Introduction to Bash scripting language

Day 3

Alessandro Sciarra

Z02 – Software Development Center

09.10.2019

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

CRC-TR 211
Strong-interaction matter under extreme conditions

Topics of the day

1. Compound commands
2. Functions
3. Built in VS external programs
4. Asynchronous commands
5. Traps
6. Awk and Sed
7. Good practices
8. What to do, now?

You already know almost everything

Today we will finalise your knowledge adding a few powerful bricks that will provide you with lots of versatility and (almost) infinite power!

Compound commands

The Skogafoss cascade

Compound commands

- `if` statements
- `for` loops
- `while`, `until` loops
- `[[` keyword
- `case`, `select` constructs

- Subshells
- Command grouping
- Arithmetic evaluation

  { Slightly different from the Arithmetic expansion we already discussed }

- [Functions] { Discussed in detail in a separate section }

Compound commands

- `if` statements
- `for` loops
- `while`, `until` loops
- `[[` keyword
- `case`, `select` constructs

} Already discussed in detail

- Subshells
- Command grouping
- Arithmetic evaluation

} What we are going to discuss

{ Slightly different from the Arithmetic expansion we already discussed }

- [Functions] { Discussed in detail in a separate section }

Strictly speaking not a compound command, but they work in a similar way

Subshells

Definition

A subshell is **a child process** but it is one **that inherits more than a normal external command** does. It can see all the variables of your script, not just the ones that have been exported to the environment.

Unix process

Every process on a Unix system has its own parcel of memory, for holding its variables, its file descriptors, its copy of the Environment inherited from its parent process, and so on. The changes to the variables (and other private information) in one process do not affect any other processes currently running on the system.

Use them consciously

Forking a subshell leads to a speed penalty which often is irrelevant but which you should keep in mind!

Subshells

Definition

A subshell is **a child process** but it is one **that inherits more than a normal external command** does. It can see all the variables of your script, not just the ones that have been exported to the environment.

- It is explicitly forced using the parenthesis `(...)`
- Changes e.g. to variables done in a subshell are not remembered when exiting the subshell: A subshell can be thought as a temporary shell!
- There are many instances in which a shell creates a subshell without parentheses being placed by the programmer
  - In pipelines ⟵ **Every command in a pipeline is run in its own subshell!**
  - In command substitution
  - Executing other programs or scripts
  - In any external command that executes new shells (e.g. `awk`, `sed`, `perl`)
  - In process substitution
  - In backgrounded commands and coprocs

From Bash v4.2 the shell option `lastpipe` can be enabled so that the last command in a pipeline is not run in a subshell.

Subshells: Examples

```
# Changes in a subshell do not propagate back
$ aVar="Hello"; pwd
/home/sciarra/Documents
$ ( aVar="Goodbye"; echo "${aVar}" ); echo "${aVar}"
Goodbye
Hello
$ ( cd /tmp; pwd ); pwd
/tmp
/home/sciarra/Documents
# It is often a feature to take advantage of!
$ (cd /tmp || exit 1; tar ...)
$ (source ~/AuxiliaryBashTools.bash; ...)
# In a subshell the script variable are accessible
$ ( echo "${aVar} from the subshell" )
Hello from the subshell
# Implicit subshells: be aware of them!
$ echo "Goodbye" | read aVar; echo "${aVar}"
Hello
$ read aVar <<< "Goodbye"; echo "${aVar}"; unset aVar
Goodbye
```


Advanced reading about pipelines and subshells: Why can't I pipe data to read?

Command grouping

We already said something about it, let us go through once again

- Commands may be grouped together using curly braces `{...}`
- Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow
- All compound commands such as `if` statements and `while` loops do this as well, but command groups do **only** this
- Command groups are executed in the same shell as everything else, NOT in a new one!

```
$ { echo "$(date)"
>   rsync -av . /backup; echo "$(date)"; } >backup.log 2>&1
```

The above example truncates and opens the file backup.log on stdout, then points stderr at where stdout is currently pointing (backup.log), then runs each command with those redirections applied. The file descriptors remain open until all commands within the command group complete before they are automatically closed. This means backup.log is only opened a single time, not opened and closed for each command.

Command grouping

We already said something about it, let us go through once again

- Commands may be grouped together using curly braces `{...}`
- Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow
- All compound commands such as `if` statements and `while` loops do this as well, but command groups do **only** this
- Command groups are executed in the same shell as everything else, NOT in a new one!

```
$ { echo "$(date)"
>   rsync -av . /backup; echo "$(date)"; } >backup.log 2>&1
```


Why is this semicolon absolutely **mandatory**?

Command grouping

We already said something about it, let us go through once again

- Commands may be grouped together using curly braces `{...}`
- Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow
- All compound commands such as `if` statements and `while` loops do this as well, but command groups do **only** this
- Command groups are executed in the same shell as everything else, NOT in a new one!

```
$ { echo "$(date)"
>   rsync -av . /backup; echo "$(date)"; } >backup.log 2>&1
```

Why is this semicolon absolutely **mandatory**?

Because otherwise the closing curly bracket would be a argument of the final command in the group, `echo` in this case

Arithmetic evaluation: The `let` builtin

- Sometimes we want to do arithmetic instead of string operations
- One way to do so is to use the `let` builtin

```
$ aVar=4+5; echo "${aVar}"
4+5
$ let aVar=4+5; echo "${aVar}"; unset aVar
9
```

- However, it requires quotes to use arithmetic operators {help let}

```
$ let aVar=2<3
bash: 3: No such file or directory
$ let aVar='2<3'; echo "${aVar}"; unset aVar
1
```

- `let` accepts more than one expression

```
$ let aVar='2<3' bVar=3*7; echo "${aVar} ${bVar}"
1 21
$ unset aVar bVar
```

- If the last expression evaluates to 0, `let` returns 1; `let` returns 0 otherwise.

Arithmetic evaluation: The command grouping `( (...) )`

- `(( expression ))` is equivalent to `let "expression"`
- No quote is needed in it, since only arithmetic operations are there performed
- However, only one expression can be evaluated (not bad for the exit code)

```
$ (( aVar=7*3**2 )); echo "${aVar}"
63
$ (( aVar=1+${aVar}/20 )); echo "${aVar}"; unset aVar
4          # '${}' not really needed*
```

- Although not a compound command, the arithmetic substitution uses the same syntax

```
$ (( aVar=7*3**2 )); echo "${aVar}"
63
$ echo "aVar=$(( 7*3**2 ))"; unset aVar
63
```

- Assignments in arithmetic substitution work but are confusing and should be avoided!

```
$ echo "_${bVar}_ $(( bVar=7*3**2 )) _${bVar}_"
__ 63 _63_
```


* Using the `${}` makes Bash use `''` for uninitialised variables and might trigger errors (0 is used for uninitialised variables referenced without `${}` syntax).

Arithmetic evaluation: The command grouping ((...))

- Arithmetic evaluation is very helpful in combination with conditionals

```
$ ((aVar=(5+2)*3))
$ if ((aVar == 21)); then
>   echo 'Blackjack!'
> fi
Blackjack!
```

- Because the inside of ((...)) is C-like, a variable (or expression) that evaluates to zero will be considered false for the purposes of the arithmetic evaluation. Then, because the evaluation is false, it will exit with a status of 1. Likewise, if the expression inside ((...)) is non-zero, it will be considered true; and since the evaluation is true, it will exit with status 0.

```
$ flag=0  # no error
$ while read line; do
>   if [[ ${line} == *err* ]]; then flag=1; fi
> done < inputfile
$ if ((flag)); then echo 'Houston, we have a problem!'; fi
```

Functions

The Keifarvatn lake

Functions, finally! Overview

- Functions are the last basic Bash feature we'll learn
  - They give you an incredible opportunity to structure and increase readability of your script
  - They can be used to split large script across multiple files in an elegant way
  - Learn them well, for real!
- Functions are a tricky world!
- They have several features that we might call **issues** if compared to other languages
- Indeed, functions in Bash are not as powerful as we might expect
  - → Return value
  - → Reusability
  - → Scope
  - → I/O + ...

> Don't bite the newbie for not understanding all this. Shell functions are totally f***ed.
>
> Greg's Wiki

Do not be scared!

Learn, understand and use functions for what they are, not for what you would like them to be!

Basic syntax

```
# POSIX compliant syntax
NAME () COMPOUND-COMMAND [ REDIRECTIONS ]
# Totally equivalent syntax (in Bash), but not POSIX
function NAME [()] COMPOUND-COMMAND [ REDIRECTIONS ]
```

NAME:

A **sane** function name should be an alphanumeric string, maybe containing underscore and not starting with a number. However, insane names are accepted in Bash and, in principle, names like `:` or `[}{` are allowed. But please, avoid them! Really!! Exploring allowed names

COMPOUND-COMMAND:

The body of a function can be any compound command.

- `{ list; }` ⟵ Use this if you do not have a reason to use a different one
- `(list)` or `((expression))` or `[[ expression ]]`

REDIRECTIONS:

They take place when the function is called and they refer to the whole compound-command.

Functions: The most basic example

Just accomplish a **detached** task

Whenever a block of code can be executed as standalone, without needing either input information nor variables, it is straightforward to include it in a dedicated function

```
1  #!/bin/bash
2
3  function CreateListOfFiles()
4  {
5      printf "#%19s %20s %15s %20s %20s %6s      %s\n" \
6             "user" "group" "permissions"                \
7             "size(KB)" "permissions" "type" "path"
8      find "${PWD}" -printf "%20u %20g %15m %20k %20M %6y      %p\n"
9  }
```

Note

This will do absolutely nothing when run. This is because it has only been stored in memory, much like a variable, but it has not yet been called.

```
10 CreateListOfFiles
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.

```
1  #!/bin/bash
2
3  LevelOne() {
4      LevelTwo
5      echo "In ${FUNCNAME}, aVar = ${aVar}"
6  }
7  LevelTwo() {
8      local aVar; aVar="${FUNCNAME} local"
9      LevelThree
10 }
11 LevelThree() { echo "In ${FUNCNAME}, aVar = ${aVar}"; }
12
13 aVar='global'; LevelOne
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.

```
14 $ ./script.bash
15 In LevelThree, aVar = LevelTwo local
16 In LevelOne, aVar = global
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.
- Unless you have a reason not to do so, declare all variables in functions as `local`
- Do not assign a value to a local variable at declaration, because you might obfuscate/loose an exit code!

```
17 $ function Test { aVar="$(exit 1)"; echo $?; }; Test
   1  # Fine, but we pollute caller with 'aVar' variable
19 $ function Test { local aVar="$(exit 1)"; echo $?; }; Test
   0  # <- local's exit code!
21 $ function Test { local aVar; aVar="$(exit 1)"; echo $?; }; Test
   1  # GOOD CODE
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.
- Unless you have a reason not to do so, declare all variables in functions as `local`
- Do not assign a value to a local variable at declaration, because you might obfuscate/loose an exit code!
- If a variable at the current local scope is unset, it will remain so until it is reset in that scope or until the function returns. Once the function returns, any instance of the variable at a previous scope will become visible.
  However, if the unset acts on a variable at a previous scope, any instance of a variable with that name that had been shadowed will become visible!

Passing arguments to functions

So far so good

Delegating tasks to functions, even if the task requires variables, is quite straightforward, provided one keeps scope rules in mind

Passing arguments to functions

So far so good

Delegating tasks to functions, even if the task requires variables, is quite straightforward, provided one keeps scope rules in mind

The Pandora's box

When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter # that expands to the number of positional parameters is updated to reflect the change. Special parameter 0 is unchanged.

Bash manual

Passing arguments to functions

So far so good

Delegating tasks to functions, even if the task requires variables, is quite straightforward, provided one keeps scope rules in mind

The Pandora's box

> When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter # that expands to the number of positional parameters is updated to reflect the change. Special parameter 0 is unchanged.
>
> Bash manual

- Arguments to functions are meant to provide **input** to a function
- Bash strictly uses call-by-value semantics
- You can't pass arguments "by reference"

  { at least not until Bash 4.3 (and even there the `declare -n` mechanism has serious security flaws) }
- Passing the name of a variable to a function, which should use it, requires `eval` acrobatics and it should be as far as possible avoided!

Passing arguments to functions: Example

```
1  #!/bin/bash
2
3  function SecondsToTimeStringWithDays()
4  {
5      local inputTime days hours minutes seconds
6      inputTime=$1
7      ((    days=(inputTime)/86400 ))
8      ((   hours=(inputTime - days*86400)/3600 ))
9      (( minutes=(inputTime - days*86400 - hours*3600)/60 ))
10     (( seconds=inputTime%60 ))
11     printf "%d-%02d:%02d:%02d\n" \
12             "${days}" "${hours}" "${minutes}" "${seconds}"
13 }
14
15 for example in 31536000 86400 3600; do
16     SecondsToTimeStringWithDays ${example}
17 done
```

```
18 $ ./scriptAbove
19 365-00:00:00
20 1-00:00:00      # What is bad in the function above?
21 0-01:00:00
```

Passing arguments to functions: Example

```
22 #!/bin/bash
23
24 function SecondsToTimeStringWithDays() # Better code!
25 {
26     if [[ ! ${1} =~ ^[1-9][0-9]*$ ]]; then
27         echo "ERROR: Function ${FUNCNAME} wrongly called!"
28         return 1
29     fi
30     local inputTime days hours minutes seconds
31     inputTime=$1
32     ((    days=(inputTime)/86400 ))
33     ((   hours=(inputTime - days*86400)/3600 ))
34     (( minutes=(inputTime - days*86400 - hours*3600)/60 ))
35     (( seconds=inputTime%60 ))
36     printf "%d-%02d:%02d:%02d\n" \
37             "${days}" "${hours}" "${minutes}" "${seconds}"
38 }
39
40 for example in 31536000 86400 3600 ''; do
41     SecondsToTimeStringWithDays ${example}
42 done
```

The `return` builtin and functions' exit code

Functions' exit code

When executed, the exit status of a function is the exit status of the **last command executed** in the body.

```
$ help return
return: return [n]              # ATTENTION: 0 <= N <=255
Return from a shell function.

Causes a function or sourced script to exit with the
return value specified by N. If N is omitted, the
return status is that of the last command executed
within the function or script.

Exit Status:
Returns N, or failure if the shell is not executing
a function or script.
```

Do not use `return` to retrieve the function result!

