UNIX 101

or "How to feel like a true hack3r"

Subject 1

Saturday 2nd February, 2019

Contents

1 Foreword

1.1 Notions seen in the tutorial

Welcome to subject-1! If you have finished tutorial-1 (and I hope you do, if you have not yet, please finish it now), you should know:

- What the syntax of a command is
- A few useful commands
- How to navigate in a filesystem
- How to create, edit and remove files
- What is a packet manager and how to use one
- How to create your own executable scripts

This is just enough to begin writing code in a local environment.

1.2 Objectives

The goal of this subject is to make you use the notions seen in the tutorial through coding exercises. Additionally, we will have a look over key concepts to development and UNIX in general.

Last but not least, you are going to be evaluated on the exercises of this document. You are not expected to finish them during the course: you have an additional two weeks to work on them before having to hand in your solutions.

2 Setup

2.1 A word on git

> *Git is a fast, scalable, distributed* ***revision control system*** *with and unusually rich command set that provides both high-level operations and full access to internals.*
>
> (`man git`)

To explain it in clearer terms, `git` is a tool that allows one to versionate documents. In software engineering (actually, in all of the fields of computer engineering), it is primordial to keep track of its work, especially when collaborating on the same projects.

- You have been working alone on a project for two months but one day you mess things up and delete some files? `git` can rollback the changes.
- You want to try to revamp your code but are not sure that you will be able to have a working version on time for the deadline? With `git`, you can keep a marker on the last working version and can rollback to it at any time.
- You are working in a team on the same project and want to share your work without conflicts? A `git` server can save your life.

Every serious company in the world uses a revision control system and fast, scalable and distributed attributes, `git` is by far the most popular.

2.2 Installation

We are not going to go into the details on `git`'s huge set of features as it would take some time, but we are going to at least use it to download resources for the exercises.

First of all, install `git` if you do not already have it somewhere in your system. You should know how to install a package by now, if not, read again section 4 of the tutorial!

To check out how it works, let us download the git repository[1] containing the exercises. The Github page is available here: `https://github.com/nikointhehood/unix-101`. In git language, we *clone* a repository. So, let's clone it! Click on "Clone or download" and retrieve the clone url : `https://github.com/nikointhehood/unix-101.git`. Then, invoke the command `git clone`.

```
$ls # The current directory is empty, let  us clone the repo unix-101
$git clone https://github.com/nikointhehood/unix-101.git
Cloning into 'unix-101'...
remote: Enumerating objects: 40, done.
remote: Counting objects: 100% (40/40), done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 40 (delta 4), reused 40 (delta 4), pack-reused 0
Unpacking objects: 100% (40/40), done.
Checking connectivity... done.
$ls
unix-101/ # There we have it!
$ls unix-101/lesson-1/subject/exercises
exercise-0/  exercise-1/  exercise-2/  exercise-3/  exercise-4/
   exercise-5/  exercise-6/  exercise-7/  tests_framework.py
```

[1] A git repository is a directory managed by git

Github is a service hosting git servers. There exist others (Gitlab, Bitbucket, even private instances of Gitlab/Github/Bitbucket for companies, etc) but Github is widely used for open-source projects. People from all over the world collaborate into writing software and sharing them to the world, for free. This kind of project is called *open-source*: anybody can participate and add new features to them by suggesting changes to the code. Actually, a lot of the most impressive computer engineering projects, extensively used by companies all over the world are open-source. The Linux kernel is findable on Github, for example[2].

3 Exercise 0: max (example)

Each exercise you are going to have to do is bundled with tests. Let us proceed with an example.

3.1 Goal

Script name: `biggest.py`
Online help: Check out the following links for more documentation
- Command line arguments: **Python3 official documentation**
- Type conversion: **Wikibooks examples**

You have to write a Python3 script that will take two integer arguments and print the bigger one, followed by a line feed[3].

If the two integers are equal, it should print `"The integers are equal"`, followed by a line feed

In case of an error, it should print `"Usage: ./biggest.py integer1 integer2"`, followed by a line feed.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

The error cases are:

- If there are less than two arguments
- If there are more than two arguments
- If the arguments are not integers

3.2 Example

[2]Here: `https://github.com/torvalds/linux`. Anybody can contribute, but that necessites some mad skillz.

