Shell

From linkedin learning
Linux: Bash Shell and Scripts

PART 1: INTRO TO BASH SCRIPTING

Manual:

https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.htm

man and *info* commands.

Shebang:

Used in scripting languages like python, perl, awk, bash etc.
Putted in start, using '#!', then providing the interpreter path.

Typically, its
#! /bin/bash

but people also write
#! /usr/bin/env bash #this makes env search the bash interpreter.

This is executed via the kernel system **execve**()

execve() which executes the program referred to by *pathname*.

```
int execve(const char *pathname, char *const _Nullable argv[],
                  char *const _Nullable envp[]);
```

execve takes the program's path name, then executes it with argv and gives it envp as environment setup things. (*envp* is an array of pointers to strings, conventionally of the

```
       form key=value, which are passed as the environment of the new
       program.)
```

Thus, a file test.sh with #!/bin/bash will be transformed somewhat as
<interpreter_path> <actual later code of test.sh> <args for test.sh and env stuffs and all>

Basically, as soon as user initiates a script, a fork is made creating a child process with its unique PID, and there the interpreter specified is used to execute the commands in the main script.

Executables:

Executables are able to be executed themselves from the pov of the executor.

But if files are not executable, and are readable from pov of executor, then the executor can feed the readable file in another executioner program like bash/sh who can read the file and execute them.

Thus,
if a file has read permission from user perspective, then without making it executable, user can do

```
$ bash file.sh
```

And if it is made executable as:

```
$ chmod u+x  #chmod is called change mod
```

Then we can use both previous method and `$ ./file.sh`

So basically, executables is an entitlement given to files which terminal or program executor (called shell) know what to do with.

Real world example:
If i give a poet a song, he can sing.
If i give a poet a chair, well, he can do transcendental sitting, but not poet thing.

So, program executor knows executables
and files are therefore in two colors from their perspective.
Normal ones, and executables.

Why dot slash on executables?
Typically current path isn't in path (searchable path list, which i think PATH or some such variables/storage have listed), so we need to specify the path using current directory (.) and relation /filename.
Thus, if the executable is in the listed paths, then we can directly call

```
$ file.sh
```

or simeply

```
$ programname
```

There is no need to give '.sh' or such names to files. Its just for information purpose, and doesn't matter from the pov of the executor shell.

Shell vs Bash/ksh... vs terminal:

shell is a concept, a way to interact with OS .
From wik: **shell** is a computer program that exposes an operating system's services to a human user or other programs.

Implementations are of various types, bash/ksh/csh/sh/zsh etc.
Terminal (of mordern distros like ubuntu etc) is a UI provided which essentially serves a shell inside it.
Using

```
$ echo $0
```

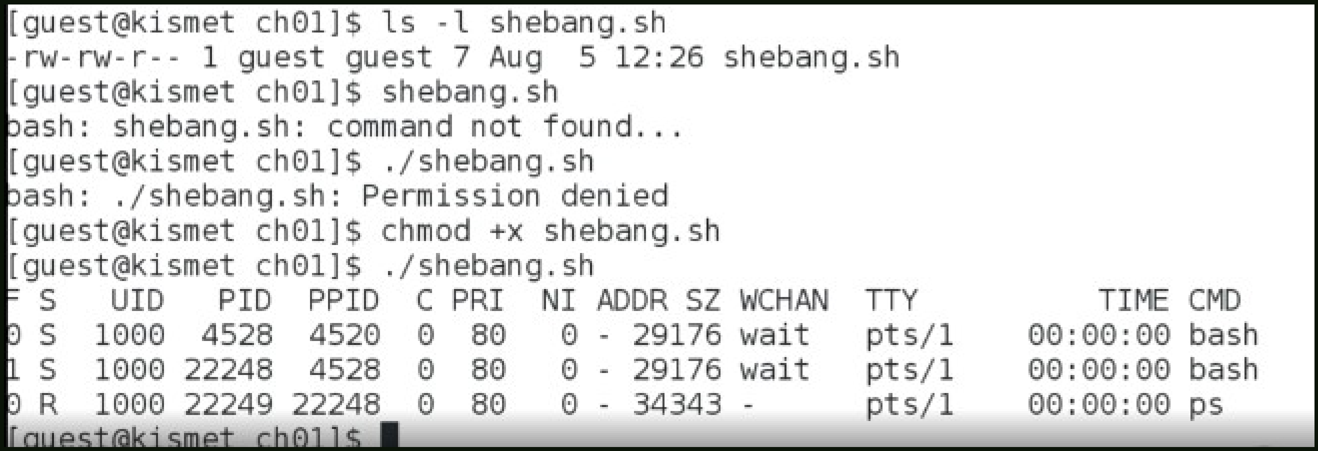
will give shell's name

Or using

```
$ ps $$
```

will give process details about the current shell’s PID ($$)

```
$ echo filename: $0, PID: $$
filename: bash, PID: 3496
$ ps $$
    PID TTY      STAT   TIME COMMAND
   3496 pts/0    Ss     0:00 bash
```

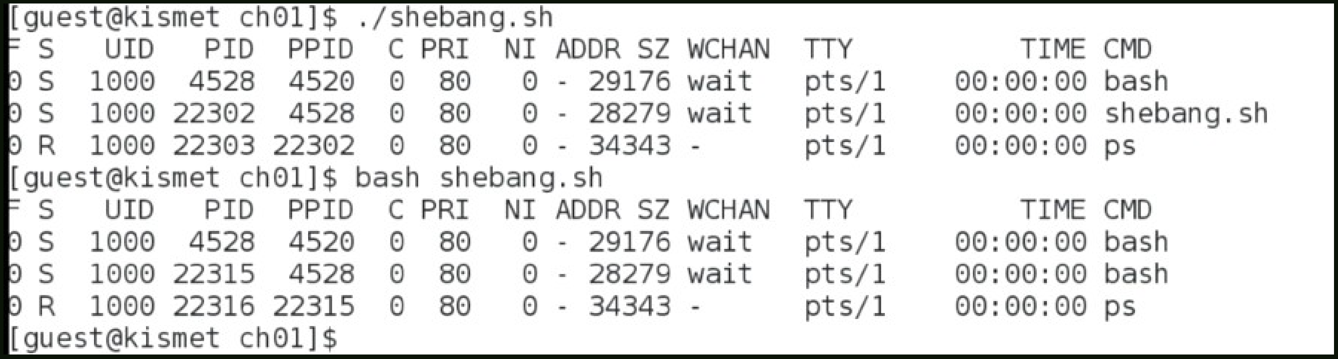


```
[guest@kismet ch01]$ ls -l shebang.sh
-rw-rw-r-- 1 guest guest 7 Aug  5 12:26 shebang.sh
[guest@kismet ch01]$ shebang.sh
bash: shebang.sh: command not found...
[guest@kismet ch01]$ ./shebang.sh
bash: ./shebang.sh: Permission denied
[guest@kismet ch01]$ chmod +x shebang.sh
[guest@kismet ch01]$ ./shebang.sh
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  4528  4520  0  80   0 - 29176 wait   pts/1    00:00:00 bash
1 S  1000 22248  4528  0  80   0 - 29176 wait   pts/1    00:00:00 bash
0 R  1000 22249 22248  0  80   0 - 34343 -      pts/1    00:00:00 ps
[guest@kismet ch01]$
```

User can invoke bash to execute a file no matter executable or not.
User can’t ask already invoked bash to execute a non-executable file.

From the shell’s pov, first operation is me feeding it non-executable, and second operation is me feeding it executable with params for it.

PID is process ID
PPID is parent process ID of PID



```
[guest@kismet ch01]$ ./shebang.sh
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  4528  4520  0  80   0 - 29176 wait   pts/1    00:00:00 bash
0 S  1000 22302  4528  0  80   0 - 28279 wait   pts/1    00:00:00 shebang.sh
0 R  1000 22303 22302  0  80   0 - 34343 -      pts/1    00:00:00 ps
[guest@kismet ch01]$ bash shebang.sh
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  4528  4520  0  80   0 - 29176 wait   pts/1    00:00:00 bash
0 S  1000 22315  4528  0  80   0 - 28279 wait   pts/1    00:00:00 bash
0 R  1000 22316 22315  0  80   0 - 34343 -      pts/1    00:00:00 ps
[guest@kismet ch01]$
```

Here we see, difference between bash invoked by user vs using shell bash.

POSIX:
Portable OS interface.
Its a standard, specified by IEEE. An attempt to maintain compatibility between OSes.
When Unix variants were coming, a need for common OS features was felt. Many user-level programs, services, and utilities (including awk, echo, ed) were standardized, along with required program-level services (including basic I/O: file, terminal, and network). POSIX also defines a standard threading library API which is supported by most modern operating systems.

Variables:
name=value (name = value doesn't work, cuz that makes shell see it as command 'name' is to be executed with args '=' and 'value')

Since most of the time, child processes are spawned, and they might need the variable, so we can use 'export'.

`export` is defined in POSIX:

> The shell shall give the export attribute to the variables corresponding to the specified names, which shall cause them to be in the environment of subsequently executed commands. If the name of a variable is followed by = word, then the value of that variable shall be set to word.

To remove a variable, we can use 'unset' command as:

```
$ export var=23
$ echo $var
23
$ unset var
$ echo $var
$
```

To just declare a variable, we can use '**declare**' command as:
$ declare var
And to also export it, we can use
$ declare -x var

To export a function, we can use $ export -f funcname

To check what's exported in current shell's env:
$ export

Braces {} and parenthesis () :

{} just groups items.
() starts new process, and items declared there ends, goes away.

Enable and Disable builtins:

Builtin commands are there.
Sometimes we might wanna use disk commands instead which have same name, so disabling builtins gives a power.
Since there is disable, we have enable too.

```
enable: enable [-a] [-dnps] [-f filename] [name ...]
    Enable and disable shell builtins.

    Enables and disables builtin shell commands.  Disabling allows you to
    execute a disk command which has the same name as a shell builtin
    without using a full pathname.

    Options:
      -a	print a list of builtins showing whether or not each is enabled
      -n	disable each NAME or display a list of disabled builtins
      -p	print the list of builtins in a reusable format
      -s	print only the names of Posix `special' builtins

    Options controlling dynamic loading:
      -f	Load builtin NAME from shared object FILENAME
      -d	Remove a builtin loaded with -f
```

What if we do

```
$ enable -n ls
bash: enable: ls: not a shell builtin
```

ls, and cat aren’t shell builtins, but cd is, so let’s try it.

```
$ enable -n cd
$ cd
cd: command not found
```

To renable, just $ enable cd

Creating new shell:

```
$ bash
$
```

Time and sleep:
There is ‘time’ command to determine the time used by arg process, in real time, in cpu time etc.
There is ‘sleep’ command, similar to python’s sleep.

```
$ sleep 2   # make it sleep for 2 seconds.
```

Bash startup:

<u>Shell config files and their need:</u>

When a shell starts, either when a user logs in a shell/terminal, or when explicitly shells are invoked, both are expected to have some GK (general knowledge) instead of being completely blank.
This is quite an observation in CS terms. But yeah, so this is where shell config files are used.

<u>A note on that from an SO answer (a/415444):</u>

The main difference with shell config files is that some are only read by "login" shells (eg. when you login from another host, or login at the text console of a local unix machine). these are the ones called, say, `.login` or `.profile` or `.zlogin` (depending on which shell you're using).

Then you have config files that are read by "interactive" shells (as in, ones connected to a terminal (or pseudo-terminal in the case of, say, a terminal emulator running under a windowing system). these are the ones with names like `.bashrc`, `.tcshrc`, `.zshrc`, etc.

`bash` complicates this in that `.bashrc` is **only read by a shell that's both *interactive* and *non-login***, so you'll find most people end up telling their `.bash_profile` to also read `.bashrc` with something like`

```
[[ -r ~/.bashrc ]] && . ~/.bashrc
```

Other shells behave differently - eg with `zsh`, `.zshrc` is always read for an interactive shell, whether it's a login one or not.

The manual page for bash explains the circumstances under which each file is read. Yes, behaviour is generally consistent between machines.

`.profile` is simply the login script filename originally used by `/bin/sh`. `bash`, being generally backwards-compatible with `/bin/sh`, will read `.profile` if one exists.

<u>The rc:</u>
The "rc" in `.bashrc` stands for "runcom," which is short for "run commands." It's a tradition that dates back to early Unix systems.

In the past, "rc" was used to refer to files that contained commands to be executed when starting a particular program or shell. For example, the "rc" file for the editor `vi` was called `.exrc`, and the `tcsh` shell has a file called `.cshrc`.

<u>What to store where, and why this matters:</u>

Because of the fact that one runs after login, and one runs with each shell startup, noting also that bashrc doesn’t run on login, we can use them to do stuffs that need to be done once, and those which needs to be done each time.

