Lecture 6: String manipulation. Arrays. Piping (`|`). sed, awk and grep

Last update: 20230630

Table of Contents

1. String manipulation

Bash offers a lot of built-in functionalities to manipulate the content of variables programmatically. Since the content of an external file can be stored in a **Bash** variable, we can to a certain extent solely with built-in **Bash** features manipulate the content of external files as well. However, performance starts to matter typically for large files, when **Linux** core utilities **sed**, **awk** and/or **grep** are more suitable. For very large files, when performance becomes critical, one needs to use the high-level programming languages, like **perl**.

String operators in **Bash** can be used only in combination with curly-brace syntax, `${Var}`, when the content of a variable is retrieved. String operators are used to manipulate the content of variables, typically in one of the following ways:

1. Remove, replace or modify a portion of variable's content that matches some patterns
2. Ensure that variable exists (i.e. that it is defined and has a non-zero value)
3. Set the default value for a variable

The generic syntax for manipulating the content of the variable is:

```
${Var/OldPattern/NewPattern}
```

or

```
${Var//OldPattern/NewPattern}
```

The first version will replace only the first occurrence of the pattern 'OldPattern' with 'NewPattern' within the string which is stored in the variable 'Var', while the second version will replace all occurrences. This is illustrated with the following code snippet:

```
Var=aaBBaa
echo ${Var/aa/CCC}
echo ${Var//aa/CCC}
```

which prints:

```
CCCBBaa
CCCBBCCC
```

It is perfectly fine to re-define the variable on the spot with the new content:

```
Var=${Var/aa/CCC}
```

The new and old patterns do not have to be hardwired, instead, they can be specified via variables:

```
Var=aaBBaa
Old=aa
New=CCC
Var=${Var/$Old/$New}
```

The curly-brace syntax interprets some characters in a special way. This is illustrated with the following examples.

Example 1: How to get programmatically the length of the string?

```
$ Var=1a3b56F8
$ echo ${#Var}
8
```

Example 2: How to lower/upper cases of all characters in the string?

```
$ Var=aBcDeF
$ echo ${Var,,}
abcdef
$ echo ${Var^^}
ABCDEF
```

If in the above example only a single `,` or `^` is used, then only the first character is printed in lower or upper case, respectively.

It is also possible with the curly-brace syntax to select substring from variable content, with the following generic syntax:

```
${Var:offset:length}
```

The above construct returns substring, starting at 'offset', and continuing up to 'length' characters. By convention, the first character in the content of variable 'Var' is at the offset 0. If 'length' is omitted, it goes all the way until the end of 'Var'. If 'offset' is less than 0, then it counts from the end of 'Var'. All this is illustrated with the following examples:

```
$ Var=abcdefghij
$ echo ${Var:0:4}
abcd
$ echo ${Var:5:2}
fg
$ echo ${Var:5}
fghij
$ echo ${Var:(-2)}
ij
$ echo ${Var:(-3):2}
hi
```

We remark that in the expression `${Var:offset:length}` both 'offset' and 'length' are evaluated automatically in mathematical context. Therefore, we can write directly code snippets like this

```
$ Var=abcdefgh
$ Start=2
$ Lentgh=5
$ echo ${Var:Start+1:Lentgh-2}
def
```

instead of lengthier version, where mathematical context is explicitly requested via `$(( ... ))`:

```
$ echo ${Var:$((Start+1)):$((Lentgh-2))}
def
```

Finally, it is mandatory to embed negative offset within round braces `( ... )` in the above examples, since otherwise **Bash** interprets negative integers after colon `:` in this context in a very special way — this is clarified next.

By using string operators one can set the default value of a variable. Most frequently, one encounters the following two use cases:

1. `${Var:-defaultValue}` — if 'Var' exists and it is not null, return its current value. Otherwise, return the hardwired 'defaultValue'. This is basically protection that variable always has some content. For instance:

   ```
    Var=44
    echo ${Var:-100} # prints 44
   ```

   However:

   ```
   unset Var
   echo ${Var:-100} # prints 100
   ```

   This syntax has a very important use case when a script or a function expects the user to supply an argument. Even if the user forgot to do it, we can nevertheless execute the code for some default and meaningful value of that argument. For instance:

   ```
   Var=${1:-defaultValue}
   ```

This literally means that 'Var' is set to the first argument the user has supplied to a script or a function, but even if the user forgot to do it, the code can still execute by setting 'Var' to 'defaultValue'.

2. `${Var:?someMessage}` — if 'Var' exists and it is not null, return its current value. Otherwise, prints 'Var', followed by hardwired text 'someMessage', and abort the current execution of a function (in case this syntax is used in a script, it only prints the error message). For instance, in the body of a function you can add protection via:

   ```
   function myFunction
   {
    local Var=${1:?first argument is missing}
    ... some code ...
   }
   ```

   In case a user has forgotten to provide the first argument, your function will terminate automatically with the error message:

   ```
   myFunction
   bash: 1: first argument is missing
   ```

   If the message is not specified, the default message will be produced. For instance:

   ```
   unset someVariable
   Var=${someVariable:?}
   ```

   will produce the following default error message:

   ```
   bash: someVariable: parameter null or not set
   ```

In both of these examples we have used colon `:` within the curly braces, but this is optional. However, if we omit the colon `:` and use instead the syntax `${Var-defaultValue}` and `${Var?someMessage}`, the meaning is slightly different: the previous phrase 'exists and it is not null' translates now only into 'exists'. This difference concerns cases like this:

```
Var= # Var exists but it is NULL
echo ${Var:-44} # prints 44
echo ${Var-44} # prints nothing
```

