МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

***Циклова комісія програмування***

**Робочий план**

з курсу «Основи алгоритмізації та програмування»

Бєлан Нікіта Володимирович

*ПІБ студента*

студента групи **2П-20**

варіант \_\_4\_\_\_

Викладач Марченко С. В.

|  |  |  |
| --- | --- | --- |
| **Вид роботи** | **Дата** | **Оцінка** |
| *Пром. контроль* |  |  |
| *Залік* |  |  |

**Черкаси-2022**

**1 бал Розгляньте правила побудови** [**послідовності Q Хофштадтера**](https://ru.wikipedia.org/wiki/Последовательность_Хофштадтера) **та реалізуйте ітератор, який дозволятиме генерувати члени цієї послідовності. Напишіть інтерфейс взаємодії, при якому користувач вводитиме номер елемента послідовності, а програма повертатиме значення цього елемента.**

**class QHofstadter:**

**def \_\_init\_\_(self, len):**

**self.q = [1, 1]**

**self.n = 0**

**self.len = len**

**def \_\_iter\_\_(self):**

**return self**

**def \_\_next\_\_(self):**

**if self.n > self.len:**

**raise StopIteration**

**if self.n >= 2:**

**val = self.q[self.n - self.q[self.n - 1]] + self.q[self.n - self.q[self.n - 2]]**

**self.q.append(val)**

**self.n += 1**

**return self.q[self.n - 1]**

**try:**

**num = int(input('num:'))**

**iter = QHofstadter(num)**

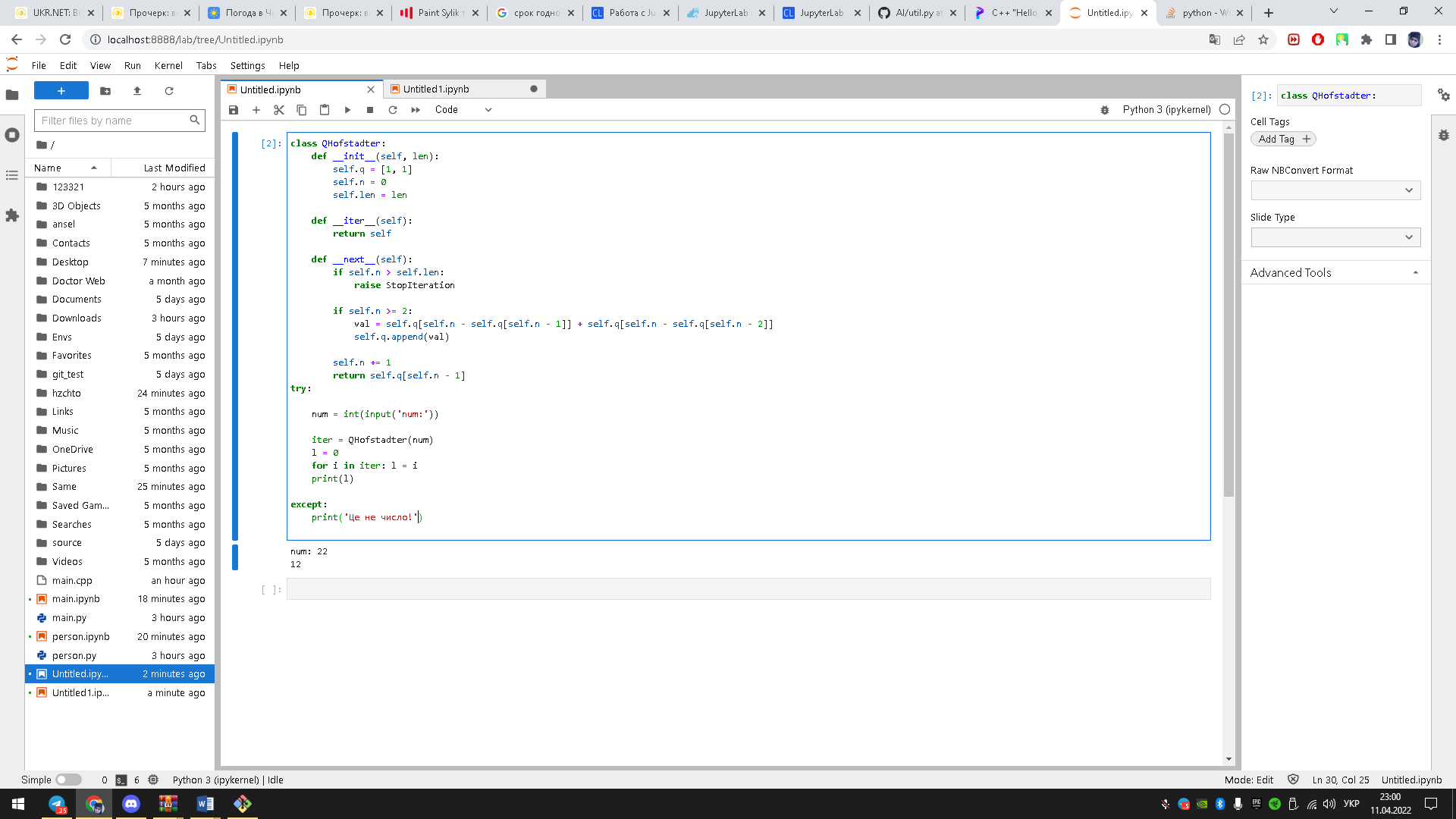
**l = 0**

**for i in iter: l = i**

**print(l)**

**except:**

**print('Це не число!')**



**1 бал Програмно реалізуйте та законспектуйте в звіті туторіали стосовно роботи** [**декораторів**](https://www.python-course.eu/python3_decorators.php)**,** [**мемоїзації**](https://www.python-course.eu/python3_memoization.php) **та** [**каррування**](https://www.python-course.eu/currying_in_python.php)**.**

**Декоратори**. Для того, щоб зрозуміти, як працюють декоратори, в першу чергу слід згадати, що функції в python є об'єктами, відповідно їх можна повертати з іншої функції або передавати як аргумент. Також слід пам'ятати, що функція в python може бути визначена і всередині іншої функції.

Згадавши це, можна сміливо переходити до декораторів.

**Декоратори** - це, по суті, "обгортки", які дають нам можливість змінити поведінку функції, не змінюючи її код.

Створимо свій декоратор вручну:

def my\_decorator(func\_to\_decorate):

def wrapper():

print('before func')

func\_to\_decorate()

print('after func')

return wrapper

def foo():

print('Hello, World!')

decorate\_foo = my\_decorator(foo)

decorate\_foo()

для зручності перезапишемо функцію:

foo = my\_decorator(foo)

foo()

Власне, це є декоратори. Ось так можна було записати попередній приклад, використовуючи синтаксис декораторів:

@my\_decorator

def foo():

print('Hello, World!')