Use `return` to early terminate a function and/or to give back an exit code!

How do I return something from a function?

How do I return something from a function?

It's simple: You don't!

How do I return something from a function?

It's simple: You don't!

1 Capturing standard output

- ✗ the function is executed in a subshell, i.e. variable assignments not seen in the caller
- ✗ **everything** printed by the function is captured (debug output!? → use stderr)
- ✓ good solution if performance is not critical

```
ExampleFunction() {
    echo "running foo()..."  >&2
    echo "this is my data"
}
aVar=$(ExampleFunction)
echo "ExampleFunction returned '${aVar}'"
```

How do I return something from a function?

It's simple: You don't!

1. Capturing standard output
2. Global variables
   - ✓ the function is **not** executed in a subshell → potentially much faster
   - ✗ if the function is executed in a subshell, the method fails!
   - ✗ the function cannot be used in a pipeline (as in any implicit subshell)

```
ExampleFunction() {
    globalVar="this is my data"
}
ExampleFunction
echo "ExampleFunction returned '${globalVar}'"
```

How do I return something from a function?

It's simple: You don't!

1. Capturing standard output
2. Global variables
3. Writing to a file
   - ✗ you need to manage a temporary file, which is always inconvenient
   - ✗ there must be a writable directory somewhere and sufficient space to hold the data therein
   - ✓ it will work regardless of whether your function is executed in a subshell

```
ExampleFunction() {
    echo "this is my data" > "$1"
}
# This is NOT solid code to handle temp files! 🔗Solid way
tmpfile=$(mktemp)   # GNU/Linux
ExampleFunction "${tmpfile}"
echo "ExampleFunction returned '$(cat < "${tmpfile}" )'"
rm "${tmpfile}"
```


If this were a real program, there would have been error checking, and a trap.

How do I return something from a function?

It's simple: You don't!

1. Capturing standard output
2. Global variables
3. Writing to a file
4. Dynamically scoped variables
   - ✗ if the function is executed in a subshell, the method fails!
   - ✗ this technique doesn't work with recursive functions
   - ✓ global variable namespace isn't polluted by the function return variable

```
ExampleFunction_implementation() {
   notGlobalVar="this is my data"
}
ExampleFunction() {
    local notGlobalVar; ExampleFunction_implementation
    # Do here something with 'notGlobalVar'
    echo "In ExampleFunction, got '${notGlobalVar}'"
}
# Here at the global scope, 'notGlobalVar' is not visible
ExampleFunction
echo "ExampleFunction returned '${notGlobalVar}'"
```

How do I return something from a function?

It's simple: You don't!

1 Capturing standard output

- ✗ the function is executed in a subshell, i.e. variable assignments not seen in the caller
- ✗ **everything** printed by the function is captured (debug output!? ⟶ use stderr)
- ✓ good solution if performance is not critical

2 Global variables

- ✓ the function is **not** executed in a subshell ⟶ potentially much faster
- ✗ if the function is executed in a subshell, the method fails!
- ✗ the function cannot be used in a pipeline (as in any implicit subshell)

3 Writing to a file

- ✗ you need to manage a temporary file, which is always inconvenient
- ✗ there must be a writable directory somewhere and sufficient space to hold the data therein
- ✓ it will work regardless of whether your function is executed in a subshell

4 Dynamically scoped variables

- ✗ if the function is executed in a subshell, the method fails!
- ✗ this technique doesn't work with recursive functions
- ✓ global variable namespace isn't polluted by the function return variable

Choose with care

Use the approach you prefer depending on the situation!

Splitting large scripts across multiple files

A Bash script should not be **too** large, though

Declaring a function does not mean to run it

You can collect functions in a separate file, which is meant to be sourced at need!

```
# Collection of tools

function ExtractColumnFromFile()
{
    # ...
}

function CalculateSizeOfFiles()
{
    # ...
}

function ReportOnLargestDirectories()
{
    # ...
}
```

Splitting large scripts across multiple files

A Bash script should not be **too** large, though

```
#!/bin/bash

source /home/sciarra/Script/UtilityFunctions.bash
source /home/sciarra/Script/UtilityFunctions_nice.bash
source /home/sciarra/Script/UtilityFunctions_cool.bash

# Call to functions (only, maybe)
```

- As long as the sourced files contain functions only,
  - the **sourcing order does not matter** and
  - functions in a file can even use functions in another file!
- The shebang can/should be included in the main file only!
- Prevent auxiliary files from being run, only sourced!

```
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
  echo "Script \"${BASH_SOURCE[0]}\" can only been sourced!" 2>&1
  exit -1
fi
```

Functions: Miscellaneous

- Functions can be marked as read-only using the `readonly` builtin
- Functions can be unset via `unset -f`
- Functions can be recursive

```
1 function CountTill5From()
2 {
3     if [[ $1 < 5 ]]; then
4         echo $1; CountTill5From $(($1 + 1))
5     fi
6 }
```

- The `FUNCNEST` variable may be used to limit the depth of the function call stack and restrict the number of function invocations { By default, no limit is placed on the number of recursive calls }

```
$ FUNCNEST=2; CountTill5From()
1
2
bash: CountTill5From: maximum function nesting level exceeded (2)
$ unset -v FUNCNEST
```

Functions: Miscellaneous

About recursion

- «To iterate is human, to recur, divine.» – **L. Peter Deutsch**
- There are cases where it is needed
- In Bash rarely, though
- The fact that by default in Bash there is no limit to the number of function invocations is something to have clear in mind!
- For example, what does the following code do? **DON'T RUN IT!**

```
:(){ :|:& };:    # Do NOT run this line!
```

Functions: Miscellaneous

About recursion

- «To iterate is human, to recur, divine.» – **L. Peter Deutsch**
- There are cases where it is needed
- In Bash rarely, though
- The fact that by default in Bash there is no limit to the number of function invocations is something to have clear in mind!
- For example, what does the following code do? **DON'T RUN IT!**

```
:(){ :|:& };:    # Do NOT run this line!
```

It is equivalent to

```
# Do NOT run this code!
function bomb() { bomb | bomb & }; bomb
```

A **fork bomb** is a denial-of-service attack wherein a process continually replicates itself to deplete available system resources, **slowing down or crashing the system** due to resource starvation.

Wikipedia

A last big warning: The `eval` builtin

```
$ help eval
eval: eval [arg ...]
Execute arguments as a shell command.

Combine ARGs into a single string, use the result as input
to the shell, and execute the resulting commands.

Exit Status:
Returns exit status of command or success if command is null.
```

- «`eval` is a common misspelling of **evil**» – Greg's Wiki
- It causes your code to be parsed twice instead of once; this means that, for example, if your code has variable references in it, the shell's parser will evaluate the contents of that variable. This can lead to unexpected results!

Advanced, but very informative article which you should spend some time at some point on.

A last big warning: The `eval` builtin

```
# This code is evil and should never be used!
function FifthElementOf() {
  local _fifth_array=$1
  eval echo "\"The fifth element is \${$_fifth_array[4]}\""
}
```

```
# Source/define the function above
$ array=(zero one two three four five)
$ FifthElementOf array
The fifth element is four
```

You might be thinking – "It looks OK, isn't it? What's wrong man?"

Advanced, but very informative article which you should spend some time at some point on.

A last big warning: The `eval` builtin

```
# This code is evil and should never be used!
function FifthElementOf() {
  local _fifth_array=$1
  eval echo "\"The fifth element is \${$_fifth_array[4]}\""
}
```

```
# Source/define the function above
$ array=(zero one two three four five)
$ FifthElementOf array
The fifth element is four
```

You might be thinking – "It looks OK, isn't it? What's wrong man?" – Consider then:

```
$ FifthElementOf 'x}"; date; #'
The fifth element is
Wed 28 Aug 14:43:59 CEST 2019  # AAAAAARRRGGGGHHH!!!!
```

Take-home lesson

An inappropriate use of `eval` can lead to arbitrary code execution!

Advanced, but very informative article which you should spend some time at some point on.

A last big warning: The `eval` builtin

```
# This code is evil and should never be used!
function FifthElementOf() {
  local _fifth_array=$1
  eval echo "\"The fifth element is \${$_fifth_array[4]}\""
}
```

```
# Source/define the function above
$ array=(zero one two three four five)
$ FifthElementOf array
The fifth element is four
```

You might be thinking – "It looks OK, isn't it? What's wrong man?" – Consider then:

```
$ FifthElementOf 'x}"; date; #'
The fifth element is
Wed 28 Aug 14:43:59 CEST 2019  # AAAAAARRRGGGGHHH!!!!
```

What about `rm -rf /` here?

Take-home lesson

An inappropriate use of `eval` can lead to arbitrary code execution!

Advanced, but very informative article which you should spend some time at some point on.

Built in VS external programs

The volcanic beach of the Jökursárlón

Do not underestimate the difference

Internal Commands:

Commands which are built into the shell. This means that the code that implements a builtin is in `/bin/bash`.

External Commands:

When an external command has to be executed, the shell looks for its path given in `PATH` variable and also a new process has to be spawned and the command gets executed.

The precedence order for command names

1. Alias
2. Function { You can shadow builtin commands creating a function with the same name }
3. Builtin { The `builtin` builtin serves to refer to a shadowed builtin → `help builtin` }
4. Keywords
5. External command in the directories listed in `${PATH}` in order.

A trivial benchmark

```
$ mkdir tmpFolder; cd tmpFolder
$ time for i in {1000..9999}; do echo > $i; done; echo

real    0m12.026s
user    0m0.434s
sys     0m0.857s

$ time for i in {1000..9999}; do /bin/echo > $i; done; echo

real    0m27.441s
user    0m9.504s
sys     0m5.034s

$ cd ..; rm -r tmpFolder
```

Do not overdo, just keep it in mind

- «Premature optimisation is the root of all evil» – **Donald Knuth**
- **Unnecessary use of external commands are welcome**

Asynchronous commands

Huge magma bombs

Processes in the terminal

🔗 Bash manual v5.0 pages 104-107

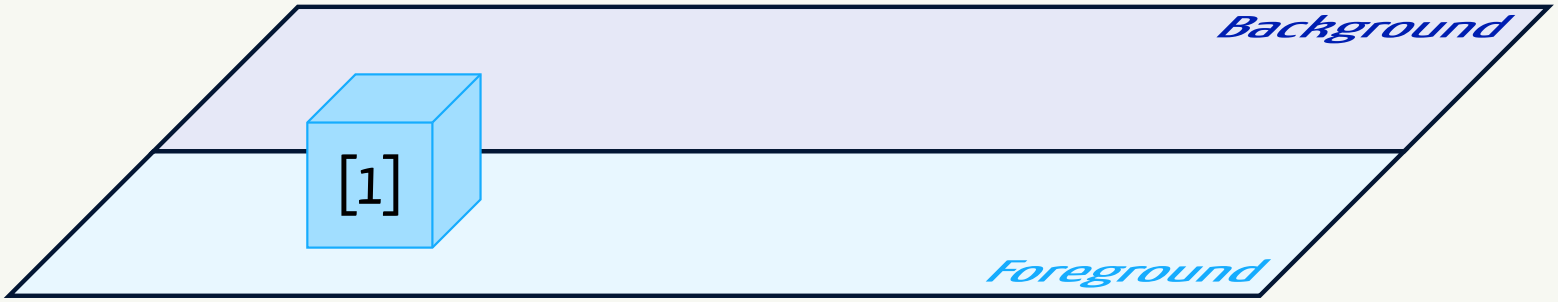
- The `ps` command displays information about (a selection of) the active processes
- A **process** is «an instance of a computer program that is being executed» – **Wikipedia**
- The shell associates a **job** with each pipeline
- It keeps a table of currently executing jobs, which may be listed with the `jobs` builtin
- When Bash (in interactive mode*) starts a job asynchronously (i.e. in background), it prints a line that looks like

```
$ sleep 30 & # <- The ampersand symbol starts
[1] 25647    #    a process in background
```

  indicating that this job is job number 1 and that the process ID of the last process in the pipeline associated with this job is 25647 { We will come back to the process ID later }
- All of the processes in a single pipeline are members of the same job
- Bash uses the job abstraction as the basis for job control

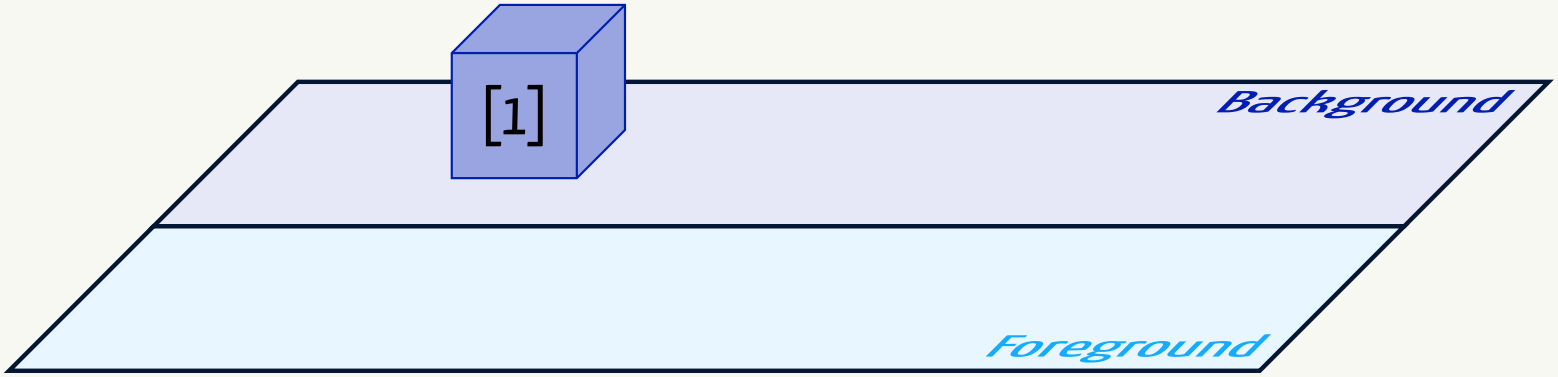
* Job control is by default turned off in non-interactive mode, but it might be activated via `set -m`.