When replacing old patterns with the new ones, **Bash** can handle a few wildcard characters. The most important wildcards are:

1. `*` : zero or more characters
2. `?` : any single character
3. `[ ... ]` : character sets and ranges

Their usage is best illustrated with a few concrete examples:

```
Var=1234a5678
echo ${Var/a*/TEST} # prints 1234TEST
```

Here the pattern with the wildcard 'a*' matches any string starting with 'a' and followed by 0 or more other characters.

```
Var=a1234a5678
echo ${Var//a?/TEST} # prints TEST234TEST678
```

The pattern with the wildcard 'a?' matches a string starting with the character 'a' and followed by exactly one other character (in the above example, it matched both 'a1' and 'a5', which were both replaced, due to `//` specification within curly braces, into a new pattern 'TEST').

```
Var=abcde12345
echo ${Var//[b24]/TEST} # prints aTESTcde1TEST3TEST5
```

The pattern '[b24]' matches any single character specified within `[ ... ]` (in the above example, 'b', '2' and '4' were all replaced with 'TEST').

```
Var=abcde12345
echo ${Var//[b-e]/TEST} # prints aTESTTESTTESTTEST12345
```

The pattern '[b-e]' matches all single characters in the specified range within `[ ... ]` (in the above example, 'b', 'c', 'd' and 'e', i.e. all characters in the range 'b-e' were all replaced with the new pattern 'TEST').

The real power of wildcards is manifested when they are combined:

```
Var=a1b2c3d4e5
echo ${Var//[b-d]?/TEST} # prints a1TESTTESTTESTe5
```

The pattern '[b-d]?' matches all single characters in the specified range 'b-d' followed up by exactly one other character (in the above example, 'b2', 'c3' and 'd4' were all replaced with 'TEST').

```
Var=acebfd11g
echo ${Var^^[c-f]} # prints aCEbFD11g
```

The pattern '^^[c-f]' will capitalize all single characters, but only in the specified range 'c-f', therefore only 'c', 'd', 'e' and 'f' in the above example get capitalized.

2. Arrays

Bash also supports arrays, i.e. variables containing multiple values. Since all variables in **Bash** by default are strings, you can store in the very same array integers, text, etc. Array index in **Bash** starts with zero, and there is no limit to the size of an array. There are a few ways in which an array can be initialized with its elements — the quickest one is to use the round braces `( ... )`. This syntax is illustrated with the following code snippet:

```
SomeArray=( 5 a ccc 44 )
```

Array elements are separated with one or more empty characters. To obtain the content of a particular array element, we use again the curly-brace notation `${ArrayName[index]}`. For instance, for the above example we have:

```
echo ${SomeArray[0]} # prints 5
echo ${SomeArray[2]} # prints ccc
```

To get programmatically all array entries, we can use `${ArrayName[*]}` or `${ArrayName[@]}` syntax, for instance:

```
echo ${SomeArray[*]} # prints 5 a ccc 44
```

The difference between `${ArrayName[*]}` or `${ArrayName[@]}` syntax matters only when used within double quotes, and the explanation is the same as for a difference between `"$*"` and `"$@"` when referring to the list of positional parameters (see Lecture #2).

This means that we can very conveniently loop over all array entries with:

```
for Entry in ${SomeArray[*]}; do
 echo $Entry
done
```

The printout is:

```
5
a
ccc
44
```

If an array element itself has an empty character, it will be correctly obtained in the above code snippet only if `${SomeArray[*]}` is replaced with `"${SomeArray[@]}"`, but such cases rarely occur in practice.

The total number of elements in an array is given by the syntax `${#ArrayName[*]}`:

```
echo ${#SomeArray[*]} # prints 4
```

We can set the value of a particular array element directly:

```
SomeArray[2]=ddd
```

Now if we print all elements, the initial 3rd element 'ccc' was replaced with the new value 'ddd', and we get:

```
echo ${SomeArray[*]} # prints 5 a ddd 44
```

In order to remove a particular element of an array, we need to explicitly use the keyword **unset**. This way, the length of an array and all indices are automatically recalculated:

```
unset SomeArray[2]
echo ${SomeArray[*]} # prints 5 a 44
echo ${#SomeArray[*]} # prints 3, the array was resized
```

On the other hand, unsetting the array element with:

```
SomeArray[2]= # WRONG!!
```

is wrong, since the total size of an array was not reset, i.e. this particular element is still counted as a part of an array, only it has now NULL content.

The whole array can be reset either with

```
unset SomeArray
```

or

```
SomeArray=()
```

To check whether an array has any set elements, we can use in the test construct one of the following two possibilities:

```
[[ -v SomeArray[@] ]] # evaluates to true for non-empty array
[[ ${#SomeArray[@]} > 0 ]] # evaluates to true for non-empty array
```

The array index works also backward. The last array element is:

```
echo ${SomeArray[-1]}
```

the penultimate array entry is:

```
echo ${SomeArray[-2]}
```

and so on. To append directly to the already existing array a new element, we can use programmatically the following code snippet:

```
SomeArray[${#SomeArray[*]}]=SomeValue
```

The above syntax works, because array indexing starts from 0 and ends with N-1, where N is the total number of array elements. Since `${#SomeArray[*]}` gives the total number of array elements N, the above syntax just appends the new N-th element.

