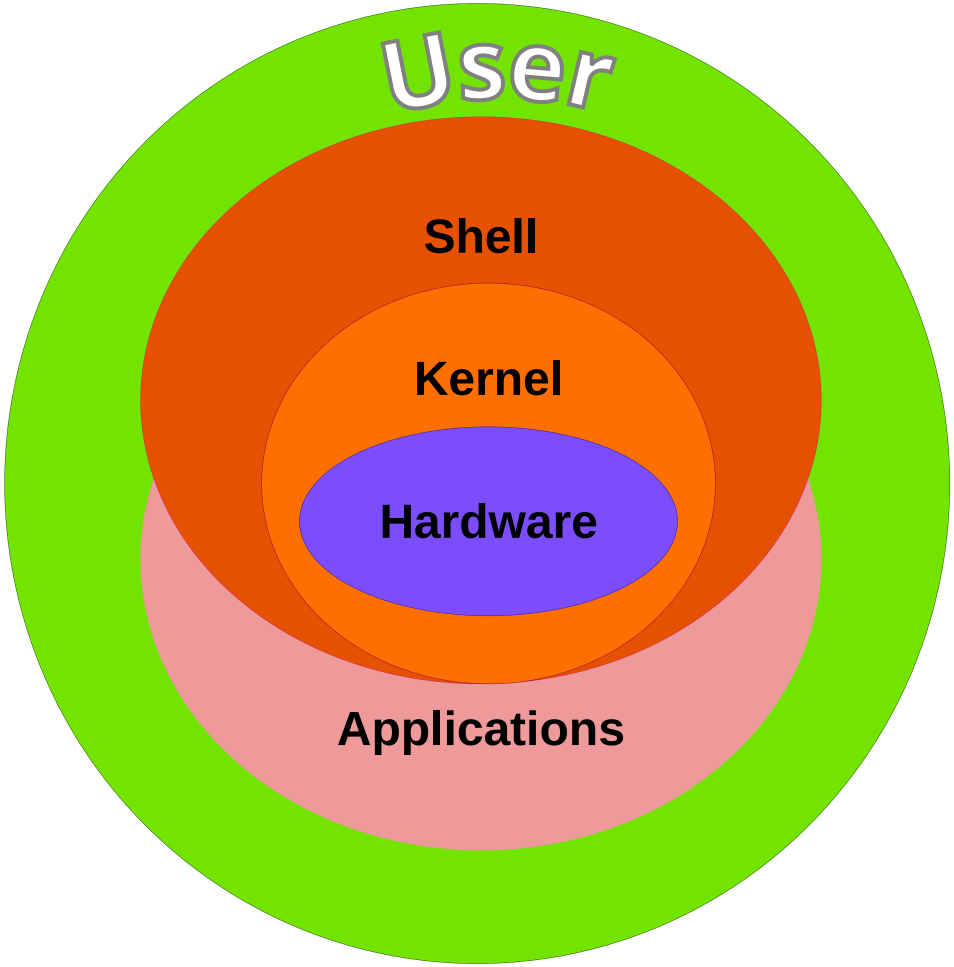
Introduction to BASH scripting

Josefina Campos (jocampos05@gmail.com)
Sol Haim (solhaim@gmail.com)
Tomas Poklepovich (tcaride@gmail.com)
Andrés Culasso (aculasso@gmail.com)

What is BASH?

- **BASH** stands for **B**ourne-**A**gain **Sh**ell
  - Bourne Shell was an improvement of Thompson shell that was the default in UNIX
  - GNU/Linux was created as a freeware version of UNIX, so it has to have a replacement (compatibility) for the shell → BASH