[3]In UNIX, a line feed is represented by a `\n`. Some printing functions, like Python's `print()` automatically add one at the end of the printed message

```
$./biggest.py 7 121
121
$./biggest.py -1 31
31
$./biggest.py 8
Usage: ./biggest.py integer1 integer2
$./biggest.py 1 2 3 4 5
Usage: ./biggest.py integer1 integer2
$./biggest.py 0.7 11
Usage: ./biggest.py integer1 integer2
```

3.3 New notions

To complete this exercise, you need to know a few more things.

3.3.1 Arguments retrieval

UNIX programs are launched through the command line. As we have seen with the many commands we learnt in the tutorial, one can pass command line arguments to change their behavior.

So how does a program retrieve the user's arguments? Well it is system-dependant but usually, programs can access them using the `argv` variable. In Python, one must first import the `sys` package and can then look for the program arguments in the variable `sys.argv`.

`sys.argv` is a list containing the following items:

- At index 0, the name of the Python program
- At index 1, the first argument
- At index 2, the second argument
- At index 3, the third argument
- ...

Let us check out an example:

print_argv.py

```
1 #! /usr/bin/python3
2
3 import sys
4 print("argv is a list of", len(sys.argv), "elements:", sys.argv)
```

Now, let us see what happens with zero, one or multiple command line arguments:

```
$./print_argv.py
argv is a list of 1 elements: ['./print_argv.py']
$./print_argv.py hello
argv is a list of 2 elements: ['./print_argv.py', 'hello']
$./print_argv.py hello 47
argv is a list of 3 elements: ['./print_argv.py', 'hello', '47']
$./print_argv.py hello 47 again?
argv is a list of 4 elements: ['./print_argv.py', 'hello', '47', '
   again?']
$./print_argv.py hello 47 again? last_one333KK4..
argv is a list of 5 elements: ['./print_argv.py', 'hello', '47', '
   again?', 'last_one333KK4..']
```

3.3.2 Type casting

There is an important concept in programmation: type casting. It allows one to change the type of a variable. The possible type of casts differ from one language to another.

It is extra useful because some functions only take some kind of types as argument.

Let us see some examples.

example_cast.py

```
1  #! /usr/bin/python3
2
3  my_str = "37.3"
4  print(my_str, "->", type(my_str))
5  # Some casts are possible:
6  print(list(my_str), "->", type(list(my_str)))
7  print(float(my_str), "->", type(float(my_str)))
8
9  # Some are impossible:
10 print(int(my_str), "->", type(int(my_str)))
```

Executing it, we see that we can make a `str` become a `float` or a `list` but not an `int`:

```
37.3 -> <class 'str'>
```

```
['3', '7', '.', '3'] -> <class 'list'>
37.3 -> <class 'float'>
Traceback (most recent call last):
  File "./example_cast.py", line 10, in <module>
    print(int(my_str), "->", type(int(my_str)))
ValueError: invalid literal for int() with base 10: '37.3'
```

Actually, the error lies in the fact that the `37.3` can be interpreted by python as a float because it looks like one, but not as an `int` because `int` have round values.

Casting a `str` with a round value works:

example_cast_again.py

```
1 #! /usr/bin/python3
2
3 my_str = "37"
4 print(my_str, "->", type(my_str))
5 # list, float and int should work with this value:
6 print(list(my_str), "->", type(list(my_str)))
7 print(float(my_str), "->", type(float(my_str)))
8 print(int(my_str), "->", type(int(my_str)))
```

Let us try:

```
./example_cast_again.py
37 -> <class 'str'>
['3', '7'] -> <class 'list'>
37.0 -> <class 'float'>
37 -> <class 'int'>
```

3.4 Walkthrough

Now this part will not always be there but I am going to guide you on how to solve exercise 0:

3.4.1 Getting started

Navigate to the directory `exercise-0`:

```
$cd lesson-1/subject/exercises/exercise-0/
$ls
run_tests.py*
```

Here, there is only one executable script. It allows you to run the tests. Let us try:

```
$./run_tests.py
Test ['1', '2'] - FAILED
--> Your script encountered an error: File not found. Make sure your
   script is correctly named
```

Oh, an error. Well, that makes sense, we did not provide our solution to the exercise. So, let us solve it then.

