

Serwery (Python)

Poniższy mini-projekt ma na celu pomóc Ci odświeżyć znajomość podstawowych zagadnień związanych z programowaniem obiektowym w języku Python.

Opis problemu


Kilka hurtowni produktów (spożywczych) postanowiło stworzyć uniwersalny system do obsługi katalogów oferowanych przez nie produktów – system ma powstać z myślą o klientach, którzy chcieliby posiadać możliwość sprawdzenia cen pewnych produktów w wybranej hurtowni.

Każda hurtownia posiada (pojedynczy) własny serwer przechowujący katalog oferowanych przez nią produktów, przy czym sposób przechowywania informacji różni się pomiędzy poszczególnymi typami serwerów. Co więcej, ponieważ każda hurtownia może stosować inne oznaczenia tych samych produktów, dlatego nie istnieje wspólna baza produktów.

System nie ma na celu łączyć wszystkich serwerów w jedną sieć, tylko umożliwić jego efektywne wdrożenie w każdej z hurtowni (z osobna). Powinien zatem być zaprojektowany w taki sposób, aby uwzględniał różnorodność reprezentacji katalogu produktów w ramach serwera w poszczególnych hurtowniach. W szczególności klienci hurtowni powinni być w stanie uzyskiwać požądane informacje bez znajomości sposobu reprezentacji katalogu produktów na konkretnym serwerze.

Poniżej podano specyfikację klas obiektów występujących w zadaniu.

Produkty

- Opis produktu składa się z nazwy produktu oraz jego ceny.
- Nazwa produktu ma postać pojedynczego ciągu znaków w postaci `<ciąg_liter><ciąg_cyfr>`, przy czym litery mogą być dowolnej wielkości (ale ich rozmiar ma znaczenie, czyli np. `a` i `A` są traktowane jako dwie różne litery).
Poprawna nazwa zawiera co najmniej jedną literę i co najmniej jedną cyfrę.
Przykładowe nazwy produktów: `x0129`, `AB12`, `ab123`.
 Zweryfikuj poprawność wprowadzonej nazwy; w przypadku błędnej nazwy rzuć wyjątek typu `ValueError` (zob. [wskazówki](#)).
- Nazwa produktu i jego cena w sposób unikalny identyfikują dany produkt (tj. dwie różne instancje klasy reprezentującej produkt, z których każda posiada tę samą kombinację nazwy i ceny produktu, traktowane są jako odwołujące się do tego samego produktu).

Serwery

- Serwery udostępniają funkcjonalność wyszukiwania produktów, których nazwa składa się z ciągu `“dokładnie n liter (dowolnej wielkości), a następnie (dokładnie) 2-3 cyfry”` (gdzie `n` stanowi parametr tej funkcji – domyślnie `$n=1$`), przy czym wyniki są zwracane w postaci listy produktów posortowanej wg ich rosnącej ceny. (Zwróć uwagę, że w nazwach produktów może występować dowolna liczba liter oraz dowolna liczba cyfr.)
Przykładowo, dla `$n=2$` serwer znajdzie m.in. produkty o nazwach: `AB12` oraz `ab123`, ale już nie `A12`, `ab1`, `Ab1234` lub `Abc12`.
- Jeśli żaden produkt nie spełnia kryterium wyszukiwania, serwer zwraca pustą listę.

- W przypadku gdy liczba znalezionych produktów przekracza pewną z góry ustaloną wartość (zdefiniowaną jako atrybut klasowy `n_max_returned_entries`, identyczną dla wszystkich serwerów), metoda służąca do wyszukiwania produktów powinna rzucić stosowny (niestandardowy, zdefiniowany przez programistę) wyjątek.
Możesz przyjąć dowolną wartość tego atrybutu, np. 3.
- Istnieją dwa typy serwerów, różniących się głównie sposobem przechowywania danych o produktach – jedne przechowują je w postaci listy produktów (typ `list`), a inne w postaci słownika (typ `dict`, kluczem jest nazwa produktu, wartością – obiekt reprezentujący produkt). W przypadku każdego typu serwera stworzymy jego nową instancję z użyciem listy produktów. Przyjmij, że interfejs (API) obu typów serwerów jest identyczny.
W tym przypadku wystarczające (i uzasadnione) będzie stworzenie odpowiedniej klasy abstrakcyjnej (zob. skrypt do C++ rozdz. “Interfejsy” oraz skrypt do Pythona rozdz. “Klasy abstrakcyjne i interfejsy”).
- W ramach każdego katalogu (tj. każdej instancji serwera) nazwy produktów są unikalne, przy czym nie trzeba tego weryfikować.
- Serwery przechowują informacje o produktach “na wyłączność” (produkty nie są współdzielone między serwerami, istnieją tylko w ramach danego serwera).