Job control: A graphical overview



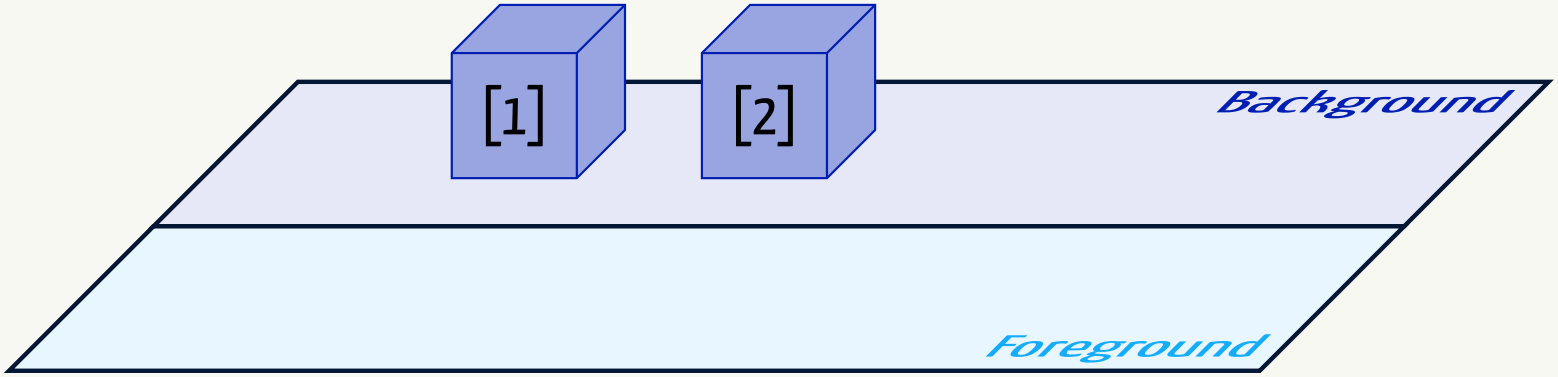

```
$ jobs
$ emacs Loops.tex   # <- each pipeline is a fg job by default
# Shell is blocked till you have the emacs process
# in foreground. You can close it to get the shell back.
```

Job control: A graphical overview



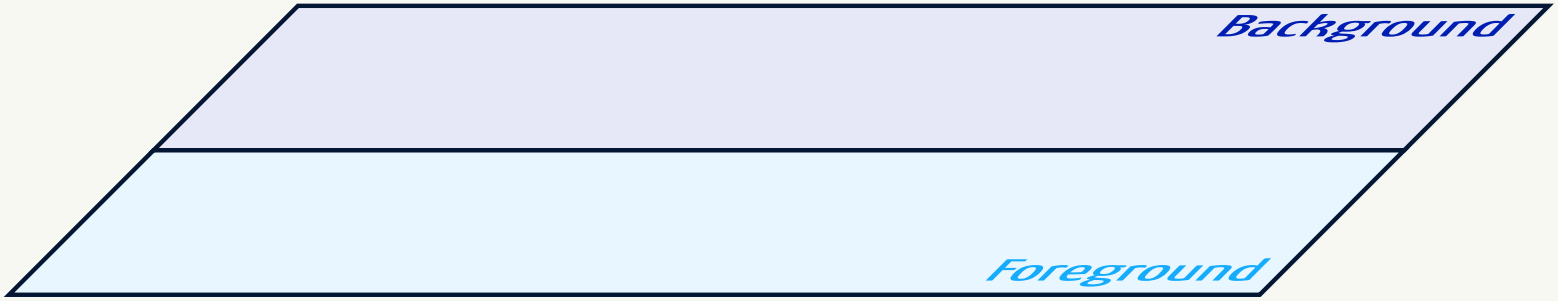

```
$ jobs
$ emacs Loops.tex   # <- each pipeline is a fg job by default
# Shell is blocked till you have the emacs process
# in foreground. You can close it to get the shell back.
$ emacs Loops.tex & # <- use the ampersand to start it in bg
[1] 5642
$ jobs
[1]+  Running                 emacs Loops.tex &
```

Job control: A graphical overview



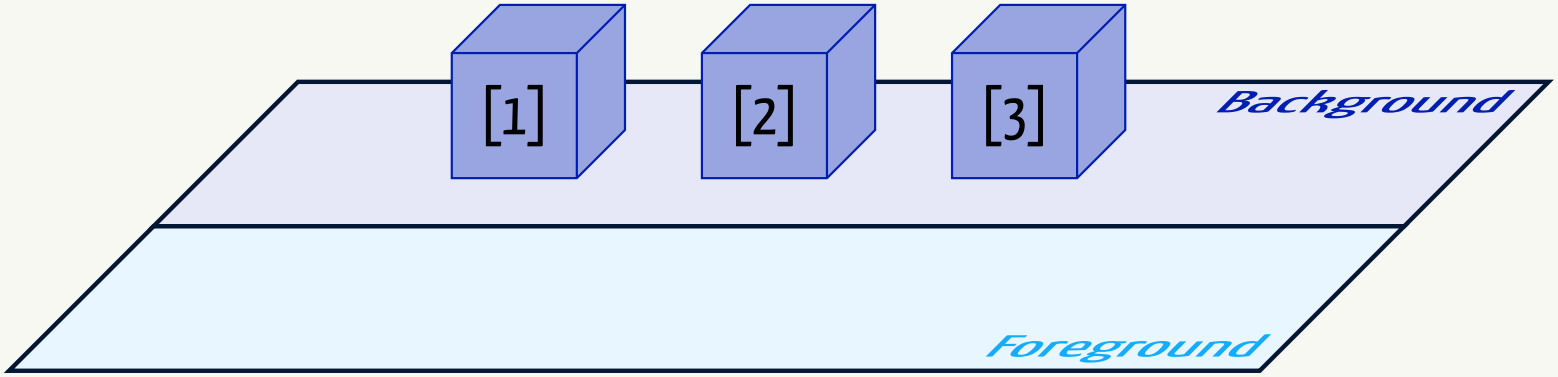

```
$ jobs
$ emacs Loops.tex   # <- each pipeline is a fg job by default
# Shell is blocked till you have the emacs process
# in foreground. You can close it to get the shell back.
$ emacs Loops.tex & # <- use the ampersand to start it in bg
[1] 5642
$ jobs
[1]+  Running                 emacs Loops.tex &
$ sleep 30 &
[2] 20943
$ jobs
[1]-  Running                 emacs Loops.tex &
[2]+  Running                 sleep 30 &
```

Job control: A graphical overview




```
14 # After half a minute and closing emacs
15 [2]+  Done                    sleep 30
16 [1]+  Done                    emacs Loops.tex
```

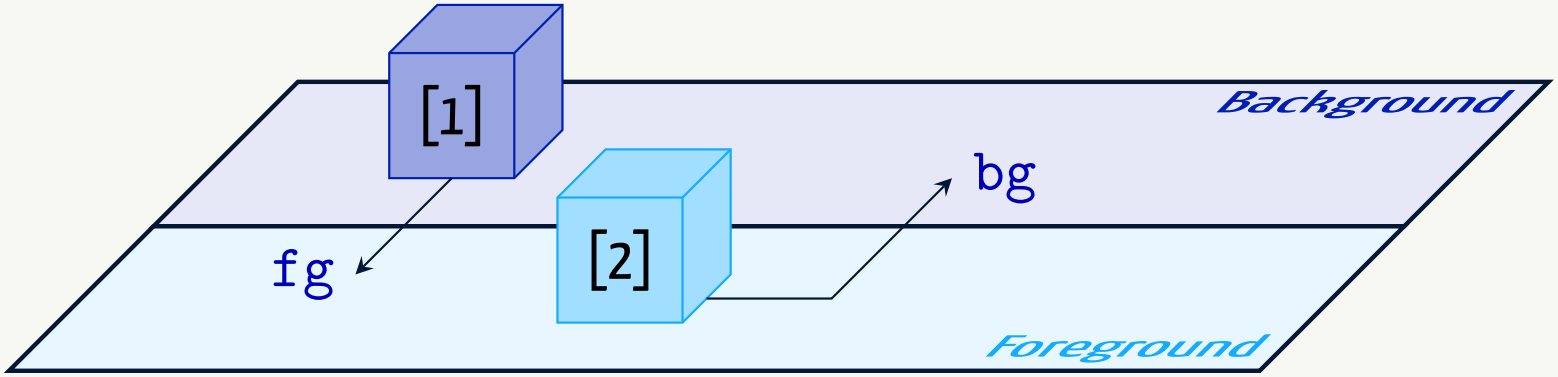
Job control: A graphical overview




```
# After half a minute and closing emacs
[2]+  Done                    sleep 30
[1]+  Done                    emacs Loops.tex
# You can start many jobs in background
$ for index in 30 60 90; do sleep ${index} & done && unset index
[1] 23559
[2] 23560
[3] 23561
$ jobs
[1]   Running                 sleep ${index} &
[2]-  Running                 sleep ${index} &
[3]+  Running                 sleep ${index} &
```

No semicolon!

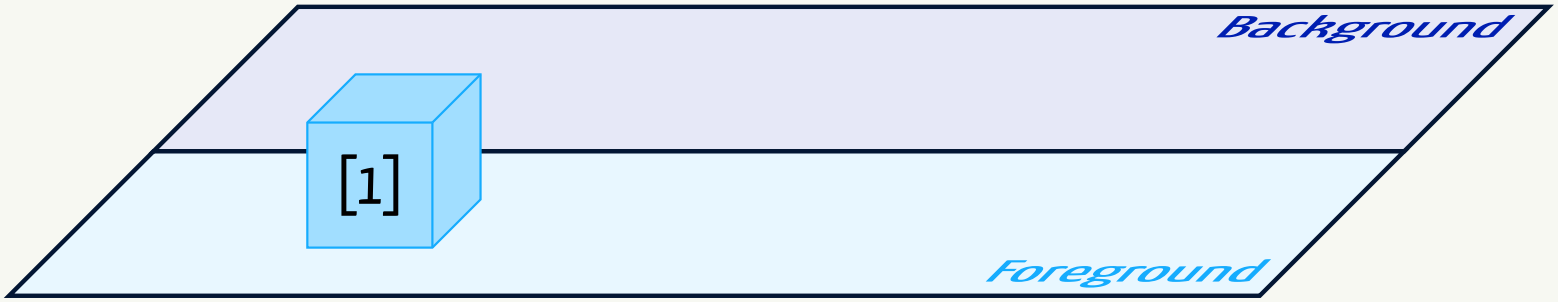
Job control: A graphical overview



- Every running process can receive signals during its execution
- Some signals are bound to common keyboard shortcuts

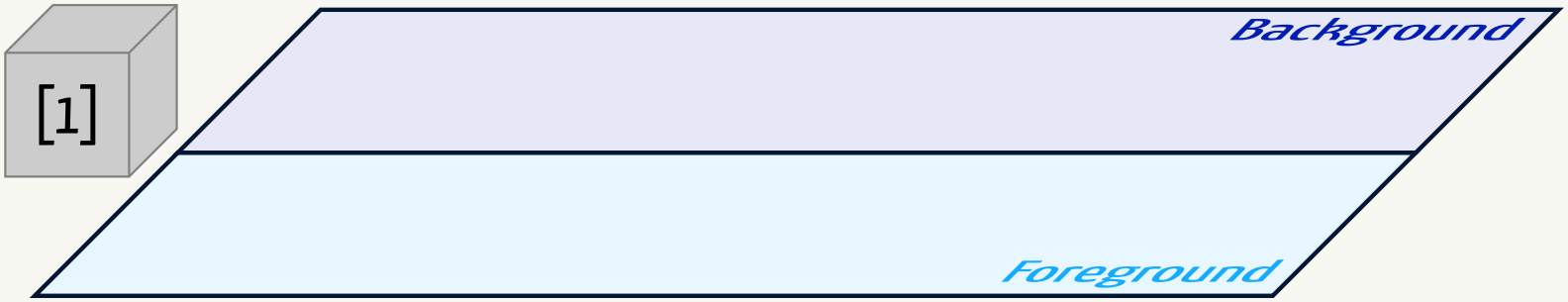
  CTRL-Z sends SIGTSTP to the foreground job { usually suspending it }
  CTRL-C sends SIGINT to the foreground job { usually terminating it }
  CTRL- sends SIGQUIT to the foreground job { usually causing it to dump core and abort }
- Jobs can be moved to background using the `bg` builtin
- Jobs can be moved to foreground using the `fg` builtin
- Any signal can be sent to a process/job using the `kill` builtin { More on it in a moment }
- The `disown` builtin make the shell "forget" a job

Job control: A graphical overview



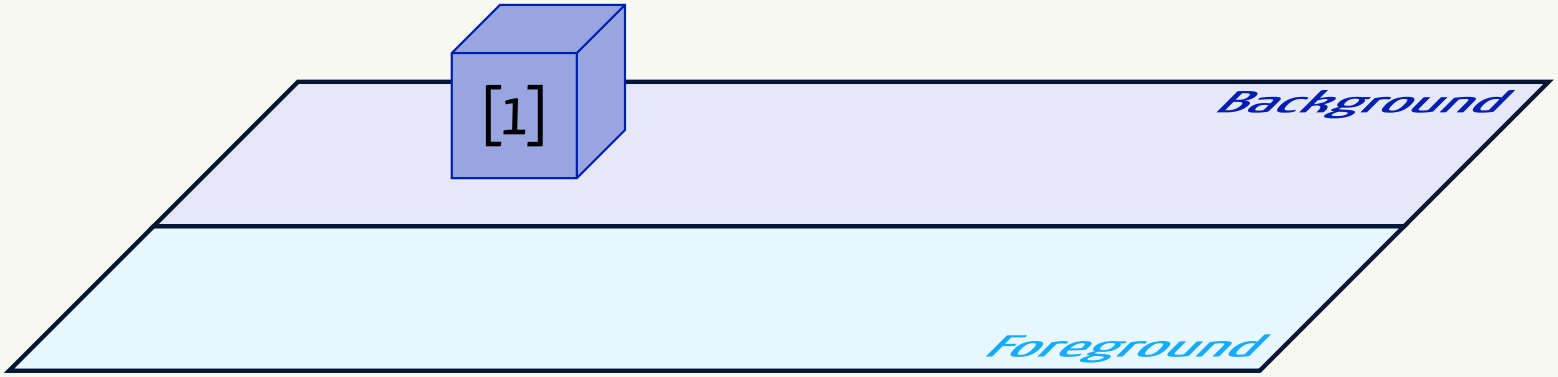

```
$ emacs Loops.tex
# Process in foreground, shell not usable
```

Job control: A graphical overview




```
$ emacs Loops.tex
# Process in foreground, shell not usable
# -> suspend via CTRL-Z
# You get your shell back, but emacs is blocked
[1]+  Stopped                 emacs Loops.tex
```

Job control: A graphical overview




```
$ emacs Loops.tex
# Process in foreground, shell not usable
# -> suspend via CTRL-Z
# You get your shell back, but emacs is blocked
[1]+  Stopped                 emacs Loops.tex
$ bg
# Alternative syntax:
#   bg %1      <- Using the jobnumber
#   bg %+      <- The last started in bg, or suspended from fg
#   bg %%      <- Equivalent to %+ (default if no job specified)
#   bg %ema    <- Job whose command begins with 'ema'
#   bg %?mac   <- Job whose command contains 'mac'
# It is an error if there is more than one matching job!
[1]+ emacs Loops.tex &
```

Handling processes in a script

Handling processes in a script

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script

- What we have discussed so far is very handy in interactive sessions
- In a script, job control is turned off by default and you will hardly need it!
- What is sometimes needed is to run processes in background
- In such a case, it is important to be aware of
  - what the `kill` builtin does
  - what is the `wait` builtin meant for
  - how to handle processes within a script → use of `$!`

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script

- What we have discussed so far is very handy in interactive sessions
- In a script, job control is turned off by default and you will hardly need it!
- What is sometimes needed is to run processes in background
- In such a case, it is important to be aware of
  - what the `kill` builtin does
  - what is the `wait` builtin meant for
  - how to handle processes within a script → use of `$!`
- Even **more important**, you have to understand the parent-child relationship!