3.4.2 Working on the exercise

Let's create a new file named `biggest.py` and write our solution.

biggest.py

```
1  #! /usr/bin/python3
2  import sys # This import allows us to interact with the command line
      arguments
3
4  # First, we declare a function which will do the comparison job
5  def biggest(int1, int2):
6      if int1 > int2:
7          print(int1)
8      elif int2 > int1:
9          print(int2)
10     else:
11         print("The integers are equal")
12
13 # Let's get the command line arguments
14 first_int = int(sys.argv[1])
15 second_int = int(sys.argv[2])
16
17 # Now, call the function with the arguments previously retrieved
18 biggest(first_int, second_int)
```

Now that the code seems good, let us test it manually.

```
$chmod +x biggest.py
$./biggest.py 20 3
20
$./biggest.py 7 11
11
$./biggest.py -1 7
7
```

It looks fine, but is that enough? Let us launch the tests.

```
$./run_tests.py
Test ['1', '2'] - PASSED
Test ['1123123', '2'] - PASSED
Test ['-17', '13'] - PASSED
Test ['0.3', '13'] - FAILED
--> Your script encountered an error:
Traceback (most recent call last):
  File "./biggest.py", line 14, in <module>
    first_int = int(sys.argv[1])
ValueError: invalid literal for int() with base 10: '0.3'
```

It seems that our script is not doing well with edge cases. This test just demonstrated that floating (e.g 1.7 or -3.1) numbers generate a `ValueError` in our code. The mission now is to handle all of the cases that we can think of and fix our script little by little.

To handle this error case, we simply have to check that the casts of the arguments to `int` do not raise a `ValueError`:

biggest.py

```
1 #! /usr/bin/python3
2 import sys # This import allows us to access argv, the list
     containing the command line arguments
3
4 # First, we declare a function which will do the comparison job
5 def biggest(int1, int2):
6     if int1 > int2:
7         print(int1)
```

```
8       elif int2 > int1:
9           print(int2)
10      else:
11          print("The integers are equal")
12
13  # Let's try to get the command line arguments
14  try:
15      first_int = int(sys.argv[1])
16      second_int = int(sys.argv[2])
17      # Now, call the function with the arguments previously retrieved
18      biggest(first_int, second_int)
19  except ValueError:
20      print("Usage: ./biggest.py integer1 integer2")
```

Running the tests again, we encounter another error on the number of arguments: this can be corrected by checking first that we are only given two additional arguments (the first one being the script filename, `biggest.py`).

biggest.py

```
1   #! /usr/bin/python3
2   import sys # This import allows us to access argv, the list
       containing the command line arguments
3
4   # First, we declare a function which will do the comparison job
5   def biggest(int1, int2):
6       if int1 > int2:
7           print(int1)
8       elif int2 > int1:
9           print(int2)
10      else:
11          print("The integers are equal")
12
13  # Let's try to get the command line arguments
14  try:
15      if len(sys.argv) != 3: # Actually, the target value is 3 because
           the first element of argv is always the name of the script
16          print("Usage: ./biggest.py integer1 integer2")
17      else:
18          first_int = int(sys.argv[1])
```

```
19          second_int = int(sys.argv[2])
20          # Now, call the function with the arguments previously
               retrieved
21          biggest(first_int, second_int)
22  except ValueError:
23      print("Usage: ./biggest.py integer1 integer2")
```

Running the tests one last time will only display green success messages:

```
./run_tests.py
Test ['1', '2'] - PASSED
Test ['1123123', '2'] - PASSED
Test ['-17', '13'] - PASSED
Test ['0.3', '13'] - PASSED
Test ['13'] - PASSED
Test ['0.1'] - PASSED
Test ['1', '2', '3', '-8'] - PASSED
Test ['mdr', '3'] - PASSED
Test ['0', '3'] - PASSED
Test ['0.7', '1'] - PASSED
Test ['19', '19'] - PASSED
Test ['0.7', '0.3'] - PASSED

All tests passed! Good job :). Make sure you did not forget any
   untested behavior
```

This exercise is completed. As asked in the exercise, we should now write a small text in a file named `README` explaining how our algorithm works and what were the pain points:

```
$ls
biggest.py*  README  run_tests.py*
$cat README
The goal of the exercise was to print the highest value of the
   command-line arguments.
I had difficulties with sanitizing the user input and I solved it
   using type casting and by checking the number of arguments in argv
   .
```

This kind of development mindset is the one you should follow for the next exercises.