Quite frequently, we need to prepend or append the same string to all array elements. This can be achieved elegantly with the following syntax:

```
SomeArray=( ${SomeArray[*]/#/SomePattern} ) # prepend
SomeArray=( ${SomeArray[*]/%/SomePattern} ) # append
```

Example 1: We have the following starting array which just contains some file names:

```
Files=( file_0 file_1 file_2 )
```

How to append to all file names the same file extension '.dat'? How to prepend to all file names the same string 'some_'?

The solution to the first question is:

```
Files=( ${Files[*]/%/.dat} )
```

In the above code snippet, we have first appended (by specifying `%`) to all array elements the same extension '.dat', and immediately redefined the array to the new content. The array elements are now:

```
$ echo ${Files[*]}
file_0.dat file_1.dat file_2.dat
```

The solution to the second question is:

```
Files=( ${Files[*]/#/some_} )
```

In the above code snippet, we have first prepended (by specifying `#`) to all array elements the same string 'some_' , and we have then redefined the array to the new content, so the array elements are now:

```
$ echo ${Files[*]}
some_file_0.dat some_file_1.dat some_file_2.dat some_file_3.dat
```

The power and flexibility of arrays come from the fact that at array declaration within `( ... )` a lot of other **Bash** functionalities are supported, for instance, the command substitution operator `$( ... )` and brace expansion `{ ... }`. That in particular means that we can effortlessly store the entire output of a command into an array, and then do some manipulation element-by-element.

Example 2: Count the number of words in an external file using arrays.

The solution is very simple and elegant:

```
FileContent=$(< SomeFile)
SomeArray=( ${FileContent} )
echo "Number of words: ${#SomeArray[*]}"
```

In the first line we have stored the content of an external file `SomeFile` into variable **FileContent**, and then just defined the array elements by obtaining its content. The empty characters which separate the words in the file, now separate the array elements in the definition.

At the expense of becoming a bit cryptic, the above solution can be condensed even further:

```
SomeArray=( $(< SomeFile) )
echo "Number of words: ${#SomeArray[*]}"
```

Example 3: How to merge entries of two arrays into one array, without using loops?

The solution is:

```
Array_1=( 1 2 3 )
Array_2=( a b c d )
NewArray=( ${Array_1[*]} ${Array_2[*]} )
echo ${NewArray[*]} # prints 1 2 3 a b c d
```

Example 4: Usage of brace expansion at array declaration.

```
SomeArray=( file_{0..3}.{pdf,eps} )
echo ${SomeArray[*]}
```

The printout is:

```
file_0.pdf file_0.eps file_1.pdf file_1.eps file_2.pdf file_2.eps file_3.pdf
file_3.eps
```

Example 5: How to store the output of some command in array?

```
SomeArray=( $(date) )
```

We can now extract from the output of **date** only a particular entry:

```
$ echo ${SomeArray[*]}
Fri May 29 16:24:25 CEST 2020
$ echo "Current month: ${SomeArray[1]}"
Current month: May
$ echo "Current time: ${SomeArray[3]}"
Current time: 16:24:25
```

Example 6: How can we catch the user's input directly into an array?

We have already seen that by using **read** command we can catch the user's input, but if we want to store the input in a few different variables, that quickly becomes inconvenient. And quite frequently, we cannot foresee the length of the user's input. For instance, how to handle the user's reply to the question: "Which countries have you visited ?" That can be solved elegantly with arrays:

```
read -p "Which countries have you visited? " -a Countries
```

By using the flag **-a** for command **read**, we have indicated that whatever user types, it will be split according to the empty character (i.e. the default input field separator) into words, and then each word is stored as a separate element in an array (in the above example, that array is named 'Countries').

We can then immediately write for instance:

```
echo "Number of countries is: ${#Countries[*]}"
echo "The first country is: ${Countries[0]}"
echo "The last country is: ${Countries[-1]}"
```

But what if the user visited New Zealand or Northern Ireland? Since these two countries contain an empty character in their names, the code above clearly cannot correctly handle these cases. In general, the problems of this type are solved by temporarily changing the default input field separator. The default input field separator is stored in the environment variable **IFS**, and a lot of **Linux** commands rely on its content. We can proceed in the following schematic way:

```
DefaultIFS="$IFS" # save default setting
IFS=somethingNew
... some code with new IFS ...
IFS="$DefaultIFS" # revert back to default setting
```

Since this is the frequently encountered case in practice, when a certain variable needs to be set only during the command execution, as we already saw before, there exists a specialized syntax applicable to cover such uses cases:

```
SomeVariable=someValue SomeCommand
```

Remember that there is no semicolon `;` between variable definition and command execution, this way the new definition of variable **SomeVariable** is visible only during the execution of **SomeCommand**. As soon as command terminates, **SomeVariable** gets automatically reset to its default value (if any).

The final solution for our example is therefore:

```
IFS=',' read -p "List (comma separated) countries you have visited: " -a
Countries
```

This way, the input field separator will be comma `,` but only during the execution of **read**.

Now if a user replied 'New Zealand,Northern Ireland' we have that:

```
echo ${Countries[0]}
# prints: New Zealand
echo ${Countries[1]}
# prints: Northern Ireland
```

As the final remark on arrays, we indicate that multidimensional (associative) arrays are rarely used in **Bash**, but nevertheless, they are supported. They need to be declared explicitly with **Bash** built-in command **declare** and flag **-A**:

```
declare -A SomeArray
```

After such declaration, **Bash** understands how to cope with the following syntax:

```
SomeArray[1,2,3]=a
SomeArray[2,3,1]=bb
```

To reference the content of elements in multidimensional arrays, we use:

```
echo ${SomeArray[1,2,3]} # prints a
echo ${SomeArray[2,3,1]} # prints bb
```

The indices do not have to be hardwired — index of **Bash** arrays can be any expression that evaluates to 0 or a positive integer.

3. Piping: `|`

We have already seen that commands can take their input directly from the user or from files. But in general, one command can take directly the output of another command as its input. This mechanism is called *piping* and it is a very generic concept in **Linux**.

In order to use the output of one command as the input to another, we use operator `|` ('pipe'), schematically as:

```
firstCommand | secondCommand
```

It is possible to chain with the pipe operator `|` multiple commands:

```
firstCommand | secondCommand | thirdCommand | ...
```

In the above example, the successful output, i.e. the *stdout* stream, of `firstCommand` has become the input, i.e. the *stdin*, to `secondCommand`. That command now processes that input, and produces its own output, which is then becoming the input to the `thirdCommand`, and so on.

It is possible to redirect simultaneously both *stdout* and *stderr* stream of one command into *stdin* of another, with the slightly modified pipe operator `|&`, schematically:

```
firstCommand |& secondCommand
```

In the above example, both the successful output stream and the error message of `firstCommand` are simultaneously redirected as an input to `secondCommand`.

Usage of pipe `|` eliminates the need for making temporary files to redirect and store the output of one command, and then supply that temporary file as an input to another command. The data flow among all commands chained with `|` in the pipeline is automated without any restriction on the size.

We now provide a few frequently use cases of piping. We have already seen that **Bash** supports directly only integer arithmetic with the construct `(( ... ))`. The floating-point arithmetic in **Bash** can be done by piping the desired expression into the external **Linux** program called **bc** ('basic calculator').

Example 1: How would you divide 10/7 at the precision of 30 significant digits?

The solution is given by the following:

```
$ echo "scale=30; 10/7" | bc
1.428571428571428571428571428571
```

The internal keyword **scale** sets the precision in **bc** program. Instead of using **bc** interactively and providing via keyboard *stdin* for its execution, we have just piped the *stdout* of **echo** as an input to **bc**.

For more sophisticated use cases, for instance when you want to use special mathematical functions, etc., use **bc -l**. The flag '-l' (ell) loads additionally in the memory the heavy mathematical libraries, which are otherwise not needed for simple calculations. If the scale is not specified, it is defaulted to 1 when only **bc** is executed, and to 20 when **bc -l** is executed.

The math library of **bc** defines the following example functions:

```
s(x) : The sine of x, x is in radians.
c(x) : The cosine of x, x is in radians.
a(x) : The arctangent of x, arctangent returns radians.
l(x) : The natural logarithm of x.
e(x) : The exponential function of raising e to the value x.
j(n,x) : The bessel function of integer order n of x.
```

Example 2: How would you calculate `e^2` to the precision of 20 significant digits?

```
$ echo "e(2)" | bc -l
7.38905609893065022723
```

Another typical use case of the pipe operator `|` is in a combination with **tee** command. Quite frequently, when a certain command is executing, we would like to see its output on the screen, but also simultaneously redirected to some file, so that at any time later we can carefully inspect the whole command output by reading through the content of that file.

This can be achieved with the **tee** command, schematically:

```
someCommand | tee someFile.log
```

For instance, the code snippet:

```
date | tee date.log
```

will print the current time on the screen, but it will also simultaneously dump it in the file named `date.log` (check its content with **cat date.log**). In the very same spirit, it is possible to keep the full execution log of any script, function, code block `{ ... }`, loops, etc.

The command **tee** writes simultaneously its input to *stdout* (screen) and redirects it to the files. By default, **tee** overwrites the content of a file — if we want instead to append to the already existing non-empty file, the following version can be used:

```
someCommand | tee -a someFile.log
```

Flag '-a' in this particular case stands for 'append'.

As the final remark on the pipelines, we consider the following important question: If the pipeline, composed of multiple commands, has failed during execution, how to figure out programmatically which particular command in the pipeline has failed? In order to answer this question, we need to inspect the status of the built-in variable **PIPESTATUS**. This variable is an array holding the exit status of each command in the last executed pipeline:

```
$ echo "scale=5000; e(2)" | bc -l | more
$ echo ${PIPESTATUS[*]}
0 0 0 # exit status of last command ('echo', 'bc' and 'more') in the pipe above
```

In the above example, we want to determine the result to 5000 significant digits, and then inspect through it screen-by-screen with the **more** command. All three commands in the pipeline, **echo**, **bc** and **more**, executed successfully, therefore the array **PIPESTATUS** holds three zeros. When only the single command has been executed, that is a trivial pipeline, and **PIPESTATUS** array has only one entry, the very same information which is stored in the special **$?** variable. The thing to remember is that **PIPESTATUS** gets updated each time we execute command, even the trivial ones like **echo**.

The power of piping is best illustrated in the combination with the three powerful commands **sed**, **awk** and **grep**, the three core **Linux** utilities for text parsing and manipulation, which we cover in the next section.

4. sed, awk and grep

Frequently a text needs to be parsed through and inspected, or updated after the search for some patterns has been performed. In general, we want to be able to modify programmatically some text for one reason or another. The text in this context can stand for any textual stream coming out of command upon execution, or for any text saved in some physical file. Clearly, there are cases in which it is impractical or even unfeasible to make all such changes in some graphics-based editors. In this section, we cover instead how the text can be manipulated programmatically, with the three core **Linux** commands: **grep**, **awk** and **sed**. Combining functionalities of all three of them gives a lot of power when it comes to programmatic text manipulation, and typically covers all cases of practical interest. The usage of these three commands is best learned from concrete examples.

grep

The command **grep** ('Globally search a Regular Expression and Print') is used to filter out from the command output or from the physical file the lines containing a certain pattern. Typically, this command is used as follows:

```
grep SomePattern(s) SomeFile(s)
```

The above syntax will select from the specified files only the lines which conform to the specified patterns, and print them on the screen.

Another frequent use case is:

```
SomeCommand | grep SomePattern(s)
```

The above syntax will select on-the-fly from the output stream of a command only the lines which conform to the specified patterns, and will print them on the screen.

Example 1: Copy and paste in the file `grepExample.txt` the following lines:

```
TEST Test test 11test test22
test TEST Test 11test test22
TeST1 TEST1 TESt1 TEST1 TEST1
test TEST Test 11test test
TeST2 TEST2 TEsT2 TEST2 tEST2
```

By using this example file, we now demonstrate the most frequently used cases of **grep** command:

```
grep "test" grepExample.txt
```

The output is:

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

By default, the specified pattern ('test' in the above example) is case sensitive and it does not have to be an exact match, therefore here the text '11test', 'test22' and 'test' were all the matching patterns. Each line which contains one or more of matching patterns is printed by **grep** on the screen by default, but it can be also redirected to a physical file:

```
grep "test" grepExample.txt > filtered.txt
```

In the next example, we instruct **grep** to use a flag '-n', to print all lines containing the pattern 'test' alongside the numbers of those lines:

```
grep -n "test" grepExample.txt
```

The result is:

```
1:TEST Test test 11test test22
2:test TEST Test 11test test22
4:test TEST Test 11test test
```

We can easily inverse the pattern search, when we need to print all lines in a file which do not contain the pattern 'test', by using the flag '-v':

```
grep -v "test" grepExample.txt
```

Now only the lines which do not contain the pattern 'test' are printed on the screen:

```
TeST1 TEST1 TESt1 TEST1 TEST1
TeST2 TEST2 TEsT2 TEST2 tEST2
```

When we need case insensitive search, we can use the flag '-i':

```
grep -i "test" grepExample.txt
```

This prints all lines in the file which contain all case insensitive variants of pattern 'test', e.g. 'TEST', 'Test', 'tEsT, etc:

```
TEST Test test 11test test22
test TEST Test 11test test22
TeST1 TEST1 TESt1 TEST1 TEST1
test TEST Test 11test test
TeST2 TEST2 TEsT2 TEST2 tEST2
```

Since each line has at least one case insensitive variant of the specified pattern 'test', the whole file is printed in this example.

Very frequently, we need to filter out all lines in the file which contain the specified pattern only at the very beginning of the line. This is achieved by using the special character `^` (caret):

```
grep "^test" grepExample.txt
```

This results in:

```
test TEST Test 11test test22
test TEST Test 11test test
```

The special character '^' is an anchor for the beginning of a line, and a lot of other commands interpret this character in the same fashion. Opposite to it, if we need to print all lines in the file which contain the specified pattern only at the end of the line, we need to use `$`:

```
grep "t22$" grepExample.txt
```

The result is the following two lines:

```
TEST Test test 11test test22
test TEST Test 11test test22
```

In this particular context, the special character `$` is an anchor for the end of a line.

We can perform the pattern search with **grep** even more differentially. If we need to filter out all lines in the file which contain at least one word *beginning* with the specified pattern, we need to use `\<`. For instance, we can proceed in the following way:

```
grep "\<TeST" grepExample.txt
```

Now both 'TeST1' and 'TeST2' will match, since they begin with the specified pattern 'TeST', and the result is:

```
TeST1 TEST1 TESt1 TEST1 TEST1
TeST2 TEST2 TEsT2 TEST2 tEST2
```

Complementary to this option, we can filter out all lines in the file which contain at least one word *ending* with the specified pattern:

```
grep "ST\>" grepExample.txt
```

Now only 'TEST' will match, since this is the only word in the file which ends up with the specified pattern 'ST', and in the printout we get only the three lines which contain word 'TEST':

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

Whet it comes to the exact pattern match, we need to use flag '-w':

```
grep -w "Test" grepExample.txt
```

This yields to the following printout:

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

Each of these three lines has at least one exact occurrence of the specified pattern 'Test'.

It is also possible to combine patterns, with the special character `\|`:

```
grep "11test\|test22" grepExample.txt
```

This prints all lines containing either the pattern '11test' or 'test22' (this is the logical OR operation):

```
TEST Test test 11test test22
test TEST Test 11test test22
test TEST Test 11test test
```

We cannot directly use **grep** to obtain the logical AND operation in the pattern search, but this limitation can be circumvented with the usage of pipe:

```
grep "11test" grepExample.txt | grep "test22"
```

This will print all lines which contain both specified patterns:

```
TEST Test test 11test test22
test TEST Test 11test test22
```

In this example, this first **grep** in the pipeline acted on a physical file, while the second **grep** got its input from the output stream of the first **grep**. Whether the input to **grep** is coming from the physical file, or via pipe `|` from the *stdout* or *stderr* stream of some other command, its usage is completely equivalent.