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does $\longrightarrow$ i.e. `init` will pick up orphaned processes

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does → i.e. `init` will pick up orphaned processes

System processes

A process PID will **never** be freed up for use after the process dies until the parent process waits for the PID to see whether it ended and retrieve its exit code. If the parent ends, the process is returned to `init`, which does this for you. This is important for one major reason: if the parent process manages its child process, it can be absolutely certain that, even if the child process dies, no other new process can accidentally recycle the child process PID until the parent process has waited for that PID and noticed the child died. This gives the parent process the guarantee that the PID it has for the child process will **always** point to that child process, whether it is alive or a "zombie". Nobody else has that guarantee.

Greg's Wiki

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does → i.e. `init` will pick up orphaned processes

System processes

A process PID will **never** be freed up for use af… the proces… until th… nt proc…s waits for the PID to see whether it ended and re… exit… ss is returned to `i…t`, which d… parent process manag… ocess dies, no oth… he parent process has wai… parent process the guarantee that the PID it h… ch… process w… point to that child process, whether it is alive or a "zombie". No…y else has that guarantee.

Greg's Wiki

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

Handling processes in a script: PIDs and parents

- In UNIX, processes are identified by a number called a PID (for Process IDentifier)
- Each **running** process has a unique identifier
- Do not assume anything about PIDs: For all intents and purposes, they are random!
- Each UNIX process also has a parent process which is the process that started it.
- The parent process can change to the `init` process if the parent process ends before the new process does → i.e. `init` will pick up orphaned processes

What happens in the shell

Shells aggressively reap their child processes and store the exit status in memory, where it becomes available to your script upon calling `wait`. But because the child might have already been reaped before you call wait, there is no zombie to hold the PID.

The kernel is free to reuse that PID, and your guarantee has been violated!

Greg's Wiki

We will see an explicit example in few slides

This article on Process Management is good (long and advanced in its second part, though). Refer to it to discover more.

A bit of syntax: Retrieving a process PID

- The `$!` special parameter holds the PID of **the most recently executed background job**

```
#!/bin/bash

#...
command &
pid=$!
# ...
```

- You cannot reliably determine when or how a process was started purely from its identifier number (PID): do not make any assumption!
- As you might know, there are some metadata stored together with the PID, such as the **process name** and the **issued command** to start the process. **Never rely on those!**

Stay away from parsing the process tree!

UNIX comes with a set of handy tools, among which is `ps`. This is a very helpful utility that you can use from the command line to get an overview of what processes are running. However, `ps` output is unpredictable, highly OS-dependent, and not built for parsing!

Do not parse it in shell scripts! Ever!

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

- If a job specification is given, all processes in the job are waited for
- Supplying the `-f` option, when job control is enabled, forces `wait` to wait for each `pid` or `jobspec` to terminate before returning its status, instead of returning when it changes status

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
# In a Bash script
date
{ sleep 1; exit 1; } & { sleep 3; exit 2; } &
{ sleep 5; exit 3; } & { sleep 7; exit 4; } &
wait # All currently active child processes are waited for
date
```

```
$ ./script-above.bash
Wed  4 Sep 16:13:27 CEST 2019
Wed  4 Sep 16:13:34 CEST 2019
```

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
# In a Bash script
date; sleep 3 & pid=$!
echo "PID=$pid";  ps -p $pid;  wait $pid; date
```

```
$ ./script-above.bash
Wed  4 Sep 16:17:19 CEST 2019
PID=11353
  PID TTY          TIME CMD
11353 pts/11   00:00:00 sleep
Wed  4 Sep 16:17:22 CEST 2019
```

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
# In a Bash script (-n option since Bash v4.3)
{ sleep 1; exit 1; } & { sleep 3; exit 2; } &
{ sleep 2; exit 3; } &
for f in {1..3}; do
   wait -n; echo "wait return status: $? at $(date +'%X')"
done
```

```
$ ./script-above.bash
wait return status: 1 at 16:36:16
wait return status: 3 at 16:36:17
wait return status: 2 at 16:36:18
```

A bit of syntax: The `wait` builtin

```
wait [-fn] [jobspec or pid ...]
```

- `wait` waits until the child process specified by each process ID `pid` or job specification `jobspec` exits and return the exit status of the last command waited for
- If no arguments are given, all currently active child processes are waited for, and the return status is zero
- `wait -n` waits for a single job to terminate and returns its exit status
- If not an active child process of the shell is passed to `wait`, the return status is **127**

```
$ wait 1234
bash: wait: pid 1234 is not a child of this shell
$ echo $?
127
```

A bit of syntax: The `kill` builtin

```
kill [-s sigspec] [-n signum] [-sigspec] jobspec or pid
```

- Despite its name, the `kill` command sends a signal to a process
- There are different way to specify a signal, use what you prefer
- If no signal is specified, SIGTERM (15) is sent

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
$ kill -l 15
TERM
```


The `kill` builtin offers more complex functionalities, like sending a signal to a process group when the used PID is negative. Read about it in the manual.

A bit of syntax: The `kill` builtin

```
kill [-s sigspec] [-n signum] [-sigspec] jobspec or pid
```

- Despite its name, the `kill` command sends a signal to a process
- There are different way to specify a signal, use what you prefer
- If no signal is specified, SIGTERM (15) is sent

`man 2 kill`

> If signal is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

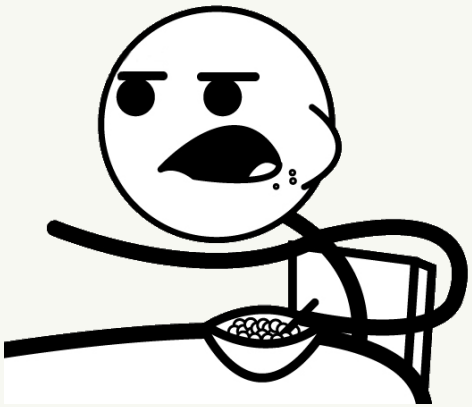
```
1 # Waiting for a process to end
2 # (when you have permissions to send signals)
3 while kill -0 "${pid}"; do
4     sleep 1
5 done
```


The `kill` builtin offers more complex functionalities, like sending a signal to a process group when the used PID is negative. Read about it in the manual.

A potentially catastrophic hidden danger

A potentially catastrophic hidden danger

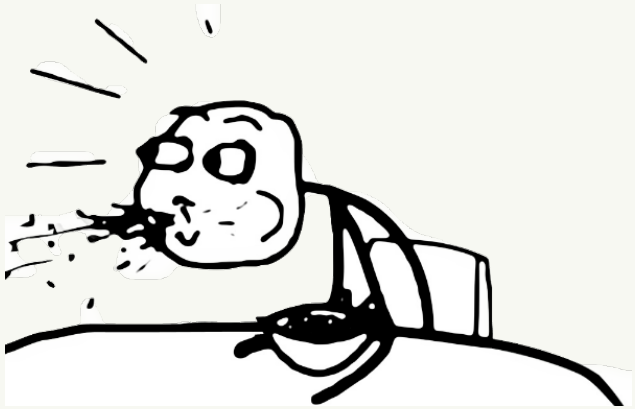
```
1  #!/bin/bash
2                                  #  EXAMPLE 1: wait
3  ( exit 12 ) &
4  pid=$!
5  while { sleep 0 & [ "$pid" != "$!" ]; }; do
6      :
7  done
8  wait "$pid"
9  echo "$?"
```



A potentially catastrophic hidden danger

```
#!/bin/bash
                                #  EXAMPLE 1: wait
( exit 12 ) &
pid=$!
while { sleep 0 & [ "$pid" != "$!" ]; }; do
    :
done
wait "$pid"
echo "$?"
```

```
$ ./script-above.bash
0   # Yes, 0 and not 12... WHAAAAAT?
```



A potentially catastrophic hidden danger

```
#!/bin/bash
                                #  EXAMPLE 1: wait
( exit 12 ) &
pid=$!
while { sleep 0 & [ "$pid" != "$!" ]; }; do
    :
done
wait "$pid"
echo "$?"
```

```
$ ./script-above.bash
0   # Yes, 0 and not 12... WHAAAAAT?
```

Sad but true!

The way shells are programmed, as soon as a child process dies, the shell will call wait() on it right away (storing the termination status as part of its internal state), which will free the PID for reuse by another process.

A potentially catastrophic hidden danger

```
10 #!/bin/bash
11                                  #  EXAMPLE 2: kill
12 long_running_command &
13 pid=$!
14 echo "Killing long_running_command on PID ${pid} in 24h"
15 sleep 86400
16 echo 'Time up!'
17 kill "${pid}"
```

Who will receive the SIGTERM signal?!

Sad but true!

The way shells are programmed, as soon as a child process dies, the shell will call wait() on it right away (storing the termination status as part of its internal state), which will free the PID for reuse by another process.

I absolutely need to know if the PID is the right one, what should I do?

I absolutely need to know if the PID is the right one, what should I do?

Do not use Bash!

Yes, implement the code launching a background process in C or Python, Perl, Ruby, etc. **Not in shell.** Those will not have this problem, since they won't reap children by default (like the shell does) and you will have to do it explicitly there. Or consider launching background processes using a system manager, such as `systemd`.

I absolutely need to know if the PID is the right one, what should I do?

Do not use Bash!

Yes, implement the code launching a background process in C or Python, Perl, Ruby, etc. **Not in shell.** Those will not have this problem, since they won't reap children by default (like the shell does) and you will have to do it explicitly there. Or consider launching background processes using a system manager, such as `systemd`.

However

Note that the kernel will typically try hard to avoid reusing PIDs, at least try to delay reusing a PID, exactly because in some cases there are no guarantees that the PID hasn't been reused, so the kernel tries to minimise this situation where a signal will be delivered to the wrong process.

I absolutely need to know if the PID is the right one, what should I do?

Do not use Bash!

Yes, implement the code launching a background process in C or Python, Perl, Ruby, etc. **Not in shell.** Those will not have this problem, since they won't reap children by default (like the shell does) and you will have to do it explicitly there. Or consider launching background processes using a system manager, such as `systemd`.

Alternatively

You can save the start time of the original process and, before using it, check that the start time of the process with that PID matches what you saved. The pair PID, start-time is a unique identifier for the processes in Linux. However, this does not solve the issue here.

```
$ sleep 10 &
[1] 4451
$ ps -p 4451 -h -o lstart
Thu Sep  5 18:22:43 2019
```

A work-around for exit codes

- If you have several (maybe long) processes in background during the execution of a script and you are interested in their exit code, you can write them to a file
- Alternatively, a possible idea is to use redirection faking a file and taking advantage of the fact that **in Bash the processes inside the process substitution are not waited for**

```
#!/bin/bash
{
    command_pid=$!
    # ...
    another_command &
    # ...
    read <&3 exitCode
    if [[ "${exitCode}" -eq 0 ]]; then
        echo "command was successful!"
    fi
} 3< <(command > logfile 2>&1; echo "$?")
```

A work-around for exit codes

- If you have several (maybe long) processes in background during the execution of a script and you are interested in their exit code, you can write them to a file
- Alternatively, a possible idea is to use redirection faking a file and taking advantage of the fact that **in Bash the processes inside the process substitution are not waited for**

```
12 #!/bin/bash
13 {
14     sleep_pid=$!
15     while { ( exit 12 ) & [ "${sleep_pid}" != "$!" ]; }; do
16         :
17     done
18     wait "${sleep_pid}"
19     waitCode=$?
20     read <&3 exitCode
21     echo "Command in process substitution: ${exitCode}"
22     echo "Command in             while loop: ${waitCode}"
23 } 3< <(sleep 1; echo "$?")
```

A work-around for exit codes

- If you have several (maybe long) processes in background during the execution of a script and you are interested in their exit code, you can write them to a file
- Alternatively, a possible idea is to use redirection faking a file and taking advantage of the fact that in Bash the processes inside the process substitution are not waited for

```
#!/bin/bash
{
    sleep_pid=$!
    while { ( exit 12 ) & [ "${sleep_pid}" != "$!" ]; }; do
        :
    done
    wait "${sleep_pid}"
    waitCode=$?
    read <&3 exitCode
    echo "Command in process substitution: ${exitCode}"
    echo "Command in           while loop: ${waitCode}"
} 3< <(sleep 1; echo "$?")
```

```
$ ./script-above.bash
Command in process substitution: 0
Command in           while loop: 12
```

Process Management: Conclusions

- It is a potentially tough world: Be scared but not too much
- Bash might not be the most appropriate tool
- Be aware of the truth (what we just discussed)
- If you want to have something running in background, why do you want so?
- There are tools which you might consider:
  1. `systemd` (system manager)
  2. `timeout` (run a command with a time limit)
  3. `xargs -P` or `parallel` (to run independent tasks in parallel)

Process Management: Conclusions

SIGTERM vs SIGKILL cartoon

- It is a potentially tough world: Be scared but not too much
- Bash might not be the most appropriate tool
- Be aware of the truth (what we just discussed)
- If you want to have something running in background, why do you want so?
- There are tools which you might consider:
  1. `systemd` (system manager)
  2. `timeout` (run a command with a time limit)
  3. `xargs -P` or `parallel` (to run independent tasks in parallel)

About using `kill -9`

Do not use kill -9, ever. For any reason. **Unless you wrote the program to which you're sending the SIGKILL, and know that you can clean up the mess it leaves.** Because you're debugging it. If a process is not responding to normal signals, it's probably in "state D" (as shown on `ps u`), which means it's currently executing a system call. If that's the case, you're probably looking at a dead hard drive, or a dead NFS server, or a kernel bug, or something else along those lines.

