Лабораторная работа №6

Задание

Лабораторная работа №6

Задания на декораторы

- 3. Напишите декоратор upcase_result, который будет переводить все символы результирующего значения функции reverse_str в верхний регистр. На вход функции reverse_str подается строка и возвращается ее инвертированное представление.
- 9. Напишите декоратор time_run, вычисляющий время выполнения декорируемой функции и возвращающий время выполнения вместо результата декорируемой функции. В качестве декорируемой напишите функцию algebraic_sum, на вход которой подается два значения N и k (по умолчанию равно 2). Функция должна возвращать результат следующего выражения: $1^k + 2^k + 3^k + \ldots + N^k$
- 10. Напишите декоратор mul_result с целочисленным аргументом N, умножающий результат выполнения декорируемой функции на N и возвращающий полученное значение. В качестве декорируемой напишите функцию add, вычисляющий сумму двух поступающих на ее вход значений.

```
'''
created by Bulat 16.02.2023
solving problems for decorators
'''
from typing import Callable
import time


def upcase_result(func: Callable[[str], str]) -> Callable[[str], str]:
    '''
    receives a function as input and returns the result of
    its operation in uppercase
    '''
    def wrapper(string: str):
        result = func(string)
        return result.upper()
    return wrapper

@upcase_result
```

```
def reverse_str(string: str) -> str:
    'return reverse string'
    return string[::-1]


def time_run(func: Callable[[int, float], float]) -> Callable[[int,
float], float]:
    def wrapper(n: int, k: float):
        start = time.time()
        func(n, k)
        scored_time = time.time() - start
        return scored_time
    return wrapper

@time_run
def algebraic_sum(n: int, k: float):
    sm = 0
    for i in range(n+1):
        sm += i ** k
    return sm


def mul_result_factory(n: int):
    def mul_result(func: Callable[[float, float], float]) ->
Callable[[float, float], float]:
        def wrapper(a: float, b:float):
            return func(a, b) * n
        return wrapper
    return mul_result

mul_result_10 = mul_result_factory(n=10)

@mul_result_10
def add(a: float, b: float):
    return a + b
```

Тестирование декораторов

```
import unittest
import decorators as dc

import time
from datetime import timedelta

class TestDecorators(unittest.TestCase):
    def test_task_3(self):
        test_cases = [
            ("foo", "OOF"),
            ("udpate", "ETAPDU")
```

```
        ]
        for test in test_cases:
            param, right_answer = test
            self.assertEqual(dc.reverse_str(param), right_answer)

    def test_task_9(self):
        test_cases = [
            (1, 1),
            (2, 10),
            (10, 10)
        ]
        for test in test_cases:
            start_test = time.time()
            scored_time = dc.algebraic_sum(*test)
            real_time = time.time() - start_test
            different = real_time - scored_time
            self.assertLess(different, 1)



    def test_task_10(self):
        test_cases = [
            ((1, 2), 30),
            ((5, 6), 110),
            ((0, 0), 0),
            ((-4, -1), -50)
        ]
        for test in test_cases:
            params, right_answer = test
            self.assertEqual(dc.add(*params), right_answer)
```

Задания на генераторы

- 1. Напишите генераторную функцию my_generator, которая позволяет итерироваться по элементам заданного отрезка с настраиваемым шагом step (по умолчанию равен 1). При написании данной функции запрещено использовать стандартную функцию верхнего уровня – range.
- 3. Напишите генераторную функцию my_enumerate, на вход которой подается список и она возвращает на каждой итерации кортеж, состоящий из двух элементов: index – текущий номер индекса элемента, val – значение, хранящееся по этому индексу. При написании данной функции запрещено использовать стандартную функцию верхнего уровня – enumerate.
- 5. Напишите генераторную функцию my_pow, на вход которой подается целочисленное значение n. На каждом шаге итерации генератор

должен возвращать $n = n^2$.

```
from typing import Any, Generator, Tuple, Union


def my_generator(start: int, end: int, step: int = 1) -> Generator[int,
None, None]:
    iterator = start
    while iterator < end:
        yield iterator
        iterator += step


def my_enumerate(array: list[Any]) -> Generator[Tuple[Any, int], None,
None]:
    index = 0
    for element in array:
        yield (index, element)
        index += 1


def my_pow(value: Union[int, float]) -> Generator[Union[int, float], None,
None]:
    while 1:
        yield value
        value **= 1
```

Тестирование Генераторов

```
import unittest
import generators as gr


class TestGenerators(unittest.TestCase):
    def test_task_3(self):
        test_cases = [
            [1, 10, 3],
            [5, 20, 2],
            [2, 2, 1]
        ]

        iterator = 1
        for test in test_cases:
            self.assertEqual(
                list(gr.my_generator(*test)),
                list(range(*test))
            )
```

```
    def test_enumerate(self):
        test_cases = [
            ['a', 'b', 'c', 'd', 'e'],
            [0, 1, 2, 3, 4],
            [0],
        ]

        for test in test_cases:
            generated_list = list(gr.my_enumerate(test))
            right_list = list(enumerate(test))
            self.assertEqual(generated_list, right_list)

    def test_my_pow(self):
        value = 2
        generator = gr.my_pow(value)
        for i in range(1, 10):
            self.assertEqual(value, next(generator))
            value *= value
```