**Знакомство с классами. Атрибуты класса. Конструктор. Что такое self? Наследование.**

План:

1. Введение.
2. Объявление и создание классов.
3. Атрибуты класса и экземпляра класса.
4. Методы класса.
5. Конструктор класса и ключевое слово «self».
6. Наследование.

**1.Введение**

**Что такое классы и зачем они нужны**

Классы в Python являются основой объектно-ориентированного программирования (ООП), что представляет собой подход к программированию, основанный на концепции «объектов», имеющих данные и действия (методы), которые могут выполняться над данными.

Класс определяет структуру объекта, его свойства и методы. Он служит как шаблон для создания объектов (экземпляров), каждый из которых является отдельным элементом с данными (атрибутами) и операциями (методами), которые можно выполнить над данными.

Классы в Python используются для следующих целей:

1. **Инкапсуляция**. Они позволяют скрывать и упаковывать данные внутри объекта, защищая их от несанкционированного доступа и изменений.

2. **Наследование**. Классы предоставляют возможность создавать новые классы на основе существующих, что позволяет повторно использовать код и упрощает разработку программ.

3. **Полиморфизм**. Они позволяют обрабатывать различные типы данных с помощью одного и того же интерфейса, что делает код более гибким и универсальным.

4. **Модульность и организация кода**. ООП и использование классов помогают группировать код, связанный с определенной функциональностью или сущностью, что делает программы более структурированными и понятными.

В общем, использование классов делает ваш код более структурированным, повторно используемым и гибким, что облегчает разработку, отладку и поддержку вашей программы

**Основы объектно-ориентированного программирования (ООП).**

Основы объектно-ориентированного программирования (ООП) включают следующие концепции:

1. **Классы (Classes)**. Класс — это шаблон или схема для создания объектов. Он описывает свойства (атрибуты) и методы (операции), которые будут общими для всех объектов, созданных на основе класса.

2. **Объекты (Objects)**. Объект — это конкретный экземпляр класса. Он содержит свойства и методы, определенные классом. Объекты также могут иметь уникальные значения свойств, которые отличаются от значений по умолчанию, установленных классом.

class Dog:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def bark(self):

print("Woof!")

my\_dog = Dog("Max", 3)

my\_dog.bark() # Output: Woof!

3. **Инкапсуляция (Encapsulation)**. Инкапсуляция заключается в объединении данных (атрибутов) и методов, которые манипулируют данными, в одной структуре (классе). Это защищает данные от несанкционированного доступа и позволяет управлять ими только с помощью методов объекта.

class BankAccount:

def \_\_init\_\_(self):

self.\_\_balance = 0

def deposit(self, amount):

self.\_\_balance += amount

def withdraw(self, amount):

if amount <= self.\_\_balance:

self.\_\_balance -= amount

# Мы не можем напрямую изменить \_\_balance, потому что оно скрыто.

account = BankAccount()

account.deposit(100)

account.withdraw(50)

4. **Наследование (Inheritance)**. Наследование позволяет создавать новые классы на основе существующих. Это делает код переиспользуемым и упрощает разработку программ.

class Animal:

def speak(self):

pass

class Dog(Animal):

def speak(self):

print("Woof!")

class Cat(Animal):

def speak(self):

print("Meow!")

dog = Dog()

cat = Cat()

dog.speak() # Output: Woof!

cat.speak() # Output: Meow!

5. **Полиморфизм (Polymorphism)**. Полиморфизм позволяет обрабатывать объекты разных классов, но с одинаковыми методами, как однотипные. Таким образом, код становится более гибким и расширяемым.

animals = [Dog(), Cat()]

for animal in animals:

animal.speak() # Output: Woof!, Meow!

1. **Объявление и создание классов**

**Как объявить класс с помощью ключевого слова "class":**

Объявление класса с помощью ключевого слова "class" является стандартным способом создания класса в объектно-ориентированных языках программирования.

Чтобы объявить класс, сначала используется ключевое слово `class`, затем указывается имя класса (желательно использовать стиль CamelCase) и двоеточие. Дальше внутри класса вы можете определить его атрибуты (свойства) и методы (функции), которые будут работать только в пределах этого класса.

Пример объявления класса:

class MyClass:

x = 5

def print\_hello(self):

print("Hello, world!")

В этом примере мы создали класс под названием «MyClass». Его атрибут «x» хранит значение 5, и есть метод «print\_hello()», который выводит «Hello, world!» на экран. Здесь «self» – это ссылка на экземпляр класса, она используется для доступа к атрибутам и методам объекта внутри самого класса.

Для того чтобы создать объект (экземпляр) этого класса, используйте следующий код:

my\_object = MyClass()

Теперь у вас есть объект «my\_object», который имеет доступ к атрибутам и методам класса «MyClass». Обратиться к атрибутам и методам объекта можно через точку. Например:

print(my\_object.x)

my\_object.print\_hello()

Выше указанный код выведет следующее:

5

Hello, world!

**Создание объектов (экземпляров класса) и их использование.**

Объектно-ориентированное программирование (ООП) - это парадигма программирования, в которой основной объект - это класс, определяющий структуру и поведение, общее для группы объектов.

Класс можно рассматривать как чертеж, который содержит атрибуты (переменные) и методы (функции), работающие с этими атрибутами. Объекты (экземпляры класса), создаваемые на основе класса, наследуют свойства и поведение класса и могут быть адаптированы к определенным задачам.

1. Создание класса.

Для начала, нужно определить класс, который будет моделировать общую структуру и поведение для объектов. Например, создадим класс `Dog` с атрибутами `name` и `breed` и методом `bark`.

class Dog:

def \_\_init\_\_(self, name, breed):

self.name = name

self.breed = breed

def bark(self):

print("Woof Woof!")

2. Создание объектов (экземпляров класса).

Теперь, когда у нас есть определенный класс `Dog`, мы можем создать на его основе объекты (экземпляры). Для этого используется конструктор класса `\_\_init\_\_()`, который вызывается при создании объекта. Этот метод инициализирует атрибуты экземпляров значениями, переданными ему в качестве аргументов.

dog1 = Dog("Buddy", "Golden Retriever")

dog2 = Dog("Max", "Labrador")

В данном случае, мы создали два объекта класса `Dog`: `dog1` и `dog2` с именами «Buddy» и «Max» и породами «Golden Retriever» и «Labrador» соответственно.

3. Использование объектов.

Теперь, когда у нас есть объекты экземпляров класса, мы можем использовать их атрибуты и методы.

print(dog1.name) # выводит "Buddy"

print(dog2.breed) # выводит "Labrador"

dog1.bark() # выводит "Woof Woof!"

dog2.bark() # выводит "Woof Woof!"

Таким образом, при помощи классов мы можем создавать и управлять объектами в объектно-ориентированном программировании. Это позволяет повысить модульность, масштабируемость и управляемость кода.

1. **Атрибуты класса и экземпляра класса**

**Объяснение атрибута класса, его объявление и использование.**

Атрибут класса – это переменная, определенная внутри класса и принадлежащая ему, которая используется для хранения информации или свойства, общего для всех объектов класса. Атрибуты класса объявляются внутри класса и вне его методов. Они обычно используются для хранения постоянных значений, которые не должны меняться при создании объектов (экземпляров) этого класса.

1. Объявление атрибутов класса.

Атрибуты класса определяются внутри класса, но вне методов. Например, создадим класс `Car` с атрибутами `wheels`, представляющим стандартное количество колес.

class Car:

wheels = 4

def \_\_init\_\_(self, make, model):

self.make = make

self.model = model

Здесь `wheels` является атрибутом класса, определенным внутри класса `Car`, а `make` и `model` являются атрибутами экземпляра (объекта), которые инициализируются при создании объекта.

2. Использование атрибутов класса.

Атрибуты класса можно использовать как для доступа к их значениям, так и для их изменения. Атрибуты класса доступны как для класса, так и для его объектов (экземпляров).

# Доступ через класс

print(Car.wheels) # выводит 4

# Создание объектов (экземпляров)

car1 = Car("Toyota", "Corolla")

car2 = Car("Honda", "Civic")

# Доступ через экземпляры

print(car1.wheels) # выводит 4

print(car2.wheels) # выводит 4

# Изменение значения атрибута класса

Car.wheels = 5

print(car1.wheels) # выводит 5

print(car2.wheels) # выводит 5

При изменении значения атрибута класса, он изменяется у всех объектов класса, так как атрибут класса разделяет свое значение между всеми объектами.

Важно отметить, что атрибуты класса следует использовать с осторожностью и только для хранения значений, общих для всех объектов класса. В противном случае лучше использовать атрибуты экземпляра, которые уникальны для каждого объекта.

**Объяснение атрибута экземпляра класса, его объявление и использование.**

Атрибут экземпляра класса – это переменная, определенная внутри класса и принадлежащая каждому отдельному объекту (экземпляру) этого класса. Они используются для хранения информации, которая является уникальной для каждого объекта. Атрибуты экземпляра инициализируются в методе `\_\_init\_\_()` и могут быть использованы или изменены только через объекты класса.

1. Объявление атрибутов экземпляра класса.

Атрибуты экземпляра класса инициализируются внутри метода `\_\_init\_\_()` класса при передаче им аргументов. Например, создадим класс `Person` с атрибутами `name` и `age`.

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

Здесь `name` и `age` являются атрибутами экземпляра класса `Person`. Ключевое слово `self` ссылается на текущий объект и используется для доступа к атрибутам или методам этого объекта.

2. Использование атрибутов экземпляра класса.

Атрибуты экземпляра можно использовать для доступа, изменения их значений и передачи их значениям других функций или методов. Атрибуты экземпляра доступны только через объекты класса.

# Создание объектов (экземпляров)

person1 = Person("Alice", 30)

person2 = Person("Bob", 25)

# Доступ к атрибутам экземпляра

print(person1.name) # выводит "Alice"

print(person2.age) # выводит 25

# Изменение значения атрибута экземпляра

person1.age = 31

print(person1.age) # выводит 31

# Использование атрибута экземпляра в методах класса

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def greet(self):

print(f"Hello, my name is {self.name} and I am {self.age} years old.")

person1 = Person("Alice", 30)

person1.greet() # выводит "Hello, my name is Alice and I am 30 years old."

Таким образом, атрибуты экземпляра класса используются для хранения уникальных значений, связанных с каждым объектом класса, и могут быть использованы в методах класса для реализации различных функций и поведения.

**Разница между атрибутами класса и атрибутами экземпляров.**

Атрибуты класса и атрибуты экземпляров имеют разные характеристики и области применения. Вот основные различия между ними:

1. Определение.

Атрибуты класса определяются внутри класса, но вне его методов. Они обычно используются для хранения значений, которые являются общими для всех объектов класса.

Атрибуты экземпляров инициализируются внутри метода `\_\_init\_\_()` или других методов класса. Они используются для хранения значений, которые являются уникальными для каждого объекта класса.

2. Доступ.

Атрибуты класса доступны через класс и его объекты (экземпляры). Это означает, что и класс, и экземпляры могут получать доступ к атрибутам класса и изменять их.

Атрибуты экземпляров доступны только через объекты (экземпляры) класса. Это означает, что только экземпляры могут получать доступ к атрибутам экземпляра и изменять их.

3. Общая память.

Атрибуты класса разделяют свои значения между всеми объектами класса. Если вы измените значение атрибута класса в одном объекте, это изменение будет отражено во всех других объектах этого класса.

Атрибуты экземпляров хранятся отдельно для каждого объекта. Если вы измените значение атрибута экземпляра в одном объекте, это не повлияет на значения атрибута в других объектах.

Пример:

class MyClass:

class\_attribute = "I am a class attribute"

def \_\_init\_\_(self, instance\_attribute):

self.instance\_attribute = instance\_attribute

Здесь `class\_attribute` является атрибутом класса, а `instance\_attribute` является атрибутом экземпляра.

Сводя всё вместе:

- Атрибуты класса используются для хранения значений, общих для всех объектов класса.

- Атрибуты экземпляров используются для хранения значений, уникальных для каждого объекта класса.

- Атрибуты класса доступны через класс и все его объекты, в то время как атрибуты экземпляров доступны только через объекты класса.

- Атрибуты класса имеют общую память между всеми объектами класса, а атрибуты экземпляров хранятся отдельно для каждого объекта.

1. **Методы класса**

**Объяснение методов класса и их использование.**

Метод класса – это функция, определенная внутри класса, которая принимает класс (а не экземпляр класса) в качестве первого аргумента. Методы класса используются для определения функциональности, которая связана с самим классом, а не с его объектами (экземплярами).

1. Определение метода класса.

Чтобы определить метод класса, необходимо использовать декоратор `@classmethod` перед определением метода. Ключевое слово `cls` является общепринятым именем первого параметра метода класса, который ссылается на сам класс.

class MyClass:

class\_variable = 10

@classmethod

def method\_class(cls):

return cls.class\_variable

def method\_instance(self):

return self.class\_variable

Здесь `method\_class` является методом класса, определенным с использованием декоратора `@classmethod`. В рамках данного примера, `method\_class` возвращает значение атрибута класса `class\_variable`.

2. Использование методов класса.

Методы класса можно вызывать через имя класса или через объекты (экземпляры) класса.

# вызов метода класса через имя класса

result = MyClass.method\_class()

print(result) # выводит 10

# вызов метода класса через объекты

my\_instance = MyClass()

result = my\_instance.method\_class()

print(result) # выводит 10

Обратите внимание, что методы класса могут использовать только атрибуты класса. Они не имеют доступа к атрибутам экземпляра.

**Преимущества методов класса**.

- Методы класса логически связаны со всем классом, а не с его экземплярами.

- Методы класса могут быть унаследованы и переопределены в подклассах, что позволяет изменять их поведение для различных классов.

- Методы класса выполняются на уровне класса, а не на уровне объекта. Это означает, что они не требуют инициализации экземпляра класса, и независимо от состояния класса возвращает одну и ту же сущность.

Методы класса обычно используются, когда вызов функции не зависит от состояния или атрибутов экземпляра класса и выполняет действия, осознанно связанные с самим классом.

**Отличие методов класса от функций в обычном программировании.**

Методы класса и функции в обычном программировании отличаются друг от друга по своим основным характеристикам и области применения. Вот некоторые основные различия между ними:

1. Область определения.

Методы класса определяются внутри класса и связаны с этим классом. Они обычно используются для работы с атрибутами класса или для реализации функциональности, связанной с классом вцелом.

Функции в обычном программировании определяются вне классов и являются самостоятельными. Они не привязаны к определенному классу и могут быть использованы в любом месте программы.

2. Доступ к данным.

Методы класса имеют доступ к данным класса через первый аргумент `cls` (ссылка на сам класс). Методы класса могут использовать и изменять атрибуты класса, а также вызывать другие методы класса.

Функции в обычном программировании не имеют прямого доступа к данным класса или атрибутам экземпляра. Они обычно работают с данными, переданными им в качестве аргументов, и их вывод зависит от переданных значений.

3. Связь с объектами и классами.

Методы класса используются в объектно-ориентированном программировании (ООП), где они связаны с определенными классами. Они могут быть вызваны с использованием экземпляров класса или напрямую через имя класса.

Функции в обычном программировании не зависят от объектной структуры и являются составными частями структурного или процедурного программирования. Они вызываются напрямую из других функций или основного блока программы.

4. Наследование.

Методы класса могут быть унаследованы и переопределены в подклассах, что позволяет изменять их поведение для различных классов и реализовывать полиморфизм в ООП.

Функции в обычном программировании не имеют механизма наследования, так как они не связаны с классами или объектами.

В заключение, методы класса тесно связаны с классами и объектами в объектно-ориентированном программировании и имеют доступ к данным класса, в то время как функции в обычном программировании являются независимыми, не имеют прямого доступа к данным класса и работают только с внешними данными, переданными через аргументы.

1. **Конструктор класса и ключевое слово «self»**

**Объяснение конструктора класса и его использования (метод `\_\_init\_\_`).**

Конструктор класса — это специальный метод, который автоматически вызывается при создании нового объекта (экземпляра) класса. В Python, конструктор класса имеет имя `\_\_init\_\_` и является встроенным методом для инициализации объекта.

Основная задача конструктора — инициализация данных объекта и выполнение необходимых действий для подготовки объекта к использованию. Конструктор обычно принимает аргументы, которые определяют начальные значения атрибутов объекта.

Пример использования конструктора класса:

class Person:

def \_\_init\_\_(self, name, age): # Объявление конструктора класса

self.name = name

self.age = age

В этом примере создается класс `Person`, который содержит конструктор `\_\_init\_\_`. Конструктор принимает два аргумента (параметры): `name` и `age`. При создании объекта класса `Person`, конструктор автоматически вызывается и инициализирует атрибуты `self.name` и `self.age` значениями, переданными при создании объекта.

Пример создания объекта и использования конструктора:

person1 = Person("Alice", 30) # Создание объекта класса Person. Значения "Alice" и 30 передаются конструктору

print(person1.name) # Выводит "Alice"

print(person1.age) # Выводит 30

person2 = Person("Bob", 25) # Создание другого объекта класса Person с другими значениями

print(person2.name) # Выводит "Bob"

print(person2.age) # Выводит 25

В этом примере создаются два объекта класса `Person`, каждый из них инициализируется разными значениями имени и возраста с помощью вызова конструктора `\_\_init\_\_`.

**Роль ключевого слова "self" и его использование при объявлении и вызове методов.**

`self` — это ключевое слово в Python, которое используется в качестве первого параметра в методах класса для представления конкретного экземпляра (объекта) класса. `self` указывает на текущий объект класса и позволяет обращаться к его атрибутам и методам.

Важно заметить, что имя ’self’ не является строго обязательным, можно использовать любое другое имя, но ’self’ является широко принятым соглашением среди разработчиков Python и рекомендуется к использованию.

**Использование при объявлении методов.**

Когда вы определяете метод в классе, первый аргумент должен быть `self`. Это указание для интерпретатора Python на то, что метод принадлежит данному классу и будет вызываться для объектов этого класса.

Пример:

class MyClass:

def my\_method(self, arg1, arg2):

self.some\_attribute = arg1 + arg2

Здесь мы определили класс `MyClass` с методом `my\_method`, который принимает три аргумента: первый — `self`, а другие два — `arg1` и `arg2`.

**Использование при вызове методов**.

Когда вы вызываете метод для объекта класса, вам не нужно явно передавать `self`. Вместо этого, Python автоматически передает ссылку на объект, когда метод вызывается.

Пример:

class MyClass:

def my\_method(self, arg1, arg2):

self.some\_attribute = arg1 + arg2

obj = MyClass()

obj.my\_method(1, 2) # self автоматически передается и ссылается на obj

В этом примере мы создали объект `obj` и вызвали его метод `my\_method(1, 2)`. Несмотря на то что в определении метода есть три параметра (включая `self`), при вызове метода мы передали только два аргумента. Python автоматически передаст ссылку на `obj` в качестве значения для параметра `self`.

**Объяснение того, что «self» ссылается на экземпляр класса, его атрибуты и методы.**

`self` в Python является ссылкой на текущий экземпляр (объект) класса. Это ключевое слово используется в методах класса для доступа к атрибутам и методам экземпляра.

Когда вы определяете метод внутри класса и указываете `self` в качестве первого аргумента, вы говорите Python, что данный метод будет работать с экземпляром класса. Во время вызова метода для объекта, Python автоматически передает ссылку на текущий объект в качестве значения параметра `self`.

