TECNICHE DI RAPPRESENTAZIONE E MODELLIZZAZIONE DEI DATI

— Part 1 —

(2 CFU out of 6 total CFU)

Link moodle: https://moodle2.units.it/course/view.php?id=14486

Codice Teams del corso: d2cmkh8

Shell scripting

Expansions. They come in different flavours:

Brace expansion: mechanism by which arbitrary strings may be generated

```
(base) milena:~ milenavalentini$ echo a{d,c,b}e
ade ace abe
```

Tilde expansion: the unquoted ~ character is usually replaced with the value of the home shell variable.

```
(base) milena:WorkingOn milenavalentini$ ls ~/Downloads/
```

Command substitution: allows the output of a command to replace the command itself. It occurs when:

```
$(command)
```

Arithmetic expansion: allows the evaluation of the expression and the substitution of the result. Format:

```
(( expression ))
```

Filename expansion: After word splitting, unless the -f option has been set, Bash scans each word for the characters ‘ * ’, ‘ ? ’, and ‘ [’. If one of these characters appears (and is not quoted), then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of filenames matching the pattern.

Shell scripting

Expansions. They come in different flavours:

Filename expansion: After word splitting, unless the -f option has been set, Bash scans each word for the characters ‘ * ’, ‘ ? ’, and ‘ [’. If one of these characters appears (and is not quoted), then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of filenames matching the pattern.

```
(base) milena:test_bash milenavalentini$ ls
file_100.txt    file_101.txt    file_102.txt    file_103.txt    script.bash
(base) milena:test_bash milenavalentini$
(base) milena:test_bash milenavalentini$ cat script.bash
#!/bin/bash

for a in file_*; do
  echo "$a"
done

(base) milena:test_bash milenavalentini$ ./script.bash
file_100.txt
file_101.txt
file_102.txt
file_103.txt
(base) milena:test_bash milenavalentini$ _
```

Shell scripting

Expansions. They come in different flavours:

Brace expansion: mechanism by which arbitrary strings may be generated

```
(base) milena:~ milenavalentini$ echo a{d,c,b}e
ade ace abe
```

Tilde expansion: the unquoted ~ character is usually replaced with the value of the home shell variable.

```
(base) milena:WorkingOn milenavalentini$ ls ~/Downloads/
```

Command substitution: allows the output of a command to replace the command itself. It occurs when:

```
$(command)
```

Arithmetic expansion: allows the evaluation of the expression and the substitution of the result. Format:

```
(( expression ))
```

Filename expansion: characters ‘ * ’, ‘ ? ’, and ‘ [’ that can be regarded as a *pattern*.

And also: shell parameter expansions, word splitting, …

Shell scripting

Exercises.

Exercise 1
Create a new directory, move into it and run the command

touch file{1..20}{.{dat,png,txt},\ backup.dat,_bkp.png}

a. — Understand what happened using ls.
b. — List only files with the .dat extension.
c. — List only files with number 13 in the name.
d. — List only backup files.
e. — List only files containing a space in the name.
f. — List files with a number that is multiple of 5 before the dot.

Write a bash script which performs all the tasks above, and execute it.

Make the script write the commands and the output of the commands on a file.

Make the script available as a command.

File Content Search Commands

To look for a pattern in one or more files, use the grep series of commands.

The **grep** commands search for a string in the specified files and display the output on standard output.
egrep (the initial e stands for extended) searches for a specified pattern in one or more files. The pattern can be a regular expression to match any single character.

* to match one or more single characters that precede the asterisk.

^ to match the regular expression at the beginning of a line.

$ to match the regular expression at the end of a line.

+ to match one or more occurrences of a preceding regular expression.

? to match zero or more occurrences of a preceding regular expression.

[ ] to match any of the characters specified within the brackets.

