Exam: Clean Code Hangman

Your task (3h)
Write a hangman game that randomly selects a word from a given text file and allows the user to interact via the command line to solve the puzzle. The following requirements shall be satisfied:

- To start the game, the user shall execute a python module, i.e., `python -m exam.hangman`
- The game interaction shall also take place via the command line (e.g. typing inputs, printed output).
- The game shall select a random word from the `resources/words.txt` list when it is started.
- In each round the game shall report on the current progress, with still missing characters masked by underscores e.g. `H_NGM_N`
- In each round the game shall report on the used characters by printing them.
- In each round the user shall have the possibility to guess a single character.
- The game shall fail, if the user tries wrong characters more than a configurable amount of times. The default shall be 12 times.
- Already used characters shall not count towards the wrong guesses.
- The game shall end automatically when all characters have been guessed and the user shall see a message that they have won.
- In each round the user shall have the possibility to solve the word by typing it in completely (e.g., more than one character).
- If correctly solved, the game shall end and the user shall see a message that they have won.
- If not solved, the game shall end and the user shall see a message that they have lost.
- At the end of the game the user shall be presented a summary of the game including used characters and number of guesses.
- All user inputs shall be handled case insensitive.
- The implementation shall be independent of the operating system (e.g., file paths).

Please add your hangman code to the `exam` and `tests/exam` sub folders in our repository.

Requirements to pass

- Code is pushed to the seminar repository by the end of the day (16:00 CET).
- Test coverage $>= 80$ % on exam code.
- All tests are executable via command line `pytest` call and pass(!).
- Implementation follows the clean code techniques.
- Implementation follows pep-8 style
- Written "Daily Reflection" (our three questions) in a `README.md` file.
  You may also add what you wanted to do next.

It is not necessary to have a full working game, a "decent portion" is enough. Your focus should be on applying the clean code techniques. However, a good grade cannot be achieved without 80 % coverage or a commit timestamp after the deadline.

You may use online resources, code generators, etc. as well as resources from the seminar. However, do not copy and paste any code without marking it with a comment.

Procedural hints

Remind yourself on all the Clean Code Principles we had in the Seminar. It is always a good idea to start with the "business logic" (here, the part that is focused on taking guesses and revealing characters). If you wonder how to start your program without the `main` method or the user interface: This is what we did all the time with unit tests.

The rest (user interface, database, ...) is exchangeable detail. This exchangability is something your code should reflect. For the design especially OCP, SoC and SRP might render useful.

You can start cleanish or messy, important is that the result is clean. This is another reason to write tests early: to ease refactoring of your code. Wherever you started it won't be perfect and you'll have to gradually shift it towards the result.

Commit often, if something goes totally wrong you can rollback to a commit and start from there again.

Environment hints

Ensure that you have all required `pip` packages installed:

```
pip install numpy scipy
pip install pytest pytest-mock pytest-cov
pip install pycodestyle
```

Remember that you can execute the `pytest` with coverage option and path to a specific (test) folder

```
pytest /path/to/test/folder/ --cov
```

If your IDE does not have that included: You can run the style check on any given folder with

```
pycodestyle path/to/folder/
```

If you manage to provide a command line interface (CLI), ensure the code is executable as python module. I.e.,

```
python -m exam.hangman
```

The prepared (but empty) `exam/hangman.py` script should allow you to do that. (But you might end up with a different structure in the end. That is completely fine, as long it's still executable.)

Structure
Please follow the existing repository structure. Thus, create your "operational" code in the `exam` folder and keep your test suites in the `tests/exam` folder.
Be reminded that `pytest` only executes tests in files called `*_test.py` and only regards test methods following the `test_*` naming convention.
A list of words to use can be found under `resources`. A word list suitable for testing can be found in `tests/resources`.

Python hints

Use the build in `input()` method to read from the command line. Example

```python
input('Please enter your name: ')
```

In order to bypass the user input during testing you can simply mock the `input()` call towards the `builtins` module:

```python
def test_bypass_input(mocker):
    mocker.patch('builtins.input', return_value='Foo')
    # ... now call the method that uses the input() function
```

The built in `print()` statement can be mocked as well. You can also assert the number of calls or if it was called with given arguments. Example:

```python
def test_output(mocker):
    printer = mocker.patch('builtins.print')

    # ... now call the method that uses the print() statement

    assert printer.call_count == 3
    printer.assert_called_with('You have won the game!')
```

In python, when you open a resource file (e.g., a txt file), the relative path refers to the location from where you start the program **not** to the location of the script you define the relative path in. The location from where our code is called can change, thus, we usually want a path relative to the file where we define it. The magic attribute `__file__` holds the absolute path of the current file. From this we can get the folder our script is in and then go relative from there.

```python
import os

current_folder = os.path.dirname(__file__)
relative_path = os.path.join('..', 'some', 'other', 'location.txt')

full_path = os.path.join(current_folder, relative_path)
```

This `full_path` can now be used with `open()` statements, etc.

You can create your own custom error classes by simply creating an empty subclass of the `RuntimeError`. Example:

```python
class CustomError(RuntimeError):
    pass
```

Git hints

At the begin of the exam pull the main (master) branch

```
git checkout master
git pull
```

Ensure there where no errors and no conflicts reported by git. Then create a new branch:

```
git checkout -b <initials>_exam
```

Where you replace `<initials>` with your initials.

Every so often you should create an intermediate commit

```
# Add all changed files to git
git add -A
# Make a commit
git commit -m "Meaningfull message"
```

At least at the end of the exam (it might be a good idea to do this also earlier) push your changes to the seminar gitlab repository:

```
# Add all changed files to git
git add -A
# Create a commit
git commit -m "Meaningful message"
# push to the remote
git push --set-upstream origin <initials>_exam
```

After pushing the final results: Go to the gitlab web page, select your branch and ensure your result is there. I will only regard whats on gitlab!
Some IDEs save automatically for you, others don't. Ensure to save all changes to the files before using `git add`.