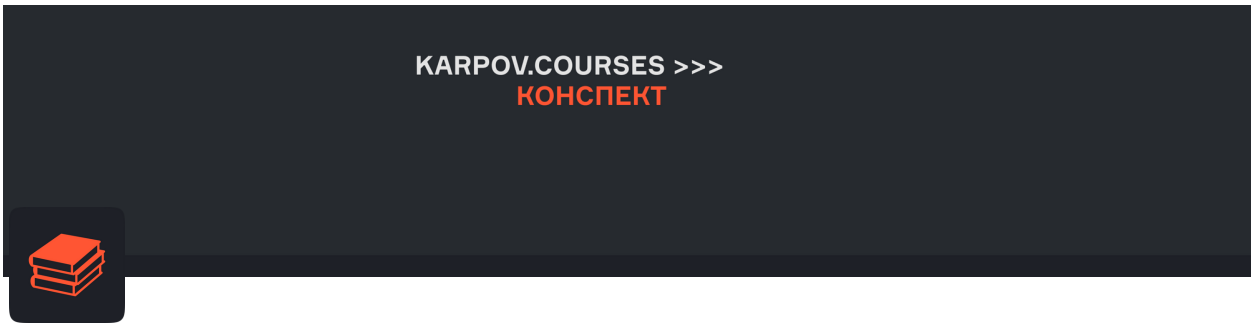

> Конспект > 7 урок > Классы и ООП

>Оглавление 7 урока

>Глоссарий

- **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.

- **Класс -** Определенный программистом прототип программируемого объекта с набором атрибутов (переменных и методов), которые описывают данный объект. Доступ к аттрибутам и методам осуществляется через точку
- **Переменная класса -** Переменная, доступная для всех экземпляров данного класса. Определяется внутри класса, но вне любых методов класса.
- **Абстрактные классы** - это класс, которые объявлен, но не содержит реализации. Он не предполагает создание своих объектов, т.е служит только для того, чтобы хранить общие свойства между другими классами. Абстрактные классы работают как шаблон для подклассов.
- **Экземпляр класса -** Отдельный объект-представитель определенного класса.
- **Переменная экземпляра класса -** Переменная определенная внутри медота класса, принадлежащая только к этому классу.
- **Метод -** Особая функция, определенная внутри класса.
- **Наследование (Inheritance) -** Передача аттрибутов и методов родительского класса дочерним классам.
- **Перегрузка функций (Function overloading) -** Изменение работы метода, унаследованного дочерним классом от родительского класса.
- **Перегрузка операторов (Operator overloading) -** Определение работы операторов с экземплярами данного класса.
- **Инверсия зависимостей -**
- **Полиморфизм -** возможность обработки разных типов данных(т. е. принадлежащих к разным классам) с помощью "одной и той же" функции, или метода.
- **Магические методы -** базовые методы, которые можно назначить любому классу.

>Объекты и их взаимосвязи

ООП или **Объектно-ориентированное программирование** — это парадигма программирования(совокупность идей и понятий, определяющих стиль написания

программ), основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса. Суть ООП заключается в объединении данных(функций, методов) для их обработки и помещении их в объекты.

Допустим, у Алексея есть машина, Если мы будем писать код, описывающий автомобиль, то нам достаточно будет определить некую переменную(пусть это будет кортеж) `car` и сложить в нее, например, марку. Выглядеть это будет примерно так:

```
car = ("Volkswagen")
```

Но, марки бывают разные, то есть, Алексей может иметь и Volkswagen и BMW и Ford. Все они какого-то цвета, у них разный тип топлива, разный кузов и т.п. В таком случае, мы уже не обойдёмся одной маркой, чтобы описать автомобиль. Словом, все машины имеют разный набор характеристик(свойств). Но в то же время, все автомобили описываются схожим образом, как по шаблону.

Как же лучше всего собрать все эти характеристики воедино? С одной стороны, для общих характеристик хочется иметь общее название, например, если вы обозначите цвет за `color`, то эта переменная должна так же называться и в других моделях. С другой стороны, навязать всем один строгий набор переменных не самая лучшая идея. А если в будущем появятся новые авто с какими-то новыми характеристиками? Нужно как-то сделать свой код расширяемым, чтобы в будущем можно было бы без труда его дописать.

```
alexeys_car = ('Green', 'Ford', 'Mustang', 'Gasoline')
alexeys_car_2 = ('Blue', 'Volkswagen', 'Golf', 'Diesel')
# car = ('color', 'manufacturer', 'series', 'fuel_type')
```

Можно написать что-то в таком духе, но если появится новое авто с новыми характеристиками(наличие автопилота, например), то неясно, как ее аккуратно добавить в такую структуру. Нам помогут классы

>Классы в Python

Класс — это объединение свойств объекта и действий, которые этот объект может совершать(шаблон по которому создаются объекты). Все объекты какого-то класса будут иметь набор одинаковых характеристик(цвет), но могут иметь разное состояние(синий, желтый, голубой). Пользовательские объекты описываются с помощью классов, по сути, класс это описание объекта. Объекты созданные на основе классов это объекты класса(экземпляры класса)

Создание класса:

```
# class - Ключевое слово
# Auto - Название класса(записывается в CamelCase(СлитноКаждоеСловоСЗаглавной))
# : - даёт понять python, что дальше идёт блок кода
# @dataclass - упрощает создание класса в нашем случае

from dataclasses import dataclass
@dataclass
class Auto:
    color: str
    manufacturer: str
    series: str
    fuel_type: str
```

Мы создали шаблон, по которому можем создавать конкретные автомобили. Давайте создадим объект автомобиля:

```
car = Auto('Green', 'Ford', 'Mustang', 'Gasoline')
car_2 = Auto('Blue', 'Volkswagen', 'Golf', 'Diesel')
print(car_2)

Output:
Auto(color='green', manufacturer='Ford', series='Mustang', fuel_type='Gasoline')
```

В примере выше мы из библиотеки dataclasses импортировали dataclass, что это такое? Модуль dataclasses предоставляет декораторы и функции для того, чтобы автоматически добавлять магические(специальные) методы, такие как `__init__()` или `__repr__()` к определяемым пользовательским классам. То есть, dataclass заменил нам объявление

```
# функция будет вызываться каждый раз,
# когда мы попросим создать новый объект.
class AutoShort:
```

```
    def __init__(self, color):  # self - сам объект, который будет создан
        self.color = color  # создаваемому объекту задаем переменную color
```

`__init__` это конструктор класса(метод, который автоматически вызывается при создании объектов), он объявляет Python как нужно создавать объекты класса.

Конструктор класс принимает аргумент и клепает объекты класса. Давайте создадим объект класса AutoShort:

```
colored_car = AutoShort('red')
colored_car.color  # наш синтаксис с точкой

Output:
red
```

Помимо того, что можно вытаскивать из класса его свойство, возможно его менять:

```
colored_car.color = 'blue'
colored_car.color
```

Класс создаёт своё пространство имён, специальный словарь, в котором хранятся имена переменных и функции с их значениями. Определение свойств класса делается с помощью присваивания переменной какого-то значения(внутри класса)

```
class Car:
  car_color = "Black"
```

В соответствии с записью выше, Python создаст свойство(атрибут, поле) car_color для класса Car.

Больше о dataclasses

Больше о магических методах

>Методы в Python

Помимо свойств у классов и объектов есть еще и методы. Т.е объект может не только иметь свойства, но и уметь что-то делать. Метод это функция объявленная

внутри класса

```
class AutoWithAlarm:
    # через  name: type  можно делать рекомендации типа
    def __init__(self, color, alarm_sound: str):
        self.color = color  # Принимаем объект и записываем в него свойство
        self.alarm_sound = alarm_sound

    # эта функция уйдет в каждый объект класса, он будет уметь ее вызывать
    def alarm(self):
        # внутри этой функции можем обращаться к свойствам объекта
        print(self.alarm_sound)
```

