Магистрант 1 курса ML&DS Сатыбалды Касымжомарт

📌 **Задачи**

1. Итераторы и генераторы

Свой итератор

Реализуй класс-итератор Countdown, который принимает число n и при итерации выводит числа от n до 1.

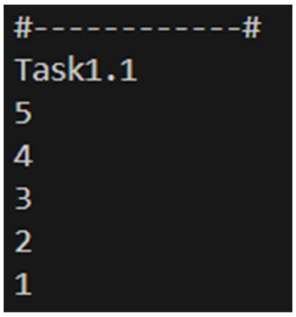
```
for num in Countdown(5):
   print(num)
# 5 4 3 2 1
```

Реализация

```
# Task 1.1
print("#------------#\nTask1.1")
class Countdown:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        self.current = self.n
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        val = self.current
        self.current -= 1
        return val


for num in Countdown(5):
    print(num)
```

При for num in Countdown(5) цикл вызывает __iter__() один раз, а потом многократно __next__(). Когда StopIteration выбрасывается, цикл останавливается.

Результат



Генератор чётных чисел

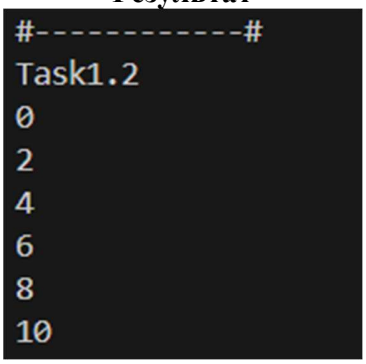
Напиши генератор even_numbers(limit), который выдаёт чётные числа до limit.

Реализация

```
#Task 1.2
print("#-----------#\nTask1.2")
def even_numbers(limit):
    for i in range(0, limit + 1, 2):
        yield i

for num in even_numbers(11):
    print(num)
```

Числа появляются по мере вызова next(). Устанавливается шаг 2.

Результат



Бесконечный генератор

Реализуй генератор infinite_cycle(lst), который по кругу перебирает элементы списка.

cycle = infinite_cycle([1, 2, 3])
next(cycle) → 1
next(cycle) → 2
next(cycle) → 3

next(cycle) → 1

Реализация

```python
#Task 1.3
print("#------------#\nTask1.3")
def infinite_cycle(lst):
    while True:
        for item in lst:
            yield item

cycle = infinite_cycle([1, 2, 3])
print(next(cycle))
print(next(cycle))
print(next(cycle))
print(next(cycle))
print(next(cycle))
```

Работает бесконечно, поэтому можно только брать next(cycle) или обрывать for вручную.

Результат

```
#------------#
Task1.3
1
2
3
1
2
```

2. Декораторы и метаклассы

Простой декоратор

Напиши декоратор logger, который перед вызовом функции печатает её имя.

```
@logger
def hello():
    print("Привет")
# Вызов → "Вызов функции hello" + "Привет"
```

Реализация

```python
#Task 2.1
print("#------------#\nTask2.1")
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"Вызов функции {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@logger
def hello():
    print("Привет")

hello()
```

@logger заменяет функцию на обёртку, которая добавляет логи и затем вызывает оригинальную функцию.

Результат

```
#------------#
Task2.1
Вызов функции hello
Привет
```

Декоратор с параметром

Напиши декоратор repeat(n), который повторяет выполнение функции n раз.

```
@repeat(3)
def hi():
    print("Hi!")
# Выведет "Hi!" три раза
```

Реализация

```python
#Task 2.2
print("#------------#\nTask2.2")
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def hi():
    print("Hi!")

hi()
```

Здесь декоратор с параметрами. @repeat(3) сначала передаёт 3 в repeat, а тот уже возвращает обычный декоратор.

Результат

```
#------------#
Task2.2
Hi!
Hi!
Hi!
```

Метакласс

Создай метакласс AutoStr, который добавляет всем классам метод __str__, выводящий название класса и словарь атрибутов.

```
class Person(metaclass=AutoStr):
   def __init__(self, name, age):
       self.name = name
       self.age = age

print(Person("Alex", 20))
# Person {'name': 'Alex', 'age': 20}
```

Реализация

```python
#Task 2.3
print("#------------#\nTask2.3")
class AutoStr(type):
    def __new__(cls, name, bases, dct):
        if "__str__" not in dct:
            def __str__(self):
                return f"{self.__class__.__name__} {self.__dict__}"
            dct["__str__"] = __str__
        return super().__new__(cls, name, bases, dct)

class Person(metaclass=AutoStr):
    def __init__(self, name, age):
        self.name = name
        self.age = age

print(Person("Alex", 20))
```

Метакласс управляет тем, как создаются классы. Если у класса нет __str__, он автоматически добавляется.

