Server System Management - Linux

Lab 7 – Bash and scripting

INHOUD

[1 Lab 7 – Bash and scripting 3](#_Toc156918112)

[1.1 Warming up... 4](#_Toc156918113)

[1.2 Split into tokens 4](#_Toc156918114)

[1.3 Brace expansiON 4](#_Toc156918115)

[1.4 Tilde expansion 5](#_Toc156918116)

[5](#_Toc156918117)

[0. Parameter expansion 6](#_Toc156918118)

[1. Command substitution 6](#_Toc156918119)

[2. Arithmetic substitution 6](#_Toc156918120)

[7](#_Toc156918121)

[3. Word splitting 7](#_Toc156918122)

[4. Filename expansion 8](#_Toc156918123)

[5. Quote removal – quoting rules 9](#_Toc156918124)

[1.5 Redirection 9](#_Toc156918125)

[13. Programming constructs 12](#_Toc156918126)

[14. Advanced topics 14](#_Toc156918127)

# 

# Lab 7 – Bash and scripting

To be effective in bash (and other shells), you need to understand how bash operates and intervenes before bash executes your command. Other shells are very similar, although there can be small (subtle) differences. We focus on the use of bash.

A diagram of a diagram

Description automatically generated

Learn to use info bash and the man pages!

Install the man and info pages, if necessary:

**sudo apt install info**

**sudo apt install bash-doc**

## Warming up...

1. Check the commands **cp**, **dd**, **ln**, **mktemp**, **mkdir**, **touch** and **cat**. Make a directory playground to play for today. Make sure your playground looks like this:

**ls -a**  
**. .. a .a a.b abc ab.c abe a.c .b b.a b.b b.c c.a d def e .e**

touch playground

chmod +x createplayground

vim createplayground

INSIDE:

# Create the directory playground

mkdir playground

# Change directory to playground

cd playground

# Create the files and directories

touch a .a a.b abc ab.c abe a.c .b b.a b.b b.c c.a d def e .e

# Confirm the creation of files and directories

ls -a

A screenshot of a computer program

Description automatically generated

1. What is the output of **head /etc/passwd** ?

The first 10 lines

A screen shot of a computer

Description automatically generated

1. Which commands outputs only the last lines of a file? Make this command show the last lines of a file while you write to it?

tail -f?

but it works somewhat strange

1. What’s the use of **-rf** of the **rm** command?

-r = recursive

-f = ignore nonexistent files and arguments, never prompt

1. Now, create a file named **-rf** in your playground without using an editor, just a command.

**$ ls -a**  
**. .. a .a a.b abc ab.c abe a.c .b b.a b.b b.c c.a d def e .e -rf**

**touch ./-rf**

**touch playground/-rf**

1. How can you remove the file **-rf** ? (QUIZ)

rm ./-rf (correct)

rm playground/-rf

## Split into tokens

If the input is not commented, the shell reads it and divides it into words and operators, employing quoting rules to define the meaning of each character of input:

* 1. read input from file, string or terminal
  2. break up into tokens separated by meta-characters and obeying quoting rules
  3. parse tokens
  4. shell expansions
  5. redirect
  6. execute
  7. optionally wait and collect exit status

For us… nothing to do here :).

## Brace expansiON

1. In your playground, predict, run and explain:

***Note:***

Can have a prefix and suffix

Should not contain metacharacters

this is like multiplying 2 parantheses (1+2)(3-4) = 1\*3 – 1\*4 + 2\*3 – 2\*4  
**echo {0,1,2,3}{a,b,c,d}**

****

**Will print every character “multiplied with” every other character  
echo {1,"",3}{a,2,c,d}**

****

**Will print every character multiplied with every character except when it meets “” then it considers it nothing (or somewhat 0, or 1, depends how you look at maths) and hence you do not get output from it.  
echo 0{1,2,3}{a,b,c}d**

****

**Same principle, but it will keep showing all the results between “a..d”  
echo {0,..,3}{a,..,d}**

****

**It considers .. to be a character, hence it multiplies it.  
echo {0,1,2,3}{a,b, c, d}**



It considers “{a,b” to be a separate object and hence does not really multiply them, i mean it does, but leaves the c, d} for the end.

