Introduction to Bash scripting language

Day 5

Alessandro Sciarra

Z02 – Software Development Center

21 February 2025

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

AxelKrypton

CRC-TR 211
Strong-interaction matter under extreme conditions

Topics of the day

1. Awk (and Sed)
2. Good practices
3. Additional useful tools
4. Bash in real life
5. Summary of the course
6. What to do, now?

The last but not the least

Today we will finish our trip discovering few more tools, having a look back to what we learnt and to what is out there!

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`

A complete GNU coreutils list is available on Wikipedia.

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 cut column sort expand
tail tr paste uniq unexpand
cat sum wc split

sed

awk

A complete GNU coreutils list is available on Wikipedia.

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 of v5.3 is 611 pages long... }
- A base tutorial

The PDF manual is 611 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 611 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 611 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.

Variables 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 and record separators can be specified as regular expression:

```
FS="[:, ]" # Colon, comma or space as field separator
```

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
$ 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.
- Variables are either strings or numbers depending on what is assigned to them!
- **By default, variables are initialized to the empty string, which is 0 if converted to a number.**
- 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
```


The conversion mechanism between strings and numbers can be found in detail in the documentation.

A (very limited) taste of awk by examples

```
$ printf '%d\n' {1..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
```

How do I modify files in place with awk?

- This feature has been missed by awk's users for a long time!
- GNU awk introduced it in v4.1.0 in 2013.
- It is a bit longer to type than `sed -i` option, but it is nice to have.
- Use the `-i inplace` option to modify file(s) in place.
- If you want to keep a backup of the to-be-modified file,
  - use the awk `INPLACE_SUFFIX` variable prior to v5.0
  - use the awk `inplace::suffix` variable from v5.0 on

  to set a suffix for the copy of the file, e.g. `.bak`.

What happens underneath

> For each regular file that is processed, the extension redirects standard output to a temporary file configured to have the same owner and permissions as the original. After the file has been processed, the extension restores standard output to its original destination. If inplace::suffix is not an empty string, the original file is linked to a backup file name created by appending that suffix. Finally, the temporary file is renamed to the original file name. [...] If the program dies prematurely, as might happen if an unhandled signal is received, a temporary file may be left behind.
>
> Awk's manual

How do I modify files in place with awk?

```
$ printf 'foo\nbar\nfoo\n' > file1
$ cp file1 file2
$ ls
file1   file2
$ awk -i inplace '{ gsub(/foo/, "bar") }; { print }' file1
$ cat file1
bar
bar
bar
$ awk --version | head -n1
GNU Awk 5.0.1, API: 2.0 (GNU MPFR 4.0.2, GNU MP 6.2.0)
# awk -i inplace -v INPLACE_SUFFIX='.bak' ...
$ awk -i inplace -v inplace::suffix='.bak'\
>       '{ gsub(/foo/, "bar") }; { print }' file2
$ ls
file1   file2   file2.bak
$ paste file2 file2.bak
bar     foo
bar     bar
bar     foo
```

Sed and Awk: Conclusions

- You should be convinced that they are very powerful tools.
- To have an idea how they work is not hard.
- Lots of documentation, examples and tutorials are available.
- You do not have to study them, learn by **effective** doing!
- When you find examples online looking for a hint, understand them.

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 12.1996
Bash v3.0 ⟶ Released in 08.2004
Bash v3.1 ⟶ Released in 12.2005
Bash v3.2 ⟶ Released in 10.2006
Bash v4.0 ⟶ Released in 02.2009
Bash v4.1 ⟶ Released in 12.2009
Bash v4.2 ⟶ Released in 02.2011
Bash v4.3 ⟶ Released in 02.2014

Best for empty arrays and `-u` shell option:

Bash v4.4 ⟶ Released in 09.2016
Bash v5.0 ⟶ Released in 01.2019
Bash v5.1 ⟶ Released in 12.2020
Bash v5.2 ⟶ Released in 09.2022
Bash v5.3 ⟶ To be released soon

{ Bash v5.3ß was released in 12.2024 }

