Python Project Tutorial 4

Linters, Code Formatters and SOLID Principles in Python

A *linter* is a code debugger that can be used to check your Python code and identify where there are errors. There is no unique choice of linter to use for this purpose, but a fast debugger of choice is *Ruff*, which can be applied to a Python file as follows:

In [ ]:
```
cd path_name # change 'path_name' as appropriate
ruff python_file.py # applies debugger to Python code
```

The result of this will be a sequence of errors in your code, labelled with the line in which they occur, and a brief description of each of the problems. If the code does not contain any errors, then the application of Ruff will not return anything (though, proceed with caution, as some problematic code is subtle and might be overlooked).

Code formatters, such as *Black*, are packages that can reformat your Python code so that the layout of code is consistent and readable. For example, here is an example of code reformatted using Black:

In [ ]:
```
# Before applying Black:
import time

def factorial(num):
    if not isinstance(num,int) or num<0:
        return '-1'
    elif num==0 or num==1:
        return 1
    else:
        return num*factorial(num-1)

# After applying Black:
import time


def factorial(num):
    if not isinstance(num, int) or num < 0:
        return -1
    elif num == 0 or num == 1:
        return 1
    else:
        return num * factorial(num - 1)
```

To apply Black, you can do the following in the command line:

In [ ]:
```
# Assuming in correct directory and virtual environment is activated:
pip install black
black factorial.py
```

Finally, SOLID principles are a set of rules that are used by software developers to write high quality code when working with OOP. These can be summarised as follows:

- Single Reponsibility Principle (SRP): a class should only have a single reponsibility or job (better maintainability, reusability and testing purposes).
- Open/Closed Principle (OCP): you should be able to extend the behaviour of a class without modifying the source code directly (better for maintainability and reusability).
- Liskov Substitution Principle (LSP): if a function takes an object of a parent class as a parameter, then it should be able to take an object of the child class as a parameter without causing errors or ambiguous behaviour.
- Interface Segregation Principle (ISP): classes should not be forced to implement interfaces with methods they do not need.
- Dependency Inversion Principle (DIP): high-level modules should not depend on low-level modules, but instead only depend on abstractions.

A more detailed discussion of OOP will be done at a later stage.