STEP IT ACADEMY

Занятие №14

Объектно-ориентированное программирование

Закрепление ранее изученного материала

Задание. Получить имена студентов из словаря.

Напишите функцию, которая вернет массив с именами студентов в алфавитном порядке. Принимать она должна словарь.

Объектно-ориентированное программирование (ООП) в Python

Python поддерживает ООП на сто процентов: все данные в нём являются объектами.

Числа всех типов, строки, списки, словари, даже функции, модули, и наконец, сами типы данных — всё это объекты!

Все вычисления в Python можно представить как взаимодействия между объектами.

Рассмотрим примеры:

Класс:
программист

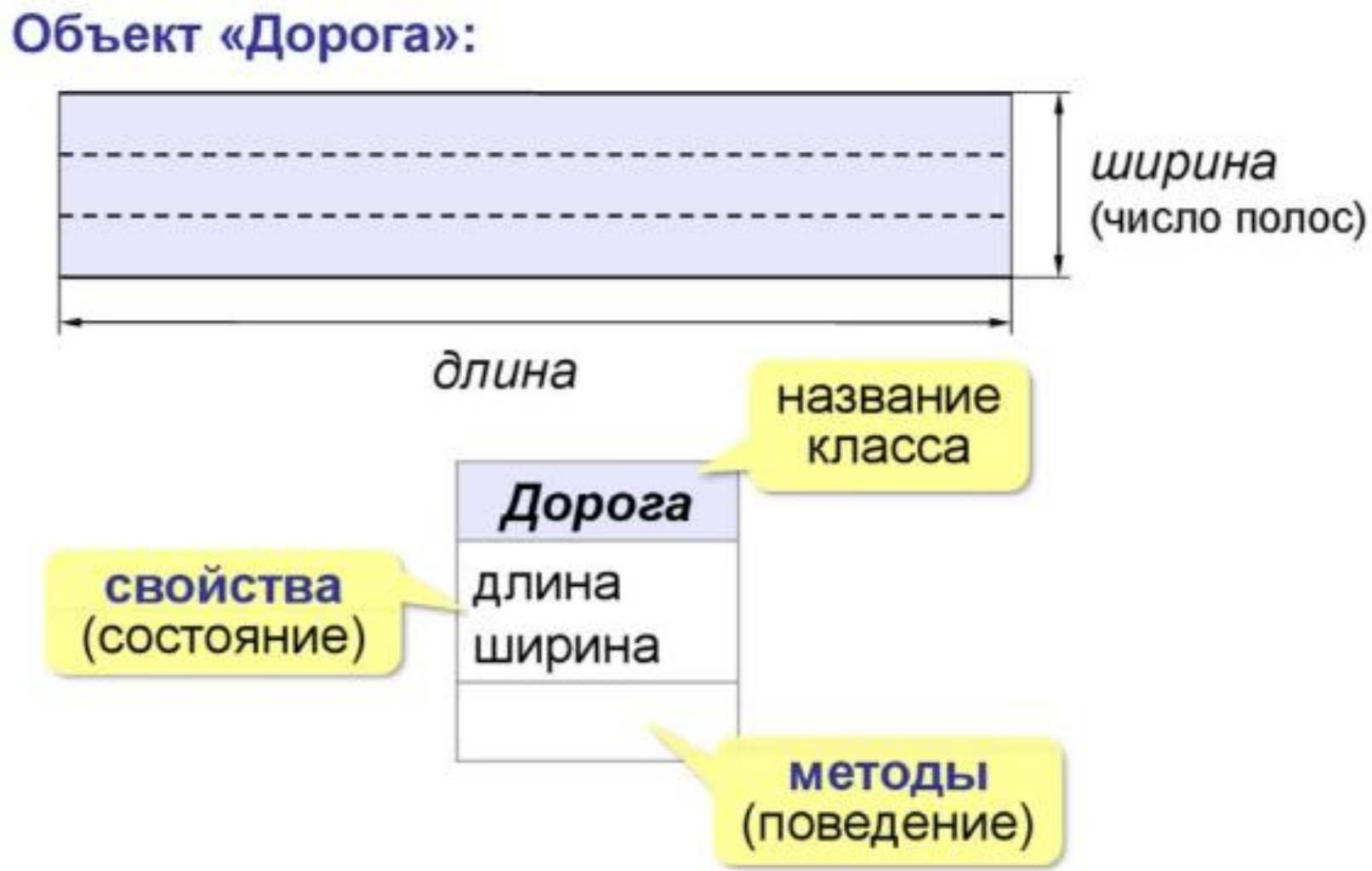
Объект:
разработчик Иван

Атрибуты:
зарплата, обязанности

Методы:
написание кода

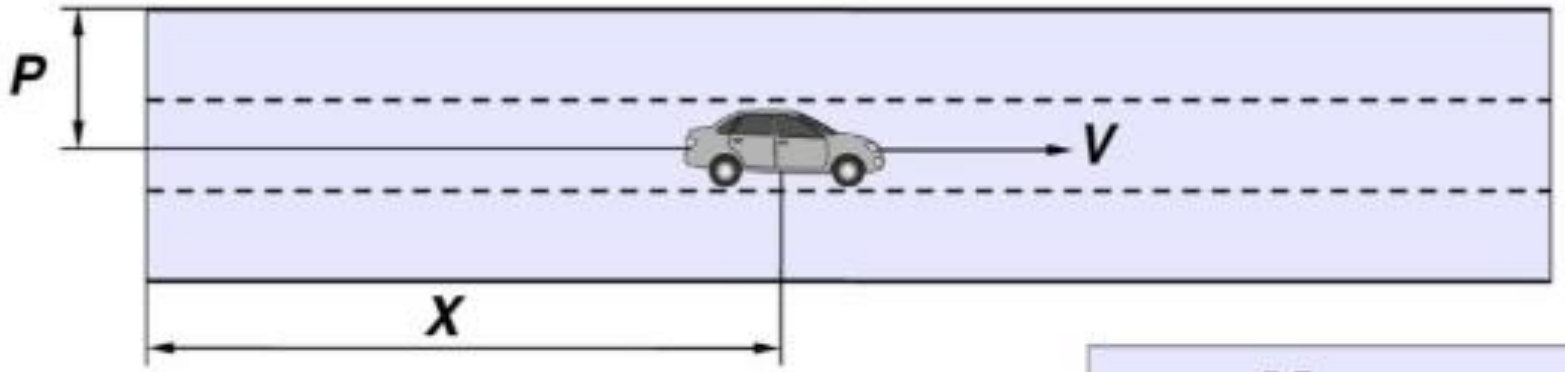
Рассмотрим примеры:

Модель дороги с автомобилями



Объект «Машина»:

свойства: координаты и скорость



- все машины одинаковы
- скорость постоянна
- на каждой полосе – одна машина
- если машина выходит за правую границу дороги, вместо нее слева появляется новая машина

Машина
X (координата) P (полоса) V (скорость)
двигаться

Метод – это процедура или функция, принадлежащая классу объектов.

Взаимодействие объектов:

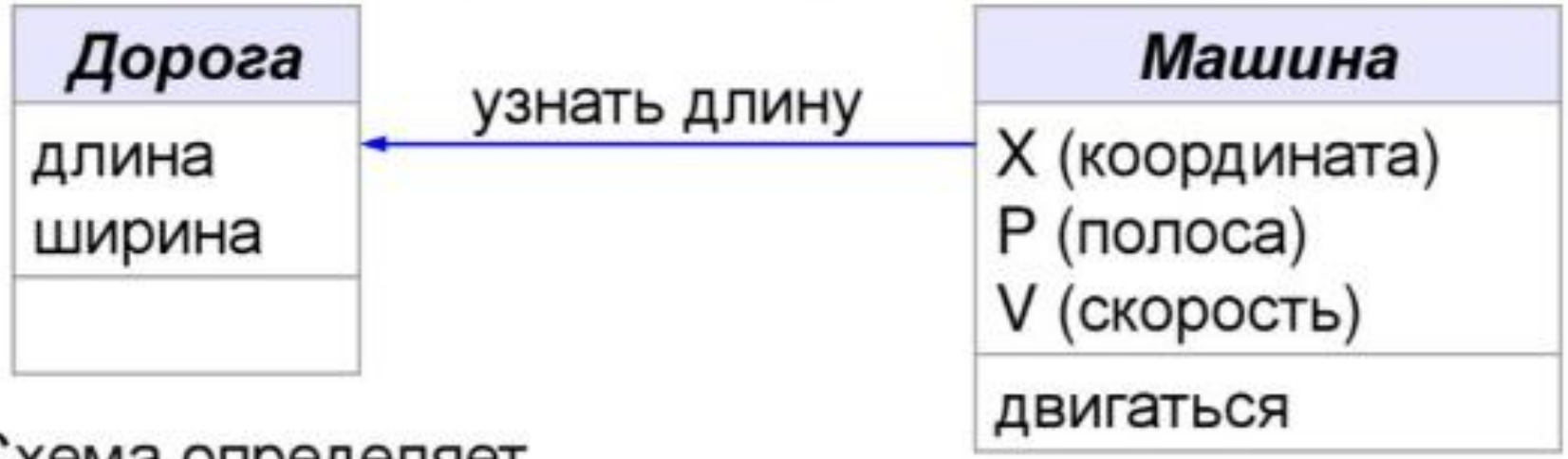


Схема определяет

- **свойства** объектов
- **методы**: операции, которые они могут выполнять
- **связи** (обмен данными) между объектами

Ни слова о внутреннем устройстве объектов!

Основные понятия

Объектно-ориентированное программирование (**ООП**) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс

Класс — тип, описывающий устройство объектов.
Класс можно сравнить с чертежом, по которому создаются объекты.

Говоря языком программиста, класс — это такой тип данных, который создается для описания сложных объектов.

Объект

Объект — это экземпляр класса.

Хранит конкретные значения свойств и информацию о принадлежности к классу.

Может выполнять методы.

Атрибут

Атрибут - свойство, присущее объекту.

Класс объекта определяет, какие атрибуты есть у объекта.

Конкретные значения атрибутов — характеристика уже не класса, а конкретного экземпляра этого класса, то есть объекта.

Метод

Метод - действие, которое объект может выполнять над самим собой или другими объектами.

Примеры

1, 2, 3, 'abc', [10, 20, 30]

int, str, list

объекты

классы

1, 2, 3 # экземпляры класса int #

'abc' экземпляр класса str

[10, 20, 30] # экземпляр класса list, в который вложены экземпляры класса int

Как узнать класс объекта?

Чтобы узнать, к какому классу относится тот или иной объект, можно воспользоваться функцией type.

```
type(123)
# => '<class 'int'>'
```

```
type([1, 2, 3])
# => '<class 'list'>'
```

Создание классов

Простейший класс

```
class Fruit: pass
```

Определение этого класса состоит из зарезервированного слова class, имени класса и пустой инструкции после отступа.

Внутри класса с дополнительным уровнем отступов должны определяться его методы, но сейчас их нет.

Однако хотя бы одна инструкция должна быть, поэтому приходится использовать пустую инструкцию-заглушку pass.

PEP 8

Имена классов по стандарту именования PEP-8 должны начинаться с большой буквы.

Встроенные классы (int, float, str, list и другие) этому правилу не следуют, однако в вашем коде его лучше придерживаться. Так делает большинство программистов на Python.

Создаём экземпляры класса

Теперь создадим два конкретных фрукта — экземпляра класса Fruit:

```
a = Fruit() b =
        Fruit()
```

Создаём атрибуты

Переменные a и b содержат ссылки на два разных объекта — экземпляра класса Fruit, которые можно наделить разными атрибутами:

```
a.name = 'apple'
a.weight = 120
# теперь a - это яблоко весом 120 грамм
b.name = 'orange'
b.weight = 150
# a b - это апельсин весом 150 грамм
```

Атрибуты

Атрибуты можно не только устанавливать, но и читать.

При чтении ещѐ не созданного атрибута будет появляться ошибка AttributeError. Вы еѐ часто увидите, допуская неточности в именах атрибутов и методов.

Атрибуты

```
print(a.name, a.weight)          # apple 120
print(b.name, b.weight)          # orange 150

b.weight -= 10                   # Апельсин долго лежал на складе и усох
print(b.name, b.weight)          # orange 140

c = Fruit() c.name =
'lemon' c.color =
'yellow'
# Атрибут color появился только в объекте c.
# Забыли добавить свойство weight и обращаемся к нему
print(c.name, c.weight)
# Ошибка AttributeError, нет атрибута weight
```

Методы классов

Создание метода класса

```
class Greeter:
        def hello_world(self):
                print("Привет,          Мир!")


greet =       Greeter()
greet.hello_world()                          # выведет      "Привет,      Мир!"
```

Создаём метод класса

Мы написали метод, с синтаксисом вызова которого вы хорошо знакомы по методу строк split или методу списков append.