У нас 2 метода: `__init__()` - конструктор класса(объявляет Python как нужно создавать объекты) и `alarm()`, который отвечает за подачу звукового сигнала

```
my_new_car = AutoWithAlarm('red', 'beep-beep-beeeeeep')
```

Теперь у объекта класса AutoWithAlarm будут следующие свойства: color = ‘red’, а alarm_sound = ‘beep-beep-beeeeeep’. Проверим

```
print(my_new_car.color)

Output:
'red'
```

```
print(my_new_car.alarm())

Output:
'beep-beep-beeeeeep'
```

Мы вызвали функцию `alarm()` у объекта `my_new_car`, т.е мы сначала создали объект класса AutoWithAlarm и этот объект получает на вход свойство и метод.

Мы создали класс как шаблон объектов, затем начали создавать объекты по образу и подобию этого класса.

Такой подход называется ООП - *объектно-ориентированное программированием*.

В ООП вы строите свой код через классы, объекты и их взаимодействие. Небольшая загвоздка в том, что в Python всё является объектами, даже классы. Функции, модули, файлы - всё это тоже объекты. Хм, если класс сам по себе объект, то какие у него атрибуты? Посмотреть доступные атрибуты можно с помощью функции dir()

```
dir(AutoWithAlarm)

Output:
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'alarm']
```

Каждая переменная в классе называется *полем* или *свойством*, каждая функция - *методом*.

>Наследование

Суть наследования сводится к тому, что на базе одного класса(предка) возможно создать другой класс(**дочерний**), который получает все свойства и методы своего **родительского класса**(или **класса предка** или **суперкласса** или **базового класса**). Вы берёте некоторый класс, забираете у него все поля и методы и строите свой класс на основе предыдущего.

Представим, что у вас класс в котором вы уже написали много функционала, написали поля, методы и пр. Но теперь вы хотите создать класс функционал которого немного отличается от старого класса. Для решения этой проблемы можно просто скопировать код прошло класса, немного дополнив его. Но есть и более элегантный способ:

```
# Очень простое описание авто
class Auto:
    color = ''  # можно заранее создать поля, можно делать в __init__
    name = ''   # в этом примере сделаем и там, и там
    alarm_sound = ''

    def __init__(self, color: str, name: str, alarm_sound: str):
        self.name = name
        self.color = color
```

```
        self.alarm_sound = alarm_sound  #

    def beep(self):
        print(self.alarm_sound)

# Мы захотели его расширить: добавить звуковой сигнал
# синтаксис: после имени класса в круглых скобках пишем, от кого наследуемся
class AutoWithAlarm(Auto):
    # добавляем новый функционал
    def alarm(self):
        print('piy-piy')
```

```
car_sound = AutoWithAlarm('red', 'Polo', 'bepe')
print(car_sound.alarm())

Output:
piy-piy
```

При наследовании надо соблюдать несколько простых правил:

1. Наследуемый класс (`AutoWithAlarm`) по логике должен расширять базовый класс, а не перечеркивать его поведение и делать по-своему.
2. Код должен продолжать работать, если заменить в нем базовый класс на какой-то из его наследников.

Механизм наследования крайне полезен, потому что позволяет сократить количество кода(особенно в ситуации когда у нас есть свойства и методы общие для нескольких классов). Все общие свойства и методы переносим в базовый класс(**родительский класс** или **суперкласс**) и все дочерние классы получают эти свойства и методы автоматически. Удобно, не правда ли? Особенно наследование удобно использовать когда у нас есть некоторая иерархия:

```
class CarColor:
  color = 'red'
  name = 'Tesla'

class ModelS(CarColor):
  pass  #пропуск

class Cybertruck(CarColor):
  pass
```

Нижние члены иерархии(классификации) наследуют свойства и методы от своих предков

Есть очень удобная функция `isinstance()`. Она проверяет имеет ли конкретный объект какой-то атрибут, который достался ему от любого родительского класса в цепочке наследования(иерархии):

```
cybertruck1 = Cybertruck()
print(isinstance(cybertruck, CarColor))
#isinstace(проверяемый объект, класс который нужно проверить)
Output:
True
```

