4 Объектно-ориентированное программирование

4.1 Понятие класса

Для определения класса применяется ключевое слово **class**. За ним следует имя класса. Имя класса, в соответствии со стандартом PEP-8, должно начинаться с большой буквы. Далее с новой строки начинается тело класса с отступом в четыре пробельных символа. Пример:

```
class Animal:
  #общие свойства используются для экономии памяти
    name = 'Name'
    age = 0

    def voice(self):#функция
        print('Voice')
```

В представленном примере создаётся класс **Animal** с атрибутами **name**, **age** и методом **voice**.

В приведённом выше примере используется служебное слово **self**, которое, в соответствии с соглашением в Python, определяет ссылку на объект (экземпляр) класса. Переменная **self** связывается с объектом класса, к которому применяются методы класса. Через переменную **self** можно получить доступ к атрибутам объекта. Когда методы класса применяются к новому объекту класса, то **self** связывается с новым объектом. Через эту переменную осуществляется доступ к атрибутам нового объекта.

Конструктором в ООП называется специальный метод, вызываемый при создании экземпляра класса. Это метод **__new__**, но в литературе часто встречается, что под конструктором понимается метод **__init__**. На самом деле **__init__** это инициализатор. Пример

```
class Animal:
    name = 'Name'
    age = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def voice(self):
        print('Voice')
```

Локальные переменные. Понятие области видимости переменных используется и в методологии ООП. Локальная переменная в классе доступна только в рамках части кода, где она определена. Например, если определить переменную в пределах метода, не выйдет получить к ней доступ из других частей программы. Пример:

```
class Auto:
    # методы класса
    def on_start(self):
        info = "Автомобиль заведен"
        return info
```

В представленном примере создаётся локальная переменная **info** в рамках метода **on_start()** класса **Auto**. Проверим работу кода, создав экземпляр класса **Auto,** и попытаемся получить доступ к локальной переменной **info** (рис.4.1).

```
a = Auto()
a.on_start()
a.info
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_480/3779664100.py in <module>
      1 a = Auto()
----> 2 print(a.info)

AttributeError: 'Auto' object has no attribute 'info'
```

Рисунок 4.1 – Результат обращения к локальной переменной

Ошибка возникает из-за отсутствия возможности получения доступа к локальной переменной вне блока, в котором переменная определена.

Глобальные переменные. Глобальные переменные, в отличие от локальных, определяются вне различных блоков кода. Доступ к ним возможен из любых точек программы (класса). Пример:

```
class Auto:
   info_1 = "Автомобиль заведён"

   def on_start(self):
       info_2 = "Автомобиль заведён"
       return info_2

Результат
a = Auto()
print(a.info_1) # Автомобиль заведён
```

В примере создаётся глобальная переменная **info_1,** и на экран выводится её значение. При этом ошибка не возникает.

4.2 Модификаторы доступа

Механизмы использования модификаторов позволяют изменять области видимости переменных. В Python ООП доступны три вида модификаторов:

– Public (публичный).
– Protected (защищённый).
– Private (приватный).

Для переменных с модификатором публичного доступа есть возможность изменения значений за пределами класса. Для публичных переменных префиксы (подчеркивания) не применяются.

Защищённая переменная создаётся с помощью добавления одного знака подчеркивания перед именем переменной. При использовании защищённых переменных их значения могут меняться только в пределах одного и того же пакета.

Приватная переменная идентифицируется с помощью двойного подчёркивания перед именем переменной. Значения приватных переменных могут изменяться только в пределах класса. Пример:

```
class Auto:
    def metod1(self, p1, p2, p3):
        print("Автомобиль заведен")
        self.auto_name = p1 #   Public (публичный).
        self._auto_year = p2 #  Protected (защищённый).
        self.__auto_model = p3  #   Private (приватный).
```

В примере создаётся класс **Auto** с конструктором и тремя переменными: **auto_name**, **auto_model**, **auto_year**. Переменная **auto_name** — публичная, а переменные **auto_year** и **auto_model** — защищённая и приватная соответственно.

Создадим экземпляр класса **Auto** и проверим доступность переменной **auto_name** (рис.4.2).

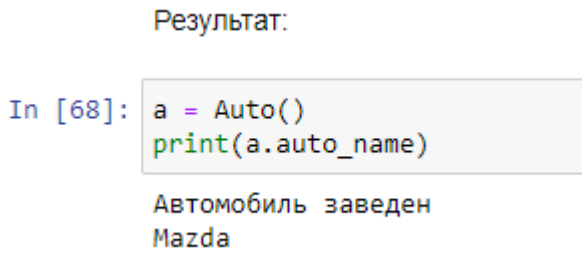

Рисунок 4.2 – Обращение к публичной переменной

Переменная **auto_name** обладает публичным модификатором. Доступ к ней возможен не из класса. Мы это увидели выше.