1. Using brace expansion, how can you print all numbers from 0 to 29 without leading 0’s using brace expansion? (0 1 2 … 28 29)

echo {0..29}



## Tilde expansion

**echo ~** shows your current home directory.

Try to find out using **info bash** how you can use tilde expansion to show:

* 1. the home directory of another user

echo ~week6

A screen shot of a computer program

Description automatically generated

* 1. your current working directory

echo ~+

* 1. your previous working directory

echo ~-

***Hint:***

run **info bash** and navigate to the section **Expansion - Tilde Expansion** or

run **info bash expansion** and navitage to T**ilde expansion**

## Parameter expansion

1. Define a variable with a random date as content: **date="27/04/2022"** and show the value of variable **date** by using **echo**

**echo $date**

**A screen shot of a computer program

Description automatically generated**

1. Read input into variables (separated by whitespace) with the **read** command:

**read firstname lastname info**

**firstname** should contain your first name  
**lastname** should contains your last name  
**info** should contain your address, phone number, …

A screen shot of a computer code

Description automatically generated

You just type them at once with everything else.

## Command substitution

***Hint:***

The syntax for command substitution in bash is **$(…)** or **`…`***(deprecated)*

1. Explain the (difference of) output of commands:

**echo date**

**will simply say word “date”**  
**echo $date**

**will show the valie of date**  
**echo ${date}**

**will also show the value of date**  
**echo $(date) echo `$date` - The first part will show the value of the command date (it executes it), but the second part will not work I guess**

**A screen shot of a computer program

Description automatically generated**

1. Make a parameter **files\_with\_extension\_c** which holds the output of the command **ls \*.c** (inside your playground) and check if this variable contains the right files:

**$ echo $files\_with\_extension\_c**  
**ab.c a.c b.c**

**A screen shot of a computer program

Description automatically generated**

**A screen shot of a computer code

Description automatically generated**

## Arithmetic substitution

1. Explain the (difference of) output of commands:

**echo 5 + 3 echo `5 + 3`**

**A screen shot of a computer program

Description automatically generated**

**First shows actual text, the other one will try to execute as command**

**echo $(5 + 3)**

**Will try to execute as well, as commands**

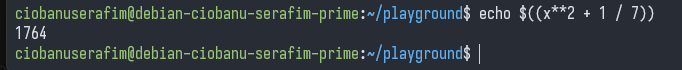
**echo $((5 + 3))**

**Will give actual computation**

1. Define a variable **x=42**. Use arithmetic substitution to calculate ( x^2 + 1 ) / 7

A screen shot of a computer program

Description automatically generated



echo "$(( (x \* x + 1) / 7 ))"

***Operations and order of precedence:***

**VAR++ VAR--** variable post-increment and -decrement

**++VAR –VAR** variable pre-increment and -decrement

**+ -** unary plus/minus

**~ !** logical/bitwise negation

**\*\*** exponentiation

**\* / %** multiplication, division, modulo

**+ -** addition and subtraction

**<< >>** left/right bitwise shift

**<= >= < >** comparison operators

**== !=** equality and inequality

**&**  bitwise AND

**^** bitwise exclusive OR

**|** bitwise OR

**&&** logical AND

**||** logical OR

**expr?expr:expr** c-style conditional operator

**= \*= /= %= +=  
 -= <<= >>= &=  
 ^= !=|** assignments

**,** expression separator

## Word splitting

***Note:***

The shell treats each character of **$IFS** as a delimiter, and splits the results of the other expansions into words using these characters as field terminators. If **IFS** is unset, the default is used:

**<space><tab><newline>** **.**

Run and explain:

**echo I write 4 words**

**echo I write 4 words**

**echo "this is five words"**

**echo without spaces**

**echo "with spaces"**

**echo with\ \ \ \ spaces**

**echo I write $((2+2)) words**

**echo I write \  
 $(( 16>>2 )) \  
 words**

**A screen shot of a computer program

Description automatically generated**

## Filename expansion

***Note:***

Expands to existing filenames which match the pattern

**If there is no match, the pattern is treated as an ordinary string**

1. Predict and explain the output of commands:

**dir a\*.\* - start with a and has anything after it and any extension (actually have an extension)**

**dir a\* - start with a (not only those that have extension)**

**dir \*a – has a somewhere in the file name but not at the start**

**dir a\\* - not a thing**

**ls ab[^c] – files that after ab have anything but letter c**

**ls ab[c-f] – files that after ab have letters**

**ls ab[c\-f] – files that contain only c?**

**ls – list normal files (without hidden)**

**printf "%s\n"[abcd] - %s is used as a placeholder, and the \n stands for newline, and it will look for files that have one of the letters from the brackets**

**printf "%s\n" [!abcd] – will look for files that DO NOT contain that any of the letters specified.**

**printf "%s\n" [^abcd] – similar to the previous one**

**printf "%s\n" [a-d] – any letter from a to d**

**printf "%s\n" [abcd]\*[abcd] – will display something similar to multiplications**

**printf "%s\n" [a-e] – will display something similar to the a-d, but with e included**

**printf "%s\n" [a/-e] – useless?**

**printf "%s\n" [a\-e] – from a to e**

**printf "%s\n" [!\!]\* - will show every file?**

**printf "%s\n" ??? – anything that contains 3 characters**

**printf "%s\n" ??e - anything that contains 3 characters and have 2 characters before letter e**

**printf "%s\n" ?? - anything that contains 2 characters**

1. What is the meaning of the characters **\** and **/** ?

***Patterns:***

**\*** 0 (inclusive!) or more characters

**?** exact 1 character

**[abcd]** exact 1 character, but only those that are listed (here: a, b, c or d)

**[a-d]** exact 1 character, but only those that are listed (here: a till d)

**[^ab] [!ab]** exact 1 character, but not those that are listed (here: not a nor b)

The - inside the square brackets is being escaped with a backslash \ to match the literal hyphen character.

the presence of the forward slash / inside the character class makes it match either the character a, the character /, or any character in the range from - to e.

1. Show all files in your playground with a name which is exactly 1 character long

A screen shot of a computer

Description automatically generated

1. Explain the difference in output between

**ls abcd\*** and **printf abcd\***

**one will try to list something that has the full “adcd” and anything after that, but the other one is like echo.**

**A screen shot of a computer

Description automatically generated**

1. Is there a difference between **find /etc -name "def\*"** and **find /etc -name def\*** ?

A screen shot of a computer program

Description automatically generated

Its overall better to run it like “def\*” as you can actually find files. The other one will first look for files and then get the find command.

* 1. Run both **find** commands and check the output from your playground.
  2. Run **touch default** in your playground
  3. Run both **find** commands again in your playground

A screen shot of a computer

Description automatically generated

* 1. Explain

The command seems to understand (in the second case) that it needs to preceed everything with the first thing that it found, inside of the directory we currently in. So overall, it does not look for files that match the pattern def\*, but it FIRST gets the filename, from our directory, and then looks for this pattern in /etc. So in this case it would look for “default” inside /etc, instead of def\* like it should.

## Quote removal – quoting rules

Try and explain the output of:

***Note:***

**\** preserve literal value of the next character

**"…"** preserves the literal value of all characters within the quotes, with the exception of **$**, **`** and **\**

**‘…’** preserves the literal value of each character (incl backslash!) within the quotes

***Remember:***

**`…`** *command substitution, not related to quoting(!)*

**echo '$date:' $date**

**echo '${date}:' ${date} echo '$(date): $(date)' echo '`date`:' `date`**

**echo '`date`:' "`date`" printf "%s\n" '$(date)'**

**printf "%s\n" "$(date)"**

## Redirection

1. Remove all work files **tmp\*.txt** (if some would exist): **rm -f tmp\*.txt**. Try all next commands in order:

***Note:***

***File descriptors:***

***0 STDIN***

***1 STDOUT***

***2 STDERR***

***Redirections:***

***> write***

***>> append***

***>| overwrite***

**du /etc**

**du /var > tmp.txt**

**du /etc 1> tmp.txt**

**du /var >> tmp.txt**

**du /etc 1>> tmp.txt**

**> tmp.txt du /etc**

**du /var > tmp1.txt > tmp2.txt**

Check each time:

* 1. the output of the command,
  2. the state of the work files **tmp\*.txt**: **wc tmp\*.txt** or **ls -l tmp\*.txt**),
  3. the content of the work files **tmp\*.txt**: **cat tmp\*.txt** or by opening in a text editor.

