# Основы ООП в Python

## FullCode Academy — Group 1

Преподаватель: Islam Duishobaev

# Дорожная Карта Мастер-Класса

01

## Понимание Основ

Что такое **классы и объекты**, а также **атрибуты и методы**.

02

## Принципы Расширяемости

Исследование **наследования** и **абстрактных классов** для повторного использования кода.

03

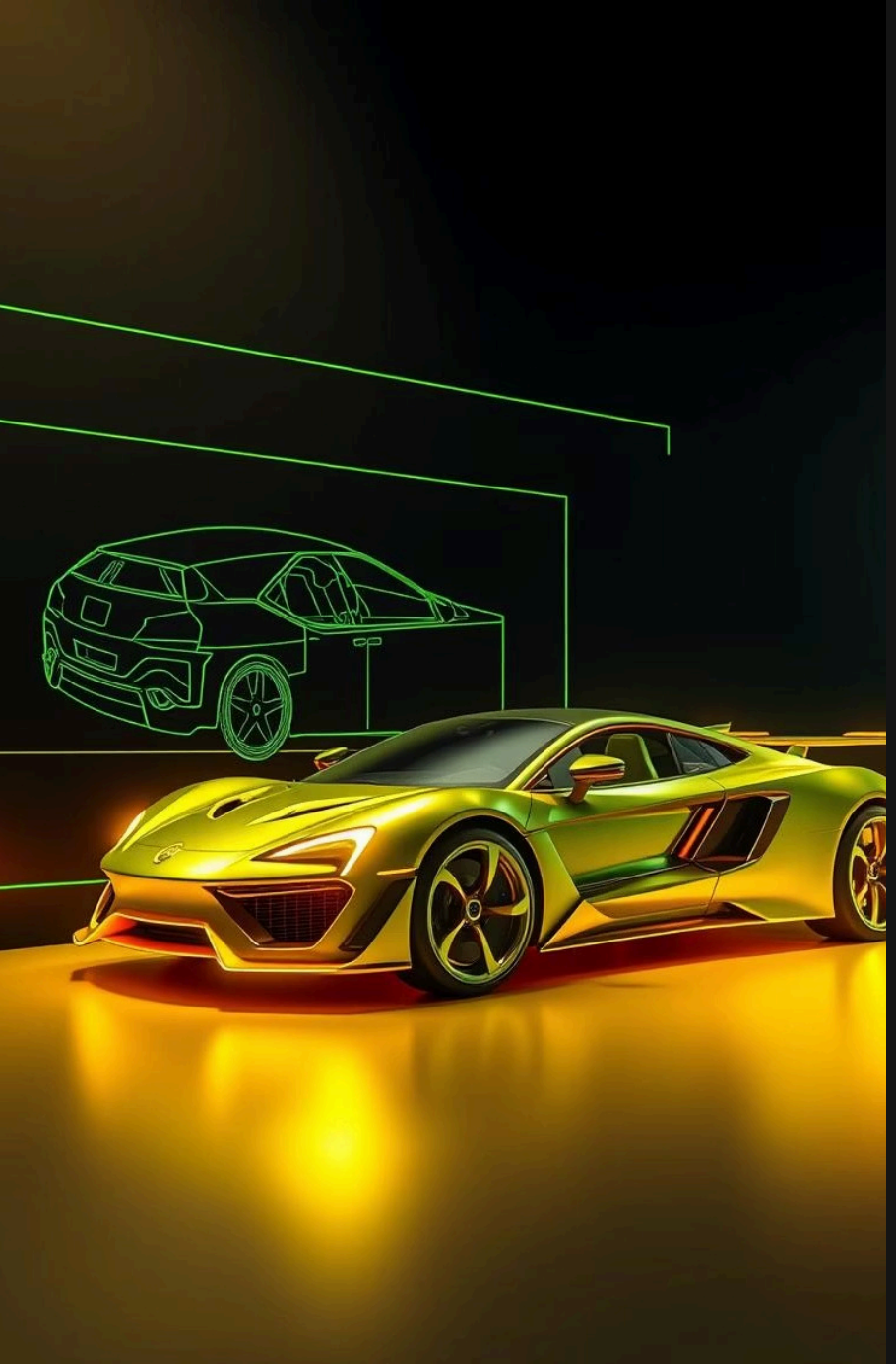
## Поведение Объектов

Знакомство с **инкапсуляцией, полиморфизмом** и **магическими методами**.