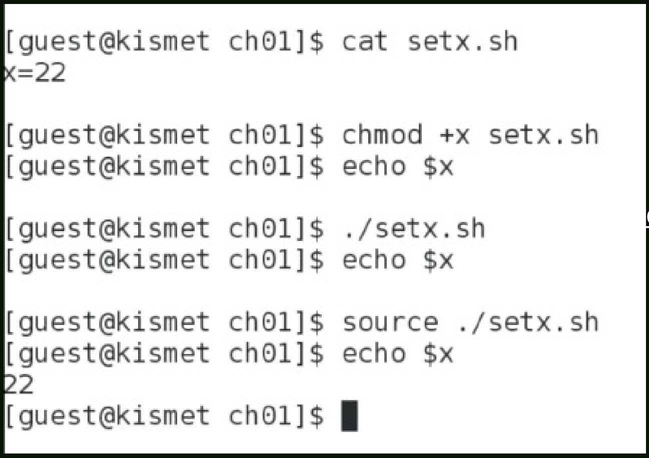
Core things like DNA are done in bash_profile, and later movable, a little flexible, and a little more complicated stuffs/extended body are kept in bashrc.

Queen needs to move quickly and fastly, but not king.
Configs that are complicated and long, and that might need refresh etc frequently, they can't think of staying specifically in login shell configs.

Cells needs to die and born again, but dna structure is made once per body.

The config files act like those were sourced in the shell.
Thus, if say I modify .bashrc, but I don't want to use new shell but still want changes to be visible, I can just do:
source ~/.bashrc

Sourcing and aliasing with bash:



```
[guest@kismet ch01]$ cat setx.sh
x=22

[guest@kismet ch01]$ chmod +x setx.sh
[guest@kismet ch01]$ echo $x

[guest@kismet ch01]$ ./setx.sh
[guest@kismet ch01]$ echo $x

[guest@kismet ch01]$ source ./setx.sh
[guest@kismet ch01]$ echo $x
22
[guest@kismet ch01]$ 
```

Executing in current vs subshells.

<u>Without sourcing, execution doesn't run in current shell process.</u>

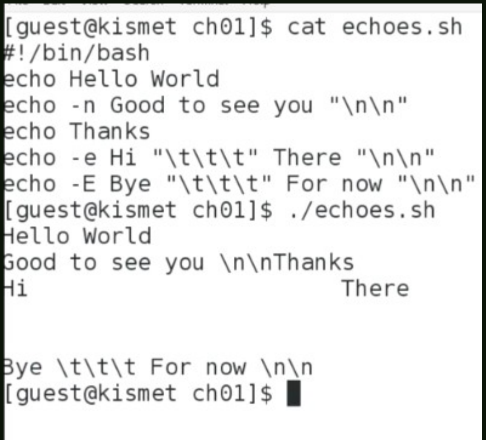
. <space> ./setx.sh will also source the file, its same as using source command.

Aliasing syntax can be like: $ alias alias_name=command string

$ alias ls=list
$ alias ll='ls -l'
etc.

To unalias, we can use $ unalias alias_name
To check alias, we can use $ alias alias_name
Example:
$ alias ll
alias ll='ls -alF'

echo:



```
[guest@kismet ch01]$ cat echoes.sh
#!/bin/bash
echo Hello World
echo -n Good to see you "\n\n"
echo Thanks
echo -e Hi "\t\t\t" There "\n\n"
echo -E Bye "\t\t\t" For now "\n\n"
[guest@kismet ch01]$ ./echoes.sh
Hello World
Good to see you \n\nThanks
Hi                          There


Bye \t\t\t For now \n\n
[guest@kismet ch01]$ 
```

echo * is like asking echo to echo everything the current directory has, so it is like ls.

ls * instead will list everything in current dir and contents of dir in current dir.

-n denies new line at the tail.
-e enables special meanings, -E disables special meanings.

Thus:

```
$ echo Hi there '\n'
Hi there \n
$ echo -e Hi there '\n'
Hi there

$
```

The -e and -E are for backslash interpretation enable/disable thing.

PART 2: VARIABLES, CONTROL STRUCTURES AND ARITHMETICS

typeset and declare commands:

```
$ help typeset
typeset: typeset [-aAfFgilnrtux] [-p] name[=value] ...
    Set variable values and attributes.

    A synonym for `declare'.  See `help declare'.
```

The ‘-i’ option if used to declare integers, this typeset sets int type for that, making compiler and post processing therefore overall performance boost.

```
$ help let
let: let arg [arg ...]
    Evaluate arithmetic expressions.

    Evaluate each ARG as an arithmetic expression.  Evaluation is done in
    fixed-width integers with no check for overflow, though division by 0
    is trapped and flagged as an error.
    ...

   Shell variables are allowed as operands.  The name of the variable
    is replaced by its value (coerced to a fixed-width integer) within
    an expression.  The variable need not have its integer attribute
    turned on to be used in an expression.

    Operators are evaluated in order of precedence.  Sub-expressions in
    parentheses are evaluated first and may override the precedence
    rules above.

    Exit Status:
            If the last ARG evaluates to 0, let returns 1; let returns 0 otherwise.
```

let doesn’t return the value of expression, it does the evaluation of expression, and that evaluation also takes assignment as expression too. So, a=b is expression.

Thus,

```
$ let "var1 = 6" "var2 = 2" "var3=var1+var2"; echo $var3
8
```

Profit happened in the sense that, arithmetic didn’t needed strange `` or $ or $(( ))

There are prefix,postfix etc options available also, like x++ etc, also var1 = 6 has space around ‘=’ yet valid.

```
$ declare
declare: declare [-aAfFgilnrtux] [-p] [name[=value] ...]
    Set variable values and attributes.

    Declare variables and give them attributes.  If no NAMEs are given,
    display the attributes and values of all variables.
    ...
    Using `+' instead of `-' turns off the given attribute.

    Variables with the integer attribute have arithmetic evaluation (see
    the `let' command) performed when the variable is assigned a value.

    When used in a function, `declare' makes NAMEs local, as with the `local'
    command.  The `-g' option suppresses this behavior.

    Exit Status:
    Returns success unless an invalid option is supplied or a variable
    assignment error occurs.
```

Taking ‘-l’ for example,

```
$ declare -l lowername #(there is then -u also)
$ lowername=Ankit
$ echo $lowername
ankit

$ declare -i var
$ var=abcd
$ echo $var
0
```

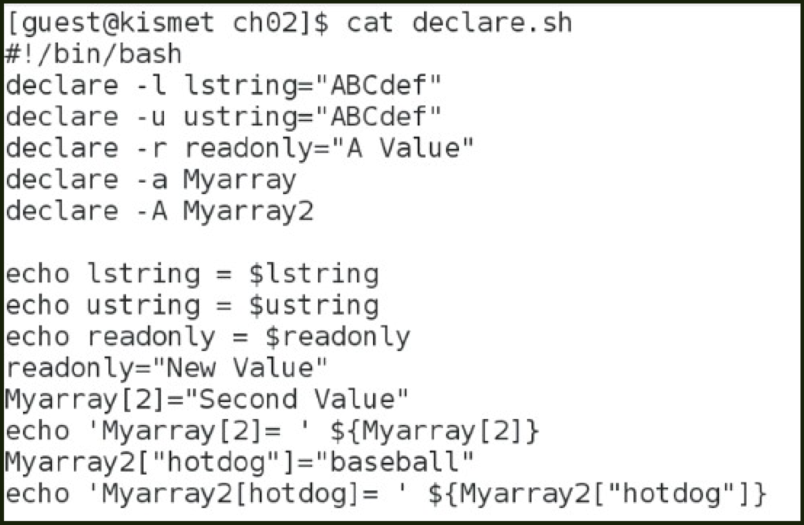
Noticable points:

1. declare and its synonym typeset if used in func, it makes var local, as with the ‘local’ command. The -g option suppresses this behavior.

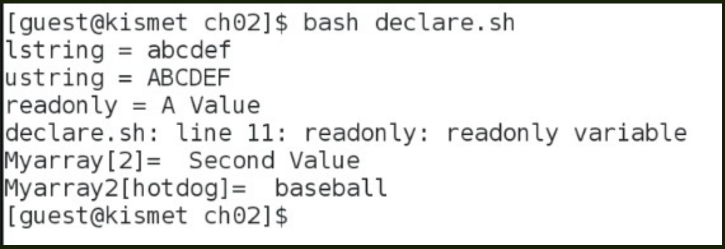
Options which set attributes:

-a	to make NAMEs indexed arrays (if supported)
-A	to make NAMEs associative arrays (if supported)
-i	to make NAMEs have the `integer' attribute
-l	to convert the value of each NAME to lower case on assignment
-n	make NAME a reference to the variable named by its value
-r	to make NAMEs readonly
-t	to make NAMEs have the `trace' attribute
-u	to convert the value of each NAME to upper case on assignment
-x	to make NAMEs export

indexed array means elements are accessible by index, associative arrays have elements accessible by an associative key, some call it key-value array, or dictionary...



```
[guest@kismet ch02]$ cat declare.sh
#!/bin/bash
declare -l lstring="ABCdef"
declare -u ustring="ABCdef"
declare -r readonly="A Value"
declare -a Myarray
declare -A Myarray2

echo lstring = $lstring
echo ustring = $ustring
echo readonly = $readonly
readonly="New Value"
Myarray[2]="Second Value"
echo 'Myarray[2]= ' ${Myarray[2]}
Myarray2["hotdog"]="baseball"
echo 'Myarray2[hotdog]= ' ${Myarray2["hotdog"]}
```

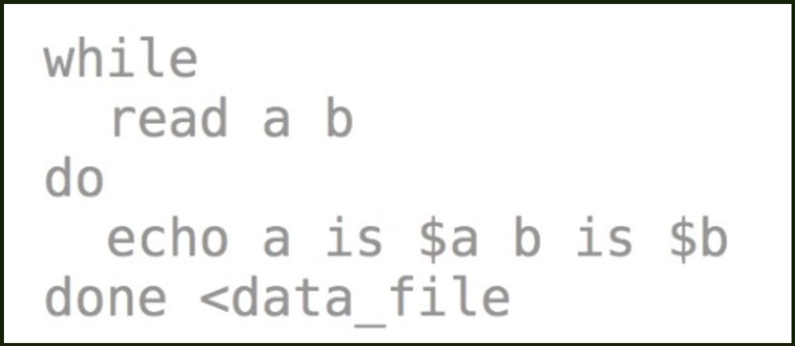


```
[guest@kismet ch02]$ bash declare.sh
lstring = abcdef
ustring = ABCDEF
readonly = A Value
declare.sh: line 11: readonly: readonly variable
Myarray[2]=  Second Value
Myarray2[hotdog]=  baseball
[guest@kismet ch02]$
```

looping with for/while sequences and reading input:

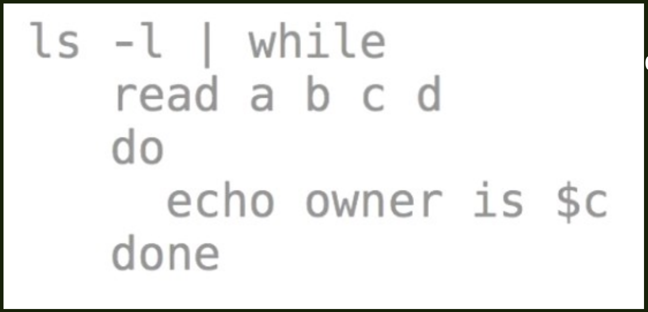
$ read a b
var oh no lets see
$ echo $a
var
$ echo $b
oh no lets see

(first word in a, left ones in b)



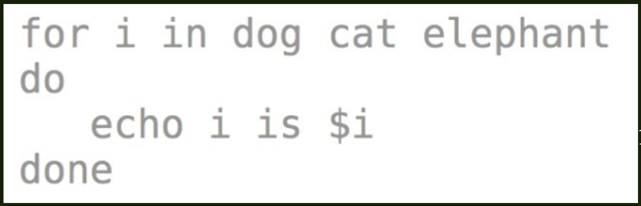
```
while
  read a b
do
  echo a is $a b is $b
done <data_file
```

To do a loopy read on an input file



```
ls -l | while
   read a b c d
   do
     echo owner is $c
   done
```

This makes while loop take piped input, loops on each line.



```
for i in dog cat elephant
do
   echo i is $i
done
```

for var in list; do commands; done

An alternate form is
for ((expr1 ; expr2 ; expr3)) ; do commands ; done

seq is like range of python,

```
$ seq 1 5
1
2
3
4
5
$
```

for var in `seq 1 5`; do echo $var; done

This backtick executes command, grabs output, gives to loop. The space /newline determines the list.
(for loop therefore goes per word of a file which contains multi words multi lines, so things like
for word in $(<datafile); do echo $word; done)

Somewhat similar use is of braces like:

```
$ echo {A..Z}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

$ echo {1..6}
1 2 3 4 5 6
```

There is cat simile **command 'nl'** that enumerates the lines. Good thing.

```
~$ nl .bashrc
     1	# ~/.bashrc: executed by bash(1) for non-login shells.
     2	# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
     3	# for examples

     4	# If not running interactively, don't do anything
     5	case $- in
     6	    *i*) ;;
     7	      *) return;;
     8	esac
```

Just like less, there is more, but more is not that recommended. From the 'man more' we get a para as:

```
       more is a filter for paging through text one screenful at a time.  This
       version is especially primitive.  Users  should  realize  that  less(1)
       provides more(1) emulation plus extensive enhancements.
```

Defining functions and using return and exit:

fname () compound-command [redirections]

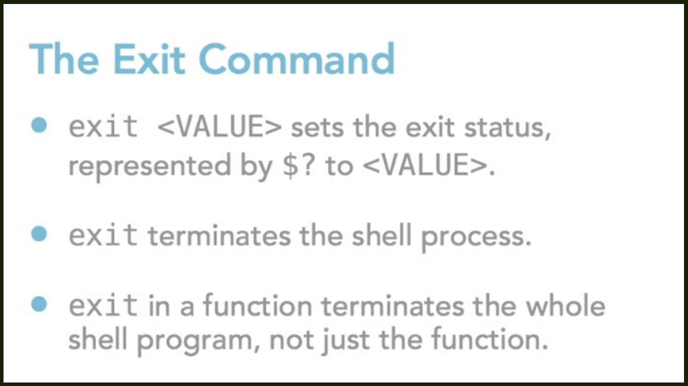
or

function fname [()] compound-command [redirections]

They group commands, and can return values.

