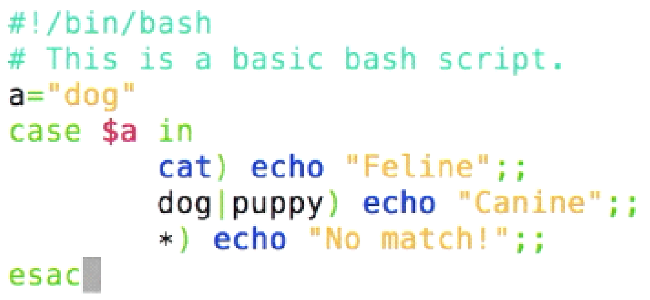


```
a="This is my string!"
if [[ $a =~ [0-9]+ ]]; then
        echo $a is greater than 4!
else
        echo $a is not greater than 4!
fi
```

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
```

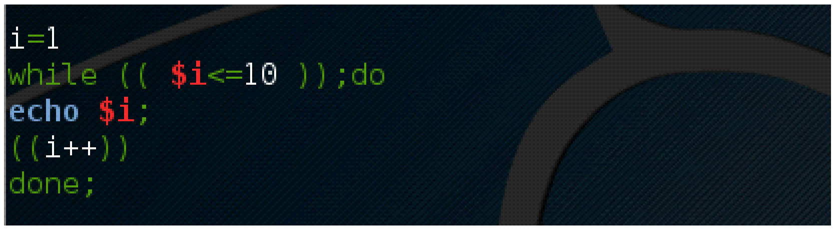
  - Starting done with 'in' and end is done with reverse case ie. 'esac'
  - We can show a option left to ')' parantheses
  - We can end result of some option with double semicolon(;;) as single is reserved for statement ending and if we have multiple statement we have to use it after each statement.
  - 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;

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

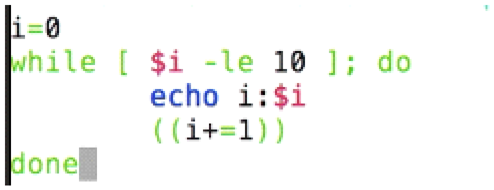
- Example- printing from 1 to 10 using integer comparison{using(( )) }



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

(we need not to use $ before 'I' in (($i<=10)) as it is inside (( )) , but here no error(so do not use $)

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



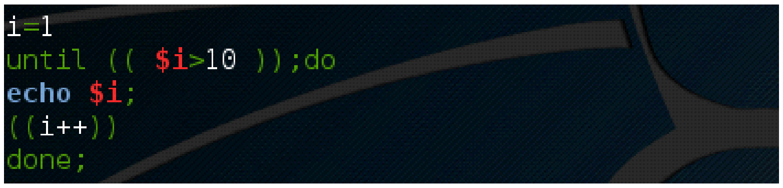
```
i=0
while [ $i -le 10 ]; do
        echo i:$i
        ((i+=1))
done
```

Until loop

Sunday, December 18, 2016 1:14 AM

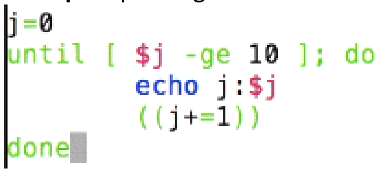
- Until loop is counterpart to while loop, so as soon as condition becomes true loop breaks(while works until condition is fal se)
- It applies condition in reverse order as it does something the **condition not become true.**

- **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
```

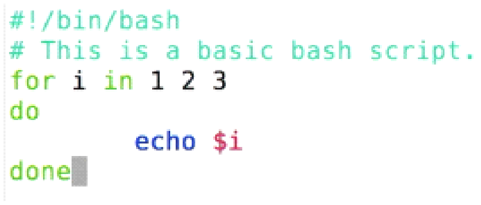
For loops

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
```

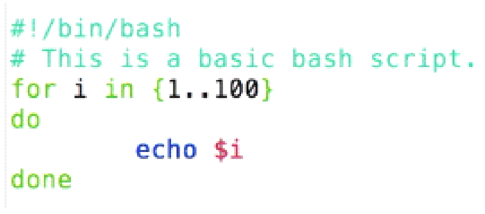
(value is assigned to 'I' one by one and printed)
OUTPUT-

```
1
2
3
```

Looping through brace expansion -

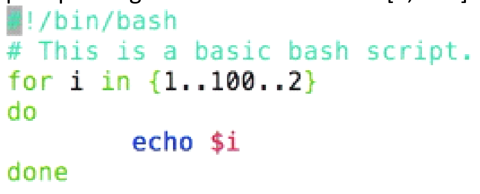
- We can use brance expansion here, so all value of brace expansion will come to I
  And we can do whatever we wanted to do with it.

example - print 1 to 100



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

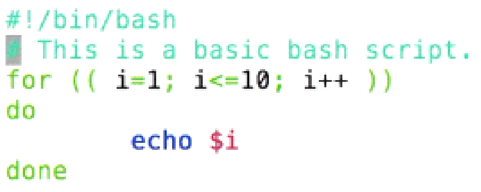
- We can also specify interval(as before)
  example - printing odd number between [1, 100]



```
#!/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



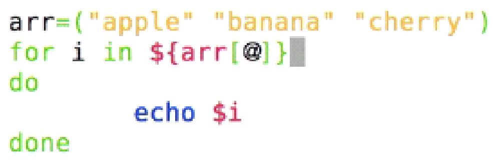
```
#!/bin/bash
# This is a basic bash script.
for (( i=1; i<=10; i++ ))
do
        echo $i
done
```

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.

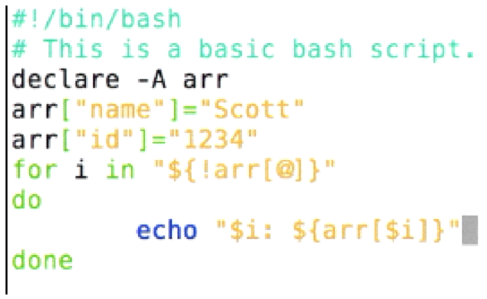
  Example- printing content of a array variable;



```
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.

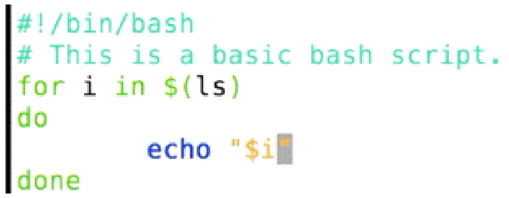


```
#!/bin/bash
# This is a basic bash script.
declare -A arr
arr["name"]="Scott"
arr["id"]="1234"
for i in "${!arr[@]}"
do
        echo "$i: ${arr[$i]}"
done
```

* herer ${!arr[@]} is not need to be inside string.
(we have to use '$i' as key to get the value).

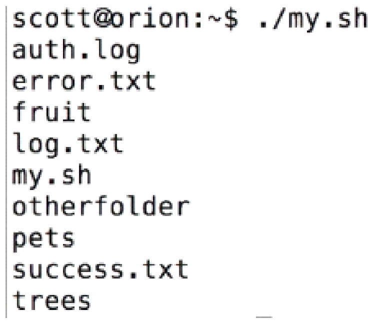
Looping through output of some command

- 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
```

functions

Sunday, December 18, 2016 2:18 AM

- Basic syntax of a function is as follows

- `Like "real" programming languages, Bash has functions, though in a somewhat limited implementation.`
- ` A function is a subroutine, a code block that implements a set of operations, a "black box" that performs a specified task. Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.`

