Bash it Out!

Strengthen your Bash knowledge with 17 scripting challenges of varied difficulties

By Sylvain Leroux

First Edition, 2017

Published by It's FOSS
https://itsfoss.com

Acknowledgement

This work has started by a humble post in the Linux Users Group on Facebook. Needless to say, I had no idea at the time there will be so many Bash Challenges that we could publish an entire book!

Certainly, I couldn't have gone so far without the help of Abhishek Prakash who invited me to publish these challenges not only on It's FOSS Facebook page but also on the It's FOSS website. Abhishek is also at the origin of the idea of compiling all those challenges—including some unreleased material—to make the book you are reading now.

But there is a whole world between publishing a regular column on a blog and making a book. Illustration, pagination, and overall project management were some of the tasks Rohini Rachita particularly shined at.

Last but not least, I must thank all the people on Internet that enjoyed those challenges. This work was made for you. My goal was to help you in discovering some subtleties or pitfalls of that great tool the Bash is. And given your comments and reactions on Internet, I know you liked that work.I I sincerely hope you will have even more fun today by reading this enhanced version of the Bash Challenges!

Sylvain Leroux
June 2017
Yes I Know It

Contents

Acknowledgement **3**

Contents **4**

Publisher’s Foreword **8**

How to use this Bash Challenge book? **9**

Before you go on bashing **10**

Challenge 1: Counting files in the current directory **12**

What I tried to achieve? 12

The solution 13

How to reproduce? 13

What was the problem here? 13

How to fix that? 13

Ignore the file's name 14

Challenge 2: My shell don't know how to count **15**

What I tried to achieve? 15

The solution 16

How to reproduce? 16

What was the problem? 16

How to fix that? 17

Removing leading zeros 17

Specifying explicitly the base 18

Challenge 3: My command outputs are in the wrong order! **19**

What I tried to achieve? 19

The solution 20

What was the problem? 20

Challenge 4: Keeping filenames containing some extension **22**

What I tried to achieve? 22

The solution 23

What was the problem? 23

How to fix that? 23

Protecting the dot from special interpretation 23

Anchoring a pattern at the end of a line 24

Challenge 5: The lazy typist challenge **25**
What I tried to achieve? 25
My solution 26
Challenge 6: The dangerous file to remove **27**
What I tried to achieve? 27
The solution 28
What was the problem? 28
How to solve that? 28
Challenge 7: The file that didn't want to go away **30**
What I tried to achieve? 30
The solution 31
What was the problem? 31
How to achieve my goal? 33
Challenge 8: Hex to ASCII conversion in Bash **35**
What I tried to achieve? 35
The solution 36
How to reproduce 36
What was the problem here? 36
How to fix that? 37
Alternate solutions 37
Challenge 9: My Bash can't sum data in columns **38**
What I tried to achieve? 38
What was the problem? 39
How to fix that? 39
Add the missing zeros 40
Split data on fixed position 40
Challenge 10: The file that survived to rm **41**
What I tried to achieve? 41
The solution 42
How to reproduce 42
What was the problem? 42
How to fix that? 43
Challenge 11: The red/blue token counter **44**
What I tried to achieve? 44
The solution 45
What was the problem? 45

The store and process approach 45
Duplicate stream 46
Handle data as they arrive 46

Challenge 12: Inserting the same header on top of several different files 48
What I tried to achieve? 48
The solution 49
What was the problem? 49
How to fix that? 50
The KISS solution 50
Using sed 50
Using ed or ex 51

Challenge 13: Converting text to uppercase 52
What I tried to achieve? 52
The solution 52

Challenge 14: The Back in Time Function 54
What I tried to achieve? 54
The solution 55
What was the problem? 55
How to fix that? 55
Converting date to quantities 55
Using the mighty powers of GNU date utils 56

Challenge 15: My Bash don't know how to count. Again! 58
What I tried to achieve? 58
The solution 58
What was the problem? 58
How to fix that? 58

Challenge 16: Sending a file between two computers 60
What I tried to achieve? 60
The solution 60
What was the problem? 60
Introducing netcat 60
Replacing the client netcat by the Bash 62

Challenge 17: Generate a fair dice roll 64
What I tried to achieve? 64
The solution 64
What was the problem? 64

What was the problem, really? 65
How to fix that? 67

Afterword **69**

Publisher’s Foreword

Bash It Out is our first attempt in bringing out unique and helpful Linux related educational material in book format.

Some of you might already know us from the It's FOSS Linux Blog (https://itsfoss.com), an open source web portal promoting Linux and Open Source. With over 250,000 members in our community, we feel pride in being one of the most prominent voices in the world of Open Source.

What you are reading as a book was started as a mere Facebook post. It was liked by many Linux enthusiasts and that encouraged us to cover it on the website itself.

Eventually, we thought of putting it all in a more organized form and thus came out Bash it Out!

It consists of a few challenges that we already put on our website and some exclusively new problems.

We plan to add more new problems in this book in future.

I hope you enjoy solving these problems and learn new things from the solutions.

Abhishek Prakash
Co-Founder, It’s FOSS

How to use this Bash Challenge book?

Since you are taking the Bash challenge, we presume that you are aware of the basic fundamentals of the Bash scripting. You don't have to be a command line ninja to take up these challenges but you must know a thing or two about Bash and Linux/Unix commands.

The problem given here doesn't require you to write a Bash script. It rather presents you with a scenario and asks you why the output is not the expected one or why the script is behaving like this while it shouldn't.

The Bash scripting challenges in this book are divided into 3 levels of difficulties. So, if you are a beginner, you'll learn a lot with higher level challenges. And if you are a pro, you can jump to expert level challenges straightaway though I suggest you take all the challenges to test your knowledge.

The primary aim of this book is not to teach you Bash scripting. We aim to provide you with tricky question that will force you to go deeper with your Bash knowledge. You won't find these things in text books.

While taking the challenges, you should refer to man pages of the commands or Google for their usage. There is no restriction on that.

One important thing here, we provide one solution to each Bash exercise here. But there can be in fact more than one ways to solve the same problem. So if you don't find your solution in the book, feel free to discuss it on our Facebook page or by sending us an email.

Enough talk! Come on, Bash It Out!

Before you go on bashing

A few details you should know. To create this challenge, I used:

- GNU Bash, version 4.4.5 (x86_64-pc-linux-gnu)
- Debian 4.8.7-1 (amd64)
- All commands are those shipped with a standard Debian distribution
- **No command was aliased**

The challenges are divided in three levels.

- level 1 challenges covers tricks accessible to all Bash scripters;
- level 2 challenges requires some knowledge of a specific command or shell feature;
- level 3 challenges are more tricky and may require more advanced concepts or more subtle solutions.

That being said, the division is rather arbitrary, so don't be afraid to try the challenges in the order you want. The only real prerequisites here are some basic Bash syntax knowledge and most important, the desire to learn while having fun.

If you feel like discussing any problem, feel free to reach out by any of these three means:

- Email: sylvain@yesik.it
- Facebook page: https://www.facebook.com/Yes.I.Know.IT/
- Website: https://yesik.it

So, let's play together!

Level 1
Bash challenge

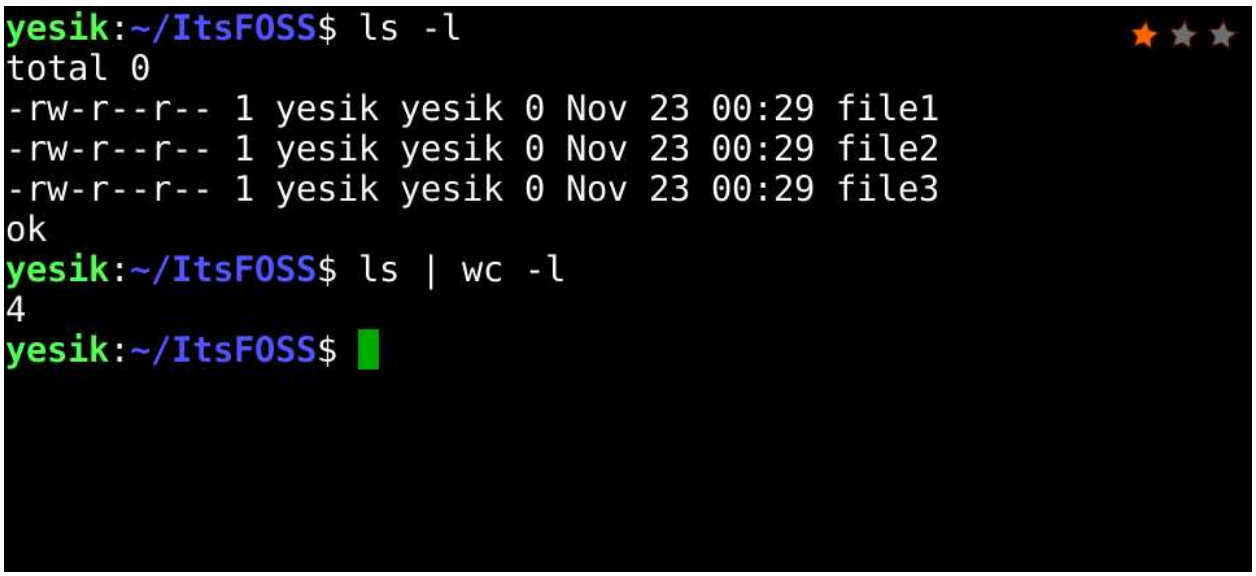
Challenge 1: Counting files in the current directory

What I tried to achieve?

I wanted to count the number of files in the current directory. For that, I used the `ls` and `wc` commands:

```
yesik:~/ItsFOSS$ ls -l
total 0
-rw-r--r-- 1 yesik yesik 0 Nov 22 22:45 file1
-rw-r--r-- 1 yesik yesik 0 Nov 22 22:45 file2
-rw-r--r-- 1 yesik yesik 0 Nov 22 22:45 file3
ok
yesik:~/ItsFOSS$ ls | wc -l
4
```

I was expecting a result of 3 since I visibly have three files in that directory. But for some unknown reason, that was not the result I've obtained. Could you explain me why? And most important, how to achieve my goal?



The solution

How to reproduce?

Here is the raw code we used to produce this challenge. If you run that in a terminal, you will be able to reproduce exactly the same result as displayed in the challenge illustration (assuming you are using the same software version as me):

```
mkdir -p ItsFOSS
cd ItsFOSS
touch file1 file2 $'file3\nok'
clear
ls -l
ls | wc -l
```

What was the problem here?

When send to a pipe, the `ls` command write each filename on its line. Just like when using `ls -1` from a terminal. And the `wc -l` command count *lines*. Apparently `ls | wc -l` should display the number of files or folder in the current directory.

However a filename may contain the newline character (often denoted `\n`). It is certainly uncommon but perfectly valid though.

In that challenge, the ok word you can see on the screen capture *was* part of the third filename, which in fact was `file3\nok`. Not `file3` as one may believe it at first sight.

This is a corner case you must take that into account in your scripts or shell commands to not break if a filename contains that character. Either by accident, or as the result of some malicious activity.

How to fix that?

Remove non-printable characters from the `ls` output

```
ls -q | wc -l
```

Modern versions of `ls` have the `-q` option that will replace non-printable characters by a question mark (`?`) The `-q` option is more portable than the `-b` option you may sometimes see used for that purpose. In both cases, we are absolutely certain `\n` embedded in filename will not interfere with our count. According to POSIX:

```
-q Force each instance of non-printable filename characters and <tab>s to be
written as the question-mark ( '?' ) character. Implementations may provide
this option by default if the output is to a terminal device.
```

On my Debian Squeeze test system, `-q` was not the default. That's why the `ok` word appears on its own line. On Debian Stretch, `-q` is enabled by default and the output is different — **only when the output is a console** — and it will display the last filename as `file3?ok`.

