
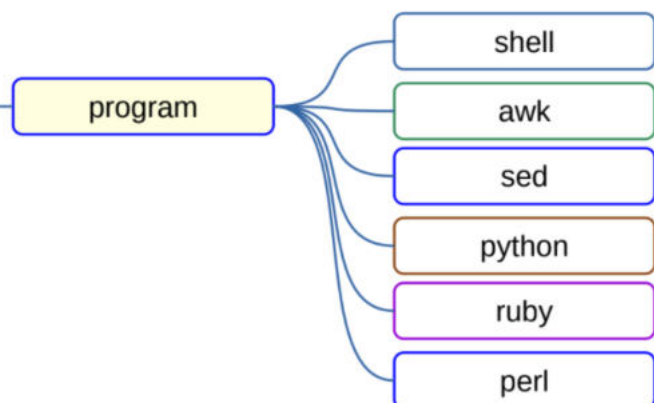
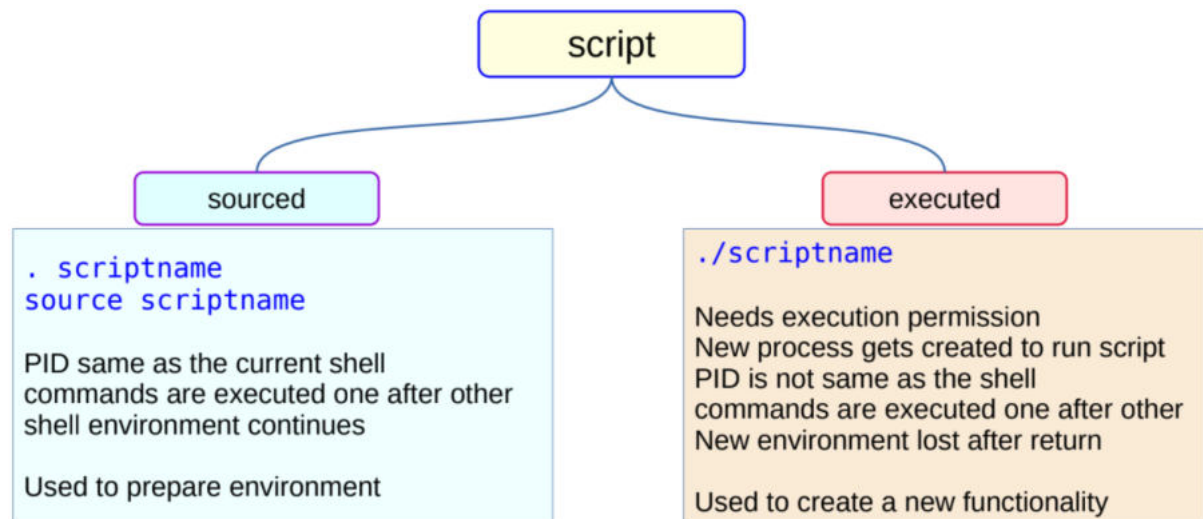


bash

▼ Type	 Lecture
📅 Date	@February 22, 2022
☰ Lecture #	?
🔗 Lecture URL	
🔗 Notion URL	https://21f1003586.notion.site/bash-60e8a6fdb5394c4385d7ac79f7c7b902
# Week #	5

```
#!/ interpreter
# comments
commands
loops
variables
case statements
functions
```



