Лекция 1. Понятие класса и объекта класса

**Оглавление**

[Введение 2](#_Toc180416165)

[1. Основы объектно‑ориентированного программирования 4](#_Toc180416166)

[2. Понятие класса и объекта класса 7](#_Toc180416167)

[3. Методы класса. Конструкторы и деструкторы 9](#_Toc180416168)

[4. Атрибуты класса 14](#_Toc180416169)

[Заключение 15](#_Toc180416170)

[Список литературы 17](#_Toc180416171)

**Понятие класса и объекта класса**

**Цель:** формирование знаний по основам объектно-ориентированного программирования высокоуровневого языка программирования.

**План лекции:**Введение.1. Основы объектно‑ориентированного программирования.  
2. Понятие класса и объекта класса.  
3. Методы класса. Конструкторы и деструкторы.4. Атрибуты класса.Заключение.Список литературы.

# **Введение**

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции "объектов", которые могут содержать данные и код для работы с этими данными. В Python, как и в других языках программирования, ООП позволяет создавать структуры, которые более интуитивно понятны и легко поддерживаются. Эта парадигма программирования широко используется в разработке программного обеспечения, поскольку она позволяет создавать более гибкие и масштабируемые приложения.

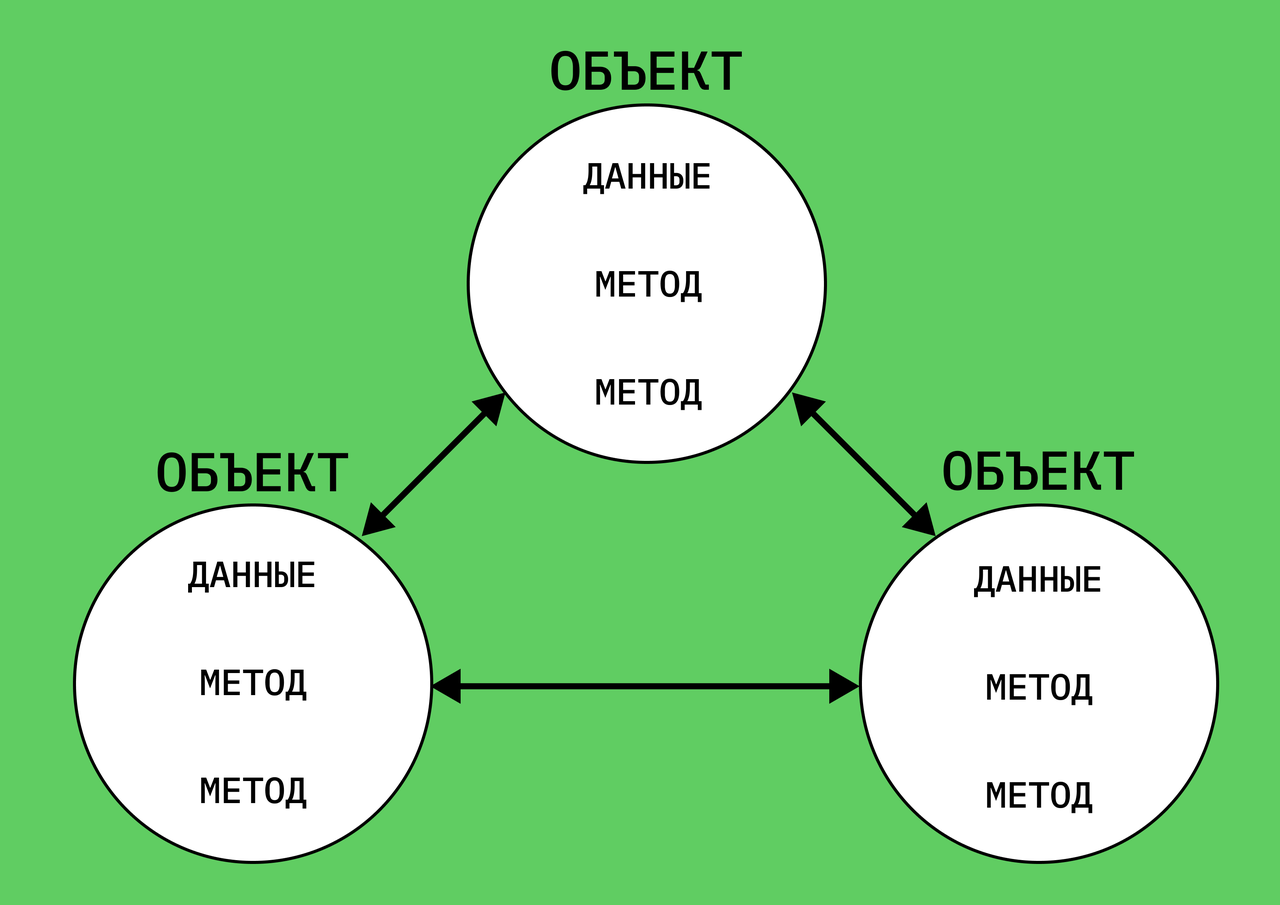
ООП помогает разбивать сложные задачи на более мелкие, управляемые части, что делает код более организованным и повторно используемым. Это особенно важно при работе над большими проектами, где нужно поддерживать и обновлять код на протяжении длительного времени. В этой статье мы рассмотрим основные концепции ООП в Python, такие как классы и объекты, а также наследование, инкапсуляцию и полиморфизм. Эти концепции являются фундаментальными для понимания ООП и их применение может значительно улучшить качество вашего кода.