And you won't be able to kill the process anyway, SIGKILL or not.

Greg's Wiki

Traps

Moulin on the Breiðamerkurjökull

Motivation

Consider the following script (not nice code):

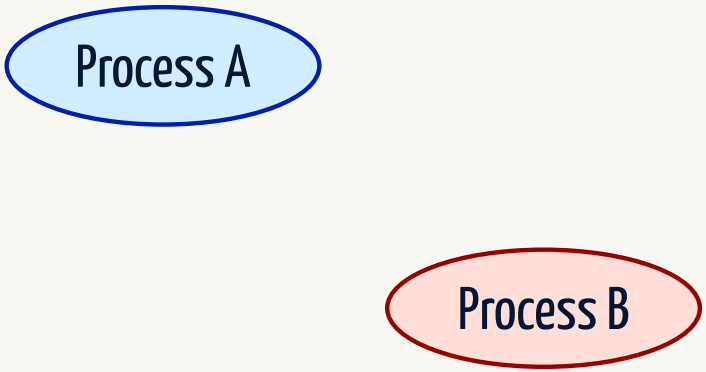
```
#!/bin/bash

function CreateAuxiliaryFiles(){
    # ...
}
function CleanAuxiliaryFiles(){
    # ...
}
CreateAuxiliaryFiles
gnuplot "${temporaryFileToPlot}"
if [[ "${savePlot}" = 'TRUE' ]]; then
    pdflatex "${outputFilename}.tex"
fi
CleanAuxiliaryFiles
exit 0
```

What happens if the script is terminated by the user, e.g. via CTRL-C?

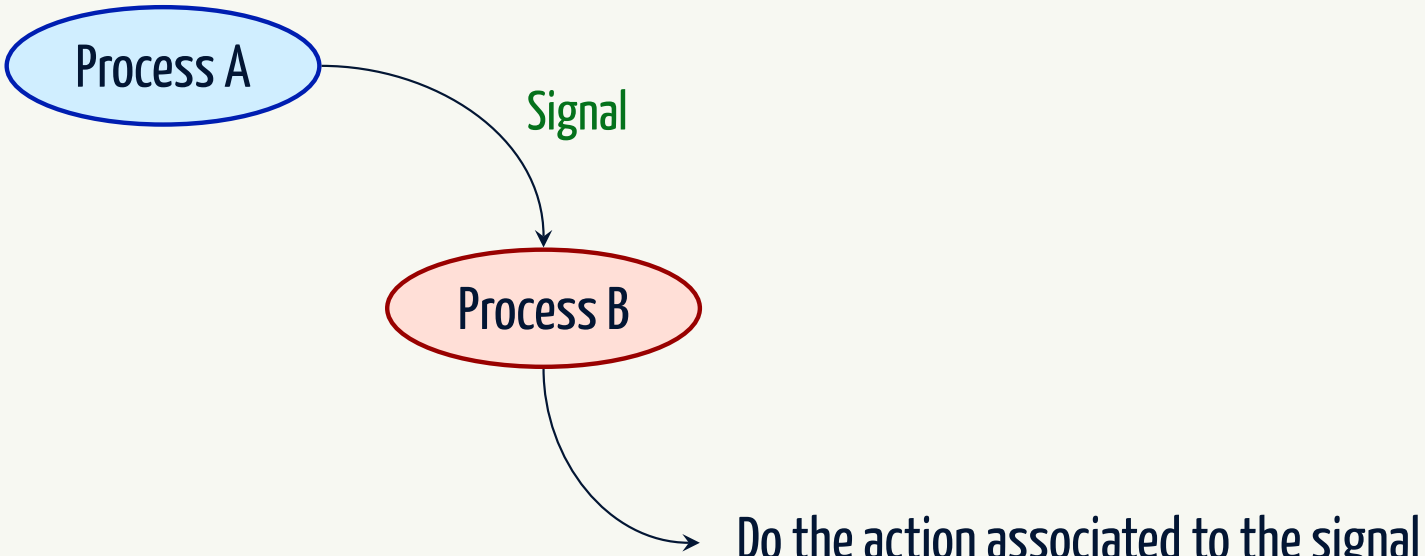
Signal handlers or traps

- As you know you can send signals to processes via the `kill` builtin
- However, in shell scripts, **it is possible to associate a behaviour upon receiving a signal!**
- If you think about it, it is a very powerful technique
- In order to perform an action when a signal is received, a trap has to be set up



Signal handlers or traps

- As you know you can send signals to processes via the `kill` builtin
- However, in shell scripts, **it is possible to associate a behaviour upon receiving a signal!**
- If you think about it, it is a very powerful technique
- In order to perform an action when a signal is received, a trap has to be set up



Signal handlers or traps

```
trap 'some code' signal(s)
```

Using this form, a signal handler is set up for each signal in the list

When one of these signals is received, the commands in the first argument will be executed

```
trap '' signal(s)
```

Using this form, each signal in the list will be ignored. Most scripts should not do this!

```
trap - signal(s)
```

Using this form, each signal in the list will be restored to its default behaviour

```
trap signal
```

Legacy syntax: As the previous but for one signal only! { Prefer the previous, slightly more explicit syntax }

```
trap
```

With no arguments, print a list of signal handlers

Common used signals

HUP Hang Up. The controlling terminal has gone away.

INT Interrupt. The user has pressed the interrupt key (usually **Ctrl-C** or **DEL**).

QUIT Quit. The user has pressed the quit key (usually **Ctrl-**). Exit and dump core.

KILL Kill. This signal cannot be caught or ignored. Unconditionally fatal. **No cleanup possible.**

TERM Terminate. This is the default signal sent by the `kill` builtin.

EXIT Not really a signal. In a Bash script (non-interactive), an EXIT trap is run on any exit, signalled or not. In other POSIX shells only when the shell process exits.

Remember

If you are asking a program to terminate, you should always use SIGTERM. This will give the program a chance to catch the signal and clean up. If you use SIGKILL, the program cannot clean up, and **may leave files in a corrupted state**.

A simple example

Common use case

Traps can be set up to intercept a fatal signal, perform cleanup, and then exit gracefully.

```
1  #!/bin/bash
2
3  temporaryFile=$(mktemp) || exit
4  trap 'rm -f "${temporaryFile}"' EXIT
```

Use a function whenever you need to achieve a more complex task

```
5  #!/bin/bash
6
7  function CleanAuxiliaryFiles(){ # If you trap INT (CTRL-C)
8      # ...                       # do not forget to exit unless
9  }                               # you have reasons not to do so
10 trap 'CleanAuxiliaryFiles' EXIT
```

Only in Bash!

The special name EXIT is preferred for any signal handler that simply wants to clean up upon exiting. So to clean up, just trap EXIT and call a cleanup function from there. **Don't trap a bunch of signals.**

When is the signal exactly handled?

A subtle but important feature

When Bash is executing an external command in the foreground, it does not handle any signals received until the foreground process terminates.

```
1 #!/bin/bash
2
3 echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
4 trap 'echo "Doing some cleaning before exiting!"' EXIT
5 echo "Wait 1h"
6 sleep 3600
```

- If you kill the script using `kill -s INT` from another terminal (not with CTRL-C), bash will wait for `sleep` to exit before calling the trap
  → **That's probably not what you expect!**

When is the signal exactly handled?

A subtle but important feature

When Bash is executing an external command in the foreground, it does not handle any signals received until the foreground process terminates.

```
7  #!/bin/bash
8
9  echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
10 trap 'echo "Doing some cleaning before exiting!"' EXIT
11 echo "Wait 1h"; sleep 3600 & wait $!
12 # Note that sleep 3600 will not be killed and will
13 # continue to run when you send a INT signal!
```

- If you kill the script using `kill -s INT` from another terminal (not with CTRL-C), bash will wait for `sleep` to exit before calling the trap
  → **That's probably not what you expect!**
- A work-around is to use a **builtin** that will be interrupted, such as `wait`
- Any bash internal command will be interrupted by a (non-ignored) incoming signal!

When is the signal exactly handled?

A subtle but important feature

When Bash is executing an external command in the foreground, it does not handle any signals received until the foreground process terminates.

```
14 #!/bin/bash
15
16 echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
17 unset -v pid
18 trap 'echo "Cleaning!"; [[ $pid ]] && kill "${pid}"' EXIT
19 echo "Wait 1h"
20 sleep 3600 & pid=$!; wait
```

- If you kill the script using `kill -s INT` from another terminal (not with CTRL-C), bash will wait for `sleep` to exit before calling the trap
  → **That's probably not what you expect!**
- A work-around is to use a **builtin** that will be interrupted, such as `wait`
- Any bash internal command will be interrupted by a (non-ignored) incoming signal!
- If you want the background job to be killed when the script is killed, add that to the trap!

What indeed is CTRL-C doing?

Let us consider again the previous example

```
#!/bin/bash

echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
trap 'echo "Doing some cleaning before exiting!"' EXIT
echo "Wait 1h"; sleep 3600
```

- As said, sending to this script the INT signal from another terminal does not terminate it
- However, CTRL-C does terminate it... but why?

Note that you can't rely on the process group ID of a script to be the same as $$, as that depends greatly on how the script was started.

What indeed is CTRL-C doing?

Let us consider again the previous example

```
#!/bin/bash

echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
trap 'echo "Doing some cleaning before exiting!"' EXIT
echo "Wait 1h"; sleep 3600
```

- As said, sending to this script the INT signal from another terminal does not terminate it
- However, CTRL-C does terminate it... but why?
- Because **processes are organised in groups**:
  - The leader process (i.e. the process that created the group)
  - Any other process started by the leader process
- **CTRL-C sends the INT signal to the entire process group**
- Terminals keep track of the foreground process group: When receiving a CTRL-C, they send the SIGINT to the entire foreground group!

Note that you can't rely on the process group ID of a script to be the same as $$, as that depends greatly on how the script was started.

What indeed is CTRL-C doing?

Let us consider again the previous example

```
#!/bin/bash

echo "My PID is $$ and my PGID is $(ps -h -p $$ -o pgid)"
trap 'echo "Doing some cleaning before exiting!"' EXIT
echo "Wait 1h"; sleep 3600
```

- As said, sending to this script the INT signal from another terminal does not terminate it
- However, CTRL-C does terminate it... but why?
- Because **processes are organised in groups**:
  - The leader process (i.e. the process that created the group)
  - Any other process started by the leader process
- **CTRL-C sends the INT signal to the entire process group**
- Terminals keep track of the foreground process group: When receiving a CTRL-C, they send the SIGINT to the entire foreground group!

```
kill -s INT -123 # will kill the process group with the ID 123
```


Note that you can't rely on the process group ID of a script to be the same as $$, as that depends greatly on how the script was started.

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT
- The idea is that, if your script calls `vi` for instance, and you press CTRL-C within `vi` to cancel an action, that should not be considered as a request to abort the script

What does it mean?

Imagine you're writing a script and that script exits normally upon receiving SIGINT.
That means that if that script is invoked from another bash script,
CTRL-C will no longer interrupt that other script!

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT

The `ping` command returns with 0 when host is reachable (the ping has been answered) and non-zero otherwise when interrupted (which is the only way for ping to return in that case).

```
1 #!/bin/bash
2 for index in {1..254}; do
3     ping -c 2 "192.168.1.${index}"
4 done
```

CTRL-C will send a INT signal to the script, but since `ping` just returns 0 or 1 and is not killed by SIGINT, only the **current** `ping` will terminate and the `for` loop will continue!

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT

Commands that don't have a SIGINT handler (like `sleep`) or **do the right thing of killing themselves with SIGINT upon receiving SIGINT** (like `bash` itself does) don't exhibit the problem!

```
5  #!/bin/bash
6  index=1
7  while [[ "${index}" -le 100 ]]; do
8    printf "%d " "$((index++))"
9    sleep 1
10 done; echo  # This script terminates correctly via CTRL-C
```

Trapping SIGINT and SIGQUIT: Be careful!

- Bash is among a few shells that implement a **wait and cooperative exit** approach at handling SIGINT/SIGQUIT delivery
- When interpreting a script, upon receiving a SIGINT,
  1. **it doesn't exit straight away**, instead
  2. it waits for the currently running command to return and
  3. it only exits – **by killing itself with SIGINT** – if that command was also killed by that SIGINT

Take-home lesson

If you choose to set up a handler for SIGINT (rather than using the EXIT trap), you should be aware that **a process that exits in response to SIGINT should kill itself with SIGINT rather than simply exiting**, to avoid causing problems for its caller. The same goes for SIGQUIT.

```
trap 'DoYourStuffHere; trap - INT; kill -s INT "$$"' INT
```

Awk and Sed

A double wave

Why do we need them?

- In science, most of our time in the terminal is spent acting on files
- Mastering file handling in general can allow us to simplify the data analysis software
- Typical operations might be
  - Prepare data for plotting in a given format
  - Search patterns in a file
  - Extract portion of a file
  - Transform a file (e.g. remove trailing spaces)
  - ...

GNU Core Utilities

`head`	`cat`	`sum`	`column`	`sort`	`join`	`expand`
`tail`	`tac`	`cksum`	`paste`	`uniq`	`comm`	`unexpand`
	`cut`	`md5sum`			`tr`	`split`

Checkout the complete GNU coreutils list.

Why do we need them?

- In science, most of our time in the terminal is spent acting on files
- Mastering file handling in general can allow us to simplify the data analysis software
- Typical operations might be
  - Prepare data for plotting in a given format
  - Search patterns in a file
  - Extract portion of a file
  - Transform a file (e.g. remove trailing spaces)
  - ...