Ignore the file's name

I want to count the number of *files*. Not the number of *filenames*. So, as an alternate solution, we could just ignore the filename and issue a *token* for each encountered file. counting the number of tokens will give the same result as counting the number of files. And as we have the control of the token, we cannot be fooled. We can use the `find` command for that purpose:

```
find . -mindepth 1 -maxdepth 1 -printf '\n' | wc -l
-or-
find . ! -path . -maxdepth 1 -printf '\n' | wc -l
```

Here I made the choice of using an empty line (`\n`) as the token. So, those two command will produce one empty line per entry in the current directory. Notice the actual filename is never written to the output. So whatever characters may be embedded in the filename, they will not interfere with following commands in the pipe. I just have then to count the number of lines to know the number of files there was.

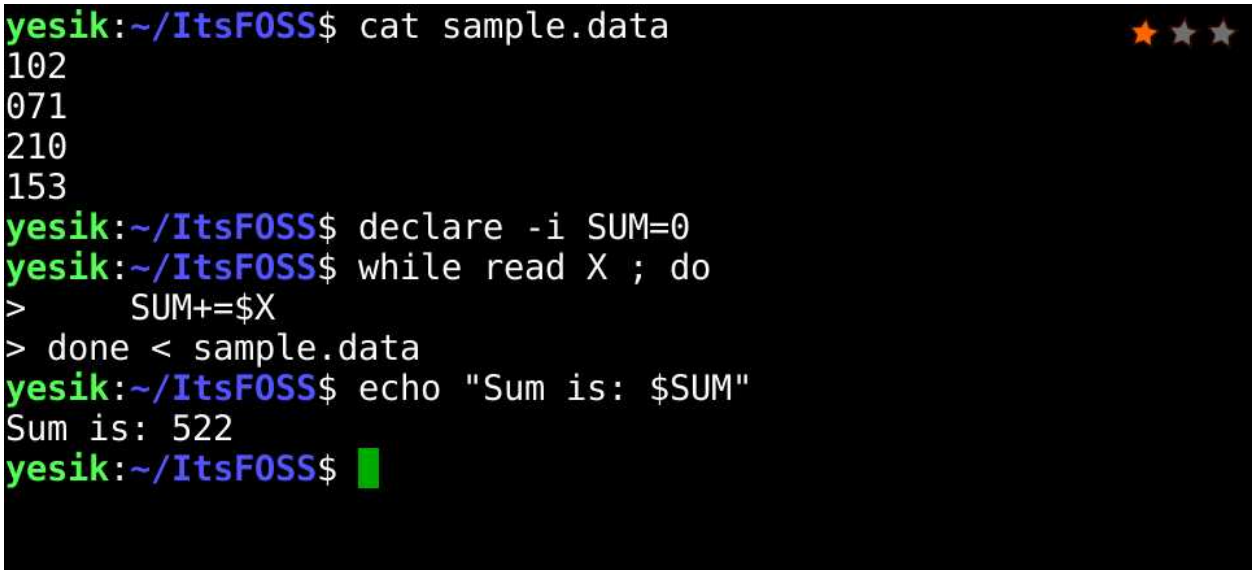
Please notice `-mindepth 1` and `! -path .`: those are two different tricks to remove the current directory (`.`) from the selection. Otherwise, your count will be off by one.

Challenge 2: My shell don't know how to count

What I tried to achieve?

I have some data file containing integer numbers, one on each line, and I want to compute the sum of all those numbers:

```
yesik:~/ItsFOSS$ cat sample.data
102
071
210
153
yesik:~/ItsFOSS$ declare -i SUM=0
yesik:~/ItsFOSS$ while read X ; do
>     SUM+=$X
> done < sample.data
yesik:~/ItsFOSS$ echo "Sum is: $SUM"
Sum is: 522
```



Unfortunately, the result I obtain is wrong (the expected result was 536).
Could you explain why the result is wrong and fix my commands to obtain the correct result.

Extra challenge:
could you find a solution using only Bash internal commands and/or shell substitutions.

The solution

How to reproduce?

Here is the raw code we used to produce this challenge. If you run that in a terminal, you will be able to reproduce exactly the same result as displayed in the challenge illustration (assuming you are using the same software version as me):

```
rm -rf ItsFOSS
mkdir -p ItsFOSS
cd ItsFOSS
cat > sample.data << 'EOT'
102
071
210
153
EOT
clear
cat sample.data
declare -i SUM=0
while read X ; do
    SUM+=$X
done < sample.data
echo "Sum is: $SUM"
```

What was the problem?

The problem was caused by the 071 value. As you noticed, this number is starting by a 0 — probably to ensure all data are formatted on three digits. Nothing complicated here, except that … following an unfortunate convention inherited from the C programming language, prefixing an integer by 0 is a way to specify that number is expressed in **octal**, and not in **decimal**.

Octal numbers are expressed with digits from 0 to 7. Here is a simple conversion table:

Decimal	Octal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Decimal	Octal
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17

Decimal	Octal
16	20
17	21
18	22
19	23
20	24
21	25
22	26
23	27

...

Decimal	Octal
56	10
57	**71**
58	72
59	73
60	74
61	75
62	76
63	77

The value in bold in the above table is the one that caused the error when evaluating the sum. The Bash read 071 and, because of the leading 0, interpreted it as the octal number 71_8 representing the 57_{10} decimal value. You can check that easily:

```
echo $((071))
57
```

How to fix that?

I can see two main strategies to fix that issue. Either removing the leading zeros. Or finding a way to make the shell understand all my numbers are **decimal** values.

Removing leading zeros

Here is a simple solution using the sed external command to remove the leading zeros:

```
declare -i SUM=0
while read X ; do
    SUM+=$X
done < <(sed -E s/^0+// sample.data)
echo "Sum is: $SUM"
```

Bonus question:
why didn't I used a *pipe* instead of a *process substitution*?

Specifying explicitly the base

The previous solution is (mostly) straightforward — but the Bash allows us to make things better. Instead of trying to fix the data, we will simply specify *explicitly* our numbers are expressed in base 10 (decimal), instead of base 8 (octal). You can do that by using the `base#value` syntax.

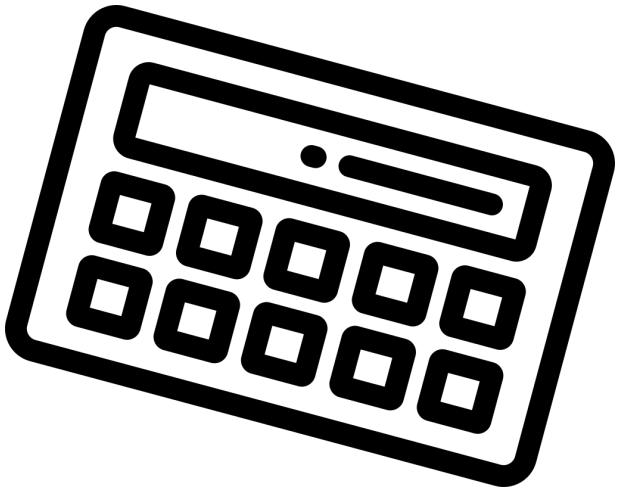
Compare those three examples:

```
echo $((071))        # The leading 0 specify the number as octal
57
echo $((8#071))      # We *explicitly* specify base 8 (octal)
57
echo $((10#071))     # We *explicitly* specify base 10 (decimal)
71
```

To fix my initial command and obtain the correct result, I only have to explicitly specify the base 10 for all my data:

```
declare -i SUM=0
while read X ; do
    SUM+=$((10#$X))
done < sample.data
echo "Sum is: $SUM"
```

I let you check that yourself, but it definitely should produce the correct result this time!



Challenge 3: My command outputs are in the wrong order!

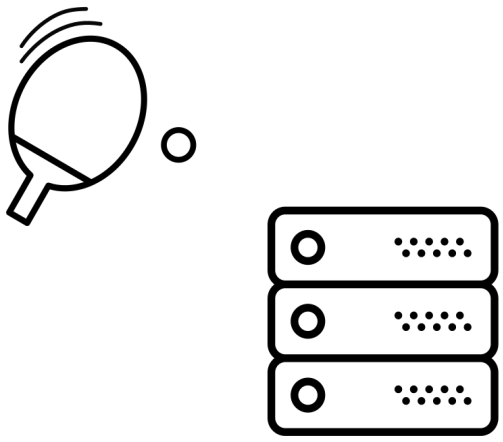
What I tried to achieve?

This time, I want a shell function to log the round trip time (rtt) to a server. Only if the ping command has succeeded, I want to record the date of the measure on the line **below** the rtt.

Given those requirements, I end up with that solution:

```
yesik:~/ItsFOSS$ probe() (
>   ping -qnc2 www.google.com | \
>     grep rtt & \
>     date +"OK %D %T"
> )
yesik:~/ItsFOSS$ rm -f log
yesik:~/ItsFOSS$ probe >> log
yesik:~/ItsFOSS$ probe >> log
yesik:~/ItsFOSS$ cat log
OK 11/22/16 22:52:36
rtt min/avg/max/mdev = 53.394/77.140/100.887/23.748 ms
OK 11/22/16 22:52:39
rtt min/avg/max/mdev = 49.142/49.731/50.320/0.589 ms
```

But, I don't understand why the date and rtt lines are *swapped* in the log file?!? The date should appear *below* the rtt. But here it appears *above*. Why? Could you fix that?



The solution

What was the problem?

I've simply made a typo: I mistaken `&` for `&&` — maybe was I confused by the pipe (`|`) symbol above? Indeed, all the `|`, `||`, `&` and `&&` operators can be used to join two shell commands. But they have completely different meanings:

cmd1 | cmd2	The pipe symbol	Run both commands in parallel in a sub-shell, using the output of cmd1 as input to cmd2. The pipe is a very common way to combine several basic commands in order to accomplish complex tasks.
cmd1 & cmd2	The ampersand	Run cmd1 as a background process, and in parallel, to run cmd2 in the foreground. The two commands are not connected in any way using that operator.
cmd1 || cmd2	The short-circuit logical OR	Run cmd2 only if cmd1 has failed. As a consequence cmd1 must complete before cmd2 is eventually run. In other words, commands run sequentially.
cmd1 && cmd2	The short-circuit logical AND	Run cmd2 only if cmd1 was successful. As a consequence cmd1 must complete before cmd2 is eventually run. In other words, commands run sequentially.

Armed with that knowledge, let's now take a look at my original code:

```
probe() (
  ping -qnc2 www.google.com | \
    grep rtt & \
    date +"OK %D %T"
)
```

1. I want to run the ping command and send its output to the grep command. The pipe is the right operator.
2. But after that, I wanted to write the date only if the pipe was successful. Here, I needed the logical AND operator (`&&`). But instead of that, I used the `&` operator, that basically run `ping | grep` into the background — always. And `date` in the foreground — always. There is a *race condition* as both processes are now running in parallel and compete to write on stdout (the terminal output). Unsurprisingly, in that particular example, the `date` command won every time over the `ping` command.

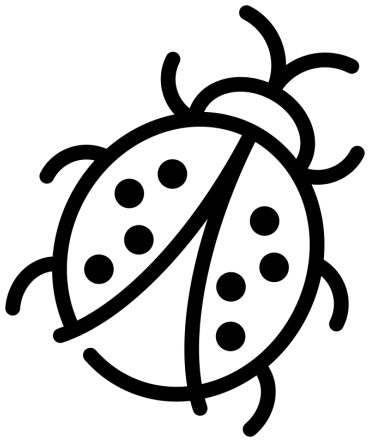
Therefore the correct syntax would have been:

```
probe() (
  ping -qnc2 www.google.com | \
    grep rtt && \
    date +"OK %D %T"
)
```

In my case, the issue was immediately visible because, obviously, the ping command takes more time to complete than the date command. But, as this is often the case with race conditions, such mistakes could easily remain hidden for a very long time when the two "contestants" of the race take *almost* the same time to complete. For example, the following example is a lot less deterministic:

```
probe() (
   ping -qnc2 itsfoss.com | sed 1q & \
   ping -qnc2 kernel.org | sed 1q
)
```

From my location in France, on 2000 runs, the first ping lost only 3 times. That means the "bug" was visible only in 0.15% of the cases. Next time you'll report some occasional software crash — be kind with your favorite FOSS developers and remember that even caused by typos, race conditions are hard to reproduce and even harder to trace!



