

Занятие 17

Декораторы

ООП

Что напечатают эти операторы?

```
some_tuple = ([1, 2], [3, 4], [5, 6])
```

```
try:
```

```
    some_tuple[2] += [7, 8]
```

```
except TypeError:
```

```
    print(some_tuple)
```

```
import re
```

```
re.findall(r'\b.*\b', 'first second third') # Жадный поиск
```

```
import re
```

```
re.findall(r'\b.*?\b', 'first second third') # Ленивый поиск
```

Задача 16-1

Напишите функцию, которая использует RE.

На вход подается строка, например, Институт точной механики оптики

На выход создается сокращение ИТМО (из первых букв, которые переведены в заглавные).

Задача 16-2

Вводится двухзначное число, например: 45.

Напишите такой шаблон, чтобы функция `re.findall` находила только те положительные целые числа, которые не больше, чем это введенное число.

Т.е. для 45 она находила бы 0, 1, 12, 45, но не 46, 100, 1000

Задача 16-3

Создайте декоратор, который переводит все слова к следующему виду, первая буква в верхнем регистре, а остальные в нижнем.

RE

Regular expressions

Замена с помощью функции

```
import re
def fullname(x):
    s = x.group() # x специальный объект, который содержит значение, индекс начала
                  # и конца
    print(x.group(), x.start(), x.end())
    match s:
        case 'Коля': return 'Николай'    # Коля 12 16
        case 'Миша': return 'Михаил'    # Миша 26 30
    #if s == 'Коля': return 'Николай'
    #elif s == 'Миша': return 'Михаил'

text = 'Здравствуй, Коля! Привет, Миша!'
print(re.sub(r"\b\w{4}\b", fullname, text))
# Здравствуй, Николай! Привет, Михаил!
```

Использование скобок в шаблоне

например, для замены местами слов

```
import re
```

```
text = 'first second'
```

```
print(re.sub(r"(first) (second)", r'\2 \1', text))
```

```
#second first
```


Некоторые полезные функции

Функция	Её смысл
<code>re.search(pattern, string)</code>	Найти в строке <code>string</code> первую строчку, подходящую под шаблон <code>pattern</code>
<code>re.fullmatch(pattern, string)</code>	Проверить, подходит ли строка <code>string</code> под шаблон <code>pattern</code>
<code>re.split(pattern, string, maxsplit=0)</code>	Аналог <code>str.split()</code> , только разделение происходит по подстрокам, подходящим под шаблон <code>pattern</code>
<code>re.findall(pattern, string)</code>	Найти в строке <code>string</code> все непересекающиеся шаблоны <code>pattern</code>
<code>re.finditer(pattern, string)</code>	Итератор всем непересекающимся шаблонам <code>pattern</code> в строке <code>string</code> (выдаются <code>match</code> -объекты);
<code>re.sub(pattern, repl, string, count=0)</code>	Заменить в строке <code>string</code> все непересекающиеся шаблоны <code>pattern</code> на <code>repl</code>

Ограничение числа замен, флаг re.I

```
import re
```

```
text = 'Программы на Java транслируются в байт-код java, выполняемый  
виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый  
код и передающей инструкции оборудованию как интерпретатор.'
```

```
res = re.sub(r'Java', r'Python', text, count=2, flags=re.I)
```

```
print(res)
```

```
# Программы на Python транслируются в байт-код Python, выполняемый  
виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый  
код и передающей инструкции оборудованию как интерпретатор.
```

Выбор части текста, используя скобки в шаблоне

```
import re
text = '123 first 234 second'
print(re.findall(r"\d+ (\w+)", text))
# ['first', 'second']
```