# **Основы объектно‑ориентированного программирования**

Объектно-ориентированное программирование, или ООП — это одна из парадигм разработки. Парадигмой называют набор правил и критериев, которые соблюдают разработчики при написании кода. Если представить, что код — это рецепт блюда, то парадигма — то, как рецепт оформлен в кулинарной книге. Парадигма помогает стандартизировать написание кода. Это снижает риск ошибок, ускоряет разработку и делает код более читабельным для других программистов.

Суть понятия объектно-ориентированного программирования в том, что все программы, написанные с применением этой парадигмы, состоят из объектов. Каждый объект — это определённая сущность со своими данными и набором доступных действий.

Например, нужно написать для интернет-магазина каталог товаров. Руководствуясь принципами ООП, в первую очередь нужно создать объекты: карточки товаров. Потом заполнить эти карточки данными: названием товара, свойствами, ценой. И потом прописать доступные действия для объектов: обновление, изменение, взаимодействие.

Рис.1. Так схематично выглядит программа, написанная по парадигме ООП

Кроме ООП, существуют и другие парадигмы. Из них наиболее распространена функциональная (*раздел 1 текущей дисциплины*), в которой работают не с объектами, а с функциями. Если использовать функциональную парадигму, чтобы сделать каталог товаров, то начинать нужно не с карточек, а с функций, заполняющих эти карточки. То есть объект будет не отправной точкой, а результатом работы функции.

Обычно написать функцию быстрее, чем создавать объекты и прописывать взаимодействие между ними. Но если объём кода большой, работать с разрозненными функциями сложно.

**Зачем нужно ООП?**

ООП помогает организовывать код так, чтобы он был более структурированным, понятным и поддерживаемым. Это особенно полезно для крупных проектов, где важна модульность и повторное использование кода.

**Структура объектно-ориентированного программирования:**

В коде, написанном по парадигме ООП, выделяют четыре основных элемента:

1. Объект.

Часть кода, которая описывает элемент с конкретными характеристиками и функциями.

2. Класс.

Шаблон, на базе которого можно построить объект в программировании.

3. Метод.

Функция внутри объекта или класса, которая позволяет взаимодействовать с ним или другой частью кода.

4. Атрибут.

Характеристики объекта в программировании — например, цена, производитель или объём оперативной памяти.