File Content Search Command **egrep**

Assume we have a file:

```
(base) milena:test_bash milenavalentini$ cat file_to_grep.txt
-- This is a test file
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.

---- This was a TEST FILE
```

File Content Search Command **egrep**

```
(base) milena:test_bash milenavalentini$ cat file_to_grep.txt
-- This is a test file
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.

---- This was a TEST FILE
```

To find all occurrences of *file*, use:

```
(base) milena:test_bash milenavalentini$ egrep file file_to_grep.txt
-- This is a test file
(base) milena:test_bash milenavalentini$ egrep FILE file_to_grep.txt
---- This was a TEST FILE
(base) milena:test_bash milenavalentini$ egrep -i file file_to_grep.txt
-- This is a test file
---- This was a TEST FILE
```

(flag **-i** for case insensitivity)

File Content Search Command **egrep**

```
(base) milena:test_bash milenavalentini$ cat file_to_grep.txt
-- This is a test file
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.

---- This was a TEST FILE
```

To display the relative line number of the line containing the searched pattern, use the **-n** flag:

```
(base) milena:test_bash milenavalentini$ egrep -i -n file file_to_grep.txt
1:-- This is a test file
12:---- This was a TEST FILE
(base) milena:test_bash milenavalentini$ egrep -n ut file_to_grep.txt
4:ut labore et dolore magna aliqua. Ut enim ad minim veniam,
6:ut aliquip ex ea commodo consequat.
7:Duis aute irure dolor in reprehenderit in voluptate velit esse
```

File Content Search Command **egrep**

```
(base) milena:test_bash milenavalentini$ cat file_to_grep.txt
-- This is a test file
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.

---- This was a TEST FILE
```

To search for a pattern as a word, use the **-w** flag:

```
(base) milena:test_bash milenavalentini$ egrep -w in file_to_grep.txt
Duis aute irure dolor in reprehenderit in voluptate velit esse
sunt in culpa qui officia deserunt mollit anim id est laborum.
```

Shell scripting

File Content Search Command **egrep**

```
(base) milena:test_bash milenavalentini$ cat file_to_grep.txt
-- This is a test file
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.

---- This was a TEST FILE
```

To find the total number of lines in which the specified pattern occurs, use the **-c** flag:

```
(base) milena:test_bash milenavalentini$ egrep -i -c test file_to_grep.txt
2
```

Shell scripting

File Content Search Command **egrep**

```
(base) milena:test_bash milenavalentini$ cat file_to_grep.txt
-- This is a test file
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.

---- This was a TEST FILE
```

To get a list of all the lines that do not contain the specified pattern, use the **-v** flag:

```
$ egrep -i -v test file_to_grep.txt
```

```
(base) milena:test_bash milenavalentini$ egrep -i -v test file_to_grep.txt
Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident,
sunt in culpa qui officia deserunt mollit anim id est laborum.
```

Shell scripting

Exercise with the command **egrep**

Create a file and write (copy and paste is OK) in it several lines of text, making sure to go often to new line. Repeat some lines more than once.

Become familiar with the following commands:

egrep *flags* pattern_being_searched file_where_to_search

with the following flags: **-i -n -v -w -c**

and with some combinations of the flags above.

Shell scripting

Variables. They are parameters with a name.

They are created and updated by the user, and available in the environment**.**

They are untyped.

The variable name is called identifier: it’s a word consisting only of letters, digits and underscores and beginning with a letter or an underscore.

Shell scripting

Variables. They are parameters with a name.

They are created and updated by the user, and available in the environment**.**

They are untyped.

No need to declare a variable, just assigning a value to its reference will create it.

```
(base) milena:test_bash milenavalentini$ STR="Hello World"
(base) milena:test_bash milenavalentini$ echo $STR
Hello World
(base) milena:test_bash milenavalentini$ a=5324; printf $a\\n
5324
```

variableName=variableContent

If not existing, the global variable **variableName** is created, and the content variableContent is put into it

If existing, the content of **variableName** is set to variableContent

If **variableName** exists and it is read-only, an error occurs.

Shell scripting

Variables. They are parameters with a name.

They are created and updated by the user, and available in the environment**.**

They are untyped.

No need to declare a variable, just assigning a value to its reference will create it.

variableName=variableContent

If not existing, the global variable **variableName** is created, and the content variableContent is put into it

If existing, the content of **variableName** is set to variableContent

If **variableName** exists and it is read-only, an error occurs.

```
(base) milena:test_bash milenavalentini$ variableName=variableContent
(base) milena:test_bash milenavalentini$ variableName = variableContent
-bash: variableName: command not found
```

Bash arithmetic expansion

Arithmetic expansion provides a powerful tool to perform (integer) arithmetic operation.