Теперь попробуем обратиться к значению переменной **auto_model** (рис.4.3).

```
In [69]: print(a.auto_model)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_480/1023684930.py in <module>
----> 1 print(a.auto_model)

AttributeError: 'Auto' object has no attribute 'auto_model'
```

Рисунок 4.3 – Обращение к приватной переменной

После запуска примера мы получили сообщение об ошибке.

4.3 Инкапсуляция

В Python инкапсуляция реализуется только на уровне соглашения, которое определяет общедоступные и внутренние характеристики. Одиночное подчёркивание в начале имени атрибута или метода свидетельствует о том, что атрибут или методы не предназначены для использования вне класса. Они доступны по этому имени. Пример:

```
class MyClass:
    _value = "значение"  #  Protected (защищённый).
    def _method(self):
        print("Это защищенный метод!")
```

Результат показан на рисунке 4.4.

```
mc = MyClass()
mc._method()
print(mc._value)
```

```
Это защищенный метод!
значение
```

Рисунок 4.4 – Вызов защищенного метода

Использование двойного подчёркивания перед именем атрибута и метода делает их недоступными по этому имени. Пример:

```
class MyClass:
    __value = "значение" # Private (приватный).
    def __method(self):
        print("Это приватный метод!")
mc = MyClass()
mc.__method()
print(mc.__value)
```

Вызываем приватный метод. Результат показан на рис.4.5

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-da436c1ba00c> in <cell line: 0>()
      1 mc = MyClass()
----> 2 mc.__method()
      3 print(mc.__value)

AttributeError: 'MyClass' object has no attribute '__method'
```

Рисунок 4.5 – Результат вызова приватного метода

Но и эта мера не обеспечивает абсолютную защиту. Обратиться к атрибуту или методу по-прежнему можно, используя следующий подход: _ИмяКласса__ИмяАтрибута.

Для работы с инкапсулированными объектами предназначены свойства-аксесоры: «геттер» и «сеттер». Последний дает возможность сделать проверки для вводимых значений. С помощью функции property() в Python можно создавать управляемые атрибуты в классах. Метод property() принимает на вход методы get, set и delete, и возвращает объекты класса property. Пример:

```
class Animal:
    name = 'Name'
    __age = 0

    def __init__(self, name, age):
        self.name = name
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, value):
        if value > 0:
            self.__age = value
            print(self.__age)
        else:
            print('Wrong data')

    age = property(fget=get_age, fset=set_age)
```

Результат показан на рисунке 4.6.

```
cat.age
```

```
3
```

```
[ ]  cat = Animal(name = 'Vaska', age = 3)
     cat.get_age()
```

```
3
```

Рисунок 4.6 – Обращение к инкапсулированной переменной

Но вместо метода property() лучше использовать декоратор property. Тогда придерживаются такого правила, что для «геттера» для «сеттера» *названия функций должны быть одинаковыми* (как в примере функция **age**). Пример:

```
class Animal:
    name = 'Name'
    __age = 0

    def __init__(self, name, age):
        self.name = name
        self.__age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, value):
        if value > 0:
            self.__age = value
        else:
            print('Wrong data')

    # age = property(fget=get_age, fset=set_age)
```

При вводе отрицательного возраста кота получаем ошибку (рис. 4.7)

```
cat = Animal(name = 'Vaska', age = 3)
cat.age
```

```
3
```

```
cat.age = -9
```

```
Wrong data
```

Рисунок 4.7 – Применение декоратора property

Под декоратором в Python подразумевается функция (или класс), расширяющая логику работы другой функции. У разработчика существует возможность написания собственных декораторов или использования существующих. Рассмотрим декоратор **@property**. Символ **@** позволяет идентифицировать объект как декоратор и установить его для некоторой функции (или метода класса).

Встроенный декоратор **@property** позволяет работать с методом некоторого класса как с атрибутом.

4.4 Наследование

Простое наследование. Сущность этого понятия соответствует его названию. Речь идёт о наследовании некоторым объектом характеристик другого объекта-родителя. Пример:

```
# Простое наследование
# Класс Transport
class Transport:
    count = 0
    def transport_method(self):
        print("Это родительский метод из класса Transport")

# Класс Auto, наследующий Transport
class Auto(Transport):
    def read_pdf(self):
        print("Это метод из дочернего класса")
```

В представленном примере создаются два класса: **Transport** (родитель), **Auto** (наследник). Для реализации наследования нужно указать имя класса-родителя внутри скобок, следующих за именем класса-наследника. В классе **Transport** реализован метод **transport_method()**, а в дочернем классе есть метод **auto_method()**. Класс **Auto** наследует характеристики класса **Transport,** то есть все его атрибуты и методы.

Множественное наследование. Механизм наследования может быть реализован с использованием нескольких родителей у одного класса. И наоборот, один класс-родитель будет передавать свои характеристики нескольким дочерним классам.