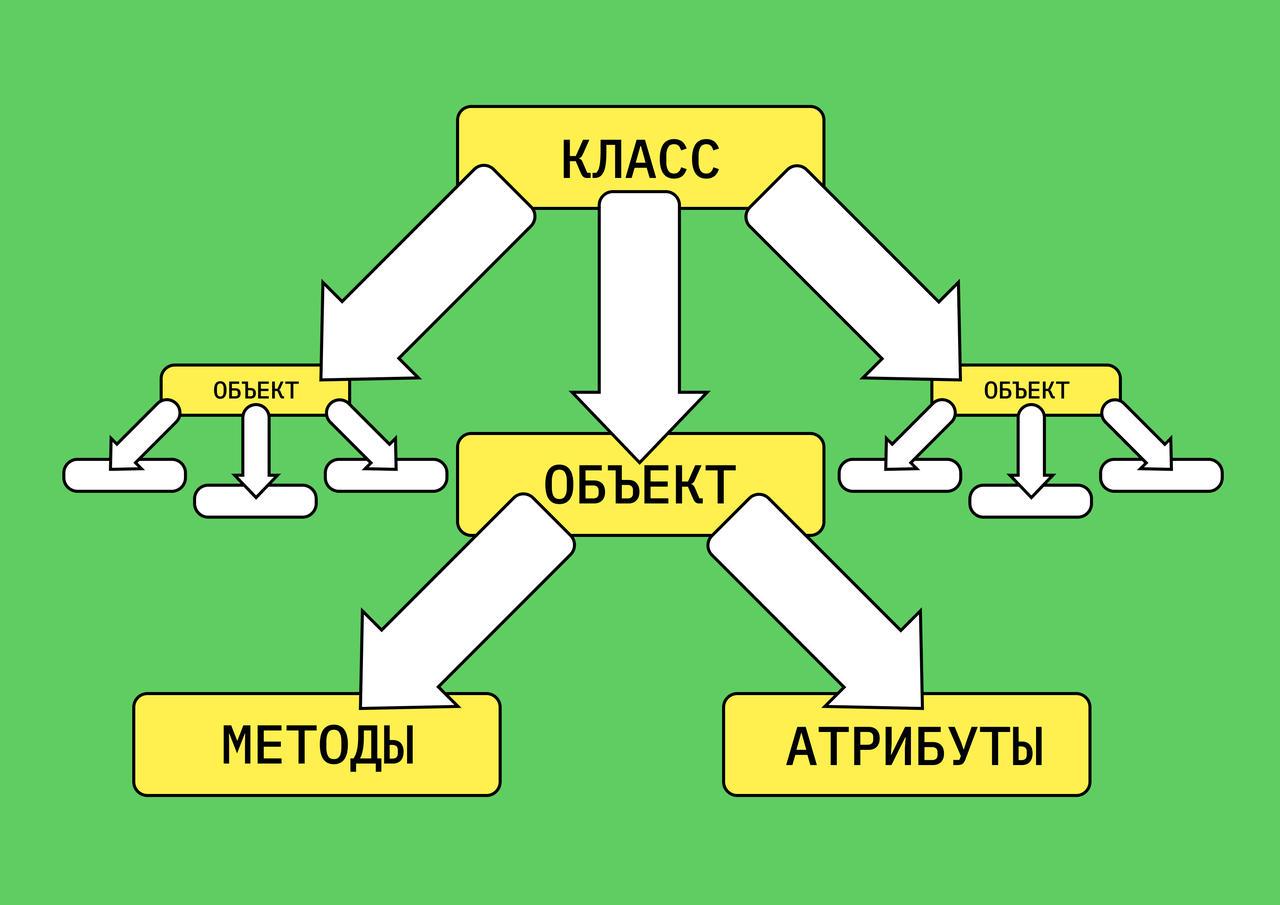


Рис. 2. Так связаны между собой элементы ООП

**Преимущества и недостатки ООП:**

**Преимущества:**

**● В парадигме объектов легче писать код.** Удобно один раз создать класс или метод, а потом его использовать. Не нужно повторно переписывать десятки строк кода. Можно пользоваться специальными рекомендациями по написанию ООП-кода — SOLID.

**● Читать код гораздо проще.** Даже в чужом коде обычно сразу видны конкретные объекты и методы, их удобно искать, чтобы посмотреть, что именно они делают.

**● Код легче обновлять.** Класс или метод достаточно изменить в одном месте, чтобы он изменился во всех наследуемых классах и объектах. Не нужно переписывать каждый объект отдельно, выискивая, где именно в коде он расположен.

**● Программистам удобнее работать в команде.** Разные люди могут отвечать за разные объекты и при этом пользоваться плодами трудов коллег.

**● Код можно переиспользовать.** Один раз написанный класс или объект можно затем переносить в другие проекты. Достаточно однажды написать объект «Кнопка заказа» и потом можно вставлять его в почти неизменном виде в разные каталоги товаров и мобильные приложения.

**● Шаблоны проектирования.** Именно на базе ООП построены готовые решения для взаимодействия классов друг с другом, которые позволяют не писать этот код с нуля, а взять шаблон.