Результат

```
#------------#
Task2.3
Person {'name': 'Alex', 'age': 20}
```

3. Модульность и пакеты

1. **Создай модуль**

Раздели программу на два файла:

- math_utils.py с функциями add(a, b) и mul(a, b)
- main.py, который импортирует и использует эти функции.

Реализация

```python
# math_utils.py
def add(a, b):
    return a + b

def mul(a, b):
    return a * b
```

```python
#Task 3.1
print("#------------#\nTask3.1")
from math_utils import add, mul

print(add(2, 3))
print(mul(4, 5))
```

math_utils.py содержит функции. main.py импортирует и использует их.

Результат

```
#------------#
Task3.1
5
20
```

2. **Пакет с подмодулем**

Создай пакет shapes/, в нём:

- circle.py с функцией area_circle(r)
- square.py с функцией area_square(a)

В main.py импортируй и протестируй обе функции.

Реализация

```python
> 1 > shapes > circle.py > ...
import math

def area_circle(r):
    return math.pi * r * r
```

```python
on > 1 > shapes > square.py > ...
def area_square(a):
    return a * a
```

```python
> 1 > shapes > __init__.py > ...
from .circle import area_circle
from .square import area_square

__all__ = ["area_circle", "area_square"]
```

```python
#Task 3.2
print("#------------#\nTask3.2")
from shapes import area_circle, area_square

print(area_circle(3))
print(area_square(4))
```

В shapes находятся два модуля и при создании __init__.py папка превращается в пакет. Дальше __init__.py экспортируем модули для пользования в main.

Результат

```
#------------#
Task3.2
28.274333882308138
16
```

4. Продвинутые структуры данных

Counter

Используя collections.Counter, подсчитай количество символов в строке.

Реализация

```python
#Task 4.1
print("#------------#\nTask4.1")
from collections import Counter

s = "hello world"
print(Counter(s))
```

Counter автоматически считает количество вхождений каждого символа/слова.

Результат

```
#------------#
Task4.1
Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

defaultdict

Используя defaultdict(list), сгруппируй студентов по курсам:

students = [("Иван", 1), ("Алина", 2), ("Макс", 1)]
# Результат: {1: ["Иван", "Макс"], 2: ["Алина"]}

Реализация

```python
#Task 4.2
print("#------------#\nTask4.2")
from collections import defaultdict
students = [("Иван", 1), ("Алина", 2), ("Макс", 1)]

groups = defaultdict(list)
for name, course in students:
    groups[course].append(name)

print(dict(groups))
```

defaultdict(list) автоматически создаёт пустой список, если ключа ещё нет. Дальше распределяем по курсам и выводим как словарь.

Результат

```
#------------#
Task4.2
{1: ['Иван', 'Макс'], 2: ['Алина']}
```

dataclass

Создай класс Book через @dataclass, у которого есть поля title, author, year.
Добавь несколько книг в список и отсортируй их по году.

Реализация

```python
#Task 4.3
print("#------------#\nTask4.3")
from dataclasses import dataclass

@dataclass
class Book:
    title: str
    author: str
    year: int

books = [
    Book("Война и мир", "Лев Толстой", 1869),
    Book("Мастер и Маргарита", "Михаил Булгаков", 1967),
    Book("Преступление и наказание", "Фёдор Достоевский", 1866),
    Book("Анна Каренина", "Лев Толстой", 1877),
]

books_sorted = sorted(books, key=lambda b: b.year)

for b in books_sorted:
    print(f"{b.year}: {b.title} - {b.author}")
```

dataclass автоматически создаёт __init__, __repr__, сравнение и другие методы. Можно легко хранить данные и сортировать их.

Результат

```
#------------#
Task4.3
1866: Преступление и наказание - Фёдор Достоевский
1869: Война и мир - Лев Толстой
1877: Анна Каренина - Лев Толстой
1967: Мастер и Маргарита - Михаил Булгаков
```