Praca domowa - funkcje, klasy, django

Rafał Korzeniewski, korzeniewski@gmail.com

November 17, 2023

1 Zadania funkcje

Dziękuję za ostatni zjazd.

Dla utrwalenia proponuję kilka ćwiczeń dotyczących deklarowania funkcji

1.1 suma

Napisz funkcję `suma`, która zsumuje elementy z podanej kolekcji. Na wejściu można podać listę, tuplę, zbiór lub słownik. Funkcja powinna wybrać z kolekcji tylko te elementy, które są liczbami i zwrócić ich sumę. W przypadku słownika sumowane powinny być wartości (a nie klucze).

Przykładowy test:

```
def test_suma():
    assert suma((1, 2, 3)) == 6
    assert suma((1, "2", 3)) == 4
    assert suma([8, 2, 3, 0, 8]) == 21
    assert suma([1, 2, "a"]) == 3
    assert suma({1:8, 2:2, 3:3, 4:0, 5:9}) == 22
    assert suma({1:8, 2:"a", 3:3, 4:0, 5:9}) == 20
```

Jako rozwinięcie zadania - dodaj opcjonalny parametr `cast_to_int`. Jeśli będzie ustawiony na `True`:

```
def test_suma_with_cast_to_int():
    assert suma((1, "2", 3), cast_to_int=True) == 6
```

Postaraj się dodać annotacje. Postaraj się zachować zasady PEP8. Sprawdź pokrycie testami przy pomocy `coverage`

1.2 palindrom

Napisz funkcję, która sprawdzi, czy zadany tekst jest palindromem. Niech funkcja nazywa się `is_palindrome`. Powinna umożliwić przekazanie tekstu jako argument. Zwracać powinna `True` lub `False` w zależności od tego czy dany tekst jest palindromem.

Zacznij od najprostszych palindromów - np. “kajak”. Potem spraw by Twój kod nie był wrażliwy na wielkość liter “Kajak” to też palindrom.

Poszukaj innych palindromów - w tym wielowyrazowych. Np. “Kobyła ma mały bok”. Znaki interpunkcyjne i białe nie powinny wpływać na wynik. “Kobyła, ma mały bok!” to też będzie palindrom.

Niektóre palindromy są naprawdę dłuuuugie i zabawne. Spróbujcie swoich sił z palindromami, które znajdziecie tutaj: http://www.palindromy.pl/pal_naj.php

1.3 trójkąt Pascala

Trójkąt Pascala to trójkątna tablica liczba zbudowana w następujący sposób:

```
0                      1
1                    1   1
2                  1   2   1
3                1   3   3   1
4              1   4   6   4   1
5            1   5   10  10   5   1
6          1   6   15  20  15   6   1
7        1   7   21  35  35   21  7   1
8      1   8   28  56  70  56   28  8   1
9    1   9  36   84  126 126  84  36  9   1
       . . . . . . . . . . . . . . . . . .
```

Napisz funkcję, która utworzy n-wierszy takiego trójkąta. Funkcja przyjmuje argument `n` a następnie zwraca listę zawierającą listę wierszy.

Opcjonalnie: doodaj parametr opcjonalny `to_str`. Jeśli będzie miał wartość `True` to postaraj się narysowac coś podobnego jak ten powyższy trójkąt

Przykładowe testy:

```
def test_pascal_triangle():
    actual = pascal_triangle(6)
    assert actual == [
        [1],
        [1, 1],
        [1, 2, 1],
        [1, 3, 3, 1],
        [1, 4, 6, 4, 1],
        [1, 5, 10, 10, 5, 1],
    ]


def test_pascal_triangle_to_str():
    actual = pascal_triangle(7, to_str=True)
    assert actual == """
             1
           1   1
         1   2   1
       1   3   3   1
     1   4   6   4   1
   1   5  10  10   5   1
 1   6  15  20  15   6   1
```

"""

Uwaga: Znacznie prościej będzie zrobić te zadania jeśli rozbijecie kod na kilka małych funkcji - np. jedna funkcja generuje n-ty wiersz trójkąta. Inna funkja generuje cały trójkąt i zwraca albo listę list zawierających liczby albo napis generowany przez jeszcze inna funkcję (w zależności od to_str)

Przyda się wiedza o f-string. Przda się wiedza o metodach napisów - np. `center`, `join`

join działa tak:

[1]:
```
"-".join(["1", "2", "3"])
```

[1]:
```
'1-2-3'
```

Uwaga 2: zadbajcie o pokrycie testami. Pilnujcie zasad PEP8. Mozna je sprawdzić instalując sobie np `flake8`. Zakładając, że wasz plik nazywa sie `zadanie_pascal.py`, to w terminalu (command line - ale nie REPL):

```
$ pip install flake8
$ flake8 zadanie_pascal.py
```

Jeśli macie błędy dotyczące PEP 8 to wynikiem tego będzie lista błędów. Np:

```
$ flake8 zad_4.py
zad_4.py:21:80: E501 line too long (83 > 79 characters)
zad_4.py:46:1: E302 expected 2 blank lines, found 1
zad_4.py:49:15: W291 trailing whitespace
```

oznacza to, że: * Linia 21 jest za dużo znaków - jest za długa. * Linia 46 - powinny być dwie linie odstępu a jest jedna * Linia 49 - jest zbędna spacja na końcu.

to akurat dotyczy tego miejsca:

```
def test_pascal_triangle_to_str():
    actual = pascal_triangle(7, to_str=True)
    assert actual == """
             1
           1   1
         1   2   1
       1   3   3   1
     1   4   6   4   1
   1   5  10  10   5   1
 1   6  15  20  15   6   1
"""
```

czepia się spacji na końcach wierszy w napisie.