Translating a string into a numerical expression can be done using:

$ (…)

double parentheses, or

let

Bash arithmetic expansion

Arithmetic expansion provides a powerful tool to perform (integer) arithmetic operation.

Translating a string into a numerical expression can be done using:

$ (...)

double parentheses, or

let

example:

```
(base) milena:test_bash milenavalentini$ z=15
(base) milena:test_bash milenavalentini$ z=$(expr $z + 3)
(base) milena:test_bash milenavalentini$ echo $z
18
```

arithmetic operators:

```
(base) milena:test_bash milenavalentini$ a=$(expr 5 + 3)
(base) milena:test_bash milenavalentini$ echo "5 + 3 = $a"
5 + 3 = 8
```

Bash arithmetic expansion

Arithmetic expansion provides a powerful tool to perform (integer) arithmetic operation.

Translating a string into a numerical expression can be done using:

$ (…)

double parentheses, or

let

incrementing a variable:

```
(base) milena:test_bash milenavalentini$ a=5
(base) milena:test_bash milenavalentini$ a=$(expr $a + 1)
(base) milena:test_bash milenavalentini$ echo "a + 1 = $a"
a + 1 = 6
```

modulo (i.e. remainder of a division):

```
(base) milena:test_bash milenavalentini$ b=$(expr 5 % 3)
(base) milena:test_bash milenavalentini$ echo "5 mod 3 = $b"
5 mod 3 = 2
```

Bash arithmetic expansion

Arithmetic expansion provides a powerful tool to perform (integer) arithmetic operation.

Translating a string into a numerical expression can be done using:

$ (…)

double parentheses, or

let

logical operators (return 1 if true):

```
(base) milena:test_bash milenavalentini$ x=10
(base) milena:test_bash milenavalentini$ y=11
(base) milena:test_bash milenavalentini$ b=$(expr $x = $y)
(base) milena:test_bash milenavalentini$ echo "b = $b"
b = 0
```

verifies

it’s false!

Bash arithmetic expansion

Arithmetic expansion provides a powerful tool to perform (integer) arithmetic operation.

Translating a string into a numerical expression can be done using:

$ (…)

double parentheses or

let

arithmetic expansion:

```
(base) milena:test_bash milenavalentini$ n=0
(base) milena:test_bash milenavalentini$ echo "n = $n"
n = 0
(base) milena:test_bash milenavalentini$ (( n += 1 ))
(base) milena:test_bash milenavalentini$ echo "n = $n"
n = 1
```

Bash arithmetic expansion

Arithmetic expansion provides a powerful tool to perform (integer) arithmetic operation.

Translating a string into a numerical expression can be done using:

$ (…)

double parentheses, or

let

The **let** operator actually performs arithmetic evaluation, rather than expansion.

```
(base) milena:test_bash milenavalentini$ z=1
(base) milena:test_bash milenavalentini$ let z=z+5
(base) milena:test_bash milenavalentini$ echo $z
6
(base) milena:test_bash milenavalentini$ let "z += 5"
(base) milena:test_bash milenavalentini$ echo $z
11
```

Double quotes allow you using spaces in variable assignment!

Exercise: try to translate examples done with expr and double parentheses by using let.

Redirection with pipe

Pipe `|` is a special character.

It passes the output (stdout) of a previous command to the input (stdin) of the next one, or to the shell.

It’s a method to chain commands together.

Redirection with pipe

Pipe | is a special character.

It passes the output (stdout) of a previous command to the input (stdin) of the next one, or to the shell.

It's a method to chain commands together.

Example:

```
(base) milena:test_bash milenavalentini$ ls -l file*4*
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file14 backup.dat
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file14.dat
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file14.png
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file14.txt
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file14_bkp.png
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file4 backup.dat
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file4.dat
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file4.png
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file4.txt
-rw-r--r--  1 milenavalentini  staff  0 Oct  4 13:42 file4_bkp.png
(base) milena:test_bash milenavalentini$ ls -l file*4* | wc -l
      10
(base) milena:test_bash milenavalentini$ _
```

ls -l # Passes the output of "ls -l" to the wc -l, the second command, which counts files.