**Мемоізація** - це метод, що використовується у обчисленнях для прискорення програм. При цьому зберігається результат виконання функції та цей результат використовується при наступному дзвінку.

Візьмемо рекурсивну функцію обчислення числа Фібоначчі:

def fib (n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return fib (n-1) + fib (n-2)

Для оптимізації подібного алгоритму добре підходить метод мемоізації, тобто збереження та повторне використання ранніх обчислених значень:

\_fib\_cache = {1: 1, 2: 1}

def mem\_fib(n):

result = \_fib\_cache.get(n)

if result is None:

result = mem\_fib(n-2) + mem\_fib(n-1)

\_fib\_cache[n] = result

return result

Проблема в тому, що ми змінили код рекурсивної функції *fib*. Наступний код не змінює нашу функцію *fib*, тому його ясність і зручність читання не порушені:

def memoize(f):

memo = {}

def helper(x):

if x not in memo:

memo[x] = f(x)

return memo[x]

return helper

@memoize

def fib(n):

if n == 0:

return 0

elif n == 1:

return 1

else:

return fib(n - 1) + fib(n - 2)

**Каррування** - це перетворення функції від багатьох аргументів на набір функцій, кожна з яких є функцією від одного аргументу. Ми можемо передати частину аргументів у функцію і отримати назад функцію, яка чекає на інші аргументи.

Створимо просту функцію *greet*, яка приймає як аргументи привітання та ім'я:

def greet(greeting, name):

print(greeting + ', ' + name)

greet('Hello', 'Nikita')

Невелике покращення дозволить нам створити нову функцію для будь-якого типу привітання та передати цій новій функції ім'я:

def greet\_curried(greeting):

def greet(name):

print(greeting + ', ' + name)

return greet

greet\_hello = greet\_curried('Hello')

greet\_hello('Nikita')

greet\_hello('Denis')

greet\_curried('Hi')('Andrey')

Або варіант з *lambda*:

greet\_curried = lambda gretting: lambda name: print(f'{gretting}, {name}!')

Або варіант із частковим застосуванням:

from functools import partial

def greet(greeting, name):

print(f'{greeting}, {name}!')

newfunc = partial(greet, greeting='Hello')

newfunc(name='Nikita')

newfunc(name='Kyrylo')

**1 бал Дескриптори** - це об'єкти Python, які реалізують метод протоколу дескрипторів, який дає можливість створювати об'єкти, які мають особливу поведінку, коли до них звертаються як до атрибутів інших об'єктів.

Давайте розглянемо приклад:

class Verbose\_attribute():

def \_\_get\_\_(self, obj, type=None) -> object:

print("accessing the attribute to get the value")

return 42

def \_\_set\_\_(self, obj, value) -> None:

print("accessing the attribute to set the value")

raise AttributeError("Cannot change the value")

class Foo():

attribute1 = Verbose\_attribute()

my\_foo\_object = Foo()

x = my\_foo\_object.attribute1

print(x)

У наведеному вище прикладі *Verbose\_attribute()* реалізує протокол дескриптора. Як тільки він створений як атрибут *Foo*, його можна розглядати як дескриптор.