Możemy to rozwiązać poprzez dodanie `#noqa` w odpowiedniej linii

```
def test_pascal_triangle_to_str():
    actual = pascal_triangle(7, to_str=True)
    assert actual == """  # noqa
             1
           1   1
```

```
          1   2   1
        1   3   3   1
      1   4   6   4   1
    1   5  10  10   5   1
  1   6  15  20  15   6   1
"""
```

2 PROJEKT - SMART HOME

Utwórzymy projekt, w którym będziemy symulować obsługę inteligentnych urządzeń. Nasz projekt poprzez wysyłanie wiadomości do urządzeń potrafi uruchamiać różne ich funkcjonalności takie jak włączenie czy wyłączenie świateł. Włączenie głosników, odtworzenie muzyki, uniesienie czy opuszczenie zasłon

Twoim zadaniem jest zaimplementować projekt w oparciu o poniższe wskazówki

Projekt podzielimy na szereg modułów. W następującej strukturze:

```
.
  iot
      __init__.py
      device.py
      devices.py
      diagnostics.py
      message.py
      service.py
  main.py
```

- `iot/message.py` zawierać będzie definicję wiadomości

```python
from dataclasses import dataclass
from enum import Enum, auto


class MessageType(Enum):
    SWITCH_ON = auto()
    SWITCH_OFF = auto()
    CHANGE_COLOR = auto()
    PLAY_SONG = auto()
    OPEN = auto()
    CLOSE = auto()


@dataclass
class Message:
    device_id: str
    msg_type: MessageType
    data: str = ""
```

- `iot/device.py` powinien zawierać interfejs opisujący urządzenie (`Device`). Powinien zawierać

następujące abstrakcyjne metody:

connect

`nie przyjmuje nic, zwraca None`

disconnect

`nie przyjmuje nic, zwraca None`

send_message

`przyjmuje message_type który jest klasy MessageType oraz data - napis`

def status_update(self) -> str:

`nie przyjmuje nic zwraca napis`

- `iot/devices.py` powinien zawierać klasy reprezentujące urządzenia. Mają one spełniać interfejs `Device`.

Np. dla urządzenia Hue Light, metody mają mieć następujące działanie:

`connect`

`drukuje napis "Connecting Hue Light"`

`disconnect`

`drukuje napis "Disconnecting Hue Light"`

`send_message`

`przyjmuje odpowiedni MessageType i data i drukuje:`

`Hue light handling message of type {message_type.name} with data [{data}].`

`status_update`

`drukuje napis hue_light_status_ok`

Oprócz Hue Light, zaimplementuj urządzenia SmartSpeaker i Curtains

- `iot/diagnostics.py`

Ma zawierać funkcję `collect_diagnostics`, która przyjmie jako argument device i zwróci None

Wewnątrz powinna być implementacja która: wypisze tekst:

`Connecting to diagnostics server.`

wywoła dla device metodę `status_update` i zwróci napis

`Sending status update <tutaj status> to server.`

- `iot/service.py`

powinien zawierać funkcję do generowania id `generate_id`, która zwróci losowy ciąg dużych liter o zadanej długości

powinien zawierać klasę `IOTService`

Będzie ona przechowywać zarejestrowane urządzenia

Klasa ta powinna mieć metody:

```
register_device
    przyjmuje Device
    wywołuje metodę connect dla Device
    generuje id: device_id
    dodaje urządzenie do listy `devices`

unregister_device
    przyjmuje device_id
    odłącza urządzenie wywołując metodę disconect dla device o zadanym id

get_device
    przyjmuje device_id
    usuwa urzadzenie z listy

run_program
    przyjmuje listę zawierającą Message
    drukuje: "=====RUNNING PROGRAM======"
    dla każdej wiadomości
    wywołuje dla wskazane w Message urządzenia metodę send_message z odpowiednimi danymi
    drukuje "=====END OF PROGRAM======"

test_devices
    nie przyjmuje argumentow
    drukuje: Start test devices
    dla każdego urządzenia z listy wywołuje funkcję collect_diagnostics
```

- moduł `main.py`

zawieraja i wywołuje funkcję `main`

```
    # tworzy instancję IOTService

    # tworzy instancje dostępnych urządzeń

    # rejestruje urządzenia

    # testuje urządzenia

    # tworzy programy (listy Message)
```

```
    np program wake up

    włącza światłą (MessageType.SWITCH_ON)
    włącza głośniki
    odgrywa piosenkę (MessageType.PLAY_SONG)
    otwiera zasłony


    program sleep
    wyłącza światła ( MessageType.SWITCH_OFF)
    wyłącza głośniki
    zasłania zasłony (MessageType.CLOSE)


# uruchamia programy

# usuwa urządzenia

# konczy program
```

3 SMART HOME DJANGO

Zaprojektuj i stworz interfejs webowy w Django umożliwiający użytkownikom kontrolowanie inteligentnych urządzeń.

- Utwórz nowy projekt Django lub użyj istniejącego.
- Utwórz nową aplikację w projekcie, np. “iot_control”.
- Zaprojektuj i zaimplementuj logikę, widoki, modele, które umożliwią pokrycie podobnych funkcjonalności jak te z main

Przy pomocy `IOTService` z poprzedniego zadania chcesz móc dodac urządzenie, przeslac jakas wiadomosc do urzadzenia, odczytac jego stan itd

[ ]: