**Лабораторная работа № 2**

**Комплект 1: Начало использования Closures, Decorators, Logging, Unittests.**

1.1: Создайте простое замыкание (closure) в виде внутренней (вложенной) функции внутри обычной функции. Внутренняя функция (замыкание, closure) должна использовать переменные и аргументы обычной функции, в которую она вложена. Внутри внутренней функции (closure) распечатайте переданные аргументы в терминале. Верните вложенную функцию из обычной функции с помощью выражения return.

**Код программы:**

def func():  
 *'''Обычная функция для создания замыкания.  
 Аргументы:  
 name - некое имя;  
 birthday - некая дата рождения;  
 place - место рождения.  
 Функция возвращает внутреннюю функцию.  
 '''* name = 'Natalya'  
 birthday = '31.03.1789'  
 place = 'Russia'  
  
 def func\_second():  
 *'''Внутренняя функция для создания замыкания.  
 Возвращает словарь с именем, датой рождения и местом рождения.  
 '''* return {'name': name, 'birthday': birthday, 'place': place}  
  
 return func\_second  
  
  
print(func()())

**Результат программы:**

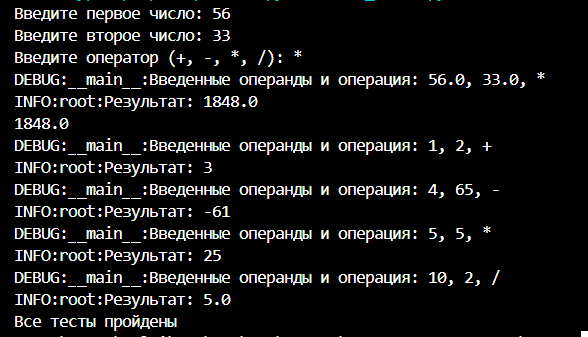
****

1.2: Изучите на примерах в интернете, что такое closure и и как их применять для создания простого декоратора (decorator) с @-синтаксисом в Python. Модернизируйте калькулятор из задачи 3.1 лабораторной работы №1. Декорируйте вашу функцию calculate. В соответствующем декорирующем замыкании, в сlosure, то есть во внутренней функции используйте простое логирование (стандартный модуль Python logging). Сделайте логирование внутри замыкания до вызова вашей функции calculate(operand1, operand2, action), в котором логируется информация о том какие операнды и какая арифметическая операция собираются поступить на вход функции calculate(operand1, operand2, action). Затем внутри того же closure следует сам вызов функции calculate(...). А затем, после этого вызова должно быть снова логирование, но уже с результатом выполнения вычисления, проделанного в этой функции.

**Код программы:**

import logging  
  
logging.basicConfig(level=logging.DEBUG)  
  
logger = logging.getLogger(\_\_name\_\_)  
  
def log\_decorator(func):  
 *'''Декорирование функции и его переменных.  
 В декорирующем замыкании происходит логирование до и после вызова функции.  
 '''* def log(oper1, oper2, operator):  
 logger.debug(f'Введенные операнды и операция: {oper1}, {oper2}, {operator}')  
 res = func(oper1, oper2, operator)  
 logging.info(f'Результат: {res}')  
 return res  
 return log  
  
@log\_decorator  
def calculate(a, b, operator):  
 *''' Выполняет определенную арифметическую операцию над двумя числами.  
 Если заданный операнд равен предложенным, то выполняется соответствующая операция и возвращается результат вычисления.  
 Аргументы:  
 a - первое числло.  
 b - второе число.  
 operator - оператор, который задается и используется для выполнения операции.  
 return - возвращает результат выполнения арифметической операции.  
 '''* if operator == '+':  
 return a + b  
 elif operator == '-':  
 return a - b  
 elif operator == '\*':  
 return a \* b  
 elif operator == '/':  
 if b == 0:  
 return 'На ноль делить нельзя!'  
 return a / b  
 else:  
 return 'Недопустимый оператор'  
  
