Программирование на Python.

Класс и объект. Принципы ООП. ООП в Python

Преподаватель: Панченко Игорь Валентинович

Введение

Python - это язык с **динамической типизацией**, который поддерживает многие парадигмы программирования:

- Процедурное программирование
- Функциональное программирование
- ООП

Старый подход – структурное программирование

Циклы, ветвления и функции – все это элементы структурного программирования (подходит для небольших и простых программ)

Однако крупные проекты часто реализуют, используя парадигму объектноориентированного программирования (ООП).

В языке Python ООП играет ключевую роль.

Концепция ООП

ООП – парадигма программирования, основой которой являются объекты и классы.

Класс – тип, описывающий устройство объекта (чертёж объекта)

Объект – экземпляр класса. Конкретная сущность, обладающая атрибутами.

КЛАСС

Шаблон для создания объектов, который содержит данные и функции для обработки этих данных.

Данные класса называют **атрибутами (или полями)**, а функции – **методами**.

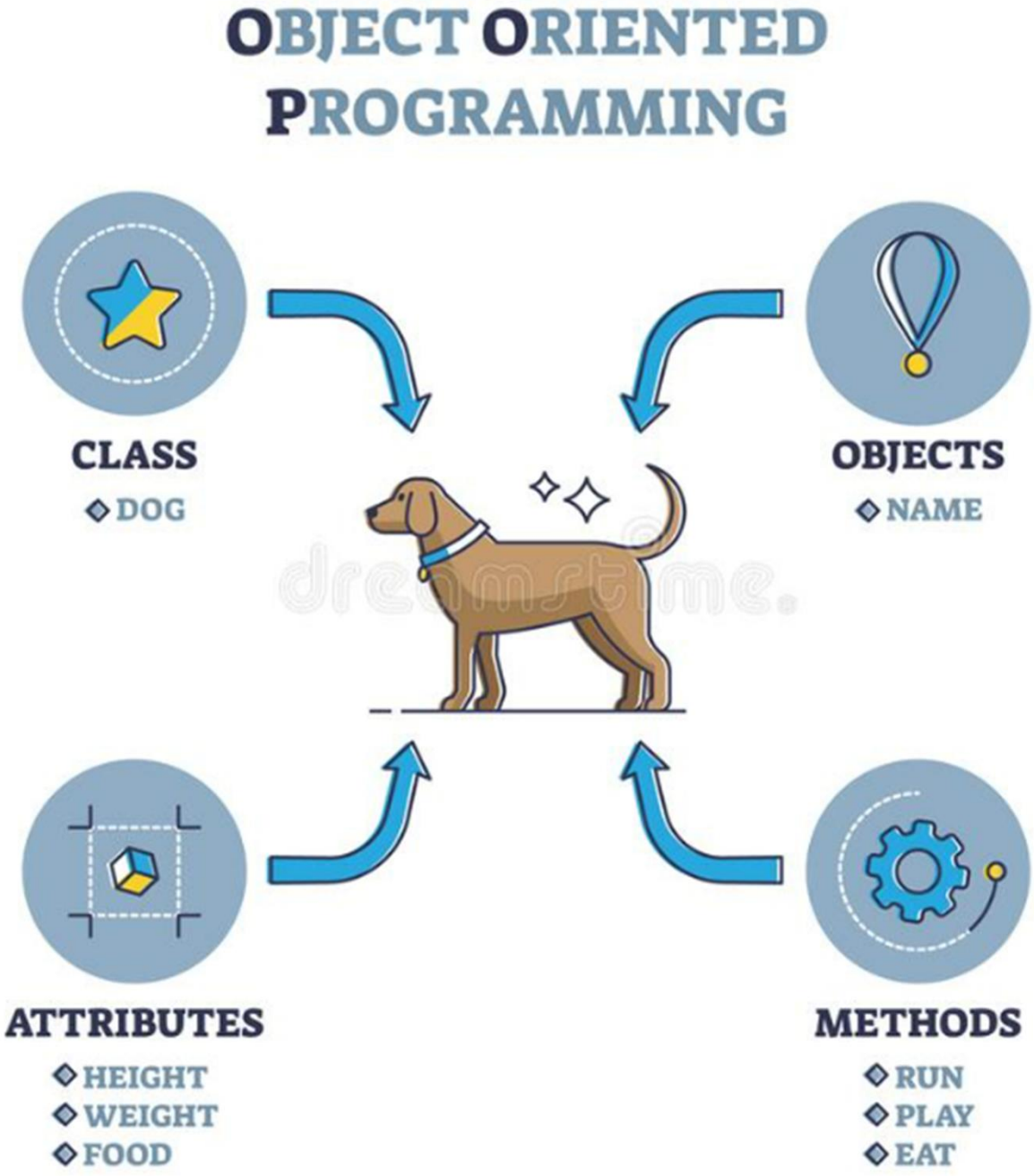
```
class ClassName:
    pass  # здесь ваш код
```