For instance, you can check if the variable contains some pattern schematically with:

```
echo "$Var" | grep SomePattern(s)
```

In cases where only the check for the pattern needs to be performed with **grep**, and there is no need for the actual printout, we can use the flag '-q' (for quite), like in this example:

```
if grep -q "11test" grepExample.txt; then
 ... some code ...
elif grep -q "test22" grepExample.txt; then
 ... some other code ...
else
 ... yet other code ...
fi
```

Example 2: How to select in the current working directory only the files whose names begin with the example pattern 'ce' and end up with the pattern '.dat'? The content of the directory is:

```
array.sh   be3.dat  be8.dat  ce1.log  ce4.dat  ce6.log  ce9.dat
array.sh~  be4.dat  be9.dat  ce2.dat  ce4.log  ce7.dat  ce9.log
be0.dat    be5.dat  ce0.dat  ce2.log  ce5.dat  ce7.log  grepExample.txt
be1.dat    be6.dat  ce0.log  ce3.dat  ce5.log  ce8.dat  test.sh
be2.dat    be7.dat  ce1.dat  ce3.log  ce6.dat  ce8.log  test.sh~
```

The solution is:

```
ls | grep "^ce" | grep ".dat$"
```

The **ls** command will print the list of all files in the current directory, and pipe that list to **grep** for further filtering. Then **grep** filters out the lines in the output of **ls** which begin (the anchor `^`) with the pattern 'ce'. That results is then filtered further by chaining another pipe. In the 2nd **grep** we have used the anchor `$` since we are interested in the ending '.dat'. The final output is:

```
ce0.dat
ce1.dat
ce2.dat
ce3.dat
ce4.dat
ce5.dat
ce6.dat
ce7.dat
ce8.dat
ce9.dat
```

Finally, we mention the flag '-r', which will force **grep** to search for specified patterns recursively in all files of specified directories, their subdirectories, etc. Generic syntax is:

```
grep -r somePattern dir1 dir2 ...
```

If directories are not specified, the top-level search directory is defaulted to the current working directory, and then the search is performed in all files in all its subdirectories.

Example 3: Print all lines in all files in the documentation for this lecture which contain word "Bash".

```
$ grep -r "Bash" ~/Lectures/PH8124
/home/abilandz/Lectures/PH8124/Homeworks/Homework_1.md:# Using **Bash** aliases
as your simplest commands
/home/abilandz/Lectures/PH8124/Homeworks/Homework_1.md:**Challenge #1**: Develop
a **Bash** script named ```timeZones.sh``` which is used as
/home/abilandz/Lectures/PH8124/Homeworks/Homework_2.md:# Using external
executable as Linux/Bash command

... many, many, more lines ...
```

awk

Now we move to **awk** (named after the initials of its authors: Aho, Weinberg and Kernighan), which is a programming language by itself, designed for text processing. One can easily teach the whole semester only about **awk**, here we will cover only its most important functionalities which are not available as built-in **Bash** functionalities. The frequently heard comment about **awk** is that its syntax and usage are awkward. Nevertheless, in a lot of cases of practical interest **awk** provides the best and the most elegant solution.

After we supply some input to **awk**, it will break each line of input into fields, which by default are separated with one or more empty characters. After that, **awk** parses the input and operates on each separate field. Just like with the **grep** command, **awk** can take its input either from a physical file, or from the output stream of another command via a pipe. For instance, if a certain command has produced an output that consists of column-wise entries separated with one or more empty characters, we can get hold of each field programmatically. For instance:

```
$ date
Wed Jun  3 15:36:12 CEST 2020
$ date | awk '{print $4}'
15:36:12
```

In the 2nd command input above, we have, by using **awk**, isolated directly only the 4th field in the output of **date**. In a similar fashion:

```
$ date | awk '{print $6}'
2020
```

prints only the year, because the 6th field in the output of **date** is reserved for a year.

We can select multiple fields, and immediately on-the-fly do some additional editing:

```
$ date | awk '{print $4, "some text", $6}'
15:36:12 some text 2020
```

In the same way **awk** operates on the content of files. It is very convenient, for instance, to use **awk** to extract only the values from the specified column(s) in a file. For instance, if the content of the file `someFile.dat` is:

```
a 1
yy 10
c 44
```

we can extract the columns separately with

```
$ awk '{print $1}' someFile.dat
a
yy
c
```

and

```
$ awk '{print $2}' someFile.dat
1
10
44
```

Typically, one can store such an output in an array, and then process further programmatically all entries, with the following code snippet which combines a few different functionalities covered by now:

```
$ SomeArray=( $(awk '{print $2}' someFile.dat) )
$ echo ${SomeArray[*]}
1 10 44
```

In order to get the total number of fields, we can use **awk** built-in variable **NF**:

```
$ date
Wed Jun  3 15:36:12 CEST 2020
$ date | awk '{print NF}'
6
```

Since the output stream of **date** has 6 entries separated with the empty character, we got 6 as a total number of fields.

The entry from the last field can be achieved directly by obtaining the content of **NF** variable:

```
$ date | awk '{print $NF}'
2020
```

Similarly, the entry from the penultimate field can be obtained directly with:

```
$ date | awk '{print $(NF-1)}'
CEST
```

and so on.