```
$ echo "${BASH_VERSINFO[@]}"; echo "${BASH_VERSION}"
5 2 26 1 release aarch64-apple-darwin21.6.0
5.2.26(1)-release
$ tr ' ' '.' <<< "${BASH_VERSINFO[@]:0:3}"
5.2.26
# You can use the 'hash' builtin to check command existence
```


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

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: `[[`

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.

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.

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.

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 and script extension. If you're writing a Bash script, you need to put `#!/usr/bin/env bash` at the top of your script.

Be sensible!

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

6. Always use the correct shebang and script extension. 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**. Any script you'll find on the Web is likely to be broken in some way. Do not copy/paste from them!

Be sensible!

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

6. Always use the correct shebang and script extension. 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**. Any script you'll find on the Web is likely to be 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.

Be sensible!

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

6. Always use the correct shebang and script extension. 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**. Any script you'll find on the Web is likely to be 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 { 64 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 **Filename 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.

«The command substitution `$(cat file)` can be replaced by the equivalent but faster `$(< file)`». → Bash manual section 3.5.4

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!

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!

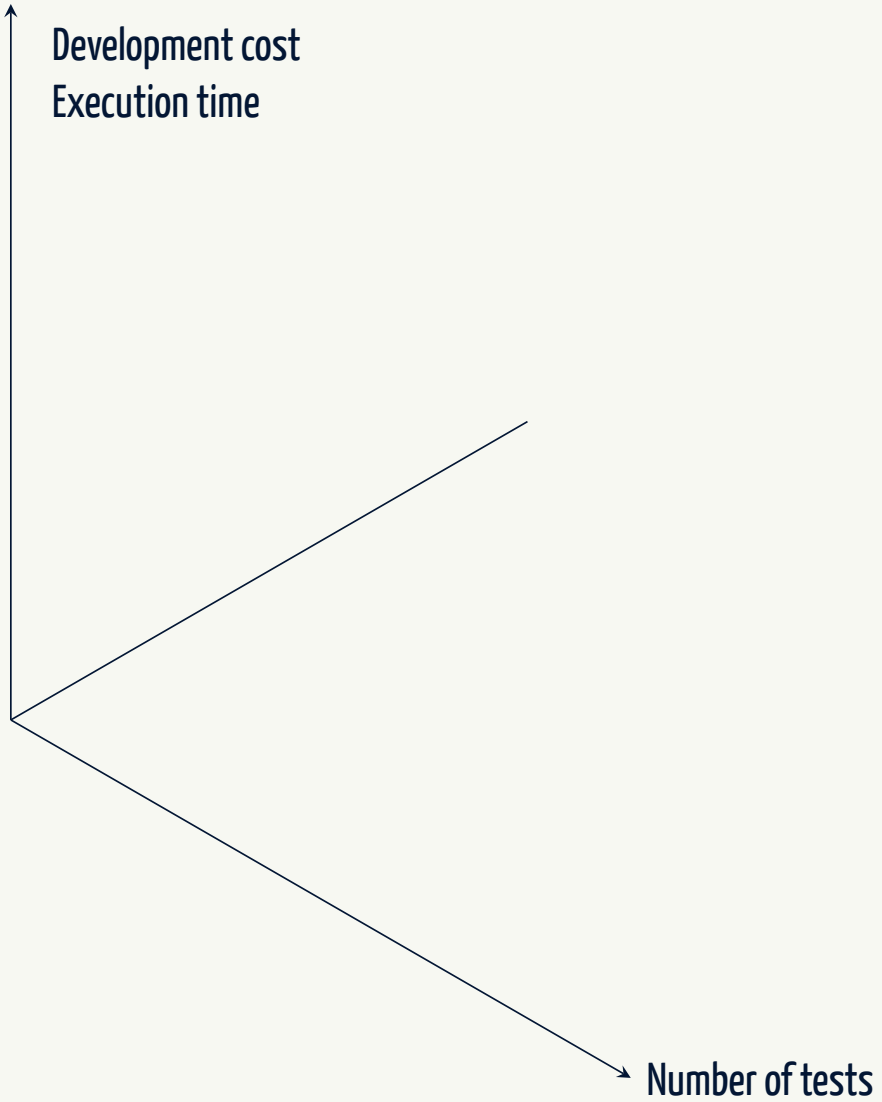
Clean code

- Bash is a scripting language
  → this does not mean at all that it has not to fulfil good coding practices
- Have a look to the first of these presentations ! { Also the second one is worth the read! }
- 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! { More on this in a moment }
- A good/clean code is
  - Readable
  - Maintainable
  - Easy to extend
  - Easy to use
  - Hard to break
  - Testable/Tested
  - ...

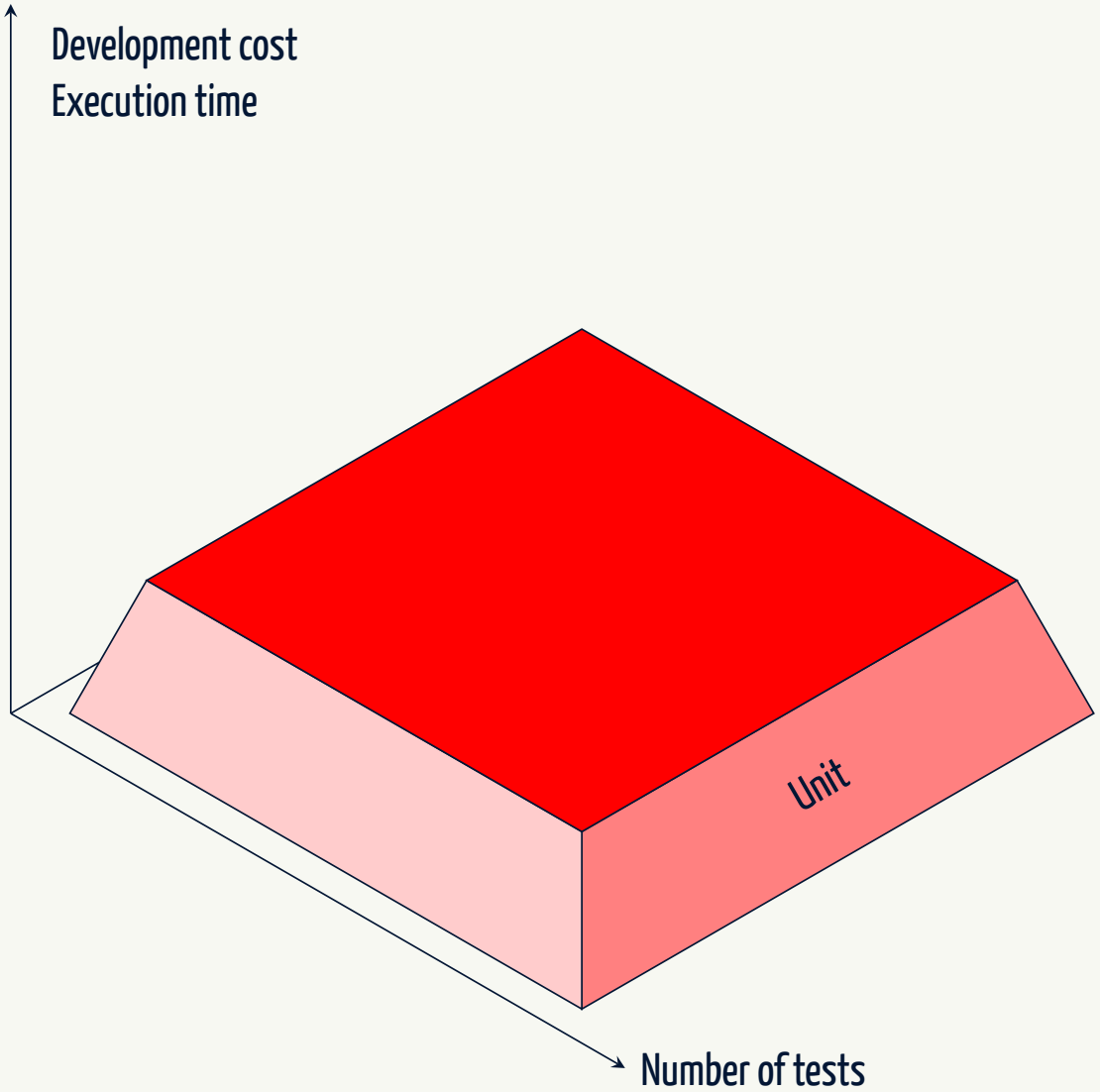
Additional useful tools

Climbing up the Road F985 (Jökulvegur)

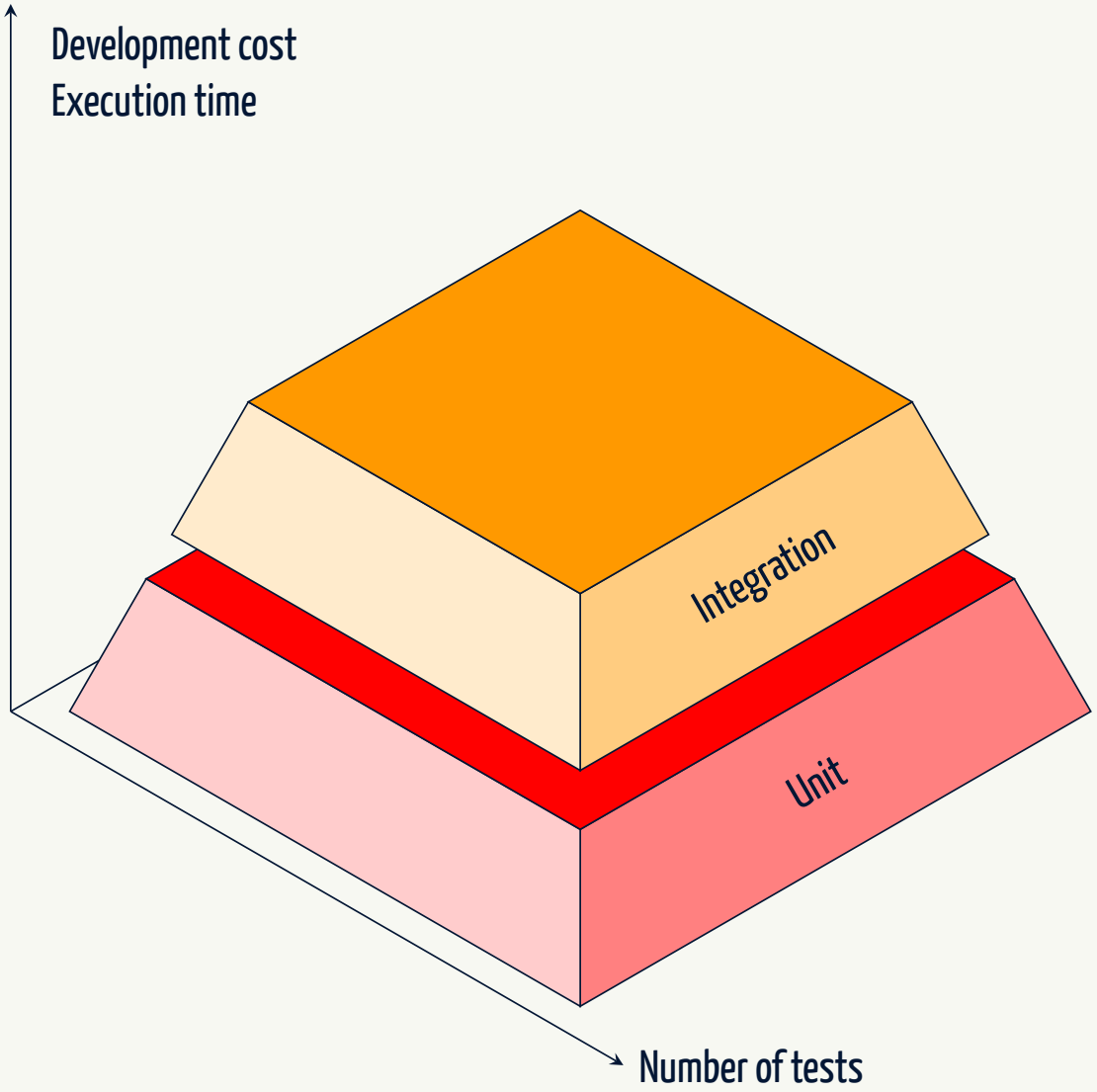
One slide about testing: The tests pyramid



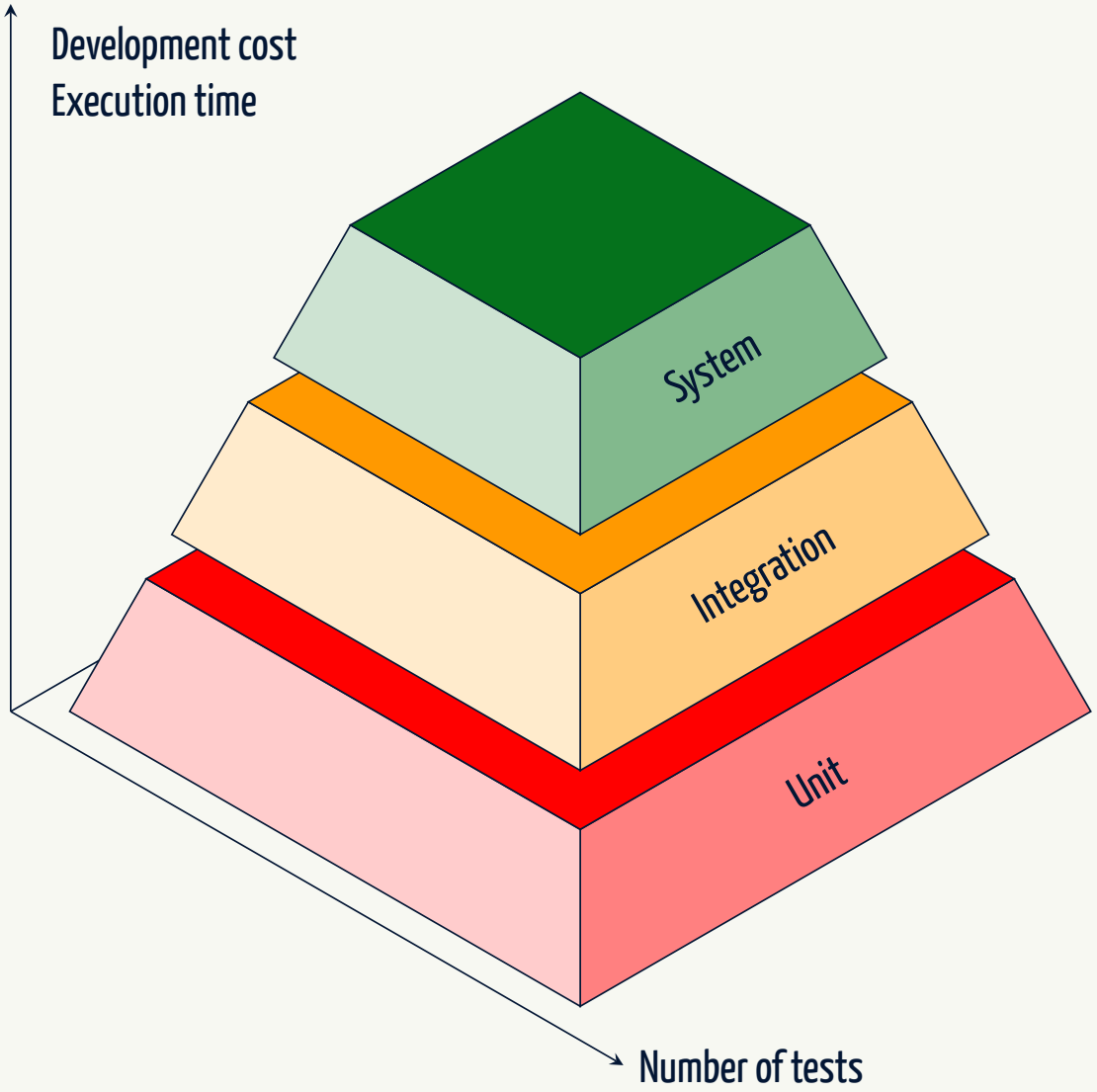
One slide about testing: The tests pyramid



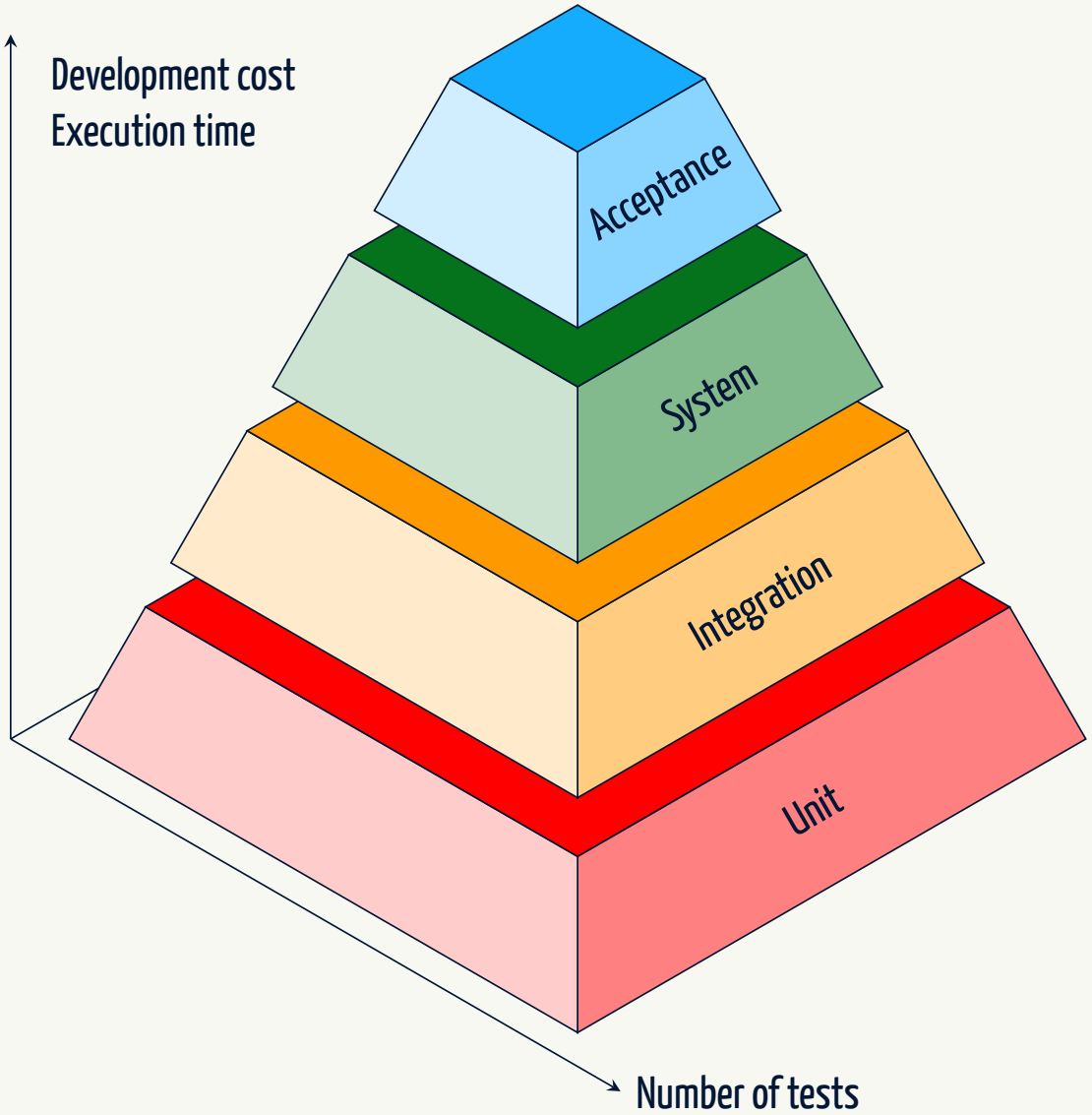
One slide about testing: The tests pyramid



One slide about testing: The tests pyramid

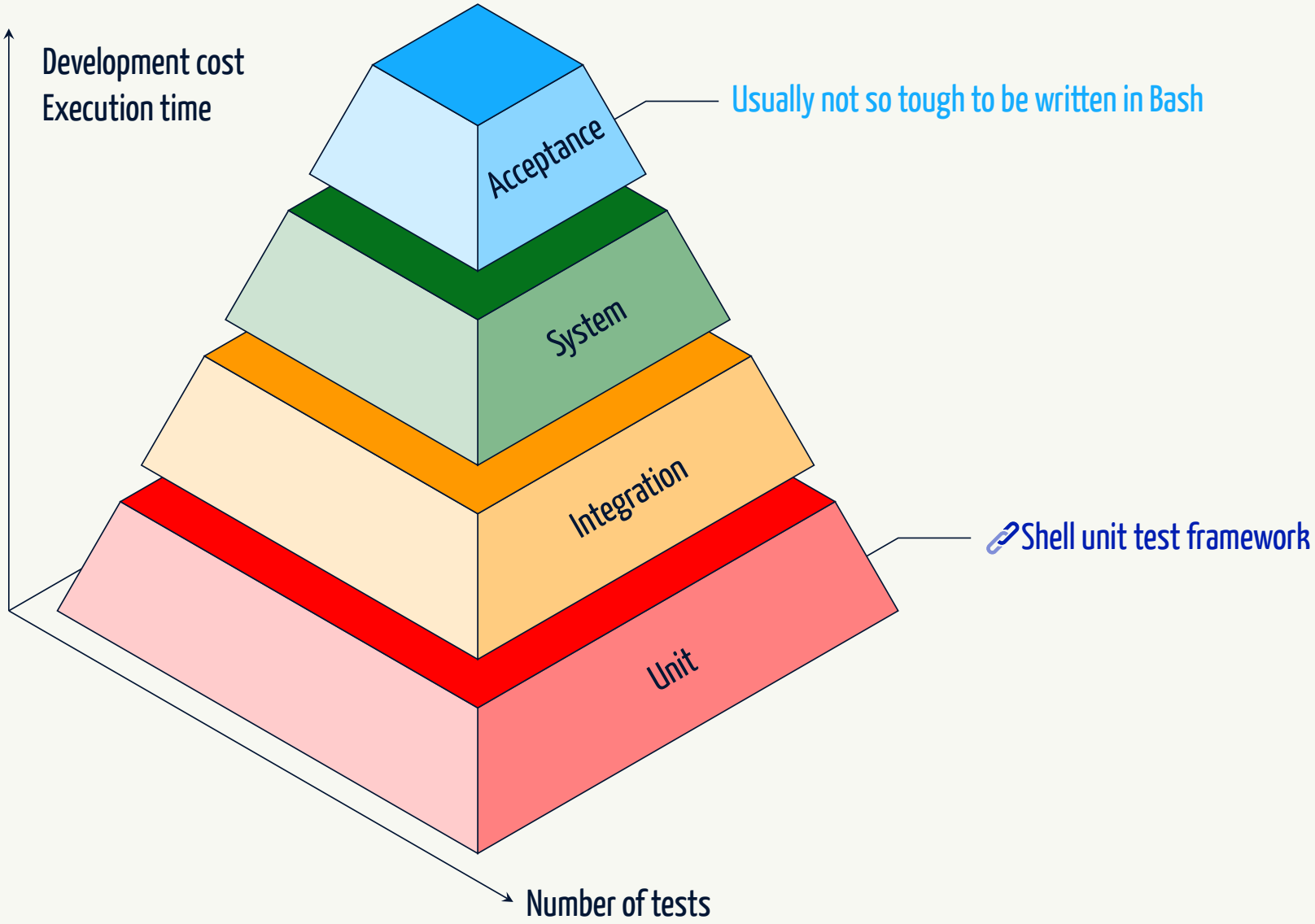


One slide about testing: The tests pyramid



If you want to know more about this subject, also more in general, refer to the second of these presentations .

One slide about testing: The tests pyramid



If you want to know more about this subject, also more in general, refer to the second of these presentations .

My script does not do what it should: What can I do?

1. Trying to debug it by hand is always a possibility... { if you do so, try to do it in a clever way }

My script does not do what it should: What can I do?

1. Trying to debug it by hand is always a possibility... { if you do so, try to do it in a clever way }
2. ...alternatively you can run ShellCheck on it or use it online !

My script does not do what it should: What can I do?

1. Trying to debug it by hand is always a possibility... { if you do so, try to do it in a clever way }
2. ...alternatively you can run ShellCheck on it or use it online !

ShellCheck

ShellCheck is a GPLv3 tool that gives warnings and suggestions for Bash/sh shell scripts:

```
$ shellcheck myscript

Line 4:
if ! grep -q backup=true.* "~/.myconfig"
             ^-- SC2062: Quote the grep pattern so the shell won't interpret it.
                         ^-- SC2088: Tilde does not expand in quotes. Use $HOME.