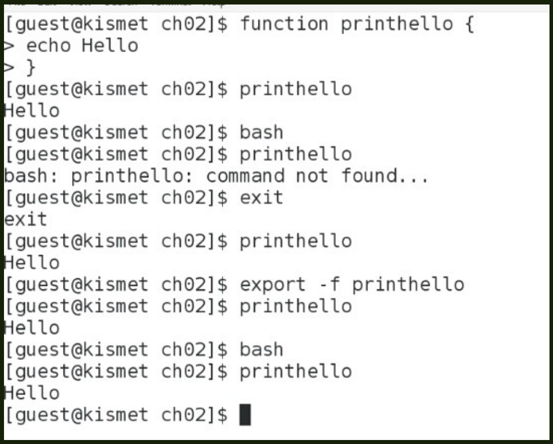
Example:

```
$ printhello () { echo 'hi'; }
$ hvar=$(printhello)
$ echo $hvar
hi
```

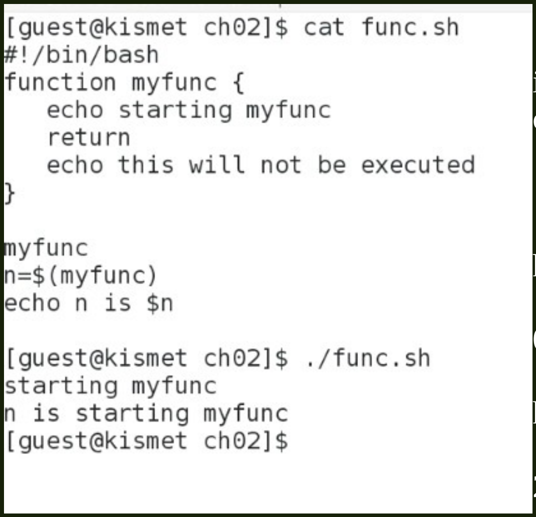


The Exit Command

- exit <VALUE> sets the exit status, represented by $? to <VALUE>.
- exit terminates the shell process.
- exit in a function terminates the whole shell program, not just the function.



```
[guest@kismet ch02]$ function printhello {
> echo Hello
> }
[guest@kismet ch02]$ printhello
Hello
[guest@kismet ch02]$ bash
[guest@kismet ch02]$ printhello
bash: printhello: command not found...
[guest@kismet ch02]$ exit
exit
[guest@kismet ch02]$ printhello
Hello
[guest@kismet ch02]$ export -f printhello
[guest@kismet ch02]$ printhello
Hello
[guest@kismet ch02]$ bash
[guest@kismet ch02]$ printhello
Hello
[guest@kismet ch02]$ 
```

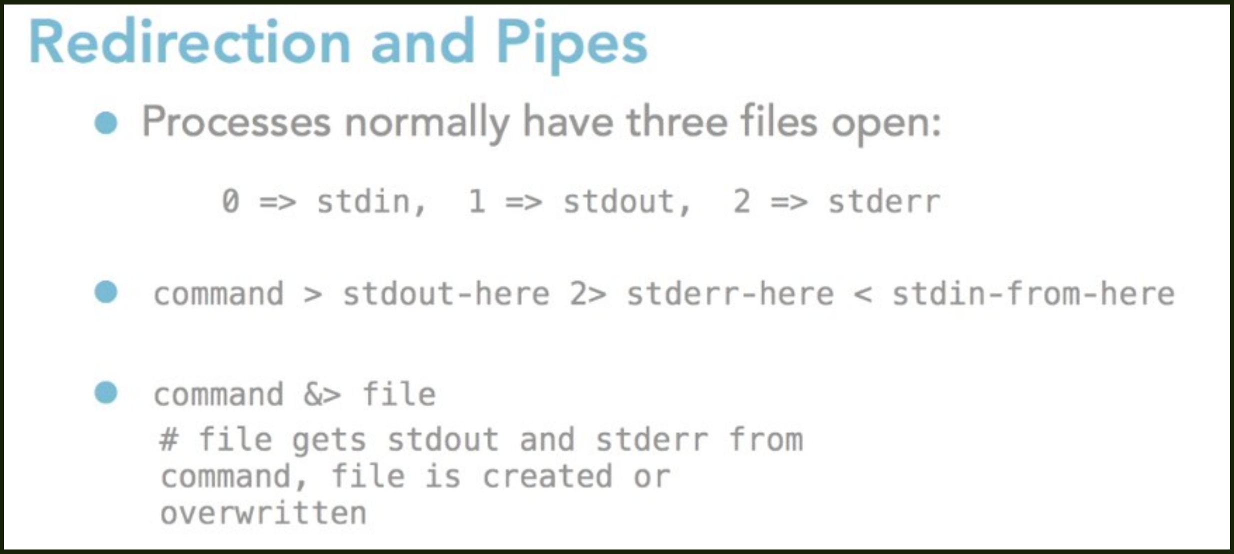


```
[guest@kismet ch02]$ cat func.sh
#!/bin/bash
function myfunc {
   echo starting myfunc
   return
   echo this will not be executed
}

myfunc
n=$(myfunc)
echo n is $n

[guest@kismet ch02]$ ./func.sh
starting myfunc
n is starting myfunc
[guest@kismet ch02]$
```

This is unexpected for me. Seems like it catches echo instead of returned value. For that I think we need to check $?

```
$ printhello () { echo 'hi'; return 23; }
$ echo $(printhello)
hi
$ echo $?
0
$ printhello
hi
$ echo $?
23
```

Using file descriptors, file redirection, pipes, and here documents:



Redirection and Pipes

- Processes normally have three files open:

```
0 => stdin,  1 => stdout,  2 => stderr
```

- `command > stdout-here 2> stderr-here < stdin-from-here`

-
  ```
  command &> file
  # file gets stdout and stderr from
  command, file is created or
  overwritten
  ```

&> means both stdout and stderr in that file.

Pipe:

$ command1 | command2

(command2's stdin is command1's stdout)

$ command1 2>&1 | command2

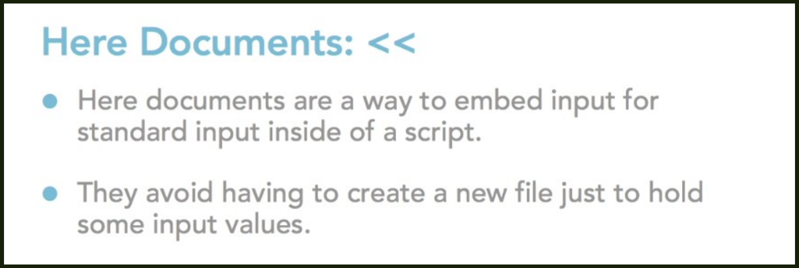
(the 2> then &1 made stderr go to where stdout goes., thus command2 gets both error and output of command1)

For at least bash, we have $ cmd1 |& cmd2 same as $ cmd1 2>&1 | cmd2

To not overwrite but to do append, we can use >>

$ cmd1 >> file
(appends stdout to end of the file)

Thus, &>> will do stdout + stderr go in that file. (ofc stdin can't go, cuz its in, so left are out and err)

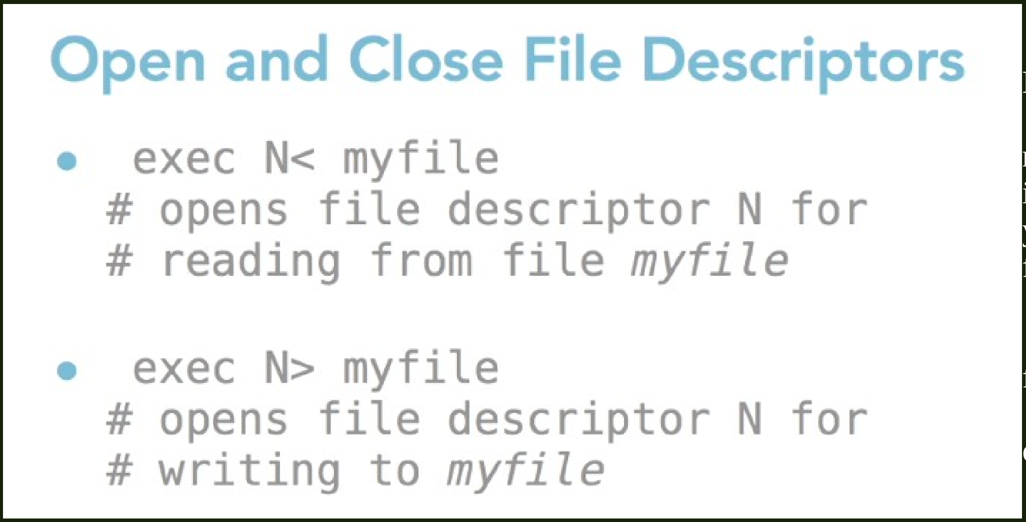


Here Documents: <<

- Here documents are a way to embed input for standard input inside of a script.
- They avoid having to create a new file just to hold some input values.

The << helps to sortof give input in a command, without actually needing a file to hold those inputs, which is needed in case of <.

```
$ cat << END
> hi
> hello
> how are you?
> END
hi
hello
how are you?
$ sort << END
> b
> c
> a
> d
> END
a
b
c
d
$
```



Open and Close File Descriptors

-
  ```
  exec N< myfile
  # opens file descriptor N for
  # reading from file myfile
  ```

-
  ```
  exec N> myfile
  # opens file descriptor N for
  # writing to myfile
  ```

N is number, not letter N.

Basically, having a number tagged file for input or output. Cool yes, kinda like opening a file with a variable.

To close these open files, we do:

```
$ exec N>&-
```
or
```
$ exec N<&-
```

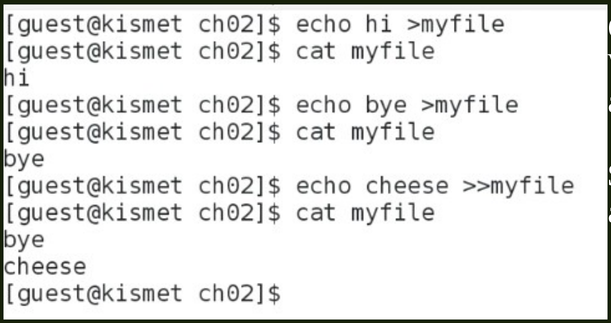
(files are not opened, neither closed, just association and disassociation)

Similarly, $ exec N<> myfile ofc takes input from myfile and writes to it.

Remember not to use 0 1 or 2 without knowledge, they’re stdin, stdout and stderr file descriptors. They’ll be reset once shell is ended then restarted.