4 Exercise 1: max advanced

4.1 Goal

Script name: `biggest_advanced.py`
Online help: Check out the following links for more documentation
- Command line arguments: **Python3 official documentation**
- Type conversion: **Wikibooks examples**

You have to write a Python3 script that will take an arbitrary number of integer arguments and print the bigger one, followed by a line feed.

In the case where all of the integers are the same, it should print one of them, followed by a line feed.

In case of an error, it should print:

`"Usage: ./biggest_advanced.py integer1 integer2 [integer3]..."`, followed by a line feed.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

The error cases are:

- If there are less than two arguments
- If the arguments are not integers

4.2 Example

```
$./biggest_advanced.py 3 17
17
./biggest_advanced.py 3 17 141
141
./biggest_advanced.py -1 77 -31 55
77
./biggest_advanced.py 3
Usage: ./biggest_advanced.py integer1 integer2 [integer3]...
./biggest_advanced.py 27 27 27
27
```

5 Exercise 2: Long long words

5.1 Goal

Script name: `longest_words.py`
Online help: Check out the following links for more documentation
- `str.join()`: **Python3 official documentation**

You have to write a Python3 script that will take an arbitrary number of `str` arguments and print the longest ones, separated by a comma, in the order in which they appeared, followed by a line feed.

In case of an error, it should print:

`"Usage: ./longest_word.py arg1 [arg2]..."`, followed by a line feed.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

The error cases are:

- If there are no argument

5.2 Example

```
$./longest_words.py bonjour ahbon
bonjour
$./longest_words.py abc def ghi
abc,def,ghi
$./longest_words.py baobab bonnet arbre
baobab,bonnet
$./longest_words.py
Usage: ./longest_words.py arg1 [arg2]...
$./longest_words.py q w e r t y
q,w,e,r,t,y
```

6 Exercise 3: my_range

6.1 Goal

Script name: `my_range.py`
Online help: Check out the following links for more documentation
- `str.join()`: **Python3 official documentation**

- `range()`: **Python3 official documentation**

You have to write a Python3 script that will take one to three integer arguments as input. It should then print the sequence of integers python's `range` function would generate, separated by spaces and followed by a line feed.

In case of an error, it should print:

`"Usage: ./my_range.py int1 [int2] [int3]"`, followed by a line feed.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

The error cases are:

- If there are no argument
- If there are more than three arguments
- If the arguments are not integers

Note: This is not a trivial exercise. Solve it little by little. First, simple cases, then more complicated cases and finally, edge cases.

6.2 Example

```
$./my_range.py 1 10
1 2 3 4 5 6 7 8 9
$./my_range.py 1 10 1
1 2 3 4 5 6 7 8 9
$./my_range.py 1 10 2
1 3 5 7 9
4./my_range.py 10 1 -1
10 9 8 7 6 5 4 3 2
$./my_range.py 10 1

$./my_range.py aaa
Usage: ./my_range.py int1 [int2] [int3]
```

7 Exercise 4: print_first_line

7.1 Goal

Script name: `print_first_line.py`

Online help: Check out the following links for more documentation
- `open()` prototype: **Python3 official documentation**
- File object functions: **Python3 official documentation**
- List of OS exceptions: **Python3 official documentation**

You have to write a Python3 script that will take one string argument as input. The string argument should be the filepath to a file.

Your script should read the file given as argument and print its first line, followed by a line feed. For this exercise, you can assume that the file will only contain ASCII characters.