Код выше проверяет является ли переменная cybertruck1 экземпляром класса CarColor в цепочке наследования. Мы получили True, значит CarColor входит в цепочку наследования класса Cybertruck(этож его родительский класс+)

Принцип Барбары Лисков

>Инверсия зависимостей

Пусть, у нас есть 2 типа авто: с автоматической и механической трансмиссией(в механической трансмиссии нужно вручную переключать передачи, а в автоматической просто выставить селектор в нужное положение).

```
# Общий, несколько абстрактный, класс - от него будут наследоваться детали
class GeneralAuto:
    color = ''
    name = ''

# добавляем специфичную функцию, не свойственную всем автомобилям
class AutoTransmissionCar(GeneralAuto):
    def set_position(self, position):
        if position == 'D': # движение вперёд
            print(f'going forward')
        # дальше сложный код по выстановке автоматической трансмиссии

class ManualTransmissionCar(GeneralAuto):
    def set_transmission(self, step):
        if step == 'R':
            print('going backwards')
        # сложный код по переключению передачи
```

Оба класс `AutoTransmissionCar` и `ManualTransmissionCar` получат свойства **color** и **name** просто потому что наследуются от `GeneralAuto`, при этом, `GeneralAuto` так устроен, что не предполагает создания своих объектов(т.е нужен только для того, чтобы хранить общие свойства других классов). Такие классы как `GeneralAuto` называются абстрактными классами.

Абстрактные классы - это классы, которые объявлены, но не содержат реализаций. Он не предполагает создание своих объектов, т.е служит только для того, чтобы хранить общие свойства между другими классами. Абстрактные классы работают как шаблон для подклассов.

У всех машин есть общий функционал, например - возможность ехать по дороге и иметь двигатель. Исходя из этого, мы можем создать абстрактный класс Автомобиль, определить в нем абстрактный метод (в нашем случае - езда, поскольку у каждой машины разная скорость) и реализовать общий функционал (наличие двигателя).

>Перегрузка

Перегрузкой называют создание в подклассах свойств и методов с теми же именами, что и в родительском классе. Класс-наследник может переопределить поведение родительской функции(никакой перегрузки на самом деле не происходит)

```
class Auto:
    color = ''
    name = ''
    alarm_sound = ''

    def __init__(self, color: str, name: str, alarm_sound: str):
        self.name = name
        self.color = color
        self.alarm_sound = alarm_sound

    def beep(self):
        print(self.alarm_sound)


class AutoWithCustomBeep(Auto):
    # берем и нагло меняем поведение - переопределяем метод родителя
    def beep(self):
```

```
        # кстати, можно использовать родительскую версию
        super().beep()
        print('beep broke, sorry')
```

Python ищет определение метода или свойства сначала в локальном словаре экземпляра, потом в самом классе, и если не находит, то идёт в родительский класс и ищет там и так далее по всей цепочки иерархии. Расширение функциональности класса это создание дочернего класса с некоторой дополнительной функциональностью, которой не было в родительском. Т.е в отличие от перегрузки, в дочернем классе создаётся то, чего не было в родительском классе

Бывают ситуации, когда при переопределнии метода в дочерних классах нужно затем передать выполнение обратно в родительский

```
class Type:
  def __init__(self, car_type):
      self.car_type = car_type

class Car(Type):
  def __init__(self, brand):
      #super().__init__(name)
      self.brand = brand
```

При такой реализации создать `car` возможно только с брендом. Если надо определеить еще и тип машины, то можно добавить свойство `car_type` в `__init__` (в классе Car), но это бы нарушило принцип DRY(Don't repeat yourself). Чтобы присвоит Car еще и тип нужно передать управление в `__init__` метод родительского класса. Для этого и спользуется функция `super().функция род.класса`

>Полиморфизм

```
print(1 + 1)
Output:
2

print('1'+'1')
Output:
'11'
```

Для разных типов данных используется один и тот же оператор сложения и получаются разные результаты(оба операнда должны относиться к одному типу). На самом деле, оператор плюс это синтаксический сахар вызова магического метода `__add__`

```
'1'.__add__('1')
Output:
'11'
```

