

Синтаксис классов

01

Атрибуты и методы класса

Атрибуты и методы являются основными компонентами классов в объектно-ориентированном программировании. Они определяют состояние и поведение объектов класса

Атрибуты

это переменные, которые хранят данные, связанные с объектами класса

Они представляют характеристики или свойства объекта.

Атрибуты могут быть определены внутри класса и доступны во всех экземплярах этого класса

Например, класс "Person" может иметь атрибуты "name", "age" и "gender"

Методы

это функции, которые определяют поведение объектов класса

Они могут использоваться для выполнения операций над данными класса или для взаимодействия с внешним миром. Методы могут изменять состояние объекта или возвращать результаты операций

Например, в классе "Person" может быть метод "say_hello", который выводит приветствие с именем человека

Атрибуты и методы класса могут быть доступны как для чтения, так и для записи, в зависимости от их настроек доступа. Обычно доступ к атрибутам и методам осуществляется через объекты класса. Например, чтобы получить значение атрибута "name" объекта класса "Person", можно использовать выражение "person.name", где "person" - экземпляр класса "Person"

Пример

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

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

    def have_birthday(self):
        self.age += 1

# Создаем экземпляр класса "Person"
person = Person("Alice", 25)

# Получаем значение атрибута "name"
print(person.name) # Output: "Alice"

# Вызываем метод "say_hello"
person.say_hello() # Output: "Hello, my name is Alice and I am 25 years old."

# Вызываем метод "have_birthday"
person.have_birthday()
print(person.age) # Output: 26
```

В этом примере класс "Person" имеет атрибуты "name" и "age", а также методы "say_hello" и "have_birthday". Атрибуты используются для хранения данных объекта, а методы определяют поведение или операции, которые можно выполнить над объектом

02

Геттеры и сеттеры

Геттеры и сеттеры

являются методами класса, которые используются для получения и изменения значений атрибутов класса соответственно

Они позволяют сделать чтение и запись значений атрибутов контролируемыми и обеспечить дополнительное поведение при доступе к этим значениям

Пример

```
...
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name

    def get_age(self):
        return self._age

    def set_age(self, age):
        if age >= 0:
            self._age = age
        else:
            raise ValueError("Age must be a positive number.")
```

В этом примере класс "Person" имеет атрибуты "_name" и "_age". Геттеры `get_name` и `get_age` возвращают значения этих атрибутов соответственно. Сеттеры `set_name` и `set_age` позволяют изменять значения атрибутов

Пример использования геттеров и сеттеров

```
...
person = Person("Alice", 25)
print(person.get_name()) # Output: "Alice"
print(person.get_age()) # Output: 25

person.set_name("Bob")
person.set_age(30)
print(person.get_name()) # Output: "Bob"
print(person.get_age()) # Output: 30
```

Геттеры и сеттеры могут быть полезными, если вам необходимо добавить дополнительные проверки или логику при чтении и записи значений атрибутов класса. Они также позволяют скрыть внутреннюю реализацию атрибутов и предоставить стабильный и контролируемый интерфейс для доступа к данным класса

03

Наследование и полиморфизм

Наследование и полиморфизм - это два важных принципа объектно-ориентированного программирования

Наследование

позволяет создавать иерархию классов, где дочерние классы наследуют свойства и методы от родительского (или базового) класса

Дочерний класс может добавлять свои собственные атрибуты и методы или переопределять родительские для уникальной функциональности. Наследование обеспечивает повторное использование кода и создание более специализированных классов

Пример

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

    def move(self):
        print("Moving...")

class Car(Vehicle):
    def __init__(self, color, brand):
        super().__init__(color)
        self.brand = brand

    def honk(self):
        print("Honking...")

car = Car("red", "Tesla")
car.move()
car.honk()
```

В этом примере класс `Car` наследует атрибут `color` и метод `move` от базового класса `Vehicle`, а также добавляет свой собственный метод `honk`. Это позволяет создавать объекты класса `Car`, которые наследуют и расширяют функциональность базового класса

Полиморфизм

позволяет использовать один и тот же интерфейс для различных типов данных или классов

Это значит, что вы можете использовать объекты разных классов, но вызывать одинаковые методы. Это упрощает работу с группой объектов, которые могут иметь разные внутренние реализации, но обеспечивают одинаковое поведение

Пример

```
...
class Shape:
    def calculate_area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius**2

shapes = [Rectangle(5, 10), Circle(3)]

for shape in shapes:
    print(shape.calculate_area())
```

В этом примере у классов `Rectangle` и `Circle` есть одинаковый метод `calculate_area`, который имеет различную реализацию в каждом классе. Однако благодаря полиморфизму мы можем вызывать этот метод одинаковым образом для объектов разных типов

Наследование и полиморфизм тесно связаны друг с другом и позволяют создавать более гибкий и модульный код в объектно-ориентированном программировании

04

Абстрактный класс

Абстрактный класс

это класс, который не предполагает создание экземпляров, но служит в качестве базового класса для других конкретных классов. Он определяет общий интерфейс и/или поведение для всех своих подклассов

Абстрактные классы обычно содержат абстрактные методы, которые являются методами без реализации. Они обязательно должны быть реализованы в классах-наследниках. Абстрактный класс может также содержать реализации методов, которые могут быть общими для всех наследников. Для создания абстрактного класса в Python можно использовать модуль `abc (Abstract Base Classes)`

Пример

```
...
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius**2

# Пример использования абстрактного класса и его наследников
rectangle = Rectangle(5, 10)
circle = Circle(3)

print(rectangle.calculate_area()) # Output: 50
print(circle.calculate_area()) # Output: 28.26
```

В этом примере `Shape` - абстрактный класс с абстрактным методом `calculate_area()`. Классы `Rectangle` и `Circle` наследуют абстрактный класс `Shape` и обязаны реализовать метод `calculate_area()`. Это гарантирует, что все наследники класса `Shape` имеют метод `calculate_area()`, который можно вызвать у объектов, созданных на основе этих классов

Абстрактные классы полезны для создания общего интерфейса или контракта для связанных классов и гарантированного соблюдения определенной структуры и функциональности в наследниках

Итог

Данный урок был посвящен важным концепциям объектно-ориентированного программирования: атрибуты и методы классов, геттеры и сеттеры, наследование, полиморфизм, абстрактный класс. Эти концепции являются фундаментальными в объектно-ориентированном программировании и помогают создавать гибкую и модульную структуру кода