Challenge 4: Keeping filenames containing some extension

What I tried to achieve?

In a file, I have a list of Windows filenames stored one per line (this is part of the logs from an auditing tool running on my Samba server). I want to extract from that list all files containing the `.bat` extension. That extension must be spelled exactly like that and must appear at the end of the filename(i.e.: I don't want to keep `.bat.orig` files nor `.batch` files).

The `grep` command is the canonical tool when you want to find lines containing some pattern in a file. Unfortunately, I wasn't able to achieve my intended result:

```
yesik:~/ItsFOSS$ cat sample.data
login.bat
login.exe
logout.batch
acrobat.exe
first-run.pdf
first-run.bat
first-run.bat.orig
yesik:~/ItsFOSS$ grep .bat sample.data
login.bat
logout.batch
acrobat.exe
first-run.bat
first-run.bat.orig
```

As you can see, my solution kept filenames I didn't want.

It is somewhat understandable for the `.batch` or `.bat.orig` files. But could you explain why `acrobat.exe` was retained too? And how could you fix my solution to achieve the intended goal?

The solution

What was the problem?

My issue here was caused by the `grep` command taking a r*egular expression* as search pattern. Not a fixed string.

Regular expressions allows to describe a *set* of strings. This is achieved through the use of metacharacters. That is characters which have a special meaning rather that matching literally with their value.

The dot is such a character. A dot in a regular expression will match *any* character. So, when I write:

```
grep .bat sample.data
```

That means I want to keep lines containing the three letters `b`, `a` and `t`, preceded by *any* character. Either a verbatim dot. Or any other character.

How to fix that?

Protecting the dot from special interpretation

To fix that, I may remove the special meaning of the dot. In a regular expression, you remove the special meaning of a metacharacter by preceding it with a backslash.

However there is a pitfall here: the argument string is processed *twice*. Once by the shell that will parse the command. Then a second time by the `grep` command that will interpret the regular expression. So, for the grep command to see the backslash, we must escape that one first from the shell interpretation by using...a second backslash:

```
grep \\.bat sample.data
```

If you don't like the "double backslash", a second option is to use single quotes to protect the whole pattern from any shell interpretation:

```
grep '\.bat' sample.data
```

Finally, if you *really* don't like backslashes, you may use a character set in the regular expression:

```
grep '[.]bat' sample.data
```

A character set will match any character between the brackets. If you only put a dot between the brackets, it can only match a verbatim dot.

Anchoring a pattern at the end of a line

The three solutions above removed the special meaning of the dot. But how to keep only files *ending* by `.bat`? Once again the solution lies in the regular expression metacharacters. Especially the dollar sign will match the end of the line. And once again, since the dollar sign has special meaning in the shell too, we must protect it from the shell interpretation.

All that finally leading to those possible solutions:

```
yesik:~/ItsFOSS$ grep '\.bat$' sample.data
login.bat
first-run.bat
```

Or

```
yesik:~/ItsFOSS$ grep \\.bat\$ sample.data
login.bat
first-run.bat
```

or … I let you find a third one, maybe using some brackets?

Challenge 5: The lazy typist challenge

What I tried to achieve?

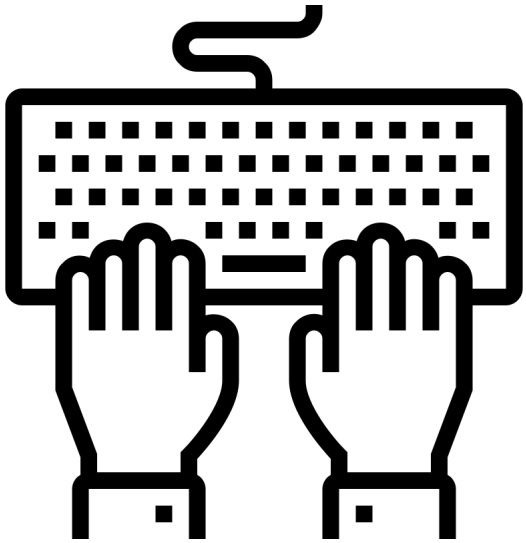
I have the following content in my current working directory:

```
yesik:~/ItsFOSS$ ls -F
archives/
report-fall-2015.pdf
report-summer-2015.pdf
report-summer-2016.pdf
report-winter-2014.pdf
```

I just want to copy the summer 2016 & fall 2015 reports in the archive folder. That's pretty simple:

```
yesik:~/ItsFOSS$ cp report-summer-2016.pdf archives/
yesik:~/ItsFOSS$ cp report-fall-2015.pdf archives/
```

However as I'm lazy, I cannot satisfy with those commands as they require 70 keystrokes. That's way too much for my poor fingers! Could you help me in finding a way to copy those files with the **minimum number of keystrokes**.



My solution

For this challenge I will not claim to have "the" solution. Maybe you will be able to achieve a better score than myself? In that case, you really deserve congratulations!

Anyway...
The first and most obvious optimization will be to replace the two commands by only one:

```
cp report-summer-2016.pdf report-fall-2015.pdf archives/
[57 characters]
```

Then we can use *glob patterns* to shorten each filename. Since there is only one "fall" report, for this one this is easy, but we must take extra cares for the summer report since I'm only interested in the 206 version:

```
cp report-s*-2016.pdf report-fall*.pdf archives/
[49 characters]
```

As a matter of fact, given the other filenames structure, I can go even further into that way:

```
cp *6* *fa* archives/
[22 characters]
```

By the way we can do the same for the destination directory—which is the only subdirectory here:

```
cp *6* *fa* */
[15 characters]
```

In some cases, using brace expansion can further reduce the number of required keystrokes. But in that case, we remain stuck at 15 characters:

```
cp *{6,fa}* */
cp *{6*,fa*,/}
```

Unless *you* found something "better"?

Challenge 6: The dangerous file to remove

What I tried to achieve?

When I was at school (in the VT100 era:/) it wasn't uncommon for the poor soul that has left his terminal without closing the session to find strangely named files in his home directory when he was back.

That was "cruel jokes" made by other students passing by. For this challenge, let's pretend you just found a file named `-rf *` in your home directory:


```
yesik:~/ItsFOSS$ ls -1
dont remove!
-rf *
```

So, how to remove the unwanted file from the command line, without removing any other file? Obviously `rm -rf *` is *not* the solution…

BEWARE:
before trying to answer this challenge, think first why this was a "cruel joke". What are the risks with the name of this file? IF YOU CAN'T ANSWER THAT QUESTION FIRST DON'T TRY THIS AS HOME !

The solution

What was the problem?

Here the problem is caused by a filename that looks like a command option. We have to find a way to avoid the `rm` command to understand the dash as introducing a set of options (and indeed the `-rf` option is particularly dangerous).

Maybe that was you first reflex, but no, using quotes or backslash is *not* the solution. None of these syntax will work:

```
yesik:~/ItsFOSS$ rm '-rf *'
rm: invalid option -- ' '
Try 'rm --help' for more information.
yesik:~/ItsFOSS$ rm \-rf\ \*
rm: invalid option -- ' '
Try 'rm --help' for more information.
```

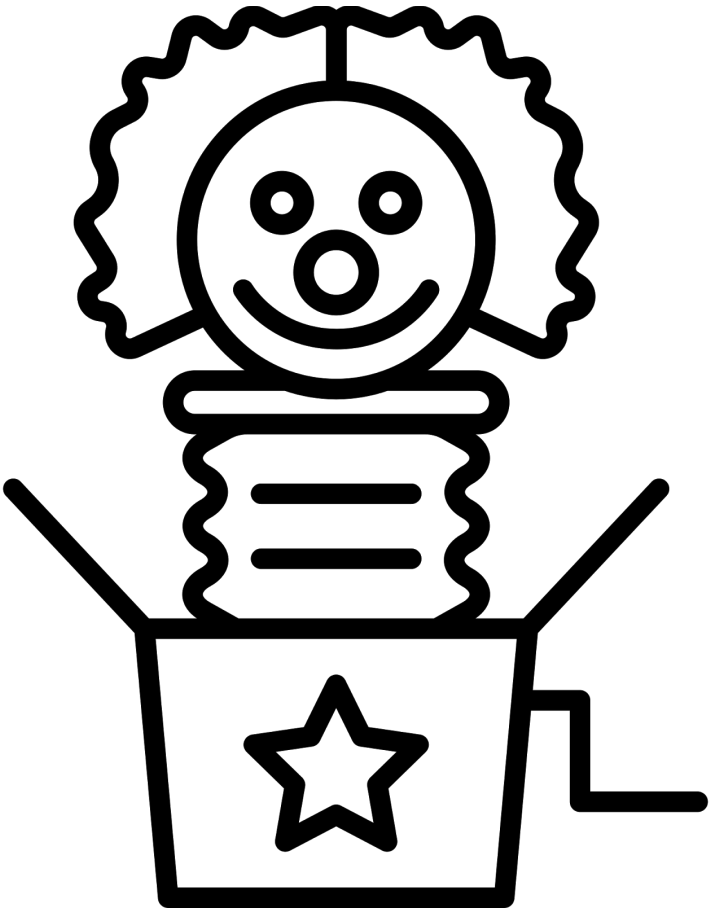
Why those are not valid solutions? Because they would protect special characters from shell's interpretation. But here the problem is not with the shell. But with the `rm` command itself.

How to solve that?

Hopefully, `rm` like several other standard command supports the `--` special option to indicate the start of filename list. After `--` the command will no longer try to interpret strings starting by a dash as an option. Which is exactly what we need:

```
yesik:~/ItsFOSS$ rm -- '-rf *'
```

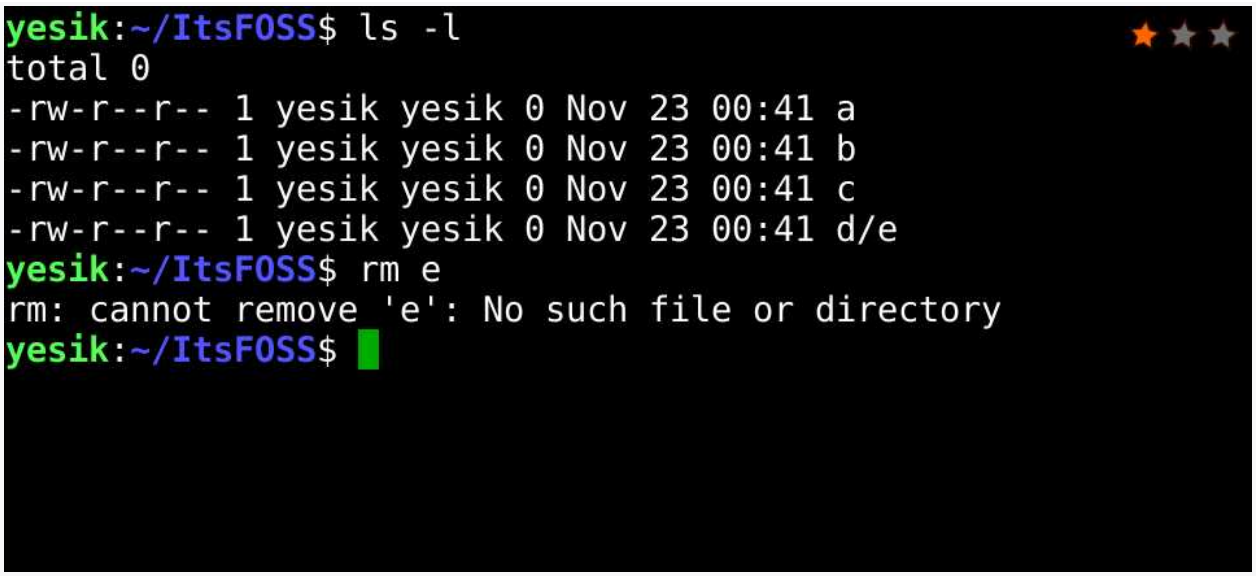
Notice however than quotes (or backslashes) are still necessary to protect the space and the star from shell interpretation. If you want to experiment more with that, I encourage you to use the `ls` and `touch` commands, since both of them supports the `--` option. And they are way less dangerous to use than the `rm` command!