Conditional blocks

```
#!/bin/bash
if [ "foo" = "foo" ]; then
 echo expression evaluated as true
else
 echo expression evaluated as false
fi
```

if is a keyword

Example of its syntax

if executes a command (or a set of commands)
and checks that command exit code
to see whether it was successful or not

command exit code = 0 means success

Conditional blocks

```
#!/bin/bash
if [ "foo" = "foo" ]; then
 echo expression evaluated as true
else
 echo expression evaluated as false
fi
```

```
#!/bin/bash
if [ "foo" = "foo" ]
then
 echo expression evaluated as true
else
 echo expression evaluated as false
fi
```

Different layouts work.

Conditional blocks

```
#!/bin/bash
if [ "foo" = "foo" ]; then
 echo expression evaluated as true
else
 echo expression evaluated as false
fi
```

```
#!/bin/bash
a=2
if [ $a == 3 ]; then
 echo equal to
elif [ $a -gt 3 ]; then
 echo greater than
else
 echo lower than
fi
```

Conditional blocks

Main conditional operators

Relational operators

- -lt (<)
- -gt (>)
- -le (<=)
- -ge (>=)
- -eq (==)
- -ne (!=)

Boolean operators

- && and
- || or
- | not

Files operators:

- if [-x "$filename"]; then # if filename is executable
- if [-e "$filename"]; then # if filename exists
- ……

Conditional loops

while Repeat as long as a command is executed successfully (exit code is 0)

until Repeat as long as a command is executed unsuccessfully (exit code is not 0).
Not that used, while is usually preferred.

for It comes in two versions

- to iterate over an integer index
- to iterate over a list

Conditional loops

The keywords **while** and **until**

```
while COMMAND; do
    # Body of the loop: entered if COMMAND's exit code = 0
done

until COMMAND; do
    # Body of the loop: entered if COMMAND's exit code ≠0
done
```

- Testing command like **[** or **[[** are often used
- Infinite loops can be achieved using the builtins **true**, **false** and **:**
- Use the **continue** builtin to skip ahead to the next iteration of a loop without executing the rest of the body
- Use the **break** builtin to jump out of the loop and continue with the script after it
- Both **continue** and **break** accept an optional integer to act on nested loops

Conditional loops

The keywords **while** and **until**

```
$ until false; do   # while true; do   # while :; do
>   echo "Infinite loop"
>   sleep 3
> done
Infinite loop
Infinite loop
^C    # Press CTRL-C
```

```
# An example of countdown...
$ deadline=$(date -d "8 seconds" +'%s'); \
> now=$(date +'%s'); \
> while [[ $((deadline - now)) -gt 0 ]]; do
>   echo "$((deadline - now)) seconds to BOOM!"
>   sleep 3
>   now=$(date +'%s')
> done; echo 'BOOOOOM!!'
8 seconds to BOOM!
5 seconds to BOOM!
2 seconds to BOOM!
BOOOOOM!!
```

Conditional loops

The keywords **while** and **until**

```
$ i=0; \
> while [[ ${i} -lt 5 ]]; do
>   j=0
>   while [[ ${j} -le 5 ]]; do
>     if [[ ${j} -le ${i} ]]; then
>       printf "$((i+j)) "
>       j=$((j + 1))
>     else
>       printf '\n'
>       i=$((i + 1))
>       continue 2
>     fi
>   done
> done
0
1 2
2 3 4
3 4 5 6
4 5 6 7 8
```

Conditional loops

The keyword **for**

```
for VARIABLE in WORDS; do
    # Body of the loop: VARIABLE set to WORD
done

for (( EXPR1; EXPR2; EXPR3 )); do # Expressions can be empty
    # Body of the loop
done
```

In the second form:

- it starts by evaluating the first arithmetic expression
- it repeats as long as the second arithmetic expression is successful
- at the end of each loop evaluates the third arithmetic expression