Line 6:
  echo 'Backup not enabled in $HOME/.myconfig, exiting'
       ^-- SC2016: Expressions don't expand in single quotes, use double quotes for that.

Line 10:
if [[ $1 =~ "-v(erbose)?" ]]
           ^-- SC2076: Don't quote right-hand side of =~, it'll match literally rather than as a regex.

Line 16:
  -iname *.tar.gz \
         ^-- SC2061: Quote the parameter to -iname so the shell won't interpret it.
         ^-- SC2035: Use ./*glob* or -- *glob* so names with dashes won't become options.

Line 18:
  -exec scp {}  “myhost:backups” +
                ^-- SC1110: This is a unicode quote. Delete and retype it (or quote to make literal).
                            ^-- SC1110: This is a unicode quote. Delete and retype it (or quote to make literal).
```

My script does not do what it should: What can I do?

1. Trying to debug it by hand is always a possibility... { if you do so, try to do it in a clever way }
2. ...alternatively you can run 🔗ShellCheck on it or 🔗use it online !

The goals of ShellCheck are

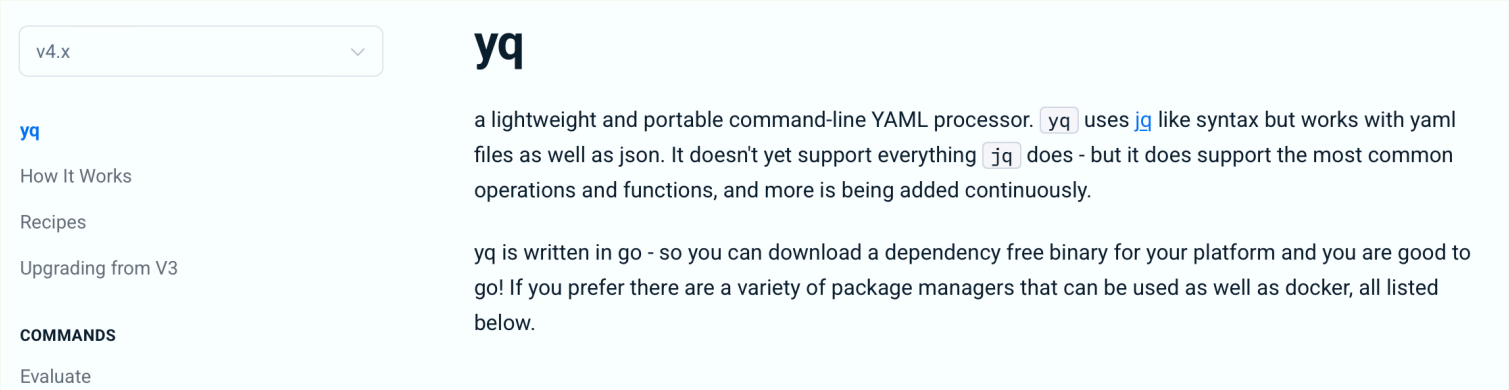
- To point out and clarify typical beginner's syntax issues that cause a shell to give cryptic error messages.
- To point out and clarify typical intermediate level semantic problems that cause a shell to behave strangely and counter-intuitively.
- To point out subtle caveats, corner cases and pitfalls that may cause an advanced user's otherwise working script to fail under future circumstances.

- You can enable it on the fly in your editor
- You can ignore specific warnings/errors using directives
- Every implemented rule has a dedicated Wiki-page { e.g. 🔗https://github.com/koalaman/shellcheck/wiki/SC1000 }

Can I parse YAML and JSON from Bash?

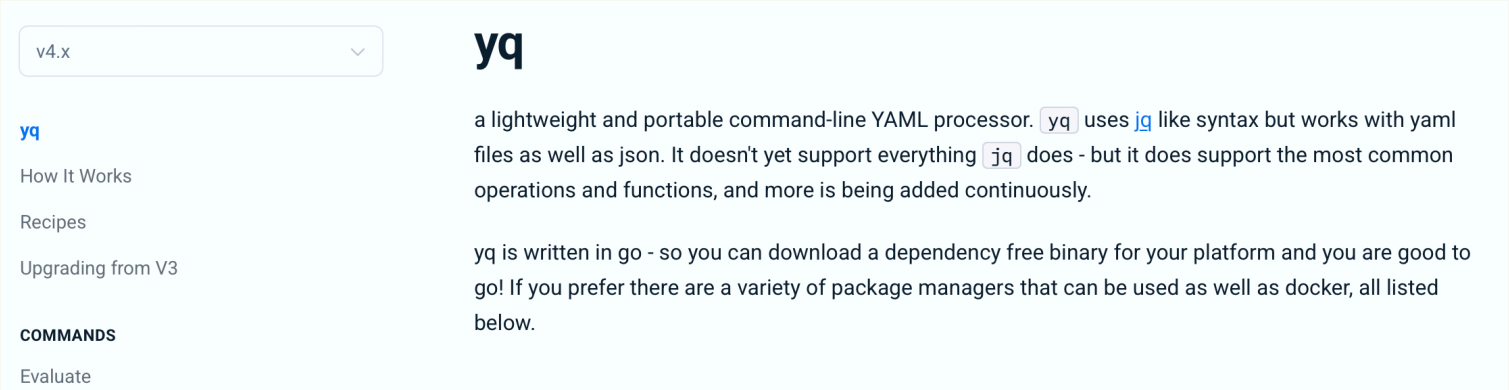
Can I parse YAML and JSON from Bash?

1 YAML? → A wonderful tool: yq

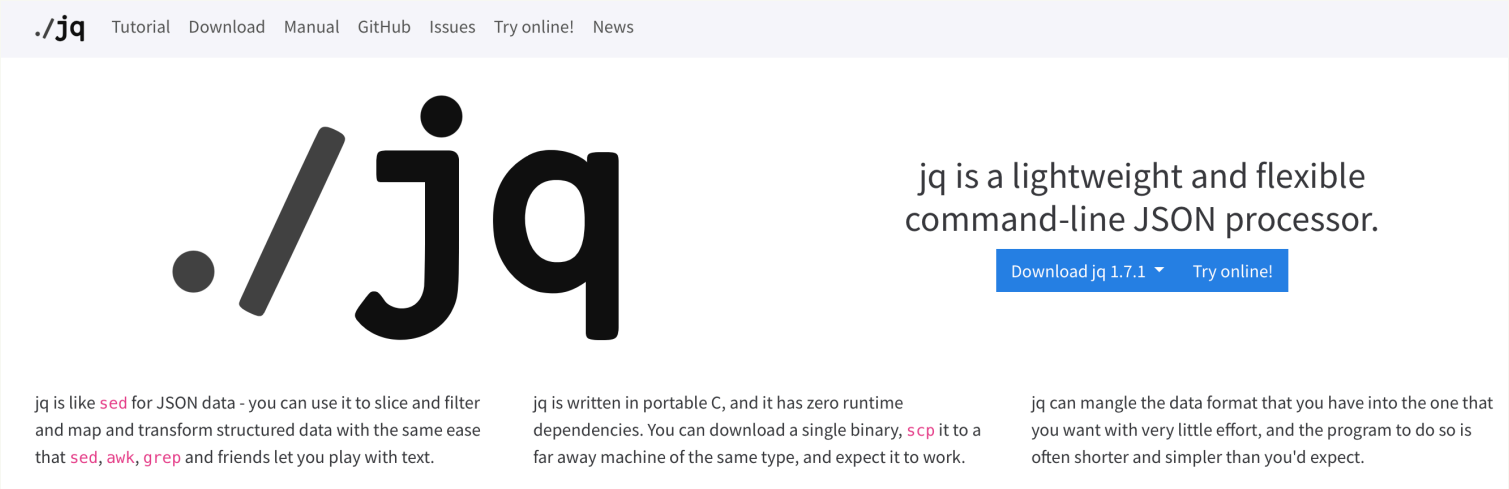


Can I parse YAML and JSON from Bash?

1 YAML? → A wonderful tool: yq



2 JSON? → Another wonderful tool: jq



Can I parse YAML and JSON from Bash?

1. YAML? → A wonderful tool: yq
2. JSON? → Another wonderful tool: jq

A minimalistic example about `yq`

```
$ cat File.yaml
a:
  b:
    - one:
      c: True
    - two: 2
    - three: 3
$ yq '.a.b[0].c' File.yaml
True
```

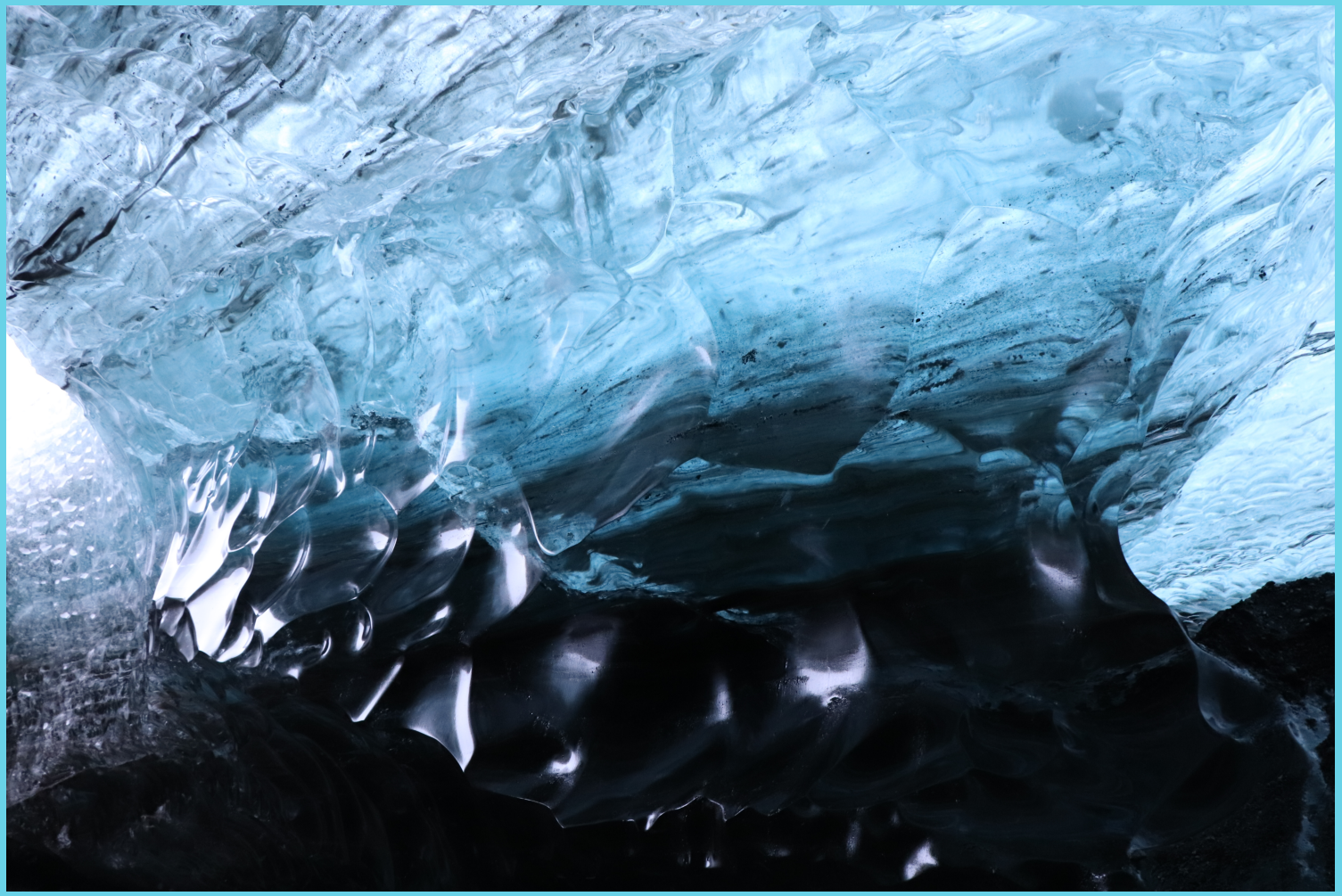
Can I parse YAML and JSON from Bash?

1. YAML? → A wonderful tool: yq
2. JSON? → Another wonderful tool: jq

A minimalistic example about `jq`