Challenge 7: The file that didn't want to go away

What I tried to achieve?

I was asked to remove the last file of the list displayed below:




```
yesik:~/ItsFOSS$ ls -l
total 0
-rw-r--r-- 1 yesik yesik 0 Nov 22 22:58 a
-rw-r--r-- 1 yesik yesik 0 Nov 22 22:58 b
-rw-r--r-- 1 yesik yesik 0 Nov 22 22:58 c
-rw-r--r-- 1 yesik yesik 0 Nov 22 22:58 d⁄e
yesik:~/ItsFOSS$ rm e
rm: cannot remove 'e': No such file or directory
```

But as you can see, a simple `rm e` command didn't work.

Why? How to remove that file?

The solution

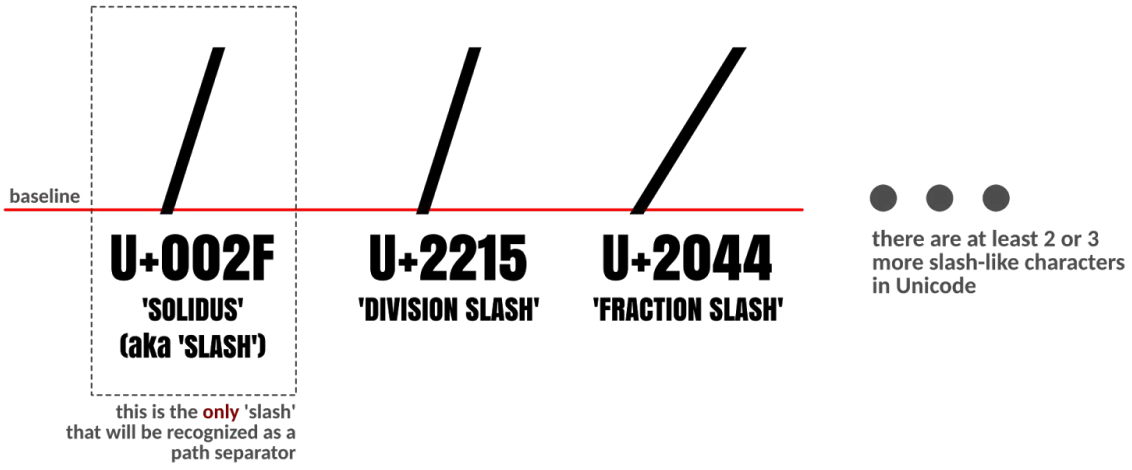
What was the problem?

You know a slash cannot be part of a filename. And you may have somehow believed the file `e` was in some subdirectory `d`.

But that's wrong.

First, there is no reason for the `ls -l` command to display the content of a subdirectory using that format, And then, we can see, from the rest of the `ls` output, there is no subdirectory named `d` here.

This problem is a typical homoglyphic confusion. You believed you've seen a slash. But in fact it was a different character, but displayed with an unfortunately confusing glyph. In that particular case, it was the U+2215 DIVISION SLASH Unicode character.



On my UTF-8 terminal the U+2215 DIVISION SLASH Unicode character is encoded using the three bytes sequence 0xE2 0x88 0x95. You can check that using the `hexdump` command (with a little bit of shell scripting for a more readable output):

```
yesik:~/ItsFOSS$ for f in *; do echo -n $f | hexdump -C; done
00000000  61                                                |a|
00000001
00000000  62                                                |b|
00000001
00000000  63                                                |c|
00000001
00000000  64 e2 88 95 65                                    |d...e|
00000005
```

If you have `iconv` installed on your system, you can even find the corresponding Unicode code point by converting the data to utf-16:

```
yesik:~/ItsFOSS$ for f in *; do echo -n $f | iconv -t utf16be |
hexdump -C; done
00000000  00 61                                             |.a|
00000002
00000000  00 62                                             |.b|
00000002
00000000  00 63                                             |.c|
00000002
00000000  00 64 22 15 00 65                                 |.d"..e|
00000006
```

How to achieve my goal?

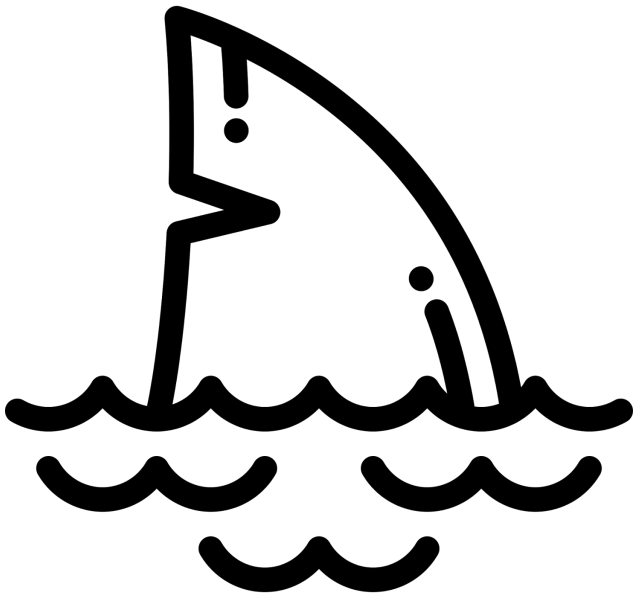
Knowing that, it is now easy to remove the file...if you know the Bash `$'string'` syntax:

```
yesik:~/ItsFOSS$ rm $'d\xe2\x88\x95e' # using the UTF-8 encoding
-or-
yesik:~/ItsFOSS$ rm $'d\u2215e'       # using unicode code point
```

By the way, this would have worked too:

```
yesik:~/ItsFOSS$ rm d*
```

But you will agree, by doing this, we would have missed an interesting discussion!



Level 2
Bash challenge

Challenge 8: Hex to ASCII conversion in Bash

What I tried to achieve?

I have a file containing some "secret" message:

```
yesik:~/ItsFOSS$ cat SECRET
43 4F 4E 47 52 41 54 55 4C 41 54 49 4F 4E 53 20 46 4F 52 20 48 41
56 49 4E 47 20 53 4F 4C 56 45 44 20 54 48 49 53 20 43 48 41 4C 4C
45 4E 47 45
yesik:~/ItsFOSS$ decode < SECRET
CONGRATULATIONS FOR HAVING SOLVED THIS CHALLENGE
```

The secret message is simply "ASCII encoded", so it is not difficult to find the original text by consulting an ASCII table:

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

For example, the code `43` is corresponding to the letter `C` (row 4, column 3 in the above table). `4F` is the letter `O`. And so on.

But I wouldn't do that by hand when a computer can do it for me. So, could you write the `decode` Bash function I used?

If needed, see http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-8.html for an introduction to Bash function syntax.

And remember, this is a *Bash* challenge. Your solution must not use any other programming language (no Perl, C, Python, …)

Extra challenge:
could you write a `decode` function using only Bash internal commands & shell expansion (i.e.: `strace -e trace=process …` will show no new process creation)

The solution

How to reproduce

Here is the raw code we used to produce this challenge. If you run that in a terminal, you will be able to reproduce exactly the same result as displayed in the challenge illustration (assuming you are using the same software version as me):

```
rm -rf ItsFOSS
mkdir -p ItsFOSS
cd ItsFOSS
M="CONGRATULATIONS FOR HAVING SOLVED THIS CHALLENGE"
( echo -n "$M" | hexdump -v -e '/1 "%02X "'; echo ) > SECRET
decode() {
    while read -d' ' c ; do
        echo -en '\x'$c
    done
    echo
}
clear
cat SECRET
decode < SECRET
```

What was the problem here?

In the SECRET file, each character of the message is represented by its ASCII code expressed in *hexadecimal*. 41 → A 42 → B … 49 → I 4A → J 4B → K … 4F → O 50 → P … 5A → Z
We had to find a way to read the data and to perform the ASCII → char conversion.

How to fix that?

```
decode() {
    while read -d' ' c ; do
        echo -en '\x'$c
    done
    echo
}
```

In that solution, the decode function will read the message hexadecimal number by hexadecimal number, and will use the echo internal command to output the corresponding character. The trick here is the `-e` option allowing the echo command to understand `\xNN` sequences as character codes.

You can try the -e option easily:

```
echo -e '\x48\x45\x4C\x4C\x4F'
```

Alternate solutions

Other solutions are available. The most obvious would be to use the `xxd` tool which has a dedicated reverse (`-r`) mode:

```
decode() { xxd -r -p; echo; }
```

In addition, I cannot resist in showing you a fragile but somewhat clever solution:

```
decode() {
  S=' '$(echo $(</dev/stdin))
  echo -e ${S// /\\x}
}
```

The first line reads the entire input data into a variable, ensuring all hexadecimal numbers are preceded by one (and only one) space. After that, in the second line, we substitute each space by the \x prefix so echo -e can perform the conversion. I said this solution was fragile as it will break for very large files. Could you understand why?

Challenge 9: My Bash can't sum data in columns

What I tried to achieve?

I have some integer data, stored as a fixed width text file. I just want to find the total per column of those data. However, some data are missing in the file. And those should be accounted as 0.

Unfortunately, that seems to confuse my Bash script:

```
yesik:~/ItsFOSS$ cat sample.data
    5    3    7    2
    1           -12
              0   -7
       -14         4
                  15
yesik:~/ItsFOSS$ declare -i SW SX SY SZ
yesik:~/ItsFOSS$ while read W X Y Z ; do
>     SW+=$W ; SX+=$X ; SY+=$Y ; SZ+=$Z
> done < sample.data
yesik:~/ItsFOSS$ printf " %4d %4d %4d %4d\n" ${W,X,Y,Z}
    7  -12    7    2
```

I expected the result "6 -11 7 2". But for some unknown reason, the total for the first two columns as calculated by my shell is off by 1. Could you fix that?

You can assume:

- ✓ the data file follow strictly the formatting used in my "printf" command.
- ✓ the last column is dense (i.e.: no missing value)

Extra challenge:
could you find a solution using only Bash internal commands and/or shell expansion?

What was the problem?

The default behavior of the `read` Bash internal command is to discard any space at the start of a line and to consider a group of spaces in a middle of a line as one and only one separator.

