Files

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` 20 Points

Write a function `read_numbers(path: str) -> list` that reads the file specified by `path` (you can assume that the file exists) and extracts all numbers (integers and floats), which are then returned as a list. The file content has the following format for each line:

```
TOKEN1 TOKEN2 ... TOKENn
```

where `TOKENi` represents a token, which can be an arbitrary string, and individual tokens can be separated by at least one whitespace character (e.g., a space or a line break). Lines can also be empty. If a token is an integer number, it must be converted to an integer object. If a token is a float number, it must be converted to a float object. All other cases are ignored/dropped.

Example 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 function call and result:

```
read_numbers("ex1_data.txt") =
    [12, 21, 21.87, 43, 91, 222, 21, 0.1, 3, 6, 3412, 0.3, 0, 0.0, 0, 0, 1,
     70, 12, 1, 9, 445, 100]
```

Exercise 2 – Submission: `ex2.py` 40 Points

In this exercise, you have to write two functions for serializing and deserializing dictionaries (given some user-specifiable character encoding, see default parameter `encoding`):

- `write_dict(d: dict, path: str, encoding: str = "utf-8")`

  This function is about serializing a given dictionary to a specified file (the inverse of `read_dict`). The dictionary only contains string keys and string values (no other data types or nested dictionaries). The implementation is completely up to you. You only have to consider that the *exact same* dictionary must be returned by `read_dict`, i.e., the following must be true:

  ```
  write_dict(some_dict, some_file)
  new_dict = read_dict(some_file)
  some_dict == new_dict  # This must evaluate to True
  ```

- `read_dict(path: str, encoding: str = "utf-8") -> dict`

  This function is about deserializing a dictionary from a specified file (the inverse of `write_dict`). The implementation is completely up to you. You only have to consider that the *exact same* dictionary must be returned as was initially passed to `write_dict` (see example code above).

You can assume correct arguments (e.g., the input dictionary does really only contain string keys and string values, or the file actually exists when trying to read it, or the file you want to read was previously created with `write_dict` rather than being some arbitrary file). You can add additional default parameters if you want. You are *not allowed* to use any modules like `pickle` or `json`.

Hints:

- While the dictionary is limited to string keys and string values, you still have to be very careful how you implement the de-/serialization. For example, simply writing a key in one line and the corresponding value in the following line does not work if the key/value contains a line break itself.
- Depending on your implementation, it might be a good idea to disable the universal newlines mode of the built-in `open` function.

Exercise 3 – Submission: `ex3.py` 20 Points

Write a function `get_abs_paths(root_path: str, ext_filter: str = None) -> list` that collects all files (not directories) starting at the specified root directory `root_path` (continuing in all subdirectories, recursively) and returns a sorted list of absolute file paths. If `root_path` is not a directory, a `ValueError` must be raised. If `ext_filter` is not `None`, only those files should be included that have this extension. In this case, it might happen that no files remain. If so, the function should return an empty list. `ext_filter` must start with a leading dot, e.g., `".txt"` or `".py"`, otherwise, a `ValueError` must be raised.

Hints:

- Use the `os` module to join paths independently of the operation system and its other functions to help you with relative/absolute paths and file/directory checking.
- You can use the `glob` module to search for files recursively.

Exercise 4 – Submission: `ex4.py` 20 Points

Write a generator function `chunks(path: str, size: int, **kwargs)` that yields data chunks of size `size` from the file specified by `path`. If `path` is not a file, a `ValueError` must be raised. If `size` is smaller than 1, a `ValueError` must be raised. `**kwargs` are (optional) keyword arguments that are passed to the built-in `open` function (which your function must use internally). Note that there can be smaller chunks than `size`, e.g., if the file is too short or if the file content length is not evenly divisible by `size`. You are *not allowed* to read the entire file content at once, i.e., make sure to also implement a chunked file reading operation.

Example function call (with file `ex1_data.txt`, see exercise above):

```
for i, c in enumerate(chunks("ex1_data.txt", 25, mode="rb")):
    print(f"Chunk {i} = {c}")
```

Example output:

```
Chunk 0 = b'12 w 21  d23g780nb deed e'
Chunk 1 = b'2 21.87\r\n43 91 - . 222 mf'
Chunk 2 = b'tg 21 bx .1 3 g d e 6 de '
Chunk 3 = b'ddd32 3412\r\n0.3 0 0. 0 0 '
Chunk 4 = b'1\r\n\r\n70\r\n\r\nn 12 1    9   '
Chunk 5 = b' m1 1m 445\r\nx 100\r\n'
```

Hints:

- Use the `os` module to check if `path` is a file.