04

## Практическое Применение

Как применять ООП для **структурирования кода** и создания элегантных решений.



# Класс и Объект: Фундамент ООП

**Класс** — это не просто чертеж, это подробный шаблон для создания чего-то. Думайте о нем как о рецепте или архитектурном плане. Он определяет структуру и поведение, которые будут у всех его экземпляров.

**Объект** — это реальное воплощение этого шаблона, конкретный экземпляр класса. Если класс — это рецепт торта, то объект — это уже испеченный торт, который можно попробовать.

```
class Car:
 def __init__(self, brand, model):
 self.brand = brand
 self.model = model

my_car = Car("Toyota", "Camry")
```

# Анатомия Объекта: Атрибуты и Методы

## Атрибуты: Состояние Объекта

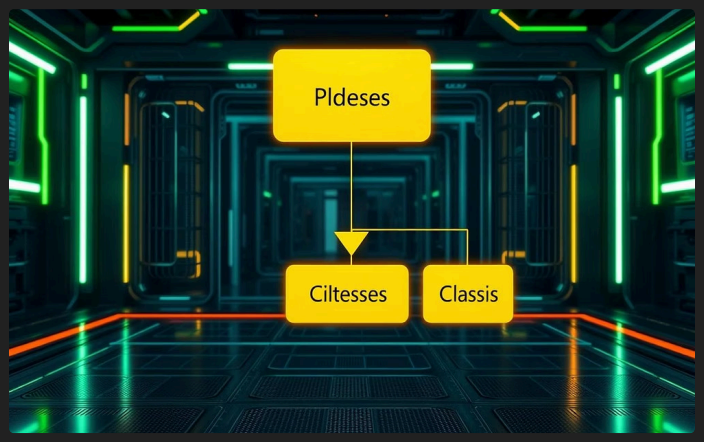
Это переменные, которые хранят данные, определяющие текущее состояние объекта. Если наш объект — это машина, то ее атрибутами могут быть **цвет, марка, год выпуска** или **текущая скорость**. Они описывают "что" представляет собой объект.

## Методы: Поведение Объекта

Это функции, которые принадлежат классу и выполняют действия, связанные с объектом. Для машины методами могут быть **завести(), остановиться(), поехать()**. Они определяют "что" объект может делать.

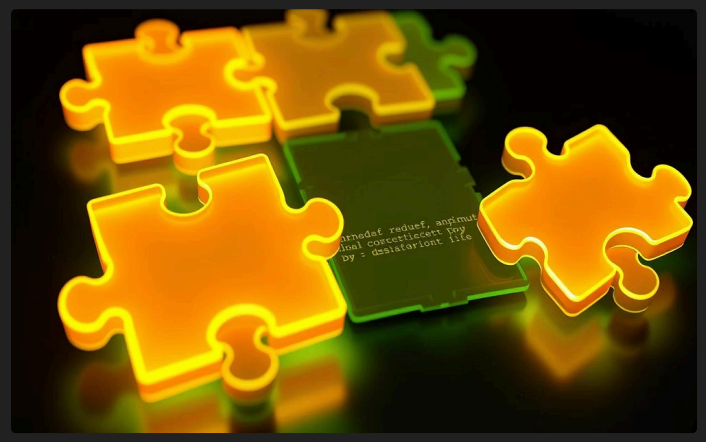
```
class Car:
 def start(self):
 print("Машина завелась!")

my_car = Car("Toyota", "Camry")
my_car.start() # Вывод: Машина завелась!
```

# Наследование: Расширение Возможностей




## Родитель и Дочерний Класс

Наследование позволяет создавать новый класс (дочерний) на основе уже существующего (родительского), формируя иерархию.



## Повторное Использование Кода

Это ключевой принцип повторного использования кода, позволяющий строить логическую иерархию в программе.

