**Программирование. Язык Python.**

**Лабораторная работа № 2. Задачи. 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 foo():  
 name = 'Ann'  
  
 surname = 'Stepanova'  
  
 patronymic = 'Andreevna'  
  
 birthday = '20.04.2005'  
  
 def inner\_boo():  
  
 return {  
 'name': name**,** 'surname': surname**,** 'patronymic': patronymic**,** 'birthday': birthday  
 }  
  
 return inner\_boo  
  
  
print(foo()())

**Результат:**

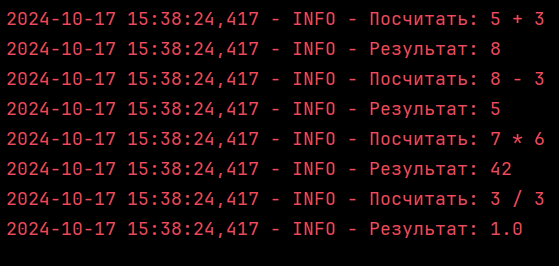
****

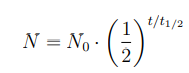
1.2: Изучите на примерах в интернете, что такое closure и и как их применять для создания простого декоратора (decorator) с @-синтаксисом в Python. Модернизируйте калькулятор из задачи 3.1 лабораторной работы №1. Декорируйте вашу функцию calculate. В соответствующем декорирующем замыкании, в сlosure, 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  
  
# Настройка логирования  
logging.basicConfig(level=logging.INFO**,** format='%(asctime)s - %(levelname)s - %(message)s')  
  
  
def log\_decorator(func):  
  
 def wrapper(a**,** b**,** operand):  
 # Логирование входных данных  
 logging.info(f"Посчитать: {a} {operand} {b}")  
  
 # Вызов оригинальной функции  
 result = func(a**,** b**,** operand)  
  
 # Логирование результата  
 logging.info(f"Результат: {result}")  
  
 return result  
  
 return wrapper  
  
  
@log\_decorator  
def calculate(a**,** b**,** operand):  
 *"""  
 Функция вычисления  
  
 Параметры:  
 a (int): Первое число  
 b (int): Второе число  
 operand (str): Операция над числами  
  
 Возвращает:  
 int: Результат вычиислений  
 или str: сообщение об ошибке при делении на ноль  
  
 """* if operand == '+':  
 return a + b  
 if operand == '-':  
 return a - b  
 if operand == '\*':  
 return a \* b  
 elif operand == '/':  
 if b == **0**:  
 return 'Ошибка: на ноль делить нельзя'  
 else:  
 return a / b  
  
  
def test\_calculator():  
 *"""  
 Функция для тестирования функции calculate  
  
 Проверяет корректность выполнения операций сложения, вычитания,  
 умножения и деления, а также обработку ошибок  
  
 """* assert calculate(**5, 3,** '+') == **8** assert calculate(**8, 3,** '-') == **5** assert calculate(**7, 6,** '\*') == **42** assert calculate(**3, 3,** '/') == **1**test\_calculator()

**Результат:**

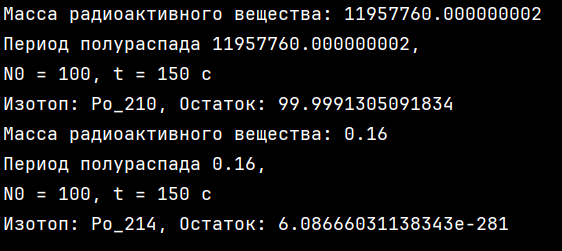


1.3: Изучите основы каррирования. Каррирование в самом простом варианте - это создание специализированной функции на основе более общей функции с предустановленными параметрами для этой более общей функции. Реализуйте каррирование на примере вычисления количества радиоактивного вещества N, оставшегося 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:

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  
  
# Периоды полураспада в секундах  
half\_per = {  
 "Po\_210": **138.4** \* **24** \* **3600,** "Po\_214": **0.16,**}  
  
radioactive\_funcs = {"Po\_210": None**,** "Po\_214": None}  
  
  
def decay\_amount(N0: float**,** t: int**,** t1\_2 = None):  
  
 *"""  
 Функция закона полураспада  
  
 Параметры:  
 N0 (float): изначальное количества вещества  
 t (int): некоторый момент времени  
 t1\_2 (NoneType): период полураспада вещества  
  
 """* N = N0 \* (**1** / **2**) \*\* (t / t1\_2)  
  
 #res = 'Масса радиоактивного вещества:' + str(t1\_2)  
 print(f'Масса радиоактивного вещества: {str(t1\_2)} \nПериод полураспада {t1\_2}, \nN0 = {N0}, t = {t} с')  
 return N  
  
  
f1 = partial(decay\_amount**,** t1\_2 = half\_per['Po\_210'])  
f2 = partial(decay\_amount**,** t1\_2 = half\_per['Po\_214'])  
  
  
def main():  
  
 *"""  
 Вызов каррирования функции  
 Создан цикл по словарю с распечаткой на экране сколько вещества осталось от одного и того же N0 в момент времени t  
 """* N0 = **100** t = **150** radioactive\_funcs["Po\_210"] = f1  
 radioactive\_funcs["Po\_214"] = f2  
  
 for isotope**,** func in radioactive\_funcs.items():  
 result = func(N0**,** t)  
 print(f'Изотоп: {isotope}, Остаток: {result}')  
  
  
main()