Полиморфизм в Python это разное поведение одного и того же метода для разных классов(релевантно, в первую очередь, для работы операторов). Возможность создавать *один и тот же метод* в классе, работающий с данными разного *типа.* В других языках этот пример бы сработал, но не в Python из-за его динамической типизации (он возьмет только лишь последнюю реализацию функции).

```
class NumberPrinter:
    # для int
    def print_number(self, num: int):
        print(f'integer, {num}')

    # для float
    def print_number(self, num: float):
        print(f'float, {num}')

NumberPrinter().print_number(5.5)
NumberPrinter().print_number(5)

Output:
float, 5.5
float, 5

#В языке с не динамической типизацией мы бы получили:
Output:
float, 5.5
integer, 5
```

Достигнется разное поведение на разных типах.

Подробнее про полиморфизм для заинтересованных.

>Магические методы

В классе можно объявить методы с особыми названиями, которые могут дать объектам класса особую функциональность. Такие методы называются ***magic methods***. Это специальные методы, с помощью которых вы можете добавить в ваши классы «магию».

Методы имеющие строгое название говорят Python как сделать ваш класс более удобным для использования. Например, с помощью магических методов вы можете добавить возможность складывать свои объекты классов:

```
class Vector3D:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    # Добавит возможность складывать объекты
    def __add__(self, other):
        return Vector3D(self.x + other.x, self.y + other.y, self.z + other.z)
```

В этом классе у нас всего 2 метода: `__init__` и `__add__`. Давайте создадим объекты класса `Vector3D`:

```
vec_1 = Vector3D(1, 3, 5)
vec_2 = Vector3D(-5, -3, 1)
sum_vec = vec_1 + vec_2 #Давайте просуммируем вектора
print(sum_vec)

Output:
<__main__.Vector3D object at 0x103875160>
# Какой некрасивый вывод
```

Метод `__add__` и правда позволил складывать вектора

```
print(sum_vec.z)
print(sum_vec.y)

Output:
6
0
```

Суммирование прошло правильно!

Ещё есть возможность красиво печатать вектор, с помощью метода `__str__`, по сути он превращает наш объект в строку:

```
class Vector3D:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __add__(self, other):
        return Vector3D(self.x + other.x, self.y + other.y, self.z + other.z)

    # добавит красивую печать
    def __str__(self):
        return f'({self.x}, {self.y}, {self.z})'
```

Сложим 2 вектора и сразу распечатаем результат:

```
vec_1 = Vector3D(1, 3, 5)
vec_2 = Vector3D(-5, -3, 1)
print(vec_1 + vec_2)

Output:
(-4, 0, 6)
```

И таких магических методов достаточно много, вы можете добавлять в свои классы возможность возводить в степень, делать логические операции, делать истинное деление, деление нацело и т.д

Обратите внимание, что все магические методы имеют интересный синтаксис, они всегда обрамлены двумя нижними подчеркиваниями (например, `__init__` или `__add__`)

Такие волшебные методы часто используются при разработке библиотек. Более подробно про них можно почитать здесь

>Множественное наследование

Python поддерживает множественное наследование. То есть, возможность у класса потомка наследовать функционал не от одного, а от нескольких родителей(которые перечисляются через запятую). Благодаря этому мы можем

создавать сложные структуры, сохраняя простой и легко-поддерживаемый код. Например, у нас есть класс автомобиля:

```
class Auto:
    def ride(self):
      print("Едет")
```

И есть класс для самолета:

```
class Airplane:
    def fly(self):
        print("Летит")
```

Теперь, если нам нужно запрограммировать летающий автомобиль, который будет летать и ездить, то мы вместо написания нового класса, можем просто унаследовать от уже существующих:

```
class FlyCar(Auto, Airplane):
    pass

a = FlyCar()
a.ride()
a.fly()
```

В данном примере, `Auto` и `Airplane` родительские классы `FlyCar`.

Классы-наследники могут использовать родительские методы. Но что, если у нескольких родителей будут одинаковые методы? Какой метод в таком случае будет использовать наследник?