What is your conclusion after comparing the results of each step?

That this is really useful and all you do is really write stuff to temp files with redirection.

1. Make a file **myinfo** with content:

**<voornaam> <achternaam> <adres telefoonnummer ...>**

run **read firstname surname info < myinfo** and check the content of the variables: **firstname**, **surname** and **info**.

A screen shot of a computer program

Description automatically generated

Cool, we can get values from the file

1. Error messages are written to a different channel. This makes it possible to distinguish between normal messages and error messages. Run the next commands and check the content of the work files **tmp\*.txt**:

**du /etc**

**du /etc 1>tmp.txt**

**du /etc 2>>tmp.txt**

**du /etc >tmp1.txt 2>tmp2.txt**

**What am I supposed to understand ? That the errors are being redirected ? Because checking the files is really not interesting or easy to understand.**

1. Try more possibilities and see how the order of the redirection matters:

**du /etc >tmp.txt 2>&1**

**du /etc 2>&1 >tmp.txt**

**du /etc &>tmp.txt**

**du /etc 2>tmp.txt >tmp.txt**

## Pipes

1. Make the output of the command **printf "this is an error"** to be interpreted as an error, not as a normal command. This means, the text should be shown on the screen, but not piped to a next command. (Remember: when you pipe, the input of the next command is the normal output of the previous command.)

printf "this is an error" >&2

1. If you know that only the standard output is piped to the next command, how can you, using a **wc**-filter, catch the number of error messages of the command **du /etc** ?

du /etc 2>tmp.log | wc -l < tmp.log

This is not even working

1. The command **uniq -d <filename>** is used to show all the lines that are present more than once in **<filename>**. Make a file **fruits** with content:

**banana**  
**apple**  
**banana**  
**pear**  
**banana**  
**pear**  
**apple**  
**mango**

However, if one runs the **uniq**-command on **fruits**, no line is shown. What is the reason? What is an easy solution for this issue, using piping?

**sort fruits | uniq -d**

1. Check how many times you ran the command man in the past (before the current login session). Use the file **.bash\_history** from your home-directory to do so. Use the commands **wc** and **grep**.

It just does not want to work as I need it.

The closest I got is this:

grep -c '^man ' ~/.bash\_history

## More advanced oneliners

1. How would you run a command and redirect the output to a file that is only writable by another user? What does work and what doesn't and why?

**sudo echo hello >> /root/hellofile**

**This one does not work, because you are not allowed to write to /root**

**sudo sh -c "echo hello >> /root/hellofile"** Useful scripting commands

**Here you are executing a full shell command and giiving the root password.**

Explain the output of the commands given below (use **man** and **info** pages!).

**sed**

**sed** is a programming language to parse and transform text (stream editor).

**echo xyz xyz | sed 's/x/was x-/g'**

s – means a substitution flag

x – the pattern to be replaced

was -x – the pattern to replace with

g – global flag, meaning all occurrences on all lines will be affected by it.

**sed 's/banana/orange/g' fruits**

**cat the fruits file, and substitute the banana with orange**

**sed -e '1,3d' fruits**

**Here it is:**

**-e – specifies the editing operation to delete lines 1 to 3**

**1,3 – range of lines to delete**

**d – delete command**

**awk**

**awk** is also a programming language, It handles files that are formatted in a predictable way.

**awk '{print}' /etc/fstab**

**works like cat**

This command uses awk to print each line of the /etc/fstab file. The pattern {print} specifies to print each record (line) of input.