**Недостатки:**

**● Сложность в освоении.** ООП сложнее, чем функциональное программирование. Для написания кода в этой парадигме нужно знать гораздо больше. Поэтому перед созданием первой рабочей программы придётся освоить много информации: разобраться в классах и наследовании, научиться писать публичные и внутренние функции, изучить способы взаимодействия объектов между собой.

● **Громоздкость.** Там, где в функциональном программировании хватит одной функции, в ООП нужно создать класс, объект, методы и атрибуты. Для больших программ это плюс, так как структура будет понятной, а для маленьких может оказаться лишней тратой времени.

**● Низкая производительность.** Объекты потребляют больше памяти, чем простые функции и переменные. Скорость компиляции от этого тоже страдает.

Получается, что основная функция объектно-ориентированного программирования — облегчить написание больших, сложных программ, над которыми трудятся группы разработчиков.

# **Понятие класса и объекта класса**

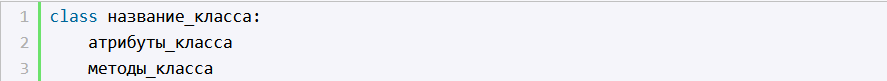
**Что такое класс?**

Класс — это своего рода чертеж или шаблон для создания объектов. Он описывает, какими характеристиками (атрибутами) и поведением (методами) будут обладать объекты, созданные на его основе.

Python имеет множество встроенных типов, например, int, str и так далее, которые мы можем использовать в программе. Но также Python позволяет определять собственные типы с помощью классов. Класс представляет некоторую сущность. Конкретным воплощением класса является объект.

Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. Человек может выполнять некоторые действия - ходить, бегать, думать и т.д. То есть это представление, которое включает набор характеристик и действий, можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек будет представлять объект этого класса.

В языке Python класс определяется с помощью ключевого слова class:



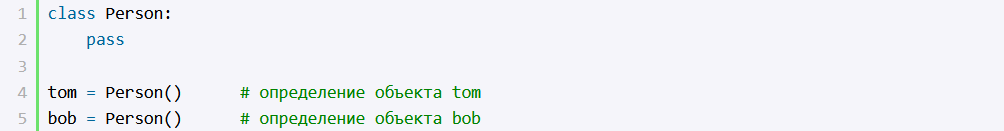
Внутри класса определяются его атрибуты, которые хранят различные характеристики класса, и методы - функции класса.

Создадим простейший класс:



В данном случае определен класс Person, который условно представляет человека. В данном случае в классе не определяется никаких методов или атрибутов. Однако поскольку в нем должно быть что-то определено, то в качестве заменителя функционала класса применяется оператор pass. Этот оператор применяется, когда синтаксически необходимо определить некоторый код, однако исходя из задачи код нам не нужен, и вместо конкретного кода вставляем оператор pass.

После создания класса можно определить объекты этого класса. **Объект** – экземпляр класса. Если класс — это чертеж, то объект — это конкретная реализация этого чертежа. Например, если у нас есть класс "Автомобиль", то объекты этого класса могут быть "BMW", "Audi" и т. д. Например:



# **Методы класса. Конструкторы и деструкторы**

**Конструкторы**

Итак, для создания объекта класса используется конструктор. Конструктор — это специальный метод класса, который вызывается при создании нового объекта. Он инициализирует объект, задавая его начальные значения атрибутов.

Так, выше когда мы создавали объекты класса Person, мы использовали конструктор по умолчанию, который не принимает параметров и который неявно имеют все классы. Однако мы можем явным образом определить в классах конструктор с помощью специального метода, который называется \_\_init\_\_() (по два прочерка с каждой стороны). К примеру, изменим класс Person, добавив в него конструктор:



Итак, здесь в коде класса Person определен конструктор - функция \_\_init\_\_. Конструктор должен принимать как минимум один параметр ссылку на текущий объект - self. Обычно конструкторы применяются для определения действий, которые должны производиться при создании объекта.

Теперь при создании объекта:



выполняется вызов конструктора \_\_init\_\_() из класса Person, который выведет на консоль строку "Создание объекта Person".

Стоит отметить, что конструктор фактически представляет обычную функцию, только для вызова конструктора используется не \_\_init\_\_, а название класса. Кроме того, при вызове конструктора параметру self явным образом не передается никакого значения. При выполнении программы Python динамически будет определять self.

**Методы классов**

Методы класса фактически представляют функции, которые определенны внутри класса и которые определяют его поведение. Например, определим класс Person с одним методом:



Здесь определен метод say\_hello(), который условно выполняет приветствие - выводит строку на консоль. При определении методов любого класса, как и конструктора, первый параметр метода представляет ссылку на текущий объект, который согласно условностям называется self. Через эту ссылку внутри класса мы можем обратиться к функциональности текущего объекта. Но при самом вызове метода этот параметр не учитывается.

Используя имя объекта, мы можем обратиться к его методам. Для обращения к методам применяется нотация точки - после имени объекта ставится точка и после нее идет вызов метода:

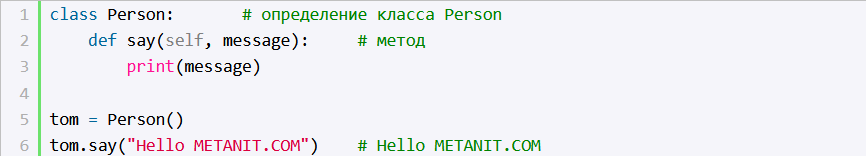


Например, обращение к методу say\_hello() для вывода приветствия на консоль:



В итоге данная программа выведет на консоль строку "Hello".

Если метод должен принимать другие параметры, то они определяются после параметра self, и при вызове подобного метода для них необходимо передать значения:

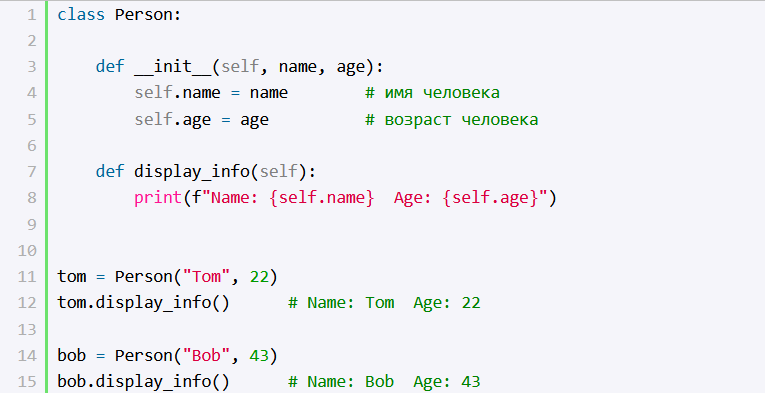


Здесь определен метод say(). Он принимает два параметра: self и message. И для второго параметра - message при вызове метода необходимо передать значение.

Для обращения к атрибутам и методам объекта внутри класса в его методах также применяется слово self:



Например, следующий класс Person:



Здесь определяется метод display\_info(), который выводит информацию на консоль. И для обращения в методе к атрибутам объекта применяется слово self: self.name и self.age

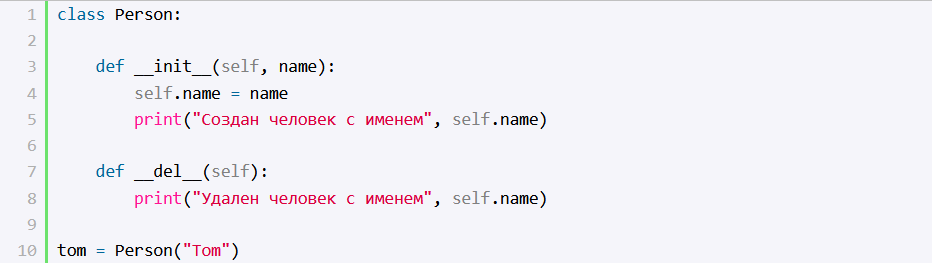
В итоге мы получим следующий консольный вывод:



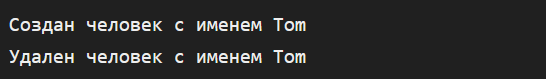
**Деструкторы**

Кроме конструкторов классы в Python также могут определять специальные методы – деструкторы, которые вызываются при удалении объекта. Деструктор представляет собой метод \_\_del\_\_(self), в который, как и в конструктор, передается ссылка на текущий объект. В деструкторе определяются действия, которые надо выполнить при удалении объекта, например, освобождение или удаление каких-то ресурсов, которые использовал объект.

