Program Arguments, Processes and Regular Expressions

Solve the following exercises and upload your solutions to Moodle until the specified due date. Make sure to use the *exact filenames* that are specified for each individual exercise. Unless explicitly stated otherwise, you can assume correct user input and correct arguments.

Exercise 1 – Submission: `ex1.py` 25 Points

Using the modules `argparse` and `re`, create a program that reads in a file, extracts user-specified patterns and writes pattern matches to output files. The program has the the following (keyword) command line arguments:

- Long form: `input_file`; short form: `i`; type: `str`; required

  Specifies the path of the input file from which patterns should be extracted. If it is not a file, a `ValueError` must be raised.

- Long form: `patterns`; short form: `p`; type: `list` with ≥ 1 `str` items; required

  Specifies a non-empty list of string patterns that are regular expressions. For each pattern, the `input_file` is parsed, and every full (e.g., in case of groups) pattern match is extracted to a separate file with the name `input_file_i.txt`, where `input_file` is the provided input file (see above) and `i` is the `i`-th pattern in the list. Each file must have the following content:

  ```
  regex: <pattern>
  <match_1><sep><match_2><sep>...<match_n>
  ```

  where `pattern` is the corresponding regex pattern followed by a line break, `<match_i>` is a found pattern match and `<sep>` is a separator string (see argument `separator` below).

- Long form: `separator`; short form: `s`; type: `str`; optional

  The string to use for separating extracted pattern matches. If not given, the default is the line break character `"\n"`.

- Long form: `encoding`; short form: `e`; type: `str`; optional

  The character encoding to use for accessing all files (input reading and output writing). If not given, the default is `"utf-8"`.

Example input file content (`ex1_data.txt`):

```
12 w 21  d23g780nb deed e2 21.87
43 91 - . 222 mftg 21 bx .1 3 g d e 6 de ddd32 3412
0.3 0 0. 0 0 1

70

n 12 1    9    m1 1m 445
x 100
```

Example command line arguments (assuming the current working directory contains `ex1_data.txt`):

```
-i
ex1_data.txt
--patterns
d(..d)
\d{3}
ab12.c
"d[^ ]+"
```

The following files will be created for the above example (in the same directory as `ex1_data.txt`):

- `ex1_data.txt_0.txt` with content:

  ```
  regex: d(..d)
  deed
  de d
  ```

- `ex1_data.txt_1.txt` with content:

  ```
  regex: \d{3}
  780
  222
  341
  445
  100
  ```

- `ex1_data.txt_2.txt` with content (no matches found):

  ```
  regex: ab12.c
  ```

- `ex1_data.txt_3.txt` with content:

  ```
  regex: d[^ ]+
  d23g780nb
  deed
  de
  ddd32
  ```

Exercise 2 – Submission: `ex2.py` 25 Points

Write a function `extract_matr_ids(text: str) -> list[int]` that extracts matriculation IDs from a given string `text` and returns the IDs as a list of integers. IDs are defined in the following two possible ways:

- Definition 1: An ID must have exactly 8 digits, fewer or more digits are not allowed. Furthermore, no leading or trailing letters (`"a"-"z"`, `"A-Z"`) are allowed. Exception: An ID can potentially have a leading `"k"` or `"K"`.
- Definition 2: An ID can also be specified via `"id=<number>"`, where `<number>` is a number with 4 up to 8 digits, fewer or more digits are not allowed. Except further digits (which would potentially violate the max-8-digits rule), any character is allowed after `<number>`.

Use the `re` module and regular expressions to solve this task. You are *not allowed* to use any manual string handling such as splitting, replacing, substring checking, converting to lowercase, etc.

Example program execution:

```
t = """This is an ID 01234567 this12345678 is not (leading "s"), nei12345678ther
(leading "i" and trailing "t" letters) is 12345678this (trailing "t"). but this
k22222222 is and also this K33333333, but again, k12345678is not valid (trailing
"i") and neither is K123456789 (too many digits) or 1234 (too few digits). Another
valid is ID id=4444 or id=5555555. Invalid examples are id=d1234 (leading "d") or
id=12 (too few digits) or id=x12345678 (leading "x") or id=123456789 (too many
digits) or ID=1234 ("ID" is not equal to "id"). id=11111xyz is valid again."""
print(extract_matr_ids(t))
```

Example output:

```
[1234567, 22222222, 33333333, 4444, 5555555, 11111]
```

Hints:

- You can use https://regex101.com/ to quickly test your regular expressions.
- It is possible to create a single regular expression that can extract IDs as specified above (the only remaining thing is the conversion to integers and returning them in a list).