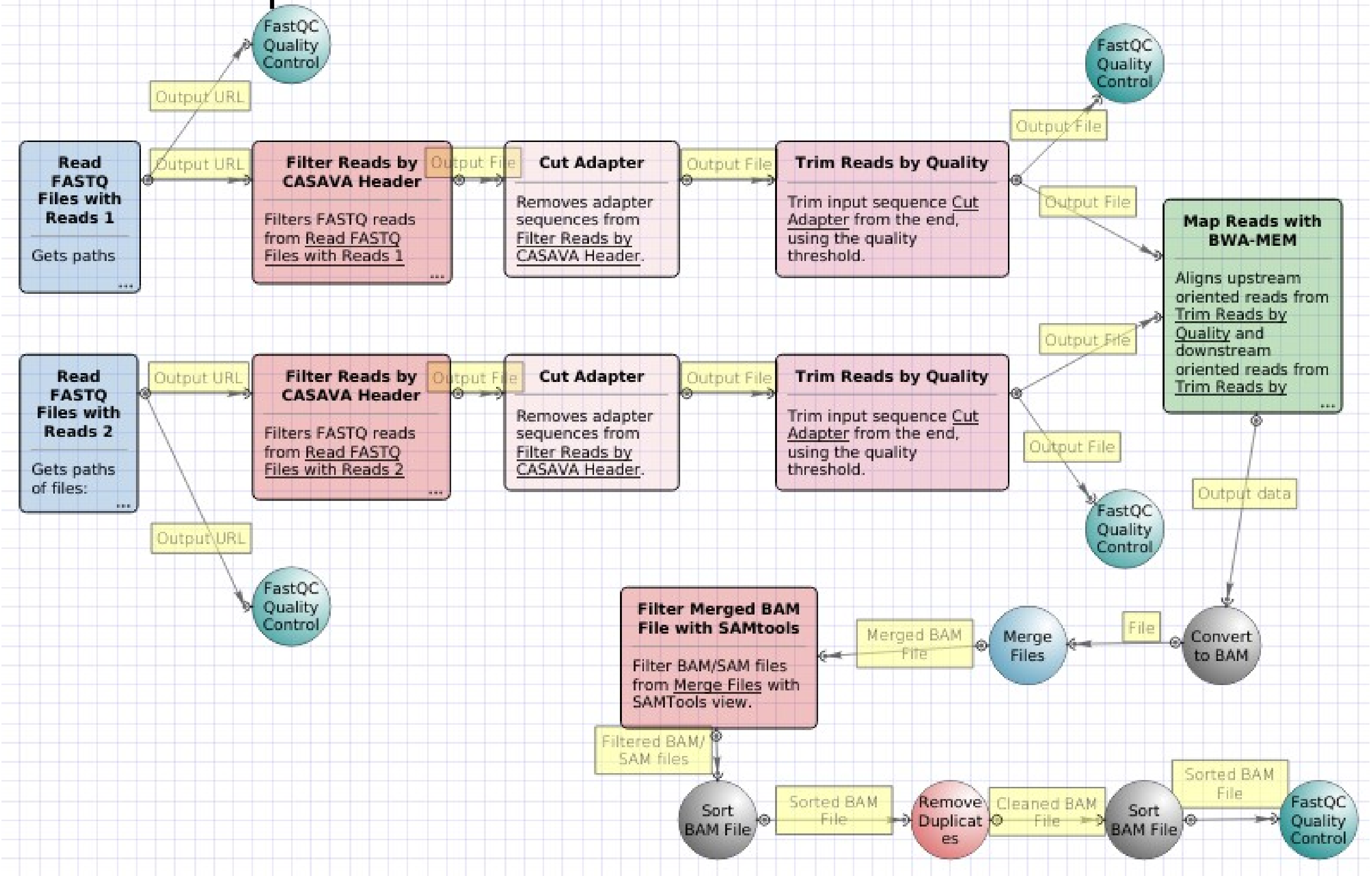
What is a shell?



- **Kernel** is the software that interact with hardware (CPU/GPU, memory, I/O, etc.)
- The user interact with the system mostly through the **Shell** and **Applications.**
  - As an example: the user tell the shell that he/she wants to run some program/application.
- There are both graphical and text based shells.

