Урок 5. Проектная деятельность. Работа с декораторами.

ФИО_____

Тест:

1. *Выберите верные утверждения про popitem()?*
   A) Удаляет первый вставленный B) Последний вставленный C) Случайный D) Не существует
2. *Что выведет данная программа?*

```
import json
json.dumps({(1,2): "p"})
```

   _____*ошибка*_____
3. *Что делает nonlocal?*
   A) создаёт глобальную переменную B) читает builtins C) Даёт доступ к значению переменной внутри вложенной функций D) ничего
4. *Что делает блок* `if __name__ == "__main__":`?

   _____
5. *Как называется данный процесс?*

```
def decorator_function(func):  1 usage
    def wrapper():
        print('Функция-обёртка!')
        print('Оборачиваемая функция: {}'.format(func))
        print('Выполняем обёрнутую функцию...')
        func()
        print('Выходим из обёртки')
    return wrapper

@decorator_function
def hello_world():
    print('Hello world!')
```

   _____

Теория:

Декораторы как функции:

Декоратор – это функция, которая добавляет новую функциональность к другой функции без изменения её кода. Он как бы оборачивает, декорирует функцию, тем самым расширяя её возможности.

Он работает так:

1. вы пишете функцию-помощника (декоратор), которая принимает другую функцию и возвращает **новую** функцию;
2. эта новая функция называется **wrapper** («обёртка»): внутри неё вы делаете «что-то до», вызываете исходную функцию, потом делаете «что-то после»;
3. синтаксис @decorator просто приклеивает эту «обёртку» к вашей функции.

```
@decorator
def func(...):
    ...
# Эквивалентно: func = decorator(func)

def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase

    return wrapper
```

- В первой строке мы указываем имя декоратора и то, что он принимает function в качестве своей переменной.
- Вторая строка — это объявление функции-обёртки wrapper (). Тело обёртки в блоке состоит из трёх строк ниже. Оно как раз и описывает, что именно мы будем делать с функцией, ранее принятой декоратором.
- Третья строка: записываем входящую переменную function() в локальную переменную func. Здесь «локальная» означает, что она действует только в рамках функции wrapper ().
- Четвёртая строка: мы применяем к func строковый метод upper и записываем результат в другую локальную переменную make_uppercase.
- Пятая строка: функция wrapper () возвращает переменную make_uppercase, то есть строку от function (), но уже прописными буквами.
- Последняя строка: декоратор возвращает нам уже саму функцию wrapper, точнее, результат её работы над функцией function.

```
# Применение декоратора
@uppercase_decorator
def say_hi():
    return 'всем привет'
```

Зачем это нужны декораторы: логирование, замер времени, проверка аргументов, кэширование, «выполнить один раз», «разрешено/запрещено», и т.п. — один раз пишете декоратор, потом ставите @... над любыми функциями.

```
# логирование
def log(func):
    def wrapper(*args, **kwargs):
        print(f"[LOG] start {func.__name__}")
        result = func(*args, **kwargs)
        print(f"[LOG] end   {func.__name__}")
        return result
    return wrapper

@log
def add(a, b):
    return a + b

# возврат времени работы функции
import time
def timeit(func):
    def wrapper(*args, **kwargs):
        t0 = time.perf_counter()
        result = func(*args, **kwargs)
        dt = (time.perf_counter() - t0) * 1000
        print(f"{func.__name__}: {dt:.2f} ms")
        return result
    return wrapper
@log
@timeit
def add(a, b):
    return a + b
```

Порядок вызова нескольких декораторов:

```
@A
@B
def f(): ...
# Эквивалентно: f = A(B(f))  # сначала применится B, потом A
```

Параметризированный декоратор

Обычный декоратор принимает функцию и возвращает «обёртку».
Параметризованный декоратор сначала принимает **настройки**, а уже потом — функцию. Поэтому у него **три слоя**:

```
def deco_factory(option1, option2=...):    # 1) фабрика: принимает
параметры декоратора
    def decorator(func):                    # 2) сам декоратор:
принимает функцию
        def wrapper(*args, **kwargs):       # 3) обёртка: принимает
аргументы функции
            # можно использовать option1/option2
            result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator
# Использование
@deco_factory(option1=123)
```

```
def my_func(...):
    ...
# Эквивалентно: my_func = deco_factory(option1=123)(my_func)
```

Задачи(где не указаны конкретные функции, то берите задачи с прошлого урока и декорируйте их):

1. **@log_call**
   Напиши декоратор, который печатает строку CALL func_name(args, kwargs) перед вызовом функции.
2. **@timeit_ms**
   Декоратор, который после вызова печатает func_name: X.XX ms. Используй time.perf_counter().
3. **@once**
   Декоратор, который выполняет функцию только первый раз. Дальше возвращает **первый** результат без повторного вычисления.
4. **@timeit**
   Реализуйте декоратор, который печатает время выполнения (в миллисекундах) и возвращает результат.

   ```
   @timeit

   def work(n):

       s = 0

       for i in range(n):

           s += i*i

       return s

   # work(100000) -> печатает "work: 2.3 ms", возвращает сумму
   ```

5. **@debug_result**
   Декоратор, который печатает: func_name(...) -> RESULT, где RESULT — то, что вернула функция. Потом возвращает RESULT.
6. **@slowdown(sec)** *(параметризованный)*
   Декоратор «замедлитель»: перед вызовом делает time.sleep(sec), затем вызывает функцию.