При создании собственных методов обратите внимание на два момента:

- Метод должен быть определѐн внутри класса (добавляется уровень отступов);
- У методов всегда есть хотя бы один аргумент, и первый по счѐту аргумент должен называться self.

Аргумент self

В него передается тот объект, который вызвал этот метод.
Поэтому self ещѐ часто называют «контекстным объектом».

greet.hello_world() преобразуется в вызов hello_world(greet)

```
class Greeter:
    def hello_world(self):
        print("Привет, Мир!")

    def greeting(self, name):
        '''Поприветствовать целовека с именем name.'''
        print("Привет, {}!".format(name))

    def start_talking(self, name, weather_is_good):
        '''Поприветствовать и нацать разговор с вопроса о погоде.'''
        print("Привет, {}!".format(name))
        if weather_is_good:
            print("Хорошая погода, не так ли?")
        else:
            print("Отвратительная погода, не так ли?")


greet = Greeter()
greet.hello_world()      # Привет, Мир!
greet.greeting("Петя")   # Привет, Петя
greet.start_talking("Саша", True)
                         !
# Привет, Саша!
# Хорошая погода, не так ли
```

Инициализация экземпляров класса

Класс «Машина»

```
class Car:
    def start_engine(self):
        engine_on = True  # К сожалению, не сработает

    def drive_to(self, city):
        if engine_on:  # Ошибка NameError
            print("Едем в город {}.".format(city))
        else:
            print("Машина не заведена, никуда не едем")


c = Car()
c.start_engine()
c.drive_to('Владивосток')
```

Конструкторы

Конструктор — это специальный метод, который вызывается по умолчанию когда вы создаете объект класса.

Для создания конструктора вам нужно создать метод с ключевым словом __init__. Взгляните на следующий пример:

class Car:

```
    # создание атрибутов класса
car_count = 0

# создание методов класса
def __init__(self):
    Car.car_count +=1
    print(Car.car_count)
```

```
class Car:
    def_init_(self):
        self.engine_on = False

    def start_engine(self):
        self.engine_on = True

    def drive_to(self, city):
        if self.engine_on:
            print("Едем в город {}.".format(city))
        else:
            print("Машина не заведена, никуда не едем.")


car1 = Car()
car1.start_engine()
car1.drive_to('Владивосток')   # Едем в город Владивосток.
car2 = Car()
car2.drive_to('Лиссабон')      # Машина не заведена, никуда не едем.
```

ООП. Инкапсуляция

Инкапсуляция

Технология сокрытия информации о внутреннем устройстве объекта за внешним интерфейсом из методов называется инкапсуляцией.

Надо стараться делать интерфейс методов достаточно полным. Тогда вы, как и другие программисты, будете пользоваться этими методами, а изменения в атрибутах не будут расползаться по коду, использующему ваш класс.

Инкапсуляция просто означает скрытие данных. Как правило, в объектно-ориентированном программировании **один класс не должен иметь прямого доступа к данным другого класса**.

Вместо этого, доступ должен контролироваться через методы класса.

Чтобы предоставить контролируемый доступ к данным класса в Python, используются **модификаторы доступа** и **свойства**.

```
# создаем класс Car
class Car:

    # создаем конструктор класса Car
    def __init__(self, model):
        # Инициализация свойств.
        self.model = model

    # создаем свойство модели.
    @property
    def model(self):
        return self.__model
```

```
    # Сеттер для создания свойств.
    @model.setter
    def model(self, model):
        if model < 2000:
            self.__model = 2000
        elif model > 2018:
            self.__model = 2018
        else:
            self.__model = model

    def getCarModel(self):
        return "Год выпуска модели " + str(self.model)

carA = Car(2088)
print(carA.getCarModel())
```

Зачем скрывать внутреннее устройство?

Объектная модель задачи:

- защита внутренних данных
- проверка входных данных на корректность
- изменение устройства с сохранением интерфейса

Инкапсуляция («помещение в капсулу») – скрытие внутреннего устройства объектов.