Якщо ви хочете використовувати дескриптори Python у своєму коді, вам просто потрібно реалізувати протокол дескрипторів. Найбільш важливими методами цього протоколу є *.\_\_get\_\_()* і *.\_\_set\_\_()*, що мають наступну сигнатуру:

\_\_get\_\_(self, obj, type=None) -> object

\_\_set\_\_(self, obj, value) -> None

При реалізації протоколу майте на увазі наступне:

* *self* – це екземпляр дескриптора, який ви пишете.
* *obj* – це екземпляр об'єкта, до якого прикріплений ваш дескриптор.
* *type* – це тип об'єкта, до якого прикріплений дескриптор.

Ще одна важлива річ, яку слід знати, - це те, що дескриптори Python створюються тільки один раз для кожного класу. Це означає, що кожен екземпляр класу, що містить дескриптор, розділяє цей екземпляр дескриптора. Це те, чого ви не очікуєте і може призвести до класичної помилки, наприклад:

class OneDigitNumericValue():

def \_\_init\_\_(self):

self.value = 0

def \_\_get\_\_(self, obj, type=None) -> object:

return self.value

def \_\_set\_\_(self, obj, value) -> None:

if value > 9 or value < 0 or int(value) != value:

raise AttributeError("The value is invalid")

self.value = value

class Foo():

number = OneDigitNumericValue()

my\_foo\_object = Foo()

my\_second\_foo\_object = Foo()

my\_foo\_object.number = 3

print(my\_foo\_object.number)

print(my\_second\_foo\_object.number)

my\_third\_foo\_object = Foo()

print(my\_third\_foo\_object.number)



Тут у вас є клас, *Foo* який визначає атрибут номер, який є дескриптором. Цей дескриптор набуває однозначного числового значення і зберігає його у властивості самого дескриптора. Однак цей підхід не спрацює, тому що кожен екземпляр *Foo* використовує один і той самий екземпляр дескриптора. По суті ви створили просто новий атрибут рівня класу.

Отже, як можна вирішити цю проблему? Ви можете подумати, що було б добре використовувати словник для збереження всіх значень дескриптора для всіх об'єктів, до яких він прикріплений. Це здається хорошим рішенням , так *.\_\_get\_\_()* і *.\_\_set\_\_()* мати *obj* атрибут, який є екземпляром об'єкта ви прикріплені. Це значення можна використовувати як ключ для словника.

class OneDigitNumericValue():

def \_\_init\_\_(self):

self.value = {}

def \_\_get\_\_(self, obj, type=None) -> object:

try:

return self.value[obj]

except:

return 0

def \_\_set\_\_(self, obj, value) -> None:

if value > 9 or value < 0 or int(value) != value:

raise AttributeError("The value is invalid")

self.value[obj] = value

class Foo():

number = OneDigitNumericValue()

my\_foo\_object = Foo()

my\_second\_foo\_object = Foo()

my\_foo\_object.number = 3

print(my\_foo\_object.number)

print(my\_second\_foo\_object.number)

my\_third\_foo\_object = Foo()

print(my\_third\_foo\_object.number)



На жаль, недоліком є те, що дескриптор зберігає сильне посилання на об'єкт-власник. Це означає, що якщо ви знищите об'єкт, пам'ять не буде звільнена, тому що збирач сміття продовжує знаходити посилання на цей об'єкт усередині дескриптора!

Ви можете подумати, що рішенням може бути використання слабких посилань. Хоча це може бути вам доведеться мати справу з тим фактом, що не все може вважатися слабким і що, коли ваші об'єкти збираються, вони зникають з вашого словника.

Найкраще рішення тут - просто не зберігати значення в самому дескрипторі, а зберігати їх в об'єкті, до якого прикріплений дескриптор. Спробуйте наступний підхід:

class OneDigitNumericValue():

def \_\_set\_name\_\_(self, owner, name):

self.name = name

def \_\_get\_\_(self, obj, type=None) -> object:

return obj.\_\_dict\_\_.get(self.name) or 0

def \_\_set\_\_(self, obj, value) -> None:

obj.\_\_dict\_\_[self.name] = value

class Foo():

number = OneDigitNumericValue()

my\_foo\_object = Foo()

my\_second\_foo\_object = Foo()

my\_foo\_object.number = 3

print(my\_foo\_object.number)

print(my\_second\_foo\_object.number)

my\_third\_foo\_object = Foo()

print(my\_third\_foo\_object.number)

Тепер *.\_\_init\_\_()* був видалений і *.\_\_set\_name\_\_()* реалізований. Це дозволяє створити дескриптор без вказівки імені внутрішнього атрибута, який потрібно використовувати для збереження значення. Ваш код тепер виглядає краще та чистіше!