Thus,

```
$ exec 7< coding_lessons.txt
$ lsof -p $$ | grep coding_lessons
COMMAND  PID  USER   FD   TYPE DEVICE SIZE/OFF     NODE NAME
bash    7423 vicro    7r   REG    8,2      500 17571310 /home/shalu/Desktop/coding_lessons.txt
```



```
[guest@kismet ch02]$ echo hi >myfile
[guest@kismet ch02]$ cat myfile
hi
[guest@kismet ch02]$ echo bye >myfile
[guest@kismet ch02]$ cat myfile
bye
[guest@kismet ch02]$ echo cheese >>myfile
[guest@kismet ch02]$ cat myfile
bye
cheese
[guest@kismet ch02]$
```

One lesson from work:
When echoing all the items, we want to echo erros also, so use this:

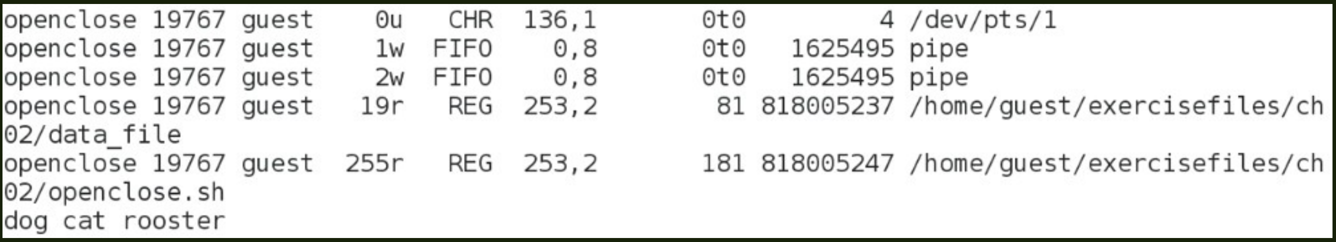
$ echo valuehere >&2 (output on file descriptor 2's address)

'find' command:

search for files in a directory hierarchy

The *find* program searches a directory tree to find a file or group of files. It traverses the directory tree and reports all occurrences of a file matching the user's specifications. The *find* program includes very powerful searching capability.

```
$ find /bin/py*
lists items /bin/py*
$ find shell -name *.sh  #lists item in shell dir having .sh ending
shell/basic.sh
shell/sps.sh
shell/cdecrypter.sh
shell/var.sh
```



```
openclose 19767 guest    0u   CHR  136,1       0t0         4 /dev/pts/1
openclose 19767 guest    1w  FIFO    0,8       0t0   1625495 pipe
openclose 19767 guest    2w  FIFO    0,8       0t0   1625495 pipe
openclose 19767 guest   19r   REG  253,2        81 818005237 /home/guest/exercisefiles/ch
02/data_file
openclose 19767 guest  255r   REG  253,2       181 818005247 /home/guest/exercisefiles/ch
02/openclose.sh
dog cat rooster
```

The lsof -p $$ gave this, and 19 was a file descriptor opened reading from data_file, and openclose.sh is the script, see, both has r mode, ie read from them.

Control-flow case statements and if-then-else with the test command:

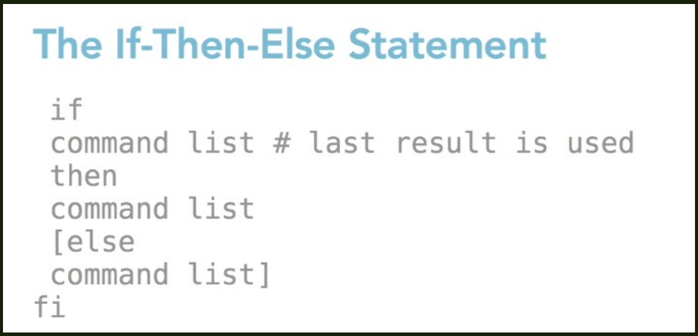
```
read -p "Please give yes or no:  " opinion

case $opinion in
y | Yes | YES | Y | yes)
    echo "Positive";;
n | no | No | NO)
    echo "Negative";;
*)
    echo "Neutral";;
esac

exit 0
```

This pipe split helps to have options

If we put n*, then nsofinwef will also give negative.
Also, if we do [nN] [oO] then No, nO, no, NO any comb will be working.

If else:

```
 $ if
> date -d "23001323" "+%s"
> echo hi
> then
> echo yess
> fi
date: invalid date ‘23001323’
hi
yess
```

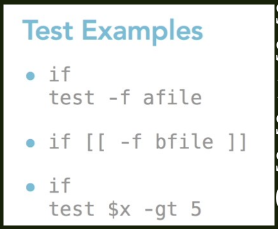


The If-Then-Else Statement

```
 if
 command list # last result is used
 then
 command list
 [else
 command list]
fi
```

The if block’s command list uses last command’s status, it was success, so element executed.

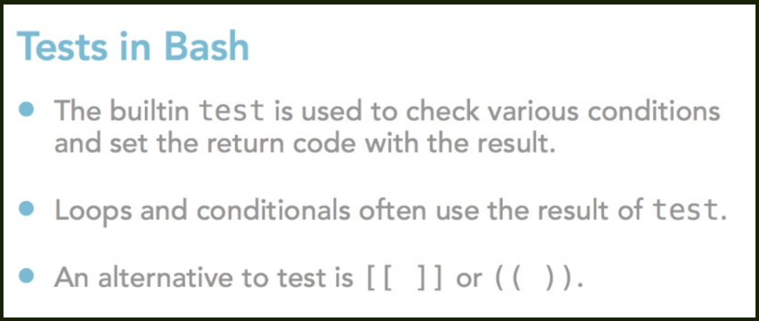
Test:



Test Examples

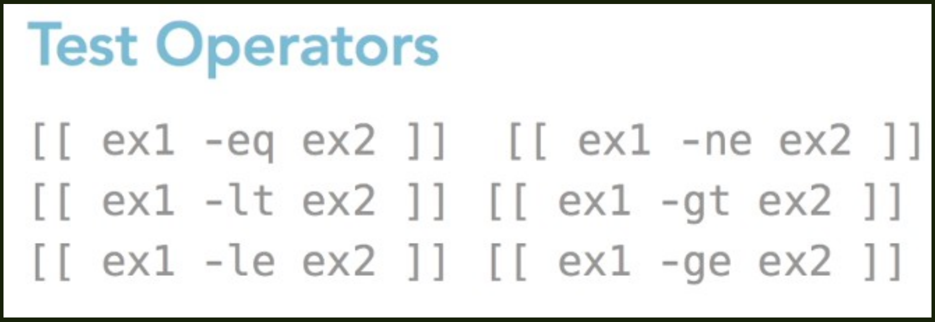
- `if`
  `test -f afile`
- `if [[ -f bfile ]]`
- `if`
  `test $x -gt 5`

```
$ test  2 -gt 4
$ echo $?
1
$ test  2 -ne 4
$ echo $?
0
```

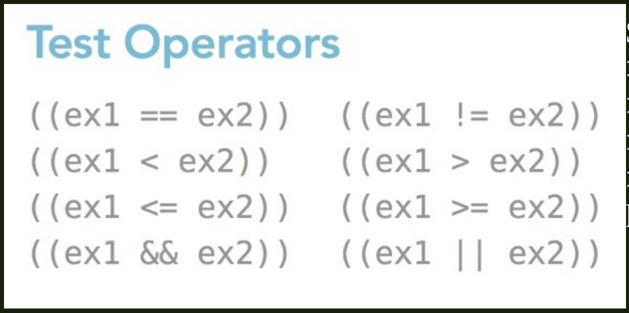


Tests in Bash

- The builtin `test` is used to check various conditions and set the return code with the result.
- Loops and conditionals often use the result of `test`.
- An alternative to test is `[[  ]]` or `((  ))`.


Test Operators

```
[[ ex1 -eq ex2 ]]  [[ ex1 -ne ex2 ]]
[[ ex1 -lt ex2 ]] [[ ex1 -gt ex2 ]]
[[ ex1 -le ex2 ]] [[ ex1 -ge ex2 ]]
```


Test Operators

```
((ex1 == ex2))   ((ex1 != ex2))
((ex1 < ex2))    ((ex1 > ex2))
((ex1 <= ex2))   ((ex1 >= ex2))
((ex1 && ex2))   ((ex1 || ex2))
```

```
$ if
> [[ 2 == 2 ]]
> then
> echo hi
> fi
hi
```

```
$ man test > mantest.txt
```

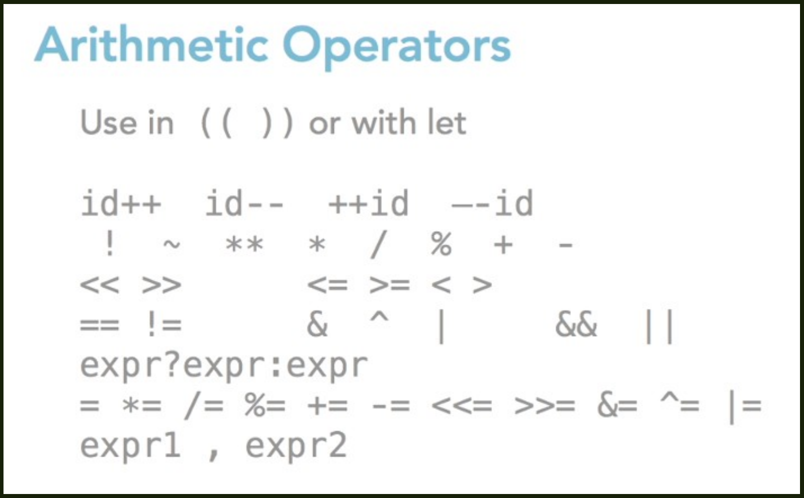
Helpful. There are many tests for a file:

```
    -b FILE
        FILE exists and is block special

    -c FILE
        FILE exists and is character special

    -d FILE
        FILE exists and is a directory

    -e FILE
        FILE exists

    -f FILE
        FILE exists and is a regular file

    -g FILE
        FILE exists and is set-group-ID

    -G FILE
        FILE exists and is owned by the effective group ID

    -h FILE
        FILE exists and is a symbolic link (same as -L)

    -k FILE
```

```
              FILE exists and has its sticky bit set

       -L FILE
              FILE exists and is a symbolic link (same as -h)

       -O FILE
              FILE exists and is owned by the effective user ID

       -p FILE
              FILE exists and is a named pipe

       -r FILE
              FILE exists and read permission is granted

       -s FILE
              FILE exists and has a size greater than zero

       -S FILE
              FILE exists and is a socket

       -t FD  file descriptor FD is opened on a terminal

       -u FILE
              FILE exists and its set-user-ID bit is set

       -w FILE
              FILE exists and write permission is granted

       -x FILE
              FILE exists and execute (or search) permission is granted
```

```
$ ls | while read var; do bob=$(test -x $var); echo file $var is executable $?; done | grep "executable 0"
file basic.sh is executable 0
file cdecrypter.sh is executable 0
file painter is executable 0
file testdoc.backup is executable 0
file var.sh is executable 0
$
```

Using arithmetic operators:



Arithmetic Operators

Use in (( )) or with let

```
id++  id--  ++id  --id
 !  ~  **  *  /  %  +  -
<< >>      <= >= < >
== !=      &  ^  |     &&  ||
expr?expr:expr
= *= /= %= += -= <<= >>= &= ^= |=
expr1 , expr2
```

((a=a+2)) is valid to initialize and assign a = 2 and also use it in loop to add 2 per loop.
Smartness that a is none when it doesn't have anything.

Part 3: USING FILTERS AND PARAMETER EXPANSION

Filter: Transformer/function/command that takes in and gives out.

Simple examples are ‘head’ and ‘tail’ to get head lines or tail lines of input file.

Example:

```
$ head -10  filename | tail -5 | wc -l
5
```

(this lists first 10 lines ,then select lines 6-10, and their line count is 5 so outputs five.

A file can change in live mode. tail is able to monitor that live.
Example:
```
$ tail -n2 -f output
```
This will print last 2 lines then will print again on each change.
(-f, --follow[={name|descriptor}]
output appended data as the file grows;)
So, if im using date command in a program to write seconds by sleeping too per second, the output file is growing each second, so if i read the file using this command, it’ll echo the changes in runtime.

Sed and AWK

stream editor.

Most basic and useful use is:
sed ‘s/old/new/’
And to do it for multiple patterns:
sed ‘s/old1/new1/; s/old2/new2/’ (the ‘/’ separater is important)
or
sed -e ‘s/b*.n/c.n/’ -e ‘s/[xX]/Y/’ (x or X will be replaced by Y and bblahblah.n will be c.n)

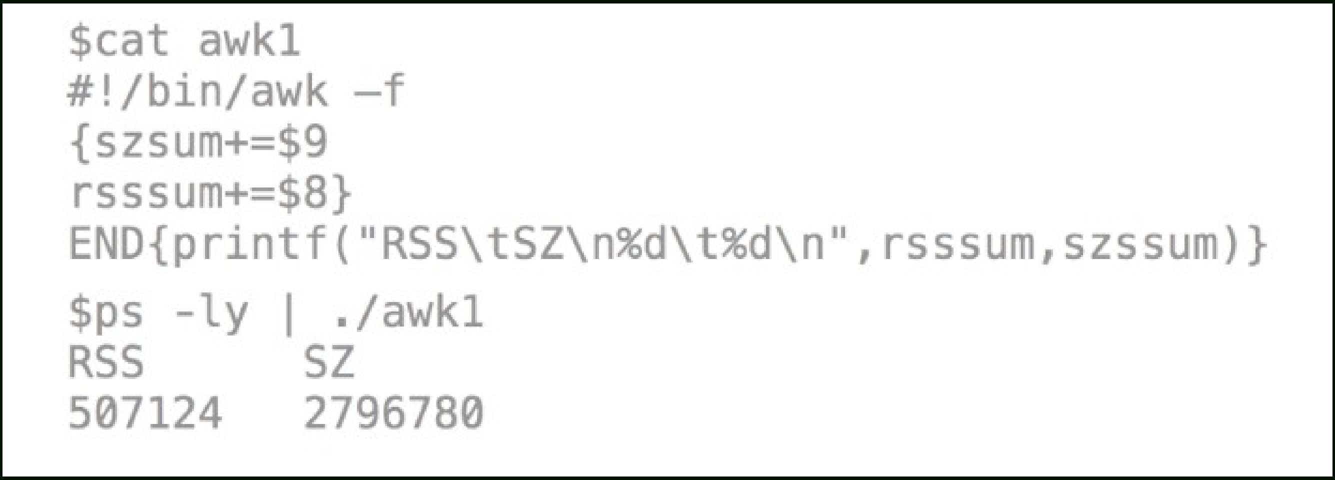
sed ‘/alpha/s/beta/gamma’ will replace beta with gamma in lines containing alpha.
sed ‘/apple/,/orange/d’ will delete the lines from apple line till orange line.