```
$ jq '.[0] | {message: .commit.message, name:
    .commit.committer.name}' File.json
{
    "message": "docs: Document repeat(exp)",
    "name": "Nico Williams"
}
```

Bash in real life



The ice under the Breiðamerkurjökull

Medium to large Bash projects I worked on

Medium to large Bash projects I worked on

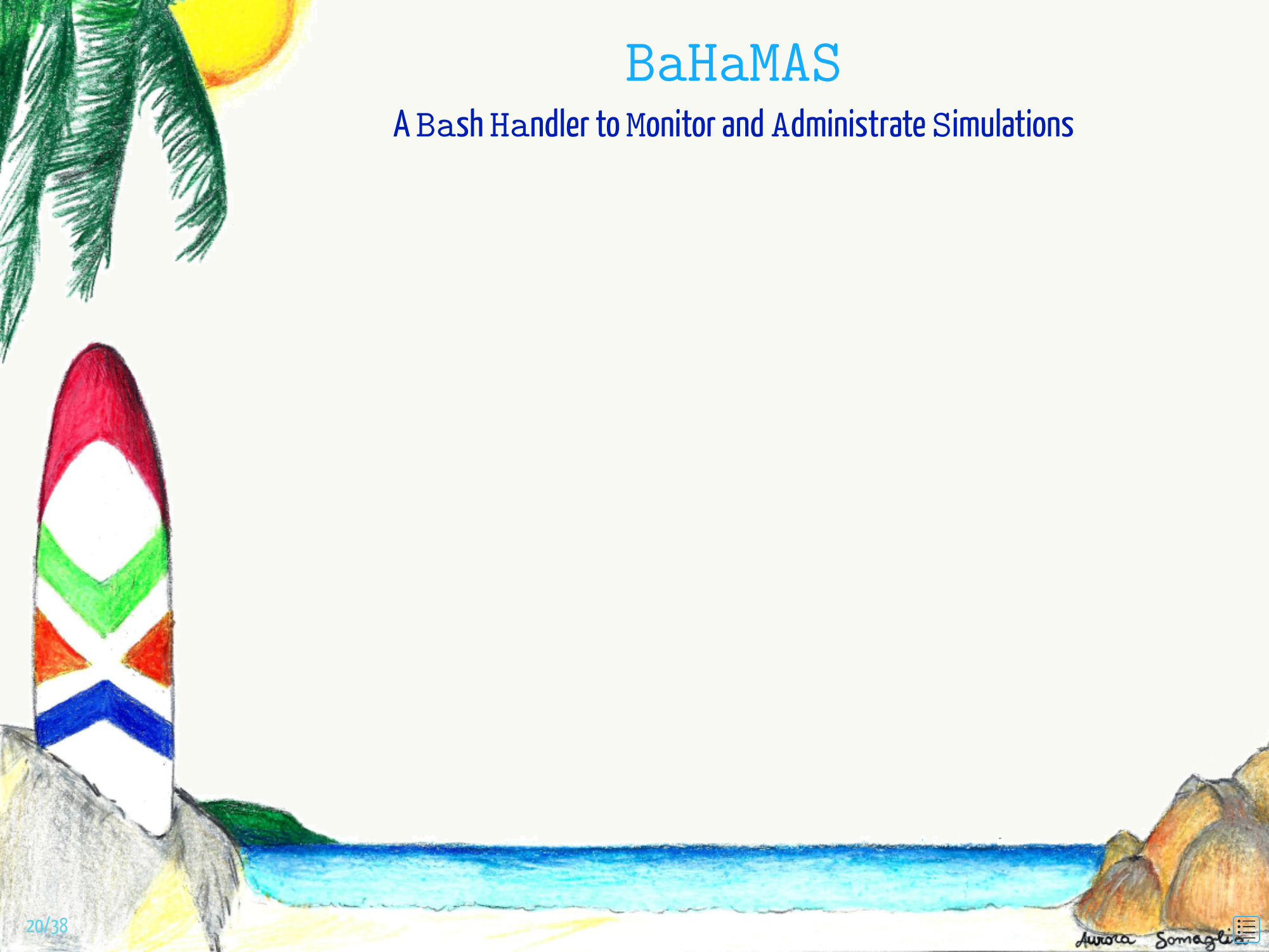
1. `BaHaMAS`
2. GitHooks
3. BashLogger
4. SMASH-vHLLE-hybrid
5. Coffee machine handler

Medium to large Bash projects I worked on

1. BaHaMAS
2. GitHooks
3. BashLogger
4. SMASH-vHLLE-hybrid
5. Coffee machine handler

BaHaMAS

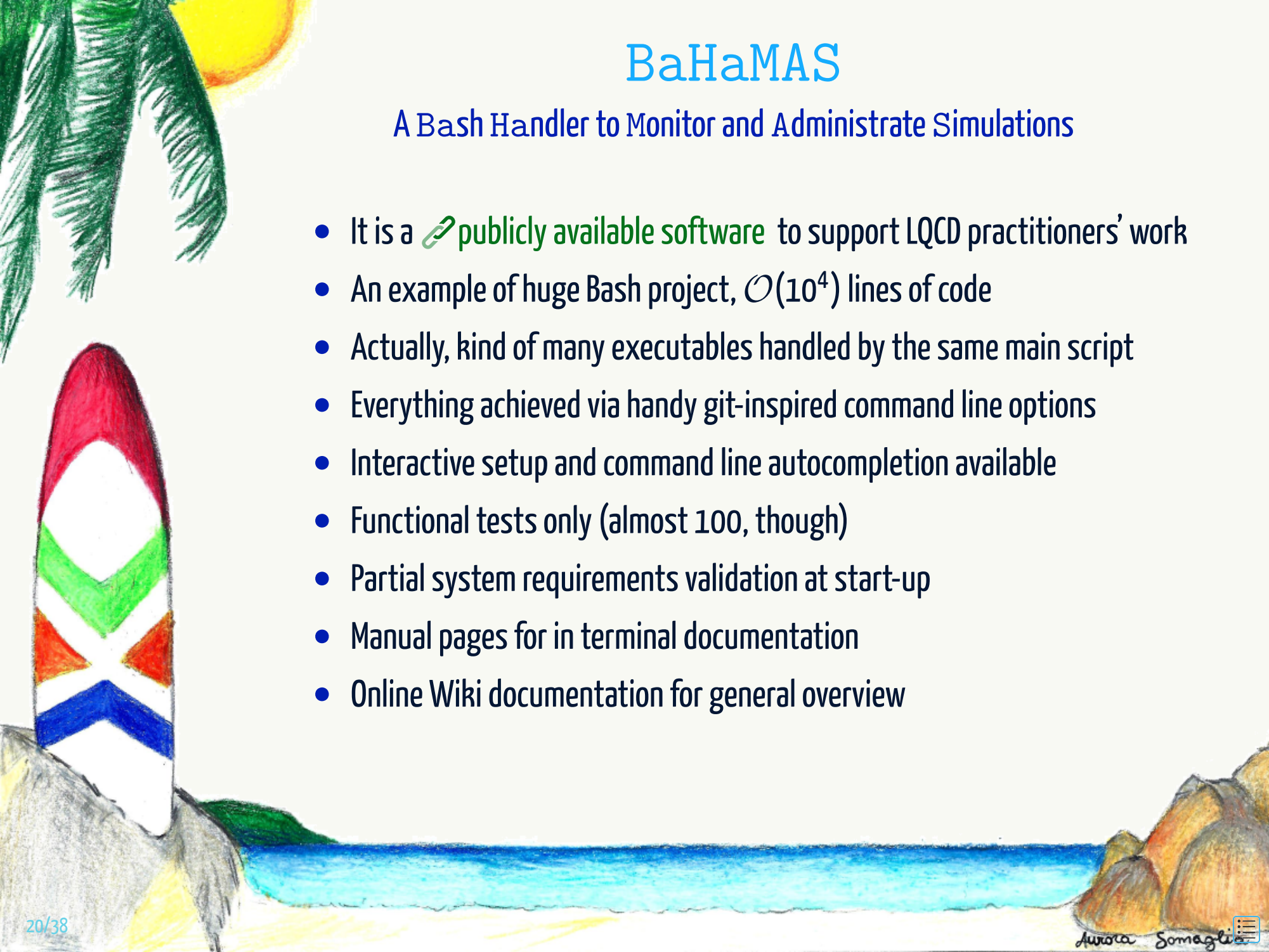
A Bash Handler to Monitor and Administrate Simulations



BaHaMAS

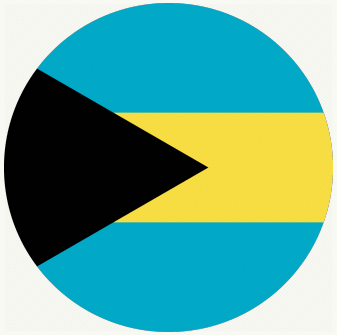
A Bash Handler to Monitor and Administrate Simulations

- It is a publicly available software to support LQCD practitioners' work
- An example of huge Bash project, $\mathcal{O}(10^4)$ lines of code
- Actually, kind of many executables handled by the same main script
- Everything achieved via handy git-inspired command line options
- Interactive setup and command line autocompletion available
- Functional tests only (almost 100, though)
- Partial system requirements validation at start-up
- Manual pages for in terminal documentation
- Online Wiki documentation for general overview



BaHaMAS

A Bash Handler to Monitor and Administrate Simulations



- You can freely run on beaches
- You have a lot of free time
- Cocktails help in relaxing

- You can effectively run on clusters
- You get a lot of free time
- Cocktails may help in developing

...after all, BaHaMAS is a good name...

Aurora Somaglia

A more generic project: GitHooks

What is it about?

This small collection of Bash script is an attempt to have a git-repository-independent hooks setup mechanism to comfortably use over and over again the own implemented hooks. The basic idea is to have a general hooks implementation and a Bash main script to set up them in any git repository with command line options to fine tune the setup. All provided hooks and setup have a highly informative output.

- Giving it a try should be straightforward
- It helps you keeping history of your repositories tidy {via the `commit-msg` hook}
- It helps you keeping the code in your repositories tidy, optionally {via the `pre-commit` hook}
  - checking C++ code style using clang-format
  - checking copyright statement
  - checking license notice

A wide-audience support script: A Bash Logger

- Extremely simple to use and/or to include in your project
- It allows to unify the style of all your output messages
- Errors are redirected to the standard error as it should be
- Messages are classified in levels which can be easily switched on or off at run-time
- Messages are printed to a given file descriptor (customizable at source time)
  ↳ Logger still usable in functions whose "return value" is printed to standard output and meant to be captured by command substitution
- Fatal errors exit with a given exit code (customizable at source time)
  ↳ Customization also possible on single usage, e.g.

```
exit_code=42 Print_Fatal_And_Exit 'Houston, we have a problem!'
```

- Super cool automatic highlighting mechanism

For a detailed description of all functionality refer to the README file of the project.

A wide-audience support script: A Bash Logger

- Extremely simple to use and/or to include in your project
- It allows to unify the style of all your output messages
- Errors are redirected to the standard error as it should be
- Messages are classified in levels which can be easily switched on or off at run-time
- Messages are printed to a given file descriptor (customizable at source time)
  ↳ Logger still usable in functions whose "return value" is printed to standard output and meant to be captured by command substitution
- Fatal errors exit with a given exit code (customizable at source time)
  ↳ Customization also possible on single usage, e.g.

```
exit_code=42 Print_Fatal_And_Exit 'Houston, we have a problem!'
```

- Super cool automatic highlighting mechanism

```
$ git clone https://github.com/AxelKrypton/BashLogger
# ...
$ emacs -nw TestLogger.bash
```


For a detailed description of all functionality refer to the README file of the project.

A wide-audience support script: A Bash Logger

```
#!/usr/bin/env bash
#
#  Copyright (c) 2019,2023-2024
#    Alessandro Sciarra <sciarra@itp.uni-frankfurt.de>
#
#  [...]

trap 'printf "\n"' EXIT
source Logger.bash "$@" || exit 1

Print_Trace                 '\nTrace message '          --emph 'highlighted' ' VS normal'
Print_Debug                 '\nDebug message '          --emph 'highlighted' ' VS normal'
Print_Info                  '\nInformation message '    --emph 'highlighted' ' VS normal'
Print_Attention             '\n' 'Attention message '   --emph 'highlighted' ' VS normal'
Print_Warning               '\n' 'Warning message '     --emph 'highlighted' ' VS normal'
Print_Error                 '\n' 'Error '               --emph 'highlighted' ' VS normal'
(
    Print_Fatal_And_Exit    '\n' 'Fatal error exit! '   --emph 'highlighted' ' VS normal'
    Print_Internal_And_Exit '\n' 'Developer error '     --emph 'highlighted' ' VS normal'
)

if [[ "${TEST}" = '' ]]; then exit 0; fi

# [...] Test with 'set -o pipefail -o nounset -o errexit'
#             and  shopt -s extglob inherit_errexit
```


For a detailed description of all functionality refer to the README file of the project.

A wide-audience support script: A Bash Logger

```
[MacBook-Pro 20.02.2025 18:17:04 BashLogger - develop +1 $ VERBOSE=TRACE ./TestLogger.bash

    TRACE: Trace message highlighted VS normal

    DEBUG: Debug message highlighted VS normal

     INFO: Information message highlighted VS normal

ATTENTION: Attention message highlighted VS normal

  WARNING: Warning message highlighted VS normal

    ERROR: Error highlighted VS normal

    FATAL: Fatal error exit! highlighted VS normal
```


For a detailed description of all functionality refer to the README file of the project.

A wide-audience support script: A Bash Logger

```
[MacBook-Pro 20.02.2025 18:17:04 BashLogger - develop +1 $ VERBOSE=TRACE ./TestLogger.bash
    TRACE: Trace message highlighted VS normal
    DEBUG: Debug message highlighted VS normal
     INFO: Information message highlighted VS normal
ATTENTION: Attention message highlighted VS normal
  WARNING: Warning message highlighted VS normal
    ERROR: Error highlighted VS normal
    FATAL: Fatal error exit! highlighted VS normal
[MacBook-Pro 20.02.2025 18:17:20 BashLogger - develop +1 $ VERBOSE=2 ./TestLogger.bash
  WARNING: Warning message highlighted VS normal
    ERROR: Error highlighted VS normal
    FATAL: Fatal error exit! highlighted VS normal
```


For a detailed description of all functionality refer to the README file of the project.

A wide-audience support script: A Bash Logger