- If the current user does not have sufficient permissions to read the file, print `"Unsufficient permissions"`, followed by a line feed.
- If the file does not exist, print `"The file does not exist"`, followed by a line feed.
- If another error is raised, print `"Unexpected error"`, followed by a line feed.
- If the number of arguments is different than 1, print the following usage, followed by a line feed: `"Usage: ./print_first_line.py filepath"`.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

7.2 Example

```
$./print_first_line.py
Usage: ./print_first_line.py filepath
$./print_first_line.py /etc/hosts aaa
Usage: ./print_first_line.py filepath
$./print_first_line.py /etc/hosts
127.0.0.1       localhost

$./print_first_line.py /etc/shadow
Unsufficient permissions
$./print_first_line.py ../tests_framework.py
from subprocess import run, PIPE, CalledProcessError, TimeoutExpired

$./print_first_line.py resources/no_line_feed.txt
This is a file which does not end with a line feed.
$./print_first_line.py resources/
Unexpected error
```

7.3 New notions

To complete this exercise, you need to know a few more things.

7.3.1 Files in UNIX

In UNIX, it is said that *Everything is a file*[4].

In fact, everything, from the status of the light of your "Caps lock" key to the configuration of your applications is either in a file, represented by a file of editable through a file or somewhere in your file system. By editing the right value in the right file, you can modify the light of your "Caps lock" key, for example! What is more, many applications keep their configuration in a file (often in `/etc`) and read it when booting.

Knowing how to open, read and edit files programmatically is thus extremely important.

7.3.2 File descriptor

To be able to read or edit a file, one must first open it. Python provides an easy-to-use function to do that: **open()**.

In UNIX, the function `open` usually returns a *file descriptor*. A *file descriptor* is simply an integer linked to a file allowing one to interact with it. Actually, you are lucky. Python is kinder than the raw system calls: its `open()` functions returns a *file object*, which is basically a *file descriptor* bundled with useful functions that you can invoke.

File objects allow one to read or write to their corresponding file descriptor (and thus, to their corresponding file): `file_object.readlines()` or `file_object.write()` are perfect examples. The many functions allowing one to manipulate a file descriptor are findable here: `https://docs.python.org/3.5/library/io.html#i-o-base-classes`.

Let's see an example:

test_readlines.py

```
1 #! /usr/bin/python3
2
3 # Open the file
4 fo = open("/etc/hostname")
5 # Read from the file object
6 lines = fo.readlines()
```

[4] A view closer to reality would be to say that *Everything is represented as a file in your filesystem*

```
7  # Do not forget to close the file! Although Python will take care of
       it if you forget in most cases, it is generally bad to keep file
       descriptors hanging for no reason
8  fo.close()
9
10 # Now, print what we have read
11 print(lines)
```

Launching it, we obtain:

```
$./test_readlines.py
['nicolas-laptop\n']
```

7.3.3 Handling files: flags

Looking at the prototype of `open()`[5], you may have noticed an optional `mode` argument, defaulting to `"r"`.

This arguments indicates for what usage the file should be open. For example, if you need to read from a file, you should use the default mode `"r"`. If you want write to a file, you should use `"w"`. If you want to read raw `bytes` from a file, you should combine the read flag and the binary mode flag (`"rb"`). As you can see, you can combine them.

There are other flags, read what they do. You will have to use them to solve the exercises.

7.3.4 Handling files: exceptions

The example we wrote worked like a charm for the file `/etc/hostname`. However, life is not always so good. There are many many (many!) reasons why your operating system would not let you peek into a file or write into a file.

If `open` or the functions bundled with a file object fail because of your system, Python will raise an `OSError`.

For example, if you do not have write permissions to a file but try to open it with the "w" flag, Python will raise a `PermissionError`. We say that `PermissionError` is a subclass of `OSError`: it is catchable using either `except PermissionError` or `except OSError`. The complete hierarchy of Python built-in exceptions is avaible at **this link**.