Если в шаблоне несколько скобок, то выдается список кортежей:

```
import re
text = 'Миша:123 Коля:234 Сеня:345'
res = re.findall(r"(\w+):(\d+)", text)
print(res)
# [('Миша', '123'), ('Коля', '234'), ('Сеня', '345')]
```

re.finditer()

```
import re  
text = 'abracadabra'  
res = re.finditer(r"a", text)  
for i in res:  
    print(i.group(), i.start(), i.end())  
  
res = re.finditer(r"[^a]", text)
```

re.split()

```
import re
```

```
text = "1    + 2222 * 3 -  7 / 2"
```

```
print(re.split(r"\s+", text))
```

```
# ['1', '+', '2222', '*', '3', '-', '7', '/', '2']
```

re.compile()

Если шаблон часто используется, то его можно скомпилировать и запомнить под каким-то именем и использовать, не переписывая шаблон с риском ошибки

```
import re  
numbers = re.compile(r'\d+')
```

Способ 1.

```
print(re.findall(numbers, 'Я живу в доме 5 в квартире 7')) #['5', '7']
```

Способ 2.

```
print(numbers.findall('Прочитайте абзац 8 на странице 356')) #['8', '356']
```

Совместное использование строк r и f

```
import re
```

```
x = 5
```

```
re.findall(fr"{x}", '112233445566') # ['5', '5']
```

```
res = '|'.join(str(i) for i in range(x)) # чему равно res?
```

```
re.findall(fr"{res}", '112233445566')
```

Выводы

1. RE – очень мощное средство
2. Необходимо очень осторожно использовать в реальных задачах
3. Очень много функциональных возможностей, которые необходимо тщательно изучить и проверить, как они реально работают.
4. Если предстоят задачи по анализу текста, то скорее всего следует освоить более плотно.
5. Если задачи не очень сложные, то обычно достаточно функций Python для работы со строками.
6. Используются во многих языках: Perl, Java, Javascript, Ruby, и др.

Задание

Напишите функцию, которая находит все слова в тексте, содержащие подслово. Под подсловом будем понимать, часть слова окруженное буквами.

Например, для текста «Косой косой косил волос на осине» для подслова «ос» функция должна вывести только первые три слова.

Декораторы

Декораторы

- Иногда нам нужно модифицировать существующую функцию, не меняя при этом ее исходный код.
- Например, оценить время выполнение функции или занести какую-то информацию о выполнении функции в файл или что-то другое
- Декоратор — это функция, которая принимает другую функцию, расширяет ее поведение, не изменяя ее явно, и возвращает новую функцию.

Для чего использовать декораторы?

- Ведение протокола операции (журналирование)
- Обеспечение контроля за доступом и аутентификацией
- Функции хронометража
- Ограничение частоты вызова API
- Кеширование и др.

Пример декоратора

```
def sample_decorator(func): # определяем декоратор
    def wrapper():
        print('Начало функции')
        func() # мы обертываем функцию func, не вмешиваясь
                # в ее работу
        print('Конец функции')
    return wrapper

def say():
    print('Привет Мир!')

say = sample_decorator(say) # декорируем функцию
say() # вызываем декорированную функцию

# При вызове функции sample_decorator(say) с переданной в качестве аргумента
# функцией say() возвращается вложенная функция wrapper() в качестве результата.
```

Декоратор, который переводит результат в верхний регистр

```
def uppercase_decorator(func):  
    def wrapper():  
        original_result = func()  
        modified_result = original_result.upper()  
        return modified_result  
    return wrapper  
  
#@uppercase_decorator  
def h():  
    return 'Hello'  
  
#h = uppercase_decorator(h)  
print(h())
```

Аргументы функций в общем виде

```
def fu(*args, **kwargs):  
    res = 0  
    for x in args:  
        if type(x) == int:  
            res += x  
    for x in kwargs:  
        if type(kwargs[x]) == int:  
            res += kwargs[x]  
    return res  
print(fu(1, 2, 'abc', a = 3, b = 'def'))
```