GNU Core Utilities

head
tail
column
paste
sum
sed
awk
expand
unexpand
split

Checkout the complete GNU coreutils list.

Sed: A marvellous utility

The awful truth about `sed`

Sed is extremely powerful. Unfortunately, most people never learn its real power. The language is simpler than the average user thinks, because the documentation is far from being ideal. Sed has several commands, but most people only learn the substitute command `'s'`. And the GNU manual begins exactly with examples about the `'s'` command.

Not bad!

The substitute command is just one command. Sed has at least 20 different commands for you. You can even write Tetris in it (not to mention that it's Turing complete).

Some references

- The official GNU manual { The PDF is 85 pages long... }
- A very nice tutorial { Bash code there often uses deprecated features, but you should know it by now! }
- One-liners, with some comments
- More one-liners, explained but advanced!

The PDF manual is 85 pages long. What should I learn here?

- We will focus on the abstract idea of **how** `sed` processes a file
- Some of the simplest commands (acting on a single line) will be introduced
- Learning Sed is almost like learning a new programming language, mastering it is probably not needed for a physicist either

The PDF manual is 85 pages long. What should I learn here?

- We will focus on the abstract idea of **how** `sed` processes a file
- Some of the simplest commands (acting on a single line) will be introduced
- Learning Sed is almost like learning a new programming language, mastering it is probably not needed for a physicist either

```
# Invocation
sed [OPTIONS...] [SCRIPT] [INPUTFILE...]

# General command syntax to be put in [SCRIPT]
[addr]X[options]
```

The PDF manual is 85 pages long. What should I learn here?

- We will focus on the abstract idea of **how** `sed` processes a file
- Some of the simplest commands (acting on a single line) will be introduced
- Learning Sed is almost like learning a new programming language, mastering it is probably not needed for a physicist either

```
# Invocation
sed [OPTIONS...] [SCRIPT] [INPUTFILE...]

# General command syntax to be put in [SCRIPT]
[addr]X[options]
```

- `X` is a `sed` command
- `[addr]` is an optional line address
  If `[addr]` is specified, the command `X` will be executed only on the matched lines;
  `[addr]` can be a single line number, a regular expression, or a range of lines
- Additional `[options]` are used for some `sed` commands
- Quote the commands to avoid shell globbing conflicts
  → usually you want to use single quotes

Some of the most used commands

s/regexp/replacement/[flags]

Match the regular-expression against the content of the pattern space.
If found, replace matched string with replacement.

p

Print the pattern space, up to the first newline.

d

Delete the pattern space; immediately start next cycle.

=

Print the current input line number (with a trailing newline).

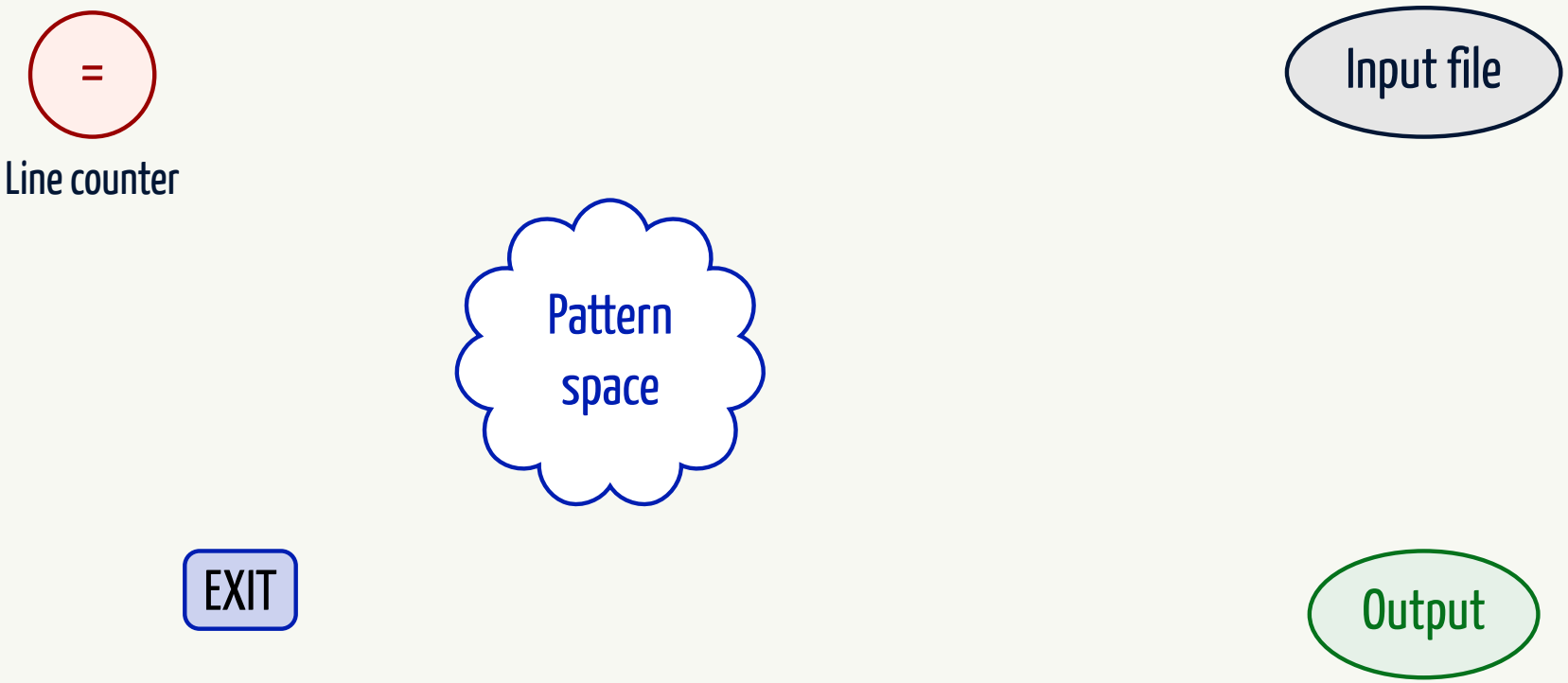
{ cmd ; cmd ...}

Group several commands together.

q

Exit `sed` without processing any more commands or input.

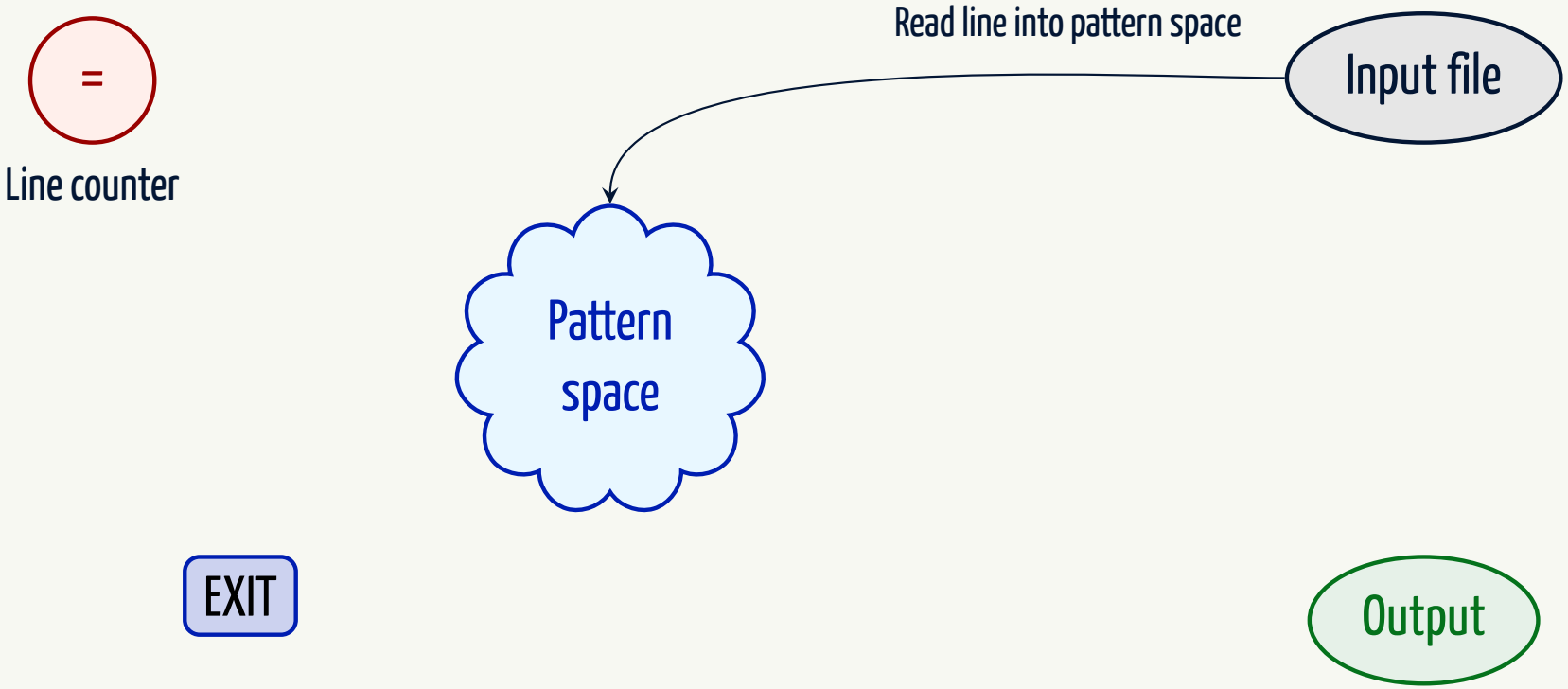
A `sed` cycle: Being line oriented



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

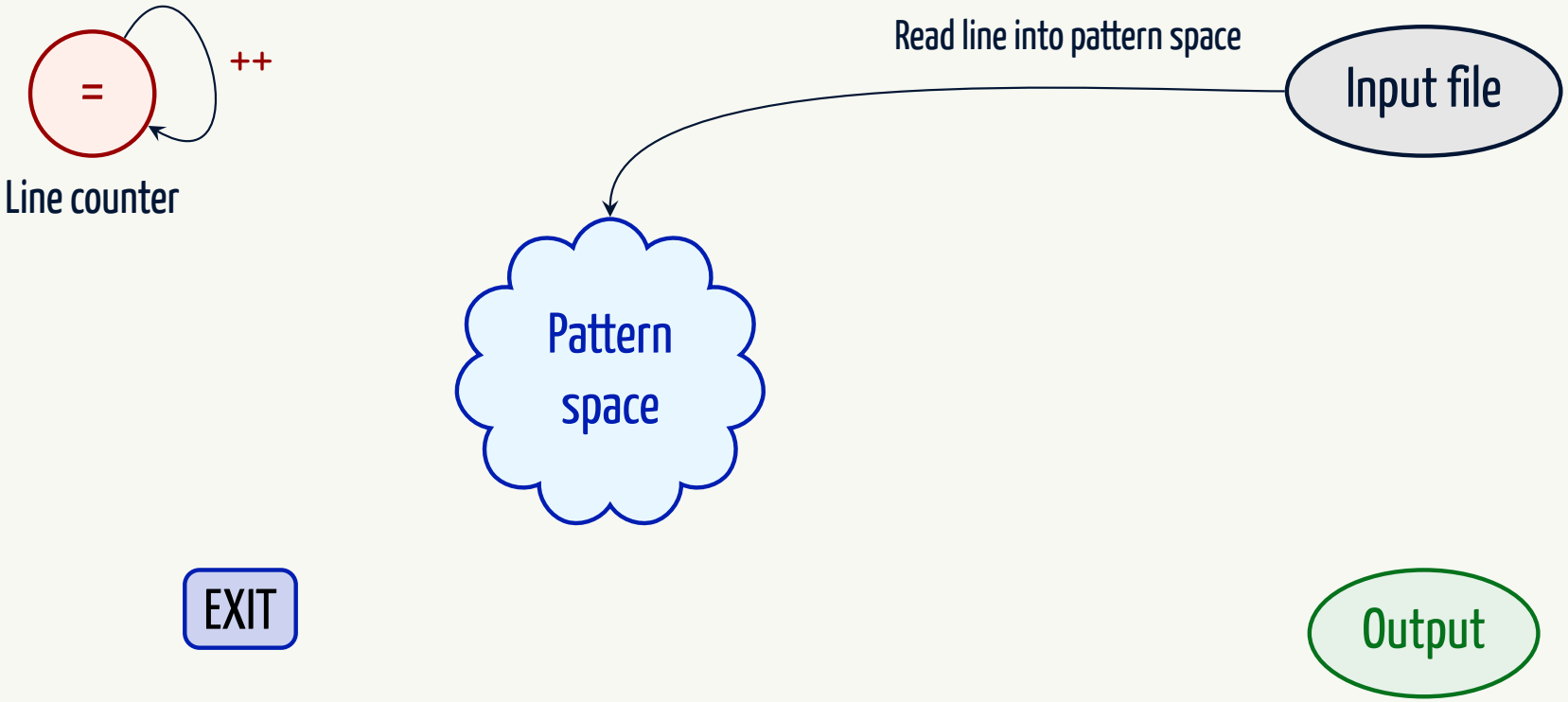
1. The next line is read from the input file and placed it in the pattern space.
   If the end of file is found, and if there are additional files to read, the current file is closed, the next file is opened, and the first line of the new file is placed into the pattern space.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