Syntax -

- `We can use 'function' or () to tell interpretter that this is function.`

Using function keyword-

```
function function_name {
command...
}
```

- `Remember there should be s space between name and  '{'   ( or use { in another line)`

Using paranthesis ()-

```
function_name () {
command...
}
```

`This second form will cheer the hearts of C programmers (and is more portable).`

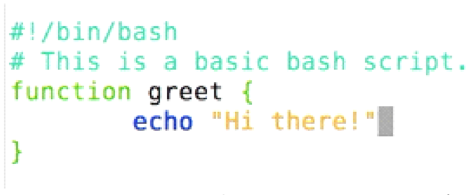
Using both together-

- It is no harm to use function keyword as well as () , which makes us syntax same as JS(but we do not take any argument here, as they represented by positional parameters)

```
function function_name () {
command...
}
```

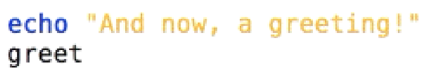
From <http://www.tldp.org/LDP/abs/html/functions.html>

Examples-1



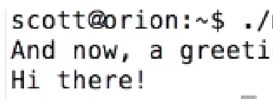
```
#!/bin/bash
# This is a basic bash script.
function greet {
        echo "Hi there!"
}
```

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



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

OUTPUT-



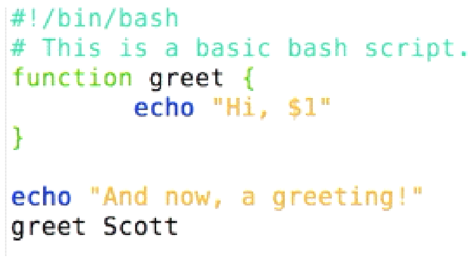
```
scott@orion:~$ ./my.sh
And now, a greeting!
Hi there!
```

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 (obviously that's why first way of defining fn is possible)
  $1 will store the first passed value and so on....($0 always have file name)

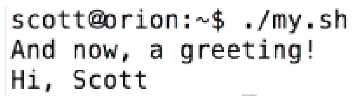
- Following example shows a function which do greeting using a argument to function

Examples-2



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

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

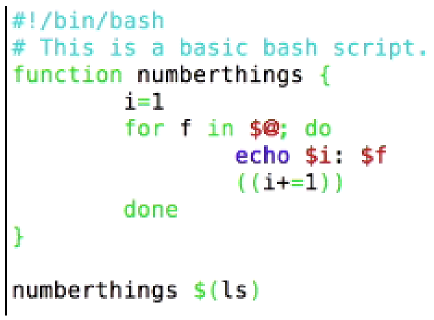
OUTPUT-



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

Passing command output to a function

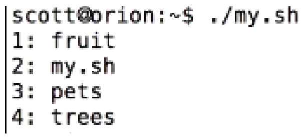
- We can pass command output to a function using command substitution.
  (because whenever we want to assign output of command to a variable we use command substitution)



```
#!/bin/bash
# This is a basic bash script.
function numberthings {
        i=1
        for f in $@; do
                echo $i: $f
                ((i+=1))
        done
}

numberthings $(ls)
```

(here $(ls) will give names of files in pwd , which can be passed to function.

OUTPUT-

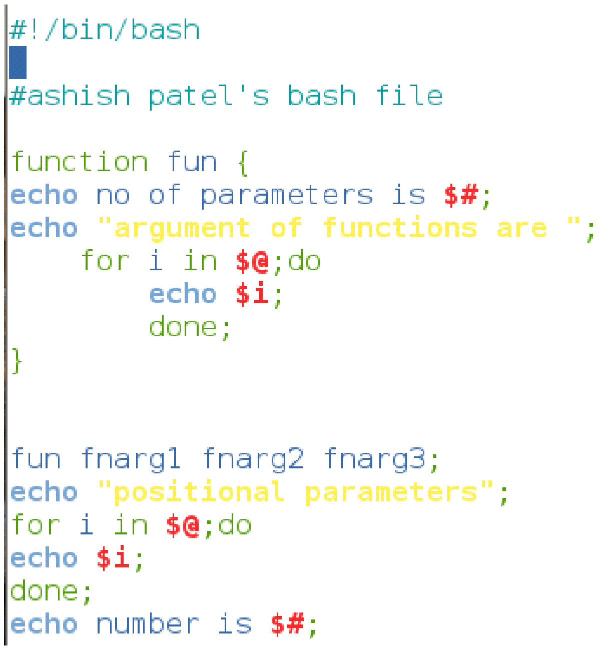


```
scott@orion:~$ ./my.sh
1: fruit
2: my.sh
3: pets
4: trees
```

Special variables

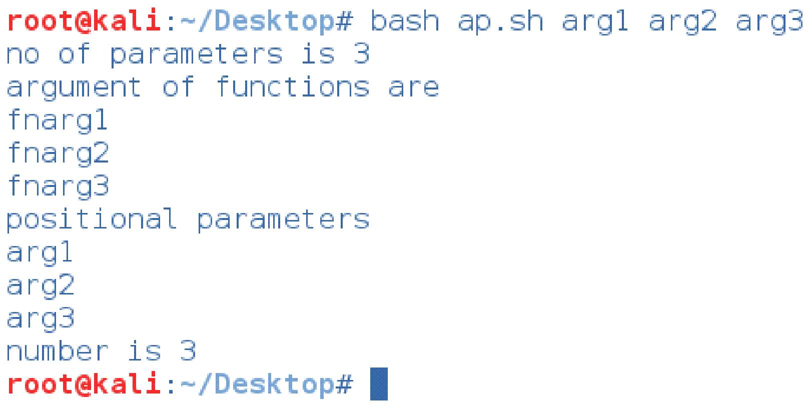
- **'*'** - all files/directories in pwd
- '$*' or '$@'- all positional parameters in following bash or argument in function
- '$#' - number of positional parameters in bash file or argument in function
- '$?' - represent output of last command

Example showing use of special variables inside fn and outside-



```
#!/bin/bash

#ashish patel's bash file

function fun {
echo no of parameters is $#;
echo "argument of functions are ";
    for i in $@;do
        echo $i;
        done;
}


fun fnarg1 fnarg2 fnarg3;
echo "positional parameters";
for i in $@;do
echo $i;
done;
echo number is $#;
```

OUTPUT-



```
root@kali:~/Desktop# bash ap.sh arg1 arg2 arg3
no of parameters is 3
argument of functions are
fnarg1
fnarg2
fnarg3
positional parameters
arg1
arg2
arg3
number is 3
root@kali:~/Desktop#
```

Returning value from a function

Wednesday, February 15, 2017 9:19 PM

The simplest way to return a value from a bash function is to just set a global variable to the result. Since all variables in bash are global by default this is easy:

```
function myfunc()
{
    myresult='some value'
}

myfunc
echo $myresult
```

The code above sets the global variable `myresult` to the function result. Reasonably simple, but as we all know, using global variables, particularly in large programs, can lead to difficult to find bugs.