[5] **click here!**

Let's see an example:

restricted_write.py

```
#! /usr/bin/python3

# Open the file
fo = open("/tmp/restricted_file.txt", "w")
# Write to the file object
fo.write("Hello")
# Close the file
fo.close()
```

Launching it, we obtain:

```
$cat /tmp/restricted_file.txt
Yaaa
$chmod -w /tmp/restricted_file.txt
$ls -lh /tmp/restricted_file.txt
-r--r--r-- 1 nicolas nicolas 5 janv. 15 23:11 /tmp/restricted_file.
   txt
$./restricted_write.py
Traceback (most recent call last):
  File "./restricted_write.py", line 3, in <module>
    fo = open("/tmp/restricted_file.txt", "w")
PermissionError: [Errno 13] Permission denied: '/tmp/restricted_file.
   txt'
```

A simple way to catch this kind of issue is to wrap the dangerous calls around a simple `try/except`:

restricted_write_safer.py

```
#! /usr/bin/python3

# Open the file
try:
    fo = open("/tmp/restricted_file.txt", "w")
    # Write to the file object
    fo.write("Hello")
    # Close the file
```

```
9       fo.close()
10  except PermissionError:
11      print("Could not open file /tmp/restricted_file.txt: Permission
            denied!")
```

The exception is nicely catched:

```
$./restricted_write_safer.py
Could not open file /tmp/restricted_file.txt: Permission denied!
```

Additionally, when one tries to perform non-allowed file object actions (like reading from a file opened in write mode), Python will raise a `io.UnsupportedOperation` (`https://docs.python.org/3.5/library/io.html#io.UnsupportedOperation`).

7.3.5 Special trick to close files

Another way to open and close a file is to use a *context manager*. It is done using the `with` keyword:

context_manager_read.py

```
1  #! /usr/bin/python3
2
3  # Open the file
4  with open("/tmp/restricted_file.txt", "r") as fo:
5      print(fo.read(10)) # Print the first 10 characters of the file
6  # When exiting the indented block, fo.close() is automatically called
```

It allows one to not bother about closing files. Also, it has another utility: if an exception is raised inside the block, the file will still be properly closed by the context manager.

Last but not least, the syntax is way clearer than the standard one. You are strongly advised to use the `with` keyword, as the fact that files you open are properly closed will be tested.

8 Exercise 5: print_first_line_better

8.1 Goal

Script name: `print_first_line_better.py`

Online help: Check out the following links for more documentation
– `print()`'s prototype: **Python3 official documentation**

Because by default `print()` automatically inserts a line feed, if a file ends with a line feed too, we have a big interval in our terminal.

You have to write a Python3 script that will do the exact same thing as the previous one, except that it should not add a line feed after printing the first line of a file.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

8.2 Example

```
$./print_first_line_better.py
Usage: ./print_first_line.py filepath
$./print_first_line_better.py /etc/hosts aaa
Usage: ./print_first_line.py filepath
$./print_first_line_better.py /etc/hosts
127.0.0.1       localhost
$./print_first_line_better.py /etc/shadow
Unsufficient permissions
$./print_first_line_better.py ../tests_framework.py
from subprocess import run, PIPE, CalledProcessError, TimeoutExpired
$./print_first_line_better.py resources/no_line_feed.txt
This is a file which does not end with a line feed.$./
   print_first_line_better.py resources/ # ^ Notice the difference?
Unexpected error
```

9 Exercise 6: hello

9.1 Goal

Script name: `hello.py`

You have to write a Python3 script that will take one string argument as input. The string argument should be the filepath to a file.

Your script should append the string `hello` to the file given as argument.

– If the current user does not have sufficient permissions to write to the file, print `"Unsufficient permissions"`, followed by a line feed.

- If the file does not exist, the file should be created and contain `hello`.
- If the file is located in a folder which does not exist, print `"The file does not exist"`, followed by a line feed.
- If another error is raised, print `"Unexpected error"`, followed by a line feed.
- If the number of arguments is different than 1, print the following usage, followed by a line feed: `"Usage: ./hello.py filepath"`.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

9.2 Example

```
$./hello.py
Usage: ./hello.py filepath
$./hello.py /etc/hosts aaa
Usage: ./hello.py filepath
$cat mdr.txt
cat: mdr: No such file or directory
$./hello.py mdr.txt
$cat mdr.txt
hello$./hello.py mdr.txt
$cat mdr.txt
hellohello$./hello.py /etc/protocols
Unsufficient permissions
$./hello.py /tmp
Unexpected error
```

10 Exercise 7: print_first_bytes

10.1 Goal

Script name: `print_first_bytes.py`

You have to write a Python3 script that will take two string argument and an optional integer argument as input. The string arguments should be the filepath to files. The first one should exist and the second one can not exist.