But what if we want to parse the command output or the file content even more differentially? For instance, what if we want to extract programmatically from the output of **date** command only the seconds, and not the full timestamp '15:36:12' by specifying the 4th field? In order to achieve that, we need to change the field separator in **awk** to some non-default value. This is achieved by manipulating the **awk** built-in variable **FS**. To set the field separator variable **FS** to some non-default value, we use schematically the following syntax:

```
awk 'BEGIN {FS="some-new-single-character-field-separator"} ... '
```

The key word 'BEGIN' next to the code snippet enclosed in `{ ... }` ensures that that particular code snippet is executed only once, at the very beginning (analogously, there exists a key word 'END' in **awk** with the opposite meaning, i.e. that code snippet is executed only once at the very end).

For instance, if we want to use colon `:` as a field separator in **awk**, we must start with the following:

```
awk 'BEGIN {FS=":"} ... '
```

Therefore, to extract only the seconds from the output of **date** command, we can use the following code snippet:

```
$ date
Wed Jun  3 16:18:44 CEST 2020
$ date | awk '{print $4}' | awk 'BEGIN {FS=":"}{print $3}'
44
```

What happened above is literally the following:

1. the command **date** produced the output stream `Wed Jun 3 16:18:44 CEST 2020`
2. that output was piped as an input for further processing to **awk** command, which extracted the 4th field, taking into account that the default field separator is one or more empty characters. The result after this step was `16:18:44`

3. this intermediate output stream `16:18:44` was then sent via another pipe to **awk** command, which, however, in the 2nd pipe runs with non-default field separator `:` . With respect to `:` as a field separator in the stream `16:18:44`, the 3rd field is seconds, which yields as the final output `44`

As a rule of thumb, field separators in **awk** shall be always single characters — composite multi-character field separators are possible, but can lead to some inconsistent behaviour among different **awk** versions (e.g. **gawk**, **mawk**, **nawk**, etc.).

Very conveniently, with **awk** we can also calculate directly the length of the field, for instance:

```
echo "a:12345:b34d" | awk 'BEGIN {FS=":"}{print length($1)}' # prints 1
echo "a:12454:b34d" | awk 'BEGIN {FS=":"}{print length($2)}' # prints 5
echo "a:12345:b34d" | awk 'BEGIN {FS=":"}{print length($3)}' # prints 4
```

On the other hand, multiple single characters can be treated as field separators simultaneously — they just all need to be embedded within `[ ... ]`. For instance, we can treat during the same **awk** execution all three characters colon `:`, semi-colon `;` and comma `,` as equivalent field separators in the following code snippet:

```
echo "1,22;abc:44:1000;123" | awk 'BEGIN {FS="[:;,]"} {print $4}'
```

The output is

```
44
```

As a side remark: If you find it very difficult to use **awk** to extract content from the specific fields, there is also a much simpler, however also much less powerful, command **cut**. For instance:

```
$ echo A BBB CC | cut -d " " -f 3
CC
```

In the above snippet we have defined the field delimiter with flag '-d' to be empty character " " (by default, the field delimiter in **cut** command is TAB), and with the flag '-f' we have specified that we want the content of the 3rd field, which is 'CC' in the example above.

The main limitation of **awk**, when used within **Bash** scripts, is that it cannot directly process the values from the **Bash** variables. We need to initialize first with additional syntax some internal **awk** variables with the content of **Bash** variables, before we can use them during **awk** execution, which in practice can be a bit, well, awkward... This particular limitation is not present in the command **sed**, which we cover next.

sed

Finally, there is **sed** ('Stream Editor'), a non-interactive text file editor. It parses the command output or file content line-by-line, and performs specified operations on them. Typically, **sed** covers the following use cases:

1. printing selected lines from a file
2. inserting new lines in a file
3. deleting specified lines in a file
4. searching for and replacing the patterns in a file

We illustrate all four use cases with a few basic examples.

Example 1: How to print only the specified lines from the command output? As a concrete example, we consider the output of **stat** command:

```
stat test.sh
```

The output is:

```
 File: test.sh
  Size: 177             Blocks: 0          IO Block: 4096   regular file
Device: 2h/2d   Inode: 21673573207029672  Links: 1
Access: (0666/-rw-rw-rw-)  Uid: ( 1000/abilandz)   Gid: ( 1000/abilandz)
Access: 2020-05-01 12:46:20.551223700 +0200
Modify: 2020-05-29 08:32:38.081673700 +0200
Change: 2020-05-29 08:32:38.081673700 +0200
 Birth: -
```

If we are interested to print on the screen only a particular line, we need to use **sed** with the flag '-n' and the specifier 'p' ('print'). Flag '-n' is needed to suppress the default printout of the original file. To print only the 2nd line, we can use the following syntax:

```
stat test.sh | sed -n 2p
```

The output is now only the 2nd line:

```
Size: 177             Blocks: 0          IO Block: 4096   regular file
```

With the slightly modified specifier, we can indicate the line ranges. For instance, the syntax

```
stat test.sh | sed -n 2,5p
```

will print lines 2, 3, 4 and 5, and so on.