Выполните эту программу. Введите любой другой набор аргументов

Шаблон декоратора общего назначения

```
def decorator(func):  
    def wrapper(*args, **kwargs):  
        # Что-то выполняется до вызова декорируемой функции  
        value = func(*args, **kwargs)  
        # декорируется возвращаемое значение функции  
        # или что-то выполняется после вызова декорируемой функции  
        return value  
    return wrapper  
# Таким образом мы можем передать аргументы для функции func
```



```
import time
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        val = func(*args, **kwargs)
        end = time.perf_counter()
        work_time = end - start
        print(f'Время выполнения {func.__name__}:{round(work_time, 4)} сек.')
        return val
    return wrapper
```

```
@timer
def test(n):    return sum([(i/99)**2 for i in range(n)])
```

```
@timer
def sleep(n):  time.sleep(n)
```

```
res1 = test(10000)
res2 = sleep(4)
print(f'Результат функции test = {res1}')
print(f'Результат функции sleep = {res2}') # Выполните эту программу
```

Задание

Создайте декоратор, который принимает функцию с неограниченным количеством позиционных переменных типа `str` и склеивает их и возвращает результат.

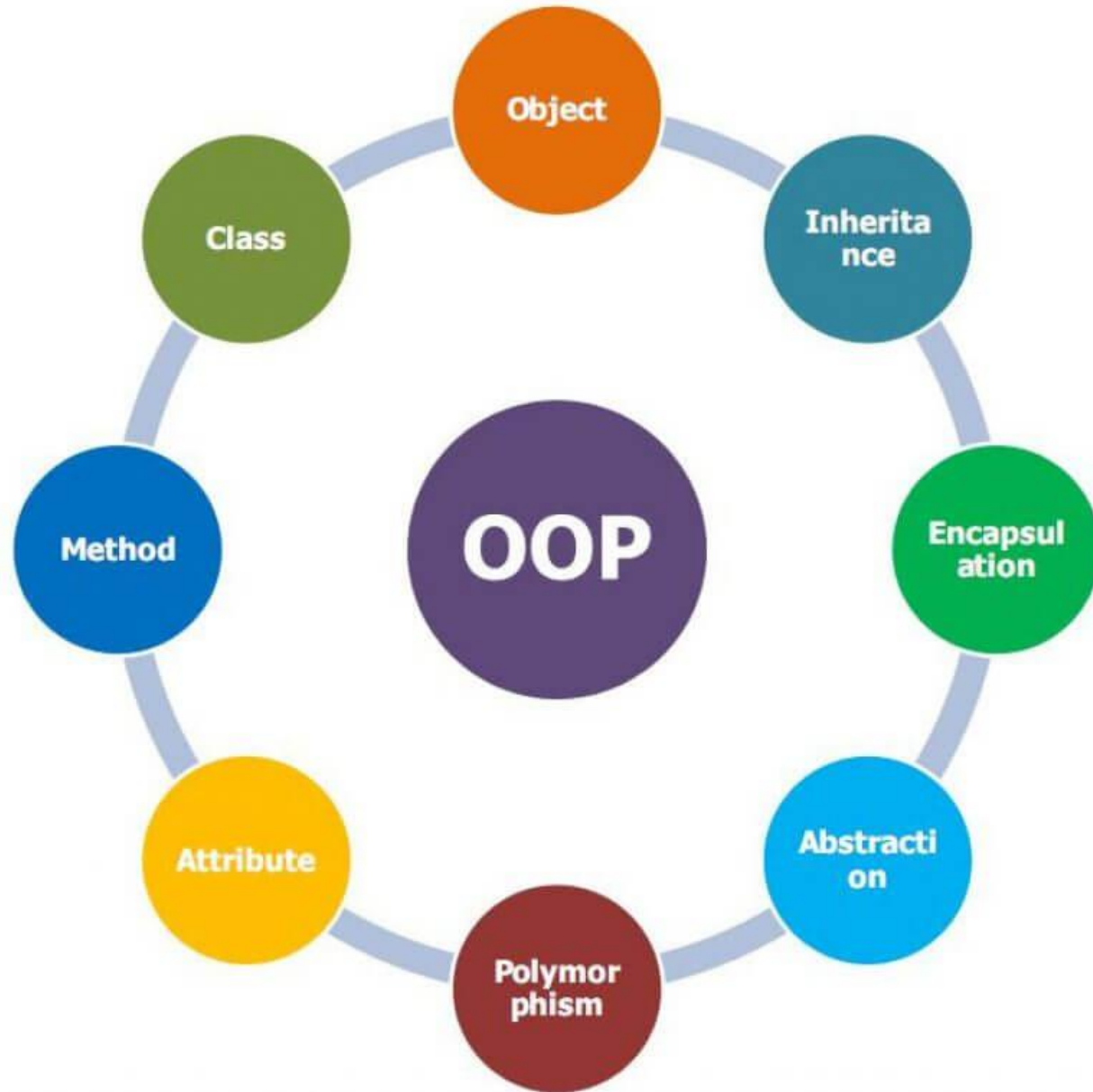
Например:

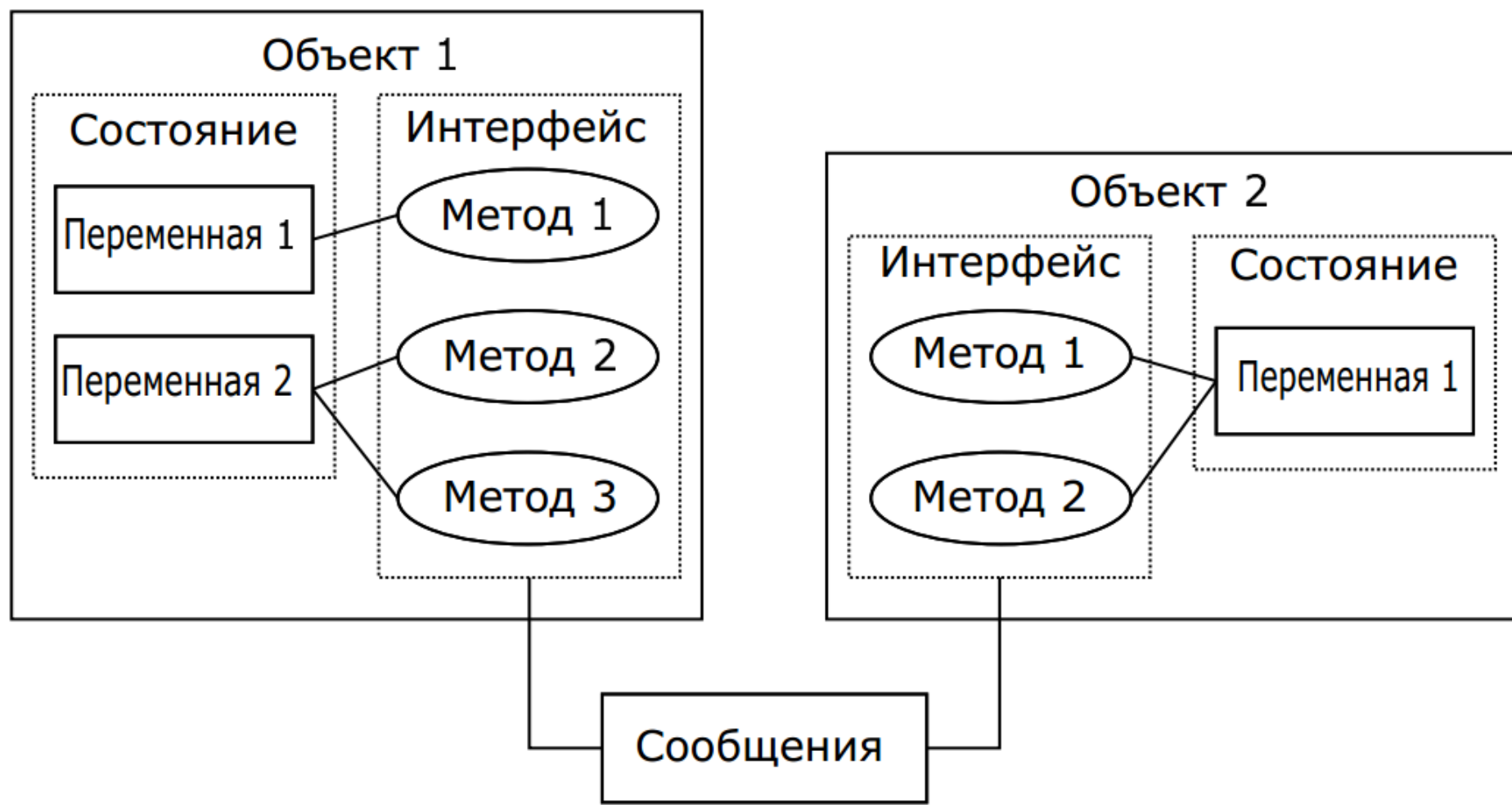
Когда декорируется функция `aaa('xxx', 'yyy', 'zzz')`, то результатом будет `'xxxуууzzz'`

ООП в Python

Определение ООП

- Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.





Понятия ООП

- Класс
- Объект
- Интерфейс

Класс

- Класс — универсальный, комплексный **тип данных**, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).
- В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определённым полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю.

Объект (экземпляр)

- Объект - это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом.
- Каждый объект имеет конкретные значения атрибутов и методы, работающие с этими значениями на основе правил, заданных в классе.

Пример

```
class Person: # Класс
```

```
    def __init__(self, name, money):
```

```
        self.name = name
```

```
        self.money = money
```

```
    def give_money(self, other, x_money):
```

```
        other.money += x_money
```

```
        self.money -= x_money
```

```
a = Person('Pete', 200) # экземпляр класса
```

```
b = Person('Nick', 300) # еще один экземпляр
```

```
print(a.money, b.money) # свойства
```

```
a.give_money(b, 100)
```

```
print(a.money, b.money)
```

```
# Выполните эту программу. А теперь верните эти деньги и проверьте, что данные  
восстановлены
```

Задание

Добавьте метод класса `Person`, который выводит на печать имя и количество денег

Добавим методы в классе

```
def info(self):
```

```
    return f"Меня зовут {self.name}. У меня {self.money} рублей"
```

```
# Реализуйте этот метод и проверьте его
```

```
# Дополните метод передачи денег проверкой  
платежеспособности. Если денег не хватает, то напишите  
сообщение об этом
```

Основные принципы ООП

- Инкапсуляция - сокрытие реализации.
- Наследование - создание новой сущности на базе уже существующей.
- Полиморфизм - возможность иметь разные формы для одной и той же сущности.
- Абстракция - набор общих характеристик.
- Посылка сообщений - форма связи, взаимодействия между сущностями.
- Переиспользование - повторное использование кода.

Инкапсуляция

- Инкапсуляция – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя, открыв только то, что необходимо при последующем использовании.
- Цель инкапсуляции — уйти от зависимости внешнего интерфейса класса (то, что могут использовать другие классы) от реализации. Чтобы малейшее изменение в классе не влекло за собой изменение внешнего поведения класса.

Наследование

- Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.
- Класс, от которого производится наследование, называется предком, базовым или родительским. Новый класс – потомком, наследником или производным классом.

Задание

Создайте класс Pet с атрибутам имя, вес и уровень сытости.

Напишите метод info, который в качестве результата выдает эти атрибуты.

Напишите метод hungry, который возвращает уровень сытости и комментирует: если меньше 5, то «голоден», если больше 10, то «сыт».

Напишите метод feed, который передает питомцу некоторую еду, которая прибавляется к уровню сытости и вызывает метод info.

Задача 17-1

Создайте регулярное выражение, которое вылавливает только четные целые числа

Задание 17-2

Создайте декоратор, которые переводит все текстовые аргументы функции в верхний регистр и возвращает их в виде списка текстовых аргументов.

Задание 17-3

- Создайте класс Shape, объекты которого имеют атрибуты Colour – строка, например, «Красный», «Синий»; Square – площадь объекта
- Создайте несколько методов:
 1. Установить цвет объекта
 2. Запросить цвет объекта и напечатать его
 3. Задать площадь объекта
 4. Запросить площадь объекта