## Расширение и Переопределение

Дочерний класс перенимает атрибуты и методы родителя, а также может добавлять новые или изменять унаследованные.

```
class Vehicle:
 def __init__(self, name):
 self.name = name

class Car(Vehicle):
 def honk(self):
 print("Бип!")

my_car = Car("Toyota")
my_car.honk() # Вывод: Бип!
```

# Абстрактные Классы: Контракты для Подклассов

Абстрактный класс — это особенный тип класса, который **не может быть инстанцирован напрямую**. Его основная цель — служить шаблоном или "контрактом" для своих подклассов, обязывая их реализовать определенные методы. Это гарантирует, что все производные классы будут иметь ожидаемое поведение, поддерживая единообразие в архитектуре программы.

| 1 | 2 | 3 |
| --- | --- | --- |
| **Нельзя инстанцировать** | **Обязательная реализация** | **Использование `ABC` и `@abstractmethod`** |
| Вы не можете создать объект напрямую из абстрактного класса. | Подклассы должны предоставить свою реализацию абстрактных методов. | В Python для этого используются модули `abc` и декоратор `@abstractmethod`. |

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
 @abstractmethod
 def start(self):
 pass

class Car(Vehicle):
 def start(self):
 print("Машина завелась!")

my_car = Car()
my_car.start() # Вывод: Машина завелась!
```

# Инкапсуляция: Скрытие Деталей

Инкапсуляция — это принцип ООП, который позволяет **скрыть внутреннее состояние объекта** от внешнего мира и предоставить контролируемый доступ к нему через публичные методы. Это повышает безопасность данных и делает код более поддерживаемым, так как изменения во внутренней реализации не затрагивают внешний интерфейс.

- Защита данных: Предотвращает несанкционированный доступ и изменение.
- Приватные атрибуты: В Python обозначаются одним (_) или двумя (__) подчеркиваниями.
- Геттеры и Сеттеры: Методы для контролируемого доступа к приватным атрибутам.

```
class Car:
 def __init__(self, brand):
 self.__brand = brand # приватный атрибут

 def get_brand(self):
 return self.__brand
```

# Полиморфизм: Единый Интерфейс, Разное Поведение

Полиморфизм (от греч. "много форм") позволяет объектам разных классов реагировать на один и тот же вызов метода по-разному, в зависимости от их собственного типа. Это означает, что вы можете взаимодействовать с различными объектами через **единый интерфейс**, не заботясь о их специфической реализации. Это делает код более гибким и расширяемым.

### Класс `Dog`

Метод `speak()` выводит "Гав!".

### Класс `Cat`

Метод `speak()` выводит "Мяу!".

### Единый вызов

Вызывая `animal.speak()`, получаем разный результат.

```
class Dog:
 def speak(self):
 print("Гав!")

class Cat:
 def speak(self):
 print("Мяу!")

for animal in [Dog(), Cat()]:
 animal.speak()
```

# Магические Методы: Особые Способности Объектов

Магические методы, или **Dunder-методы** (от "double underscore"), — это специальные методы в Python, имена которых начинаются и заканчиваются двойными подчеркиваниями (например, `__init__`, `__str__`). Они позволяют изменять стандартное поведение объектов и интегрироваться с встроенными функциями и операторами Python.

## `__str__()`: Строковое представление

Определяет, как объект будет преобразован в строку при использовании `print()` или `str()`.

```
class Car:
    def __init__(self, brand):
        self.brand = brand
    def __str__(self):
        return f"Машина: {self.brand}"

my_car = Car("Toyota")
print(my_car) # Вывод: Машина: Toyota
```

## Другие Примеры: Перегрузка Операторов

- `__add__`: для оператора **+**
- `__len__`: для функции **len()**
- `__getitem__`: для доступа по индексу (как в списках)
- `__call__`: позволяет объекту быть вызываемым как функция

# Практическое Применение: Сложение Точек

Давайте закрепим магические методы на практике, создав класс `Point`, который представляет точку в двухмерном пространстве. Мы реализуем магический метод `__add__`, чтобы можно было складывать два объекта `Point` как обычные числа.

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

    def __add__(self, other):
        # Метод __add__ вызывается, когда вы используете оператор '+'
        return Point(self.x + other.x, self.y + other.y)

# Создаем две точки
p1 = Point(1, 2)
p2 = Point(3, 4)

# Складываем точки с помощью оператора '+'
p3 = p1 + p2

# Выводим координаты новой точки
print(f"Новая точка: ({p3.x}, {p3.y})") # Вывод: Новая точка: (4, 6)
```

Этот пример демонстрирует, как магические методы делают ваш код более **интуитивно понятным** и **совместимым** с естественным поведением Python.