If Statements

(*EXPRESSION* **)**	*EXPRESSION* is true
! *EXPRESSION*	*EXPRESSION* is false
EXPRESSION1 **-a** *EXPRESSION2*	both *EXPRESSION1* and *EXPRESSION2* are true
EXPRESSION1 **-o** *EXPRESSION2*	either *EXPRESSION1* or *EXPRESSION2* is true
-n *STRING*	the length of *STRING* is nonzero
STRING	equivalent to **-n** *STRING*
-z *STRING*	the length of *STRING* is zero
STRING1 **=** *STRING2*	the strings are equal
STRING1 **!=** *STRING2*	the strings are not equal
INTEGER1 **-eq** *INTEGER2*	*INTEGER1* is equal to *INTEGER2*
INTEGER1 **-ge** *INTEGER2*	*INTEGER1* is greater than or equal to *INTEGER2*
INTEGER1 **-gt** *INTEGER2*	*INTEGER1* is greater than *INTEGER2*
INTEGER1 **-le** *INTEGER2*	*INTEGER1* is less than or equal to *INTEGER2*
INTEGER1 **-lt** *INTEGER2*	*INTEGER1* is less than *INTEGER2*
INTEGER1 **-ne** *INTEGER2*	*INTEGER1* is not equal to *INTEGER2*
FILE1 **-ef** *FILE2*	*FILE1* and *FILE2* have the same device and inode numbers
FILE1 **-nt** *FILE2*	*FILE1* is newer (modification date) than *FILE2*
FILE1 **-ot** *FILE2*	*FILE1* is older than *FILE2*
-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

Test, [and [[

Feature	new test [[	old test [	Example
string comparison	>	\> (*)	`[[ a > b ]] || echo "a does not come after b"`
	<	\< (*)	`[[ az < za ]] && echo "az comes before za"`
	= (or ==)	=	`[[ a = a ]] && echo "a equals a"`
	!=	!=	`[[ a != b ]] && echo "a is not equal to b"`
integer comparison	-gt	-gt	`[[ 5 -gt 10 ]] || echo "5 is not bigger than 10"`
	-lt	-lt	`[[ 8 -lt 9 ]] && echo "8 is less than 9"`
	-ge	-ge	`[[ 3 -ge 3 ]] && echo "3 is greater than or equal to 3"`
	-le	-le	`[[ 3 -le 8 ]] && echo "3 is less than or equal to 8"`
	-eq	-eq	`[[ 5 -eq 05 ]] && echo "5 equals 05"`
	-ne	-ne	`[[ 6 -ne 20 ]] && echo "6 is not equal to 20"`
conditional evaluation	&&	-a (**)	`[[ -n $var && -f $var ]] && echo "$var is a file"`
	||	-o (**)	`[[ -b $var || -c $var ]] && echo "$var is a device"`
expression grouping	(...)	\(... \) (**)	`[[ $var = img* && ($var = *.png || $var = *.jpg) ]] && echo "$var starts with img and ends with .jpg or .png"`
Pattern matching	= (or ==)	(not available)	`[[ $name = a* ]] || echo "name does not start with an 'a': $name"`
RegularExpression matching	=~	(not available)	`[[ $(date) =~ ^Fri\ ...\ 13 ]] && echo "It's Friday the 13th!"`

(*) This is an extension to the POSIX standard; some shells may have it, others may not.

(**) The -a and -o operators, and `( ... )` grouping, are defined by POSIX but only for strictly limited cases, and are marked as deprecated. Use of these operators is discouraged; you should use multiple [commands instead:

- `if [ "$a" = a ] && [ "$b" = b ]; then ...`
- `if [ "$a" = a ] || { [ "$b" = b ] && [ "$c" = c ];}; then ...`

If Elif Else

if [<some test>]

then

<commands>

elif [<some test>]

then

<different commands>

else

<other commands>

fi

Case Statements

case <variable> in

<pattern 1>)

<commands>

;;

<pattern 2>)

<other commands>

;;

esac

While

while [<some test>]

do

<commands>

done

Until

until [<some test>]

do

<commands>

done

For

for var in <list>

do

<commands>

done

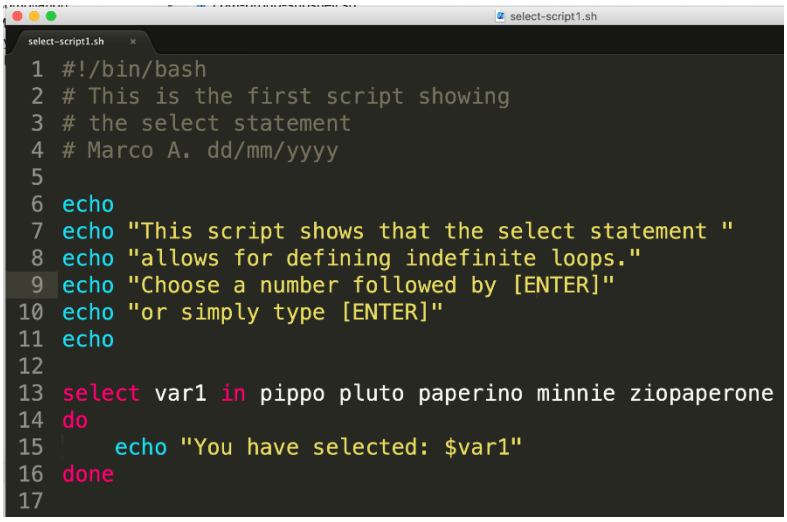
Select

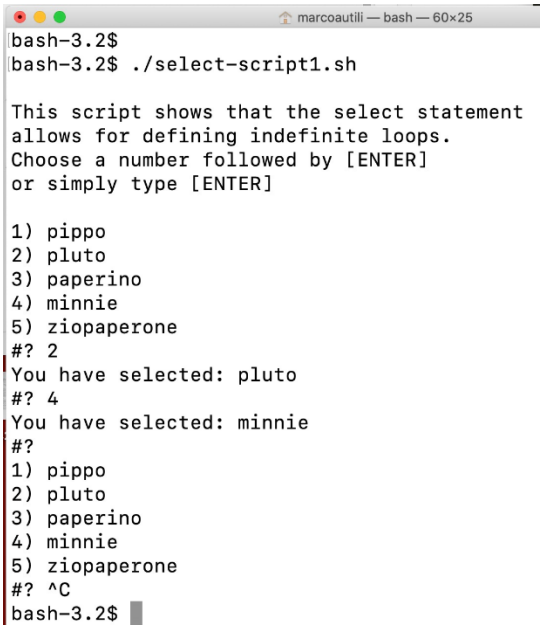
select var in <list>

do

<commands>

done





Break and Continue

Break [n] - exit from a for, while, until, or select loop

- If n is supplied, the nth enclosing loop is exited
- The return status is zero unless n is not greater than or equal to 1

Continue [n] - resume the next iteration of an enclosing for, while, until, or select loop

- If n is supplied, the execution of the nth enclosing loop is resumed
- The return status is zero unless n is not greater than or equal to 1

Bash Functions

function_name () {

<commands>

}

or

function function_name {

<commands>

}

Bash functions do not allow to return a value to the caller.

To return values you can:

- set a global variable with the result

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

myfunc
echo $myresult
```

- use command substitution

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

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

The result is output to the **stdout** and the caller uses command substitution to capture the value in a variable.

- pass as input the name of a variable to be used as the result variable

```
function myfunc()
{
    local  __resultvar=$1
    local  myresult='some value'
    eval $__resultvar="'$myresult'"
}

myfunc result
echo $result
```

No interpretation
'$myresult' is a string

First interpretation (substitution/expansion)
result='some value' (it is also a string)

Second interpretation (assignement)
'some value' is assigned to result

Since we have the name of the variable to set stored in a variable, we can't set the variable directly, we have to use eval to actually do the setting

eval [arguments] is a builtin command. The arguments are concatenated together into a single command, which is then read and executed, and its exit status is returned as the exit status of eval. If there are no arguments or only empty arguments, the return status is zero

The return statement of bash functions specify the function's status which is a numeric value assigned to the variable $?

Eval

- To execute a command stored in the string

```
cmd='ls -la'

eval $cmd

As output, you will have the list of files in the current folder
```

- To print the value of a variable which is again a variable with value assigned to it

```
bash-3.2$
bash-3.2$ a=10
bash-3.2$ b=a
bash-3.2$ echo $b
a
bash-3.2$ c='$'$b
bash-3.2$ echo $c
$a
bash-3.2$ eval c='$'$b
bash-3.2$ echo $c
10
bash-3.2$ echo $a
10
bash-3.2$
```

NOTE that the dollar sign must be escaped with '$'