С точки зрения ООП объект это контейнер состоящий из:

- Данных, отражающих текущее состояние объекта

(атрибуты)

- Поведения (методы)



Разница между функциональным подходом и ООП

Мы можем описать собаку при помощи словаря и создать функции для описания её поведения:

```
dog = {
    'name': 'GoodBoy',
    'height': 75,
    'weight': 30,
    'food': 'natural'
}

def run(object):
    print(f"{object['name']} бегает")
def eat(object):
    print(f"{object['name']} кушает")
```

```
run(dog)
#GoodBoy бегает

eat(dog)
#GoodBoy кушает
```

Разница между функциональным подходом и ООП

Либо, мы можем создать класс Dog() с необходимыми атрибутами и «спрятать» необходимое поведение внутрь класса:

```
#создаём экземпляр класса
doggo = Dog()
#вызываем методы
doggo.run()
#GoodBoy бегает
#смотрим атрибуты
print('Имя:',doggo.name)
#Имя: GoodBoy
```

```
class Dog():
    name= 'GoodBoy'
    height= 75
    weight= 30
    food = 'natural'

    def run(self):
        print(f"{self.name} бегает")

    def eat(self):
        print(f"{self.name} кушает")
```

Задача

Создайте новый экземпляр класса Dog и проверьте её имя, рост и вес. Попробуйте изменить значение атрибутов

Как проверить, относится ли объект к какому-то классу

Используйте функцию **isinstance(obj, class)** для проверки – является ли объект экземпляром класса:

```
print(isinstance(l, list))   # True
print(isinstance(tony, Dog)) # True
print(isinstance(l, Dog))    # False
```

В Python всё – объекты!

Все объекты программ в Python являются производными классов и наследуют их атрибуты. Каждый объект формирует собственное пространство имен.

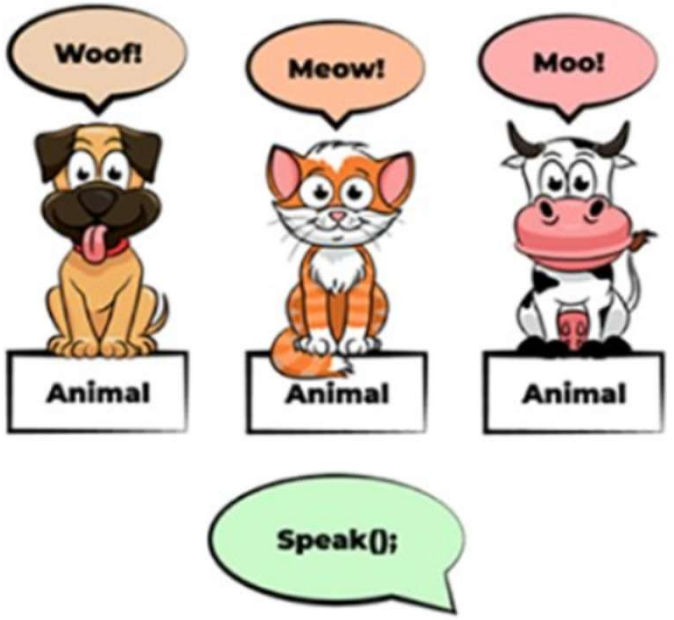
Python поддерживает основные принципы ООП:

- Полиморфизм
- Наследование
- Инкапсуляция

Полиморфизм

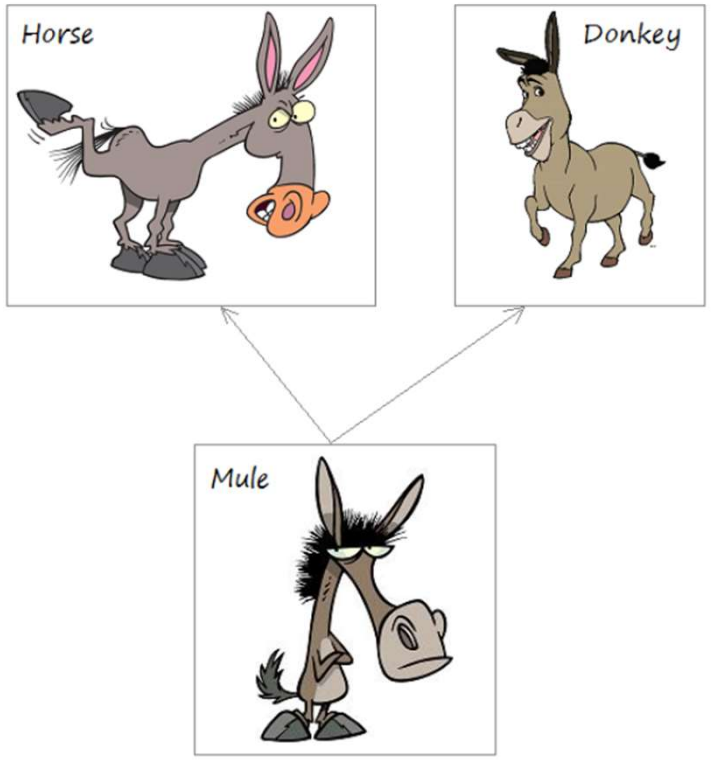
Это принцип, который позволяет объектам разных классов иметь схожие интерфейсы. Принцип реализуется путем добавления методов с одинаковыми именами

(сюда же можно отнести методы перегрузки операторов)



Наследование

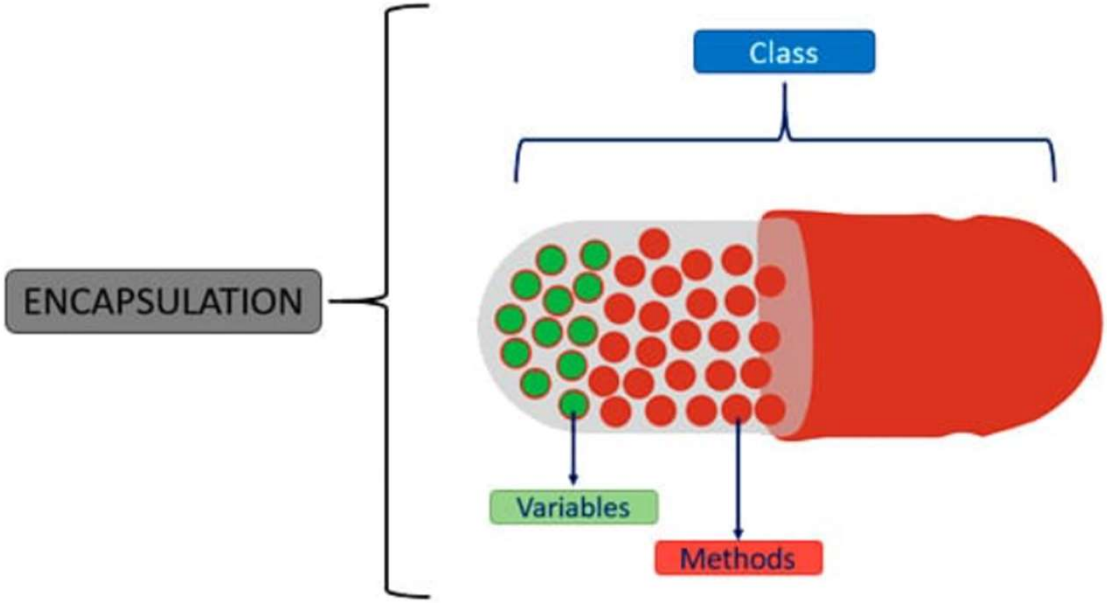
Механизм **наследования** позволяет использовать уже существующий код, вместо того, чтобы писать новый, а также – настраивать существующий код за счет добавления новых атрибутов переопределения старых.



