Практическая работа: Создание классов в Python

Цель:

Научиться создавать классы, определять их атрибуты и методы, а также работать с экземплярами классов.

1. Создание простого класса

Создайте класс Dog, который имеет атрибуты name и age. Добавьте метод bark, который выводит сообщение, содержащее имя собаки.



2. Классы с простыми и специальными методами

Создайте класс Rectangle, который имеет атрибуты width и height. Добавьте методы area (площадь) и perimeter (периметр), а также специальный метод \_\_str\_\_ для вывода информации об объекте.

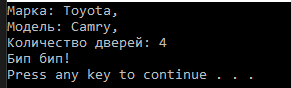
Ожидаемый результат:



3. Наследование классов

Создайте базовый класс Vehicle с атрибутами brand и model. Создайте дочерний класс Car, который добавляет атрибут num\_doors и метод honk.

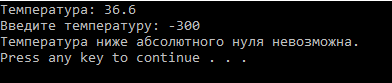
Ожидаемый результат:



4. Инкапсуляция и свойства (properties)

Сделайте атрибуты temperature класса Thermometer приватными и создайте свойства для управления доступом к ним.

Ожидаемый результат:



Теоретическая часть

В языке Python класс определяется с помощью ключевого слова **class**:

class название\_класса:

    атрибуты\_класса

    методы\_класса

Создадим простейший класс:

class Person:

    pass

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. Например:

class Person:

    pass

tom = Person()      # определение объекта tom

bob = Person()      # определение объекта bob

**Конструкторы**

Итак, для создания объекта класса используется конструктор. 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:

class Person:

    # конструктор

    def \_\_init\_\_(self):

        print("Создание объекта Person")

tom = Person()      # Создание объекта Person

Конструктор должен принимать как минимум один параметр ссылку на текущий объект - **self**. Обычно конструкторы применяются для определения действий, которые должны производиться при создании объекта.

### Атрибуты объекта

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

class Person:

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

        self.name = name    # имя человека

        self.age = age        # возраст человека

tom = Person("Tom", 22)

# обращение к атрибутам

# получение значений

print(tom.name)     # Tom

print(tom.age)      # 22

# изменение значения

tom.age = 37

print(tom.age)      # 37

Теперь конструктор класса Person принимает еще два параметра - name и age. Через эти параметры в конструктор будут передаваться имя и возраст создаваемого человека.

Внутри конструктора устанавливаются два атрибута - name и age (условно имя и возраст человека):

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

    self.name = name

    self.age = age

Атрибуту self.name присваивается значение переменной name. Атрибут age получает значение параметра age. Название атрибутов не обязательно должно соответствовать названиям параметров.

Если мы определили в классе конструктор \_\_init\_\_ с параметрами (кроме self), то при вызове конструктора этим параметрам надо передать значения:

tom = Person("Tom", 22)

Далее по имени объекта мы можем обращаться к атрибутам объекта - получать и изменять их значения:

print(tom.name)     # получение значения атрибута name

tom.age = 37        # изменение значения атрибута age

### Методы классов

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

class Person:       # определение класса Person

     def say\_hello(self):

        print("Hello")

tom = Person()

tom.say\_hello()    # Hello

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.

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

class Person:       # определение класса Person

    def say(self, message):     # метод

        print(message)

tom = Person()

tom.say("Hello World!")

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

self.атрибут    # обращение к атрибуту

self.метод()      # обращение к методу

## Инкапсуляция, атрибуты и свойства

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.

class Person:

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

        self.name = name    # устанавливаем имя

        self.age = age      # устанавливаем возраст

    def print\_person(self):

        print(f"Имя: {self.name}\tВозраст: {self.age}")

tom = Person("Tom", 39)

tom.name = "Человек-паук"       # изменяем атрибут name

tom.age = -129                  # изменяем атрибут age

tom.print\_person()              # Имя: Человек-паук     Возраст: -129

Но в данном случае мы можем, к примеру, присвоить возрасту или имени человека некорректное значение, например, указать отрицательный возраст. Подобное поведение нежелательно, поэтому встает вопрос о контроле за доступом к атрибутам объекта.

С данной проблемой тесно связано понятие инкапсуляции. **Инкапсуляция** является фундаментальной концепцией объектно-ориентированного программирования, которая предполагает скрытие функционала и предотвращение прямого доступа извне к нему.

Язык программирования Python позволяет определить приватные или закрытые атрибуты. Для этого имя атрибута должно начинаться с двойного подчеркивания - \_\_name. Например, перепишем предыдущую программу, сделав оба атрибута - name и age приватными:

class Person:

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

        self.\_\_name = name    # устанавливаем имя

        self.\_\_age = age       # устанавливаем возраст

    def print\_person(self):

        print(f"Имя: {self.\_\_name}\tВозраст: {self.\_\_age}")

tom = Person("Tom", 39)

tom.\_\_name = "Человек-паук"     # пытаемся изменить атрибут \_\_name

tom.\_\_age = -129                # пытаемся изменить атрибут \_\_

tom.print\_person()              # Имя: Tom        Возраст: 39

### Методы доступа. Геттеры и сеттеры

Может возникнуть вопрос, как обращаться к подобным приватным атрибутам. Для этого обычно применяются специальные методы доступа. Геттер позволяет получить значение атрибута, а сеттер установить его. Так, изменим выше определенный класс, определив в нем методы доступа:

class Person:

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

        self.\_\_name = name    # устанавливаем имя

        self.\_\_age = age       # устанавливаем возраст

    # сеттер для установки возраста

    def set\_age(self, age):

        if 0 < age < 110:

            self.\_\_age = age

        else:

            print("Недопустимый возраст")

    # геттер для получения возраста

    def get\_age(self):

        return self.\_\_age

    # геттер для получения имени

    def get\_name(self):

        return self.\_\_name

    def print\_person(self):

        print(f"Имя: {self.\_\_name}\tВозраст: {self.\_\_age}")

tom = Person("Tom", 39)

tom.print\_person()  # Имя: Tom  Возраст: 39

tom.set\_age(-3486)  # Недопустимый возраст

tom.set\_age(25)

tom.print\_person()  # Имя: Tom  Возраст: 25

### Аннотации свойств

Выше мы рассмотрели, как создавать методы доступа. Но Python имеет также еще один - более элегантный способ - **свойства**. Этот способ предполагает использование аннотаций, которые предваряются символом @.