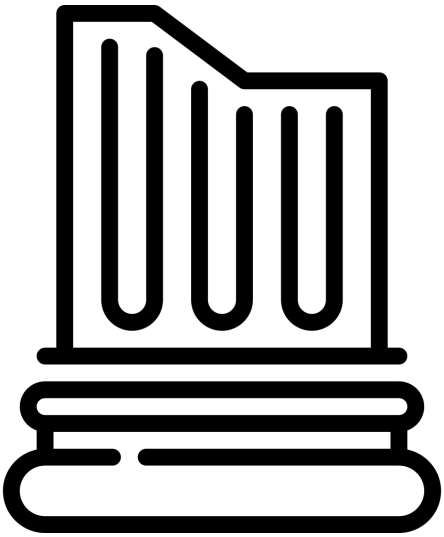
So basically, what appears to us as a fixed width data file, will be read a space-delimited data file by the `read` command. Just like if the datafile was:

```
while read ; do echo $REPLY; done < sample.data
5 3 7 2
1 -12
0 -7
-14 4
15
```

When displayed like that, it is more obvious why the sum of the first column was 7 and not 8 as we expected it.

How to fix that?

I can see two main strategies to obtain the right result. First, I could just add the missing 0 value in the empty cells, so we will always have 4 integer numbers to read on each line. The second strategy will be to enforce the fixed width format in the code.



Add the missing zeros

Here is a possible solution for the first strategy described above:

```
declare -i SW=0 SX=0 SY=0 SZ=0
while read W X Y Z ; do
    SW+=$W ; SX+=$X ; SY+=$Y ; SZ+=$Z
done < <(sed 's/     /    0/' sample.data)
printf " %4d %4d %4d %4d\n" $S{W,X,Y,Z}
```

In that solution, I put a zero in the data each time I encounter a sequence of 5 spaces. Now, I have 4 integer numbers on each line, and the read command correctly parses the data so my totals are correct.

Bonus questions:
Could you try to shorten that sed expression?

Why using a pipe to connect the `sed` command and `while` loop wouldn't have been such a good idea[1]?

Split data on fixed position

A completely different strategy would be to use the so-called Bash substring expansion using the `${parameter:offset:length}` syntax. That way you can slice the data at *fixed* positions. But for that to work, we need a second "trick": being able to read an entire line at once.

To achieve that, you need to know the `IFS` variable. It stores the separator used by the Bash to split a line into words. To read an entire line at once, that is to ignore any separator, we simply need to temporary set `ISF` to the empty string.

All that leading to:

```
declare -i SW=0 SX=0 SY=0 SZ=0
while IFS='' read LINE; do
    SW+=${LINE:0:5}   ; SX+=${LINE:5:5}
    SY+=${LINE:10:5} ; SZ+=${LINE:15:5}
done < sample.data
printf " %4d %4d %4d %4d\n" $S{W,X,Y,Z}
```

[1] Maybe you could find the answer to that question in another challenge...

Challenge 10: The file that survived to rm

What I tried to achieve?

The description for this challenge is quite short:

I have three files in a directory. As root, I used `rm *` in that directory.

But there is one file that obstinately refuses to be deleted:

```
root:011# ls -ls
total 12
4 -rw-r--r-- 1 root root 29 nov 21 21:25 a
4 -rw-r--r-- 1 root root 29 nov 21 21:25 b
4 -rw-r--r-- 1 root root 29 nov 21 21:23 c
root:012# rm *
rm: cannot remove 'c': Operation not permitted
root:013# rm -f c
rm: cannot remove 'c': Operation not permitted
root:014# ls -ls
total 4
4 -rw-r--r-- 1 root root 29 nov 21 21:23 c
```

Your challenge is to find:

1. What prevented the third file to be deleted;
2. How to actually delete that file.

The solution

How to reproduce

Here is the raw code we used to produce this challenge. If you run that in a terminal, you will be able to reproduce exactly the same result as displayed in the challenge illustration (assuming you are using the same software version as me):

```
# as root :
cd /tmp
rm -rf ItsFOSS
mkdir -p ItsFOSS
cd ItsFOSS
date > a
date > b
date > c
sudo chattr +i c
clear
ls -ls
rm *
rm -f c
ls -ls
```

What was the problem?

You may have noticed I used above the `chattr` command to set the (i)mmutable Linux filesystem attribute for the file `c`. Depending your exact filesystem, all attribute changes are not available.

But here, I am using and ext2 filesystem that *does* support the `i` flag. And to quote the man:

```
   A  file with the 'i' attribute cannot be modified: it cannot be deleted
   or renamed, no link can be created to this file  and  no  data  can be
   written  to  the  file.  Only the superuser or a process possessing the
   CAP_LINUX_IMMUTABLE capability can set or clear this attribute.
```

So basically after the `chattr +i` the file is locked until we clear this flag. Please notice the attribute is stored in the filesystem. So it will survive reboots & filesystem unmount/mount cycles.

How to fix that?

First, we can check the explanation above by using the `lsattr` command:

```
root:015# lsattr c
----i-------------- c
```

Clearly, the (i)mmutable flag is set. So, in order to remove that file (or to make any change to it), I have to clear that flag. After that, I can do whatever I want on the file as usual:

```
root:016# chattr -i c
root:017# lsattr c
-------------------- c
root:018# rm c
root:019# ls -ls
total 0
```

If you're not aware of the existence of `chattr`, its effects can be quite puzzling. Worth mentioning `chattr` is a Linux-specific command, originally written for the ext2/3/4 filesystems. But today's some of its feature are supported by other filesystems.

In the BSD-world, there is a similar command called `chflags`. Read more on Wikipedia(https://en.wikipedia.org/wiki/Chattr) for a gentle introduction to that commands compared to `chattr`.

Challenge 11: The red/blue token counter

What I tried to achieve?

This challenge is more "programming-oriented" that the previous ones. The description being a little bit abstract, try to stay with me for few minutes—hopefully, the description below should be clear enough:

I have a stream of tokens, either 'RED', 'BLUE' or 'GREEN'. If you want, you can consider that as a representation of an event stream for example[2]. I have no particular control on that stream. I just know it produces either one or the other token, unpredictably. And I know the steam is finite (i.e.: at some point, there will be no more data to read).

For the sake of this challenge, I used a Bash function to produce that stream. You are not allowed to change that in anyway.

```
# You MUST NOT change that:
 stream() {
   TOKENS=( "RED" "BLUE" "GREEN")
   for((i=0;i<100;++i)) ; do
     echo ${TOKENS[RANDOM%3]}
   done
 }
```

My goal is to to count the total of each different tokens there was in he stream. By myself, I was able find a solution to count the number of RED tokens:

```
# You MUST change that
yesik:~/ItsFOSS$ stream | \
     grep -F RED | wc -l > RED.CNT
yesik:~/ItsFOSS$ cat RED.CNT
38
```

Unfortunately, I couldn't find any solution to count each (RED, BLUE and GREEN) tokens. That's why I need your help. Any idea?

[2] This challenge is inspired from a real-world application where we had to monitor the customer flow in a store. There was sensors spread across strategic locations on the floor, and each sensor issued an event when it detected someone passing by.

The solution

What was the problem?

The only difficulty here was my initial attempt is discarding some part of the input, because I directly send the data stream to the grep.

Basically there are three approach to solve that problem:

- Store the stream data and process them afterward;
- Duplicate the stream and process independent path for RED, BLUE and GREEN tokens;
- Handle all cases in the same command as they arrive.

For what it worth, after each solution, I give the real-time usage observed on my system. This just an indication and has to be taken with caution. So feel free to make your own the comparison yourself !

The store and process approach

The simplest implementation of the store-and-process approach is obvious:

```
stream > stream.cache
grep -F RED < stream.cache | wc -l > RED.CNT
grep -F BLUE < stream.cache | wc -l > BLUE.CNT
grep -F GREEN < stream.cache | wc -l > GREEN.CNT
rm stream.cache
(1.3s for 10,000,000 tokens)
```

It works, but has several drawbacks: you have to store all the data before they are processed. In addition, we handle the three different cases sequentially. More subtle, as you read several times the `stream.cache` file, you potentially have some race condition if a concurrent process updates that file during processing.

Still in the store-and-process category, here is a completely different solution:

```
stream | sort | uniq -c
(5.9s for 10,000,000 tokens)
```

I consider that a store-and-process approach, since the `sort` command has to first read and store (either in RAM or on disk) all data before being able to process them. More precisely, on my

Debian system, the sort command creates several temporary file in `/tmp` with read-write permissions. Basically this solution has the same drawbacks as the very first one but with much worst performances.

Duplicate stream

Do we really have to *store* the data *before* processing them? No. A much more clever idea would be to split the stream in several parts, processing one kind of token in each sub-stream:

```
stream | tee >(grep -F RED | wc -l > RED.CNT) \
             >(grep -F BLUE | wc -l > BLUE.CNT) \
             >(grep -F GREEN | wc -l > GREEN.CNT) \
             > /dev/null
(0.8s for 10,000,000)
```

Here, there is no intermediate files. The `tee` command replicates the stream data as they arrive. Each processing unit gets its own copy of the data, and can process them on the fly.
This is a clever idea because not only we handle data as they arrive, but we have now parallel processing.

Handle data as they arrive

In computer science, we would probably say the previous solution took a functional approach to the problem. On the other hand, the next ones will be purely an imperative solution. Here, we will read each token in its turn, and *if* this is a RED token, *then* we will increment a RED counter, *else if* this is a BLUE token, we will increment a BLUE counter *else if* this is a GREEN token, we will increment a GREEN counter.

This is a plain Bash implementation of that idea:

```
declare -i RED=0 BLUE=0 GREEN=0
stream | while read TOKEN; do
    case "$TOKEN" in
        RED) RED+=1
             ;;
        BLUE) BLUE+=1
             ;;
        GREEN) GREEN+=1
             ;;
    esac
done
(103.2s for 10,000,000 tokens)
```

Finally, being a great fan of the `AWK` command, I will not resist the temptation of using it to solve that challenge in a neat and elegant way:

```
stream | awk '
  /RED/ { RED++ }
  /BLUE/ { BLUE++ }
  /GREEN/ { GREEN++ }
  END { printf "%5d %5d %5d\n",RED,BLUE,GREEN }
'
(2.6s for 10,000,000 tokens)
```

My AWK program is made of four rules:

1. When encountering a line containing the word RED, increase (++) the RED counter
2. When encountering a line containing the word BLUE, increase the BLUE counter
3. When encountering a line containing the word GREEN, increase the GREEN counter
4. At the END of the input, display both counters.

Of course to fully understand that you have to know, for the purpose of mathematical operators, *uninitialized* AWK variables are assumed to be zero.

That works great. But it requires duplication of the same rule for each token. Not a big deal here as we have only three different tokens. More annoying if we have many of them. To improve that solution, we could rely on *arrays*:

```
stream | awk '
  { C[$0]++ }
  END { printf "%5d %5d %5d\n",C["RED"],C["BLUE"],C["GREEN"] }
'
(2.0s for 10,000,000 tokens)
```

We only need two rules here, whatever is the number of tokens:

1. Whatever is the read token ($0) increase the corresponding array cell (here, either C["RED"], C["BLUE"] or C["GREEN"])
2. At the END of the input, display the content of the array both for the different tokens.