Conditional loops

The keyword **for**

```
$ for (( ; 1; )); do echo "Infinite loop"; sleep 1; done
Infinite loop
Infinite loop
^C    # Press CTRL-C
```

```
$ for index in {0,1}{0,1}; do
>   echo "${index} in base 2 is $(( 2#${index})) in base 10"
> done; unset index
00 in base 2 is 0 in base 10
01 in base 2 is 1 in base 10
10 in base 2 is 2 in base 10
11 in base 2 is 3 in base 10
```

```
# BAD code!

$ for file in $(ls *.mp3); do    # AAAARGH!
>   rm "$file"
> done; unset file
```

```
$ ls
Happy birthday.mp3   Hello.mp3
$ for file in *.mp3; do    # GOOD code
>   rm "$file"
> done; unset file
```

Conditional loops

continue statement in a for loop

- **continue** stops the execution of the commands in the loop and jumps to the next value in the series.

Pseudo-code example:

```
for i in [series]
do
   command 1
   command 2
   if (condition)          # Condition to jump over command 3
         continue          # skip to the next value in "series"
   fi
   command 3
done
```

continue statement in iteration

- **continue** is used in scripts to skip the current iteration of a loop and continue to the next iteration of the loop.

Conditional loops

break command in iteration

break: used to exit out of the current loop before the actual end of the loop

When used in scripts with **multiple loops**, we use break 2 if we are in an inner loop and we want to jump out of the outer loop

```
#!/bin/bash
# Breaking outer loop from inner loop
for (( a = 1; a < 5; a++ ))
do
echo “outer loop: $a”
for (( b = 1; b < 100; b++ ))
do
if [ $b –gt 4 ]
then
break 2
fi
echo “Inner loop: $b ”
done
done
```

Conditional loops

Examples

Counting:

```
#!/bin/bash
for i in {1..25}
do
   echo $i
done
```

or:

```
#!/bin/bash
for ((i=1;i<=25;i+=1)
do
   echo $i
done
```

Counting on "n" steps

```
#!/bin/bash
for i in {0..25..5}
do
   echo $i
done
```

That will count with 5 to 5 steps.

Counting backwards

```
#!/bin/bash
for i in {25..0..-5}
do
   echo $i
done
```

Acting on files

```
#!/bin/bash
for file in ~/*.txt
do
echo $file
done
```

That example will just list all files with "txt" extension. It is the same as ls *.txt

Calculate prime numbers

```
#!/bin/bash
read -p "How many prime
numbers ?: " num
c=0
k=0
n=2

numero=$[$num-1]
while [ $k -ne $num ]; do
    for i in `seq 1 $n`;do
      r=$[$n%$i]
            if [ $r -eq 0 ]; then
                  c=$[$c+1]
            fi
    done

     if [ $c -eq 2 ]; then
           echo "$i"
           k=$[$k+1]
      fi
      n=$[$n+1]
      c=0
done
```

Conditional loops

Examples

with some versions of the bash shell or with some bash shells on Mac OS these lines may not produce the expected outcome. Try instead:

```
#!/bin/bash
for ((i=0; i<=25; i+=5))
do
    echo $i
done
```

Counting:

```
#!/bin/bash
for i in {1..25}
do
   echo $i
done
```

or:

```
#!/bin/bash
for ((i=1;i<=25;i+=1)
do
   echo $i
done
```

Counting on "n" steps

```
#!/bin/bash
for i in {0..25..5}
do
   echo $i
done
```

That will count with 5 to 5 steps.

