Assignment 1: Functions Advanced

Solve the following exercises and upload your solutions to Moodle by the due date.

> **Important Information!**
>
> Please try to *exactly match the outputs* provided in the examples, also use the unit test provided in Moodle to check your solutions.
>
> Use the *exact filenames* specified for each exercise (the default suggestions from the heading). Your main code (example prints, etc.) should be guarded by `if __name__ == '__main__':`. Unless explicitly stated otherwise, you can assume correct user input and correct arguments.
>
> You may use **only standard libraries**, plus modules covered in the lecture. NumPy was done in Python 1 and can also be used.

Exercise 1 – Submission: `a1_ex1.py` 25 Points

You should write a function that takes a list of products in our company's catalog and sorts entries based on a given mode. The function signature is:

```
organize_catalog(products: list[Product], mode: str = "cheapest") -> list[Product]:
```

The function parameters are:

- `products`: a list of `Product` objects, where each object has the attributes
  - `name` (string)
  - `category` (string)
  - `price` (float)
  - `rating` (float)
  - `launch_date` (string)
  - `in_stock` (boolean)
- `mode`: a string that is used sort the products. It can be one of the following:
  - `"cheapest"`: sort by `price` in ascending order
  - `"category_then_price"`: sort by `category` in ascending order, then by `price` in ascending order
  - `"best_rated"`: sort by `rating` in descending order
  - `"newest"`: sort by `launch_date` in descending order
  - `"all"`: display all products sorted by name in ascending order
- The function should return a list of products sorted according to the given mode (not in-place, return a new list).

Consider the following points for the function:

- Only for mode `"all"`, all products should be shown. For the other modes, only the ones that are in stock should be considered. For this, you can, but do not have to, use a filter function with lambda.
- Use lambdas for the sorting.
- The function should raise a `ValueError` if an invalid mode is given. The error text should be `"Unknown mode: {mode}"`.

- Also implement the necessary Class `Product` to represent the products with the attributes shown above. Implement the `__init__` and `__str__` methods for the class. The string representation should be `"{name} ({category}, {rating}, {launch_date}) - {price}"`, so for example: `Game Console (Electronics, 4.5, 2022-09-20) - 299.00`

Example code:

```
if __name__ == "__main__":
    catalog = [
        Product(name="Fancy Headphones", category="Electronics",
                price=199.99, rating=4.7, launch_date="2022-11-05", in_stock=True),
        Product(name="Wireless Mouse", category="Electronics",
                price=29.99, rating=4.3, launch_date="2023-03-01", in_stock=True),
        Product(name="Hiking Backpack", category="Outdoors",
                price=59.50, rating=4.8, launch_date="2021-07-15", in_stock=False),
        Product(name="Novel: The Great Adventure", category="Books",
                price=15.99, rating=4.9, launch_date="2023-01-10", in_stock=True),
        Product(name="Game Console", category="Electronics",
                price=299.0, rating=4.5, launch_date="2022-09-20", in_stock=True),
    ]

    print("--- cheapest (in_stock only) ---")
    for p in organize_catalog(catalog, "cheapest"):
        print(p)

    print("\n--- best_rated (in_stock only) ---")
    for p in organize_catalog(catalog, "best_rated"):
        print(p)

    print("\n--- newest (in_stock only) ---")
    for p in organize_catalog(catalog, "newest"):
        print(p)

    print("\n--- category_then_price (in_stock only) ---")
    for p in organize_catalog(catalog, "category_then_price"):
        print(p)

    print("\n--- all (including out_of_stock), default sort by name ---")
    for p in organize_catalog(catalog, "all"):
        print(p)

    print("\n--- wrong mode (lowest_rated) ---")
    try:
        organize_catalog(catalog, "lowest_rated")
    except ValueError as e:
        print(e)
```

Example output:

```
--- cheapest (in_stock only) ---
Novel: The Great Adventure (Books, 4.9, 2023-01-10) - 15.99
Wireless Mouse (Electronics, 4.3, 2023-03-01) - 29.99
Fancy Headphones (Electronics, 4.7, 2022-11-05) - 199.99
Game Console (Electronics, 4.5, 2022-09-20) - 299.00

--- best_rated (in_stock only) ---
Novel: The Great Adventure (Books, 4.9, 2023-01-10) - 15.99
Fancy Headphones (Electronics, 4.7, 2022-11-05) - 199.99
Game Console (Electronics, 4.5, 2022-09-20) - 299.00
Wireless Mouse (Electronics, 4.3, 2023-03-01) - 29.99

--- newest (in_stock only) ---
Wireless Mouse (Electronics, 4.3, 2023-03-01) - 29.99
Novel: The Great Adventure (Books, 4.9, 2023-01-10) - 15.99
Fancy Headphones (Electronics, 4.7, 2022-11-05) - 199.99
Game Console (Electronics, 4.5, 2022-09-20) - 299.00

--- category_then_price (in_stock only) ---
Novel: The Great Adventure (Books, 4.9, 2023-01-10) - 15.99
Wireless Mouse (Electronics, 4.3, 2023-03-01) - 29.99
Fancy Headphones (Electronics, 4.7, 2022-11-05) - 199.99
Game Console (Electronics, 4.5, 2022-09-20) - 299.00

--- all (including out_of_stock), default sort by name ---
Fancy Headphones (Electronics, 4.7, 2022-11-05) - 199.99
Game Console (Electronics, 4.5, 2022-09-20) - 299.00
Hiking Backpack (Outdoors, 4.8, 2021-07-15) - 59.50
Novel: The Great Adventure (Books, 4.9, 2023-01-10) - 15.99
Wireless Mouse (Electronics, 4.3, 2023-03-01) - 29.99

--- wrong mode (lowest_rated) ---
Unknown mode: lowest_rated
```