  
def test\_calc():  
 *'''Проводит тесты на определенных примерах'''* assert calculate(1, 2, '+') == 3  
 assert calculate(4, 65, '-') == -61  
 assert calculate(5, 5, '\*') == 25  
 assert calculate(10, 2, '/') == 5  
 print('Все тесты пройдены')  
  
  
def main():  
 *'''Основная функция.  
 Аргументы:  
 number1 - первое числло.  
 number2 - второе число.  
 operator - оператор, который задается и используется для выполнения операции.  
 '''* number1 = float(input('Введите первое число: '))  
 number2 = float(input('Введите второе число: '))  
 operator = str(input('Введите оператор (+, -, \*, /): '))  
  
 print(calculate(number1, number2, operator))  
  
main()  
test\_calc()

**Результат программы:**



1.3: Изучите основы каррирования. Каррирование в самом простом варианте - это создание специализированной функции на основе более общей функции с предустановленными параметрами для этой более общей функции. Реализуйте каррирование на примере вычисления количества радиоактивного вещества N, оставшегося в некоторый 1 момент времени t от радокактивного вещества с периодом полураспада t1/2, если изначально это количество было равно N0. Закон распада задан формулой:

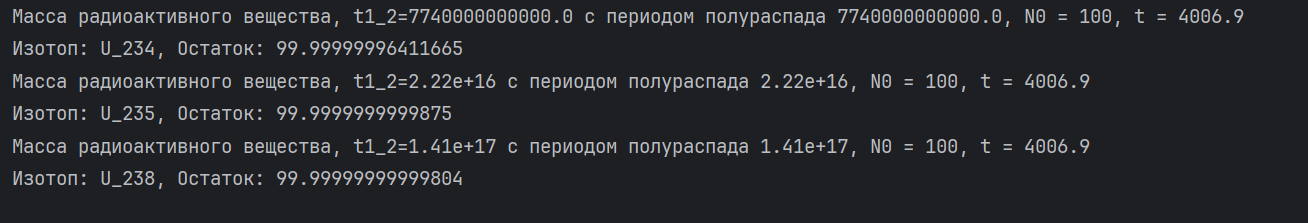
(1)

В качестве проставленного заранее параметра в данном примере должно быть значение периода полураспада t1/2, которое постоянно для каждого типа радиоактивного материала (радиоактивного изотопа химического элемента). Сделайте словарь, где в качестве колючей используются строки с символами радиоактивных изотопов, а в качестве значений им сопоставлены каррированные с характерными периодами полураспада. В основном коде вашей программы организуйте цикл по этому словарю и продемонстрируйте в нём вызовы каррированных функций с распечаткой на экране сколько вещества осталось от одного и того же N0 в некоторый момент времени t в зависимости от типа изотопа.

**Код программы:**

from functools import partial  
  
t1\_2\_elems = {"U\_234": 7.74 \* 10\*\*12, "U\_235": 2.22 \* 10\*\*16, "U\_238": 1.41 \* 10\*\*17}  
radioactive\_funcs = {"U\_234": None, "U\_235": None, "U\_238": None}  
  
def decay\_amount(N0, t, t1\_2):  
 *'''Вычисляет количество радиоактивного вещества, оставшегося в некоторый момент t по закону распада.  
 Аргументы:  
 N0 - изначальное количества вещества;  
 t - некоторый момент времени;  
 t1\_2 - период полураспада вещества.  
 '''* N = N0 \* (1/2) \*\* (t/t1\_2)  
 res = "Масса радиоактивного вещества, t1\_2=" + str(t1\_2)  
 print(f'{res} с периодом полураспада {t1\_2}, N0 = {N0}, t = {t}')  
 return N  
  
f1 = partial(decay\_amount, t1\_2 = t1\_2\_elems['U\_234'])  
f2 = partial(decay\_amount, t1\_2 = t1\_2\_elems['U\_235'])  
f3 = partial(decay\_amount, t1\_2 = t1\_2\_elems['U\_238'])  
  
def main():  
 *'''Основной код программы, где вызываются каррированные функции.  
 И организован цикл по словарю с распечаткой на экране сколько вещества осталось от одного и того же N0 в некоторый момент времени t.  
 '''* N0 = 100  
 t = 4006.9  
 radioactive\_funcs["U\_234"] = f1  
 radioactive\_funcs["U\_235"] = f2  
 radioactive\_funcs["U\_238"] = f3  
  
 for isotope, func in radioactive\_funcs.items():  
 result = func(N0, t)  
 print(f'Изотоп: {isotope}, Остаток: {result}')  
  
  
main()