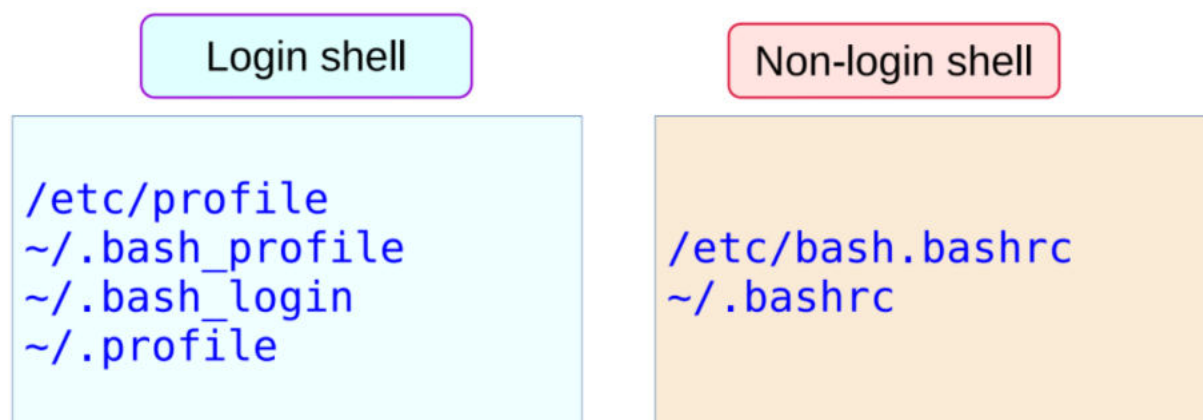


Script location

- Use absolute path or relative path while executing the script
- Keep the script in the folder listed in `$PATH`
- Watch out for the sequence of directories in `$PATH`

bash environment

- When you are already logged in to a system and using the GNOME environment to open a terminal, then the shell we are opening is not asking for a login and is called a **Non-login shell**
- When we ssh into a remote machine, then the shell we are logging into would have checked for the login credentials and then opened up the command line environment for us, therefore that shell is called a **Login shell**



Output from the shell scripts

- `echo`
 - Terminates with a newline if `-n` flag is not given
 - `echo My home is $HOME`
- `printf`
 - Supports format specifiers like in C
 - `printf "My home is %s\n" $HOME`

Input to shell scripts

- `read var`
 - String read from command line is stored in `$var`

Shell script arguments

```
./myscript.sh -l arg2 -v arg4
```

- `$0` → name of the shell program
- `$#` → number of arguments passed
- `$1` or `${1}` → first argument
- `${11}` → eleventh argument
- `$*` or `$@` → all arguments at once
- `"$*"` → all arguments as a single string
- `"$@"` → all arguments as separate strings

Command substitution

```
var=`command`  
var=$(command)
```

`command` is executed and the output is substituted

Here, the variable `var` will be assigned with that output

`for do` loop

```
for var in list  
do
```

```
commands
done
```

`commands` are executed once for each item in the `list`

space is the field delimiters

set IFS if required

`case` statement

```
case var in
pattern1)
  commands
  ;;
pattern2)
  commands
  ;;
esac
```

`commands` are executed

each pattern matched

for var in the options

`if` loop

```
if condition
then
  commands
fi
```

```
if condition; then
  commands
fi
```

`commands` are executed only if `condition` returns true

`conditions` in `if`

- `test -e file` or any `text expression`
 - True, if the file exists in the current directory of the shell script
 - else False

- `[-e file]` or any `[expression]`
 - Same as the above one, just a different syntax
- `[[expression]]` like `[[$ver == 5.*]]`
 - If we intend to use regex or other complex operations
- `((expression))` like `(($v ** 2 > 10))`
 - To perform complex arithmetic operations for the test conditions
- Or just use a `command` like `wc -l file`
 - If the command returns True, i.e. upon successful execution, this means the condition is satisfied
 - else False
- `pipeline` like `who | grep "joy" > /dev/null`
 - A set of commands which can be combined with other commands and redirection of output and combination of logical expressions using the `&&` or `||`
 - These can be put as the condition

For negation, use `!` before the condition

test numeric comparisons

- `$n1 -eq $n2`
 - Check if n1 is equal to n2
- `$n1 -ge $n2`
 - Check if n1 is greater than or equal to n2
- `$n1 -gt $n2`
 - Check if n1 is greater than n2
- `$n1 -le $n2`
 - Check if n1 is less than or equal to n2
- `$n1 -lt $n2`
 - Check if n1 is less than n2
- `$n1 -ne $n2`

- Check if n1 is not equal to n2

test string comparisons

- `$str1 = $str2`
 - Check if str1 is the same as str2
- `$str1 != $str2`
 - Check if str1 is not the same as str2
- `$str1 < $str2`
 - Check if str1 is less than str2
 - Compares based on lexicographical (alphabetical) order
- `$str1 > $str2`
 - Check if str1 is greater than str2
 - Compares based on lexicographical (alphabetical) order
- `-n $str1`
 - Check if str1 has length greater than zero
- `-z $str1`
 - Check if str1 has the length zero

Unary file comparisons

- `-e file`
 - Check if the file exists
- `-d file`
 - Check if the file exists and is a directory
- `-f file`
 - Check if the file exists and is a file
- `-r file`
 - Check if the file exists and is readable
- `-s file`
 - Check if the file exists and is not empty

- `-w file`
 - Check if the file exists and is writable
- `-x file`
 - Check if the file exists and is executable
- `-0 file`
 - Check if the file exists and is owned by the current user
- `-G file`
 - Check if the file exists and the default group is the same as that of the current user

Binary file comparisons

- `file1 -nt file2`
 - Check if file1 is newer than file2
- `file1 -ot file2`
 - Check if file1 is older than file2

`while do` loop

```
while condition
do
  commands
done
```

`commands` are executed only if the `condition` returns `true`

`until do` loop

```
until condition
do
  commands
done
```

`commands` are executed only if the `condition` returns `false`

functions

definition

```
myfunc()  
{  
  commands  
}
```

call

```
myfunc
```

`commands` are executed each time `myfunc` is called

Definitions must be before the calls

Debugging

```
set -x  
./myscript.sh
```

Prints the command before executing it

Place the `set -x` inside the script

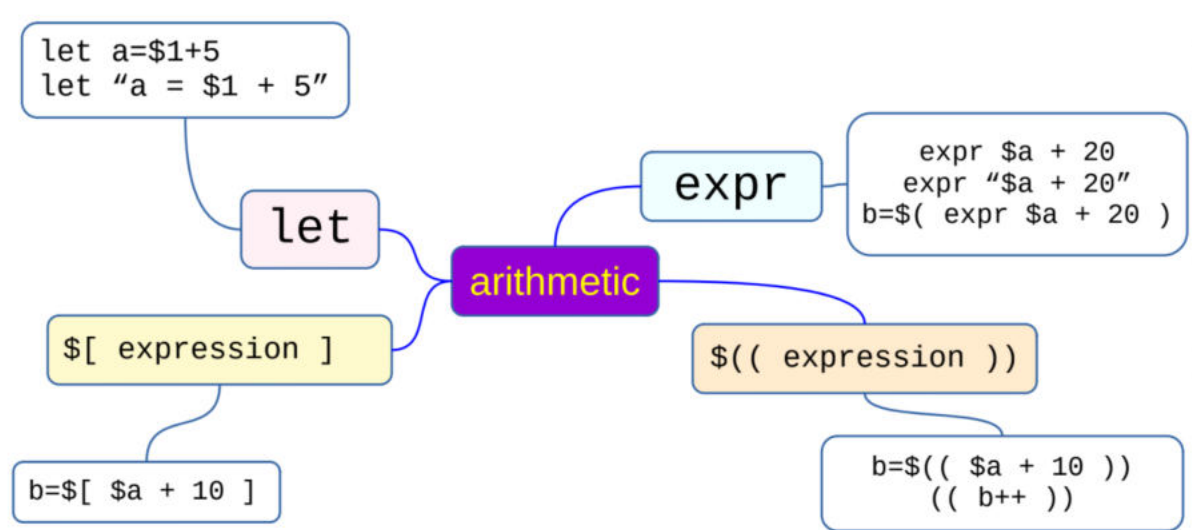
```
bash -x ./myscript.sh
```

Combining conditions

```
[ $a -gt 3 ] && [ $a -lt 7 ]
```

```
[ $a -le 3 ] || [ $a -ge 7 ]
```