Your script should copy the bytes of the first file to the second one. If a third integer argument is given, you should only copy the first `x` bytes of the source file.

- If the current user does not have sufficient permissions to access to a file, print `"Unsufficient permissions"`, followed by a line feed.
- If the destination file does not exist, it should be created.
- If the destination file already exists, it should be replaced.
- If another error is raised, print `"Unexpected error"`, followed by a line feed.
- If there are more than 3 arguments or if the third argument is not an integer or if there is less than 2 arguments, print the following usage, followed by a line feed:
  `"Usage: ./print_first_bytes.py src_filepath dst_filepath [int]"`.

Also, you are asked to provide a small file named `README` explaining how you solved the exercise.

10.2 Example

```
$./print_first_bytes.py /bin/ls test 4
$cat test
ELF$./print_first_bytes.py /bin/ls my_own_ls
$chmod +x my_own_ls
$./my_own_ls
my_own_ls  print_first_bytes.py  resources  run_tests.py
$./print_first_bytes.py /bin/ls test abcd
Usage: ./print_first_bytes.py src_filepath dst_filepath [int]
$./print_first_bytes.py /bin/ls test 4 yoo
Usage: ./print_first_bytes.py src_filepath dst_filepath [int]
```

11 Evaluation

11.1 Instructions

You have to send a tarball (a `.tar`, not a `.zip` nor a `.rar`) of your work before Wednesday 6th February, 2019, 23h59 at nikointhehood@gmail.com.

The subject and the content of your email are not important but your tarball should follow the format `firstname_lastname.tar.bz2` (of course, you have to replace `firstname` and `lastname` by yours).

The command to work with tarballs is `tar`. To create one, use the following command:

```
$ls
subject/ tutorial/
```

```
# c stands for create, v stands for verbose, f allows one to input a
   custom name for the archive and j indicates the usage of bzip2
$tar cvfj firstname_lastname.tar.bz2 subject/
(...)
$ls
subject/ tutorial/ firstname_lastname.tar.bz2
```

To verify that your archive contains everything you want, copy it somewhere else and decompress it:

```
$ls
firstname_lastname.tar.bz2  subject/  tutorial/
$mkdir /tmp/test
$cp firstname_lastname.tar.bz2 /tmp/test
$cd /tmp/test
# x stands for eXtract, v for verbose and f allows one to indicate
   which file tar should work with
$tar xvf firstname_lastname.tar.bz2
# You should see what I will obtain when extracting your archive
$tree --charset=ascii
.
|-- firstname_lastname.tar.bz2
`-- subject
    |-- exercises
    |   |-- exercise-0
    |   |   |-- biggest.py
    |   |   |-- README
    |   |   `-- run_tests.py
# (...)
    |   |-- exercise-7
    |   |   |-- print_first_bytes.py
    |   |   |-- README
    |   |   |-- resources
    |   |   |   |-- hosts
    |   |   |   |-- ls_only_4_original
    |   |   |   |-- ls_only_4_target
    |   |   |   |-- ls_only_50_original
    |   |   |   |-- ls_only_50_target
    |   |   |   |-- ls_original
    |   |   |   |-- ls_target
    |   |   |   `-- no_line_feed.txt
```

```
    |   |   `-- run_tests.py
    |   `-- tests_framework.py
    `-- subject-1.tex

14 directories, 49 files
```

11.2 A word on cheating

You have to remember that you should be studying for your own good. Cheating will not bring you any good in the long term; it is fine not to be able to finish every exercise of the subject, your main goal is to train and learn things.

Any form of cheating will immediately bring your grade down to 0. Additionally, your main teacher will be taken notice of that.

A particular attention will be given to your `README` files.

Note: Changing the name of some variables will of course not trick the anti-cheat engine :)