**Результат программы:**



1.4: Напишите unit-тесты для калькулятора из задачи 3.1 лабораторной работы № 1 используя стандартный модуль unittest библиотеки Python. Базовый пример: <https://docs.python.org/3/library/unittest.html#basic-example>

Затем перепешите теже тесты с использованием пакета pytest. Ссылка на сайт библиотеки с базовым примером: <https://pytest.org/en/7.2.x/>

**Код программы:**

Задача 1 из Лабораторной работы № 1

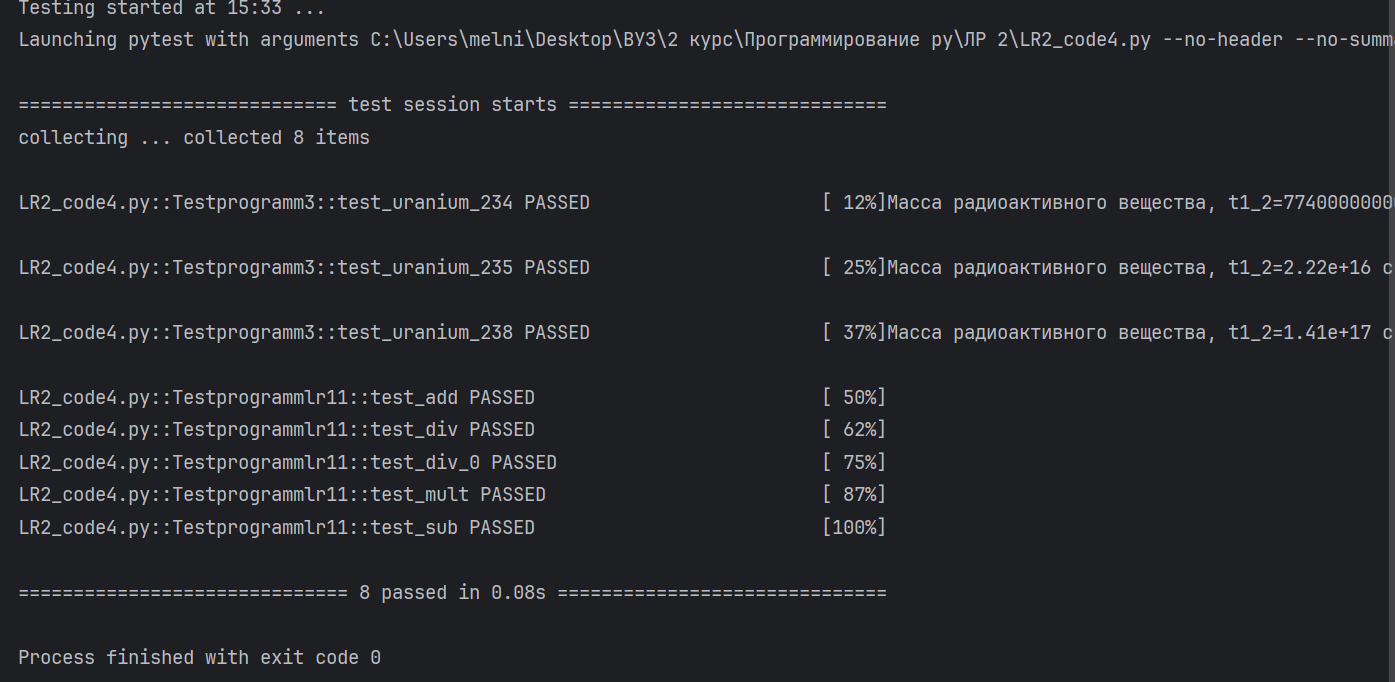
def calculate(a: float, b: float, operator: str):  
 *''' Выполняет определенную арифметическую операцию над двумя числами.  
 Если заданный операнд равен предложенным, то выполняется соответствующая операция и возвращается результат вычисления.  
 Аргументы:  
 a - первое числло.  
 b - второе число.  
 operator - оператор, который задается и используется для выполнения операции.  
 return - возвращает результат выполнения арифметической операции.  
 '''* if operator == '+':  
 return a + b  
 elif operator == '-':  
 return a - b  
 elif operator == '\*':  
 return a \* b  
 elif operator == '/':  
 if b == 0:  
 return 'На ноль делить нельзя!'  
 return a / b  
 else:  
 return 'Недопустимый оператор'  
  