Exercise 3 – Submission: `ex3.py` 25 Points

Write a program where a user can interactively execute arbitrary programs with arguments. The program should work as follows (see the example program execution below for more details on how the command line interface (CLI) must look like):

- First, the user must enter the program name to execute or press the *Enter* key (results in an empty input string) to exit. Use `"Enter program or press ENTER to exit: "` for the CLI.
- Then, the user can repeatedly enter arguments that will be passed to this program. The arguments must be collected in a list. The user can press the *Enter* key (results in an empty input string) once all arguments have been entered. Use `"Enter argument or press ENTER to skip: "` for the CLI.
- Lastly, the user can enter a character encoding which should be used to decode the program output. The user can press the *Enter* key (results in an empty input string) to not use any encoding. Use `"Enter encoding or press ENTER to skip: "` for the CLI.
- After the user input is finished, the specified program with the arguments (if any) must be executed using the module `subprocess`. If the user specified an encoding, it must be used to decode any potential program output (both error as well as standard output). To visualize this step in the CLI, print the strings:
  - `"Running program '<program>' without any arguments and no encoding"` in case there are no arguments for the specified program `<program>` and no encoding was specified.
  - `"Running program '<program>' without any arguments and encoding '<encoding>'"` in case there are no arguments for the specified program `<program>` and some encoding `<encoding>` was specified.
  - `"Running program '<program>' with arguments <args> and no encoding"` in case there are some arguments `<args>` for the specified program `<program>` and no encod-

ing was specified. `<args>` is simply the string representation of the list of arguments collected earlier (see example output below).

– `"Running program '<program>' with arguments <args> and encoding '<encoding>'"` in case there some arguments `<args>` for the specified program `<program>` and some encoding `<encoding>` was specified. `<args>` is simply the string representation of the list of arguments collected earlier (see example output below).

- If the program or the character encoding does not exist, catch the corresponding exceptions `FileNotFoundError` (program) and `LookupError` (encoding), and print an appropriate warning message: `"The specified program '<program>' could not be found"` or `"The specified encoding '<encoding>' could not be found\n"`, where `<program>` and `<encoding>` are the program name and encoding name, respectively. In all other cases, the specified program is executed normally.
- Once the specified program has finished, print its return/status code (`"The program finished with exit code <code>"`, where `<code>` is this return/status code). Also print (potentially decoded) output, but only if it is not empty: For non-empty error output, print `"The program produced the following error output:"` followed by the error output (on a new line). For non-empty standard output, print `"The program produced the following standard output:"` followed by the standard output (on a new line).

This procedure is repeated until the user presses the *Enter* key in the first step (when asked to input the program name).

Example program execution (output might vary due to OS differences):

```
Enter program or press ENTER to exit: pthon
Enter argument or press ENTER to skip:
Enter encoding or press ENTER to skip: uutf-8
Running program 'pthon' without any arguments and encoding 'uutf-8'
The specified encoding 'uutf-8' could not be found


Enter program or press ENTER to exit: pthon
Enter argument or press ENTER to skip:
Enter encoding or press ENTER to skip: utf-8
Running program 'pthon' without any arguments and encoding 'utf-8'
The specified program 'pthon' could not be found


Enter program or press ENTER to exit: python
Enter argument or press ENTER to skip: eex1.py
Enter argument or press ENTER to skip:
Enter encoding or press ENTER to skip:
Running program 'python' with arguments ['eex1.py'] and no encoding
The program finished with exit code 2
The program produced the following error output:
b"python: can't open file 'C:\\Repos\\python\\courses\\Programming-in-Python-I\\
    WS2022\\assignments\\a9\\code\\eex1.py': [Errno 2] No such file or directory\n"


Enter program or press ENTER to exit: python
Enter argument or press ENTER to skip: ex1.py
```