Shell arithmetic



expr command operators

<code>a + b</code>	Return arithmetic sum of a and b
<code>a - b</code>	Return arithmetic difference of a and b
<code>a * b</code>	Return arithmetic product of a and b
<code>a / b</code>	Return arithmetic quotient of a divided by b
<code>a % b</code>	Return arithmetic remainder of a divided by b
<code>a > b</code>	Return 1 if a greater than b; else return 0
<code>a >= b</code>	Return 1 if a greater than or equal to b; else return 0
<code>a < b</code>	Return 1 if a less than b; else return 0
<code>a <= b</code>	Return 1 if a less than or equal to b; else return 0
<code>a = b</code>	Return 1 if a equals b; else return 0

a b	Return a if neither argument is null or 0; else return b
a & b	Return a if neither argument is null or 0; else return 0
a != b	Return 1 if a is not equal to b; else return 0
str : reg	Return the position upto anchored pattern match with BRE str
match str reg	Return the pattern match if reg matches pattern in str
substr str n m	Return the substring m chars in length starting at position n
index str chars	Return position in str where any one of chars is found; else return 0
length str	Return numeric length of string str
+ token	Interpret token as string even if its a keyword
(exprn)	Return the value of expression exprn

```
#!/bin/bash

if [ $# -lt 2 ]; then
    echo "Use 2 natural numbers as arguments";
    exit 1;
fi;

regex='^[0-9]+$'
if ! [[ $1 =~ $regex ]]; then
    echo "$1 is not a natural number";
    exit 1;
fi;

if ! [[ $2 =~ $regex ]]; then
    echo "$2 is not a natural number";
    exit 1;
fi;

let a=$1*$2;
echo "Product a is $a";
(( a++ ));
echo "Product a incremented is $a";

let b=$1**$2;
echo "Power is $b";

c=$(( $1 + $2 + 10 ));
echo "sum + 10 is $c";

d=$((expr $1 + $2 + 20));
echo "sum + 20 is $d";

f=$(( $1 * $2 * 2 ));
echo "Product times 2 is $f";
```

```

#!/bin/bash
# The following line is for debugging
# set -x;

# Code starts here
a=256;
b=4;
c=3;

ans=$( expr $a + $b );
echo $ans;

ans=$( expr $a - $b );
echo $ans;

ans=$( expr $a \* $b );
echo $ans;

ans=$( expr $a / $b );
echo $ans;

ans=$( expr $a % $c );
echo $ans;

ans=$( expr $a \> $b );
echo $ans;

ans=$( expr $a \>= $b );
echo $ans;

ans=$( expr $a \< $b );
echo $ans;

ans=$( expr $a \<= $b );
echo $ans;

ans=$( expr $a = $b );
echo $ans;

ans=$( expr $a != $b );
echo $ans;

ans=$( expr $a \| $b );
echo $ans;

ans=$( expr $a \& $b );
echo $ans;

str="octavio version as in Jan 2022 is 6.4.0";
reg="[oO]ctav[aeiou]*";
ans=$( expr "$str" : $reg );
echo $ans;

ans=$( expr substr "$str" 1 6 );
echo $ans;

```

```
ans=$( expr index "$str" "vw" );
echo $ans;

ans=$( expr length "$str" );
echo $ans;
```

heredoc feature

When writing shell scripts you may be in a situation where you need to pass a multiline block of text or code to an interactive command, such as `tee`, `cat`, or `sftp`

In `bash` and other shells like `zsh`, a Here document (`heredoc`) is a type of redirection that allows you to pass multiple lines of input to a command.

This is what a general syntax looks like

```
[COMMAND] <<[-] 'DELIMITER'
    HERE-DOCUMENT
DELIMITER
```

```
a=2.5
b=3.2
c=4
d=$(bc -l << EOF
scale = 5
($a+$b)^$c
EOF
)
echo $d
```

1055.6001

Marker designation need not be EOF

```
a=2.5
b=3.2
c=4
d=$(bc -l <<- ABC
    scale = 5
    ($a+$b)^$c
    ABC
)
echo $d
```