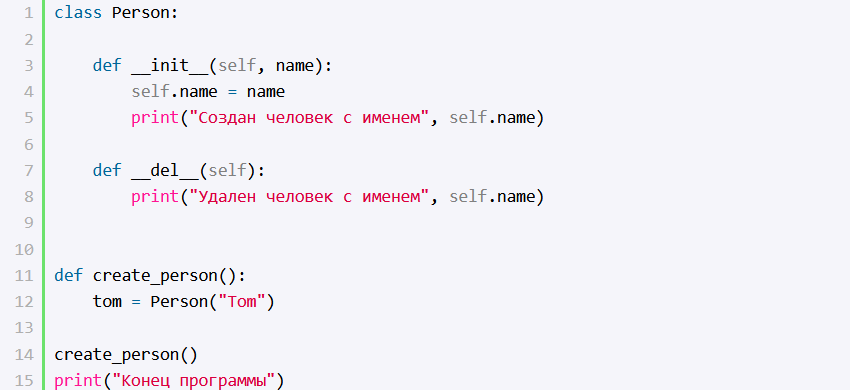
Деструктор вызывается автоматически интерпретатором, нам не нужно его явным образом вызывать. Простейший пример:



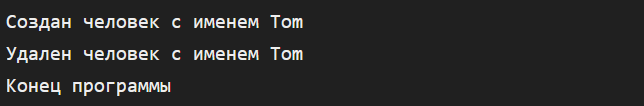
Здесь в деструкторе просто выведится уведомление об удалении объекта Person. Программа создает один объект Person и хранит ссылку на него в переменной tom. Создание объекта вызовет выполнение конструктора. При завершении программы автоматически будет выполняться деструктор объекта tom. В итоге консольный вывод программы будет следующим:



Другой пример:



Здесь объект Person создается и используется внутри функции create\_person, поэтому жизнь создаваемого объекта Person ограничена областью этой функции. Соответственно, когда функция завершит свое выполнение, у объекта Person будет вызываться деструктор. В итоге мы получим следующий консольный вывод:



### **Зачем нужны конструкторы и деструкторы?**

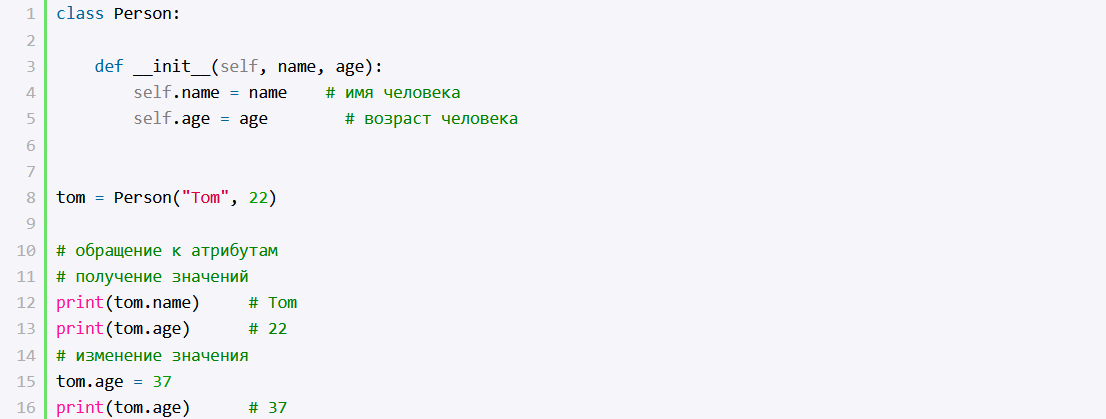
Конструкторы и деструкторы помогают управлять жизненным циклом объекта:

* **Конструктор** инициализирует объект и задает ему начальные параметры.
* **Деструктор** завершает работу объекта и освобождает занятые ресурсы.

# **Атрибуты класса**

Атрибут – характеристики объекта в программировании — например, цена, производитель или объём оперативной памяти. В классе прописывают, что такие атрибуты есть, а в объектах с помощью методов заполняют эти атрибуты данными.

