Exercise session 11

Object-oriented programming in Python. Classes, inheritance and polymorphism. Modules and packages.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

03 Dec 2025

Exercise 1: decorators and generators for the solution of ODEs

The file `hints/ex1.py` contains a generator function that solves the ODE $u' = -\sin(u)$ using explicit Euler method:

$$u_{n+1} = u_n - h \sin(u_n).$$

1. Implement a decorator `step_counter` that counts how many values are yielded from the generator, i.e., how many steps the method has performed.
2. Apply the decorator to the `explicit_euler` function.
3. Create a generator with $u_0 = 1$ and $h = 0.1$.
4. Consume exactly 10 steps from the generator and print each value.

Exercise 2: Polynomial class (1/2)

Starting from `hints/ex2.py`, implement a `Polynomial` class with the following features:

1. **Constructor and string representation**:
   - Accept variable coefficients: `Polynomial(1, 2, 3)` represents $1 + 2x + 3x^2$.
   - Implement `__repr__` to print: `"1 + 2x + 3x^2"`.
2. **Operator overloading**:
   - Implement `__add__` for polynomial addition.
   - Test: `Polynomial(1, 2) + Polynomial(3, 4)` → `"4 + 6x"`.
3. **Inheritance and polymorphism**:
   - Create abstract method `evaluate(self, x)` (`x` can be a single point or a list).
   - Subclass `StandardPolynomialEvaluator`: implements standard evaluation.
   - Subclass `HornerPolynomialEvaluator`: implements Horner's rule.

Exercise 2: Polynomial class (2/2)

4. **Decorator and performance**:
   - Implement a `@measure_time` decorator.
   - Compare performance on 10000 points between standard and Horner methods.
   - Verify that results match.

Hints:

- Use `time.time()` or `time.perf_counter()` in the decorator.
- Standard evaluation: $P(x) = \sum_{i=0}^{n} a_i x^i$.
- Horner's rule: $P(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + \ldots + x \cdot (a_{n-1} + x \cdot a_n) \ldots))$.

Exercise 3: modular data processing package (1/2)

Refactor the existing data processing code provided in `hints/ex3.py` into a modular package with multiple modules, functions, and classes.

1. Refactor the code into a modular package `dataprocessor`

```
dataprocessor/
├── __init__.py      # Package entry point.
├── operations.py    # Data processing functions.
└── data_analysis.py # DataAnalyzer class.
```

- **`__init__.py`**: Import and export necessary functions and classes. Define `__all__` to control what the package exports with `from dataprocessor import *`. In particular, it should export `process_data`, `additional_operation`, `DataAnalyzer`, `pd`.

Exercise 3: modular data processing package (2/2)

- **`operations.py`**: Contains functions for data processing:
  - `process_data(df, column1, column2, result_column)`: Adds values from two columns.
  - `additional_operation(df, column1, column2, additional_column1, other_column2)`: Performs multiplication and division operations.
- **`data_analysis.py`**: Contains a class `DataAnalyzer`, replacing `DataProcessor`, that encapsulates data analysis functionality:
  - Constructor accepts the name of the column to analyze.
  - Method `analyze_data(df)` returns mean and max values.

2. In a separate folder, create a `main_dataprocess.py` script that demonstrates the package usage.