Awk loops on each line of the input file like sed. The field separater is stored with name FS which is typically the whitespace. (There is also NF which stands for number of field in input line)
$ ps -el | awk ‘/pts/{printf(‘hi\n’)}’ will print hi if it finds pattern ‘pts’ in a line. printf is C-like print function.
$1 $2 etc can be used to refer fields by index in a line.
Example:
```
$ ps -el | tail -4
1 I     0    5670       2  0  80   0 -     0 -      ?        00:00:00 kworker/u8:3-events_unbound
1 I     0    5730       2  0  80   0 -     0 -      ?        00:00:00 kworker/1:0-events
4 R  1000    5827    3923  0  80   0 -  2854 -      pts/0    00:00:00 ps
```

```
0 S  1000    5828    3923  0  80   0 -  2031 pipe_r pts/0    00:00:00 tail
```

Then:

```
$ ps -el | tail -4 | awk '/kworker/{printf("ID: %d\n",$4)}'
ID: 5730
```



```
$cat awk1
#!/bin/awk –f
{szsum+=$9
rsssum+=$8}
END{printf("RSS\tSZ\n%d\t%d\n",rsssum,szssum)}
$ps -ly | ./awk1
RSS      SZ
507124   2796780
```

awk program was specified in shebang.

When we do:

```
$ ps -ly | ./awk1
```

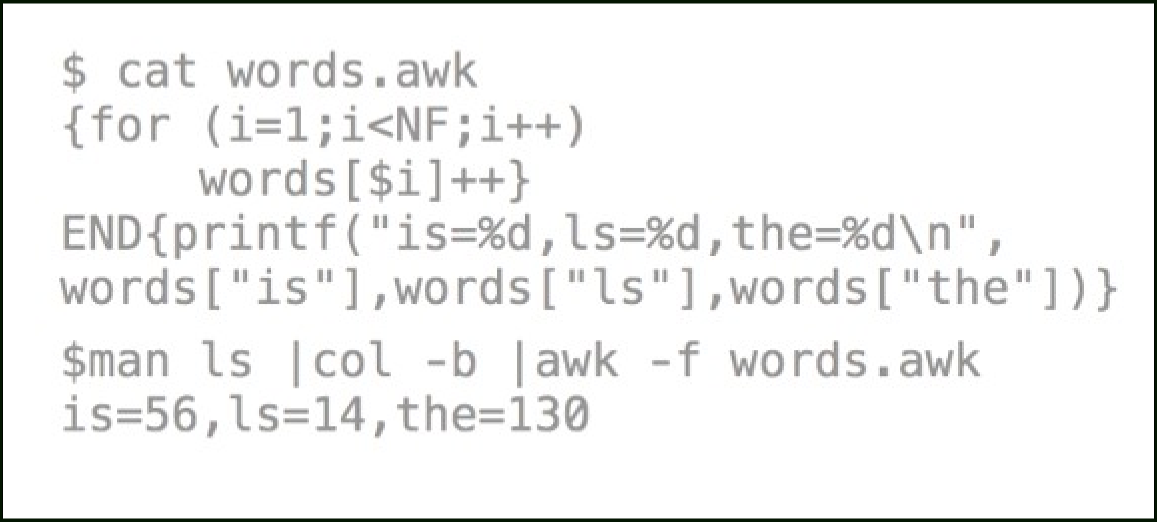
Then bash takes the output of ps -ly as input, and awk1 as parameter, and shebang /bin/awk -f as the processor. Thus actual command that gets executed is:

```
$ /bin/awk -f awk1 < $(ps -ly)
```

In general to understand faster:

```
$ shebang_processor filename
```

gets done.



```
$ cat words.awk
{for (i=1;i<NF;i++)
     words[$i]++}
END{printf("is=%d,ls=%d,the=%d\n",
words["is"],words["ls"],words["the"])}
$man ls |col -b |awk -f words.awk
is=56,ls=14,the=130
```

They echo the output once processed. They're filter. sed and awk have body, and that can be in a file too.

Thus, if you have sed code in a file called sed_code, then do it:

```
$ sed -f sed_code input_file
```

Parameters:

$0 – path of script itself.
$1 to $n – parameters.
For multidigits: ${192} like this.

Parameter motion in $1:

Use *shift* to move $2 to $1, $3 to $2 ...
This way, using shift multiple times, we can 'call' the parameters into $1.

Indirection:

```
$ x=abc;abc=def;echo ${!x}
def
```

Other examples:

```
$ x=abc;abc=def;def=ghi;echo ${!x}
def
$ x=abc;abc=def;def=ghi;echo ${!def}

$ x=abc;abc=def;def=ghi;echo ${!abc}
ghi
```

This tells that indirection talks to parso value.

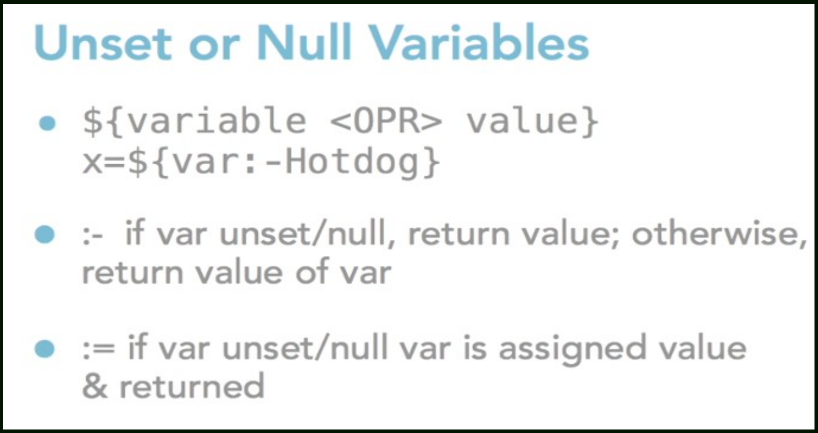
dict.get simile:

```
d = dict(...)
x = d.get(y,z) #z is default value if d doens't have y as key
```

In bash:

x=${variable:-default*Value*}

Its general form is:



Unset or Null Variables

- ${variable <OPR> value}
  x=${var:-Hotdog}
- :- if var unset/null, return value; otherwise, return value of var
- := if var unset/null var is assigned value & returned

Example:

```
$ b1=23
$ echo $a1 $b1
23
$ x=${a1:-23};y=${b1:-46};
echo $x $y
23 23
```

:-   If null/unset -> return val
:= If null/unset -> assign val to var and return val
:? If null/unset -> Display error and exit (example: $ ${unsetvar:?}
bash: unsetvar: parameter null or not set)
:+ If null/unset -> Return nothing, else return value (kinda like NOT of :-)

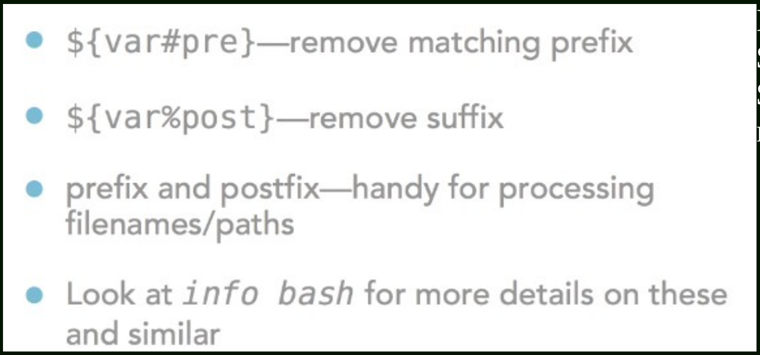
String operation:

For ${seq_var:offset/start_idx} or ${seq:offset/start_idx:len}

```
$ var=mynameissomething
$ echo ${var:3}
ameissomething
$ echo ${var:3:2}
am
$
```

length of a variable:

```
$ var=hithere
$ echo ${#var}
7
```

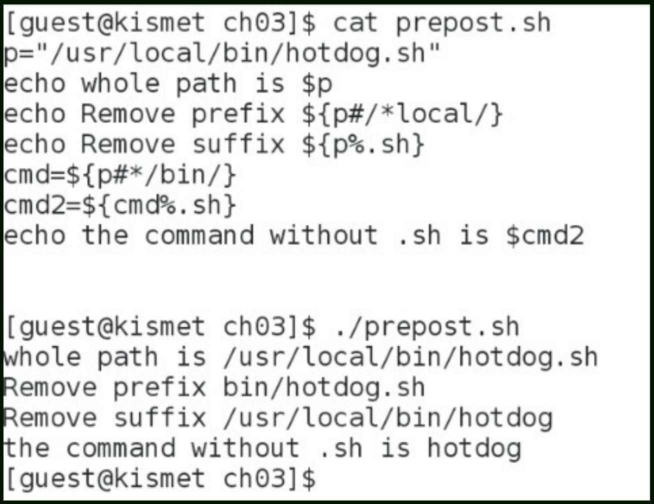


- ${var#pre}—remove matching prefix
- ${var%post}—remove suffix
- prefix and postfix—handy for processing filenames/paths
- Look at *info bash* for more details on these and similar

Example:

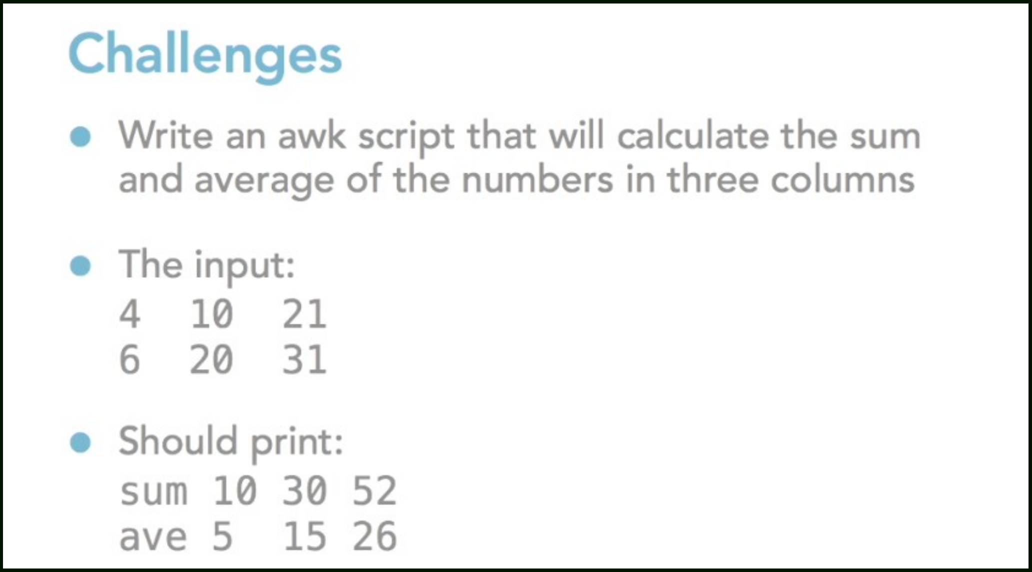
```
$ filenamevar=myfile.py
$ echo ${filenamevar%.py}
myfile
```

$ program {A..Z}
will make program get the params A B C... Z



```
[guest@kismet ch03]$ cat prepost.sh
p="/usr/local/bin/hotdog.sh"
echo whole path is $p
echo Remove prefix ${p#/*local/}
echo Remove suffix ${p%.sh}
cmd=${p#*/bin/}
cmd2=${cmd%.sh}
echo the command without .sh is $cmd2


[guest@kismet ch03]$ ./prepost.sh
whole path is /usr/local/bin/hotdog.sh
Remove prefix bin/hotdog.sh
Remove suffix /usr/local/bin/hotdog
the command without .sh is hotdog
[guest@kismet ch03]$
```

Challenge:



Challenges

- Write an awk script that will calculate the sum and average of the numbers in three columns
- The input:

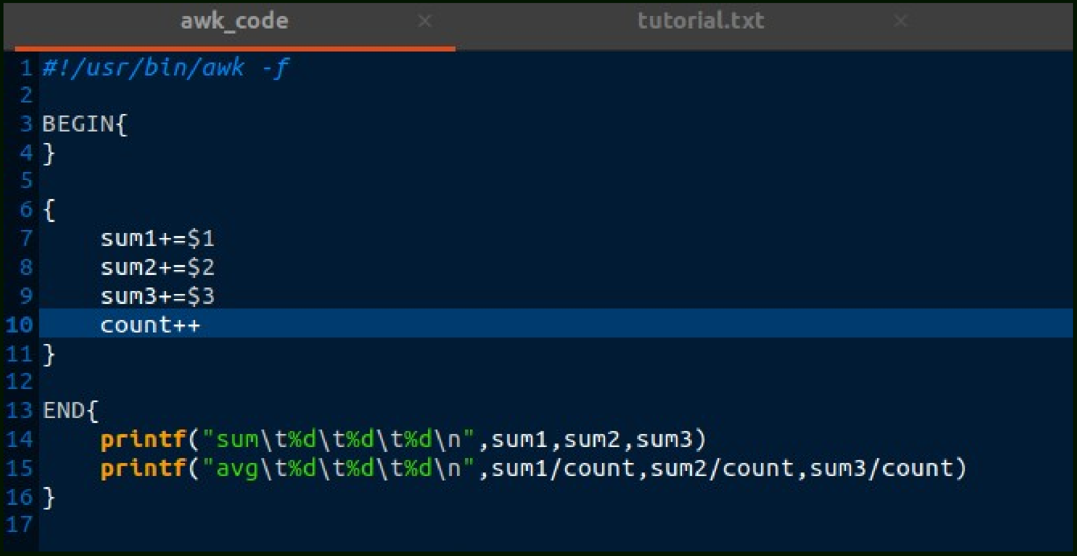
```
4  10  21
6  20  31
```

- Should print:

```
sum 10 30 52
ave 5  15 26
```

while read a b c and count increment can do the work and we can use <&0 also.

For awk, we know it loops on its input.
So, $ awk -f awkcode inp.txt > out.txt will do the work where awkcode is for one line.



```
#!/usr/bin/awk -f

BEGIN{
}

{
    sum1+=$1
    sum2+=$2
    sum3+=$3
    count++
}

END{
    printf("sum\t%d\t%d\t%d\n",sum1,sum2,sum3)
    printf("avg\t%d\t%d\t%d\n",sum1/count,sum2/count,sum3/count)
}

```

Res:

```
$ chmod 777 awk_code
$ ./awk_code < inp.txt
sum     10      30      52
avg     5       15      26
```

Part 4: Advanced Bash

<u>Using the coproc command</u>:

Coprocess is a process, it has its PID, like other processes.

We can see them using normal PID or $ jobs

When they are running, they have their file descripters of input and output in an array.

The default name of that array is COPROC, and we can set specific name if we want to.

And ofcourse we can kill them too.

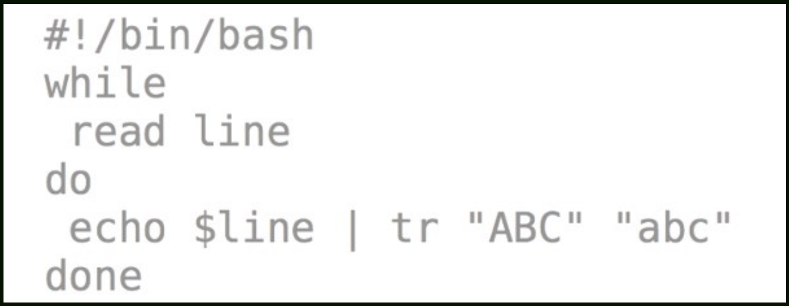
$ coproc ./myfile.sh

will create a coprocess where *filter*(a program having input and output, a transformer) is myfile.sh

We can then write input to input file descriptor, and fetch the resultant transformation from output file descriptor.

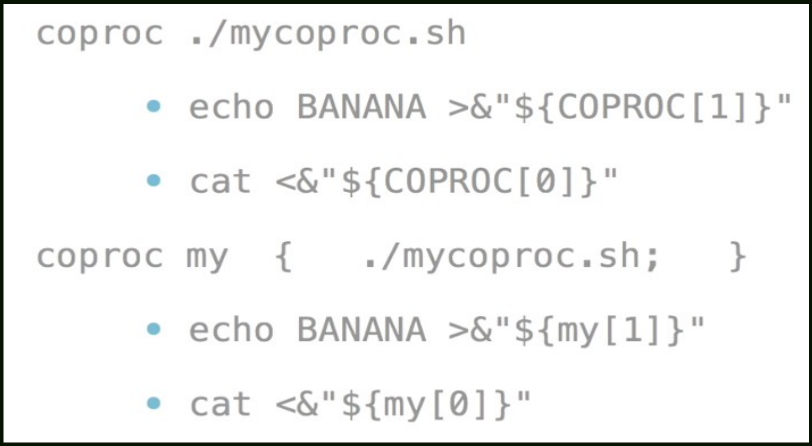
Example:

Here's mycoproc.sh



```
#!/bin/bash
while
 read line
do
 echo $line | tr "ABC" "abc"
done
```

The tr command is a map command, mapping A to a B to b and so on here...



```
coproc ./mycoproc.sh
```

- `echo BANANA >&"${COPROC[1]}"`
- `cat <&"${COPROC[0]}"`

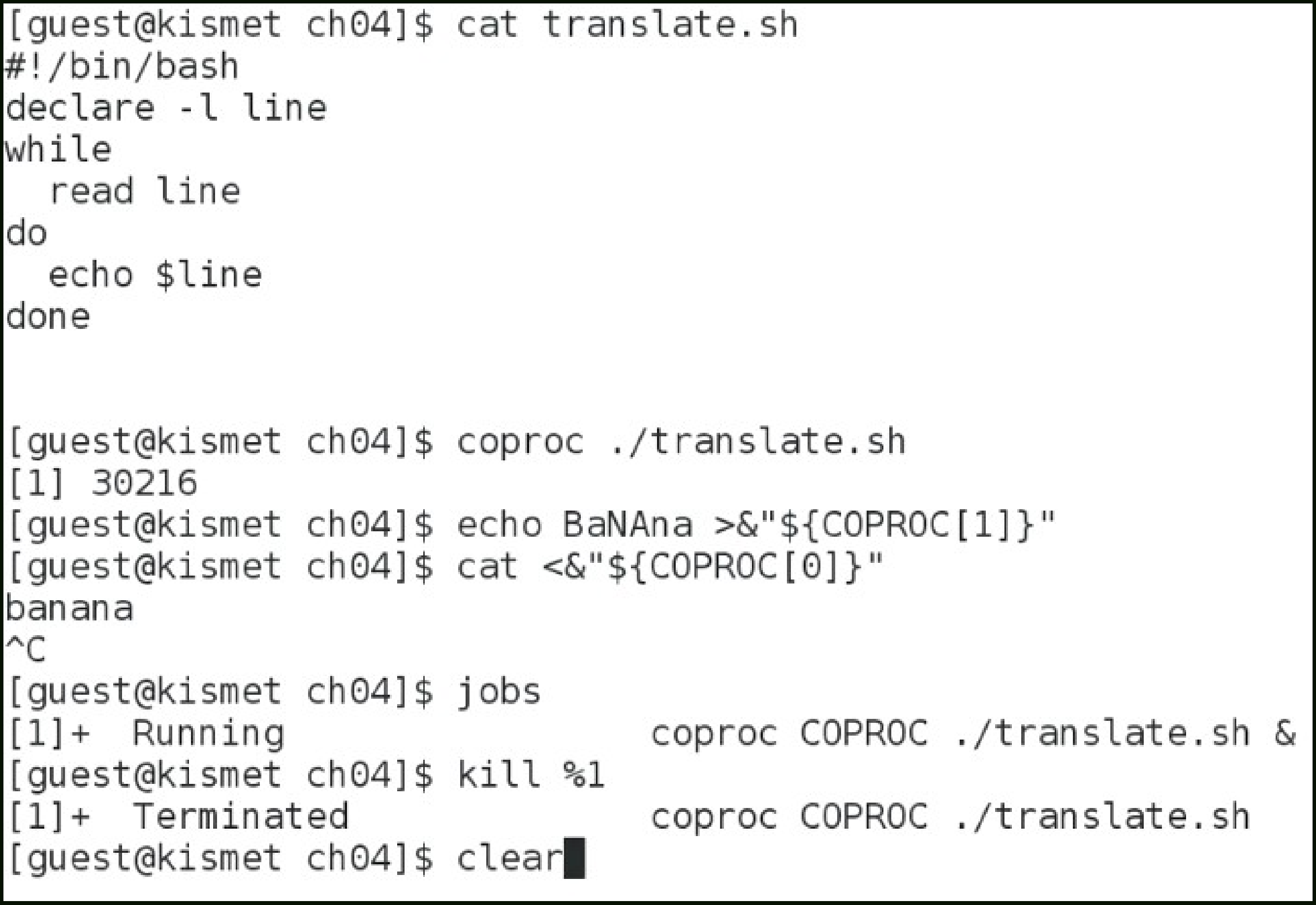
```
coproc my  {   ./mycoproc.sh;   }
```

- `echo BANANA >&"${my[1]}"`
- `cat <&"${my[0]}"`

##KEY POINT##

Key point is, input of coproc is not coproc[0], its coproc[1], design is such that we're only from the POV of the current bash. So, from bash we see whereever we write, is 1 numbered file descriptor, and where we read from is 0.

(SPACE is necessary between braces)



```
[guest@kismet ch04]$ cat translate.sh
#!/bin/bash
declare -l line
while
  read line
do
  echo $line
done


[guest@kismet ch04]$ coproc ./translate.sh
[1] 30216
[guest@kismet ch04]$ echo BaNAna >&"${COPROC[1]}"
[guest@kismet ch04]$ cat <&"${COPROC[0]}"
banana
^C
[guest@kismet ch04]$ jobs
[1]+  Running                 coproc COPROC ./translate.sh &
[guest@kismet ch04]$ kill %1
[1]+  Terminated              coproc COPROC ./translate.sh
[guest@kismet ch04]$ clear
```

The ‘declare -l line’ was the reason why ‘line’ is always going to be lowercase.

We can test this loop thing very simply via:

```
$ declare -l line; while read line;do echo $line; done << EOF
> GreEN
> TREe
> EOF
green
tree
```

Debugging Scripts using -x and -u:

```
$ bash -x script.sh
```

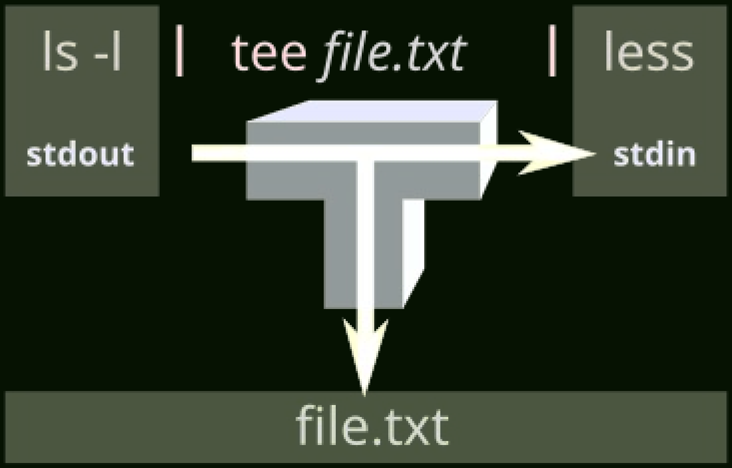
will echo commands.

And if we do ‘set -x’ inside script itself and close with ‘set +x’, then it will echo commands in between these.

Since bash has a common bug of not throwing unset variable errors, we can use:

set -u or bash -u to throw errors on use of such variables.

Tee command: Copies input to output as well as some other files. Can be used for debugging. And it is often used in conjuction with filters and pipe.



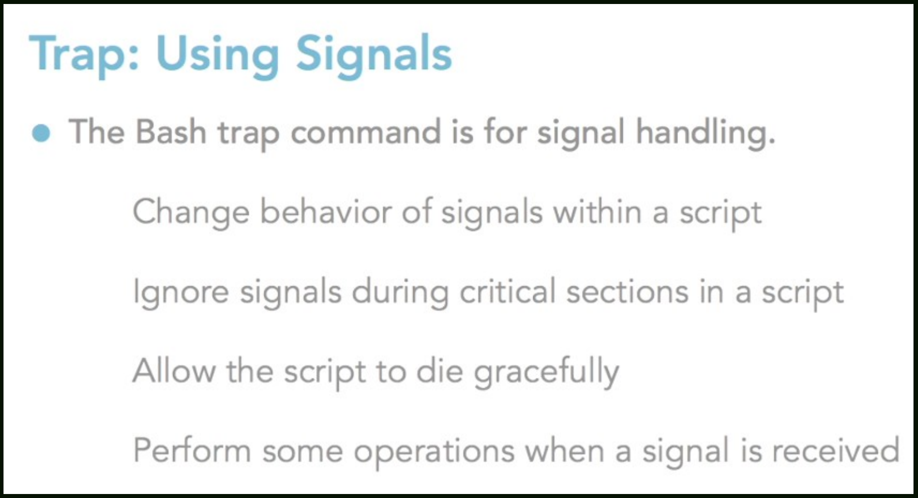
Example usage of tee: The output of ls -l is redirected to tee which copies them to the file file.txt and to the pager less. The name tee comes from this scheme - it looks like the capital letter T

$ ls -l | tee ls_l_outp.txt | less

Use tee -a to append.