Klienci

- Każdy klient posiada skojarzony z sobą serwer (serwery mogą być współdzielone między klientami) oraz metodę służącą do obliczania łącznej ceny produktów spełniających kryterium wyszukiwania.
- Przytoczona wyżej metoda pobiera jako argument liczbę początkowych liter w nazwie produktu – wspomniane wcześniej `n` – i zwraca:
 - albo łączną cenę produktów,
 - albo `None` – w przypadku, gdy serwer rzucił wyjątek lub gdy nie znaleziono ani jednego produktu spełniającego kryterium.

Zadanie

Stwórz diagram UML reprezentujący występujące w systemie klasy (i ich składowe) wraz z zależnościami między nimi – w tym celu skorzystaj z narzędzia **PlantUML**.

⚠ Zaproponowany diagram nie będzie w żaden sposób wiążący w kontekście późniejszej implementacji systemu. Powinien jednak “nosić znamiona sensowności” i ilustrować koncepcję, która pozwoli na realizację funkcjonalności opisanej w treści zadania (np. definiowanie klasy *Klient* jako klasy potomnej dla *Produkt* jest zdecydowanie błędne itp.).

Każdą z klas w kodzie diagramu UML zdefiniuj jawnie, za pomocą słowa kluczowego `class` – to ułatwi weryfikowanie rozwiązania.

W rozwiązaniu oprzyj się na poniższym szkieletcie programu (możesz dodawać m.in. nowe klasy i funkcje, lecz w Twoim rozwiązaniu muszą znaleźć się poniższe elementy):

[servers_skeleton.py](#)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from typing import Optional
```

```
class Product:
    # FIXME: klasa powinna posiadać metodę inicjalizacyjną przyjmującą
    # argumenty wyrażające nazwę produktu (typu str) i jego cenę (typu float)
    # -- w takiej kolejności -- i ustawiającą atrybuty `name` (typu str) oraz
    # `price` (typu float)

    def __eq__(self, other):
        return None # FIXME: zwróć odpowiednią wartość

    def __hash__(self):
        return hash((self.name, self.price))

class TooManyProductsFoundError:
    # Reprezentuje wyjątek związany ze znalezieniem zbyt dużej liczby
    # produktów.
    pass

# FIXME: Każda z poniższych klas serwerów powinna posiadać:
# (1) metodę inicjalizacyjną przyjmującą listę obiektów typu
# `Product` i ustawiającą atrybut `products` zgodnie z typem
# reprezentacji produktów na danym serwerze,
# (2) możliwość odwołania się do atrybutu klasowego
# `n_max_returned_entries` (typu int) wyrażający maksymalną dopuszczalną
# liczbę wyników wyszukiwania,
# (3) możliwość odwołania się do metody `get_entries(self,
# n_letters)` zwracającą listę produktów spełniających kryterium
# wyszukiwania

class ListServer:
    pass

class MapServer:
    pass

class Client:
    # FIXME: klasa powinna posiadać metodę inicjalizacyjną przyjmującą
    # obiekt reprezentujący serwer