**awk '/UUID/' /etc/fstab**

The command awk '/UUID/' /etc/fstab will search for lines in the /etc/fstab file that contain the string "UUID" and print those lines.

**awk '/^UUID/ {print $1;}' /etc/fstab**

The command awk '/^UUID/ {print $1;}' /etc/fstab will search for lines in the /etc/fstab file that start with "UUID". When such a line is found, it will print the first field (column) of that line.

A screen shot of a computer program

Description automatically generated

First column in this case is not delimited by space hence it reads full UUID

**sudo awk 'BEGIN { FS=":"; print**   
**"User\t\tUID\t\tGID\t\tHome\t\tShell\n--------------"; }**   
**{print $1,"\t\t",$3,"\t\t",$4,"\t\t",$6,"\t\t",$7;}**   
**END { print "---------\nFile Complete" }' /etc/passwd | more**

The provided awk command is designed to parse the /etc/passwd file, extract specific fields, format them, and display them in a tabular format with a header and footer. Here's a breakdown of what each part of the command does:

**grep**

**grep banaan fr\***

**Look for banana in a file that starts with fr**

**grep a fr\***

**Look for words that contain letter a in file that starts with fr**

**grep pe fruits**

same

**grep .\*pe fruits**

**grep .\*pe. fruits**

**grep .\*pe.. fruits**

**grep .\*pe... Fruits**

**.\* - matches zero or more occurrences of any character**

**pe – matches the characters “pe” literally**

**.. – matches exactly two characters**

**sort**

**sort fruits**

**sort alphabetically**

**sort -r fruits**

sort in reverse

**sort -u fruits**

show only unique fruits

**wc**

**ls -l | wc**

The command ls -l | wc is used to count the number of lines, words, and characters in the output of the ls -l command.

**wc fruits**

**lines, words, characters of fruits**

**wc -L fruits**

The command wc -L fruits is used to find the length of the longest line in the file named fruits.

**sort -u fruits | wc**

unique fruits and lines, words, characters.

**uniq**

**sort fruits | uniq**

sort fruits | uniq: This command first sorts the lines of the fruits file alphabetically, then passes the sorted lines to the uniq command. uniq removes adjacent duplicate lines, so this command effectively removes duplicate lines from the file and outputs the unique lines in sorted order.

**sort fruits | uniq –d**

sort fruits | uniq -d: This command first sorts the lines of the fruits file alphabetically, then passes the sorted lines to the uniq command with the -d option. The -d option tells uniq to only output duplicate lines. So, this command effectively outputs only the duplicate lines found in the fruits file.

**sort fruits | uniq –c**

count amount of each fruit (occurrence)

## Programming constructs

### debugging scripts

Bash scripts can be debugged by running it with a -x flag:

For example create this script and find the error (cause):

**$ nano some-script**   
**#!/bin/bash**   
   
**echo "this is the first line";**   
   
**printf "this is the second line text" >$2 | wc**

Close & save  
**$ chmod +x some-script**

**$ ./some-script**

**$ ./some-script**

**this is the first line**

**./some-script: line 5: $2: ambiguous redirect**

**0       0       0**

**$ bash -x some-script**   
**+ echo 'this is the first line'**

**this is the first line**

**+ printf 'this is the second line text\n'**

**some-script: line 5: $2: ambiguous redirect**

**+ wc**

**0       0       0**

A screen shot of a computer

Description automatically generated

### variables

Make a file showpid with this content:

**#!/bin/bash**   
**echo $BASHPID**

set the executable flag of the file to x and run **./showpid**

chmod +x showpid

./showpid

Make a script which shows your own homedirectory and username using the environmental variables containing your username and path to your home directory – use **env** to find (and **grep**) to find the right environmental variables.

A screen shot of a computer program

Description automatically generated

#!/bin/bash

echo "homedirectory $HOME"

echo "username $USER"

### functions and scope

Make a file **function-scope** with this content:

**#!/bin/bash**

**SCOPE="outer scope"**

**function hello {**

**local LOCAL="local"**

**INNER="non-local"**

**SCOPE="inner scope"**

**echo "inside hello:"**

**echo -e "\tSCOPE: $SCOPE"**

**echo -e "\tINNER: $INNER"**

**echo -e "\tLOCAL: $LOCAL"**

**}**

**echo -e ''$\_{1..80}'\b\_'**

**echo ''; echo "before hello:"**

**echo -e "\tSCOPE: $SCOPE"**

**echo -e "\tINNER: $INNER"**

**echo -e "\tLOCAL: $LOCAL"**

**echo -e ''$\_{1..80}'\b\_' ; echo ''**

**hello**

**echo -e ''$\_{1..80}'\b\_'**

**echo ''; echo "after hello:"**

**echo -e "\tSCOPE: $SCOPE"**

**echo -e "\tINNER: $INNER"**

**echo -e "\tLOCAL: $LOCAL"**

**echo -e ''$\_{1..80}'\b\_' ; echo ''**

* set the executable flag of the file to x, run **./function-scope** and explain the output.(explain the difference between scope and visibility of the different variables).

chmod +x function-scope

SCOPE is both globally scoped and visible inside the function. Changes made to it inside the function affect its global value.

INNER is non-locally scoped. It's defined outside the function and is accessible inside the function. Changes made to it inside the function don't affect its value outside the function.

LOCAL is locally scoped. It's defined only inside the function and is not accessible outside the function. Changes made to it inside the function don't affect its visibility outside the function.

* remove duplicate code (refactor the code: make a function **show\_vars** which shows the different values of scopes before, in and after the hello function)

#!/bin/bash

SCOPE="outer scope"

INNER=""

LOCAL=""

function hello {

local LOCAL="local"

INNER="non-local"

SCOPE="inner scope"

echo "inside hello:"

echo -e "\tSCOPE: $SCOPE"

echo -e "\tINNER: $INNER"

echo -e "\tLOCAL: $LOCAL"

}

function show\_vars {

echo "SCOPE: $SCOPE"

echo "INNER: $INNER"

echo "LOCAL: $LOCAL"

echo ''

}

echo -e ''$\_{1..80}'\b\_'

echo ''

echo "before hello:"

show\_vars

echo -e ''$\_{1..80}'\b\_' ; echo ''

hello

echo -e ''$\_{1..80}'\b\_'

echo ''

echo "after hello:"

show\_vars

echo -e ''$\_{1..80}'\b\_' ; echo ''

### conditionals

1. make a file **if-elif** with this content, run (debug if necessary):

**#!/bin/bash**

**N=2**

**if [$n -eq 1]; then**

**echo value of N is 1**

**elif [$n -eq 2]; then**

**echo value of N is 2**

**else**

**echo value of N is not 1 neither 2**

**fi**

#!/bin/bash

N=2

if [ "$N" -eq 1 ]; then

echo "value of N is 1"

elif [ "$N" -eq 2 ]; then

echo "value of N is 2"

else

echo "value of N is neither 1 nor 2"

fi

1. Make a file **case-example** with this content, run (debug if necessary):

**#!/bin/bash**

**VAL="today is `date`"**

**case $VAL in**

**to)**

**echo case to;;**

**2021)**

**echo we're in the year 2021;;**

**to\*|from\*)**

**echo VAL starts with to or from;;**

**\*)**

**echo catch-all**

**esac**

#!/bin/bash

VAL="today is $(date)"

case $VAL in

to)

echo "case to";;

2021)

echo "we're in the year 2021";;

to\*|from\*)

echo "VAL starts with to or from";;

\*)

echo "catch-all";;

esac

predict the output, set the executable flag of the file to x, fix the bugs(!), run **./case-example** and check if your prediction is right and if the script is working correctly.

### **loops**

1. Make a file **for-example** with this content:

**#!/bin/bash**

**for val in {1..7..3}{a..c} do**

**echo "value is $val"**

**done**

#!/bin/bash

for val in {1..7..3}{a..c}; do

echo "value is $val"

done

predict the output, set the executable flag of the file to x, fix any bugs, run **./for-example** and check if your prediction is right.