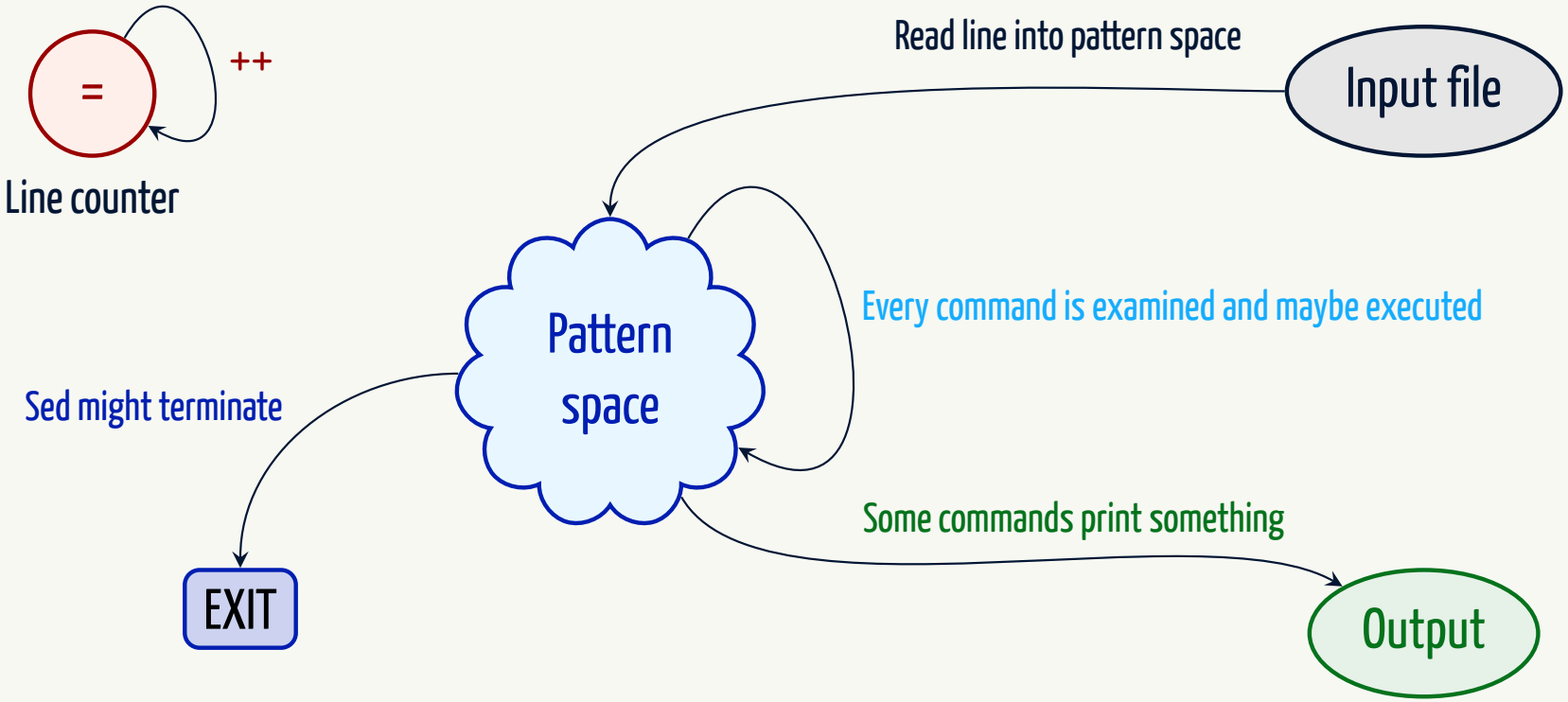
2 The line count is incremented by one.
Opening a new file does not reset this number.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

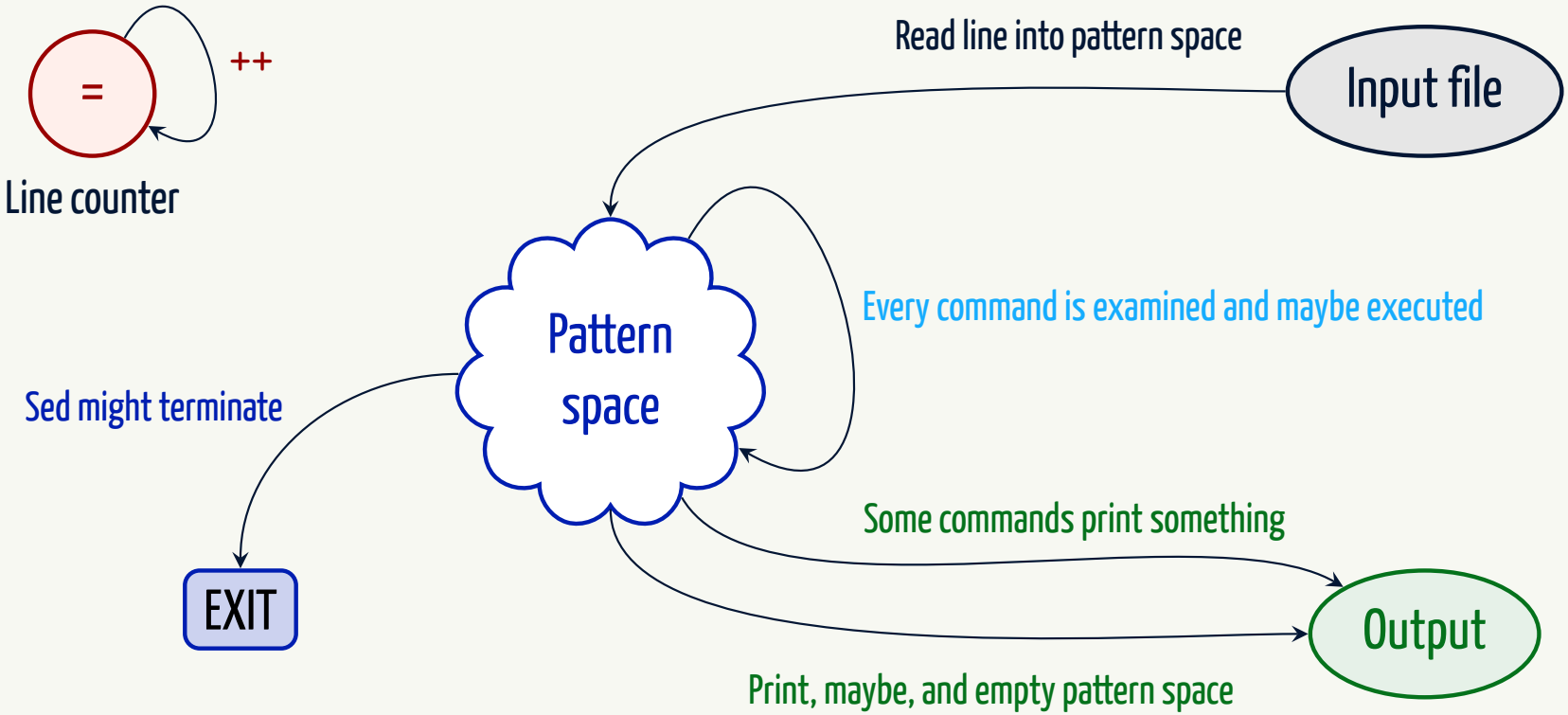
3. Each `sed` command is examined. If there is a restriction placed on the command, and the current line in the pattern space meets that restriction, the command is executed. Some commands, like `'d'` cause `sed` to go to the top of the loop. The `'q'` command causes `sed` to stop. Otherwise the next command is examined.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

A `sed` cycle: Being line oriented

4 After all of the commands are examined, the pattern space is printed to the output (unless sed has the optional "-n" argument) and the pattern space is emptied.



There is also a hold space and this flow gets modified by more complex commands. This is a good approximation.

Address restrictions

A command can be limited to some lines using restrictions

- A positive line number (use $ to refer to the last line)
- A regular expression `/regex/` or `\•regex•` where • is any character { not to be matched! }
- Use an exclamation mark after the address to negate that address
- A comma separated pair to specify a range of lines

```
                    # Command applies
3                   #  - to line 3 only
/^#/                #  - to lines starting by #
\%^//%              #  - to lines starting by //
5!                  #  - not to line 5
1,3                 #  - to first three lines
3,$                 #  - from line 3 to the end
1,/one/             #  - till the first line matching 'one'
/begin/,/end/       #  - from the first line matching 'begin'
                    #    to the first matching 'end' (included)
```


Basic Regular Expression (BRE) VS Extended Regular Expression (ERE)

The main commands: Substitute

`s/regexp/replacement/[flags]`

The `'s'` command attempts to match the pattern space against the supplied regular expression `regexp`; if the match is successful, then that portion of the pattern space which was matched is replaced with `replacement`.

- The `/` is called delimiter and can replaced with any character
- The delimiter can appear in `regexp` or `replacement` only if escaped
- Unescaped `&` characters reference the whole matched portion of the pattern space
- The replacement can contain `\1`, ..., `\9` references, which refer to the portion of the match which is contained between the first, ..., nineth `\(` and its matching `\)`
- Important-to-know flags are

`g`	Apply the replacement to all matches to the `regexp`, not just the first
number	Only replace the number-th match of the `regexp` (between 1 and 512)
`p`	If the substitution was made, then print the new pattern space

The main commands: Substitute

```
$ seq 11 12 | sed 's/12/twelve/'
11
twelve
$ seq 11 12 | sed 's/1/X/'
X1
X2
$ seq 11 12 | sed 's/1/X/g'
XX
X2
$ seq 11 12 | sed 's/1/X/2'
1X
12
$ seq 11 12 | sed 's/2/Y/p'
11
1Y
1Y
$ sed 's/[0-9]\+/(&)/' <<< 'Wed 09 Sep 2019'
Wed (09) Sep 2019
$ sed 's/[0-9]\+/(&)/g' <<< 'Wed 09 Sep 2019'
Wed (09) Sep (2019)
$ sed -r 's/[0-9]+/(&)/g' <<< 'Wed 09 Sep 2019'
Wed (09) Sep (2019) # -r option for ERE
```

The main commands: Substitute

```
$ seq 11 12 | sed 's/1/&&/'
111
112
$ sed 's/\([a-z]\+\) \([a-z]\+\)/\2 \1/' <<< 'hello world'
world hello
$ sed 's/^\(.\)\(.\)\(.\)\(.\) /\3\2\1\4/' <<< 'nice star'
cinestar
$ printf "=%0.s" {1..20} | sed 's/./&:/10'; echo
==========:==========
$ cd /usr/local/bin
$ sed 's/\/usr\/local\/bin/\/common\/bin/' <<< "${PWD}/emacs"
/common/bin/emacs
$ sed 's_/usr/local/bin_/common/bin_' <<< "${PWD}/emacs"
/common/bin/emacs
$ sed 's:/usr/local/bin:/common/bin:' <<< "${PWD}/emacs"
/common/bin/emacs
$ sed 's|/usr/local/bin|/common/bin|' <<< "${PWD}/emacs"
/common/bin/emacs
$ value='3 5'; sed 's/Y/'"${value}"'/' <<< "XYZ"
X3 5Z
$ value='3 5'; sed 's/Y/'${value}'/' <<< "XYZ"
sed: -e expression #1, char 5: unterminated `s' command
```

The main commands: Print

- Print out the pattern space
- Note that the printing induced by the `p` command is unrelated to the printing done by `sed` at the end of the cycle
- Hence, this command is usually only used in conjunction with the `-n` command-line option of `sed` (whose long name is `--quiet` or `--silent`)
- It is commonly used together with an address restriction

```
$ sed 'p' <<< 'Echo me'
Echo me
Echo me
$ sed -n 'p' <<< 'Only once'
Only once
$ seq 1 9 | sed -n '5 p'
5
$ seq 1 9 | sed -n '7,$ p'
7
8
9    # Try out: seq 10 | sed -n '1~3 p'
```

The main commands: Delete

- Delete the pattern space and immediately start next cycle
- It is used to delete lines from the input file(s)
- It is commonly used together with an address restriction

```
$ seq 10 | sed '3,$ d'
1
2
$ sed '2 d' <<< $'A\n nice but\n cold place'
A
 cold place
$ sed '/b/,/f/ d' <<< $'a\nb\nc\nd\ne\nf\ng'
a
g
$ sed '/^#/ d' filename  # Remove bash comments
# ... <- look up option -i of sed to edit the file in place
$ sed '/^$/ d' filename  # Remove empty lines
```

The main commands: quit, line number and groups

- The `'q'` command exits `sed` without processing any more commands or input. This command exits at the end of the cycle, i.e. it prints the current pattern space.
- The `'='` command prints out the current input line number (with a trailing newline).
- A group of commands may be enclosed between `{` and `}` characters. This is particularly useful when you want a group of commands to be triggered by a single address (or address-range) match.

```
$ seq 15 | sed '2q'
1
2
$ sed '=' <<< $'aaa\nbbb'
1
aaa
2
bbb
$ sed -n '$=' filename  # Equivalent to 'wc -l < filename'
$ seq 3 | sed -n '2{s/2/X/ ; p}'
X
```

Awk: Another marvellous utility

A bit of history

1 part `egrep`	1 part `snobol`
2 parts `ed`	3 parts `C`

Blend all parts well using `lex` and `yacc`. Document minimally and release.

After eight years, add another part `egrep` and two more parts `C`. Document very well and release.

The name `awk` comes from the initials of its designers: Alfred V. Aho, Peter J. Wein-berger, and Brian W. Kernighan. The original version of `awk` was written in 1977 at AT&T Bell Laboratories.

Awk manual

Some references

Awk can be considered for all intents and purposes as a programming language!

- The official GNU manual { The PDF is 570 pages long... }
- A base tutorial

The PDF manual is 570 pages long!! What should I learn here?

- We will focus on the abstract idea of **how** `awk` processes a file
- Some of the simplest aspects will be introduced
- Awk is so powerful that just knowing something is worth it; mastering it is probably not needed for a physicist, though

There are other special pattern, which are less common and which can be found in the manual.

The PDF manual is 570 pages long!! What should I learn here?

- We will focus on the abstract idea of **how** `awk` processes a file
- Some of the simplest aspects will be introduced
- Awk is so powerful that just knowing something is worth it; mastering it is probably not needed for a physicist, though

```
# Invocation
awk [OPTIONS...] 'program' [INPUTFILE...]

# General structure of program (single quoted, usually)
  BEGIN { action }      # Optional block
pattern { action }      #
...                     # { action } can be omitted
pattern { action }      #
    END { action }      # Optional block
```


There are other special pattern, which are less common and which can be found in the manual.

The PDF manual is 570 pages long!! What should I learn here?

- We will focus on the abstract idea of **how** `awk` processes a file
- Some of the simplest aspects will be introduced
- Awk is so powerful that just knowing something is worth it; mastering it is probably not needed for a physicist, though

```
# Invocation
awk [OPTIONS...] 'program' [INPUTFILE...]

# General structure of program (single quoted, usually)
  BEGIN { action }      # Optional block
pattern { action }      #
...                     # { action } can be omitted
pattern { action }      #
    END { action }      # Optional block
```

- `BEGIN` and `END` are special patterns, indeed
- The default action is print the record (i.e. the line)
- If only the `BEGIN` block is given, no input is read
- If no input is given but one is needed, `awk` reads from standard input

There are other special pattern, which are less common and which can be found in the manual.