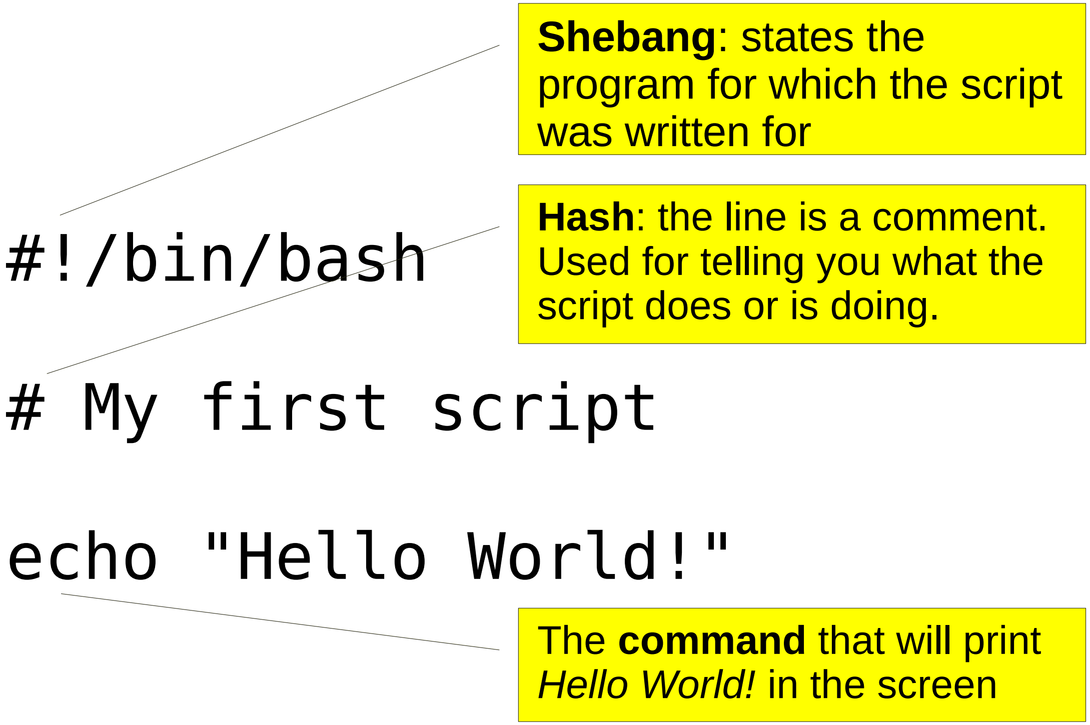
What is a script?

- The script is a program.
  - A program is a set of orders required to do a more or less complex task. You can think in it as a recipe or an experiment protocol.
- All scripts are programs, but no all programs are scripts.
- Scripts allows the automation of repetitive tasks and the creation of pipelines.
- However some tasks require user interaction or checks and thus it cannot be easily “scripted”.



FastQC Quality Control
Output URL
Read FASTQ Files with Reads 1
Gets paths
Output URL
Filter Reads by CASAVA Header
Filters FASTQ reads from Read FASTQ Files with Reads 1
Output File
Cut Adapter
Removes adapter sequences from Filter Reads by CASAVA Header.
Output File
Trim Reads by Quality
Trim input sequence Cut Adapter from the end, using the quality threshold.
Output File
FastQC Quality Control
Output File
Map Reads with BWA-MEM
Aligns upstream oriented reads from Trim Reads by Quality and downstream oriented reads from Trim Reads by
Output File
Read FASTQ Files with Reads 2
Gets paths of files:
Output URL
Filter Reads by CASAVA Header
Filters FASTQ reads from Read FASTQ Files with Reads 2
Output File
Cut Adapter
Removes adapter sequences from Filter Reads by CASAVA Header.
Output File
Trim Reads by Quality
Trim input sequence Cut Adapter from the end, using the quality threshold.
Output File
FastQC Quality Control
Output data
Output URL
FastQC Quality Control
Filter Merged BAM File with SAMtools
Filter BAM/SAM files from Merge Files with SAMTools view.
Merged BAM File
Merge Files
File
Convert to BAM
Filtered BAM/SAM files
Sort BAM File
Sorted BAM File
Remove Duplicates
Cleaned BAM File
Sort BAM File
Sorted BAM File
FastQC Quality Control

BASH Script file

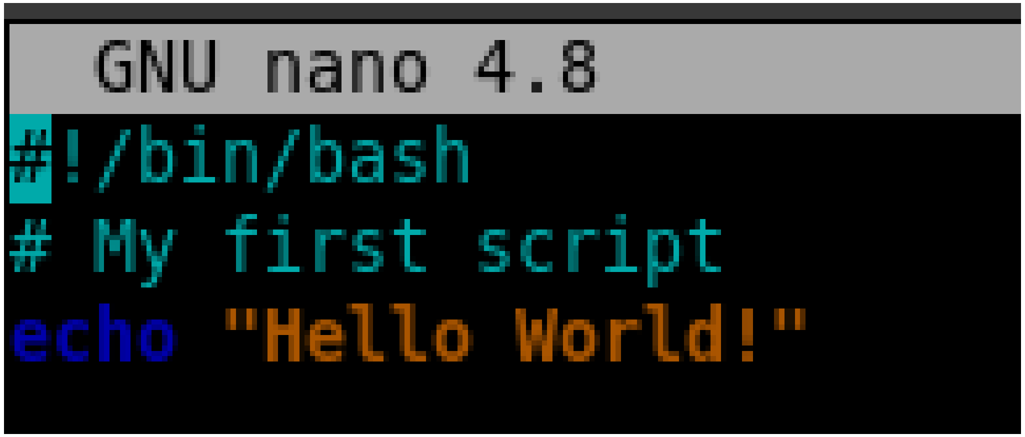


```
#!/bin/bash

# My first script

echo "Hello World!"
```

Shebang: states the program for which the script was written for

Hash: the line is a comment. Used for telling you what the script does or is doing.

