Intermediate python

For each exercise, it is expected to submit the code and small document that describes how you have solved the problem. For instance, which design patterns you have used, which methods or which classes you have created, etc.

You do need to implement all the exercises, but you can choose which ones you want to implement. Except exercise 4, the exercises are independent of each other, **so you can implement only two of them and get 10 points** (the MCQ worth 10 points).

At the end of the exam, you should submit a zip file with the code and the documents that you have created, with one folder by exercise. The zip file should be named with your name and the date of the exam, for instance, `john_doe.zip` and its content should be like this:

```
john_doe.zip
├── exercise1
│   ├── piece.py
│   ├── test_piece.py
│   └── README.md
├── exercise2
│   ├── pawn.py
│   ├── test_pawn.py
│   ├── ...
│   └── README.md
├── exercise3
│   ├── play_game.py
│   ├── ...
│   └── README.md
└── exercise4
    ├── game.py
    ├── ...
    └── README.md
```

Exercise 1: Piece factory (5 points)

Based on the factory pattern seen in the course (the fromJSON method in the `GeometryElement` class of the first practical work), create a factory class that creates instances of chess pieces. The factory should have a method called `from_string` that receives a string with the symbol of the piece and returns an instance of the corresponding class.

For instance `Piece.from_string(♔)` should return an instance of the class `King` with white color, and `Piece.from_string(♚)` should return an instance of the class `King` with black color.

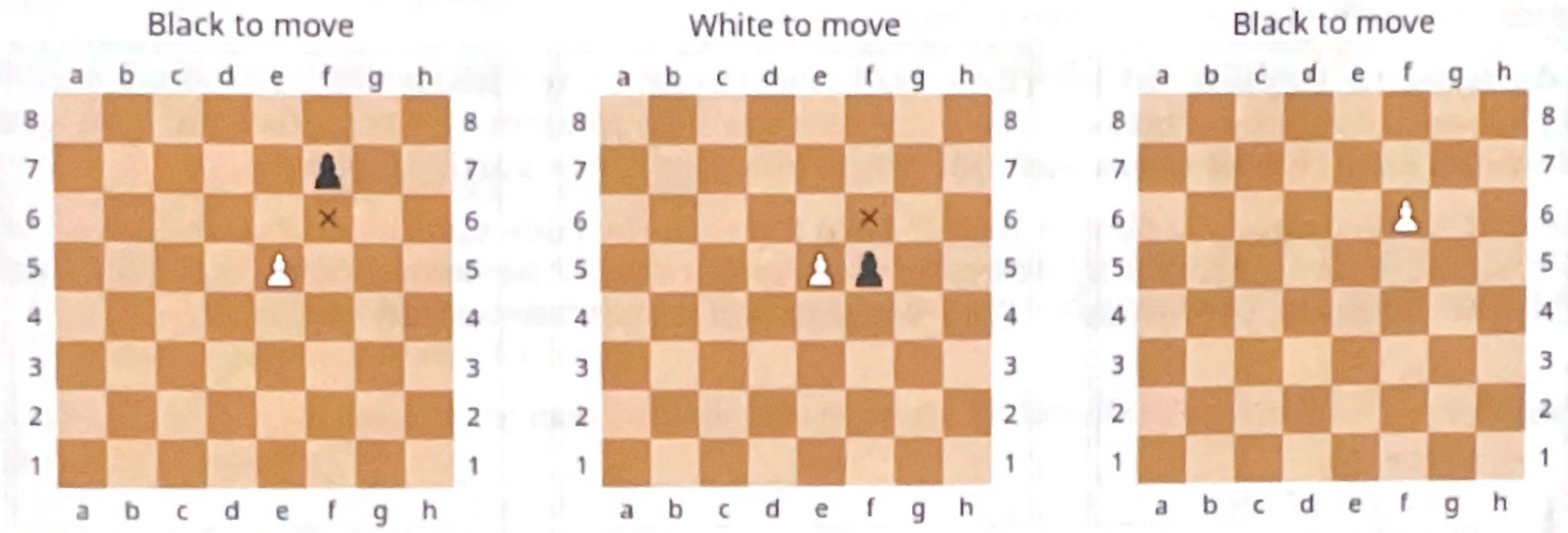
1. Add and implement the factory method `Piece.from_string` in the `Piece` class. Use the same symbols as those used when you print the chess board on the console.
2. Create a test file that tests the factory method with some pieces.

3. Write a small document that describes how you have implemented the factory method and the test cases that you have created. It can be a simple README file or a *.docx* that you provide with your code submission (a zip file with the files).

Exercise 2: en passant (5 points)

In chess, there is the rule of "en passant" which allows a pawn to capture an opponent's pawn that has moved two squares forward from its starting position. The capturing pawn moves to the square where the captured pawn landed and removes the captured pawn from the board.

Example of an *en passant* capture



The black pawn is on its initial square. If it moves to f6 (marked by ×), the white pawn can capture it.

Black moves their pawn forward two squares in a single move from f7 to f5, "passing" f6.

White captures the black pawn *en passant*, as if it had moved only one square to f6.

1. Update the `Pawn` class to include the `en_passant` rule. In the list of moves, add the `en_passant` move if the opponent's pawn has moved two squares forward in the previous move.
2. Add new tests to your test file that check the `en_passant` rule.
3. Write a small document that describes how you have implemented the `en_passant` rule and the test cases that you have created. It can be a simple README file or a *.docx* that you provide with your code submission (a zip file with the files).

Exercise 3: Playing a game (5 points)

In this exercise, you will implement a script that simulates a game of chess where moves are described in algebraic notation in a file. Here is an example of a file `game.txt` that contains:

```
1. e2-e4 e7-e5
2. Ng1-f3 Nb8-c6
3. Bf1-c4 Bf8-c5
4. Qd1-h5 Nc6-c6
5. Qh5xf7#
```

1. Create a new *main* file called `play_game.py` that reads a file with chess moves and simulates the game. The first argument to the script should be the file name. It should print the board after each move and stop when the last move has been played.