**Доступ к атрибутам с использованием self**.

С использованием `self` в методе класса вы можете обращаться к атрибутам экземпляра и изменять их значения.

Пример:

class Circle:

def \_\_init\_\_(self, radius):

self.radius = radius # Инициализация атрибута радиуса с помощью self

def update\_radius(self, new\_radius):

self.radius = new\_radius # Обновление значения радиуса с помощью self

Здесь `self.radius` ссылается на атрибут `radius` экземпляра класса `Circle`.

**Доступ к методам с использованием self**.

Кроме доступа к атрибутам, `self` также используется для вызова других методов экземпляра из текущего метода.

Пример:

class Circle:

def \_\_init\_\_(self, radius):

self.radius = radius

def calculate\_area(self):

return 3.14 \* self.radius \* self.radius

def print\_area(self):

area = self.calculate\_area() # Вызов метода calculate\_area с использованием self

print(f"Area of circle with radius {self.radius} is {area}.")

В этом примере `self.calculate\_area()` вызывает метод `calculate\_area` на текущем объекте экземпляра класса `Circle`.

В целом, `self` позволяет различать атрибуты и методы разных объектов одного класса. Когда вы создаете несколько объектов, каждый из них имеет свою собственную копию атрибутов и свой контекст выполнения. Использование `self` позволяет обращаться именно к атрибутам и методам текущего объекта.

1. **Наследование**

Наследование — один из основных принципов объектно-ориентированного программирования (ООП), который позволяет создавать новые классы на основе уже существующих, наследуя их атрибуты и методы. Это обеспечивает повторное использование кода и модульность, а также упрощает разработку и поддержку программы.

В наследовании присутствуют два типа классов:

1. Родительский (базовый, супер) класс – это существующий класс, на основе которого создаются новые классы.

2. Дочерний (производный, подкласс) класс – это новый класс, который создается на основе родительского и наследует его атрибуты и методы.

**Как объявить класс-наследник**.

Чтобы объявить класс-наследник, вам нужно указать родительский класс в круглых скобках рядом с именем нового класса.

Пример:

class Animal: # Базовый класс (родительский)

def speak(self):

return "Some sound"

class Dog(Animal): # Класс-наследник (дочерний)

def speak(self):

return "Woof!"

В этом примере класс `Dog` наследует класс `Animal`. Это означает, что объекты класса `Dog` будут иметь доступ к атрибутам и методам класса `Animal`.

**Использование базового (родительского) класса**.

Дочерний класс может использовать атрибуты и методы родительского класса, а также переопределить их или добавить новые.

1. Использование атрибутов и методов родительского класса.

Если дочерний класс не переопределяет метод в своем определении, он использует метод родительского класса.

class Animal:

def speak(self):

return "Some sound"

class Dog(Animal):

pass

my\_dog = Dog()

print(my\_dog.speak()) # Выводит "Some sound"

2. Переопределение методов родительского класса.

Дочерний класс может изменить поведение метода, переопределив его в своем определении.

class Animal:

def speak(self):

return "Some sound"

class Dog(Animal):

def speak(self): # Переопределение метода speak

return "Woof!"

my\_dog = Dog()

print(my\_dog.speak()) # Выводит "Woof!"

3. Использование методов родительского класса с помощью `super()`.

Если вы хотите вызвать метод родительского класса внутри дочернего класса, можно использовать встроенную функцию `super()`.

class Animal:

def speak(self):

return "Some sound"

class Dog(Animal):

def speak(self):

parent\_sound = super().speak() # Вызов метода speak() родительского класса

return f"Dog sound: Woof! Parent sound: {parent\_sound}"

my\_dog = Dog()

print(my\_dog.speak()) # Выводит "Dog sound: Woof! Parent sound: Some sound"

В этом примере функция `super().speak()` позволяет вызвать метод `speak()` родительского класса `Animal` и использовать его результат.