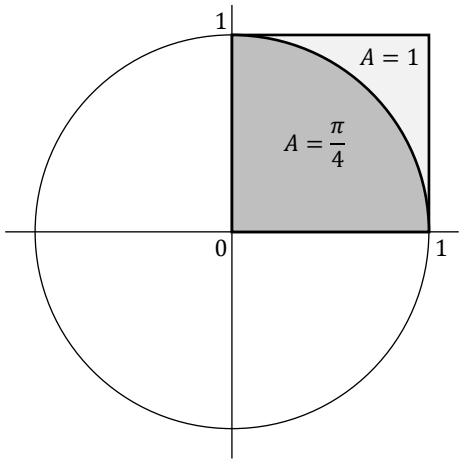
```
Enter argument or press ENTER to skip: -i
Enter argument or press ENTER to skip: some_dir
Enter argument or press ENTER to skip: -p
Enter argument or press ENTER to skip: some_pattern
Enter argument or press ENTER to skip:
Enter encoding or press ENTER to skip: utf-8
Running program 'python' with arguments ['ex1.py', '-i', 'some_dir', '-p', '
    some_pattern'] and encoding 'utf-8'
The program finished with exit code 1
The program produced the following error output:
Traceback (most recent call last):
  File "C:\Repos\python\courses\Programming-in-Python-I\WS2022\assignments\a9\code\
    ex1.py", line 19, in <module>
    raise ValueError(f"'{args.input_file}' is not a file")
ValueError: 'some_dir' is not a file


Enter program or press ENTER to exit: python
Enter argument or press ENTER to skip: ex1.py
Enter argument or press ENTER to skip: -i
Enter argument or press ENTER to skip: ex1_data.txt
Enter argument or press ENTER to skip: -p
Enter argument or press ENTER to skip: some_pattern
Enter argument or press ENTER to skip:
Enter encoding or press ENTER to skip: utf-8
Running program 'python' with arguments ['ex1.py', '-i', 'ex1_data.txt', '-p', '
    some_pattern'] and encoding 'utf-8'
The program finished with exit code 0


Enter program or press ENTER to exit:
```

Exercise 4 – Submission: `ex4.py` 25 Points

Write a program that can approximate the mathematical constant π based on the following idea: The area of a circle with radius $r = 1$ is π ($r^2 \cdot \pi = 1^2 \cdot \pi = \pi$), and the area of a quarter of this circle is thus $\frac{\pi}{4}$. This quarter circle can be fit into a square of size $r \times r = 1 \times 1$ with an area of 1, as shown in the following figure:



If we now sample a random 2D point in this square, then the probability that we hit the quarter circle is $\frac{\pi}{4}$ (given that the random sample was drawn from a uniform distribution). If we repeat this process N times and let C represent the number of circle hits, then the fraction $\frac{C}{N}$ will approximate this probability $\frac{\pi}{4}$. To get an approximation for π, all we have to do is multiplying this fraction by 4, since $4 \cdot \frac{C}{N} \approx 4 \cdot \frac{\pi}{4} = \pi$. If N is large enough, this should be a decent approximation.

Your program should work as follows:

- Create a function `get_quarter_circle_hits(n: int) -> int` that creates a random 2D point (x, y) and checks whether it hits the quarter circle as described above. Such a point hits the quarter circle if it fulfills the equation $x^2 + y^2 < 1$. This process is repeated `n` times, and the number of quarter circle hits is returned. To get appropriate random numbers for x and y that are drawn from a uniform distribution in the range $[0, 1)$, you can use the module `random` and the function `random.random()`.
- Create the main entry point of your script with `if __name__ == "__main__"`. There, set up a `multiprocessing.Pool` with `p` processes, where each process invokes the above function `get_quarter_circle_hits` with some `n`. Once all processes are finished, sum up the returned quarter circle hits, which will be the C from the approximation description above. N is the total number of quarter circle hit checks, i.e., the number of processes `p` multiplied by `n` (for example, with `p` = 2 processes and `n` = 100 checks each, the total number of checks is 200). The number of processes `p` and the number of quarter circle hit checks per process `n` must be specified via the following (keyword) command line arguments (`argparse` module):
  - Long form: `processes`; short form: `p`; type: `int`; optional; default: 1
  - Long form = short form: `n`; type: `int`; optional; default: 1000
- Print the following output statistics at the end of the program:

  ```
  <p> processes with <n> checks each
  total tries: <N>
  total hits: <C>
  pi approximation: <pi_approximation>
  ```

  where `<p>` is the number of process `p`, `<n>` the number of quarter circle hit checks per process `n`, `<N>` and `<C>` are N and C from the approximation description above, and `<pi_approximation>` is $4 \cdot \frac{C}{N}$.

Example command line arguments:

```
-p
4
-n
1000000
```

Example output (might vary due to randomization):

```
4 processes with 1000000 checks each
total tries: 4000000
total hits: 3140946
pi approximation: 3.140946
```