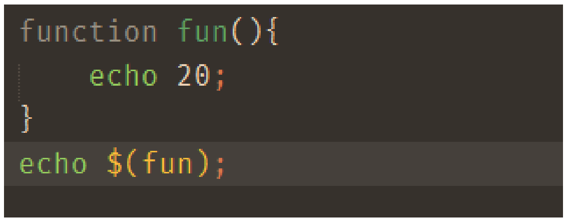
METHOD 1 - echo and take as command substitution

A better approach is to use local variables in your functions. The problem then becomes how do you get the result to the caller. One mechanism is to use command substitution:

```
function myfunc()
{
    local  myresult='some value'
    echo "$myresult"
}

result=$(myfunc)   # or result=`myfunc`
echo $result
```

Here the result is output to the stdout and the caller uses command substitution to capture the value in a variable. The variable can then be used as needed.

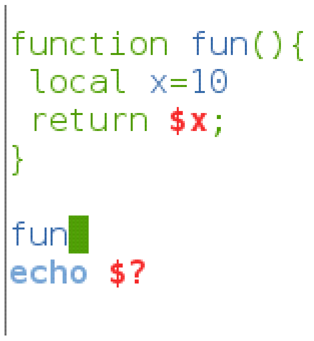
EXAMPLE-



```
function fun(){
    echo 20;
}
echo $(fun);
```

- Here we know, if we use command substitution then 'echo' will not print in output as it will assign value as the output.

METHOD 2 - return and get using previous output $? variable



```
function fun(){
 local x=10
 return $x;
}

fun
echo $?
```

Using command line argument

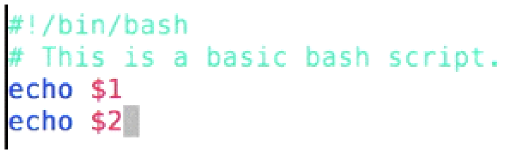
Monday, December 19, 2016 7:15 PM

- User can interact with script giving command line argument.
- By default space is delimiter, so if have space in our input, we can use double quote(" ") to make it one
- We can use special variable to access some kind of special functionality
- First argument will be $1 and so on..(these are called positional parameters.)

Special variables

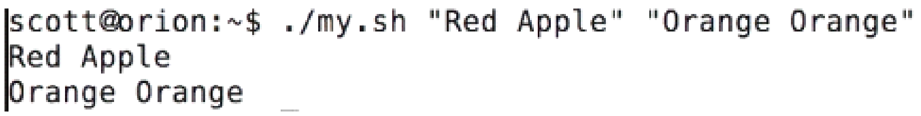
- **'*' -** all files/directories in pwd
- '$*' or '$@'- all positional parameters in following bash or argument in function
- '$#' - number of positional parameters in bash file or argument in function
- '$?' - represent output of last command

Example- using positional parameters



```
#!/bin/bash
# This is a basic bash script.
echo $1
echo $2
```

RUNNING-



```
scott@orion:~$ ./my.sh "Red Apple" "Orange Orange"
Red Apple
Orange Orange
```

Using flags(options)

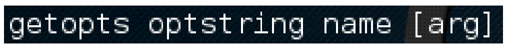
Monday, December 19, 2016 7:22 PM

- Although we can take input from use in form of command line argument, but how do we know the order Of input .
    - For example- if we want to take username and password from user, what will happen if he gives password First and username after it, so flags come in place
    - We may create two flag -u and -p
    - After -u we give username and after -p we give password, and we can give in any order

Example-
```
$  mycommand -u ashishpatel -p mypassword;
```

Implementing option in a command

- For implementing options, we have to use '**getopts**' command with 'while' loop, while loop will help to take input of all flags.
- Syntax **getopts** -



```
getopts optstring name [arg]
```

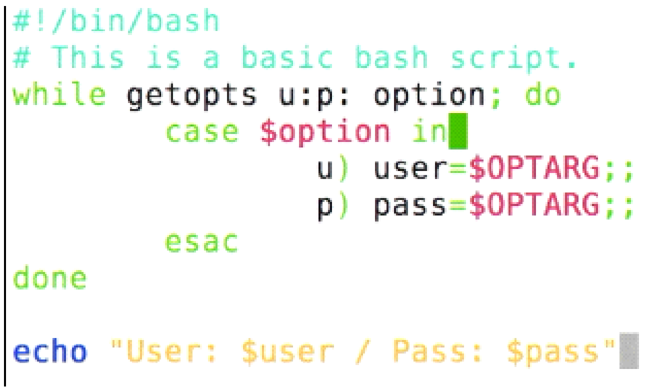
-
    - here 'optstring' is created with some syntaxes.
    - Option will be assigned to 'name' variable for comparing.
    - <<< ADD FUNCTIONALITY OF [arg] HERE >>>
- $OPTARG will contain the value passed after flag.

Rules for optstring

- It contains all options concatinated
- If there is a colon(:) after an option it is expected to have value
- If there is not colon(:) then it might be passed a value or not

Example - 1 for taking username and password in -u and -p.

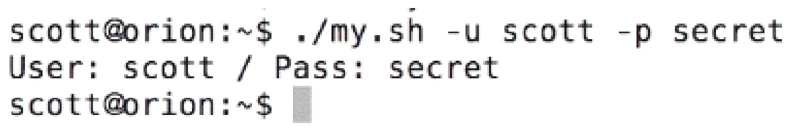
- Colon(:) is used after an option to tell that this is necessary to be passed otherwise our calculation fails.



```
#!/bin/bash
# This is a basic bash script.
while getopts u:p: option; do
        case $option in
                u) user=$OPTARG;;
                p) pass=$OPTARG;;
        esac
done

echo "User: $user / Pass: $pass"
```

- Here both -u and -p should be passed with value otherwise work we are doing with them , After 'switch case' will not be able to complete(however here we printing only)
- We can apply 'null' compare using ' **[[-z $user]]** ' to check if the value is passed or not to print some error accordingly.