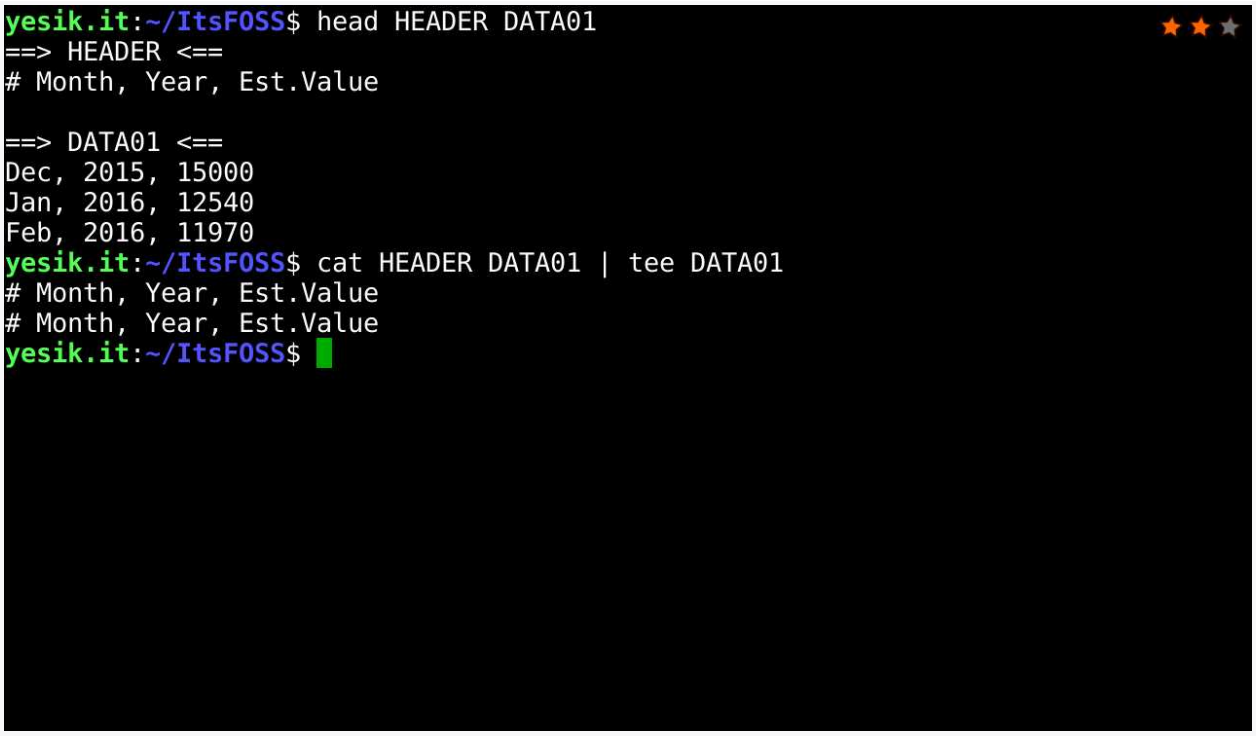
Please notice that "RED", "BLUE" and "GREEN" are now handled as character strings in the program (did you see the double quotes around them?) And that's not an issue for AWK since it does support associative arrays. And just like plain variables, uninitialized cells in an AWK associative array are assumed to be zero for mathematical operators.

As I explained it before, I made the choice of using AWK here. But Perl fans might have a different opinion of the subject. If you're one of them, why writing your own solution?

Challenge 12: Inserting the same header on top of several different files

What I tried to achieve?

This time I work with several data files and one header file. I just want to insert the content of the header file on top of each data file:




```
yesik.it:~/ItsFOSS$ head HEADER DATA01
==> HEADER <==
# Month, Year, Est.Value

==> DATA01 <==
Dec, 2015, 15000
Jan, 2016, 12540
Feb, 2016, 11970
```

For the sake of the demonstration, I only displayed the content of one file. But you may imagine I have many of them — too many for considering manual editing.

It thought I found a solution using the `cat` command:

```
yesik.it:~/ItsFOSS$ cat HEADER DATA01 | tee DATA01
# Month, Year, Est.Value
# Month, Year, Est.Value
```

Unfortunately, for some reason that solution didn't work: not only I've lost the data but my header appears twice.

As you can see, I *really* need your help here — both to explain to me what was going on and to help me in solving that issue.

The solution

What was the problem?

In a pipeline, all commands are launched in parallel. That means the `cat` command reading the DATA01 file *and* the `tee` command overwriting that same file are launched simultaneously.

This is really a **race condition.** On my system, `tee` had time to overwrite the destination file before `cat` had the opportunity to read it. To illustrate that, we can delay the commands and see the output is clearly dependent on the timing:

```
cat HEADER DATA01 | ( sleep 1; tee DATA01 )
# Month, Year, Est.Value
Dec, 2015, 15000
Jan, 2016, 12540
Feb, 2016, 11970
```

```
(sleep 1 ; cat HEADER DATA01 ) | tee DATA01
# Month, Year, Est.Value
```

I would have a similar issue (albeit deterministic this time) using the simpler:

```
cat HEADER DATA01 > DATA01
```

In that case, the shell *always* overwrites the destination file before launching the `cat` command. So the content of the file is lost long before `cat` had even the opportunity to read it.

How to fix that?

Obviously, no one would even consider using the sleep hack to solve that challenge in a real situation. But this is not an issue: as part of the standard POSIX tools, we have several commands at our disposal to insert the header on top of a file. Before that, let's take a look at the most basic solution.

The KISS solution

```
cat HEADER DATA01 > DATA01.NEW
mv -f DATA01.NEW DATA01
```

Do I really need to comment that? Well, while being rudimentary, this solution has a nice feature: since `mv` will use the system call `rename`, which itself is atomic in that sense other process referencing the DATA01 file will either see the old content or the new content—but neither a *half-written* content.

A somewhat similar solution, but avoiding to create a temporary file *visible* on the filesystem would obtain first a file descriptor to read from the original file before overwriting it:

```
exec 3<DATA01              # (1)
rm -f DATA01               # (2)
cat HEADER - <&3 >DATA01   # (3)
exec 3<&-                  # (4)
```

1. Open the file DATA1 for reading using the file descriptor 3;
2. Unlink the original file (i.e.: remove its directory entry, but not the data since the file is still open);
3. Use `cat` to read the header first, followed by a `stdin` read from file descriptor 3 and write to a new DATA01 file;
4. Close the file descriptor 3. This will effectively delete the old DATA01 content.

Please note this solution is no longer atomic in the sense described above. Anyways, Kudos to Adithya Kiran Gangu for having suggested me that solution!

Using sed

While encountering such kind of problems, my first idea is often to use `sed`. It is quite easy to insert a "header" after the first line using `sed`. Unfortunately, it's much more difficult to insert something *before* the first line. In fact, to achieve that, we will need a little bit of magic:

```
sed -i '1{
  r HEADER
  N
}' DATA01
```

To fully understand, you need to know the (r)ead command inserts the content of a file in the destination stream, but **only once the current line processing has ended**. That's why I used the (N)ext command: it will end the line 1 processing early (i.e.: before normal line output). So, when encountering that command, sed ends processing of line 1. Which triggers output of the content of the HEADER file. But the line 1 itself is not sent to the output. It is kept in the `sed` buffer.

Then `sed` reads the next line of input, append it to the buffer, and as we do not have any rule for line 2, process it as usual by sending its buffer to the output (remember at that stage, the buffer contains *both* line 1 *and* line 2).

This solution has a major drawback: it assumes *there is* a line 2. If the data file contains only one line, this will fail miserably.

Using ed or ex

We have very few occasions of using `ed` or its cousin `ex`. Both are line oriented editors. Their behavior is very similar to `vi` in that sense you load file into memory, and send commands to the editor to modify that file. The only difference here is we will script the commands instead of using them interactively.

```
ed DATA01 << .
0r HEADER
wq
.
ex -s DATA01 << .
0r HEADER
wq
.
```

This works great, but as we have to load the whole file into memory which could be an issue for very large files.

As always, those are probably only a subset of all possible solutions. For example, I bet you could find more solutions using AWK or Perl, for example!

Challenge 13: Converting text to uppercase

What I tried to achieve?

I have some text file containing twice the same sentence, once written in English and once written French:

```
yesik.it:~/ItsFOSS$ cat text
This text should be written in all
uppercase letters!

Ce text doit être affiché uniquement
en lettres majuscules!
```

I need to display the content of that file in all uppercases. Do you have any idea how I could perform that task? I mean without having to retype all the text…

The solution

The canonical tool to perform character substitution is `tr`--which stands for *transliteration*. The `tr` command takes two arguments: the source alphabet and the destination alphabet. And it performs a one-to-one mapping from the former to the latter.

```
yesik.it:~/ItsFOSS$ tr '[a-z]' '[A-Z]' < text
THIS TEXT SHOULD BE WRITTEN IN ALL
UPPERCASE LETTERS!

CE TEXT DOIT êTRE AFFICHé UNIQUEMENT
EN LETTRES MAJUSCULES!
```

This worked. But only for the US-ASCII letters. The accentuated letter, and any letter outside the US-ASCII range were not transliterated.

If you have some experience in using the regular expression[3], you may be tempted to use character classes instead of an explicit range:

[3] The `tr` command do *not* use regular expression. But it accepts a syntax similar to character ranges to specify the source and destination alphabets. For example, the syntax `[a-z]` is a shorthand for `abcdefghijklmnopqrstuvwxyz`

```
yesik.it:~/ItsFOSS$ tr '[[:lower:]]' '[[:upper:]]' < text
THIS TEXT SHOULD BE WRITTEN IN ALL
UPPERCASE LETTERS!

CE TEXT DOIT êTRE AFFICHé UNIQUEMENT
EN LETTRES MAJUSCULES!
```

Unfortunately this doesn't work better. A solution would be to explicitly add the mapping for the required characters:

```
yesik.it:~/ItsFOSS$ tr 'éê[a-z]' 'ÉÊ[A-Z]' < text
THIS TEXT SHOULD BE WRITTEN IN ALL
UPPERCASE LETTERS!

CE TEXT DOIT ÊTRE AFFICHÉ UNIQUEMENT
EN LETTRES MAJUSCULES!
```

This time, it works, but this is tedious. Especially if you don't know in advance the set of letters that will be present in the text file. Surprisingly enough, the Bash is much better suited than the `tr` command to change the character case:

```
yesik.it:~/ItsFOSS$ while read; do
                      echo ${REPLY^^}
                    done < text
THIS TEXT SHOULD BE WRITTEN IN ALL
UPPERCASE LETTERS!

CE TEXT DOIT ÊTRE AFFICHÉ UNIQUEMENT
EN LETTRES MAJUSCULES!
```

In that code, the outer `while` loop simply read the input file line by line. The Bash `read` command by default store the read value into the `REPLY` variable. And I just have to `echo` that variable, but using the special `${var^^}` notation that will expand to the content of a variable but with letters converted to uppercase.

As an exercise, I let you experiment with the `${var,,}` parameter expansion which works the same, but converting to lowercase rather than to uppercase.

Challenge 14: The Back in Time Function

What I tried to achieve?

This time, I want a shell function to display the date and time it was *two hours ago*. The function output must follow the `YYYY-MM-DD hh:mm` format.

I came to a solution using simple shell arithmetics:

```
minus-two-hours() {
    date -d "$1" +"%F %H:%M" | \
    {
        IFS=": " read -a COMP
        echo "${COMP[0]} $((10#${COMP[1]}-2)):${COMP[2]}"
    }
}
```

As you noticed, the function takes a date as an argument, parse it, and write back that date minus two hours. Unfortunately, the result is far from being satisfactory as the expected format is not always respected and I even have negative hours sometimes:

```
yesik.it:~/ItsFOSS$ minus-two-hours now
2016-11-22 20:55
yesik.it:~/ItsFOSS$ minus-two-hours "2016/11/21 05:27:18"
2016-11-21 3:27
yesik.it:~/ItsFOSS$ minus-two-hours "2016/11/21 01:10:42"
2016-11-21 -1:10
```

Could you help me finding a solution to obtain the desired result?

The solution

What was the problem?

Date arithmetic is much more complicated that one might expect. I strongly discourage you from taking the path I used in my initial attempt: never do date and time calculations by yourself. If you really need an argument to convince you, think for example about issues with daylight saving time.