```
# Несколько дочерних классов у одного родителя
class Parent:
    def greet(self):
        return "Hello from Parent"

class Child1(Parent):
    def greet(self):
        return "Hello from Child1"
```

```
class Child2(Parent):
    def greet(self):
        return "Hello from Child2"

# Пример использования
child1_instance = Child1()
child2_instance = Child2()

print(child1_instance.greet())  # Вывод: Hello from Child1
print(child2_instance.greet())  # Вывод: Hello from Child2
```

Рассмотрим ещё один пример, в котором класс-родитель **Parent** определяет атрибуты. Эти атрибуты могут быть характерны для всех классов-наследников. Здесь в конструкторах классов-наследников инициализируются параметры. Часть их - собственные атрибуты классов-наследников, а некоторые наследуются от родителей. Чтобы работать с унаследованными атрибутами, нужно их перечислить, например, **super().__init__(name, age)**. Тем самым мы показываем, что хотим иметь возможность работы с атрибутами класса-родителя. Если атрибуты не перечислить, то при попытке обращения к ним через экземпляр класса-наследника возникнет ошибка. Пример:

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

    def greet(self):
        return f"Hello, {self.name}!"

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Вызов конструктора
родительского класса
        self.age = age

    def greet(self):
        parent_greeting = super().greet()  # Вызов метода
родительского класса
        return f"{parent_greeting} You are {self.age} years old."

# Пример использования
child_instance = Child("Alice", 10)
print(child_instance.greet())  # Вывод: Hello, Alice! You are 10
years old.
```

В Python возможно множественное наследование, когда несколько родителей у одного класса. Пример:

```
class Parent1:
    def method1(self):
        return "Method from Parent1"

class Parent2:
    def method2(self):
        return "Method from Parent2"

class Child(Parent1, Parent2):
    def child_method(self):
        return "Method from Child"

# Пример использования
child_instance = Child()
print(child_instance.method1())  # Вывод: Method from Parent1
print(child_instance.method2())  # Вывод: Method from Parent2
print(child_instance.child_method())  # Вывод: Method from Child
```

Возможна ситуация, когда у классов-родителей совпадают имена атрибутов и методов. В этом случае обращение к такому атрибуту или методу через «наследник» будет адресовано к атрибуту или методу того класса-родителя, который значится первым.

```
class Shape:
    def __init__(self, param_1, param_2):
        self.param_1 = param_1
        self.param_2 = param_2

    def get_params(self):
        return f"Параметры Shape: {self.param_1}, {self.param_2}"

class Material:
    def __init__(self, param_1, param_2):
        self.param_1 = param_1
        self.param_2 = param_2

    def get_params(self):
        return f"Параметры Material: {self.param_2},
{self.param_1}"

class Triangle(Shape, Material):
    def __init__(self, param_1, param_2):
        super().__init__(param_1, param_2)

#Пример использования
t = Triangle(10, 20)
print(t.get_params())
# Параметры Shape: 10, 20
```

4.5 Полиморфизм

Дословный перевод этого понятия — «имеющий многие формы». В методологии ООП — это способность объекта иметь различную функциональность. В программировании полиморфизм проявляется в перегрузке или переопределении методов классов.

Перегрузка методов. Реализуется в возможности метода отражать разную логику выполнения в зависимости от количества и типа передаваемых параметров.

```
class Engine:
    def start_engine(self, sound):
        print(f"Engine starts with sound: {sound}")

class Wheels:
    def rotate_wheels(self, speed):
        print(f"Wheels are rotating at {speed} km/h")

class Auto(Engine, Wheels):
    def auto_start(self, param_1, param_2=None):
        if param_2 is not None:
            print(param_1 + param_2)
        else:
            print(param_1)

# Пример использования
my_car = Auto()
my_car.start_engine("Vroom!")  # Запускаем двигатель
my_car.rotate_wheels(60)        # Крутятся колеса
my_car.auto_start("Starting car", " with ignition")  # Вывод:
Starting car with ignition
my_car.auto_start("Starting car")  # Вывод: Starting car
```

Классы Engine и Wheels: Каждый из них имеет свой метод. Класс Auto: Наследует от Engine и Wheels. Метод auto_start позволяет запускать автомобиль с одним или двумя параметрами. При создании экземпляра Auto можно вызывать методы как от Engine, так и от Wheels, а также использовать метод auto_start. значение переданного параметра, во втором — сумма параметров.