The **command** that will print *Hello World!* in the screen

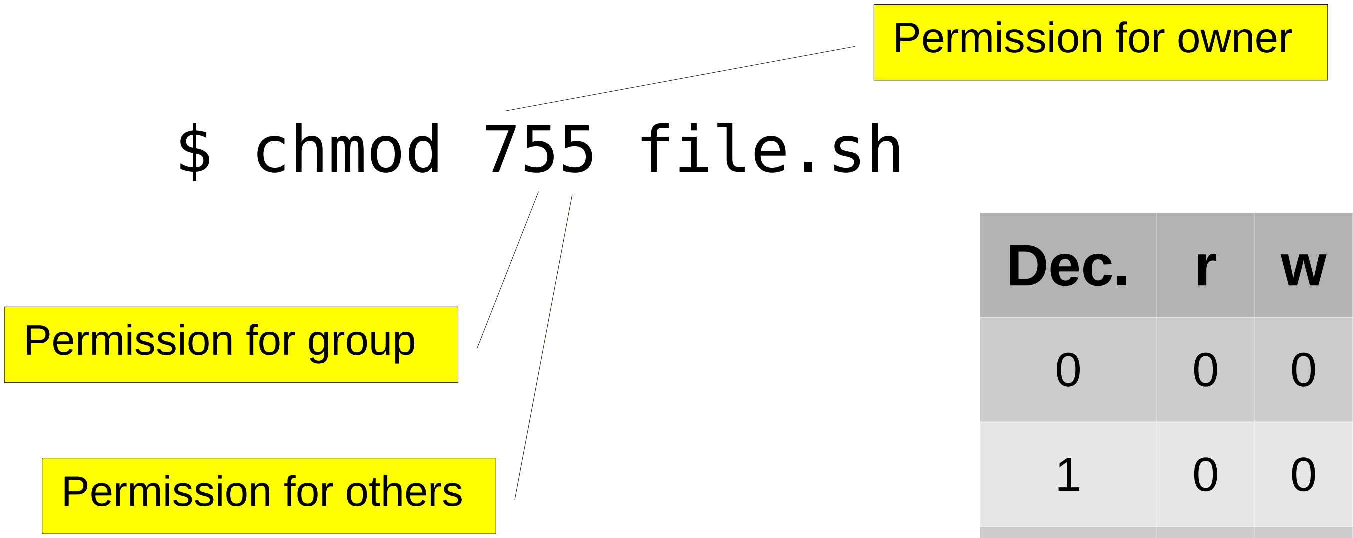


```
  GNU nano 4.8
#!/bin/bash
# My first script
echo "Hello World!"
```

- Command interpreters (as BASH) read the scripts from a **simple text** file.
- Some text editors could highlight known commands.
- To be executed the text file require *execution permission.*

Just to remember... file permissions

- **R**ead: the user can see what is inside the file
- **W**rite: the user can change (or delete the file)
- E**x**ecute: the user can execute the file or cd into the directory



```
$ chmod 755 file.sh
```

Permission for owner

Permission for group

Permission for others

```
$ chmod +x file.sh
```

?

```
$ chmod -r file.sh
```

Dec.	r	w	x
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Hands-on contents

- Variable setting and string manipulation
  - Definition and use
  - Concatenation
  - Sub-strings by position
  - Sub-strings by match
- Condition statement (flow control)
  - Single condition
  - Multiple conditions (Boolean operators)
- Loops
  - For loop

Other options available at *man bash* and:
https://tldp.org/LDP/abs/html/string-manipulation.html