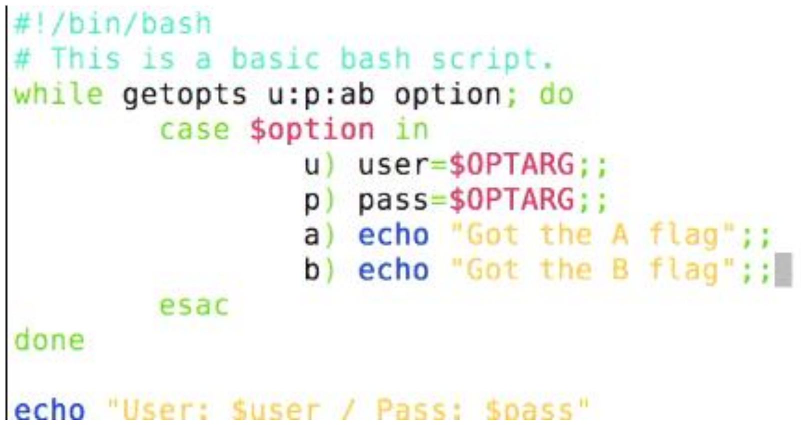
RUNNING-(remember after using flag order of options is not fixed)



```
scott@orion:~$ ./my.sh -u scott -p secret
User: scott / Pass: secret
scott@orion:~$
```

Example - 2 for taking -u ,-p and some other(which are not necessary).

- If we do not use colon(:) after option name in '**optstring'** then it becomes optional and it means we are not using in real calculation.



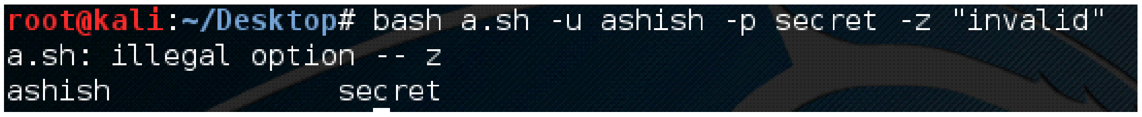
```
#!/bin/bash
# This is a basic bash script.
while getopts u:p:ab option; do
        case $option in
                u) user=$OPTARG;;
                p) pass=$OPTARG;;
                a) echo "Got the A flag";;
                b) echo "Got the B flag";;
        esac
done

echo "User: $user / Pass: $pass"
```

RUNNING-

- When we get 'a' or 'b' option in 'case' will be executed

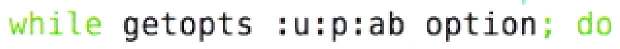
Checking for invalid flag symbols

- User may pass the flag which has not been implemented by us,
- by default, error will occur with ' illegal option'
- Running previous script with illegal flag 'z'



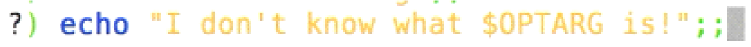
```
root@kali:~/Desktop# bash a.sh -u ashish -p secret -z "invalid"
a.sh: illegal option -- z
ashish              secret
```

- We can tell bash for taking illegal flag by adding starting colon(:) in 'optstring', by doing so we can tackle the invalid options by displaying our error message or doing appropriate action.



```
while getopts :u:p:ab option; do
```

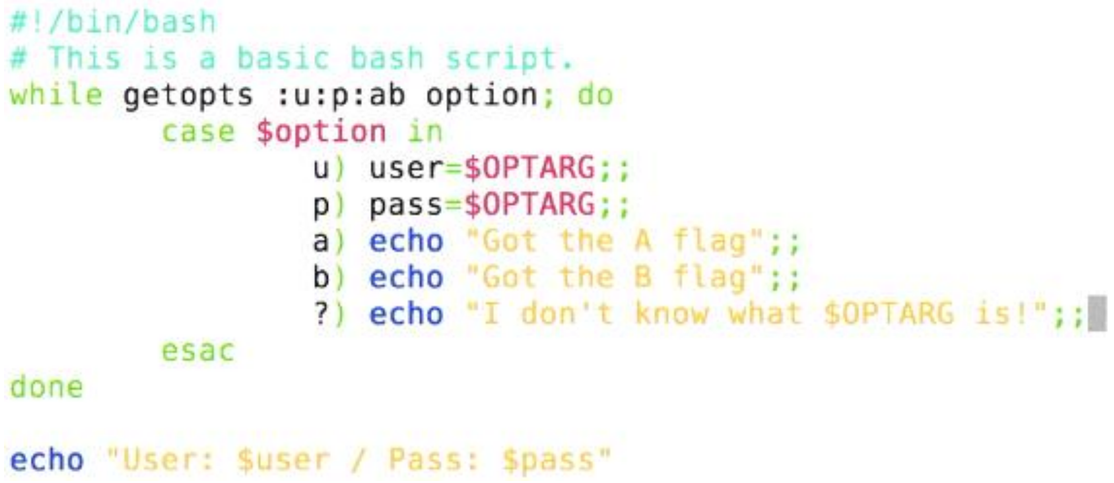
- Now we can use default option ('*)') or ?) option to capture illegal flag



```
?) echo "I don't know what $OPTARG is!";;
```

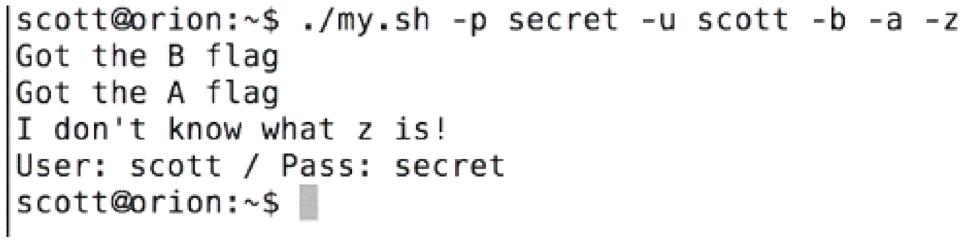
(now whenever a invalid flag come it will be captured by ?) or *) and we can know or prompt message.
Note- it is better to use ?) option as it is made specifically for this purpose.

Example - 3 for taking -u ,-p and some other(which are not necessary) and some invalid flags as well.



```
#!/bin/bash
# This is a basic bash script.
while getopts :u:p:ab option; do
        case $option in
                u) user=$OPTARG;;
                p) pass=$OPTARG;;
                a) echo "Got the A flag";;
                b) echo "Got the B flag";;
                ?) echo "I don't know what $OPTARG is!";;
        esac
done

echo "User: $user / Pass: $pass"
```

RUNNING-

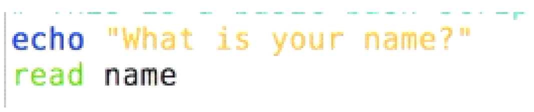


```
scott@orion:~$ ./my.sh -p secret -u scott -b -a -z
Got the B flag
Got the A flag
I don't know what z is!
User: scott / Pass: secret
scott@orion:~$ 
```

Read Keyword

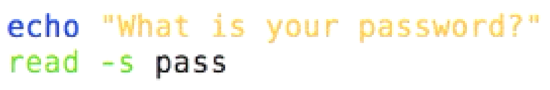
Monday, December 19, 2016 8:23 PM

- Some times we might need to get input at runtime,
- We can use 'read' keyword for it
- read keyword stops script until input is given and **'enter'** is clicked



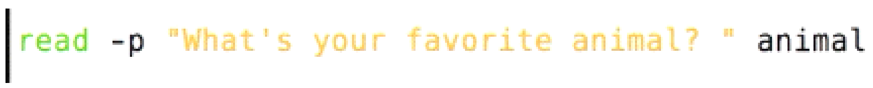
```
echo "What is your name?"
read name
```

- Silence option (-s) lets 'read' not to show letter your enter(used for password), in this nothing will be displayed not even astricks(*)



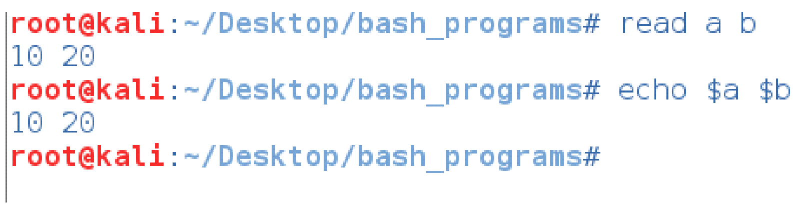
```
echo "What is your password?"
read -s pass
```

- We need to use 'echo' for prompting because 'read' has prompt option(-p) Prompt option prompt content passed to it , and read in variable after it.