```
[MacBook-Pro 20.02.2025 18:17:04 BashLogger - develop +1 $ VERBOSE=TRACE ./TestLogger.bash
    TRACE: Trace message highlighted VS normal
    DEBUG: Debug message highlighted VS normal
     INFO: Information message highlighted VS normal
ATTENTION: Attention message highlighted VS normal
  WARNING: Warning message highlighted VS normal
    ERROR: Error highlighted VS normal
    FATAL: Fatal error exit! highlighted VS normal
[MacBook-Pro 20.02.2025 18:17:20 BashLogger - develop +1 $ VERBOSE=2 ./TestLogger.bash
  WARNING: Warning message highlighted VS normal
    ERROR: Error highlighted VS normal
    FATAL: Fatal error exit! highlighted VS normal
[MacBook-Pro 20.02.2025 18:20:52 BashLogger - develop +1 $ VERBOSE=1 ./TestLogger.bash
    ERROR: Error highlighted VS normal
    FATAL: Fatal error exit! highlighted VS normal
[MacBook-Pro 20.02.2025 18:20:55 BashLogger - develop +1 $ VERBOSE=0 ./TestLogger.bash
    FATAL: Fatal error exit! highlighted VS normal
```


For a detailed description of all functionality refer to the README file of the project.

Another remarkable handler: SMASH-vHLLE-Hybrid

SMASH-vHLLE-Hybrid

Event-by-event hybrid model for the description of relativistic heavy-ion collisions in the low and high baryon-density regime. This model constitutes a chain of different submodules to appropriately describe each phase of the collision with its corresponding degrees of freedom. It consists of the following modules:

- **SMASH** hadronic transport approach to provide the initial conditions
- **vHLLE** 3+1D viscous hydrodynamics approach to describe the evolution of the hot and dense fireball
  - ↳ CORNELIUS tool to construct a hypersurface of constant energy density from the hydrodynamical evolution (embedded in vHLLE)
- **Sampler** to perform Cooper-Frye particlization of the elements on the freezeout hypersurface
- **SMASH** hadronic transport approach to perform the afterburner evolution

Give credit appropriately

If you are using the SMASH-vHLLE-hybrid, please cite Eur.Phys.J.A 58(2022)11,230. You may also consult this reference for further details about the hybrid approach.

- Another example of large Bash project, $\mathcal{O}(10^4)$ lines of code
- Own test framework with unit and functional tests
- Appealing documentation of the project

Summary of the course

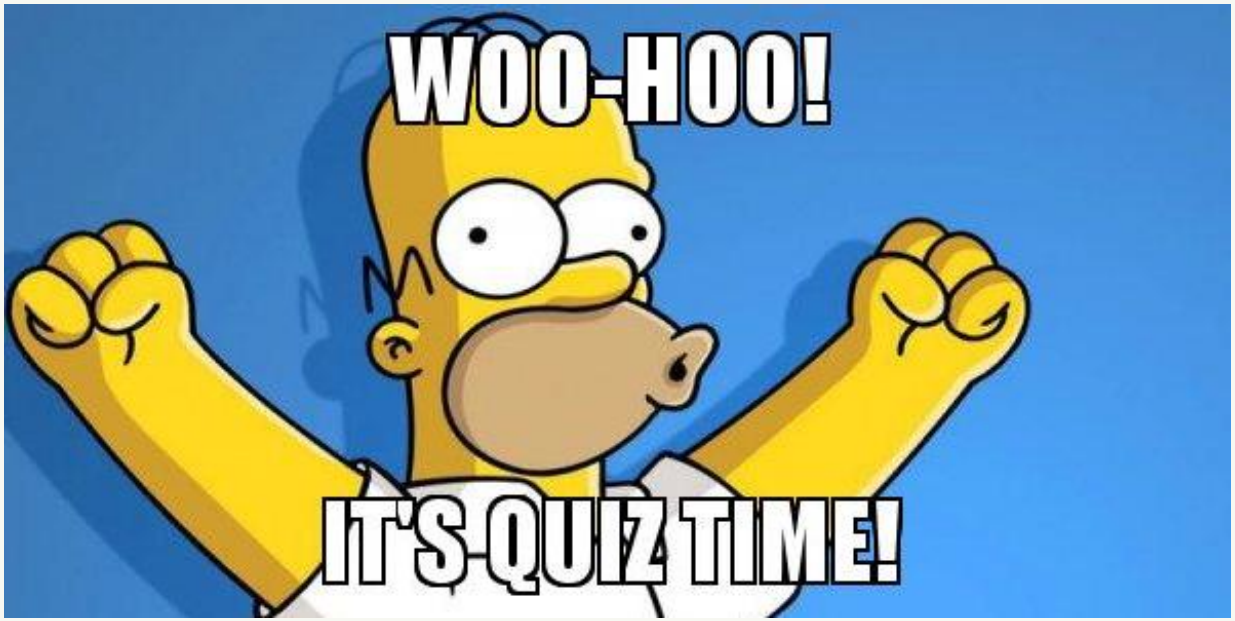
The US Navy DC plane on the black beach at Sólheimasandur

Before we say each other goodbye

- I would like to make an interactive review of the material.
- Do not hesitate to ask!
- It is an opportunity to refresh your mind and clarify your doubts.

Before we say each other goodbye

- I would like to make an interactive review of the material.
- Do not hesitate to ask!
- It is an opportunity to refresh your mind and clarify your doubts.



Quiz time (I) – Day 1

```
if [[ -f 'File with spaces.dat' ]]; then
    rm File with spaces.dat
fi
```

1. Is the above snippet correct? Why?
   - Yes, if the file exists, it gets removed.
   - No, one file should be removed, but three different ones are potentially deleted!
   - Yes, if the file does not exist, nothing is removed.

Quiz time (I) – Day 1

```
if [[ -f 'File with spaces.dat' ]]; then
    rm File with spaces.dat
fi
```

1. Is the above snippet correct? Why?

✗ Yes, if the file exists, it gets removed.

✓ No, one file should be removed, but three different ones are potentially deleted!

✗ Yes, if the file does not exist, nothing is removed.

Quiz time (I) – Day 1

```
if [[ -f 'File with spaces.dat' ]]; then
    rm File with spaces.dat
fi
```

1 Is the above snippet correct? Why?

✗ Yes, if the file exists, it gets removed.

✓ No, one file should be removed, but three different ones are potentially deleted!

✗ Yes, if the file does not exist, nothing is removed.

```
function echo() {
    # A clever echo implementation
    # ...
}
```

2 Sourcing the snippet above in a terminal, what will it happen?

- An error will occur, no function can take the name of a builtin.
- Nothing, the echo builtin will continue to be used by the shell.
- Functions have higher priority than builtins. The `echo` builtin will be shadowed.

Quiz time (I) – Day 1

```
if [[ -f 'File with spaces.dat' ]]; then
    rm File with spaces.dat
fi
```

1 Is the above snippet correct? Why?

- ✗ Yes, if the file exists, it gets removed.
- ✓ No, one file should be removed, but three different ones are potentially deleted!
- ✗ Yes, if the file does not exist, nothing is removed.

```
function echo() {
    # A clever echo implementation
    # ...
}
```

2 Sourcing the snippet above in a terminal, what will it happen?

- ✗ An error will occur, no function can take the name of a builtin.
- ✗ Nothing, the echo builtin will continue to be used by the shell.
- ✓ Functions have higher priority than builtins. The `echo` builtin will be shadowed.

Quiz time (II) – Day 1

```
$ IFS='_'; echo ${IFS}; unset -v 'IFS'
```

3 What does the above command print?

- Nothing, the internal field separator cannot be printed.
- Nothing, the shell performs word splitting after the variable is expanded.
- Since the `IFS` is set to a visible character, an underscore is here printed.

Quiz time (II) – Day 1

```
$ IFS='_'; echo ${IFS}; unset -v 'IFS'
```

3 What does the above command print?

- ✗ Nothing, the internal field separator cannot be printed.
- ✓ Nothing, the shell performs word splitting after the variable is expanded.
- ✗ Since the `IFS` is set to a visible character, an underscore is here printed.

Quiz time (II) – Day 1

```
$ IFS='_'; echo ${IFS}; unset -v 'IFS'
```

3 What does the above command print?

- ✗ Nothing, the internal field separator cannot be printed.
- ✓ Nothing, the shell performs word splitting after the variable is expanded.
- ✗ Since the `IFS` is set to a visible character, an underscore is here printed.

```
function Join() {
    IFS='_'; echo $*
}
```

4 What does the above function does? Is it good code?

- The function prints an underscore-separated list of its arguments. To change the `IFS` globally should be avoided, though.
- This is a clever way to concatenate strings with a symbol as separator. Good code.
- The function prints its arguments and on top it changes the `IFS` globally. Wrong code!

Quiz time (II) – Day 1

```
$ IFS='_'; echo ${IFS}; unset -v 'IFS'
```

3 What does the above command print?

- ✗ Nothing, the internal field separator cannot be printed.
- ✓ Nothing, the shell performs word splitting after the variable is expanded.
- ✗ Since the `IFS` is set to a visible character, an underscore is here printed.

```
function Join() {
    IFS='_'; echo $*
}
```

4 What does the above function does? Is it good code?

- ✗ The function prints an underscore-separated list of its arguments. To change the `IFS` globally should be avoided, though.
- ✗ This is a clever way to concatenate strings with a symbol as separator. Good code.
- ✓ The function prints its arguments and on top it changes the `IFS` globally. Wrong code!

Quiz time (III) – Day 1

```
$ echo {1..$(date +'%d')}  # %d is the day of the month
```

5 What does the above command print today?

- It prints the numbers from 1 to the present day of the month.
- `{1..29}` gets printed since brace expansion happens before command substitution.
- An error is printed, command substitution cannot be used in brace expansion.

Quiz time (III) – Day 1

```
$ echo {1..$(date +'%d')}  # %d is the day of the month
```

5 What does the above command print today?

- ✗ It prints the numbers from 1 to the present day of the month.
- ✓ `{1..29}` gets printed since brace expansion happens before command substitution.
- ✗ An error is printed, command substitution cannot be used in brace expansion.

Quiz time (III) – Day 1

```
$ echo {1..$(date +'%d')}  # %d is the day of the month
```

5 What does the above command print today?

✗ It prints the numbers from 1 to the present day of the month.

✓ `{1..29}` gets printed since brace expansion happens before command substitution.

✗ An error is printed, command substitution cannot be used in brace expansion.

```
echo 3.2*4 | bc -l
```

6 Does the above code do the required multiplication?

- Yes. Bash does not support floating point operations and `bc` can be used in this case.
- Yes, the code above always works. However, a here-string construct would be better to avoid to use `echo` and the pipeline.
- This code is wrong!

Quiz time (III) – Day 1

```
$ echo {1..$(date +'%d')}  # %d is the day of the month
```

5 What does the above command print today?

- ✗ It prints the numbers from 1 to the present day of the month.
- ✓ `{1..29}` gets printed since brace expansion happens before command substitution.
- ✗ An error is printed, command substitution cannot be used in brace expansion.

```
echo 3.2*4 | bc -l
```

6 Does the above code do the required multiplication?

- ✗ Yes. Bash does not support floating point operations and `bc` can be used in this case.
- ✗ Yes, the code above always works. However, a here-string construct would be better to avoid to use `echo` and the pipeline.
- ✓ This code is wrong! Shell automatically does filename extension. Often it might work, but only if the filename extension fails and `nullglob` is not set!

Quiz time (IV) – Day 2

```
if [ ! $1 =~ ^([0-9]+-)?[0-9]{1,2}:[0-9]{2}:[0-9]{2}$ ]; then
    echo "Error: Walltime specified in a wrong format!"
fi
```

7 Is there anything wrong in the above snippet of code?

- No, a regular expression test is done to check the format of `$1`.
- The `[` command does not support the `=~` comparison. With `[[` keyword the code works.
- Even with `[[`, the code is wrong, because the regex pattern must be quoted!

Quiz time (IV) – Day 2

```
if [ ! $1 =~ ^([0-9]+-)?[0-9]{1,2}:[0-9]{2}:[0-9]{2}$ ]; then
    echo "Error: Walltime specified in a wrong format!"
fi
```

7 Is there anything wrong in the above snippet of code?

- ✗ No, a regular expression test is done to check the format of `$1`.
- ✓ The `[` command does not support the `=~` comparison. With `[[` keyword the code works.
- ✗ Even with `[[`, the code is wrong, because the regex pattern must be quoted!

Quiz time (IV) – Day 2

```
if [ ! $1 =~ ^([0-9]+-)?[0-9]{1,2}:[0-9]{2}:[0-9]{2}$ ]; then
    echo "Error: Walltime specified in a wrong format!"
fi
```

7 Is there anything wrong in the above snippet of code?

✗ No, a regular expression test is done to check the format of `$1`.

✓ The `[` command does not support the `=~` comparison. With `[[` keyword the code works.

✗ Even with `[[`, the code is wrong, because the regex pattern must be quoted!

```
for file in ${array[@]}; do
    # do something with file
done
```

8 Does the above for loop always work?

- Yes, using arrays is the best way to iterate over files.
- The snippet above works, but it is better to use globbing directly in the `for` construct.
- Although the code above might works, it is in general wrong!

Quiz time (IV) – Day 2

```
if [ ! $1 =~ ^([0-9]+-)?[0-9]{1,2}:[0-9]{2}:[0-9]{2}$ ]; then
    echo "Error: Walltime specified in a wrong format!"
fi
```

7 Is there anything wrong in the above snippet of code?

- ✗ No, a regular expression test is done to check the format of `$1`.
- ✓ The `[` command does not support the `=~` comparison. With `[[` keyword the code works.
- ✗ Even with `[[`, the code is wrong, because the regex pattern must be quoted!

```
for file in ${array[@]}; do
    # do something with file
done
```

8 Does the above for loop always work?

- ✗ Yes, using arrays is the best way to iterate over files.
- ✗ The snippet above works, but it is better to use globbing directly in the `for` construct.
- ✓ Although the code above might works, it is in general wrong!
  Use `"${array[@]}"` to avoid word splitting!

Quiz time (V) – Day 2

```
for((index = 0; index < ${#array[@]}; index ++)); do
    # Do something with ${array[index]}
done
```

9 What would you say about the for loop above?

- Nothing, it uses the C-like for loop to iterate over an array.
- It is advisable to iterate over the indices via `"${!array[@]}"`. If the array is sparse, the code above will fail.
- The code will fail. Spaces in assignments and in the index increment are wrong in Bash.

Quiz time (V) – Day 2

```
for((index = 0; index < ${#array[@]}; index ++)); do
    # Do something with ${array[index]}
done
```

9 What would you say about the for loop above?

- ✗ Nothing, it uses the C-like for loop to iterate over an array.
- ✓ It is advisable to iterate over the indices via `"${!array[@]}"`. If the array is sparse, the code above will fail.
- ✗ The code will fail. Spaces in assignments and in the index increment are wrong in Bash.

Quiz time (V) – Day 2

```
for((index = 0; index < ${#array[@]}; index ++)); do
    # Do something with ${array[index]}
done
```

9 What would you say about the for loop above?

- ✗ Nothing, it uses the C-like for loop to iterate over an array.
- ✓ It is advisable to iterate over the indices via `"${!array[@]}"`. If the array is sparse, the code above will fail.
- ✗ The code will fail. Spaces in assignments and in the index increment are wrong in Bash.