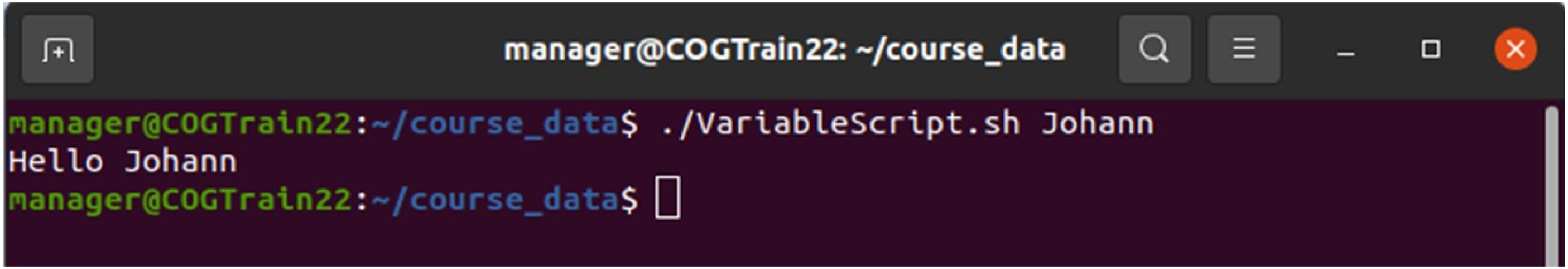
Variables

- The variables allows you to assign a name to a value that can be referred later in the script. Also, it allows you to pass information to a script so you don’t have to edit it to change target file names or options
- The variable name can include any letter or number or _
- They are CASE SENSITIVE so myvar and MyVar are different variables.
- The values are assigned with “=” sign
- After assignation they are accessed by using a $ before the name

VariableScript.sh example

```
#!/bin/bash

# My script using variable

myname=$1

echo "Hello $myname"
```

- Variable definition (without “$”), no spaces after nor before the = sing.
- Another variable that refers to the first command line argument ($1)
- Variable referred in a command (with “$”)

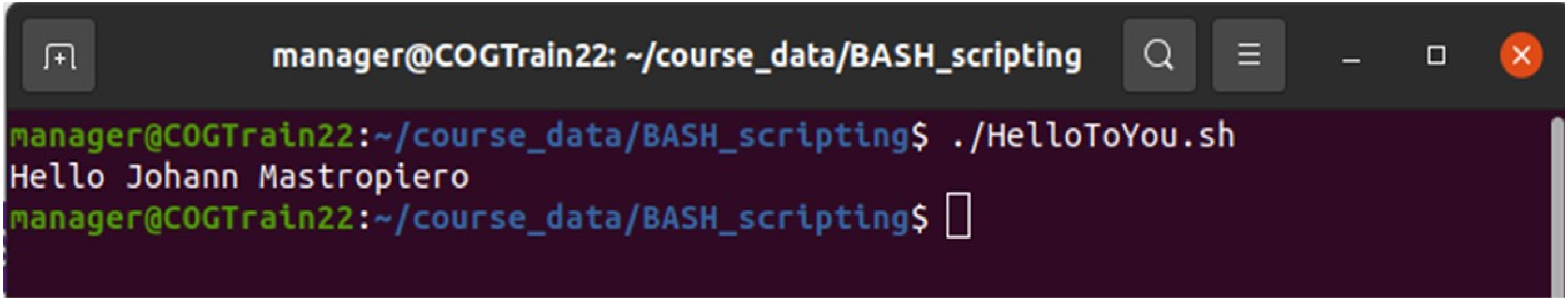


```
manager@COGTrain22:~/course_data$ ./VariableScript.sh Johann
Hello Johann
manager@COGTrain22:~/course_data$ 
```

HelloToYou.sh example

- Strings can be joined (**concatenated**) just by referring one after other.
- Note that the space within $a and $b is also included in $c

```
#!/bin/bash

a="Johann"

b="Mastropiero"

c="$a $b"

echo "Hello $c"
```



```
manager@COGTrain22:~/course_data/BASH_scripting$ ./HelloToYou.sh
Hello Johann Mastropiero
manager@COGTrain22:~/course_data/BASH_scripting$ 
```

Substring.sh example

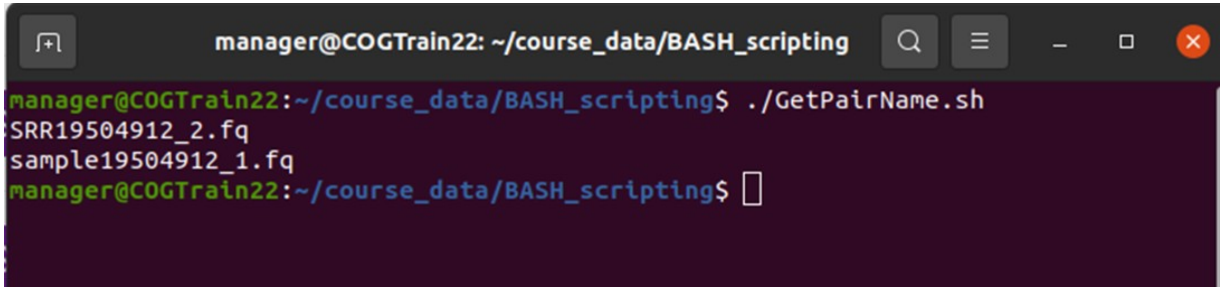
```
#!/bin/bash
filename="SRR19504912_1.fq"

# Print string length
echo ${#filename}

# Delete first 3 chars
beg=${filename:3}
echo $beg

# Delete first 3 chars and
# print 7 chars
mid=${filename:3:7}
echo $mid

# Print last 5 chars
end=${filename: -5}
echo $end
```

- The length of the string can be retrieved with **${#*var*}** (where “var” is the variable name)
- A string can be truncated an arbitrary number of characters from the beginning (left to right) with **${*var*:*L*}** (where “var” is the variable name, and “L” is the length of the truncated string)
- A part of a string can be retrieved using **${*var*:S:L}** (where “var is the variable name, “S” the start position and “L” the length of the substring)
- Finally a string can be truncated counting from the last character (right to left) with **${*var*: -*L*}** (where “var” is the variable name, and “L” the length of the substring; beware of the space between “:” and “-”)

GetPairName.sh example

- A substring can be deleted by it match from left to right with **${*var*#substring}**
- Conversely, a substring can be eleted by it match from right to left with ***${var%substring}***
- In both cases “var” is the variable name and “substring” is the text to match. Substring may contain a wilcard “*” to mach any text

```
#!/bin/bash

filename1="SRR19504912_1.fq"

filename2=${filename1%_1.fq}_2.fq

echo $filename2

sample1=sample${filename1#SRR}

echo $sample1
```



```
manager@COGTrain22:~/course_data/BASH_scripting$ ./GetPairName.sh
SRR19504912_2.fq
sample19504912_1.fq
manager@COGTrain22:~/course_data/BASH_scripting$ 
```

Breakout rooms #1

- **Exercise 1**: Write a SecondScript.sh that lists (ls) the files in your directory

- **Exercise 2**: Write a CountScript.sh that counts the lines (wc –l) in the file SRR19504912_1.fastq present in /home/manager/course_data/NGS_file_formats_and_QC

- **Exercise 3**: Modify your SecondScript.sh so that it lists the files in any specified directory as the input to the script.
  The command line execution would look like:
  `SecondScript.sh /path/to/a/directory`

- **Exercise 4**: Modify your CountScript.sh so that it counts the lines in any specified file that is the input to the script.
  The command line execution would look like:
  `CountScript.sh /path/to/a/file`

- **Exercise 5**: Modify the HelloToYou.sh script so that it takes two arguments (your first name as $1 and surname as $2) from the command line.
  Command line execution would be:
  `HelloToYou.sh Johann Mastropiero`

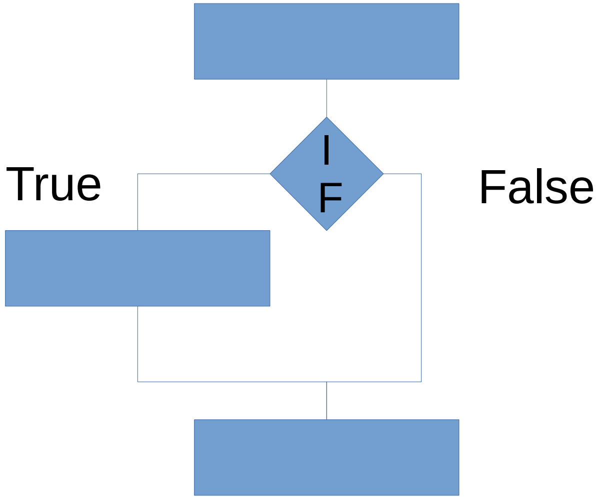
- **Exercise 6**: Modify your CountScript.sh file so that it takes the pair of files SRR19504912_1.fastq and SRR19504912_2.fastq (/home/manager/course_data/NGS_file_formats_and_QC) as input and outputs the number of lines in each file.

- **Exercise 7**: Modify the GetPairName.sh script so the user can provide any file name as input to the script.