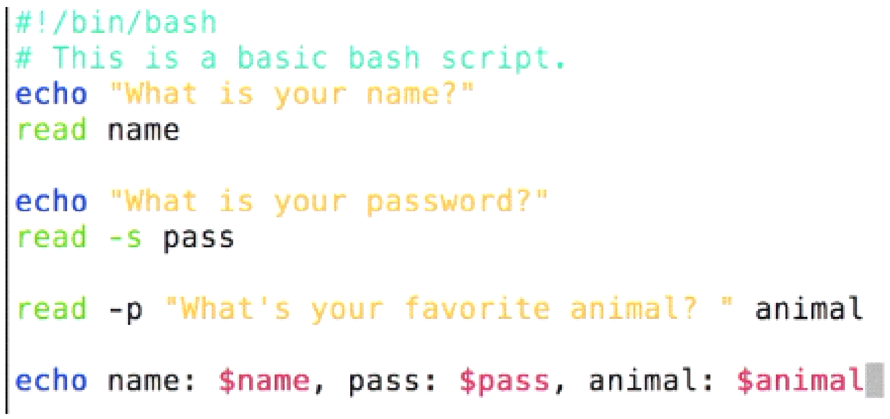
```
read -p "What's your favorite animal? " animal
```

- One '**enter**' is required for 1 '**read'** so if we get multiple input from 1 read we will give space in between..



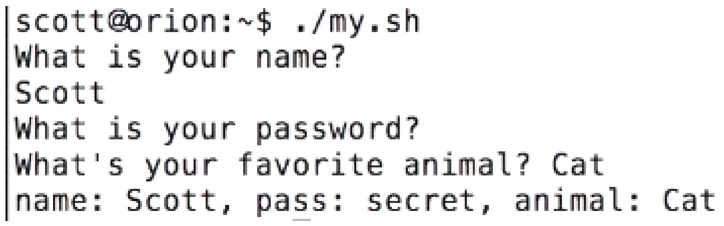
```
root@kali:~/Desktop/bash_programs# read a b
10 20
root@kali:~/Desktop/bash_programs# echo $a $b
10 20
root@kali:~/Desktop/bash_programs#
```

Example-



```
#!/bin/bash
# This is a basic bash script.
echo "What is your name?"
read name

echo "What is your password?"
read -s pass

read -p "What's your favorite animal? " animal

echo name: $name, pass: $pass, animal: $animal
```

OUTPUT-



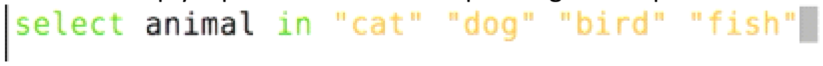
```
scott@orion:~$ ./my.sh
What is your name?
Scott
What is your password?
What's your favorite animal? Cat
name: Scott, pass: secret, animal: Cat
```

(note that password 'secret' is not shown as it is silent)

Select Input

Monday, December 19, 2016 8:32 PM

- Select input is very helpful when we want user to select a choice from some options, User can simply input a number corresponding to an option.

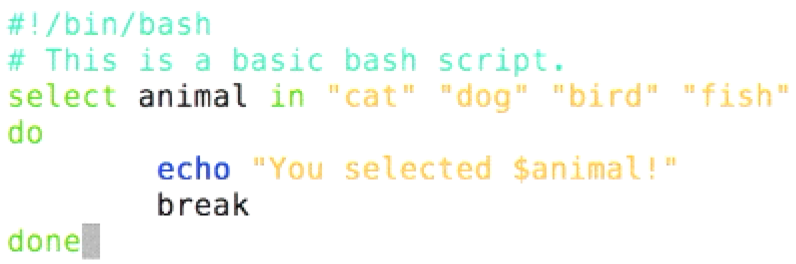


```
select animal in "cat" "dog" "bird" "fish"
```

- Value of option will be assigned to variable(here 'animal') which we can use .
- Select input has two parts
  1- Question (using select)
  2- Response (using do loop)

- In Question we ask question and in Response we do whatever we want to do with that input given by user.

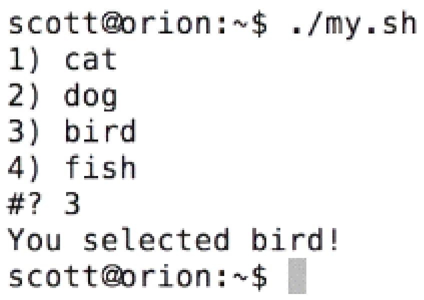
Example-1 using select input to select an animal,



```
#!/bin/bash
# This is a basic bash script.
select animal in "cat" "dog" "bird" "fish"
do
        echo "You selected $animal!"
        break
done
```

- It is important to 'break' out of loop otherwise control will be there in do loop forever.

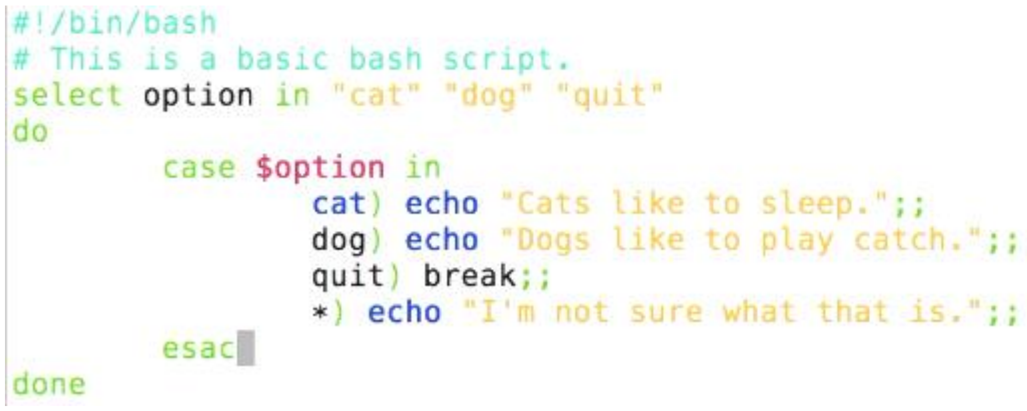
OUTPUT-



```
scott@orion:~$ ./my.sh
1) cat
2) dog
3) bird
4) fish
#? 3
You selected bird!
scott@orion:~$
```