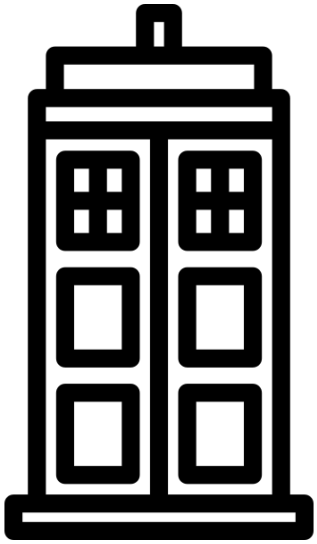
That being said, which options do we still have? Any decent programming language should have some facilities to deal with time specific issues. Here we are using the Bash, and we have to rely on the `date` tool for that purpose.

How to fix that?

Converting date to quantities

When faced with similar problems, the typical solution will be to convert the (human readable) date and time to some numeric *quantity*.

Usually, we convert dates to a number of seconds (or milliseconds) since some reference time. Having that numeric quantity, we can now use classic arithmetics to add or remove homogeneous quantities (say remove 7200s — that is 2×60×60s — to obtain the date it was two hours ago). Finally, using the same facilities as in the initial step, we can convert back the result to a date-time format.



In practice, in Unix-like systems, the reference date is usually 00:00:00 UTC on 1 January 1970—sometimes known as Unix Epoch[4]. And the `date` utility does provide:

- the %s specifier to convert a date to the number of seconds since the Epoch
- and the "@" symbol to specify an input date is expressed as a number of seconds since the Epoch (BSD will use the `-r` option for that purpose)

So here is a possible solution to my issue:

```
minus-two-hours() {
    # 1. Convert to number of seconds since Unix Epoch
    SRC=$(date -d "$1" +"%s")
    # 2. Remove two hours (expressed as a number of seconds)
    DST=$((SRC-2*60*60))
    # 3. Display the result using the required format
    date -d "@$DST" +"%F %H:%M"
}
```

Using the mighty powers of GNU date utils

The solution above is highly portable — even beyond the limits of shell programming.

But when using GNU `date` as we do on Linux for example, we have access to a whole world of subtleties to express the date. In particular, you can simply write that:

```
minus-two-hours() {
    date -d "$1 2 hours ago"   +"%F %H:%M"
}
```

Yes: *"2 hours ago"* is part of the date specification and is understood by GNU date as a way to say "remove two hours to the previous date".

As you can see, when portability is not a concern, it worth taking the time to explore a little bit your specific tools documentation as they may contain hidden gems!

[4] I always wondered if the *Apoc* character name in *Matrix* was related to that. In "French English" both are pronounced the same...

Level 3
Bash challenge

Challenge 15: My Bash don't know how to count. Again!

What I tried to achieve?

I just want to add the first and last integer stored in some data file. So I typed that at my Bash prompt:

```
yesik:001$ cat sample.data
1      2      3
yesik:002$ cut -d' ' -f1,3 sample.data | read X Z
yesik:003$ echo $((X+Z))
0
```

And the result displayed by my shell is 0 (zero). C'mon Bash: 1+3 is 4, not 0.
My bash don't know how to count ! Is it broken?

The solution

What was the problem?

Kudos to It's FOSS reader Riccardo Bernard for a great explanation of the "problem":

> *the line [2] is executed by starting the two sides of '|' in subprocesses. Therefore, read is not run in the shell, but in a subprocess and sets the variables in the subshell.*

Indeed, you can only change the content of the variable of the current shell. You cannot change the content of a parent shell's variable. So, since the data are read here in a sub-process, only the sub-process environment was changed. When line [3] is executed, we are back in the parent shell—where variables were *never* modified. Notice this is different in `zsh`.

How to fix that?

There is several way to fix the problem. All of them require we read the data in the same shell as the one that will perform calculations.

A simple solution would be to completely discard the `cut` command, and read the data using a simple redirection:

```
yesik:002$ read X Y Z < sample.data
```

If you really want to read the output of some external command though, you may use a process redirection instead:

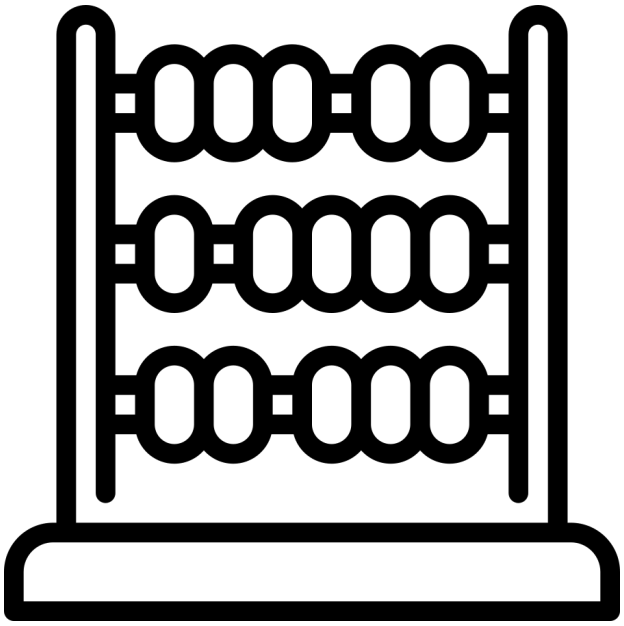
```
yesik:002$ read X Z < <(cut -d' ' -f1,3 sample.data)
```

...or a *here string*:

```
yesik:002$ read X Z <<< $(cut -d' ' -f1,3 sample.data)
```

Finally worth mentioning the Bash in not necessary the best solution for that kind of problem. A simple `AWK` program may do the work much better—especially since the implied loop allows you to handle with exactly the same program a file containing one row and a file containing thousands of them:

```
yesik:002$ awk '{ print $1+$3 }' sample.data
```



Challenge 16: Sending a file between two computers

What I tried to achieve?

I have two computers connected on the same local network. Their respective IPv4 addresses are 192.168.10.10 and 192.168.10.11.

I want to send a file from 192.168.10.10 to 192.168.10.11. How can I do that, knowing they do not have access to any kind of file sharing service and none of them has remote terminal facilities like `ssh` or `telnet` enabled.

All you may assume here is you can `ping` them from each other and you have access to a terminal running Bash on each host. You should not assume root access to any of the hosts.

Obviously, using a removable media like a USB pen drive would be a solution. But I've forgotten my pen drive at home. And it's late Sunday, so all shops are closed. Anyway, I *want* a network-based solution!

The solution

What was the problem?

The challenge here is to find a solution to open some kind of network communication channel between the hosts. And then copy a file from the source host to the destination host through that channel.

That's the kind of solution that will be used under the hood by a tool like `scp`. Or `rsync` in client/server mode. But I assumed in the challenge description those facilities were not available.

So, I have to find a way to create that communication channel by myself. The traditional tool for *ad-hoc* connections is `netcat` (or `nc` as it may be spelled in your distribution) There are chances `netcat` was installed as part of your standard Linux installation. So let's start using that program—unless your administrator removed it?

Introducing netcat

Note: if you don't have access to two different computers on the same network, you can still experiment by testing from two different terminals on your computer and using `localhost` instead of both IP addresses.

In order to use `netcat` to transfer data between two hosts, you may start two different instances of the tool. One on each host.

The first instance will open a port and wait for incoming requests. It is more straightforward to open the listener on the destination host. This is what I will do here:

```
# On the destination host
192.168.10.11$ netcat -l 1234
```

With that command, `netcat` will open the port 1234 (an arbitrary number) and wait for an incoming connection (option `-l` for "listen").

On the source host, we will use `netcat` too, but this time to connect to the remote host and port we've just opened:

```
# On the source host
192.168.10.10$ echo hello | netcat 192.168.10.11 1234
```

Press enter and the "hello" message will be displayed on the destination host.

What has happened here can be summarized in the following few steps:

1. The destination `netcat` has opened the port `1234` for incoming connections
2. The source `netcat` has established the connection to the destination host `192.168.10.11` port `1234`
3. The source `netcat` then started to read from its standard input and to send the data through the connection established previously
4. The destination `netcat` displayed on its standard output the data received from the connection.
5. At some point, the source `netcat` detected there was no more data to read and closed the connection (the exact behavior when encountering EOF is controlled by the `-N` option)
6. The destination `netcat` detected the connection has closed and has terminated too.

Once you know that, transferring a file can be achieved using some simple redirection:

```
# On the destination host
192.168.10.11$ netcat -l 1234 > my.file
```

```
# On the source host
192.168.10.10$ netcat 192.168.10.11 1234 < my.file
```

Replacing the client netcat by the Bash

Unfortunately, we can't solve that challenge using only the Bash since it is not able to open a server socket (the "listener" part of the communication channel). But the Bash may act as a client to connect to an existing server socket. So, server-side, we will use exactly the same command as above:

```
# On the destination host
192.168.10.11$ netcat -l 1234
```

But client-side, we no longer need netcat:

```
# On the source host
192.168.10.11$ echo hello > /dev/tcp/192.168.10.11/1234
```

In that case, the shell redirection to `/dev/tcp/…` will handle under the hood the connection to the remote host just like `netcat` did it previously.

By the way, no need to search in `/dev` for the `tcp` device. It does not really exist. It is simply a *magic filename* the Bash handle specifically. That means you can only use those special filenames in redirections.

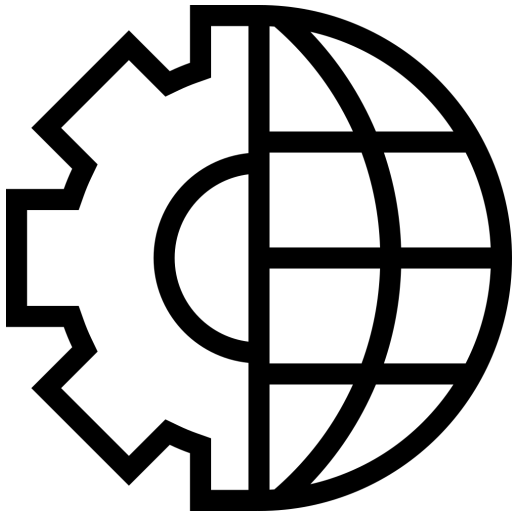
Despite some limitations, the Bash `/dev/tcp/…` pseudo-devices can be remarkably useful, for testing purposes or to diagnose network issues. To conclude on that topic, I can't resist in showing you some fun use case. I let up to you to do the necessary researches to understand the details:

```
# Open a connection with a remote web server
exec 3<> /dev/tcp/httpbin.org/80

# send a HTTP GET request
awk -v ORS=$'\r\n' '{print}' >&3 << EOT
GET /ip HTTP/1.1
host: httpbin.org
connection: close

EOT
# (the line above EOT must be empty. i.e. not even containing spaces)


# display the server's reply
cat <&3-
```

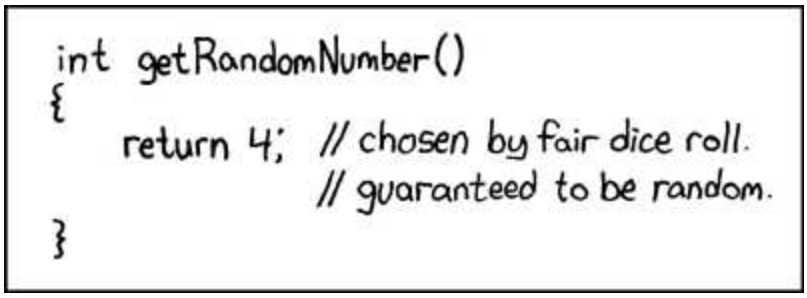


Challenge 17: Generate a fair dice roll

What I tried to achieve?

I'm starting to write a text adventure game in the purest tradition of the Colossal Cave Adventure or Zork. But written in Bash.

For that game, I need a way to generate fair 6-faced dice rolls. Of course, I do not need a "cryptographically strong pseudorandom number generator". But, on the other hand, the following solution in probably not satisfying either:



(from https://xkcd.com/221/ © the xkcd author some right reserved)

Could you help me in generating a random number in the 1-6 range, each result being equally probable?