Condition statement: if

- Allows to execute part of the script if a certain condition is met. The condition is a Boolean expression (or zero for false and non-zero for true). Complex expressions could be created with Boolean operators as “OR”, “AND” and “NOT” (“||”, “&&”, “!” respectively)

```
if [ EXPRESSION ]; then
ACTION
fi

if [ EXPRESSION_1 ] && [ EXPRESSION_2 ]; then
ACTION
fi

if [ EXPRESSION_1 ] || [ EXPRESSION_2 ]; then
ACTION
fi
```



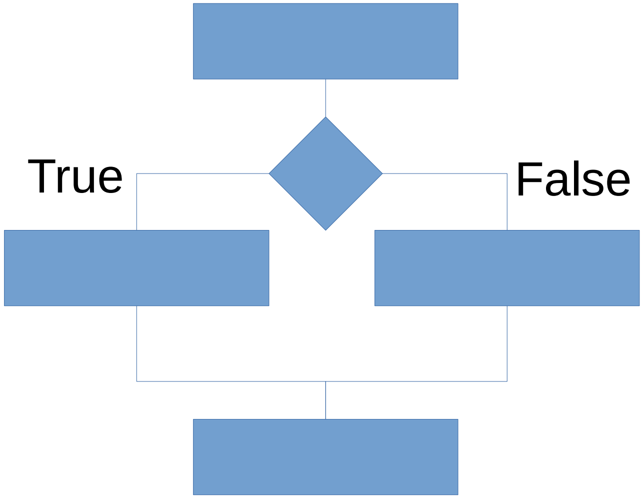
Hamlet in a script:

[2b] || [!2b]

Condition statement: if-else

- Works basically as if statement, but allows to execute a different part of the script when the original condition is not met.

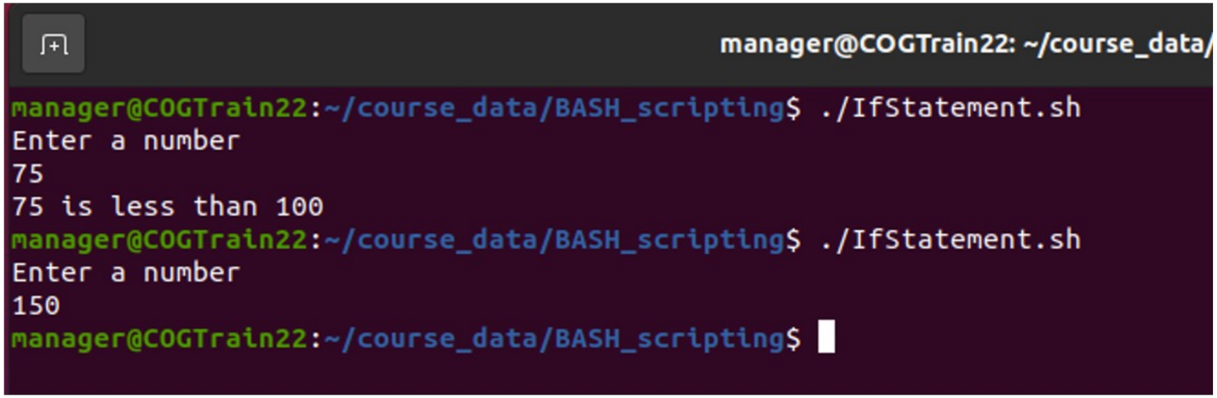
```
if [ EXPRESSION ];then
ACTION_1
else
ACTION_2
fi
```



- Action_1 will be executed if EXPRESSION is true, but Action_2 will be executed if EXPRESSION is false
- Off course, the expression could be more complex with the use of AND, OR and NOT operators.

IfStatement.sh example

```
#!/bin/bash

#Get input number from user input
echo "Enter a number"
read n

#Check if input number less than 100
if [ $n -lt 100 ]; then
  echo "$n is less than 100"
fi
```



```
manager@COGTrain22:~/course_data/BASH_scripting$ ./IfStatement.sh
Enter a number
75
75 is less than 100
manager@COGTrain22:~/course_data/BASH_scripting$ ./IfStatement.sh
Enter a number
150
manager@COGTrain22:~/course_data/BASH_scripting$ 
```

- Assigns to variable *n* whatever the users writes
- Uses the numeric test operator less than (-lt) other operators are gt, eq,le and ge for greater than, equals to, less or equals to and greater or equals to respectively.
- The output text is only written to the terminal if the user enters a number lower than 100

CheckFile.sh example

```
#!/bin/bash

# Set the path for our file

file="reference.fasta"

# Check whether file exists, is readable and has data

if [[ -e ${file} ]] && [[ -r ${file} ]] && [[ -s ${file} ]];then
    # Execute this code if file meets those conditions
    echo "File is good"
fi
```

