GeekBrains

Основы Python

Урок 6. ООП. Введение

Пришло время познакомиться с важнейшей парадигмой программирования — объектно-ориентированным программированием. Оно играет важную роль в Python и позволяет формировать структуру программы из обособленных компонентов. Важные понятия парадигмы — класс, конструктор, атрибут, метод, экземпляр класса. В рамках урока разбираются важнейшие свойства ООП: инкапсуляция, наследование и полиморфизм. Приведено описание механизмов перегрузки и переопределения методов.

Оглавление

На этом уроке студент:

1. Познакомится с преимуществами и недостатками механизма ООП.
2. Узнает, как создаются классы, что такое объекты, атрибуты и методы.
3. Познакомится с локальными и глобальными переменными, модификаторами доступа.
4. Разберёт особенности применения инкапсуляции, наследования и полиморфизма в Python.
5. Научится реализовывать множественное наследование, перегрузку и переопределение методов.

Достоинства и недостатки механизма ООП

Достоинства:

1. Возможность повторного использования кода. Классы — это шаблоны, описывающие различные объекты (их свойства) и операции, выполняемые со свойствами (атрибутами) этих объектов. Эти шаблоны можно использовать повторно, в других файлах-модулях.
2. Повышение читаемости и гибкости кода. Классы и их код можно хранить в отдельных файлах-модулях и импортировать в другие модули. Модульный принцип организации программ ускоряет изучение кода программы и его модернизации.
3. Ускорение поиска ошибок и их исправления. Опять же, модульность программы предусматривает её разбиение на блоки-классы для решения определённой задачи. Соответственно, для поиска ошибок не нужно просматривать весь код. Необходимо искать ошибку в конкретном классе.
4. Повышение безопасности проекта. Благодаря такому важному свойству ООП, как инкапсуляция, разрабатываемая программа получает дополнительный уровень безопасности.

Недостатки:

1. Для реализации взаимосвязи классов необходимо хорошо разбираться в особенностях предметной области, а также чётко представлять структуру создаваемого приложения.
2. Сложность в разбиении проекта на классы. Новичкам может быть тяжело определить для проекта классы-шаблоны.
3. Сложность в модификации проекта. С добавлением в проект новой функциональности придётся вносить всё больше изменений в структуру классов.

Классы, объекты, атрибуты

Понятие класса

Класс в ООП — чертёж объекта. Если проводить аналогию с объектами реального мира, то, например, автомобиль — это объект, а чертёж, описывающий структуру автомобиля, его параметры и функции, — класс. Таким образом, понятие «Машина» будет соответствовать классу. Объектами этого класса будут марки автомобилей с различными характеристиками (атрибутами) и функциональными возможностями (методами), например, Audi, Lexus, Mercedes.

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 Auto:
    # атрибуты класса
    auto_name = "Lexus"
    auto_model = "RX 350L"
    auto_year = 2019

    # методы класса
    def on_auto_start(self):
        print(f"Заводим автомобиль")

    def on_auto_stop(self):
        print("Останавливаем работу двигателя")
```

В представленном примере создаётся класс **Auto** с атрибутами **auto_name**, **auto_model**, **auto_year** и методами **on_auto_start()** и **on_auto_stop()**.

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.

Понятие объекта

Ранее мы разобрались, что класс — чертёж, на основе которого создаётся некоторый объект. Для создания объекта (экземпляра класса) необходимо в отдельной строке указать имя класса с открывающей и закрывающей круглыми скобками. Эту инструкцию можно связать с некоторой переменной, которая будет содержать ссылку на созданный объект.

Создадим экземпляр для класса, описанного выше.

Пример:

```
a = Auto()
print(a)
print(type(a))
print(a.auto_name)
a.on_auto_start()
a.on_auto_stop()
```

Результат:

```
<__main__.Auto object at 0x0000001381FD8B38>
<class '__main__.Auto'>
Lexus
Заводим автомобиль
Останавливаем работу двигателя
```

В первой строке примера создаётся экземпляр класса **Auto**, ссылка на который связывается с переменной **a**. Содержимое этой переменной выводится во второй строке. В третьей строке проверяется тип переменной **a** — это класс **Auto**. В четвёртой строке осуществляется получение доступа к одному из атрибутов класса, а в пятой и шестой — запуск методов класса.

Понятие атрибута

Согласно методологии ООП, выделяют атрибуты классов и экземпляров. Атрибуты класса доступны из всех экземпляров класса. Атрибуты экземпляров относятся только к объектам класса. Атрибуты класса объявляются вне любого метода, а атрибуты экземпляра — внутри любого метода. Разберёмся на примере:

Пример:

```
class Auto:

    # атрибуты класса
    auto_count = 0

    # методы класса
    def on_auto_start(self, auto_name, auto_model, auto_year):
        print("Автомобиль заведен")
        self.auto_name = auto_name
        self.auto_model = auto_model
        self.auto_year = auto_year
        Auto.auto_count += 1
```

В приведённом примере создаётся класс **Auto**, содержащий один атрибут класса **auto_count** и три атрибута экземпляра класса: **auto_name**, **auto_model** и **auto_year**. В классе реализован один метод **on_auto_start()** с указанными атрибутами экземпляра. Их значения передаются в виде параметров методу **on_auto_start()**. Внутри этого метода значение атрибута **auto_count** класса увеличивается на единицу.

Важно отметить, что внутри методов атрибуты экземпляра идентифицируются ключевым словом **self** перед именем атрибута. При этом атрибуты класса идентифицируются названием класса перед именем атрибута.

Пример:

```
a = Auto()
a.on_auto_start("Lexus", "RX 350L", 2019)
print(a.auto_name)
print(a.auto_count)
```

Результат:

```
Автомобиль заведён
Lexus
1
```

В этом примере выводятся значения атрибута экземпляра класса (**auto_name**) и атрибута класса (**auto_count**).

Теперь, если создать ещё один экземпляр класса **Auto** и вызвать метод **on_auto_start()**, результат будет следующим:

```
a_2 = Auto()
a_2.on_auto_start("Mazda", "CX 9", 2018)
print(a_2.auto_name)
print(a_2.auto_count)
```

Результат:

```
Автомобиль заведен
Mazda
2
```

Теперь значение атрибута **auto_count** равняется двум, из-за того, что он — атрибут класса и распространяется на все экземпляры. Значение атрибута **auto_count** в экземпляре **a** увеличилось до 1, а его значение в экземпляре **a_2** достигло двух.

Конструкторы, методы

Понятие конструктора

Конструктором в ООП называется специальный метод, вызываемый при создании экземпляра класса. Этот метод определяется с помощью конструкции **__init__**.

Пример:

```
class Auto:
    # атрибуты класса
    auto_count = 0

    # методы класса
    def __init__(self):
        Auto.auto_count += 1
        print(Auto.auto_count)
```

В примере создаётся класс **Auto** с одним атрибутом **auto_count** уровня класса. В классе реализован конструктор, увеличивающий значение **auto_count** на единицу и выводящий на экран итоговое значение.

Теперь при создании экземпляра класса **Auto** вызывается конструктор, значение **auto_count** увеличивается и отображается на экране. Создадим несколько экземпляров класса:

Пример:

```
a_1 = Auto()
a_2 = Auto()
a_3 = Auto()
```

Результат:

```
1
2
3
```

В результат запуска выводятся значения 1, 2, 3, так как для каждого экземпляра значение атрибута **auto_count** возрастает и выводится на экран. На практике конструкторы используются для инициализации значений атрибутов. Это важно при создании объекта класса.

Понятие метода

Ранее мы уже познакомились с методами в ООП, то есть, функциями, получающими в качестве обязательного параметра ссылку на объект и выполняющими определённые действия с атрибутами объекта. Мы уже создали методы **on_auto_start()** и **on_auto_stop()** для класса **Auto**. Вспомним ещё раз, как создаётся метод.

Пример:

```
class Auto:

    def get_class_info(self):
        print("Детальная информация о классе")

a = Auto()
a.get_class_info()
```

Результат:

```
Детальная информация о классе
```

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:
    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**.

Пример:

```
a = Auto()
```

```
print(a.info)
```

Результат:

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

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

a = Auto()
print(a.info_1)
```

Результат:

```
Автомобиль заведён
```

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

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 __init__(self):
        print("Автомобиль заведен")
        self.auto_name = "Mazda"
        self._auto_year = 2019
        self.__auto_model = "CX9"
```

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**.

Пример:

```
a = Auto()
print(a.auto_name)
```

Результат:

```
Mazda
```

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**.

Пример:

```
print(a.auto_model)
```

Результат:

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

После запуска примера мы получили сообщение об ошибке.

Инкапсуляция

Пришло время познакомиться с ключевыми принципами ООП: инкапсуляцией, наследованием и полиморфизмом.

Начнём с инкапсуляции, то есть с механизма сокрытия данных. 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:
    _attr = "значение"
    def _method(self):
        print("Это защищенный метод!")

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

Результат:

```
Это защищённый метод!
значение
```

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

Пример:

```
class MyClass:
    __attr = "значение"
    def __method(self):
        print("Это защищенный метод!")

mc = MyClass()
mc.__method()
print(mc.__attr)
```

Результат:

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

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

Пример:

```
class MyClass:
    __attr = "значение"
    def __method(self):
        print("Это защищённый метод!")

mc = MyClass()
mc._MyClass__method()
print(mc._MyClass__attr)
```

Результат:

```
Это защищенный метод!
значение
```

Наследование

Сущность этого понятия соответствует его названию. Речь идёт о наследовании некоторым объектом характеристик другого объекта-родителя. Объект называется дочерним и обладает не только характеристиками родителя, но и собственными свойствами. Благодаря наследованию можно избежать дублирования кода.

Суть принципа наследования заключается в том, что класс может перенимать (наследовать) параметры другого класса. Класс, наследующий характеристики другого класса, называется дочерним, а класс, предоставляющий свои характеристики, — родительским.

Пример:

```
# Класс Transport
class Transport:
    def transport_method(self):
        print("Это родительский метод из класса Transport")


# Класс Auto, наследующий Transport
class Auto(Transport):
    def auto_method(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.