The solution

What was the problem?

If you only had to know the `$RANDOM` Bash variable to produce pseudo-random numbers, this challenge would have probably been put into the "Level 2" section. But there is a subtle pitfall involved here. What was it? Let's first examine a *wrong* solution to discover that:

```
# This does NOT solve the challenge
getRandomNumber() {
  echo $((RANDOM%6+1));

}
```

I repeat: this does **not** solve the challenge, despite the fact by repeatedly calling that function you obtain pseudorandom numbers in the 1-6 range.

So what as wrong with that?

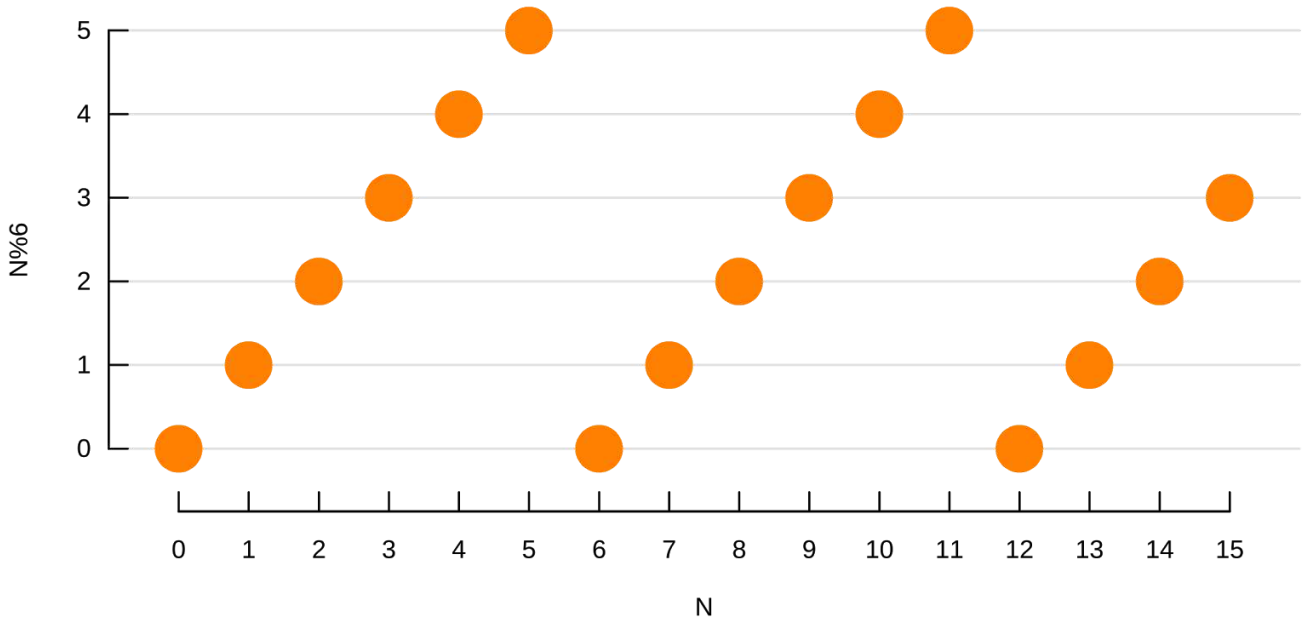
Simple: the outcome is **not** fair.

What was the problem, really?

The Bash random function returns a pseudo-random integer between 0 and 32767. That is 2^{15} possibilities. In the above function, I used the modulo operator (%) to map that range into the 0-5 range. But this mapping can only maintain equiprobability if the destination range size is an integer divisor of the original range size.

Simply said, in our case, since the cardinality of the original range is a power of two, you can only have equiprobable results if you map to another set whose size is a power of two itself. Something 6 isn't.

As this is a little bit hard to explain with words, here is a graphical representation of the mapping from the 0-15 range (2^4 items) to the 0-5 range:



It is quite obvious in the above graph there are *greater* chances of obtaining an outcome in the 0-3 range rather than in the 4-5 range. As an exercise I let you produce the same graph for N in the 0-32767 range if you want, you will discover a similar issue.

Knowing that, a solution you might end up with is the following one:

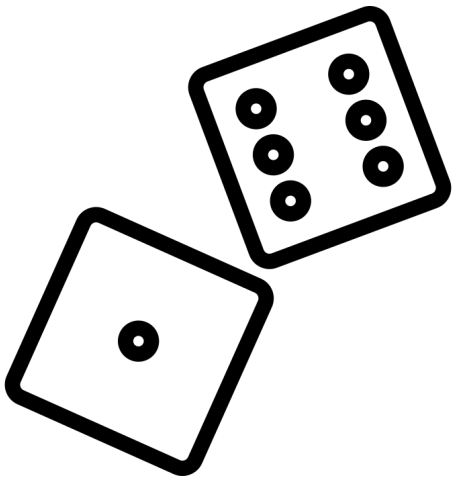
```
# This does NOT solve the challenge
getRandomNumber() {
  echo $((RANDOM%2+RANDOM%2+RANDOM%2+RANDOM%2+RANDOM%2+1));

}
```

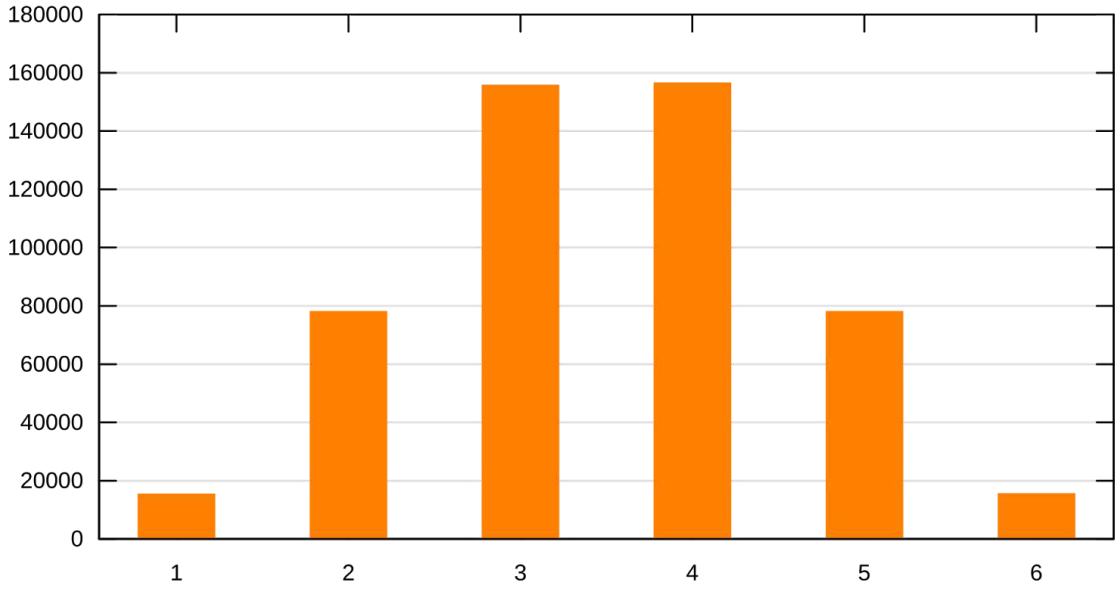
There is some logic here: I said if you used a power of two in the modulo operator you will have fair results. 2 is a power of two. And %2 will produce either 0 or 1. Adding five times a number between 0 and 1 will produce a result between 0 and 5. However:

```
# Gather some statistical data
for ((i=0; i < 500000; ++i))
do
  getRandomNumber
done | sort | uniq -c

  15531 1
  78134 2
 155880 3
 156604 4
  78184 5
  15667 6
```



It is quite obvious the result is not fair. In fact, you can recognize the typical shape of a normal distribution:



How to fix that?

The solution is to consider the modulo operator leads to a fair solution, but *only* in a subset of the original range. For example, if the original range is 0-15, and the destination range is 0-5, for input values in the 0-11 range, the modulo will produce an output *as fair as* the input. The problem lies for input values in the 12-15 range. This is *exactly* was is illustrated in my initial graph.

A trick is then simply to ignore values in the problematic range, and take another value if the random generator produced such value. All that leading to:

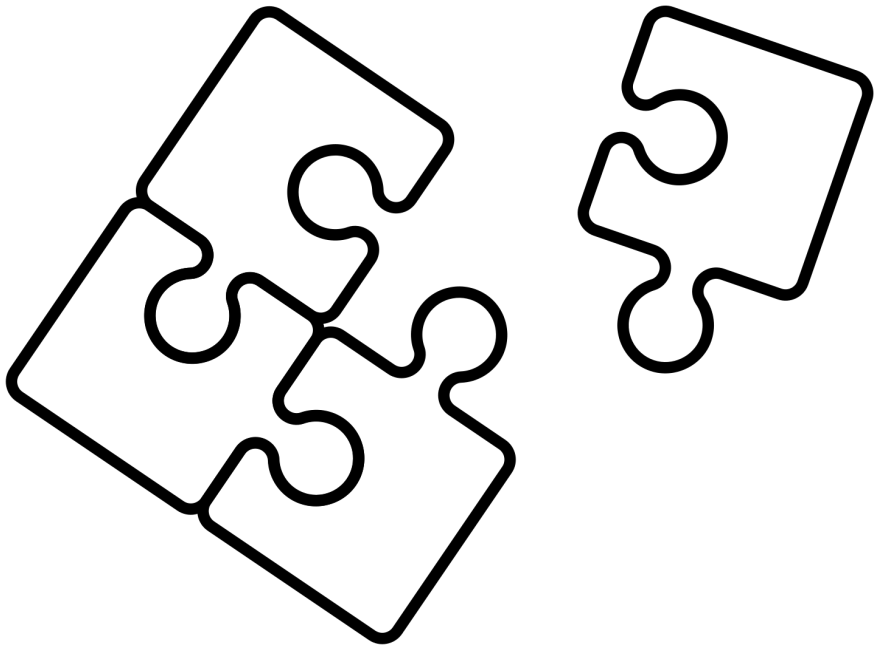
```
# This DO solve the challenge
getRandomNumber() {
  N=$RANDOM

  while ((N>=32768/6*6))
  do
    # N is in the problematic range. Take another value:
    N=$RANDOM
  done

  echo $((N%6+1));
}
```

Three remarks to conclude that challenge:

- `32768/6*6` is not the same as `32768*6/6`. Can you tell why?
- With this solution, there is an infinitesimal chance the `getRandomNumber` will *never* return.
- And of course, all that demonstration is based on the premise the `$RANDOM` pseudo-variable produces fair results. But does it *really*?



Afterword

If you reached that point you probably took all the challenges. Well, most of them. Or perhaps just a few?

In any cases, CON-GRA-TU-LA-TIONS! Solving even only one single challenge requires time and efforts. Since we learn more by trials and errors rather than by succeeding at the first try, there are chances you learned more than just the challenge's solution in that process. And hopefully, you had fun while doing that.

Wait a minute, you had fun. Don't you?

I couldn't conclude that book without a word for the readers that will find a solution for all the challenges. If that's your case, you are truly A-MA-ZING. Either because you already had an extended knowledge of the subject or because of your efforts, you are Bash gurus! And you should definitely consider sharing and spreading your wisdom. Why not by joining the Facebook Linux User Group (https://www.facebook.com/groups/822286747871993/) ?

Finally, you learned a lot of tricks and pitfalls about the Bash here. But this book can't replace a proper course on the topic. If you want to support your new knowledge by strong foundations, you may enjoy my Bash and Linux Command Line Course (https://yesik.it/BSH101/BASH-CHALLENGE-2017). By following that link you will get a special discount for the readers of this book!

Love. Enjoy. Learn.
Sylvain Leroux