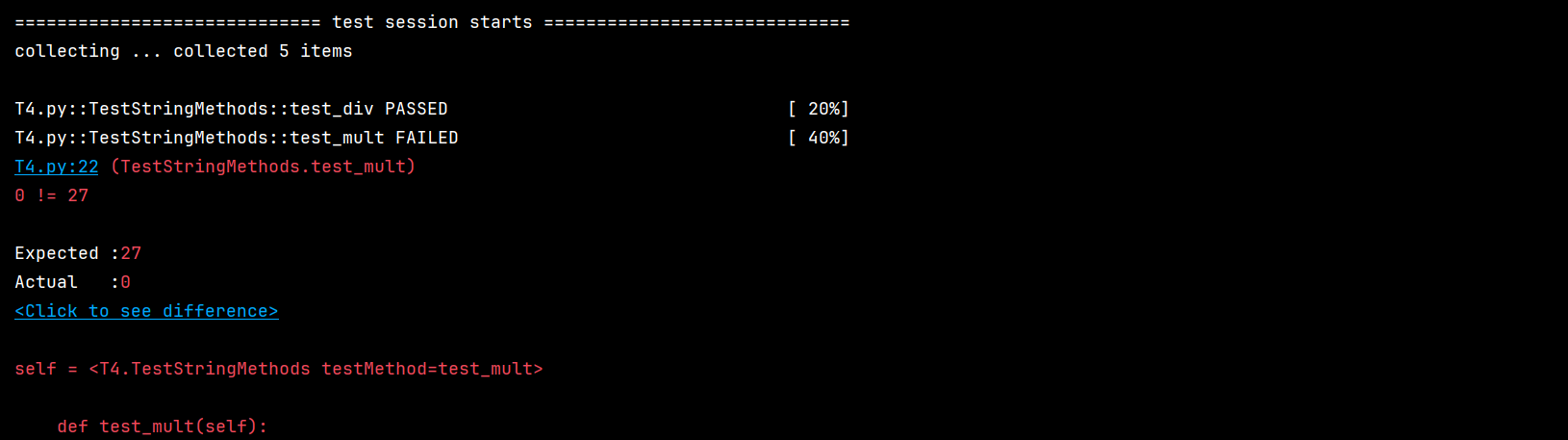
**Результат:**

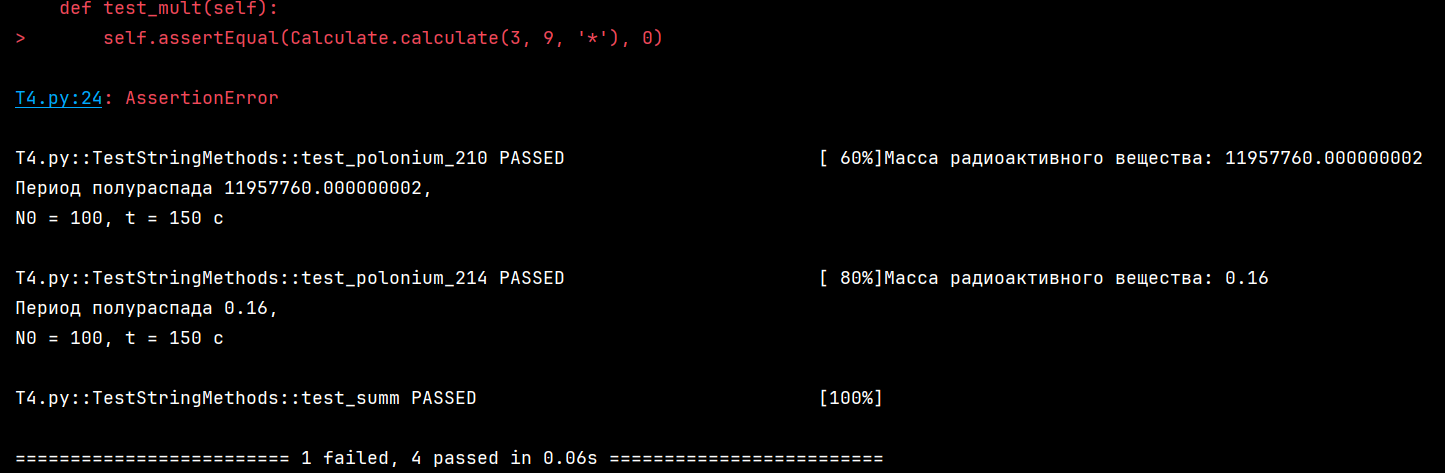
1.4: Напишите unit-тесты для калькулятора из задачи 3.1 лабораторной работы № 1 используя стандартный модуль unittest библиотеки Python. Затем перепешите теже тесты с использованием пакета pytest.

**Код программы:**

import unittest  
import Calculate  
import T3  
  
  
class TestStringMethods(unittest.TestCase):  
  
  
 def test\_polonium\_210(self):  
 self.assertEqual(T3.radioactive\_funcs["Po\_210"](**100, 150**)**, 99.9991305091834**)  
  
 def test\_polonium\_214(self):  
 self.assertEqual(T3.radioactive\_funcs["Po\_214"](**100, 150**)**, 6.08666031138343e-281**)  
  
  
  
 def test\_summ(self):  
 self.assertEqual(Calculate.calculate(**5, 3,** '+')**, 8**)  
  
 def test\_div(self):  
 self.assertEqual(Calculate.calculate(**3, 3,** '/')**, 1**)  
  
 def test\_mult(self):  
 self.assertEqual(Calculate.calculate(**3, 9,** '\*')**, 0**)  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

**Результат:**

****В тесте калькулятора специально была сделана ошибка

****