<u>Signals and traps:</u>

Sense signals and do something.



Trap: Using Signals

- The Bash trap command is for signal handling.

    Change behavior of signals within a script

    Ignore signals during critical sections in a script

    Allow the script to die gracefully

    Perform some operations when a signal is received

Getting the list of signals: $ trap -l or $ kill -l

```
$ trap -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

trap: trap [-lp] [[arg] signal_spec ...]

-Trap signals and other events.

-Defines and activates handlers to be run when the shell receives signals or other conditions.

SIGQUIT is triggered when we do Ctrl + \ (backslash)

```
$ trap 'echo hello' QUIT
$ ^\hello
```

(entered space)

```
$
$ ^\hello
^\hello
^\hello
```

(entered space)

```
$ ^\hello
^\hello
^C
$
```

The Ctrl + C sends SIGINT.

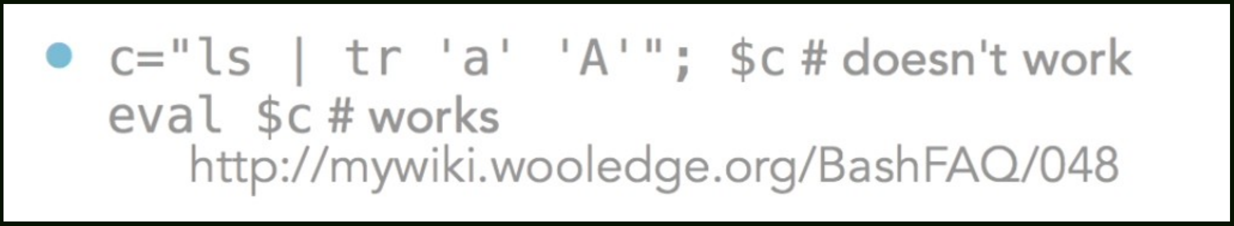
How It Works:

1. **Before the Handler Starts:**
    - When a signal (e.g., `SIGINT` from `Ctrl+C`) is received, the shell **immediately stops executing the current command**.
    - The shell then checks if there's a `trap` set for that signal. If so, it **executes the trap handler**.
2. **During Trap Handling:**
    - The trap handler runs **in the same shell process** that received the signal.
    - **No new process is spawned**; it's just a part of the current script or shell session.
3. **After the Handler Finishes:**
    - Once the trap handler finishes executing, **the shell continues running the script from the point where it was interrupted**, if applicable.
    - For signals like `SIGINT`, if the handler doesn't explicitly exit or kill the script, execution resumes after the interrupted command.
    - For `EXIT`, the trap runs **when the script exits**, whether due to normal completion, `exit`, or receiving a termination signal.

Special Cases:

- **Signals like `SIGKILL` (9) and `SIGSTOP` (19)** cannot be trapped, ignored, or handled.
- **Using `trap '' SIGNAL`**: This clears the trap for the specified signal.

Using the eval and getopt commands:



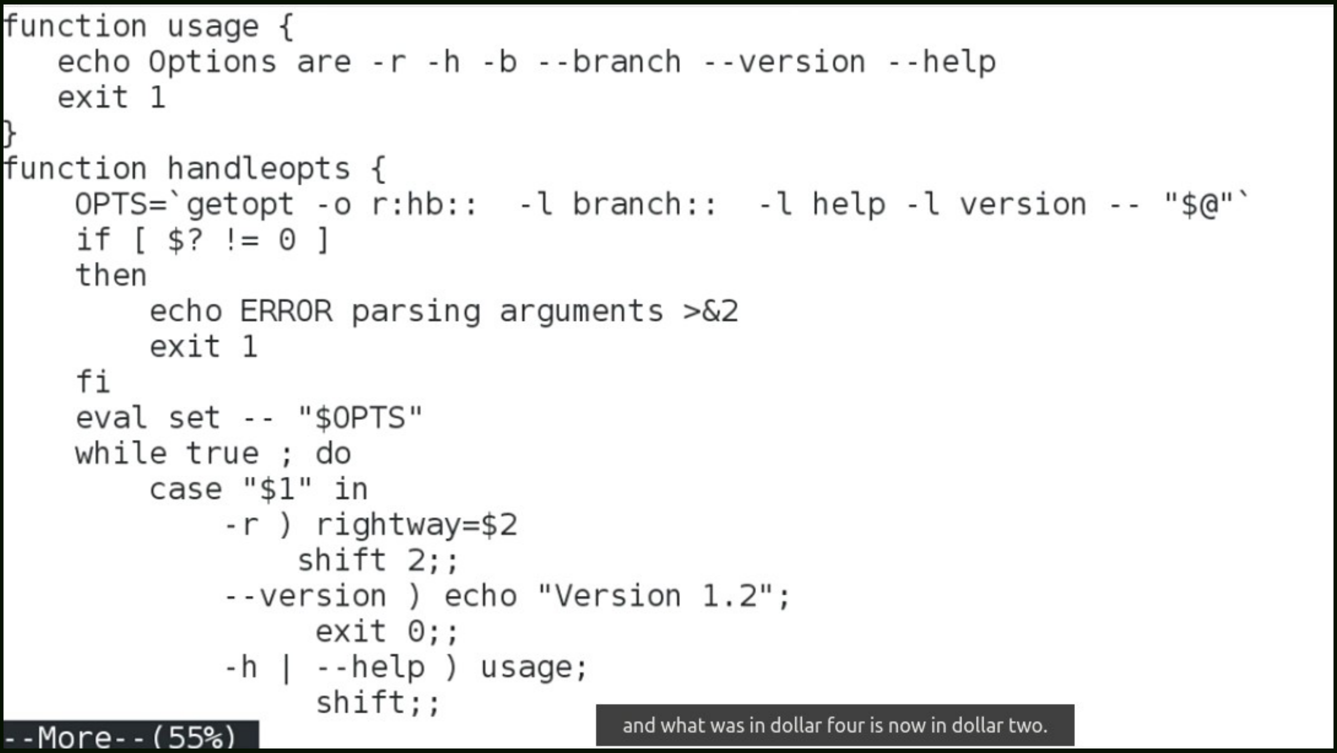
- `c="ls | tr 'a' 'A'"; $c` # doesn't work
  `eval $c` # works
  http://mywiki.wooledge.org/BashFAQ/048

It evaluates, and while evaluating, it runs the commands if there's any.

getopt is different than getopts, but similar.

The -- separates options etc from arguments.

This '--' is available for both 'set' and 'getopt'



```
function usage {
   echo Options are -r -h -b --branch --version --help
   exit 1
}
function handleopts {
    OPTS=`getopt -o r:hb::  -l branch::  -l help -l version -- "$@"`
    if [ $? != 0 ]
    then
        echo ERROR parsing arguments >&2
        exit 1
    fi
    eval set -- "$OPTS"
    while true ; do
        case "$1" in
            -r ) rightway=$2
                shift 2;;
            --version ) echo "Version 1.2";
                 exit 0;;
            -h | --help ) usage;
                 shift;;
--More--(55%)
```

and what was in dollar four is now in dollar two.

Basically, getopt was given a command, options, long options etc and then they give argument $@.

That was executed and stored in OPTS, and then set -- "$OPTS" was used to set the values, which I think are $1 $2 etc and their corresponding values, and then we evaluated them.

Few questions arise, like why did we need to set things? Coundn't just running that command set the $1 $2, and are they actually setting $1 and $2, and why did we need to eval them?

Answer to these questions is visible when we see three behaviors:

First behavior

```
$ set help
set: set [-abefhkmnptuvxBCHP] [-o option-name] [--] [arg ...]
    Set or unset values of shell options and positional parameters.
    ....
$ cat script.sh
echo $1 $2
eval set -- "-o hi"
echo $1 $2
exit 0
$ script.sh pie apple
pie apple
-o hi
---------
```

Second behavior:

```
$ cat var.sh
echo first: $1
echo second: $2
set -- "-o hi"
echo first: $1
echo second: $2
eval set -- "-o hi"
echo first: $1
echo second: $2
exit 0
$ ./var.sh pie apple
first: pie
second: apple
```

first: -o hi

second:

first: -o

second: hi

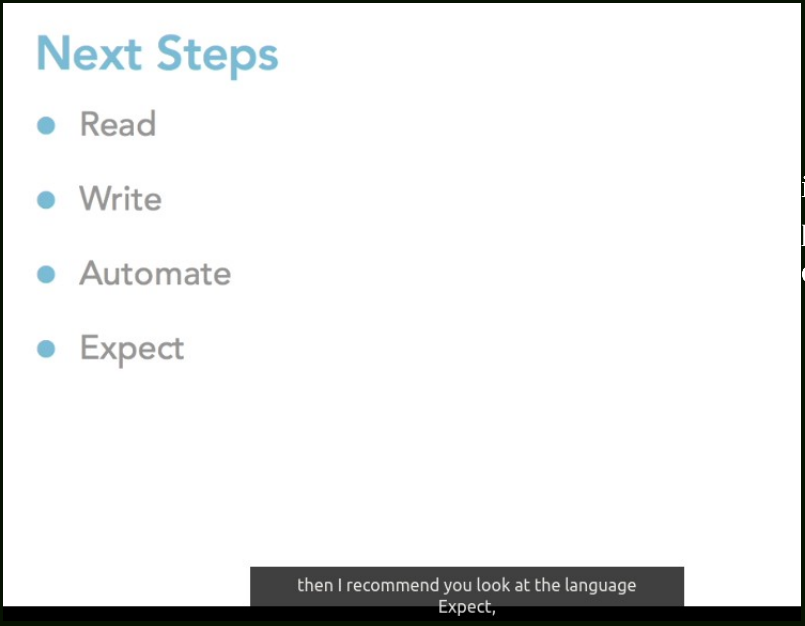
Third behavior

bash scriptname.sh -oTrue

This is valid when we parse the options using getopts and then set them again, instead of directly looping and checking $1 and $2 because in later case, the “-oTrue” is a single argument

Example:

```
$ cat script.sh
OPTS=$(getopt -o o: -l override -- "$@")
echo $OPTS , $1, $2, $3

eval set -- $OPTS
echo $OPTS , $1, $2, $3
exit 0

$ bash script.sh -oTrue
-o 'True' -- , -oTrue, ,
-o 'True' -- , -o, True, --
```



Next Steps

- Read
- Write
- Automate
- Expect

then I recommend you look at the language Expect,

(Expect is used to automate software interactions like say SSH login process, but its more than this, a little deeper).

Why is it called PUTTY:

TTY (tele typewriter) (tty) is a command used, which tells if output medium is terminal or not.

Pseudo terminals (pseudotty) are simulated terminals that acts like terminals. A pseudo-terminal is **a special interprocess communication channel that acts like a terminal**. (from gnu.org) One end of the channel is called the master side or master pseudo-terminal device, the other side is called the slave side. So yes, this is why I think its called PUTTY.

Btw Putty interface is actually SSH client, more of UI work, so its terminal emulator, and it uses pseudoterminal thing (master and slave).

A good bash intro:
https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Introduction

Notes from video: https://youtu.be/GtovwKDemnI

```
#------------------------------- COMMAND LINE UTILITIES -----------

-Create a file and write (in-shell editor):

nano filename.txt


-Some very basic commands and explaination:

ls    (lists) (ls -d to list dirs)
        ls -d */ (the */ gives a pattern to match all names who has / in them)
        Desktop/    Downloads/  Pictures/  Templates/
        Documents/  Music/      Public/    Videos/
      (other method is: ls -l | grep ^d)
        (d starting lines are of directory when detailed lines are printed for
listed items)
      ( other methods are t for sort by modification time, and r for reversing the
list,
      so, $ ls -tr will give oldest to newest modified files)

cd    (changes dir, cd .. to parent dir, cd - for past dir, cd ~ for home)
pwd   (print working directory)
mkdir (mkdir)
mv    (move)
rm    (remove)
cp    (copy)
date  (date -d "now" +"+%s") (to get current time in seconds form epoch jan 1 1970)
echo  (echo)
cat   ( “concatenate”. Primarily used to read, display, and concatenate(display
together) text files.)
less  (cat but in another separate in-window (exitable by q) like help does, for
cleanliness and separation)
       (a terminal pager program used to view the contents of a text file one
screen at a time)
grep  (used searching plaintext *datasets* for lines that match a regular
expression.
        Its name comes from the ed command g/re/p (global regular expression search
and print) )
       (example: $ grep -r 'command' .     will search for word 'command' in all
files in current dir recursively)
       (example: $ ls | grep '.txt'  will list all filles with .txt in their names)
touch (to create an empty file or update the timestamp of an existing file.)
chmod ( +x for making it executable, +r or +w for read and write permissions)
      ( chmod [options] mode (user-group-others) filename/dir     this is basic
syntax)
      (example: chmod u+x file.txt  # Adds execute permission for the user only)
      (group is a group of users)
      ( 4,2,1 for read, write, execute )
      ( 0 for nothing )
      ( 3 in binary goes 0+2+1, so write+execute)
      ( 5 in binary goes 4+0+1, so read+execute)
      ( 6 in binary goes 4+2+0, so read+write)
      ( 7 in binary goes 4+2+1, so rwx)
      (so $ chmod 777 file.sh will grant all rwx to all ugo)
```

```
man   (for getting manual, like: $ man python3)
ps # processes’ snapshot

    $ ps -e   # enlist all processes for all users

    $ ps -f  # full details of the processes

    $ ps -ef is therefore used

    $ ps aux # show all processes with BSD-style Output

    a – all users
    u – show processes with user oriented details
    x – include processes not attached to a terminal.