A hyphen tells bash to ignore leading tabs

Notice no-indent (left) vs indentation (right) in the above example

```
#!/bin/bash

# set -x;
echo "path is set as $PATH";
i=0;
IFS=;;

for n in $PATH; do
    echo "$i $n";
    (( i++ ));
done;
```

if-elif-else-fi loop

```
if condition1
then
    commandset1
else
    commandset2
fi
```

```
if condition1
then
    commandset1
elif condition2
then
    commandset2
elif condition3
then
    commandset3
else
    commandset4
fi
```

```
#!/bin/bash

if [ $# -gt 2 ]; then
    echo "More than 2 arguments";
elif [ $# -gt 1 ]; then
    echo "More than 1 argument";
elif [ $# -gt 0 ]; then
    echo "Not enough arguments";
else
    echo "Arguments required";
fi;
```

case statement options

```
case $var in
    op1)
        commandset1;;
    op2 | op3)
        commandset2;;
    op4 | op5 | op6)
        commandset3;;
    *)
        commandset4;;
esac
```

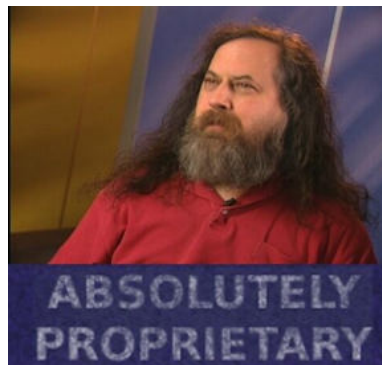
commandset4 is the default
for values of **\$var** not
matching what are listed

```
#!/bin/bash

echo "What is your favourite image processor?";
read pname;

case $pname in
    [gG]imp | inkscape)
        echo "Good choice";
        ;;
    [aA]dobe*)
        echo "Absolutely proprietary and costs a lot";
        ;;
    imagej)
        echo "Measuring things on the image?";
        ;;
    *)
        echo "$pname is a new find for me";
        ;;
esac;
```

When you enter “adobe”



C-style `for` loop → one variable

```
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
    echo $a
done
```

```
#!/bin/bash

begin=1;
finish=10;

for (( a = $begin; a < $finish; a++ )); do
    b=$(( a**2 ));
    echo $b;
done;
```

C-style **for** loop → two variables

```
begin1=1
begin2=10
finish=10
for (( a=$begin1, b=$begin2; a < $finish; a++, b-- ))
do
    echo $a $b
done
```

NOTE: only one condition to close out the for loop

```
#!/bin/bash

begin1=1;
begin2=20;
finish=10;

for (( a = $begin1, b = $begin2; a < $finish; a++, b-- )); do
    c=$(( a**2 ));
    d=$(( b**2 ));
    echo $c $d;
done;
```

Processing output of a loop

```
filename=tmp.$$
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
    echo $a
done > $filename
```

NOTE: Output of the loop is re-directed to the tmp file

```
#!/bin/bash

filename=largefile.txt;
if [ -e $filename ]; then
    echo "file $filename exists";
    exit 1;
fi;

i=1;
while [ $i -lt 10 ]; do
    echo "$i ${i+1}";
    (( i++ ));
done > $filename;

echo "file $filename written";
ls -l $filename;
```

break

To break out of a loop


```

n=10
i=0
while [ $i -lt $n ]
do
    echo $i
    (( i++ ))
    if [ $i -eq 5 ]
    then
        break
    fi
done

```

break out of inner loop

```

n=10
i=0
while [ $i -lt $n ]
do
    echo $i
    j=0
    while [ $j -le $i ]
    do
        printf "$j "
        (( j++ ))
        if [ $j -eq 7 ]
        then
            break 2
        fi
    done
    (( i++ ))
done

```

break out of outer loop

```

0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
0 1 2 3 4 5 6
0 1 2 3 4 5 6

```

`break 2` refers to the outer loop as seen from the nesting

`continue`

Opposite of `break`

```

n=9
i=0
while [ $i -lt $n ]
do
    printf "\n loop $i:"
    j=0
    (( i++ ))
    while [ $j -le $i ]
    do
        (( j++ ))
        if [ $j -gt 3 ] && [ $j -lt 6 ]
        then
            continue
        fi
        printf "$j "
    done
done

```

```

loop 0:1 2
loop 1:1 2 3
loop 2:1 2 3
loop 3:1 2 3
loop 4:1 2 3 6
loop 5:1 2 3 6 7
loop 6:1 2 3 6 7 8
loop 7:1 2 3 6 7 8 9
loop 8:1 2 3 6 7 8 9 10

```

Continue will skip rest of the commands in the loop and goes to next iteration

shift

```

i=1
while [ -n "$1" ]
do
    echo argument $i is $1
    shift
    (( i++ ))
done

```

shift will shift the command line arguments by one to the left.