```
$ index=1; array=(one two three)
$ unset -v 'array[${index}]'; printf '%s\n' "${array[@]}"
```

10 Is the above usage of `unset` correct?

- Yes, it is. The `index` variable is accessed in arithmetic context when the array element is being unset.
- No, it isn't. Single quotes prevent parameter expansion and `unset` will then fail.
- No, it isn't. No quotes should be used around `array[${index}]`.

Quiz time (V) – Day 2

```
for((index = 0; index < ${#array[@]}; index ++)); do
    # Do something with ${array[index]}
done
```

9 What would you say about the for loop above?

- ✗ Nothing, it uses the C-like for loop to iterate over an array.
- ✓ It is advisable to iterate over the indices via `"${!array[@]}"`. If the array is sparse, the code above will fail.
- ✗ The code will fail. Spaces in assignments and in the index increment are wrong in Bash.

```
$ index=1; array=(one two three)
$ unset -v 'array[${index}]'; printf '%s\n' "${array[@]}"
```

10 Is the above usage of `unset` correct?

- ✓ Yes, it is. The `index` variable is accessed in arithmetic context when the array element is being unset.
- ✗ No, it isn't. Single quotes prevent parameter expansion and `unset` will then fail.
- ✗ No, it isn't. No quotes should be used around `array[${index}]`.

Quiz time (VI) – Day 2

11 Which of the following statements is correct?

- A Bash script can change the environment of the executing shell when executed.
- Using `export` builtin, environment variables of the parent process can be changed.
- Sourcing a script is a way to affect the environment where it is sourced.
- It is possible to specify a temporary environment change which only takes effect for the duration of a command.

Quiz time (VI) – Day 2

11 Which of the following statements is correct?

- ✗ A Bash script can change the environment of the executing shell when executed.
- ✗ Using `export` builtin, environment variables of the parent process can be changed.
- ✓ Sourcing a script is a way to affect the environment where it is sourced.
- ✓ It is possible to specify a temporary environment change which only takes effect for the duration of a command.

Quiz time (VI) – Day 2

11 Which of the following statements is correct?

- ✗ A Bash script can change the environment of the executing shell when executed.
- ✗ Using `export` builtin, environment variables of the parent process can be changed.
- ✓ Sourcing a script is a way to affect the environment where it is sourced.
- ✓ It is possible to specify a temporary environment change which only takes effect for the duration of a command.

```
for file in *; do
    # Do something with files
    stat "${file}" | head -n2
done
```

12 Do you have any comment about the for-loop above?

- Yes, the code above does not work. Filenames should be stored in an array first.
- No. Using globbing is the right way to iterate over files in the present folder.
- The code might do something unexpected!

Quiz time (VI) – Day 2

11 Which of the following statements is correct?

- ✗ A Bash script can change the environment of the executing shell when executed.
- ✗ Using `export` builtin, environment variables of the parent process can be changed.
- ✓ Sourcing a script is a way to affect the environment where it is sourced.
- ✓ It is possible to specify a temporary environment change which only takes effect for the duration of a command.

```
for file in *; do
    # Do something with files
    stat "${file}" | head -n2
done
```

12 Do you have any comment about the for-loop above?

- ✗ Yes, the code above does not work. Filenames should be stored in an array first.
- ✗ No. Using globbing is the right way to iterate over files in the present folder.
- ✓ The code might do something unexpected! Be sure to handle the no-file-scenario correctly (or set `nullglob`)!

Quiz time (VII) – Day 3

```
# Keep only first three lines of a file
$ head -n3 filename > filename
```

13 Does the code above work?

- Yes, provided that the file `filename` exists.
- No, the `head` command must run in a subshell to access the file content before it gets overwritten.
- No, because redirection is the first thing that Bash does. The content of the file is lost before running `head`!

Quiz time (VII) – Day 3

```
# Keep only first three lines of a file
$ head -n3 filename > filename
```

13 Does the code above work?

- ✗ Yes, provided that the file `filename` exists.
- ✗ No, the `head` command must run in a subshell to access the file content before it gets overwritten.
- ✓ No, because redirection is the first thing that Bash does. The content of the file is lost before running `head`!

Quiz time (VII) – Day 3

```
# Keep only first three lines of a file
$ head -n3 filename > filename
```

13 Does the code above work?

- ✗ Yes, provided that the file `filename` exists.
- ✗ No, the `head` command must run in a subshell to access the file content before it gets overwritten.
- ✓ No, because redirection is the first thing that Bash does. The content of the file is lost before running `head`!

```
$ wc -c <<< Hello
```

14 Which number is printed by the above command?

- 6. In the here-string construct Bash appends an endline to the string.
- 5. Although an endline is appended to the string before passing it on to `wc` standard input, this is not counted.
- The command above gives an error since the string is not quoted.

Quiz time (VII) – Day 3

```
# Keep only first three lines of a file
$ head -n3 filename > filename
```

13 Does the code above work?

- ✗ Yes, provided that the file `filename` exists.
- ✗ No, the `head` command must run in a subshell to access the file content before it gets overwritten.
- ✓ No, because redirection is the first thing that Bash does. The content of the file is lost before running `head`!

```
$ wc -c <<< Hello
```

14 Which number is printed by the above command?

- ✓ 6. In the here-string construct Bash appends an endline to the string.
- ✗ 5. Although an endline is appended to the string before passing it on to `wc` standard input, this is not counted.
- ✗ The command above gives an error since the string is not quoted.

Quiz time (VIII) – Day 3

```
$ counter=0
$ grep -o '[aeiou]' <<< "Hello world" | wc -l | read
    counter; echo "${counter}"
```

15 What does the command above print?

- 3. The vowels in the string are counted and the number stored in `counter`.
- 0. The variable `counter` is set in a subshell and hence its change is not seen after the pipeline.
- The command above gives an error, since `read` does not read from the standard input.

Quiz time (VIII) – Day 3

```
$ counter=0
$ grep -o '[aeiou]' <<< "Hello world" | wc -l | read
    counter; echo "${counter}"
```

15 What does the command above print?

- ✗ 3. The vowels in the string are counted and the number stored in `counter`.
- ✓ 0. The variable `counter` is set in a subshell and hence its change is not seen after the pipeline. Use command substitution to initialise `counter` and avoid `read`.
- ✗ The command above gives an error, since `read` does not read from the standard input.

Quiz time (VIII) – Day 3

```
$ counter=0
$ grep -o '[aeiou]' <<< "Hello world" | wc -l | read
    counter; echo "${counter}"
```

15 What does the command above print?

- ✗ 3. The vowels in the string are counted and the number stored in `counter`.
- ✓ 0. The variable `counter` is set in a subshell and hence its change is not seen after the pipeline. Use command substitution to initialise `counter` and avoid `read`.
- ✗ The command above gives an error, since `read` does not read from the standard input.

16 Which of the following statements about functions is correct?

- The `return` builtin is meant to return values from functions.
- Variable in functions are by default global. Use the `local` builtin to declare local ones.
- Syntactical errors in unused functions still make the script fail.
- In order to preserve the OS, Bash limits by default the number of recursive calls.
- Arguments to a function should preferably be passed by value.

Quiz time (VIII) – Day 3

```
$ counter=0
$ grep -o '[aeiou]' <<< "Hello world" | wc -l | read
    counter; echo "${counter}"
```

15 What does the command above print?

- ✗ 3. The vowels in the string are counted and the number stored in `counter`.
- ✓ 0. The variable `counter` is set in a subshell and hence its change is not seen after the pipeline. Use command substitution to initialise `counter` and avoid `read`.
- ✗ The command above gives an error, since `read` does not read from the standard input.

16 Which of the following statements about functions is correct?

- ✗ The `return` builtin is meant to return values from functions.
- ✓ Variable in functions are by default global. Use the `local` builtin to declare local ones.
- ✗ Syntactical errors in unused functions still make the script fail.
- ✗ In order to preserve the OS, Bash limits by default the number of recursive calls.
- ✓ Arguments to a function should preferably be passed by value.

Quiz time (IX) – Day 4

```
$ for index in 1 3 5; do sleep ${index} &; done
```

17 What does the command above do?

- It runs three `sleep` commands at the same time in background.
- Using `&;` instead of simply `&`, the shell will queue the `sleep` commands and execute them after each other in background.
- The command above will fail. The `&` is a command terminator as well as `;` is. Hence, Bash will complain bacause an empty command is terminated (between `&` and `;`).

Quiz time (IX) – Day 4

```
$ for index in 1 3 5; do sleep ${index} &; done
```

17 What does the command above do?

- ✗ It runs three `sleep` commands at the same time in background.
- ✗ Using `&` `;` instead of simply `&`, the shell will queue the `sleep` commands and execute them after each other in background.
- ✓ The command above will fail. The `&` is a command terminator as well as `;` is. Hence, Bash will complain bacause an empty command is terminated (between `&` and `;`).

Quiz time (IX) – Day 4

```
$ for index in 1 3 5; do sleep ${index} &; done
```

17 What does the command above do?

- ✗ It runs three `sleep` commands at the same time in background.
- ✗ Using `&;` instead of simply `&`, the shell will queue the `sleep` commands and execute them after each other in background.
- ✓ The command above will fail. The `&` is a command terminator as well as `;` is. Hence, Bash will complain bacause an empty command is terminated (between `&` and `;`).

18 Which of the following statements about `kill` is correct?

- It is used to exclusively terminate processes.
- If no signal is specified, SIGTERM (15) is sent.
- The SIGKILL signal should almost always be avoided.
- The signal specification via name should be preferred, because the name-to-number mapping can vary between UNIX variants.

Quiz time (IX) – Day 4

```
$ for index in 1 3 5; do sleep ${index} &; done
```

17 What does the command above do?

- ✗ It runs three `sleep` commands at the same time in background.
- ✗ Using `&;` instead of simply `&`, the shell will queue the `sleep` commands and execute them after each other in background.
- ✓ The command above will fail. The `&` is a command terminator as well as `;` is. Hence, Bash will complain bacause an empty command is terminated (between `&` and `;`).

18 Which of the following statements about `kill` is correct?

- ✗ It is used to exclusively terminate processes.
- ✓ If no signal is specified, SIGTERM (15) is sent.
- ✓ The SIGKILL signal should almost always be avoided.
- ✓ The signal specification via name should be preferred, because the name-to-number mapping can vary between UNIX variants.

Quiz time (X) – Day 4

```
trap 'CleanAuxiliaryFiles; exit 1' INT
```

19 Would you consider the code above good practice?

- Yes, a clean-up is done if the script gets interrupted.
- No, a clean-up should be done also in other cases, e.g. if an error occur.
- No. Any script that set up a trap on INT is highly encouraged to kill itself in the trap to avoid troubles to the caller.

Quiz time (X) – Day 4

```
trap 'CleanAuxiliaryFiles; exit 1' INT
```

19 Would you consider the code above good practice?

- ✗ Yes, a clean-up is done if the script gets interrupted.
- ✗ No, a clean-up should be done also in other cases, e.g. if an error occur.
- ✓ No. Any script that set up a trap on INT is highly encouraged to kill itself in the trap to avoid troubles to the caller. Trap on EXIT instead.

Quiz time (X) – Day 4

```
trap 'CleanAuxiliaryFiles; exit 1' INT
```

19 Would you consider the code above good practice?

- ✗ Yes, a clean-up is done if the script gets interrupted.
- ✗ No, a clean-up should be done also in other cases, e.g. if an error occur.
- ✓ No. Any script that set up a trap on INT is highly encouraged to kill itself in the trap to avoid troubles to the caller. Trap on EXIT instead.

```
cd temporaryFolder
touch file{01..1000000}.dat # Code to create many files
```

20 What would you say about the code above?

- Indeed, before creating many files it is wise to move into a temporary folder.
- The code is fine, but the commands could be merged as `touch temporaryFolder/file{01..1000000}.dat`.
- Error handling is missing!

Quiz time (X) – Day 4

```
trap 'CleanAuxiliaryFiles; exit 1' INT
```

19 Would you consider the code above good practice?

- ✗ Yes, a clean-up is done if the script gets interrupted.
- ✗ No, a clean-up should be done also in other cases, e.g. if an error occur.
- ✓ No. Any script that set up a trap on INT is highly encouraged to kill itself in the trap to avoid troubles to the caller. Trap on EXIT instead.

```
cd temporaryFolder
touch file{01..1000000}.dat # Code to create many files
```

20 What would you say about the code above?

- ✗ Indeed, before creating many files it is wise to move into a temporary folder.
- ✗ The code is fine, but the commands could be merged as `touch temporaryFolder/file{01..1000000}.dat`.
- ✓ Error handling is missing! If the `cd` fails we mess up the folder where we are!

Quiz time (XI) – Day 5

```
$ permissions=$(ls -l file | awk '{print $1}')
```

21 Does the code above work?

- It always does. It extracts the permissions of `file` storing them in a variable.
- It always does, because a filename contains only letters. The output of `ls` should not be parsed, though.
- In general, it does not!

Quiz time (XI) – Day 5

```
$ permissions=$(ls -l file | awk '{print $1}')
```

21 Does the code above work?

- ✗ It always does. It extracts the permissions of `file` storing them in a variable.
- ✗ It always does, because a filename contains only letters. The output of `ls` should not be parsed, though.
- ✓ In general, it does not! The output format of `ls -l` is not guaranteed consistent across platforms!

Quiz time (XI) – Day 5

```
$ permissions=$(ls -l file | awk '{print $1}')
```

21 Does the code above work?

- ✗ It always does. It extracts the permissions of `file` storing them in a variable.
- ✗ It always does, because a filename contains only letters. The output of `ls` should not be parsed, though.
- ✓ In general, it does not! The output format of `ls -l` is not guaranteed consistent across platforms!