    $ ps -p <PID>   # for listing the processes with that PID



Piping: a | b

shebang    #!/usr/bin/env bash (flexible pointing to bash)
            (do not be fooled by scripts or examples on the Internet that use
/bin/sh as the interpreter. sh is not bash! Bash itself is a "sh-compatible" shell
(meaning that it can run most 'sh' scripts and carries much of the same syntax)
however, the opposite is not true; some features of Bash will break or cause
unexpected behavior in sh. )
Bash: Bash, short for Bourne-Again SHell

Bourne shell vs C shell (two main shell types):
    Bourne shell types:
        Bourne shell (sh), Korn Shell (ksh), Bourne Again SHell(bash), Posix shell
    C shell types (The C Shell has a C-like syntax, which makes it more appealing
to programmers familiar with C):
        C shell (csh), Z shell(zsh), Tenex/Tops C shell

#---------------------------------

Basic shell scripting


Comment: '#'

Similar of 'input' of python:
read input: $ read VAR_NAME_TO_STORE_VALUE_TO
readonly VAR will not allow VAR to be changed.

Assigning:
a=b   #no space

Argument taking:
$0  : script name
$1-n: args
$#  : number of args
$*  : All positional parameters as a single string
$@  : All positional parameters (arguments) as separate words
        (If you enclose $@ in double quotes ("$@"), each argument remains as a
separate entity even if it contains spaces. This is a common approach when passing
arguments to another command.)
```

```
$?  : exit status of the last command
$$  : Process ID (PID) of the current shell
$!  : ($!) Expands to the process ID of the job most recently placed into the
background, whether executed as an asynchronous command or using the bg builtin

$ python3 "var" "var2" means there are 2 parameters passed.


Single quoting characters preserves literal value of each character,
double quoting also does, but except for some characters like $, '`','"','\' or
newline.
$ echo "hi $var"
hi 2
$ echo 'hi $var'
hi $var
Since they preserve literal value, thus, unlike python, a=5;if [[ "$a" -le
10 ]];then echo 'hi'; fi will say 'hi' as "$a" is not a string.


expr: expr is a command used for evaluating expressions.
    In newer versions of shells, expr is somewhat outdated, and shell built-in
operators or modern tools like (( )), let, or bc are often used instead.

backticks: Backticks, also called backquotes "`", are used in shell scripting to
execute commands within another command.
    Example: $ echo "The date is `date`"
    Better readability is achieved by: "The date is $(date)" (using subshell)
From bash docs:
    ( expr )

    Returns the value of expr. This may be used to override the normal precedence
of operators.




#------------------------------

Basic operators

    Arithmetic:
        +,-,*,/,%,==,=, != (all like python)

        Examples:
            $a-$b, [ $a==$b ] etc.

    Relational:
        Equality: -eq, -ne,
        Greater/less: -gt, -lt, -ge, -le

        Example:
            [ $a -ge $b ]

    Boolean:
        !, -o, -a  (not, or, and)
        [ !false expression here ] returns true

    String opeartor:
        equality: =,!=
        zero or non-zero length: -z, -n
        Like python, it returns false if we check boolean value of empty string
[$a] is false if a is 0 len str

    File test operator:
        -b file, -d file, ... like  -c,-f,-g,-k,-p,-t,-u,-r,-w,-x,-s,-e file

        Example:
            [ -e filename ] will give true if file exists

#---------------------------------
```

Note that wherever a ‘;’ appears in the description of a command’s syntax, it may be replaced with one or more newlines.

Shell loops

For loop:

```
    for token in iterable
    do
    ...
    done

        Example:

            for token in $*
            do
                echo $tokan
            done

            for var in 1 2 3 4 5
            do
                echo $var
            done
```

An alternate form of the for command is also supported:

```
    for (( expr1 ; expr2 ; expr3 )) ; do commands ; done
```

While loop:

```
    while
```

The syntax of the while command is:

```
    while test-commands; do consequent-commands; done

    Example:
        counter=5

        while [ $counter -ge 1 ]
        do
         echo $counter
         counter=`expr $counter - 1`
         #or could have done $((counter-1)), learning is, $expr evaluates expr
value
        done
    Important point:
        The syntax therefore tells us, that it works like while the test commands
end successfully, do this that.
        Thus, "while getopts $OPTSTRING optvar; do ... ; done"
        is fine, as getopts only gives exit status 1 when no more options are left
to be parsed.


until

    The syntax of the until command is:

    until test-commands; do consequent-commands; done


There are 'break' and 'continue' words too, like python


#--------------------------------

Conditional context:

if

    The syntax of the if command is:

    if test-commands; then
      consequent-commands;
    [elif more-test-commands; then
      more-consequents;]
    [else alternate-consequents;]
    fi


case

    The syntax of the case command is:

    case word in
        [ [(] pattern [| pattern]…) command-list ;;]…
    esac

    Example:
        echo -n "Enter the name of an animal: "
        read ANIMAL
        echo -n "The $ANIMAL has "
        case $ANIMAL in
          horse | dog | cat) echo -n "four";;
          man | kangaroo ) echo -n "two";;
          *) echo -n "an unknown number of";;
        esac
        echo " legs."

select

    The select construct allows the easy generation of menus. It has almost the
same syntax as the for command:
    select name [in words …]; do commands; done
```

#-----------------------------------

Functions:
    Functions are declared using this syntax:

```
        fname () compound-command [ redirections ]
```

    or

```
        function fname [()] compound-command [ redirections ]
```

    Example:

```
        $ function foo() { echo 'hi'; }
        $ foo
        hi
```

    Example with args:

```
        $ Foo() {
        > echo $1 $2
        > }
        $ Foo

        $ Foo hi there
        hi there
```

    Example with other type of compounding:

```
        $ func () ( echo "hi" )
        $ func
        hi
```

The FUNCNEST variable, if set to a numeric value greater than 0, defines a maximum function nesting level.

Variables local to the function may be declared with the local builtin (local variables).

For example, if a variable var is declared as local in function func1, and func1 calls another function func2, references to var made from within func2 will resolve to the local variable var from func1, shadowing any global variable named var.
(this tells that functions if called inside another one, will be wholly contained inside the local scope of parent function, so even variables are affected yk, unlike python)

    Example:

```
        func1()
        {
            local var='func1 local'
            func2
        }

        func2()
        {
            echo "In func2, var = $var"
        }

        var=global
        func1

    Outp:
        In func2, var = func1 local
```

We have 'return' also, can return the value. We can capture the return using $?

#------------------------------

Grouping commands:

()

```
    ( list )
```

    Placing a list of commands between parentheses forces the shell to create a subshell (see Command Execution Environment), and each of the commands in list is executed in that subshell environment. Since the list is executed in a subshell, variable assignments do not remain in effect after the subshell completes.
{}

```
    { list; }
```

    Placing a list of commands between curly braces causes the list to be executed in the current shell context. No subshell is created. The semicolon (or newline) following list is required.

The braces are reserved words, so they must be separated from the list by blanks or other shell metacharacters.

#---------------------------

(from $ man bash)

getopts

```
getopts optstring name [args]
              getopts  is used by shell procedures to parse positional parame-
              ters.  optstring contains the option  characters  to  be  recog-
              nized;  if a character is followed by a colon, the option is ex-
              pected to have an argument, which should be separated from it by
              white  space.  The colon and question mark characters may not be
              used as option characters.  Each time  it  is  invoked,  getopts
              places  the next option in the shell variable name, initializing
              name if it does not exist, and the index of the next argument to
              be processed into the variable OPTIND.  OPTIND is initialized to
              1 each time the shell or a shell script is invoked.  When an op-
              tion requires an argument, getopts places that argument into the
              variable OPTARG.  The shell does not reset OPTIND automatically;
              it  must  be  manually  reset  between multiple calls to getopts
              within the same shell invocation if a new set of  parameters  is
              to be used.
              When the end of options is encountered, getopts exits with a re-
              turn value greater than zero.  OPTIND is set to the index of the
              first non-option argument, and name is set to ?.

              getopts  normally  parses the positional parameters, but if more
              arguments are given in args, getopts parses those instead.

              getopts can report errors in two ways.  If the  first  character
              of  optstring  is  a  colon, silent error reporting is used.  In
              normal operation, diagnostic messages are printed  when  invalid
              options  or  missing  option  arguments are encountered.  If the
              variable OPTERR is set to 0, no  error  messages  will  be  dis-
              played, even if the first character of optstring is not a colon.

              If an invalid option is seen, getopts places ? into name and, if
              not silent, prints an  error  message  and  unsets  OPTARG.   If
              getopts  is  silent, the option character found is placed in OP-
              TARG and no diagnostic message is printed.

              If a required argument is not found, and getopts is not  silent,
              a  question  mark  (?) is placed in name, OPTARG is unset, and a
              diagnostic message is printed.  If getopts  is  silent,  then  a
              colon  (:)  is  placed  in  name and OPTARG is set to the option
              character found.

              getopts returns true if an option, specified or unspecified,  is
              found.  It returns false if the end of options is encountered or
              an error occurs.

Example:

$ getopts "p:h:" current_option_face
will read the first option if its there, and if its not valid it'll through error.
Then again executing
$ getopts "p:h:" current_option_face
will check second option and so on...

    #  h - check for option -h without parameters; gives error on unsupported
options;

    #  h: - check for option -h with parameter; gives errors on unsupported
options;

    # abc - check for options -a, -b, -c; gives errors on unsupported options;

    # :abc - check for options -a, -b, -c; silences errors on unsupported options;

    # Notes: In other words, colon in front of options allows you handle the errors
in your code. Variable will contain ? in the case of unsupported option, : in the
case of missing value.

    #  OPTARG - is set to current argument value,

    # OPTERR - indicates if Bash should display error messages.

Thus, example:
    #
    # while getopts "p:h:" current_option_face; do
    #   echo "$current_option_face, $OPTARG"
    # done
    #
    # then we get:
    #
    # $ ./var.sh -p hival -h hival2
    # p, hival
    # h, hival2
    #
```

Shell variables:

OPTARG

    The value of the last option argument processed by the getopts builtin.
OPTIND

    The index of the last option argument processed by the getopts builtin.

PS1

    The primary prompt string. The default value is ‘\s-\v\$ ’. See Controlling the Prompt, for the complete list of escape sequences that are expanded before PS1 is displayed.
PS2

    The secondary prompt string. The default value is ‘> ’. PS2 is expanded in the same way as PS1 before being displayed.
...

#-------------------------

shift

    shift [n]

    Shift the positional parameters to the left by n. The positional parameters from n+1 … $# are renamed to $1 … $#-n. Parameters represented by the numbers $# down to $#-n+1 are unset. n must be a non-negative number less than or equal to $#. If n is zero or greater than $#, the positional parameters are not changed. If n is not supplied, it is assumed to be 1. The return status is zero unless n is greater than $# or less than zero, non-zero otherwise.

    Therefore,
    if getopts is used, OPTIND - 1 denotes last option flag's presence' index (either flag name or val)
    Thus
    shift $((OPTIND - 1)) makes OPTINT - 1 + 1 as the base
    if getopts is not used, we want 1,2,3 to be 1,2,3, and by default OPTIND is 1, so OPTIND - 1=1-1=0
    => new idxs are 0, 1-0, 2-0, 3-0,... works for this case too.

#-----------------------

User and group commands:

    adduser: add a user to the system.

    userdel: delete a user account and related files.

    addgroup: add a group to the system.

    delgroup: remove a group from the system.

    usermod: modify a user account.

    change: change user password expiry information.

```
#---------------
```

((a=a+2)) is valid to assign a = 2 and also use it in loop to add 2 per loop.
Smartness that a is none when it doesn't have anything.

How I united pdf:

```
$ ls /bin/pdf*
/bin/pdf2dsc    /bin/pdffonts     /bin/pdfsig      /bin/pdftops
/bin/pdf2ps     /bin/pdfimages    /bin/pdftocairo  /bin/pdftotext
/bin/pdfattach  /bin/pdfinfo      /bin/pdftohtml   /bin/pdfunite
/bin/pdfdetach  /bin/pdfseparate  /bin/pdftoppm
$ pdfunite p*.pdf out.pdf
```