It keeps on shifting (read: delete/destroy) the arguments to the left

```

#!/bin/bash

echo "Number of args: ";
echo $#;
i=1;

while [ -n "$1" ]; do
    echo "arg $i is $1";
    shift;
    (( i++ ));
done;

echo "Number of args now: ";
echo $#;

```

exec

```
exec ./my-executable --my-options --my-args
```

- To replace shell with a new program or to change i/o settings
- If new program is launched successfully, it will not return control back to the shell
- If new program fails to launch, the shell continues

```
#!/bin/bash
echo "PID of shell running this command: $$";
echo "Leaving bash and opening xterm if available";
exec xterm;
echo "Looks like xterm is not available or failed to start";
```

eval

eval my-arg

- Executes argument as a shell command
- Combines arguments into a single string
- Returns control to the shell with exit status

```
#!/bin/bash
cmd="date";
fmt="+%d-%B-%Y";
eval $cmd $fmt;
```

Functions in bash

```
#!/bin/bash

usage() {
    echo "usage $1 str1 str2";
}

swap() {
    echo "$2 $1";
}

if [ $# -lt 2 ]; then
    usage $0;
    exit 1;
fi;

swap $1 $2;
```

getopts

```

while getopts "ab:c:" options;
do
    case "${options}" in
        b)
            barg=${OPTARG}
            echo accepted: -b $barg
            ;;
        c)
            carg=${OPTARG}
            echo accepted: -c $carg
            ;;
        a)
            echo accepted: -a
            ;;
        *)
            echo Usage: -a -b barg -c carg
            ;;
    esac
done

```

This script can be invoked with only three options: **a**, **b**, **c**. The options **b** and **c** will take arguments.

```

#!/bin/bash
while getopts "ab:c:" options; do
    case "${options}" in
        b)
            barg=${OPTARG};
            echo "accepted: -b $barg";
            ;;
        c)
            carg=${OPTARG};
            echo "accepted: -c $carg";
            ;;
        a)
            echo "accepted: -b";
            ;;
        *)
            echo "Usage: -a -b barg -c carg";
            ;;
    esac;
done;

```

Usage

```

khaqu@DESKTOP-77CS341 MINGW64 ~/Documents/iitm-term4/sc/week6/lec1
$ bash getopts.sh -a -b bargument -c cargument
accepted: -b
accepted: -b bargument
accepted: -c cargument

khaqu@DESKTOP-77CS341 MINGW64 ~/Documents/iitm-term4/sc/week6/lec1
$ bash getopts.sh -a -b bargument -c cargument -d -e
accepted: -b
accepted: -b bargument
accepted: -c cargument
getopts.sh: illegal option -- d
Usage: -a -b barg -c carg
getopts.sh: illegal option -- e
Usage: -a -b barg -c carg

```

select loop

```

echo select a middle one
select i in {1..10}
do
    case $i in
        1 | 2 | 3)
            echo you picked a small one;;
        8 | 9 | 10)
            echo you picked a big one;;
        4 | 5 | 6 | 7)
            echo you picked the right one
            break;;
    esac
done
echo selection completed with $i

```

Text menu !

```

#!/bin/bash

echo "select a middle one";
select i in {1..10}; do
    case $i in
        1 | 2 | 3)
            echo "you picked a small one";
            ;;
        8 | 9 | 10)
            echo "you picked a big one";
            ;;
        4 | 5 | 6 | 7)
            echo "you picked the right one";
            break;
            ;;
    esac
done

```

```
    esac;  
done;  
  
echo "selection completed with $i";
```