```
$ filename='data.dat'
$ extension="$(awk -F '.' '{print $2}' <<< "${filename}")"
```

22 What would you say about the code above?

- It fails. Bash cannot handle nested double quotes in the second line.
- It works but it is bad practice!
- It works and it is a clever way of using `awk` with a customised field separator.

Quiz time (XI) – Day 5

```
$ permissions=$(ls -l file | awk '{print $1}')
```

21 Does the code above work?

- ✗ It always does. It extracts the permissions of `file` storing them in a variable.
- ✗ It always does, because a filename contains only letters. The output of `ls` should not be parsed, though.
- ✓ In general, it does not! The output format of `ls -l` is not guaranteed consistent across platforms!

```
$ filename='data.dat'
$ extension="$(awk -F '.' '{print $2}' <<< "${filename}")"
```

22 What would you say about the code above?

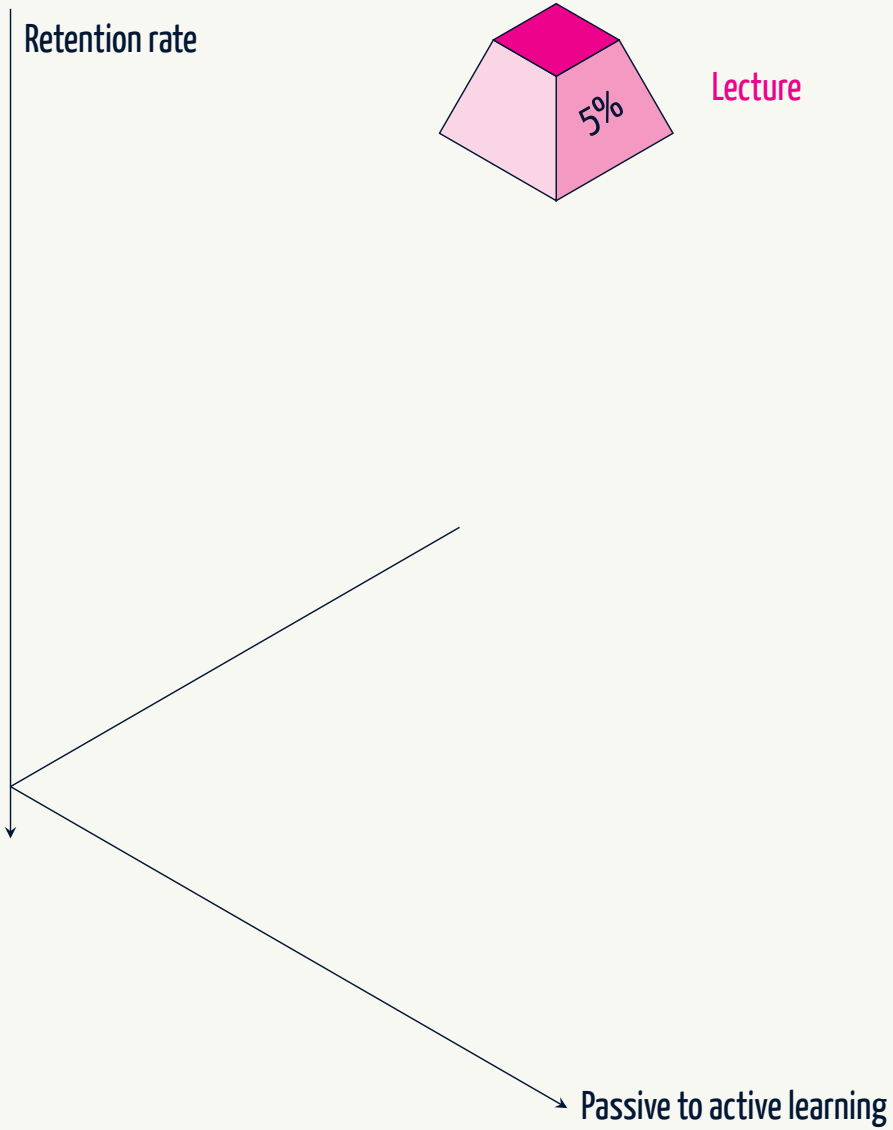
- ✗ It fails. Bash cannot handle nested double quotes in the second line.
- ✓ It works but it is bad practice! Parameter expansion should be used here!
- ✗ It works and it is a clever way of using `awk` with a customised field separator.

What to do, now?

The Þingvellir national park

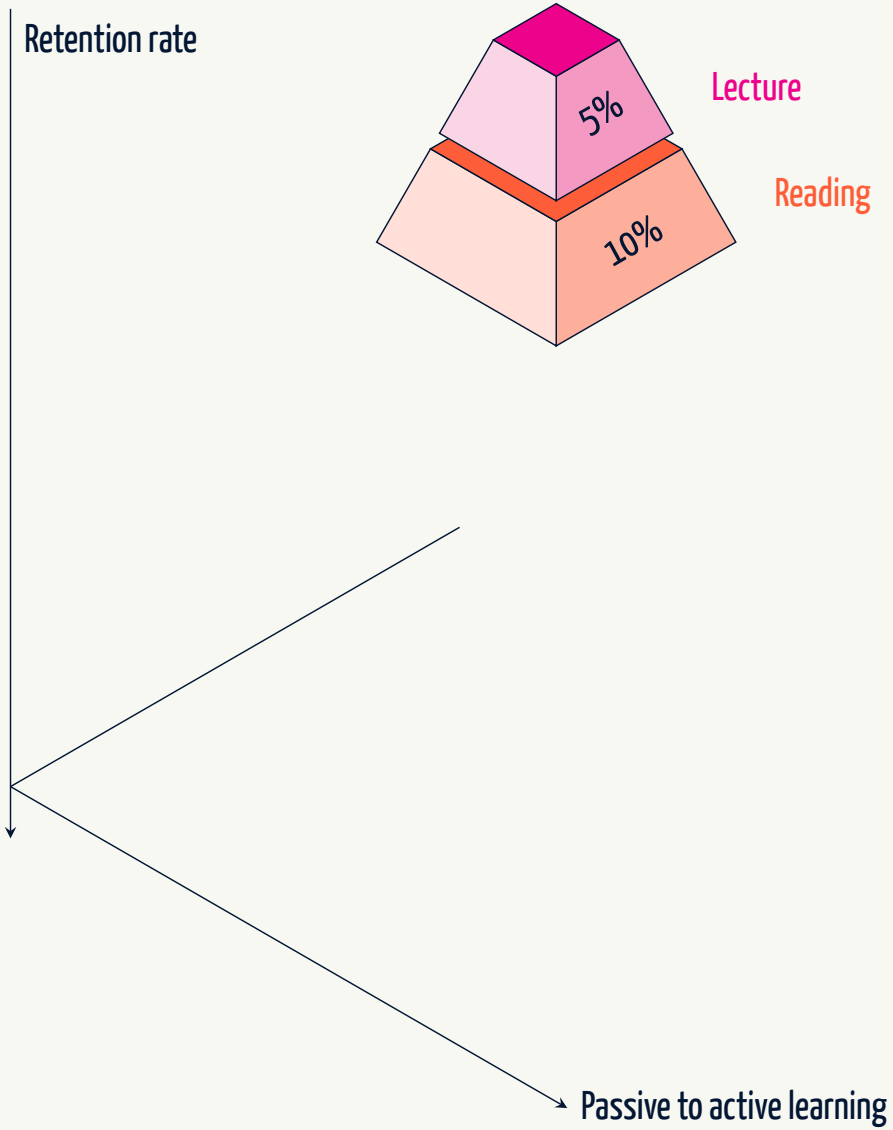
Average students retention rates

Source: The learning Pyramid, researched and created by the National training Laboratories in Betel, Maine.



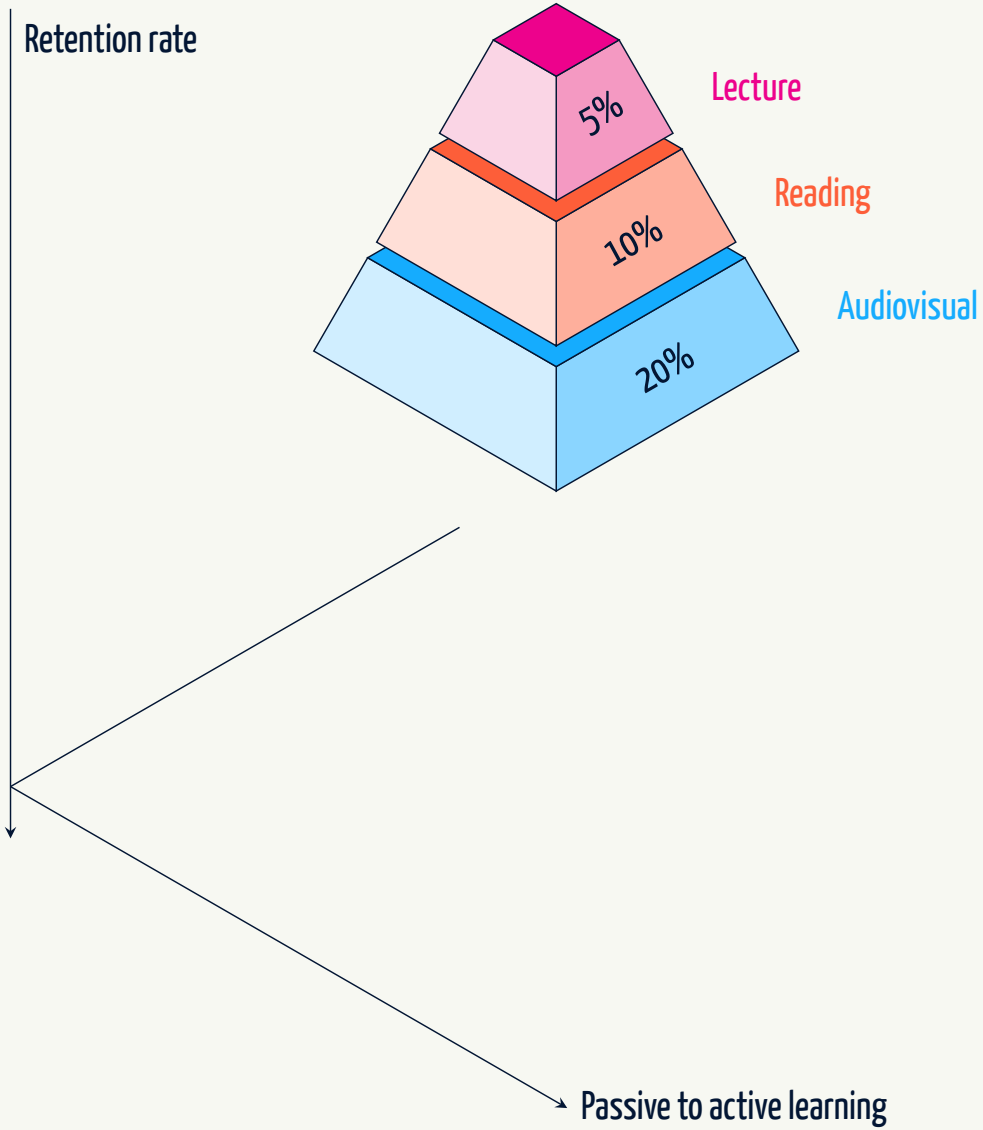
Average students retention rates

Source: The learning Pyramid, researched and created by the National training Laboratories in Betel, Maine.



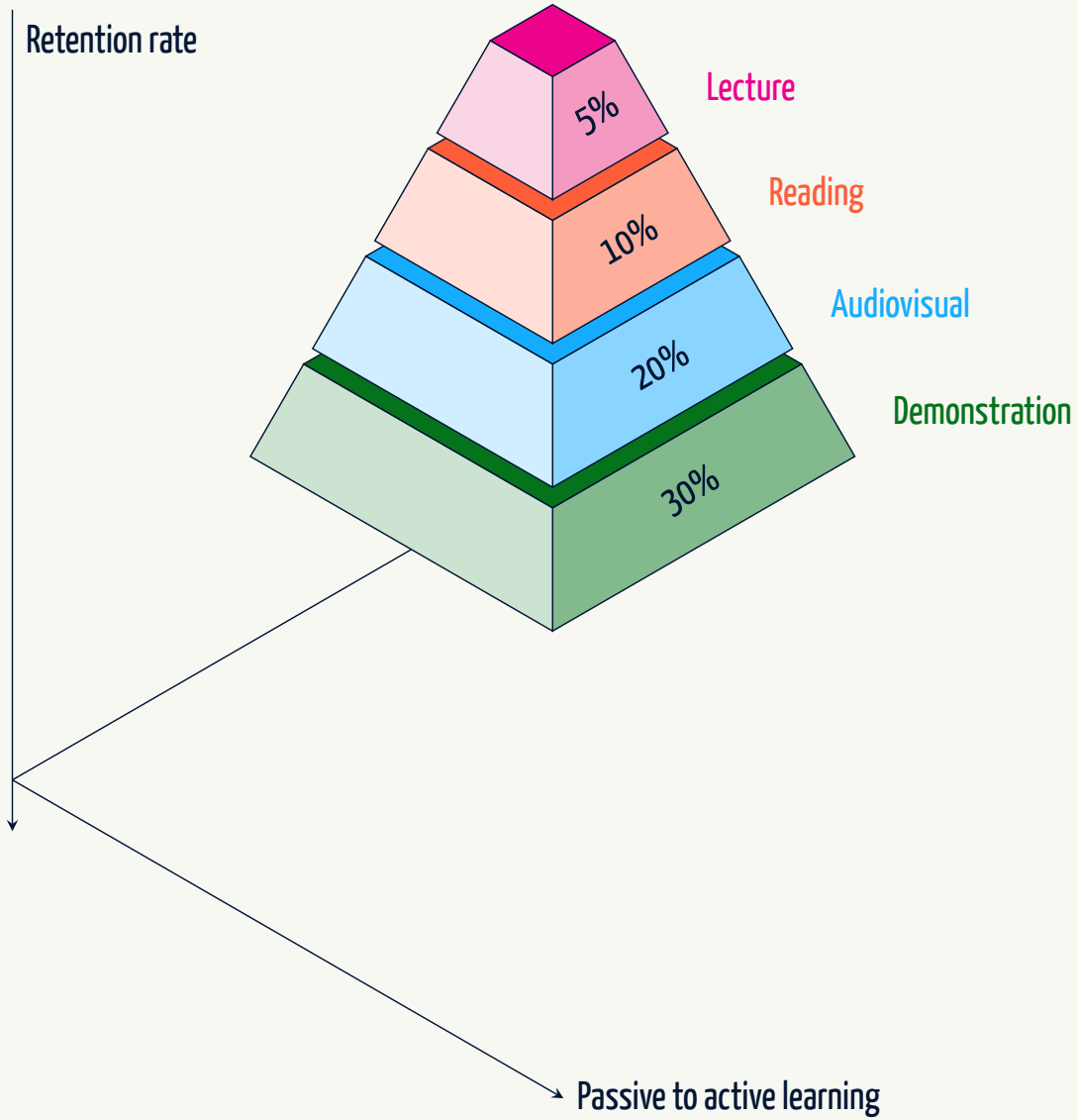
Average students retention rates

Source: The learning Pyramid, researched and created by the National training Laboratories in Betel, Maine.



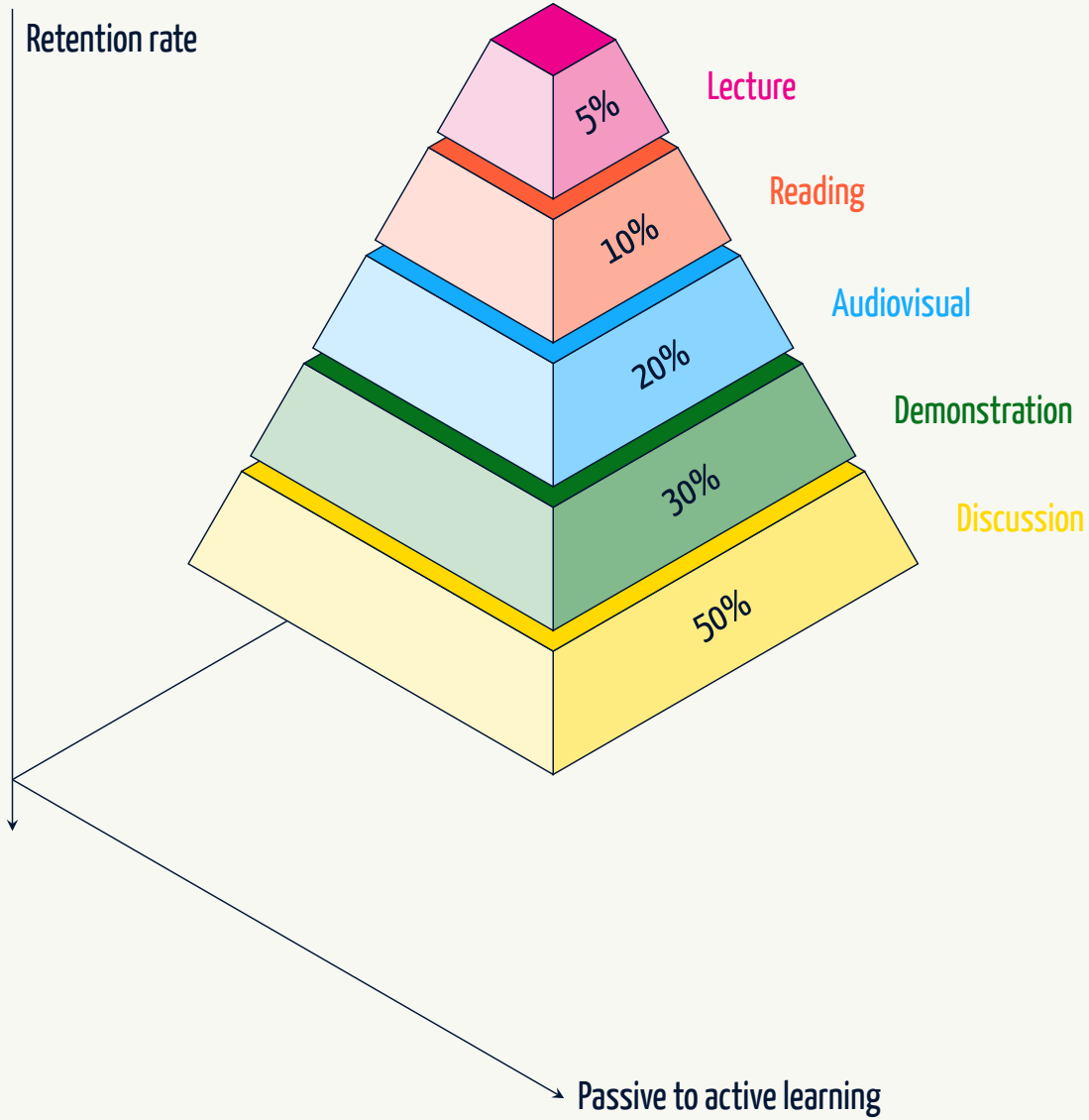