Exercise 2 – Submission: a1_ex2.py 25 Points

Create a callable class RateLimiter that tracks how often it is called and enforces a limit of a certain number of calls within a specified time window (in seconds). If the limit is exceeded, it should return False, indicating that the call was ouside the time window. Otherwise, it should return True. The class should have the following methods:

- `__init__(max_calls: int, window_seconds: float)`: sets up an internal data structure to store timestamps of the last calls (make sure to call the attribute 'timestamps', otherwise the unittest doesn't work). Within `__init__`, you might want to use the `deque` data structure from the `collections` module to store the timestamps. It is quite convenient because it allows you to easily remove old timestamps (i.e. at the beginning of the list).
- `__call__()`: every time you call the instance, it should:
  1. Remove any timestamps that are older than `window_seconds`.
  2. Check if the number of recorded calls within the time window is still below `max_calls`.
  3. If yes, record the new call (add current timestamp) and return True.
  4. If no, return False.

Example code:

```python
if __name__ == "__main__":
    limiter = RateLimiter(3, 5.0)  # 3 calls allowed per 5 seconds
    for i in range(5):
        time.sleep(1)
        success = limiter()
        print(f"Call #{i+1} => {success}")
    for i in range(3): # 5 seconds have passed, calls are allowed again
        time.sleep(2)
        success = limiter()
        print(f"Call #{i+6} => {success}")
```

Example output:

```
Call #1 => True
Call #2 => True
Call #3 => True
Call #4 => False
Call #5 => False
Call #6 => True
Call #7 => True
Call #8 => True
```

Exercise 3 – Submission: `a1_ex3.py` 25 Points

Write a decorator `@file_cache(cache_dir: str = cache)` that stores the return values of a function in a cache txt file and puts the file in `cache_dir`. When the decorated function is called again with the same arguments, it should load the result from the cache file instead of recomputing it. Consider the following points:

- If the cache directory does not exist, create it.
- Create a file name for the text file, it should have the format `funcname_arg1_arg2_.txt` (for two arguments).
- If the function is called with new arguments, compute and store the result in the cache file.
- If the function is called with the same arguments again, return the cached result by reading the file and returning its content. Also print on the console
  `"[CACHE] Using cached result for {func.__name__} with args={args}"`
- Make sure that the decorated function keeps its name.
- Important: import the `time` module in your script, it is required by the unittest.

Example code:

```python
@file_cache(cache_dir="cache_txt")
def expensive_calculation(x: int, y: int) -> str:
    import time
    time.sleep(2) # Simulates a long computation
    return f"The result is {x * y}"

if __name__ == '__main__':
    print(expensive_calculation(10, 20))
    # Second call with the same argument: should return immediately from cache.
    print(expensive_calculation(10, 20))
    # New argument => computation happens again.
    print(expensive_calculation(5, 15))
```

Example output:

```
The result is 200
[CACHE] Using cached result for expensive_calculation with args=(10, 20)
The result is 200
The result is 75
```

Exercise 4 – Submission: a1_ex4.py 25 Points

Create an asynchronous function

merge_files_concurrently(input_files: list[str], output_file: str)

that concurrently reads multiple text files (each line is a string), merges all lines in memory, and writes them to a single output file. Consider the following points:

- For each file in input_files, spawn a task that reads it line by line (async, define a separate function read_file_async(file_path: str) -> list[str]).
- Collect all lines in a list.
- Ensure that the collected lines are sorted in ascending order alphabetically.
- Write the sorted lines to output_file.

Hints:

- You can use aiofiles for asynchronous file reading, you might need to install it.
- Sorting can be done once all lines are gathered.
- When collecting the lines, your editor might show you a syntax error when you try to work with the results that you got from async read operations. You can ignore this, the code will work.

Example code:

```
if __name__ == "__main__":
    async def main():
        input_files = ["file1.txt", "file2.txt", "file3.txt"]
        output_file = "merged.txt"

        for i, f in enumerate(input_files, start=1):
            if not os.path.exists(f):
                with open(f, 'w', encoding='utf-8') as ff:
                    ff.write(f"Line {i}_1\nLine {i}_2\nLine {i}_3\n")

        await merge_files_concurrently(input_files, output_file)
        print(f"Merged {input_files} into {output_file}.")

    asyncio.run(main())
```

Example output:

```
Merged ['file1.txt', 'file2.txt', 'file3.txt'] into merged.txt.
```