The `awk` spirit: A data driven processing

1. Before starting processing the input file(s), the `BEGIN` block, if present, is processed.
   There is no default action for this block.
2. Input is read until the **R**ecord **S**eparator is found (generally an end-of-line).
3. Some default record processing is done, e.g. splitting it in fields (removing separators), and builtin variables are set.
4. Processing of the part of input is done, executing the blocks in the given orders.
   An action in a block might jump to reading from the input, skipping the following blocks.
5. Input is read again, i.e. go to 2.
6. After having processed the last bunch of input (i.e. that terminated by the end-of-file of the last file), the `END` block, if present, is processed.
   There is no default action for this block.

Some terminology

Record: The bunch of input read at each iteration

Field: Each piece in which each record is automatically split

There are exceptions to the flow above (e.g. the `exit` keyword). Use this slide as approximated overview.

8 powerful `awk` builtin variables

You can change this to a string or regular expression

FS Field Separator. White-space by default.

OFS Output Field Separator. Single white-space by default.

RS Record Separator. End of line by default.

ORS Output Record Separator. End of line by default.

Variable automatically set

NR Number of Record.

NF Number of Field.

FILENAME The name of the file being processed.

FNR Number of Record within the File being processed.

The field placeholders

Field placeholders, automatically set

- `$n` The fields can be accessed via $1, ...,$9, $10, ...
- `$0` This special placeholder can be used to access the full record

```
$ awk '{print $1}' <<< $'A 1 \n B 2 \n C 3'
A
B
C
$ awk '{print $1, $10, $(10)}' <<< "1 2 3 4 5 6 7 8 9 A"
1 A A
$ awk '{print $1 "|" $2}' <<< "A:B C::D"
A:B|C::D
$ awk 'BEGIN{FS=":"}{print $1 "|" $2}' <<< "A:B C::D"
A|B C
$ awk 'BEGIN{FS="::"}{print $1 "|" $2}' <<< "A:B C::D"
A:B C|D
$ awk 'BEGIN{FS=":"; OFS="|"}{print $1, $2}' <<< "A:B C::D"
A|B C
$ awk 'BEGIN{ORS="|"}{print $1}' <<< $'A 1 \n B 2 \n C 3'
A|B|C  # without final new line
```

Variables in `awk` and passing Bash variables to `awk`

- Strings that do not refer to keyword or to builtin variables/functions in `awk` programs are treated as variables
- Numeric variables in `awk` are implicitly initialised to 0, use them without worries!
- The `awk` command-line option `-v var=val` sets the `awk` variable `var` to the value `val` before execution of the program begins
- The `-v` option can only set one variable, but it can be used more than once, setting another variable each time Avoid setting `awk` builtin variables!

```
$ awk 'BEGIN{print uninitialisedVariable}'

$ awk 'BEGIN{var++; print var}'
1
$ aVar='Hello'
$ awk -v hi="${aVar}" -v sum=1 'BEGIN{sum++; print hi " " sum}'
Hello 2
$ awk -v index=1 'BEGIN{print index}'
awk: fatal: cannot use gawk builtin `index' as variable name
```

A (very limited) taste of awk by examples

```
$ seq 100 | awk '{sum+=$1}END{print "Gauss answered " sum}'
Gauss answered 5050

$ shuf -i 1-20 -n 100 -r |
> awk '$1>10{high++}END{print "Drawn " high " numbers >10."}'
Drawn 52 numbers >10.

# Print every line that is longer than 80 characters
awk 'length($0) > 80' filename

# Print the length of the longest input line
awk '    { if (length($0) > max) {max = length($0)} }
     END { print max }' filename

# Print every line that has at least one field
awk 'NF > 0' filename

# Count the lines in a file
awk 'END { print NR }' filename
```

A (very limited) taste of awk by examples

```
# Print the even-numbered lines in the data file
awk 'NR % 2 == 0' data

# Sum two columns of a file line by line adding a column
awk '{print $1, $2, $1+$2}' data

# Prepare powers of each column adding columns
awk '{
        for(field=1; field<=NF; field++){
            for(n=1; n<=4; n++){
                printf "%f ", $field**n
            }
        }
        printf "\n";
    }' data
```

As you can imagine, awk has plenty of features!

A (very limited) taste of awk by examples

```
# Throwing two dice 1000 times and getting distribution
$ paste <(shuf -i 1-6 -n 1000 -r) <(shuf -i 1-6 -n 1000 -r) |
> awk '{dice[$1+$2]++}
>      END
>      {
>        for(key in dice){
>          printf "P(%2d) ~ %.3f\n", key, dice[key]/NR
>        }
>      }'
P( 2) ~ 0.038  # 1/36 = 0.028
P( 3) ~ 0.051  # 2/36 = 0.056
P( 4) ~ 0.081  # 3/36 = 0.083
P( 5) ~ 0.100  # 4/36 = 0.111
P( 6) ~ 0.156  # 5/36 = 0.139
P( 7) ~ 0.161  # 6/36 = 0.167
P( 8) ~ 0.148  # 5/36 = 0.139
P( 9) ~ 0.115  # 4/36 = 0.111
P(10) ~ 0.070  # 3/36 = 0.083
P(11) ~ 0.052  # 2/36 = 0.056
P(12) ~ 0.028  # 1/36 = 0.028
```

Good practices

A nice icelandic Christmas tradition

Require a minimum Bash version

Using modern Bash features is important, but be sure you can then use them where your scripts are needed!

- Bash v2.0 ⟶ Released in 1996
- Bash v3.0 ⟶ Released in 2004
- Bash v4.0 ⟶ Released in 2009
- Bash v4.1 ⟶ Released in 2009
- Bash v4.2 ⟶ Released in 2011
- Bash v4.3 ⟶ Released in 2014
- Bash v4.4 ⟶ Released in 2016
- Bash v5.0 ⟶ Released in 2019

```
$ echo "${BASH_VERSINFO[@]}"
4 4 20 1 release x86_64-pc-linux-gnu
$ echo "${BASH_VERSINFO[@]:0:3}" | tr ' ' '.'
4.4.20
# You can use the 'hash' builtin to check command existence
```


Keep in mind that locally install a recent Bash version is pretty straightforward.

Be sensible!

Using Bash v3.0 or higher means you can avoid ancient scripting techniques!

1. When writing Bash scripts, do not use the `[` command.
   Bash has a far better keyword: `[[`
2. It's also time you forget about `` `...` ``. It isn't consistent with the syntax of expansion and is terribly hard to nest without a dose of painkillers. Use `$(...)` instead.
3. And for heaven's sake, **use more quotes!**
   Protect your strings and parameter expansions from word splitting.
   Word splitting will eat your babies if you don't quote things properly.
4. Learn to use parameter expansions instead of `sed`, `awk` or `cut` to manipulate simple strings in Bash.
5. Use built-in arithmetic instead of `expr` to do simple calculations, especially when just incrementing a variable!

Be sensible!

Using Bash v3.0 or higher means you can avoid ancient scripting techniques!

6. Always use the correct shebang. If you're writing a Bash script, you need to put `#!/usr/bin/env bash` at the top of your script.
7. Stay away from scripting examples you see on the Web! If you want to get inspired, understand them **completely**. Mostly any scripts you will find on the Web are broken in some way. Do not copy/paste from them!
8. Almost as important as the result of your code is the **readability of your code**. Chances are that you aren't just going to write a script once and then forget about it.

> Trust me when I say, no piece of code is ever 100% finished, with the exception of some very short and useless chunks of nothingness.
>
> — Greg's Wiki

Being aware is the first step

- Writing good Bash scripts is after all tough
- Bash shell allows you to do quite a lot of things, offering you considerable flexibility
- However, it does very little to discourage misuse and other ill-advised behaviour
- Many awful and dangerous scripts end up in production or even in Linux distributions!!
- The result of these, and also your very own scripts written in a time of neglect, can often be **disastrous**

Important reading

- Bash Pitfalls { 59 discussed items }
- Obsolete and deprecated syntax
- Shell Hall of Shame

Never do anything along these lines in your scripts!

1 Don't ever parse the output of the `ls` command!

```
ls -l | awk '{ print $8 }'   # BAD CODE!
```

- `ls` will mangle the filenames of files if they contain characters unsupported by your locale. As a result, parsing filenames out of ls is **never** guaranteed to actually give you a file that you will be able to find. `ls` might have replaced certain characters in the filename by question marks.
- Secondly, `ls` separates lines of data by newlines. This way, every bit of information on a file is on a new line. **Unfortunately** filenames themselves can **also** contain newlines! This means that if you have a filename that contains a newline in the current directory, it will completely break your parsing and as a result, break your script!
- Last but not least,
  → **the output format** of `ls -l` **is not guaranteed consistent across platforms**.

Use e.g. **globbing** or `stat` or `find`.

🔗 Instructive and very encouraged reading

Never do anything along these lines in your scripts!

2 Don't ever test or filter filenames with the `grep` command!

```
if echo "${file}" | fgrep '.txt'; then  # BAD CODE!

ls *.txt | grep 'story'                 # ALSO BAD CODE!
```

- Unless your `grep` pattern is really smart it will probably not be trustworthy
- In the first example above, the test would match both `story.txt` **and** `story.txt.exe`!
- If you make `grep` patterns that are smart enough, they'll probably be so ugly, massive and unreadable that you shouldn't use them anyway!

The alternative is called **globbing**.

Bash has a feature called **Pathname Expansion**. This will help you enumerate all files that match a certain pattern. Also, you can use globs to test whether a filename matches a certain glob pattern.

Never do anything along these lines in your scripts!

3 Avoid unnecessary use of the `cat` command!

```
cat filename | grep 'pattern'  # BAD CODE!
```

- Don't use `cat` to feed a single file's content to a filter. `cat` is a utility used to **concatenate** the contents of several files together.

To feed the contents of a file to a process you will probably be able to
→ pass the filename as an argument to the program!
If the manual of the program does not specify any way to do this,
→ you should use redirection!

4 Don't use a `for` loop to read the lines of a file!

```
for line in $(<file); do  # BAD CODE!
```

- command substitutions remove any trailing newlines!

Use a `while read` loop instead.

Never do anything along these lines in your scripts!

5 For the love of god and all that is holy, do not use `seq` to count!

```
for index in $(seq 1 10); do  # BAD CODE!
```

- Bash is able enough to do the counting for you. You do not need to spawn an external application (especially a single-platform one) to do some counting and then pass that application's output to Bash for word splitting.

Use one of the following approaches.

- In general, C-style for loops are the best method for implementing a counter

```
for ((i=1; i<=10; i++)); do  # BEST WAY
```

- If you actually wanted a stream of numbers separated by newlines as test input, consider

```
printf '%d\n' {1..10}
```

You can also loop over the result of a sequence expansion if the range is constant, but the shell needs to expand the full list into memory before processing the loop body. This method can be wasteful and is less versatile than other arithmetic loops.

Never do anything along these lines in your scripts!

6 `expr` is a relic of ancient Rome. Do not wield it!

```
i=`expr $i + 1`   # VERY BAD CODE!
```

- You're basically
  - spawning a new process,
  - calling another C program to do some arithmetic for you and
  - returning the result as a string to Bash.

  Bash can do all this itself and so much faster, more reliably and in all, better!

  { no numbers ⟶ string ⟶ number conversions }

In Bash you should use either `let i++` or `((i++))`.

Do not worry!

We all did at least one of the previous in the past.
You learn by your mistakes as soon you know they are mistakes!

Clean code

- Bash is a scripting language
  → this does not mean at all that it has not to fulfil good coding practices
- If you ignore what clean code is, have a look here !
- Use meaningful names and strive for readability.
- If you do tricky stuff in your script, comment it!
- If you need to work on a larger script in Bash,
  - split your source code across files and
  - test your code, for real, i.e. write unit tests!
    Shell unit test framework
- A good/clean code is
  - Readable
  - Maintainable
  - Easy to extend
  - Easy to use
  - Hard to break
  - Testable/Tested

What to do, now?

The Þingvellir national park

Do not be afraid

- You received a huge amount of information
- Nobody can expect to have everything well settled in their head **right now**
- **You have these slides, work on them**
- Links are important to make a further step forward

Do not be afraid

- You received a huge amount of information
- Nobody can expect to have everything well settled in their head **right now**
- **You have these slides, work on them**
- Links are important to make a further step forward

Bash is more than that

You learnt a lot, but there is much more out there!

Well, not yet, I would say!

Unfortunately it is not that easy!

You need practice and our job offers plenty of possibility for that!

So, what should I do now?

1. **It is matter of doing small steps!**
   - Whenever reasonable, give it a chance
   - Take every daily-life opportunity to recall aspects of Bash
   - These three days should be a very good starting point for your work!

So, what should I do now?

1. **It is matter of doing small steps!**
   - Whenever reasonable, give it a chance
   - Take every daily-life opportunity to recall aspects of Bash
   - These three days should be a very good starting point for your work!
2. **I hope you enjoyed the course**
   - I would love to get feedback from each of you
   - Feel free to send me an email ✉
     - Did I meet your expectation? If no, why?
     - Was the lecture **prohibitively** dense? { Dense it had to be... }
     - What would you change–add–remove?

So, what should I do now?

1. **It is matter of doing small steps!**
   - Whenever reasonable, give it a chance
   - Take every daily-life opportunity to recall aspects of Bash
   - These three days should be a very good starting point for your work!
2. **I hope you enjoyed the course**
   - I would love to get feedback from each of you
   - Feel free to send me an email ✉
     - Did I meet your expectation? If no, why?
     - Was the lecture **prohibitively** dense? { Dense it had to be... }
     - What would you change–add–remove?
3. Last but not least, **haven't you already planned your trip to Iceland?** 😉

So, what should I do now?

1. **It is matter of doing small steps!**
   - Whenever reasonable, give it a chance
   - Take every daily-life opportunity to recall aspects of Bash
   - These three days should be a very good starting point for your work!
2. **I hope you enjoyed the course**
   - I would love to get feedback from each of you
   - Feel free to send me an email ✉
     - Did I meet your expectation? If no, why?
     - Was the lecture **prohibitively** dense? { Dense it had to be... }
     - What would you change–add–remove?
3. Last but not least, **haven't you already planned your trip to Iceland?** 😉

Thank you for attending!