Average students retention rates

Source: The learning Pyramid, researched and created by the National training Laboratories in Betel, Maine.



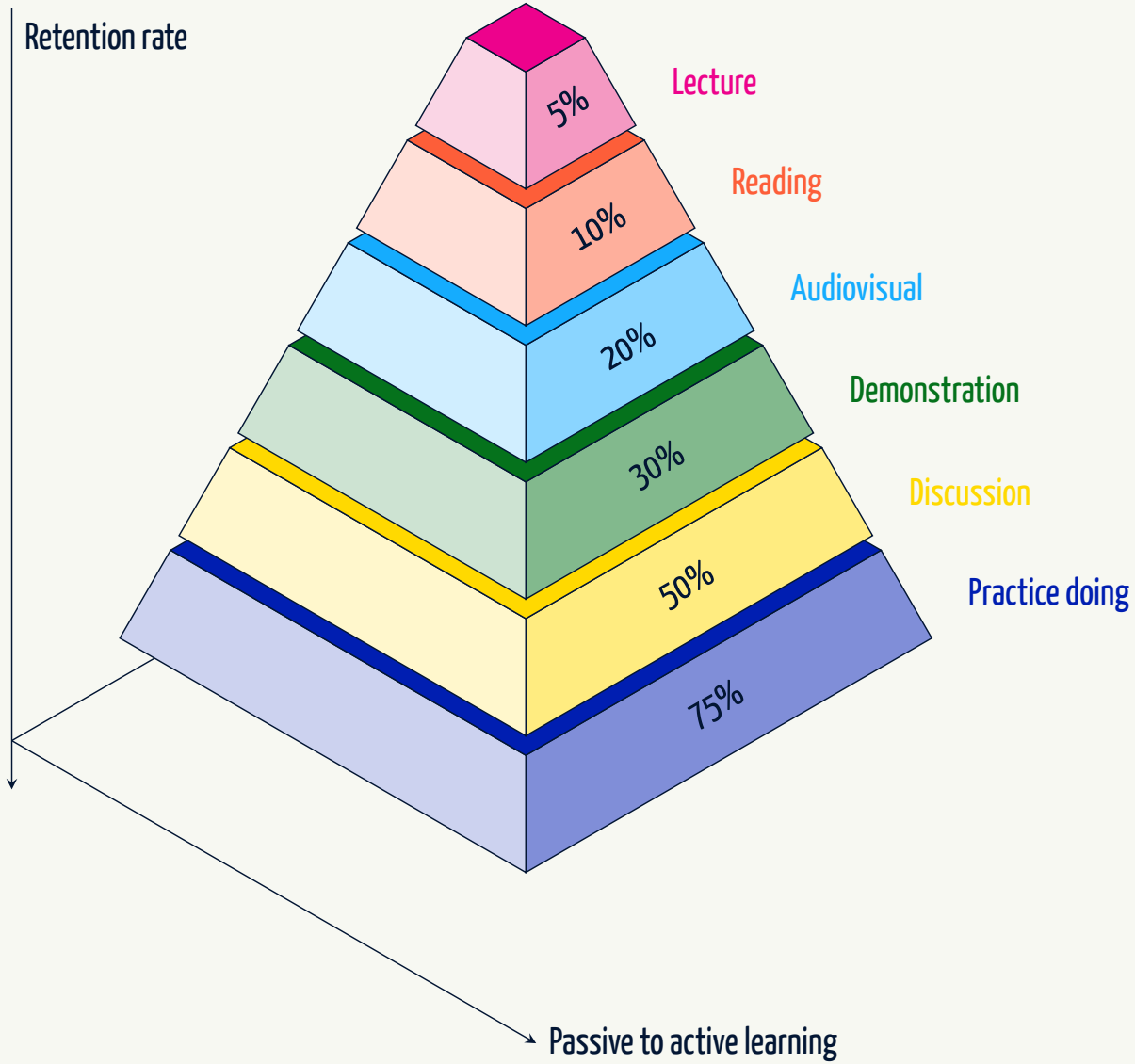
Average students retention rates

Source: The learning Pyramid, researched and created by the National training Laboratories in Betel, Maine.



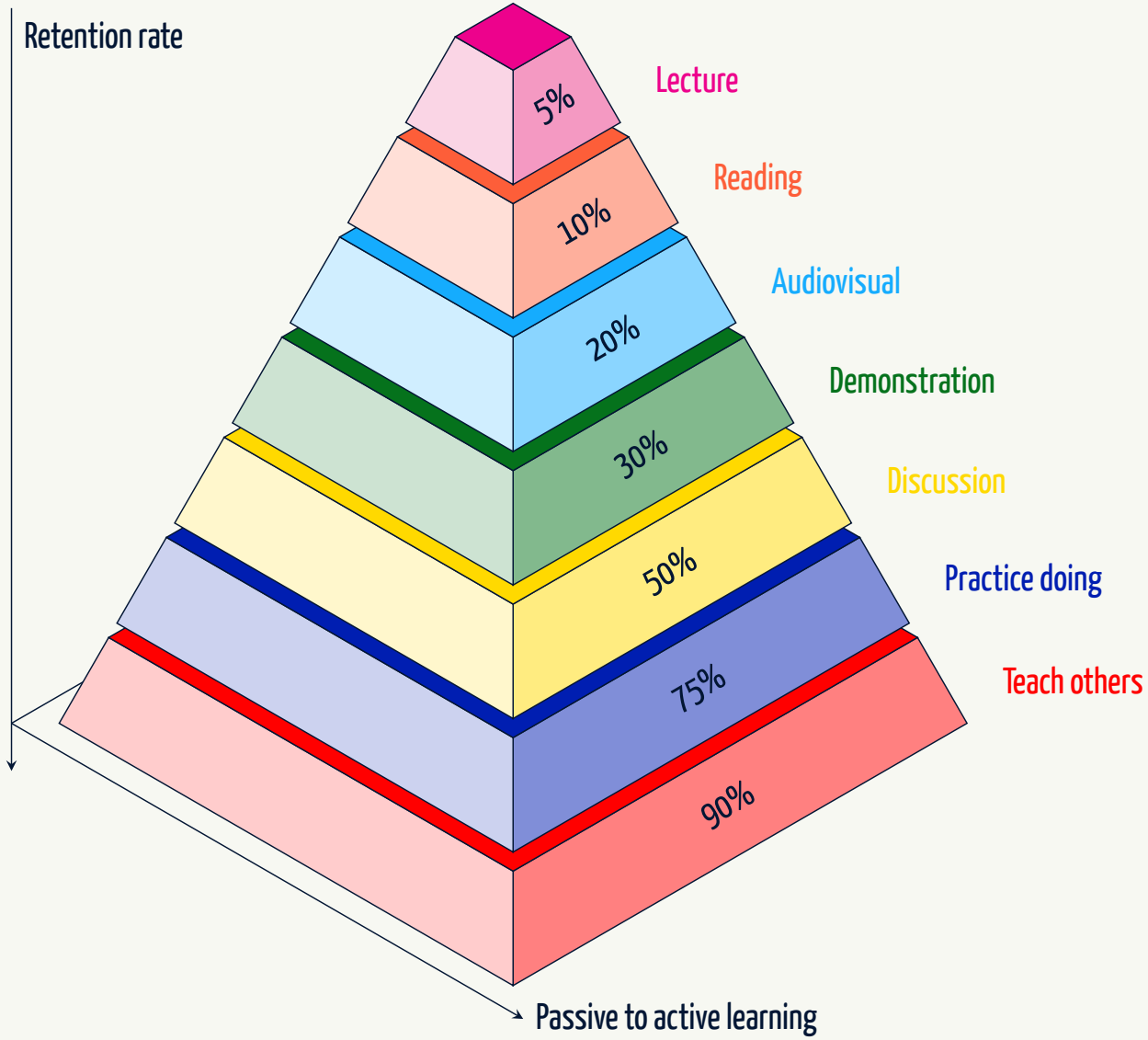
Average students retention rates

Source: The learning Pyramid, researched and created by the National training Laboratories in Betel, Maine.



Average students retention rates

Source: The learning Pyramid, researched and created by the National training Laboratories in Betel, Maine.



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 the material of the course, work on it**
- Links in the slides 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 the material of the course, work on it**
- Links in the slides are important to make a further step forward

It will be probably enough for your next future

The material discussed in these 5 days is enough to be able to say that you know how Bash works!

Well, probably not yet, I would say!

Unfortunately it is not that straightforward!

You need practice to digest most of aspects 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 five days should be a very good and complete 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 five days should be a very good and complete 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 you feel at ease during the lecture?
     - 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 five days should be a very good and complete 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 you feel at ease during the lecture?
     - 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!