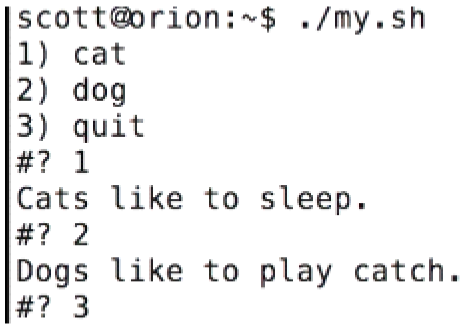
Example -2 using select input with case

- Select inputs are most commonly used with 'case' statement as we can implement separate functionality for each option using 'case' statement.
- Here do loop runs until quite is selected which does break.



```
#!/bin/bash
# This is a basic bash script.
select option in "cat" "dog" "quit"
do
        case $option in
                cat) echo "Cats like to sleep.";;
                dog) echo "Dogs like to play catch.";;
                quit) break;;
                *) echo "I'm not sure what that is.";;
        esac
done
```

RUNNING-



```
scott@orion:~$ ./my.sh
1) cat
2) dog
3) quit
#? 1
Cats like to sleep.
#? 2
Dogs like to play catch.
#? 3
```

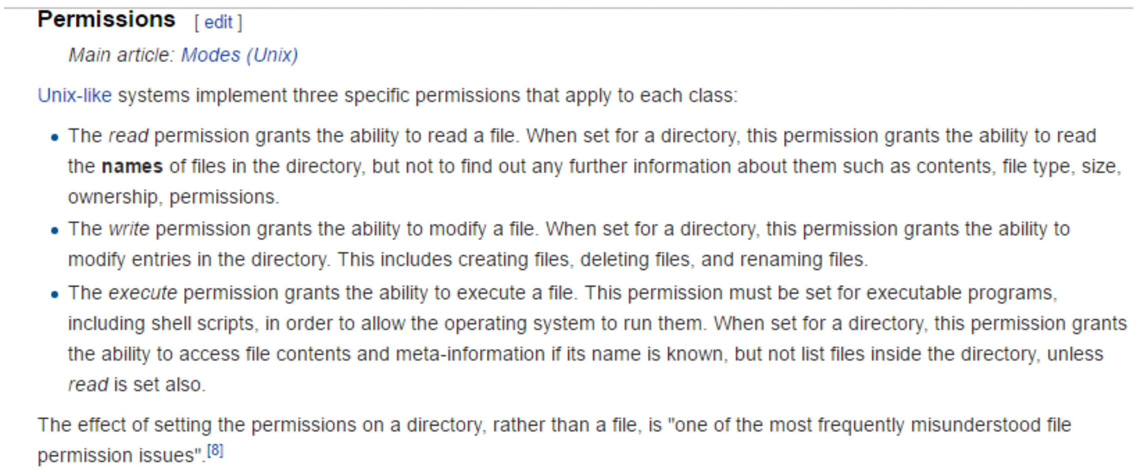
Insuring responce

Monday, December 19, 2016 8:45 PM

- This includes implementing using loop
  (see video)

Permissions

Wednesday, February 15, 2017 11:36 PM



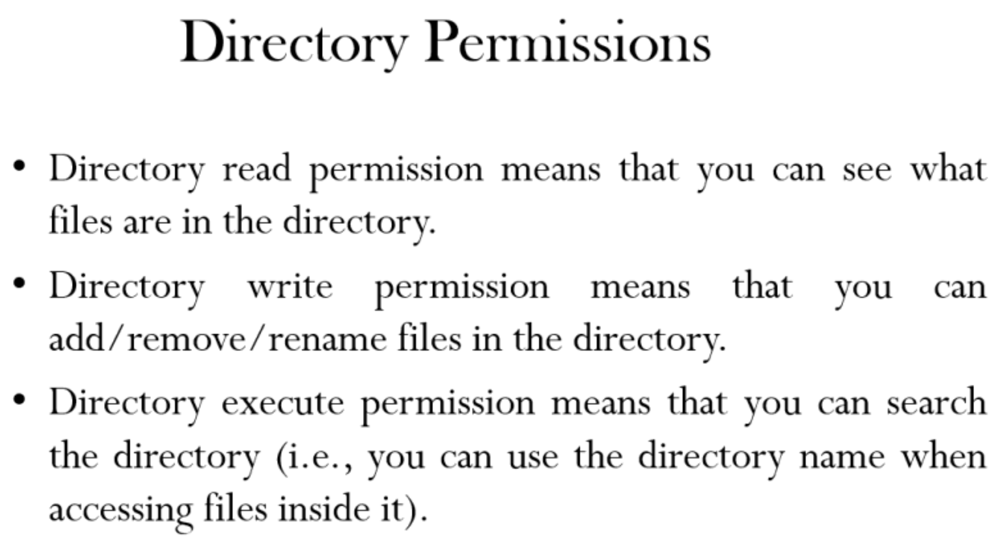
Permissions [edit]

Main article: Modes (Unix)

Unix-like systems implement three specific permissions that apply to each class:

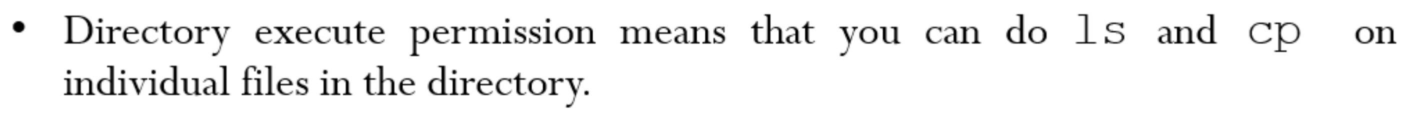
- The *read* permission grants the ability to read a file. When set for a directory, this permission grants the ability to read the **names** of files in the directory, but not to find out any further information about them such as contents, file type, size, ownership, permissions.
- The *write* permission grants the ability to modify a file. When set for a directory, this permission grants the ability to modify entries in the directory. This includes creating files, deleting files, and renaming files.
- The *execute* permission grants the ability to execute a file. This permission must be set for executable programs, including shell scripts, in order to allow the operating system to run them. When set for a directory, this permission grants the ability to access file contents and meta-information if its name is known, but not list files inside the directory, unless *read* is set also.

The effect of setting the permissions on a directory, rather than a file, is "one of the most frequently misunderstood file permission issues".[8]



Directory Permissions

- Directory read permission means that you can see what files are in the directory.
- Directory write permission means that you can add/remove/rename files in the directory.
- Directory execute permission means that you can search the directory (i.e., you can use the directory name when accessing files inside it).



- Directory execute permission means that you can do `ls` and `cp` on individual files in the directory.