Эта ситуация, так называемая проблема ромбовидного наследования(**The Diamond Problem**). Классический пример:

```
class A:
    def hello(self):
        print("it`s A")

class B(A):
    def hello(self):
        print("it`s B")
```

```
class C(A):
    def hello(self):
        print("it`s C")

class D(B, C):
    pass
```

Класс **D** является дочерним классом классов **B** и **C**. Классы **B** и **C** являются дочерними классами класса **A**. Давайте вызовем метод `hello()` у экрепляра класса **D**:

```
s = D()
s.hello()

Output:
it`s B
```

Мы получили надпись `“it`s B”` потому что он наследует свойство в соответствии с правилами MRO.

В 3 версии Python интерпретатор будет искать метод `hello()` в классе **B**(он указан первым родителем **D**), если его там нету - в классе **C**, потом в классе **A**, и так далее по иерархии. Сответсвенно, порядок в котором указываются родителя класса(при множественном наследовании) имеет значение. Давайте поменяем местами родительские классы класса D:

```
class A:
    def hello(self):
        print("it`s A")

class B(A):
    def hello(self):
        print("it`s B")

class C(A):
    def hello(self):
        print("it`s C")

class D(C, B):   # В предыдущем примере class D(B, C):
    pass
```

Давайте вызовем метод `hello()` у экрепляра класса **D**:

```
s = D()
s.hello()

Output:
it`s C
```

В Python можно посмотреть в каком порядке будут проинспектированы родительские классы при помощи метода класса `mro()`:

```
D.mro()

Output:
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.B'>,
<class '__main__.A'>, <class 'object'>]
```

В связи с наличием множественного наследования в Python есть идея **миксинов**(mixin, классы-примеси). Это маленькие классы, которые имеют не очень большой функционал и как-бы подмешиваются" к другим классам за счёт того, что они тоже становятся родителями этого класса. Идея миксинов предполагает их использование только вместе с другими классами(для кастомизации, расширения функционала и т.д)

Предположим, мы программируем класс для автомобиля. Мы хотим, чтобы у нас была возможность слушать музыку в машине. Конечно, можно просто добавить метод radio() в класс Car:

```
class Car:
    def ride(self):
        print("Едет")

    def radio(self, song):
        print("Сейчас играет: {} ".format(song))
```

Но а что если, у нас есть еще и телефон или любой другой девайс, с которого мы хотим слушать музыку. В таком случае, лучше вынести функционал проигрывания музыки в отдельный класс-миксин:

```
class MusicPlayerMixin:
    def radio(self, song):
```

```
        print("Сейчас играет: {}".format(song))
```

Всё, теперь мы можем "домешивать" этот класс в любой, где нужна функция проигрывания музыки:

```
class FlyCar(Auto, Airplane, MusicPlayerMixin):
   pass

a = FlyCar()
a.ride()
a.fly()
a.radio('Mozart - Queen of the Night')

Output:
Едет
Летит
Сейчас играет: Mozart - Queen of the Night
```

Целесообразно использовать миксины в ситуации, когда необходимо обеспечить какой-то класс дополнительной(но не очень важной) функциональностью или когда нужно добавить какую-то конкретную фичу большому количеству не связанных “родственными узами” классов

>Инкапсуляция

Суть инкапсуляции заключается в создании объекта, который содержит в себе некоторые данные методы для работы с этими данными, при этом во вне объекта реализация методов не выставляется.

Публичный интерфейс - набор атрибутов(свойств и методов) доступные снаружи объекта. Публичный интерфейс необходимо тщательно продумывать, т.к при его изменении может сломаться обратная совместимость. У компьютера из публичного интерфейса: монитор, клавиатура и мышь. Детали реализации в виде материнской платы, процессора и кулера скрыты от пользователя(это **приватный интерфейс**).

