Laboratorium 6 – Programowanie obiektowe w Python.

Języki skryptowe

Cele dydaktyczne

1. Poznanie i ćwiczenie kluczowych mechanizmów OOP w Pythonie: konstruktorów, metod specjalnych, właściwości, dziedziczenia, klas abstrakcyjnych i polimorfizmu.
2. Praktyczne zastosowanie kaczekiego typowania i wzorców projektowych.

Wprowadzenie

Celem niniejszego laboratorium jest przećwiczenie elementów programowania obiektowego w języku Python. Język ten wspiera (choć nie wymusza) paradygmat klasowo-obiektowy m.in. zakresie związków klasa-instancja, generalizacja-specjalizacja (w celu realizacji mechanizmu dziedziczenia), w tym daje możliwość wielodziedziczenia. Polimorfizm możliwy jest przez tzw. kacze typowanie.
Zadania operują na tym samych zbiorze danych, co poprzednia lista, tzn – udostępnianym przez bank danych pomiarowych Głównego Inspektoratu Ochrony Środowiska – dostępnym pod tym linkiem. Zadania z niniejszej listy można rozwiązać, modyfikując lub rozszerzając wytworzone artefakty w ramach poprzedniej listy, jeśli jest taka potrzeba.

Zadania

1. Utwórz klasę `Station`, która reprezentuje dane jednej stacji na podstawie pliku `stacje.csv`. Zaimplementuj metody `__init__`, `__str__`, `__repr__` i

`__eq__`. Dwa obiekty uznaj za równe, jeśli mają taki sam kod stacji.

2. Utwórz klasę `TimeSeries`, która reprezentuje dane pomiarowe jednej wielkości (np. PM10") dla jednej stacji. Przechowuj:
    a. nazwę wskaźnika,
    b. kod stacji,
    c. czas uśredniania,
    d. listę dat pomiaru (`datetime`),
    e. listę (lub tablicę numpy) wartości (`float` lub `None`),
    f. jednostkę pomiaru.
    g. Dodaj metodę `__getitem__`, która
        i. przyjmuje indeks lub obiekt `slice` i zwraca odpowiednie wartości: krotki postaci `(datetime, value)`.
        ii. przyjmuje obiekt `datetime.date` lub `datetime.datetime` i zwraca wartość przypisaną do danego znacznika czasu, listę wartości, względnie rzuca `KeyError`.
3. Dodaj do klasy `TimeSeries` właściwości (properties):
    a. `mean` – średnia arytmetyczna (lub `None`, jeśli brak danych),
    b. `stddev` – odchylenie standardowe.
4. Utwórz klasę abstrakcyjną `SeriesValidator` (moduł abc), definiującą metodę abstrakcyjną `analyze(series: TimeSeries)`. zwracającą listę komunikatów (str) opisujących wykryte anomalie lub pustą listę, gdy wszystko jest w porządku. Następnie zdefiniuj konkretne klasy dziedziczące:
    a. OutlierDetector, która wykrywa wartości oddalone o więcej niż *k* odchyleń standardowych od średniej, gdzie *k* jest parametrem klasy.
    b. ZeroSpikeDetector, która wykrywa co najmniej 3 zera lub braki danych z rzędu,
    c. ThresholdDetector, która wykrywa wartości przekraczające zadany jako parametr próg.
    d. **Wersja rozszerzona:** CompositeValidator – opracuj klasę realizującą złożony, kompozytowy walidator, który łączy wiele różnych walidatorów w dysjunktywnym (OR) lub koniunktywnym (AND) trybie.
5. Utwórz klasę `Measurements`, która agreguje dane z wielu plików CSV zawierających pomiary jednej wielkości zanieczyszczeń (np. „toluen", „benzen") w różnych lokalizacjach (stacjach) i czasie.
    a. Klasa powinna leniwo wczytywać dane z katalogu zawierającego pliki CSV, w których:
        i. każdy plik dotyczy jednego wskaźnika (np. „toluen"), jednej

częstotliwości (np. „1g”) i jednego roku,
   ii. każda kolumna reprezentuje jedną stację,
   iii. każdy wiersz to pomiar dla wszystkich stacji w tej samej godzinie lub dniu.
b. Konstruktor klasy powinien przyjmować ścieżkę do katalogu zawierającego pliki CSV, identyfikować pliki po nazwie, nie wczytywać danych od razu (tzn. realizować leniwe ładowanie przy pierwszym dostępie).
c. Klasa powinna implementować metody:
   i. `__len__()` – zwraca liczbę obiektów `TimeSeries`, możliwych do załadowania
   ii. `__contains__(parameter_name: str)` – zwraca `True`, jeśli co najmniej jeden `TimeSeries` zawiera podany wskaźnik.
   iii. `get_by_parameter(param_name: str)` – zwraca wszystkie obiekty `TimeSeries`, których `parameter_name == param_name`.
   iv. `get_by_station(station_code: str) -> list[TimeSeries]` – zwraca wszystkie serie danych dla danej stacji (różne wskaźniki, lata i częstotliwości).

6. Dodaj do klasy `Measurements` metodę `detect_all_anomalies(validators: list[SeriesValidator], preload: bool = False)`.
   a. Jeśli `preload` ustawione jest na `True`, wymuś pełne załadowanie danych. Jeśli `preload=False` – waliduj tylko te serie, które zostały wcześniej załadowane .
7. Następnie uruchom każdą strategię (`SeriesValidator`) na każdym obiekcie szeregu czasowego (`TimeSeries`), zagreguj dane i zwróć jako słownik.
8. Zademonstruj działanie kaczego typowania poprzez utworzenie klasy `SimpleReporter`, która:
   a. nie dziedziczy po klasie SeriesValidator,
   b. implementuje metodę analyze(series),
   c. zwraca listę zawierającą jedną informację, np:
      i. [f“Info: {param_name} at {station_code} has mean = …”]
   d. Następnie przygotuj przykładowy obiekt TimeSeries i utwórz listę “obiektów analizujących”, zawierających m.in. instancje specjalizacji SeriesValidator oraz obiekt SimpleReporter.. Przeiteruj po liście obiektów wywołując metode analyze(series) – nie sprawdzając typu obiektu (bez

isinstance, hasattr, itp.).

e. Zademonstruj w ten sposób, że Python wspiera polimorfizm strukturalny – czyli kacze typowanie: obiekty mogą być traktowane jako analizatory, jeśli zachowują się jak analizatory (mają odpowiednią metodę), niezależnie od ich typu czy dziedziczenia.