4.6 Переопределение методов

Переопределение методов в полиморфизме выражается в наличии метода с одинаковым названием для родительского и дочернего классов. При этом логика методов различается, но названия идентичны.

```
class Animal:
    def sound(self):
        return "Some generic animal sound"
```

```
class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Пример использования
def make_sound(animal):
    print(animal.sound())

dog = Dog()
cat = Cat()

make_sound(dog)  # Вывод: Bark
make_sound(cat)  # Вывод: Meow
```

Родительский класс Animal: Содержит метод sound, который возвращает общий звук. Дочерние классы Dog и Cat: Переопределяют метод sound, предоставляя свои уникальные звуки. Функция make_sound: Принимает объект Animal и вызывает метод sound, демонстрируя полиморфизм. При вызове make_sound с экземплярами Dog и Cat выводятся соответствующие звуки, даже несмотря на то, что метод называется одинаково.

4.7 Перегрузка операторов

Под перегрузкой операторов понимается изменение логики работы различных операторов языка с использованием специальных методов. Эти методы идентифицируются двойным подчеркиванием до и после имени метода.

Под операторами имеются в виду знаки **+**, **-**, *****, /, отвечающие за выполнение привычных математических операций, а также особенности синтаксиса языка, обеспечивающие создание объекта, вызова его как функции, получение доступа к элементу объекта по индексу и т. д. К перегружаемым операторам также относятся >, <, **≤**, **≥**, ==, **!**=, +=, -=. При перегрузке каждого из этих операторов происходит вызов соответствующего метода, представленного в таблице 4.1.

Таблица 4.1 – Методы перезагрузки операторов

Методы	Описание
__init__()	Соответствует конструктору объектов класса, срабатывает при создании объектов
__del__()	Соответствует деструктору объектов класса, срабатывает при удалении объектов

__str__()	Срабатывает при передаче объекта функциям **str()** и **print()**, преобразует объект к строке
__add__()	Срабатывает при участии объекта в операции сложения в качестве операнда с левой стороны, обеспечивает перегрузку оператора сложения
__setattr__()	Срабатывает при выполнении операции присваивания значения атрибуту объекта
__getitem__()	Срабатывает при извлечении элемента по индексу
__call__()	Срабатывает при обращении к экземпляру класса как к функции
__gt__()	Соответствует оператору>
__lt__()	Соответствует оператору <
__ge__()	Соответствует оператору ≥
__le__()	Соответствует оператору ≤
__eq__()	Соответствует оператору ==
__iadd__()	Соответствует операции «Сложение и присваивание» +=
__isub__()	Соответствует операции «Вычитание и присваивание» -=

Перегрузка операторов относится к редко используемым на практике механизмам. На деле разработчику чаще всего приходится сталкиваться с перегрузкой в конструкторе. Но в рамках концепции ООП эта тема важна. В списке выше приведена только часть методов, используемых при реализации перегрузки операторов в Python.

Благодаря механизму перегрузки операторов пользовательские классы встают в один ряд со встроенными, поскольку все встроенные типы в Python относятся к классам. В итоге все объекты класса получают одинаковый интерфейс.

4.8 Интерфейсы

Под интерфейсом в ООП понимается описание поведения объекта, т. е., совокупность публичных методов объекта, которые могут применяться в других частях программы для взаимодействия с ним.

Рассмотрим подробнее понятие интерфейса в привязке к абстрактным классам, которые реализуются в Python с помощью встроенного в стандартную библиотеку модуля abc (Abstract Base Classes). Абстрактные классы позволяют контролировать поведение классов-наследников, например, проверять, что они обладают одинаковым интерфейсом.

```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def my_method_1(self):
        pass
```

```
    @abstractmethod
    def my_method_2(self):
        pass

class MyClass(MyAbstractClass):
    pass
```

В этом примере создается абстрактный класс **MyAbstractClass** и в случае наследования от него во всех классах-потомках необходимо реализовать два базовых метода, т. е., все классы-потомки наследуют интерфейс родителя. Соответственно, логику класса **MyClass** в примере выше необходимо изменить:

```
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def my_method_1(self):
        print("Метод my_method_0()")
        pass
    @abstractmethod
    def my_method_2(self):
        print("Метод my_method_00()")
        pass

class MyClass(MyAbstractClass):
    def my_method_1(self):
        print("Метод my_method_1()")

    def my_method_2(self):
        print("Метод my_method_2()")
```