На этапе реализации и проработки идеи класса важно подумать: какие данные и методы необходимо скрыть от пользователя, чтобы у него не было к ним доступа. Этот процесс сокрытия реализации и называется **инкапсуляцией**(от In-Capsula). Простыми словами, инкапсуляция это управление способностью пользователя

видеть/изменять внутреннее содержимое класса.
Существует несколько уровней доступа, предоставляемых большинством ООП языков. Обобщая можно сказать что данные объекта могут быть:

- публичными (`public`)—данные доступны всем(монитор, клавиатура и мышь).
- приватными (`private`)—данные доступны только объекту/классу которому они принадлежат(материнская плата, процессор, кулер и т.п).

Чтобы указать приватные атрибуты нужно использовать специальное именование. Приватные атрибуты класса обозначаются одним нижним подчеркиванием перед название атрибута:

```
class Auto:
    def __init__(self, manufacturer, series):
        self._series = series
        self._manufacturer = manufacturer
        self.series = f'{self._manufacturer} {self._series}'
```

Создадим экземпляр класса:

```
p = Auto('Ford', 'Mustang')
print(p.series)

Output:
Ford Mustang
```

Свойства `_series` и `_manufacturer` являются приватными(т.е их использование вне этого класса не предполагается).

>Интерфейс

Классы позволяют проводить еще один трюк. Допустим, вы разрабатываете библиотеку для подключения к различным типам СУБД: к PostgreSQL, к ClickHouse, к MySQL и т.п.

Как главный разработчик, вы принимаете решение писать по классу на каждый тип СУБД и при этом обязать всех программистов реализовать функцию `.connect()` в этих классах. С помощью ООП можно наложить это обязательство не только словами, но и кодом:

```
import time

# Создаем класс с функцией, которую требуется реализовать
class Connectable:
    def connect(self, conn_uri: str):
        raise NotImplemented("you should override this method")

# И наследуемся от нее во всех классах
class PostgresqlConnection(Connectable):
    # теперь мы получили в наследство неисправную функцию
    # надо ее переопределить на исправную версию
    def connect(conn_uri: str):
        print('connecting to postgres')
        time.sleep(3)  # имитируем подключение к postgres :)
        print('connection done')
```

Класс `Connectable` в этом случае будет называться *интерфейсом*. Грубо говоря, интерфейс - это класс, который ничего не делает по существу и только дает обязательства на реализацию того или иного метода.

Интерфейсы используются, чтобы напоминать другим разработчикам, какие методы они должны реализовать для нормальной работы класса в остальной системе.

>SOLID-принципы

SOLID — это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании — Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion. Пять принципов, по которым стоит строить классы. Запоминать их не обязательно, но лучше держать в голове и потихоньку к ним приближаться.

1. **Принцип единственной ответственности** (single responsibility). У каждого объекта должна быть только одна ответственность. Все поведение этого объекта должно быть направлено на обеспечение этой самой ответственности и никаких других. Это позволяет избежать внесения непредвиденных изменений в систему. Подробнее про SRP.
2. **Принцип открытости/закрытости (**Open-closed**)**. Он утверждает, что классы должны быть открыты для расширения, но при этом закрыты для изменения. Этот принцип является важным, потому что внесение изменений в

существующие компоненты системы может также привести к непредвиденным изменениям в работе самой этой системы. Подробнее про OCP.

3. **Принцип подстановки Лисков** (Liskov substitution). Функции, которые используют базовый тип, должны иметь возможность использовать его подтипы не зная об этом. Согласно этому принципу, поведение подклассов некоторого класса не должно противоречить поведению самого этого класса. . Объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы. Подробнее про LSP.
4. **Принцип разделения интерфейса (**Interface segregation**)**. Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения. Не следует делать слишком сложный интерфейс какого-то объекта: у каждого компонента должна быть возможность доступа лишь к той части функциональности, которая необходима именно ему. Если интерфейс не может этого обеспечить, то его следует разделить на более мелкие. Подробнее про ISP.
5. **Принцип инверсии зависимостей** (Dependency inversion). Он говорит, что, во-первых, модули верхних уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракций. Во-вторых, абстракции не должны зависеть от деталей, это детали должны зависеть от абстракций. Подробнее про DIP.

Использование SOLID-принципов позволит сделать вашу систему более структурированной, гибкой и удобной в использовании и обслуживании.