Атрибуты хранят состояние объекта. Для определения и установки атрибутов внутри класса можно применять слово self. Например, определим следующий класс Person:



# **Заключение**

ООП — это парадигма разработки, набор правил и критериев, по которым пишут код. Её суть в том, что весь код состоит из объектов, которые взаимодействуют друг с другом. Существуют и другие парадигмы, например, функциональное программирование.

В ООП выделяют четыре основных элемента: классы, объекты, методы и атрибуты.

Объектно-ориентированный подход к программированию строится на трёх основных принципах: наследование, инкапсуляция и полиморфизм.

Программы, созданные по принципам ООП, более структурированные, легче читаются и хорошо масштабируются. При этом они сложнее в написании. Давайте рассмотрим несколько примеров:

**Пример 1: Разработка игры**

Создание компьютерной игры часто требует использования ООП. В такой игре можно выделить множество объектов, таких как персонажи, враги, предметы, уровни и т. д.

1. **Классы**:
   * Класс "Персонаж" может включать атрибуты, такие как здоровье, скорость, сила атаки, и методы, такие как атака, защита, передвижение.
   * Класс "Враг" может быть наследником класса "Персонаж" с добавлением специфических атрибутов и методов, например, искусственного интеллекта.
2. **Объекты**:
   * Конкретный персонаж игры (например, рыцарь) является объектом класса "Персонаж".
   * Конкретный враг (например, дракон) является объектом класса "Враг".
3. **Методы и атрибуты**:
   * Методы включают действия, которые могут выполнять персонажи и враги (например, атака, передвижение).
   * Атрибуты включают характеристики (например, здоровье, уровень).

Использование ООП позволяет легко добавлять новых персонажей и врагов, модифицировать их поведение и атрибуты, а также повторно использовать код в разных частях игры.

**Пример 2: Управление библиотекой**

Представьте себе систему управления библиотекой, которая позволяет отслеживать книги, читателей и выдачу книг.

1. **Классы**:
   * Класс "Книга" включает атрибуты, такие как название, автор, ISBN, и методы, такие как проверка наличия в библиотеке.
   * Класс "Читатель" включает атрибуты, такие как имя, номер читательского билета, и методы, такие как взятие и возврат книги.
2. **Объекты**:
   * Конкретная книга (например, "Война и мир") является объектом класса "Книга".
   * Конкретный читатель (например, Иван Иванов) является объектом класса "Читатель".
3. **Методы и атрибуты**:
   * Методы включают действия, которые могут выполнять книги и читатели (например, выдача книги, возврат книги).
   * Атрибуты включают характеристики (например, название книги, имя читателя).

ООП позволяет создать гибкую и легко модифицируемую систему, которая упрощает добавление новых функций и поддержание кода.

**Пример 3: Разработка веб-приложения**

Веб-приложения часто требуют взаимодействия с различными объектами, такими как пользователи, заказы, продукты и т. д.

1. **Классы**:
   * Класс "Пользователь" включает атрибуты, такие как имя, электронная почта, и методы, такие как регистрация, авторизация.
   * Класс "Продукт" включает атрибуты, такие как название, цена, и методы, такие как добавление в корзину, покупка.
2. **Объекты**:
   * Конкретный пользователь (например, John Doe) является объектом класса "Пользователь".
   * Конкретный продукт (например, смартфон) является объектом класса "Продукт".
3. **Методы и атрибуты**:
   * Методы включают действия, которые могут выполнять пользователи и продукты (например, авторизация, покупка).
   * Атрибуты включают характеристики (например, имя пользователя, цена продукта).

ООП позволяет легко масштабировать веб-приложение, добавлять новые функции и обеспечивать безопасность и стабильность кода.

**Преимущества ООП в этих ситуациях:**

* **Модульность**: легко разделить код на модули, каждый из которых отвечает за свою функциональность.
* **Повторное использование кода**: ООП позволяет повторно использовать классы и методы в разных частях программы.
* **Гибкость**: легко добавлять новые функции и объекты без необходимости изменять существующий код.
* **Удобство обслуживания**: классы и объекты делают код более читаемым и упрощают его обслуживание.

# **Список литературы**

1. <https://practicum.yandex.ru/blog/obektno-orientirovannoe-programmirovanie/>
2. <https://metanit.com/python/tutorial/7.1.php>
3. https://sky.pro/wiki/python/osnovy-oop-v-python/