  
def test\_add():  
 *'''Проверяет функцию calculate() на сложения чисел через тестирование.'''* assert calculate(1, 2, '+') == 3  
 return 'test passed'  
  
  
def test\_sub():  
 *'''Проверяет функцию calculate() на вычитание чисел через тестирование.'''* assert calculate(4, 65, '-') == -61  
 return 'test passed'  
  
  
def test\_mult():  
 *'''Проверяет функцию calculate() на умножение чисел через тестирование.'''* assert calculate(5, 5, '\*') == 25  
 return 'test passed'  
  
  
def test\_div():  
 *'''Проверяет функцию calculate() на деление чисел через тестирование.'''* assert calculate(10, 2, '/') == 5  
 return 'test passed'  
  
  
def main():  
 *'''Основная функция.  
 Аргументы:  
 number1 - первое числло.  
 number2 - второе число.  
 operator - оператор, который задается и используется для выполнения операции.  
 '''* # number1 = float(input('Введите первое число: '))  
 # number2 = float(input('Введите второе число: '))  
 # operator = str(input('Введите оператор: '))  
  
 # print(calculate(number1, number2, operator))  
 print(test\_add())  
 print(test\_sub())  
 print(test\_div())  
 print(test\_mult())  
  
main()

1.4.

import unittest  
import LR2\_code3  
import LR1\_3\_1  
  
  
class Testprogramm3(unittest.TestCase):  
 *'''Тесты для задания 3, в которой тестируются функции радиоактивного распада, взятые из модуля LR2\_code3.py.'''* def test\_uranium\_234(self):  
 self.assertEqual(LR2\_code3.radioactive\_funcs["U\_234"](100, 3850.9), 99.99999996551368)  
  
 def test\_uranium\_235(self):  
 self.assertEqual(LR2\_code3.radioactive\_funcs["U\_235"](100, 3980.9), 99.99999999998757)  
  
 def test\_uranium\_238(self):  
 self.assertEqual(LR2\_code3.radioactive\_funcs["U\_238"](100, 4006.9), 99.99999999999804)  
  
  
class Testprogrammlr11(unittest.TestCase):  
 *'''Тесты для задания 1 ЛР 1, в которой тестируются функции для тестирования арефмитических действий, взятые из модуля LR1\_3\_1.py.'''* def test\_add(self):  
 self.assertEqual(LR1\_3\_1.calculate(1, 2, '+'), 3)  
  
 def test\_sub(self):  
 self.assertEqual(LR1\_3\_1.calculate(4, 65, '-'), -61)  
  
 def test\_mult(self):  
 self.assertEqual(LR1\_3\_1.calculate(5, 5, '\*'), 25)  
  
 def test\_div(self):  
 self.assertEqual(LR1\_3\_1.calculate(10, 2, '/'), 5)  
  
 def test\_div\_0(self):  
 self.assertEqual(LR1\_3\_1.calculate(10, 0, '/'), 'На ноль делить нельзя!')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

**Результат программы:**

Unit-тест



pytest