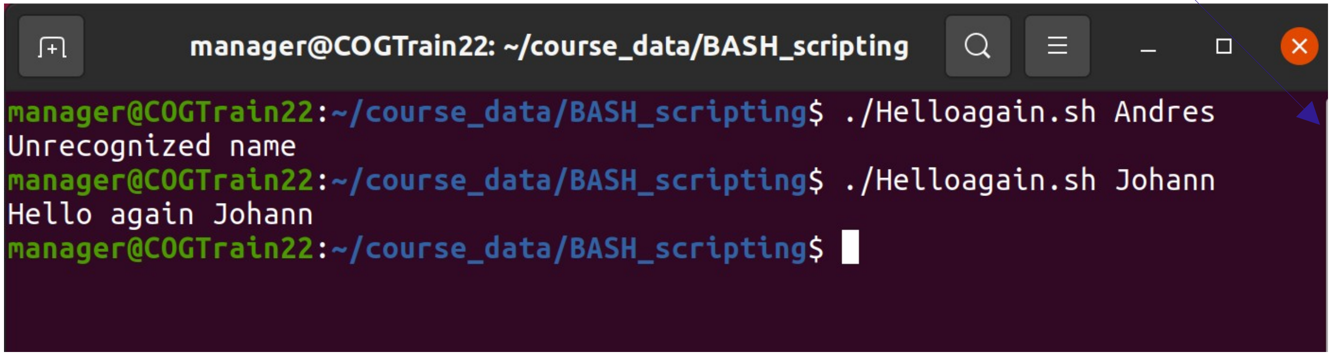
- Tests
  - -e checks if the file exists
  - -r checks if the file is redeable
  - -s checks if the file has some content
- Conditions are nested with “&&” (AND) operator, so the global expression will be true only if ALL conditions are true.

To see a complete list of available tests, use *man test* or *help test* commands.

Helloagain.sh example

```
#!/bin/bash

a=$1

if [ "$a" == "Johann" ];then
  echo "Hello again Johann"
else
  echo "Unrecognized name"
fi
```

- What does this do?
- Uses the “=” (also “==”) operator to test if one string is equal to other. Note that *-eq* is used for numerical evaluation and it will not work here. Also note the quotes around the variable “a” and the tested name Johann
- This output text is written to the terminal if the user write Johann as command line parameter.
- This output text is written if the user enter any other (or none) command line parameter



```
manager@COGTrain22:~/course_data/BASH_scripting$ ./Helloagain.sh Andres
Unrecognized name
manager@COGTrain22:~/course_data/BASH_scripting$ ./Helloagain.sh Johann
Hello again Johann
manager@COGTrain22:~/course_data/BASH_scripting$ 
```

Loops

- A loop in a program is a part of code that is executed a number of times
- BASH support several kind of loops with the commands *while*, *until* and *for*.
- We will see the *for* loop.

```
for ITEM in LIST
do
  ACTION
done
```

- The code between *do* and *done* will be executed as many times as the elements contained in *LIST*.
- These are called iterations.
- The value of the variable *ITEM* will be an element of the list and will change each iteration.

Loop.sh example

- Create a variable called *f* that will contain an element of the list “*.fastq” at each iteration.
  - Note that “*” is a wildcard character that match any string in filenames, so bash will **expand** this string to a list that contains all files in current (fastq_sets) directory which names end with “.fastq”. Therefore, the *for* command will not see any “*”, instead it will see a list of filenames.
- The *do* and *done* statements create a block of commands that will be executed at each iteration.
- The indentation is not needed in BASH (not the case for Python) but makes the script easier to read.
- Finally, I like to point out that “word count” (*wc*) command can read multiple files, so the one line statement **`wc -l *.fastq`** will produce a similar output.

```
#!/bin/bash

for f in *.fastq

do

  echo $f

  wc -l $f

done
```

Breakout rooms #2

- **Exercise 8**: Use your GetPairName.sh script as the base for a new one that will check with an (if) that the input file has _1.fastq (end=${filename: -8}) and only then print out the paired sample name.
- **Exercise 9**: Write a script called Loop2.sh to loop (for) through the directory fastq_sets and copy (cp) the files to your current directory.
- **Exercise 10**: Modify your Loop2.sh script so that the files are renamed from .fastq to .fq
- **Exercise 11**: Write a script that loops through the fastq_sets directory (for) and if the file has _1.fq (end=${filename: -5}), it counts the number of lines in the file (wc –l).

Sources

- Bash manpage (man bash)
- Builtin bash commands help
  - help
  - help test
  - help for
  - help if
- String manipulations: Advanced Bash-scripting guide (chapter 10): https://tldp.org/LDP/abs/html/string-manipulation.html
- WC infopage (info wc).
- Life in general… well, a lot of stack-overflow threads.
- Test and error (mostly with quotations)