Также объединение данных и методов в одном объекте!

Рассмотрим пример

```
class TPen:
  def __init__ ( self ):
    self.color = "000000"
```

R G B

По умолчанию все члены класса открытые (в других языках – `public`)!

```
class TPen:
  def __init__ ( self ):
    self.__color = "000000"
```

Как обращаться к полю?

Имена скрытых полей (`private`) начинаются с двух знаков подчёркивания!

```
class TPen:
  def __init__( self ):
    self.__color="000000"
  def getColor( self ):
    return self.__color
  def setColor( self, newColor ):
    if len(newColor) !=6:
      self.__color="000000"
    else:
      self.__color=newColor
```

метод чтения

метод записи

если ошибка, чёрный цвет

Защита от неверных данных!

Использование:

```
pen = TPen()
pen.setColor( "FFFF00" )
print ( "цвет пера:", pen.getColor() )
```

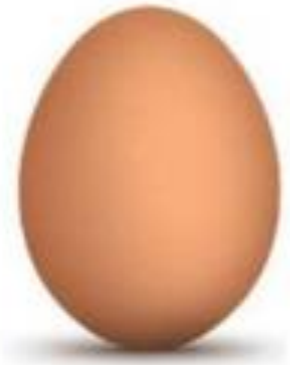
установить цвет

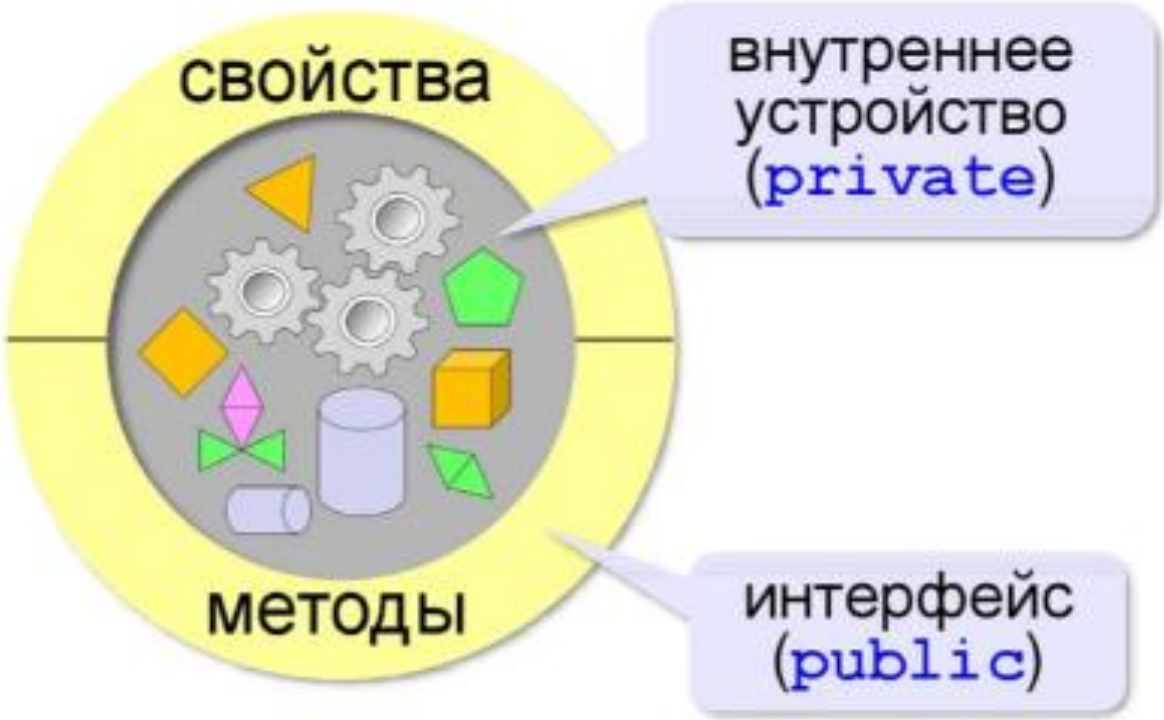
прочитать цвет

Не очень удобно!

```
pen.color = "FFFF00"
print ( "цвет пера:", pen.color )
```

Инкапсуляция («помещение в капсулу»)






STEP
IT ACADEMY