    def get_total_price(self, n_letters: Optional[int]) ->
    Optional[float]:
        raise NotImplementedError()
```

☺ Metody `__eq__` i `__hash__`

Domyślnie interpreter języka Python stosuje następującą implementację metody `__eq__()` – obiekty `x` i `y` są sobie równe, jeśli `id(x) == id(y)`¹⁾. Jednak w przypadku tego projektu zakładamy, że dwie instancje klasy `Product` o tej samej nazwie i tej samej cenie w istocie opisują ten sam produkt – zatem musimy odpowiednio nadpisać metodę `__eq__()`.

Po co dodatkowo nadpisywać metodę `__hash__()`?

Choć metoda `get_entries()` powinna zwracać elementy w określonym porządku, często spotykanym błędem jest zwracanie ich w losowej kolejności – w związku z tym jeden z testów jednostkowych weryfikuje to, czy w ogóle zwrócone zostały poprawne elementy. Najprościej można to sprawdzić dokonując rzutowania zwróconej listy na zbiór i skorzystać z asercji `assertSetEqual`.

Jednak typ `set` (podobnie jak typ `dict`) w celu efektywnego dostępu do elementów (w tym operacji wstawiania/usuwania) przechowuje elementy w postaci [tablicy z haszowaniem](#) i wymaga, aby typ elementu posiadał zdefiniowaną tzw. [funkcję haszującą](#). Nadpisanie domyślnej implementacji metody `__eq__()` sprawia, że domyślna implementacja metody `__hash__()` przestaje być poprawna i ją również należy odpowiednio nadpisać. Funkcja haszująca przyporządkowuje dowolnemu obiektowi kolekcji pewną liczbę całkowitą o stałej szerokości. Kilka obiektów może posiadać tę samą wartość funkcji haszującej, jednak wartość funkcji haszującej dla tego samego obiektu nie może się zmieniać. Wartości funkcji haszującej powinny być nadawane w taki sposób, aby zminimalizować ryzyko kolizji (tj. sytuacji, gdy różnym obiektom przyporządkowywana jest ta sama wartość). W tym przypadku wystarczy, jeśli obliczymy wartość wbudowanej funkcji haszującej dla krotki złożonej z tych samych atrybutów, które zostały użyte do sprawdzenia równości dwóch instancji.

Zagadnienie funkcji skrótu oraz jej związku z typami `set` i `dict` zostało przystępnie omówione w poniższych źródłach:

- [What does hash do in Python?](#)
- [Hashing and Equality in Python](#)

Napisz program realizujący funkcjonalność przedstawioną w opisie problemu. Pamiętaj o dobrych praktykach programistycznych w Pythonie, w szczególności o stosowaniu podpowiedzi typów (*type hinting*).

Napisz [testy jednostkowe](#) weryfikujące poprawność działania programu w poniższych scenariuszach:

- Czy wyniki zwrócone przez serwer przechowujący dane w liście są poprawnie posortowane?
- Czy przekroczenie maksymalnej liczby znalezionych produktów powoduje rzucenie wyjątku?
- Czy funkcja obliczająca łączną cenę produktów zwraca poprawny wynik w przypadku rzucenia wyjątku oraz braku produktów pasujących do kryterium wyszukiwania?

Przykładowe testy jednostkowe:

`servers_tests.py`

```
import unittest
from collections import Counter

from servers import ListServer, Product, Client, MapServer

server_types = (ListServer, MapServer)

class ServerTest(unittest.TestCase):

    def test_get_entries_returns_proper_entries(self):
        products = [Product('P12', 1), Product('PP234', 2),
Product('PP235', 1)]
        for server_type in server_types:
            server = server_type(products)
            entries = server.get_entries(2)
            self.assertEqual(Counter([products[2], products[1]]),
Counter(entries))

class ClientTest(unittest.TestCase):
    def test_total_price_for_normal_execution(self):
        products = [Product('PP234', 2), Product('PP235', 3)]
        for server_type in server_types:
            server = server_type(products)
            client = Client(server)
            self.assertEqual(5, client.get_total_price(2))

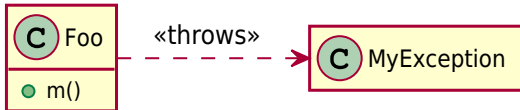
if __name__ == '__main__':
    unittest.main()
```

Wskazówki

Pamiętaj o:

- wywołaniu w metodzie inicjalizującej definiowanej przez siebie klasy metody inicjalizującej jej klasy macierzystej
- możliwości stosowania wyrażeń regularnych (zob. [re.fullmatch\(\)](#)) – przydatne przy realizacji wyszukiwania produktów na podstawie wzorca etykiety
- możliwości sortowania kolekcji elementów według ręcznie zdefiniowanego kryterium (zob. [sorted\(\)](#) i [Sorting HOW TO – Key Functions](#))

Na diagramie UML relację związaną z rzucaniem wyjątku możesz oznaczyć jako:



Więcej wskazówek znajdziesz tu: **Projekt Python - Serwery - wskazówki**

1)

Ściślej – zob. [How is __eq__ handled in Python and in what order?](#)

From:
<http://home.agh.edu.pl/~mdig/dokuwiki/> - **MVG Group**

Permanent link:
<http://home.agh.edu.pl/~mdig/dokuwiki/doku.php?id=teaching:programming:soft-dev:topics:servers>

Last update: **2020/11/25 14:38**