Counting backwards

```
#!/bin/bash
for i in {25..0..-5}
do
   echo $i
done
```

Acting on files

```
#!/bin/bash
for file in ~/*.txt
do
echo $file
done
```

That example will just list all files with "txt" extension. It is the same as ls *.txt

Calculate prime numbers

```
#!/bin/bash
read -p "How many prime
numbers ?: " num
c=0
k=0
n=2

numero=$[$num-1]
while [ $k -ne $num ]; do
    for i in `seq 1 $n`;do
      r=$[$n%$i]
            if [ $r -eq 0 ]; then
                  c=$[$c+1]
            fi
    done

     if [ $c -eq 2 ]; then
           echo "$i"
           k=$[$k+1]
      fi
      n=$[$n+1]
      c=0
done
```

Conditional loops

Read files

```
read -r riga < $file
echo $riga
```

```
#!/bin/bash
while IFS='' read -r line || [[ -n "$line" ]];
do
    echo "Text read from file: $line"
done < "$1"
```

```
for word in $(cat $file); do
  echo $line
done
```

Conditional loops

Conditional statements: files and dir

```
If [ -e $file_name]; then
   echo exists
fi
```

```
If [ -d $file_name]; then
   echo is a dir
fi
```

```
If [ -f $file_name]; then
   echo exists and is a file
fi
```

```
If [ !-d $file_name]; then
   echo is not a dir
fi
```

Sed

https://www.gnu.org/software/sed/manual/sed.html

Sed is a non interactive editor. https://www.gnu.org/software/sed/manual/sed.html

It is generally used to parse and transform text, using a simple, compact programming language.

It allows to modify a file using scripts with instructions for sed editing plus the filename.

Example of string substitution:

$sed 's/old_text/new_text/g' /tmp/testfile

Sed substitute the string 'old_text' with the string 'new_text' reading from file /tmp/testfile.

The result is redirected to stdout, but it can be redirected also to a file using '>'

$sed 12, 18d /tmp/testfile

Sed displays all the rows from 12 to 18. The original file is not modified by this command, but if you redirect stdout on a new file, it is different from the original one.

Awk

https://www.gnu.org/software/gawk/manual/gawk.html

Awk matches a string on the base of a regular expression and execute a required action:

Create a file filetext as follow:

```
cat << EOF > filetext
test123
test
Tteesstt
EOF
```

```
$awk '/test/ {print}' /tmp/filetext
test123
test
```

The regular expression requires to match the string ‘test'

The required action is to ‘print' how many times the string ‘test’ is found.

```
$awk '/test/ {i=i+1} END {print i}' /tmp/filetext
3
```

Conditional loops

Exercises

Exercise 1.
Write a bash script which lists all the items in the current directory.

Exercise 2.
Write a bash script which displays all the numbers ranging from 1 to 10.

Exercise 3.
Write a bash script which prints in the standard output the values of the index i, which ranges from 0 to 20 with an increment of 2.

Conditional loops

Exercises

Exercise 4.
Write a bash script which outputs the value of a counter (initially set to 0) as long as it is smaller than 10.

Exercise 5.
Write a bash script which displays the value of an index (set to 20 at the beginning) until it drops below 10.

Sed, Awk and more

Exercise

Generate two set of files: file1.dat, file2.dat, …, file20.dat and file1.txt, file2.txt, …, file20.txt

Write in each of them the following test:

test123 test12 test1

1 2 3 4

Create one file by concatenating all the files having the extension “.dat” in a file_TOT.dat

Count in how many lines the string test12 occurs in the file_TOT.dat and print this number in the standard output

Write in a file called extracted_file.dat only the line including words of the file_TOT.dat

Python

What's Python?

It's a **high-level** programming language closer to human thinking
than to details of the machine behaviour

It's an **interpreted** language you need a compiler/interpreter to translate
this kind of programming language into a machine code

Python

What's Python?

It's a **high-level** programming language — closer to human thinking than to details of the machine behaviour

It's an **interpreted** language — you need a compiler/interpreter to translate this kind of programming language into a machine code

Why Python?