Для создания свойства-геттера над свойством ставится аннотация **@property**.

Для создания свойства-сеттера над свойством устанавливается аннотация **имя\_свойства\_геттера.setter**.

Перепишем класс Person с использованием аннотаций:

class Person:

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

        self.\_\_name = name    # устанавливаем имя

        self.\_\_age = age       # устанавливаем возраст

    # свойство-геттер

    @property

    def age(self):

        return self.\_\_age

    # свойство-сеттер

    @age.setter

    def age(self, age):

        if 0 < age < 110:

            self.\_\_age = age

        else:

            print("Недопустимый возраст")

    @property

    def name(self):

        return self.\_\_name

    def print\_person(self):

        print(f"Имя: {self.\_\_name}\tВозраст: {self.\_\_age}")

tom = Person("Tom", 39)

tom.print\_person()  # Имя: Tom  Возраст: 39

tom.age = -3486     # Недопустимый возраст  (Обращение к сеттеру)

print(tom.age)      # 39 (Обращение к геттеру)

tom.age = 25        # (Обращение к сеттеру)

tom.print\_person()  # Имя: Tom  Возраст: 25

Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.

Во-вторых, и сеттер, и геттер называются одинаково - age. И поскольку геттер называется age, то над сеттером устанавливается аннотация @age.setter.

После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение tom.age.

При этом можно определить только геттер, как в случае с свойством name - его нельзя изменить, а можно лишь получить значение.

## Наследование

**Наследование** позволяет создавать новый класс на основе уже существующего класса. Наряду с инкапсуляцией наследование является одним из краеугольных камней объектно-ориентированного программирования.

Ключевыми понятиями наследования являются **подкласс** и **суперкласс**. **Подкласс** наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (base class) или родительским (parent class), а подкласс - производным (derived class) или дочерним (child class).

Синтаксис для наследования классов выглядит следующим образом:

class подкласс (суперкласс):

    методы\_подкласса

Например, у нас есть класс Person, который представляет человека:

class Person:

    def \_\_init\_\_(self, name):

        self.\_\_name = name   # имя человека

    @property

    def name(self):

        return self.\_\_name

    def display\_info(self):

        print(f"Name: {self.\_\_name} ")

Предположим, нам необходим класс работника, который работает на некотором предприятии. Мы могли бы создать с нуля новый класс, к примеру, класс Employee:

class Employee:

    def \_\_init\_\_(self, name):

        self.\_\_name = name  # имя работника

    @property

    def name(self):

        return self.\_\_name

    def display\_info(self):

        print(f"Name: {self.\_\_name} ")

    def work(self):

        print(f"{self.name} works")

Однако класс Employee может иметь те же атрибуты и методы, что и класс Person, так как работник - это человек. Так, в выше в классе Employee только добавляется метод works, весь остальной код повторяет функционал класса Person. Но чтобы не дублировать функционал одного класса в другом, в данном случае лучше применить наследование.

Итак, унаследуем класс Employee от класса Person:

class Person:

    def \_\_init\_\_(self, name):

        self.\_\_name = name   # имя человека

    @property

    def name(self):

        return self.\_\_name

    def display\_info(self):

        print(f"Name: {self.\_\_name} ")

class Employee(Person):

    def work(self):

        print(f"{self.name} works")

tom = Employee("Tom")

print(tom.name)     # Tom

tom.display\_info()  # Name: Tom

tom.work()          # Tom works

## Переопределение функционала базового класса

Что, если мы хотим что-то изменить из этого функционала? Например, добавить работнику через конструктор, новый атрибут, который будет хранить компанию, где он работает или изменить реализацию метода display\_info. Python позволяет переопределить функционал базового класса.

Например, изменим классы следующим образом:

class Person:

    def \_\_init\_\_(self, name):

        self.\_\_name = name   # имя человека

    @property

    def name(self):

        return self.\_\_name

    def display\_info(self):

        print(f"Name: {self.\_\_name}")

class Employee(Person):

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

        super().\_\_init\_\_(name)

        self.company = company

    def display\_info(self):

        super().display\_info()

        print(f"Company: {self.company}")

    def work(self):

        print(f"{self.name} works")

tom = Employee("Tom", "Microsoft")

tom.display\_info()  # Name: Tom

                    # Company: Microsoft

Здесь в классе Employee добавляется новый атрибут - self.company, который хранит компания работника. Соответственно метод \_\_init\_\_() принимает три параметра: второй для установки имени и третий для установки компании. Но если в базом классе определен конструктор с помощью метода \_\_init\_\_, и мы хотим в производном классе изменить логику конструктора, то в конструкторе производного класса мы должны вызвать конструктор базового класса. То есть в конструкторе Employee надо вызвать конструктор класса Person.

Для обращения к базовому классу используется выражение **super()**. Так, в конструкторе Employee выполняется вызов:

super().\_\_init\_\_(name)

Это выражение будет представлять вызов конструктора класса Person, в который передается имя работника. И это логично. Ведь имя работника устанавливается именно в конструкторе класса Person. В самом конструкторе Employee лишь устанавливаем свойство company.