Проверим работу механизма наследования:

```
a = Auto()
a.transport_method()  # Вызываем метод родительского класса
```

Результат:

```
Это родительский метод из класса Transport
```

В примере создаётся экземпляр класса **Auto**. Для экземпляра класса вызывается метод **transport_method()**. Важно, что в классе **Auto** отсутствует метод с названием **transport_method()**. Так как класс **Auto** унаследовал характеристики класса **Transport**, то экземпляр класса **Auto** работает с методом **transport_method()** класса **Transport**.

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

Пример:

```
# класс Transport
class Transport:
    def transport_method(self):
        print("Родительский метод класса Transport")


# класс Auto, наследующий Transport
class Auto(Transport):
    def auto_method(self):
        print("Дочерний метод класса Auto")


# класс Bus, наследующий Transport
class Bus(Transport):
    def bus_method(self):
        print("Дочерний метод класса Bus")
```

В этом примере у нас есть класс-родитель **Transport**, наследуемый дочерними классами **Auto** и **Bus**. В обоих дочерних классах возможен доступ к методу **transport_method()** класса-родителя. Для запуска скрипта создадим экземпляры класса.

Пример:

```
a = Auto()
a.transport_method()
b = Bus()
b.transport_method()
```

Результат:

```
Родительский метод класса Transport
Родительский метод класса Transport
```

Рассмотрим ещё один пример, в котором класс-родитель **Shape** определяет атрибуты. Эти атрибуты могут быть характерны для всех классов-наследников. Например, цвет фигуры, ширина и высота, основание и высота.

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__(color, param_1, param_2)**. 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 Shape:
    def __init__(self, color, param_1, param_2):
        self.color = color
        self.param_1 = param_1
        self.param_2 = param_2

    def square(self):
        return self.param_1 * self.param_2


class Rectangle(Shape):
    def __init__(self, color, param_1, param_2, rectangle_p):
        super().__init__(color, param_1, param_2)
        self.rectangle_p = rectangle_p

    def get_r_p(self):
        return self.rectangle_p


class Triangle(Shape):
    def __init__(self, color, param_1, param_2, triangle_p):
        super().__init__(color, param_1, param_2)
        self.triangle_p = triangle_p

    def get_t_p(self):
        return self.triangle_p

r = Rectangle("white", 10, 20, True)
print(r.color)
print(r.square())
print(r.get_r_p())
t = Triangle("red", 30, 40, False)
print(t.color)
print(t.square())
print(t.get_t_p())
```

Результат:

```
white
200
True
red
1200
False
```

Несколько родителей у одного класса

Пример:

```
class Player:
    def player_method(self):
        print("Родительский метод класса Player")


class Navigator:
    def navigator_method(self):
        print("Родительский метод класса Navigator")


class MobilePhone(Player, Navigator):
    def mobile_phone_method(self):
        print("Дочерний метод класса MobilePhone")
```

В этом примере создаются классы: **Player**, **Navigator**, **MobilePhone**. Причём классы **Player** и **Navigator** — родительские для класса **MobilePhone**. Поэтому класс **MobilePhone** имеет доступ к методам классов **Player** и **Navigator**. Проверим это.

Пример:

```
m_p = MobilePhone()
m_p.player_method()
m_p.navigator_method()
```

Результат:

```
Родительский метод класса Player
Родительский метод класса Navigator
```

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.

Пример:

```
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_1}, {self.param_2}"


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

t = Triangle(10, 20)
print(t.get_params())
```

Результат:

```
Параметры Shape: 10, 20
```

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.

Пример:

```
# класс Auto
class Auto:
    def auto_start(self, param_1, param_2=None):
        if param_2 is not None:
            print(param_1 + param_2)
        else:
            print(param_1)
```

В этом примере возможны несколько вариантов логики метода **auto_start()**. Первый вариант — при передаче в метод одного параметра. Второй — при передаче двух параметров. В первом случае будет выведено значение переданного параметра, во втором — сумма параметров.

Пример:

```
a = Auto()
a.auto_start(50)
a = Auto()
a.auto_start(10, 20)
```

Результат:

```
50
30
```

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.

Пример:

```
# класс Transport
class Transport:
    def show_info(self):
        print("Родительский метод класса Transport")


# класс Auto, наследующий Transport
class Auto(Transport):
    def show_info(self):
        print("Родительский метод класса Auto")
```

```
# класс Bus, наследующий Transport
class Bus(Transport):
    def show_info(self):
        print("Родительский метод класса Bus")
```

В примере классы **Auto** и **Bus** наследуют характеристики класса **Transport**. В этом классе реализуется метод **show_info()**, переопределенный классом-потомком. Теперь, если вызвать метод **show_info()**, результат будет зависеть от объекта, через который осуществляется вызов метода.

Пример:

```
t = Transport()
t.show_info()

a = Auto()
a.show_info()

b = Bus()
b.show_info()
```

Результат:

```
Родительский метод класса Transport
Родительский метод класса Auto
Родительский метод класса Bus
```

В этом примере методы **show_info()** вызываются с помощью производных классов одного общего базового класса. Но дочерние классы переопределяются через метод класса-родителя, методы обладают разной функциональностью.

Практическое задание

1. Создать класс **TrafficLight** (светофор).

    - определить у него один атрибут **color** (цвет) и метод **running** (запуск);
    - атрибут реализовать как приватный;
    - в рамках метода реализовать переключение светофора в режимы: красный, жёлтый, зелёный;
    - продолжительность первого состояния (красный) составляет 7 секунд, второго (жёлтый) — 2 секунды, третьего (зелёный) — на ваше усмотрение;

- переключение между режимами должно осуществляться только в указанном порядке (красный, жёлтый, зелёный);
- проверить работу примера, создав экземпляр и вызвав описанный метод.

Задачу можно усложнить, реализовав проверку порядка режимов. При его нарушении выводить соответствующее сообщение и завершать скрипт.

2. Реализовать класс **Road** (дорога).

- определить атрибуты: **length** (длина), **width** (ширина);
- значения атрибутов должны передаваться при создании экземпляра класса;
- атрибуты сделать защищёнными;
- определить метод расчёта массы асфальта, необходимого для покрытия всей дороги;
- использовать формулу: длина*ширина*масса асфальта для покрытия одного кв. метра дороги асфальтом, толщиной в 1 см*число см толщины полотна;
- проверить работу метода.

Например: 20 м*5000 м*25 кг*5 см = 12500 т.

3. Реализовать базовый класс **Worker** (работник).

- определить атрибуты: **name**, **surname**, **position** (должность), **income** (доход);
- последний атрибут должен быть защищённым и ссылаться на словарь, содержащий элементы: оклад и премия, например, **{"wage": wage, "bonus": bonus};**
- создать класс **Position** (должность) на базе класса **Worker**;
- в классе **Position** реализовать методы получения полного имени сотрудника (**get_full_name**) и дохода с учётом премии (**get_total_income**);
- проверить работу примера на реальных данных: создать экземпляры класса **Position**, передать данные, проверить значения атрибутов, вызвать методы экземпляров.

4. Реализуйте базовый класс **Car**.

- у класса должны быть следующие атрибуты: **speed**, **color**, **name**, **is_police** (булево). А также методы: **go**, **stop**, **turn(direction)**, которые должны сообщать, что машина поехала, остановилась, повернула (куда);
- опишите несколько дочерних классов: **TownCar**, **SportCar**, **WorkCar**, **PoliceCar**;
- добавьте в базовый класс метод **show_speed**, который должен показывать текущую скорость автомобиля;

- для классов **TownCar** и **WorkCar** переопределите метод **show_speed**. При значении скорости свыше 60 (**TownCar**) и 40 (**WorkCar**) должно выводиться сообщение о превышении скорости.

Создайте экземпляры классов, передайте значения атрибутов. Выполните доступ к атрибутам, выведите результат. Вызовите методы и покажите результат.

5. Реализовать класс **Stationery** (канцелярская принадлежность).

   - определить в нём атрибут **title** (название) и метод **draw** (отрисовка). Метод выводит сообщение «Запуск отрисовки»;
   - создать три дочерних класса **Pen** (ручка), **Pencil** (карандаш), **Handle** (маркер);
   - в каждом классе реализовать переопределение метода **draw**. Для каждого класса метод должен выводить уникальное сообщение;
   - создать экземпляры классов и проверить, что выведет описанный метод для каждого экземпляра.

Дополнительные материалы

1. Объектно-ориентированное Программирование в Python.
2. Объектно-ориентированное программирование. Классы и объекты.
3. Обучение ООП.
4. Python — объектно-ориентированное программирование (ООП).

Используемая литература

1. Язык программирования Python 3 для начинающих и чайников.
2. Программирование в Python.
3. Учим Python качественно (habr).
4. Самоучитель по Python.
5. Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).