1. Make a file **while-example** which calculates the factorial of a variable **count**. The content of the file is:

**#!/bin/bash**

**count=5**

**fac=1**

**while [ $count -lt 0 ]**

**do**

**fact=$(($fact \*\* $count))**

**count=$((--count))**

**done**

**echo $fact**

#!/bin/bash

count=5

fact=1

while [ $count -gt 0 ]

do

fact=$(($fact \* $count))

count=$(($count - 1))

done

echo $fact

predict the expected output (factorial 5 = 120), set the executable flag of the file to x, run **./while-example** and fix any bugs.

### arguments of a script : $1, $2, … with buit-in iterator : $@

1. **sum** is a script that must return the sum of 2 numbers. This is how it looks like now:

**#!/bin/bash**

**NUM1=$1**   
**echo NUM1 + $2**

The script doesn't return the sum though (run **./sum 3 5** ). Solve the bugs.

#!/bin/bash

NUM1=$1

NUM2=$2

echo $((NUM1 + NUM2))

**./sum 3 5 = 8**

1. How can you make **sum** take any number of arguments using the iterator ? How can you provide better feedback to the user using sed ?  (screenshot)

**$ ./sum 3 5 8 13**

**3+5+8+13=29**

A computer screen shot of a program

Description automatically generated

## Advanced topics

1. It is possible to group commands in bash by using **(… ; … ; …)** or **{… ; … ; …}** . There is, however, a difference between both. Explain the difference, by running the examples below from your playground:

**( cd ~ ; ls ; )**

**{ cd ~ ; ls ; }**

**( echo test ; exit 1 ; )**

Creates subshell

**{ echo test ; exit 1 ; }**

Does not create a subshell

1. Similar to **(… ; … ; …)** or **{… ; … ; …}** , there are different ways to invoke a script as well. Look first at the code of the function **function-scope** again. Now run:

**./function-scope ; echo $SCOPE**

**. function-scope ; echo $SCOPE**

**unset SCOPE**

**./function-scope ; echo $SCOPE**

**source function-scope ; echo $SCOPE**

Make a file **pyscript** with content:

**#!/usr/bin/python**   
**print("test")**

Set the executable flag and run:

**./pyscript**

**. pyscript**

**source pyscript**

Explain the difference between

**./<some-script>**

This command executes the script <some-script> located in the current directory

**. <some-script>**

This command sources the script <some-script> into the current shell environment.

**source <some-script>**

This command is equivalent to . <some-script>.

**Original Author**

* Kurt Sys Linux BASH Lab Howest ©2017-2018

**Expanded by**

* Daan Pareit & Guy Van Eeckhout (Linux Lab Howest 2019-2021)
* Small corrections Guy Van Eeckhout (Server System Management 2022)
* Updated Alex De Smet (Server System Management 2024)

**Additional resources**

* <https://www.gnu.org/software/bash/manual>
* <http://wiki.bash-hackers.org/howto/getopts_tutorial>
* <https://devhints.io/bash>
* <http://wiki.bash-hackers.org/howto/redirection_tutorial>
* <http://www.tldp.org/LDP/Bash-Beginners-Guide/html/>
* <http://bruxy.regnet.cz/linux/bash_cheatsheet/bash_cheatsheet.pdf>
* “Introduction to the Bash Shell on Mac OS and Linux” - section 4 “Using Bash More Effectively”: <https://app.pluralsight.com/player?course=introduction-bash-shell-linux-mac-os&author=reindertjan-ekker&name=introduction-bash-shell-linux-mac-os-m4-effect&clip=0&mode=live>

**Sources**

* <http://www.programering.com/q/MjN5cjNwATU.html>
* Theory and Lab Course “Besturingssystemen II” (UGent), G. Van hoogenbemt, J. Moreau, W. Vandenbreen and D. Pareit
* B. Ward, “How Linux Works – what every superuser should know”, 2nd edition, 2014.
* <https://www.gnu.org/software/bash/manual>
* <https://unix.stackexchange.com/questions/136547/what-is-the-difference-between-running-bash-script-sh-and-script-sh>