Sed Command

Wednesday, April 26, 2017 7:12 PM

Substitution in SED

- 's' character will be used for substitution.
- This will only replace first occurance if wanted to replace all use
  sed 's/6.30/7.30/g'
- For specifically nth use n in place of g
  sed 's/6.30/7.30/3'
  This will replace all 3rd occurance.
- We can use or to replace
  sed 's/ashish|amit/mohit/g' file.txt
- We can use egrep regex also.
  sed 's/a+/amit/g' file.txt

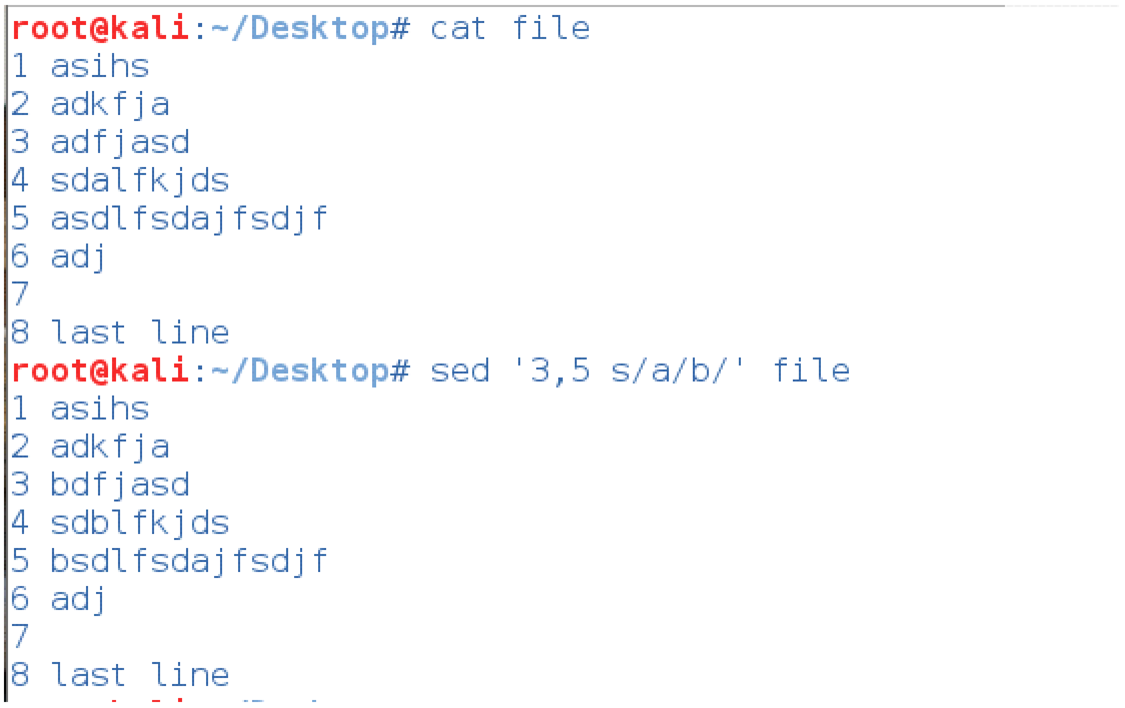
Filtering Some lines

- We can just give range of lines to filter

  $ sed '1,20 s/amit/ashish/g'

- If we want to filter everything except this we can also do using ! operator

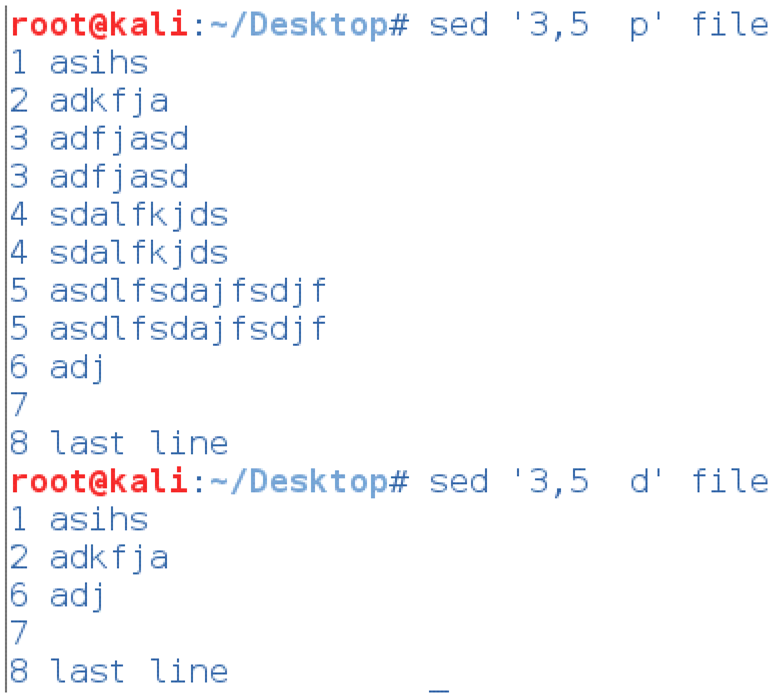
  $ sed '1,20 !s/amit/ashish/g'
  It says to substitute everywhere except from line 1 to 20



```
root@kali:~/Desktop# cat file
1 asihs
2 adkfja
3 adfjasd
4 sdalfkjds
5 asdlfsdajfsdjf
6 adj
7
8 last line
root@kali:~/Desktop# sed '3,5 s/a/b/' file
1 asihs
2 adkfja
3 bdfjasd
4 sdblfkjds
5 bsdlfsdajfsdjf
6 adj
7
8 last line
```

Printing and Deleting

- Similarly like 's' used for substitution we can use 'p' for printing 'd' for deletion.


```
root@kali:~/Desktop# sed '3,5  p' file
1 asihs
2 adkfja
3 adfjasd
3 adfjasd
4 sdalfkjds
4 sdalfkjds
5 asdlfsdajfsdjf
5 asdlfsdajfsdjf
6 adj
7
8 last line
root@kali:~/Desktop# sed '3,5  d' file
1 asihs
2 adkfja
6 adj
7
8 last line
```

Awk Command

Wednesday, April 26, 2017 7:13 PM

Awk Introduction and Printing Operations

Awk is a programming language which allows easy manipulation of structured data and the generation of formatted reports. Awk stands for the names of its authors "**A**ho, **W**einberger, and **K**ernighan"

The Awk is mostly used for pattern scanning and processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform associated actions.

Some of the key features of Awk are:

- Awk views a text file as records and fields.
- Like common programming language, Awk has variables, conditionals and loops

- Awk has arithmetic and string operators.
- Awk can generate formatted reports

Awk reads from a file or from its standard input, and outputs to its standard output. Awk does not get along with non-text files.

```
Syntax:

awk '/search pattern1/ {Actions}
     /search pattern2/ {Actions}' file
```

In the above awk syntax:

- search pattern is a regular expression.
- Actions – statement(s) to be performed.
- several patterns and actions are possible in Awk.
- file – Input file.
- Single quotes around program is to avoid shell not to interpret any of its special characters.