Example 2: How to insert a new 2nd line of text in the already existing file `sedTest.dat`, which has the following content:

```
line 1
line 2
line 3
line 4
```

In general, to insert a new line with **sed**, we need to use the specifier 'i'. The solution is:

```
sed "2i Some text" sedTest.dat
```

This will insert in the second line (the meaning of '2i' specifier) of the file `sedTest.dat` the new text 'Some text'. The original file is not modified, only the **sed** output stream. The **sed** output stream on the screen is:

```
line 1
Some text
line 2
line 3
line 4
```

We remark that number of empty characters between the specifier 'i' and the following text is irrelevant — the very same results as above is achieved for instance with:

```
sed "2iSome text" sedTest.dat
sed "2i    Some text" sedTest.dat
sed "2   i    Some text" sedTest.dat
```

In case we want to start a new text with literal empty character, we have to escape it:

```
$ sed "2i\ Some text" sedTest.dat
line 1
 Some text
line 2
line 3
line 4
```

The above modified output stream can be redirected to a new file with `1> someFile`, but we can also modify in-place the original file. To achieve this, we need to use the flag `-i` ('in-place edit') for **sed** :

```
sed -i "2i Some text" sedTest.dat
```

This will in the 2nd line of the file `sedTest.dat` insert the new text 'Some text' and the original file is modified, without backup. Remember in this context the different meaning of 'i':

- '-i' used as a flag instructs **sed** that we want to modify the original file in-place
- 'ni' used as an argument indicates that we want to insert something on the nth line

Clearly, it can be potentially dangerous to modify directly the original file in-place, because once the original file is overwritten, there is no way back. To prevent that, we can automatically create the backup of the original file by using the slightly modified flag '-i.backup':

```
sed -i.backup "2i Some text" sedTest.dat
```

This will in the second line of the file `sedTest.dat` insert the new text 'Some text'. The original file is modified, but now also the backup of the original file was created automatically, and is saved in new file named `sedTest.dat.backup`.

Analogously, we can insert a new line on-the-fly in the output stream of some command:

```
stat test.sh | sed "4i => File permissions, and other thingies:"
```

The output is:

```
  File: test.sh
  Size: 62              Blocks: 0          IO Block: 4096   regular file
Device: 2h/2d   Inode: 26740122787573808  Links: 1
=> File permissions, and other thingies:
Access: (0666/-rw-rw-rw-)  Uid: ( 1000/abilandz)   Gid: ( 1000/abilandz)
Access: 2020-05-11 12:38:23.820690300 +0200
Modify: 2020-05-14 13:13:46.970442600 +0200
Change: 2020-05-14 13:13:46.970442600 +0200
 Birth: -
```

Using this functionality, we can easily personalize the printout of any command.

Example 3a: How to delete the 4th line from the above file `sedTest.dat`?

To delete lines in the file's or in the command's output stream, we need to use the specifier 'd' ('delete') in **sed**. For instance, if we want to delete the 4th line, we can use the following syntax:

```
sed "4d" sedTest.dat
```

This will delete the 4th line ('4d' specifier) in the file `sedTest.dat`. We can also specify the line ranges for deletion, for instance:

```
sed "2,4d" sedTest.dat
```

This will delete the 2nd, 3rd and 4th lines in the file `sedTest.dat`. The previous comments about in-place modification and how to make a backup of the original file apply also in this context.

Example 3b: How to delete the lines holding only specific text pattern?

This is another frequently used case of **sed** command, and the generic solution is:

```
sed "/somePattern/d" someFile
```

or equivalently:

```
someCommand | sed "/somePattern/d"
```

For instance:

```
$ stat test.sh | sed "/Access/d"
 File: test.sh
  Size: 177             Blocks: 0          IO Block: 4096   regular file
Device: 2h/2d   Inode: 21673573207029672  Links: 1
Modify: 2020-05-29 08:32:38.081673700 +0200
Change: 2020-05-29 08:32:38.081673700 +0200
 Birth: -
```

In the above output, the lines holding the string "Access", namely:

```
Access: (0666/-rw-rw-rw-)  Uid: ( 1000/abilandz)   Gid: ( 1000/abilandz)
Access: 2020-05-01 12:46:20.551223700 +0200
```

have been deleted.

Example 4: Finally, we also illustrate how to replace one pattern in the file with another. This is achieved with the following generic syntax:

```
sed "s/firstPattern/secondPattern/" someFile
```

This will substitute (the 's' specifier) in each line of file `someFile` only the first occurrence of `firstPattern` with `secondPattern`. On the other hand, if we want to replace all occurrences, we need to use the following, slightly modified syntax:

```
sed "s/firstPattern/secondPattern/g" someFile
```

Note the additional specifier 'g' (for 'global') at the end of an expression. For instance, if we consider the file `example.log` with the following content:

```
momentum energy
energy momentum momentum
momentum energy momentum
```

We can replace only the first occurrence of 'momentum' with 'p' on each line with the following syntax:

```
sed "s/momentum/p/" example.log
```

The result is:

```
p energy
energy p momentum
p energy momentum
```

On the other hand, we can replace all occurrences of 'momentum' with 'p' on each line with the slightly modified syntax:

```
sed "s/momentum/p/g" example.log
```

Now the result is:

```
p energy
energy p p
p energy p
```

The very convenient thing about **sed** is that it can interpret **Bash** variables directly. It is perfectly feasible in to have in a script something like:

```
Before=OldPatern
After=NewPatern
sed "s/${Before}/${After}/" someFile
```

This gives a lot of flexibility, because old and new patterns can be supplied via arguments to scripts or functions, etc. In the same spirit, we can use **sed** to modify on-the-fly the output stream of any command:

```
$ date
Wed Jun  3 21:08:49 CEST 2020
$ date | sed "s/Wed/Wednesday/"
Wednesday Jun  3 21:08:49 CEST 2020
```

As a concluding remarks about **sed**, we indicate that multiple commands can be specified and executed in one go by using option '-e' and by separating multiple commands with ';' — for instance:

```
$ echo "some text" | sed -e "s/text/TEXT/; s/some/SOME/"
SOME TEXT
```

Finally, **sed** provides full support for pattern matching via regular expressions, which increases its power and applicability tremendously.