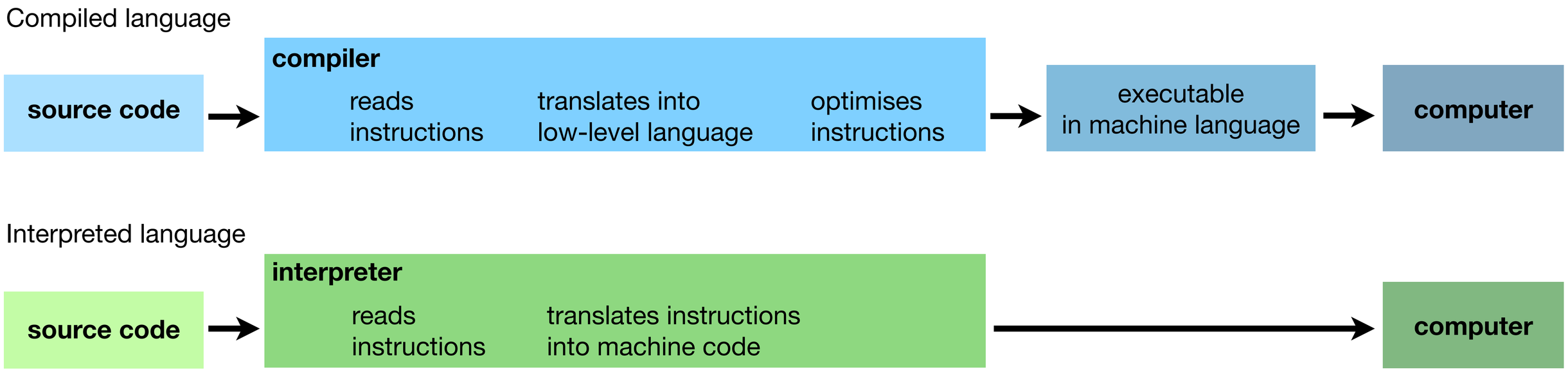
- Human-readable and close to human thinking
- Open source
- Developed by a community effort
- Contribution from users encouraged

Python

What's Python?

It's a **high-level** programming language — closer to human thinking than to details of the machine behaviour

It's an **interpreted** language — you need a compiler/interpreter to translate this kind of programming language into a machine code



Python

Language

Code

Natural language

Formal language

Structure

Syntax Set of rules which determines how a program is written and interpreted

Programming language

Python

Programming language

Quite strict

Instructions (statements) are interpreted (parsed).
To be understood they must be formally correct and only use the expected language constituents (token).

Formal language

Unique meaning independent on the context.

Syntax

Semantics Meaning of an instruction whose syntax is correct

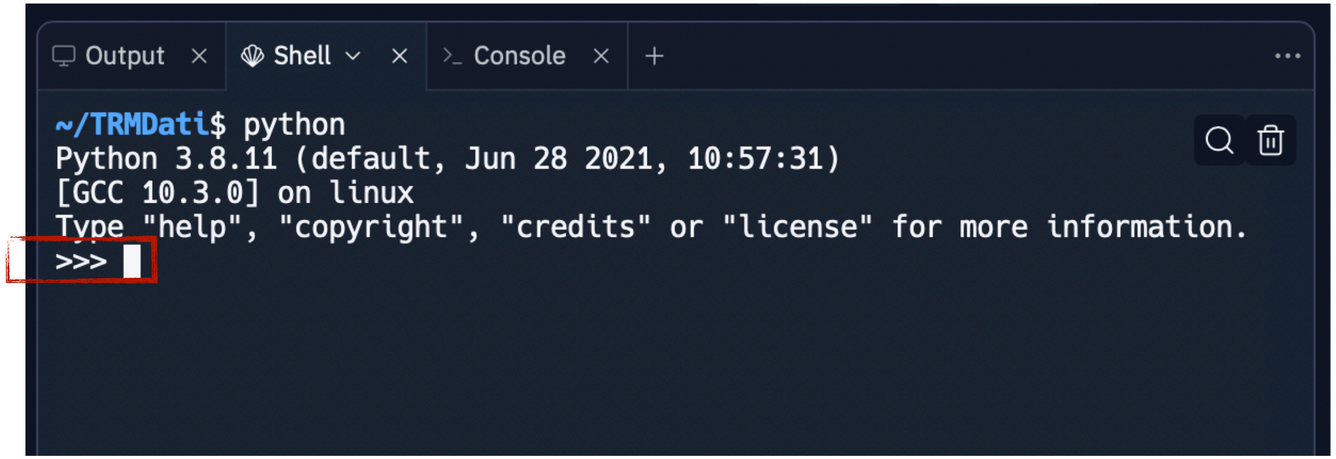
Python

Programming language

Can be used in two ways:

- interactively: the interpreter is given instructions directly, one by one
- with scripts: the interpreter is provided with a set of instructions in a text file

Different versions, use Python > 3.7



On replit shell, type
python
to launch the interpreter

Python errors

Errors

- Syntax errors
- Runtime errors
- Semantic errors

If the syntax of the instruction is not correct, Python returns a syntax error

```
>>> 1 + 5&
  File "<stdin>", line 1
    1 + 5&
         ^
SyntaxError: invalid syntax
>>>
```

Python errors

Errors

- Syntax errors
- Runtime errors
- Semantic errors

If the syntax of the instruction is not correct, Python returns a syntax error

Python returns a runtime error if something goes wrong while executing an instruction

```
>>> 2 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Python errors

Errors

- Syntax errors
- Runtime errors
- Semantic errors

If the syntax of the instruction is not correct, Python returns a syntax error

Python returns a runtime error if something goes wrong while executing an instruction

If semantic errors are there, Python does not return what you expect
(likely without issues during runtime)

Python scripts

Program/script: set of instructions in a given order that tells the interpreter how to compute or perform something

Types of instructions:

- Input
- Computation
- Condition check
- Iterate/repeat
- Output

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

```
>>> a = 1 + 2
>>>
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

For a multi-line statements use the character \

```
>>> a = 1 + 2 \
... + 3 + 4 \
... + 5
>>>
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

For a multi-line statements use the character \

Multi-line statements are implicitly assumed with parentheses.

```
>>> a = ( 1 + 2
... + 3 )
>>>
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token =

For a multi-line statements use the character \

Multi-line statements are implicitly assumed with parentheses.

Multiple statements can stay on the same line, divided by the character ;

```
>>> a = 1 ; b = 2
```

Python statements

Statement: instruction that the Python interpreter executes.
Before execution, each instruction is split into tokens during parsing.

Statements do not produce output/results.

Example of a statement where a variable is assigned
a value through the token **=**

Besides assignments, there are other statements, e.g., **import, while, if, for**

import allows you to import in your script instructions written in another file

```
>>> import this
```

Python comments

Comments: they describe in simple words what the source code is doing

Start with the hash character # and end with enter/new line

```
>>> # Add 2 to 1
>>> 1 + 2
3
>>>
```

Python interpreter neglects comments while executing
the set of instructions the script is made of

For multi-line comments, either start every line with # ,
or type the comment within triple quotes (“‘ comment ”’ , “““ here ”””)

Python keywords

Keywords: ensemble of reserved words that cannot be used as variable names, function names, or any other identifiers instructions

Case sensitive: apart from False, None, True, all the others do not have capital letters

```
>>> # Can I assign a value to a keyword?
>>> False = 3
  File "<stdin>", line 1
    False = 3
    ^^^^^
SyntaxError: cannot assign to False
```

To check Python keywords:

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif'
, 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or'
, 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
>>>
```

Python values

Values: data that the program uses for computation (e.g., 1, 3.5, ‘hello’)

Values have different types and can be grouped into classes.

The built-in Python function *type* returns the type of a value.

```
>>> type(1)
<class 'int'>
>>> type(3.5)
<class 'float'>
>>> type('hello')
<class 'str'>
>>>
```

integer number

float number

string of character

Python values

Values: data that the program uses for computation (e.g., 1, 3.5, ‘hello’)

Different types can do different things.

Python has the following built-in **data types**:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

Python values

Values: data that the program uses for computation (e.g., 1, 3.5, ‘hello’)

Python has the following built-in **data types**:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
Sequence Types:	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`
None Type:	`NoneType`

covered in this course

Python variables

How to access the content of a string and slice it:

```
In [10]: string = 'Information'

In [11]: type(string)
Out[11]: str

In [12]: print(string[3])
o

In [13]: print(string[-1])
n

In [14]: print(string[0:4])
Info

In [15]: print(string[3:6])
orm

In [16]: print(string[2:-2])
formati
```

i-th element of a string

from the i-th to the j-th character of a string [i, j)

Lists behave similarly.