Awk Working Methodology

1. Awk reads the input files one line at a time.
2. For each line, it matches with given pattern in the given order, if matches performs the corresponding action.
3. If no pattern matches, no action will be performed.
4. In the above syntax, either search pattern or action are optional, But not both.
5. If the search pattern is not given, then Awk performs the given actions for each line of the input.
6. If the action is not given, print all that lines that matches with the given patterns which is the default action.
7. Empty braces with out any action does nothing. It wont perform default printing operation.
8. Each statement in Actions should be delimited by semicolon.

If the column separator is something other than spaces or tabs, such as a comma, you can specify that in the awk statement as follows:

```
awk -F, '{ print $3 }' table1.txt > output1.txt
```

Let us create employee.txt file which has the following content, which will be used in the examples mentioned below.

```
$cat employee.txt
100  Thomas  Manager    Sales       $5,000
200  Jason   Developer  Technology  $5,500
300  Sanjay  Sysadmin   Technology  $7,000
400  Nisha   Manager    Marketing   $9,500
500  Randy   DBA        Technology  $6,000
```

Awk Example 1. Default behavio r of Awk

By default Awk prints every line from the file.

```
$ awk '{print;}' employee.txt
100  Thomas  Manager    Sales       $5,000
200  Jason   Developer  Technology  $5,500
300  Sanjay  Sysadmin   Technology  $7,000
400  Nisha   Manager    Marketing   $9,500
500  Randy   DBA        Technology  $6,000
```

In the above example pattern is not given. So the actions are applicable to all the lines. Action print with out any argument prints the whole line by default. So it prints all the lines of the file with out fail. Actions has to be enclosed with in the braces.

Awk Example 2. Print the lines which matches with the pattern.

```
$ awk '/Thomas/
> /Nisha/' employee.txt
100  Thomas  Manager    Sales       $5,000
400  Nisha   Manager    Marketing   $9,500
```

In the above example it prints all the line which matches with the 'Thomas' or 'Nisha'. It has two patterns. Awk accepts any number of patterns, but each set (patterns and its corresponding actions) has to be separated by newline.

NOTE - ">" is not written here by us it automatically comes when we enter new line.

Awk Example 3. Print only spec ific field.

Awk has number of built in variables. For each record i.e line, it splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4. $0 represents whole line. NF is a built in variable which represents total number of fields in a record.

```
$ awk '{print $2,$5;}' employee.txt
Thomas $5,000
Jason $5,500
Sanjay $7,000
Nisha $9,500
Randy $6,000

$ awk '{print $2,$NF;}' employee.txt
Thomas $5,000
```

Awk Example 4. Initialization an d Final Action

Awk has two important patterns which are specified by the keyword called BEGIN and END.

```
Syntax:

BEGIN { Actions}
{ACTION} # Action for everyline in a file
END { Actions }
```

NF

Awk Example 4. Initialization an d Final Action

Awk has two important patterns which are specified by the keyword called BEGIN and END.

```
Syntax:
```

NF

```
BEGIN { Actions}
{ACTION} # Action for everyline in a file
END { Actions }

# is for comments in Awk
```

Actions specified in the BEGIN section will be executed before starts reading the lines from the input.
END actions will be performed after completing the reading and processing the lines from the input.

```
$ awk 'BEGIN {print "Name\tDesignation\tDepartment\tSalary";}
> {print $2,"\t",$3,"\t",$4,"\t",$NF;}
> END{print "Report Generated\n--------------";
> }' employee.txt
Name    Designation     Department      Salary

Thomas   Manager         Sales           $5,000
Jason    Developer       Technology      $5,500
Sanjay   Sysadmin        Technology      $7,000
Nisha    Manager         Marketing       $9,500
Randy    DBA             Technology      $6,000
Report Generated
--------------
```

In the above example, it prints headline and last file for the reports.

Awk Example 5. Find the emplo yees who has employee id gr eater than 200

```
$ awk '$1 >200' employee.txt
300  Sanjay  Sysadmin   Technology  $7,000
400  Nisha   Manager    Marketing   $9,500
500  Randy   DBA        Technology  $6,000
```

In the above example, first field ($1) is employee id. So if $1 is greater than 200, then just do the default print action to print the whole line.

Awk Example 6. Print the list of employees in Technology department

Now department name is available as a fourth field, so need to check if $4 matches with the string "Technology", if yes print the line.

```
$ awk '$4 ~/Technology/' employee.txt
200  Jason   Developer  Technology  $5,500
300  Sanjay  Sysadmin   Technology  $7,000
500  Randy   DBA        Technology  $6,000
```

Operator ~ is for comparing with the regular expressions. If it matches the default action i.e print whole line will be performed.

- As we can see tilde operator(**~) can be used to match patter with a given field.**

Awk Example 7. Print number o f employees in Technology department

The below example, checks if the department is Technology, if it is yes, in the Action, just increment the count variable, which was initialized with zero in the BEGIN section.

```
$ awk 'BEGIN { count=0;}
$4 ~ /Technology/ { count++; }
END { print "Number of employees in Technology Dept =",count;}' employee.txt
Number of employees in Tehcnology Dept = 3
```

Then at the end of the process, just print the value of count which gives you the number of employees in Technology department.

Grep

Wednesday, April 26, 2017 7:13 PM

See copy

Comm command

Wednesday, April 26, 2017 7:52 PM

Compare sorted files **FILE1** and **FILE2** line-by-line.

With no options, **comm** produces three-column output. Column one contains lines unique to **FILE1**, column two contains lines unique to **FILE2**, and column three contains lines common to both files. Each of these columns can be suppressed individually with options.

Examples

Let's say you have two text files, **recipe.txt** and **shopping-list.txt**.

recipe.txt contains these lines:

```
All-Purpose Flour
Baking Soda
Bread
Brown Sugar
Chocolate Chips
Eggs
Milk
Salt
Vanilla Extract
White Sugar
```

And **shopping-list.txt** contains these lines:

```
All-Purpose Flour
Bread
Brown Sugar
Chicken Salad
Chocolate Chips
Eggs
Milk
Onions
Pickles
Potato Chips
Soda Pop
Tomatoes
White Sugar
```

As you can see, the two files are different, but many of the lines are the same. Not all of the recipe ingredients are on the shopping list, and not everything on the shopping list is part of the recipe.

If we run the **comm** command on the two files, it will read both files and give us three columns of output:

```
comm recipe.txt shopping-list.txt
```

OUTPUT -

```
                All-Purpose Flour
Baking Soda
                Bread
                Brown Sugar
        Chicken Salad
                Chocolate Chips
                Eggs
                Milk
        Onions
        Pickles
        Potato Chips
Salt
        Soda Pop
        Tomatoes
Vanilla Extract
                White Sugar
```

Here, each line of output has either zero, one, or two tabs at the beginning, separating the output into three columns:

1. The first column (zero tabs) is lines that only appear in the first file.
2. The second column (one tab) is lines that only appear in the second file.
3. The third column (two tabs) is lines that appear in both files.