Это позволяет обеспечить совместимость программных объектов и легкую расширяемость кода.

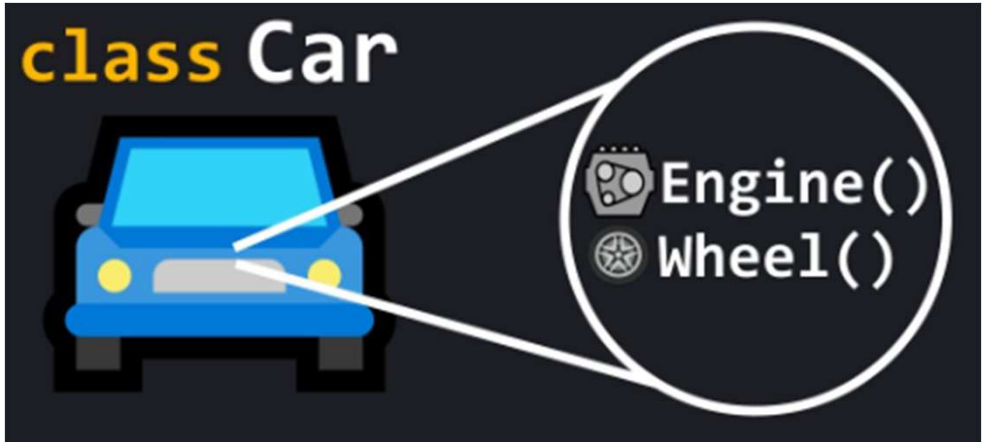
Инкапсуляция

Это принцип, который позволяет обеспечить управляемый доступ к атрибутам и методам программных объектов.



Прочие принципы ООП

Композиция (так же называют ассоциация или агрегирование) – когда класс включает в себя вызовы других классов. В результате, при создании объекта от класса-агрегата, создаются объекты других классов, являющиеся составными частями первого.



Преимущества ООП

- Все функции, связанные с объектом хранятся «внутри» него.
- Не требуется повторять код – вы создаёте класс, выступающий в качестве шаблона для объектов, и создаёте объекты с необходимыми атрибутами.

Использование атрибутов объекта

В этом примере – мы сохраняем результаты вычислений, прямо внутри словаря функции!

```
def func(x):
    if x not in func.__dict__:
        func.__dict__[x] = x**2
        print('Новинка!')

        return func.__dict__[x]
    else:
        print('Уже знаем!')
        return x**2
```

```
print('Первая команда')
print(func(3))
print('Вторая команда')
print(func(3))
```

```
Первая команда
Новинка!
9
Вторая команда
Уже знаем!
9
```

Задача

Создайте функцию для вычисления чисел Фибоначчи, которая использует внутренний словарь в качестве кэш-хранилища для вычисленных ранее чисел.

Решение

```
def f(n):
    if n in f.__dict__: #если вычисляли ранее - результат
        return f.__dict__[n]
    else: #иначе - вычисляем n-ное число Фибоначчи
        if n == 0:
            f.__dict__[n] = 0
            return 0
        if n == 1:
            f.__dict__[n] = 1
            return 1
        else: # Рекурсивный вызов функции
            res = f(n-1) + f(n-2)
            f.__dict__[n] = res
            return res
```

Доп. задачи

1. Реализуйте класс Circle, представляющий окружность. Класс должен включать методы, вычисляющие его площадь и периметр.
2. Реализуйте класс Person, представляющий человека. Класс должен включать атрибуты имя, национальность и дата рождения. Добавьте метод, вычисляющий возраст человека.
3. Реализуйте класс Calculator, реализуйте методы для базовых арифметических операций.

Решение №1

```python
#Упражнение 1
class Circle:
    def p(self,r):
        return 2 * r * 3.14
    def s(self,r):
        return 3.14*(r**2)


c = Circle()

# c.radius = 5
# print(c.p(c.radius))
# print(c.s(c.radius))
```

Решение №2

```python
import datetime

#Упражнение №2
class Person():
    name = 'Человек'
    citizenship = 'Россия'
    birthdate = '1984-09-21'

    def age(self):
        print('Год рождения:',self.birthdate[:4])
        return datetime.date.today().year-int(self.birthdate[:4])

p = Person()
print(p.age())
```