Объектно-ориентированное программирование

Что такое ООП?

Раньше программисты просто писали код — лишь бы компьютер понял. Со временем они осознали: чтобы работать с кодом, его необходимо как-то организовать. Код — не просто последовательность команд. Его можно и нужно объединять в логические блоки.

И разработчики придумали, как это сделать. Так появились парадигмы программирования — совокупности правил, которым вы следуете, когда пишете код.

Самая популярная парадигма — **объектно-ориентированное программирование (ООП)**.

ООП — как раз совокупность правил и принципов, которые используют программисты, когда пишут программы.

Каким должен быть код по принципам ООП:

1. Используется множество объектов — они взаимодействуют между собой и образуют единую структуру, которой и является любая программа. Объекты хранят данные и взаимодействуют между собой: вызывают методы друг друга и передают в них информацию.
2. Соблюдаются принципы — инкапсуляция, наследование и полиморфизм. Об этих принципах расскажу позже.

Объекты в ООП

Всё вокруг можно рассматривать как объекты. Компьютер, за которым вы сейчас работаете, — объект. У него есть свойства: цвет, размер, яркость экрана. И методы: включить, выключить, нажать клавишу. Свойства и методы объекта часто связаны. Так, нажатие кнопки увеличения громкости влияет на свойство «громкость».

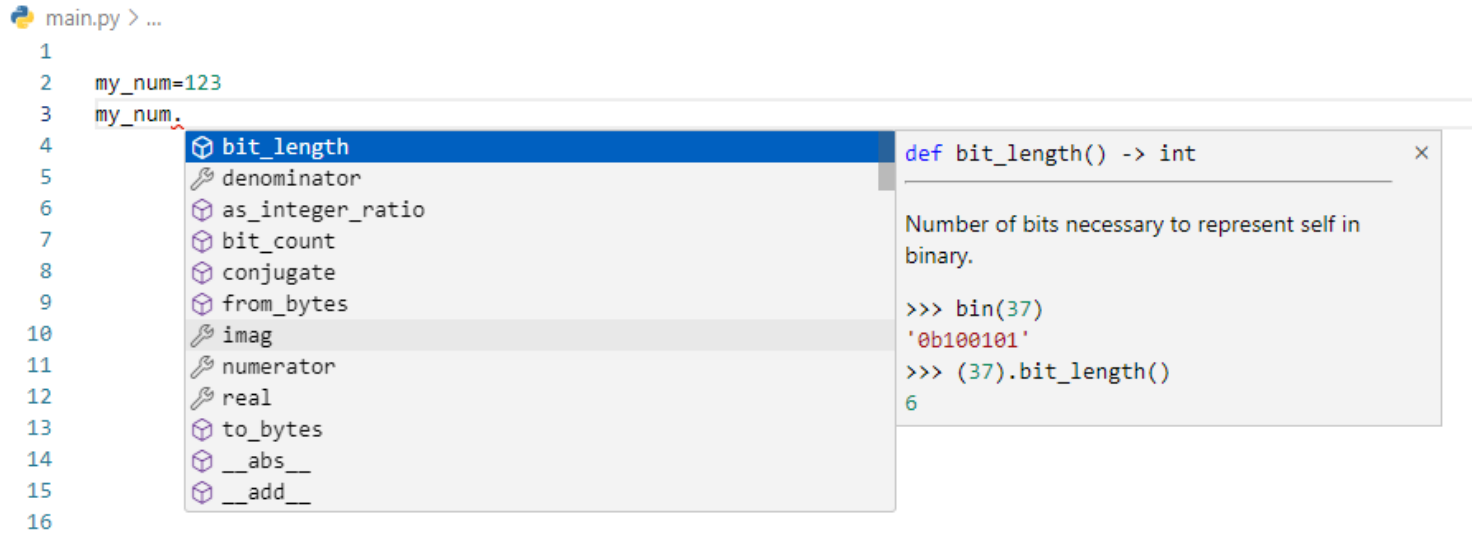
Таким образом, свойства объекта — это данные, которые его характеризуют. Методы объекта — это его функциональность.

Основная идея ООП — данные и функциональность объединены в объекты.

Объекты в Python

При объявлении переменных в Python мы на самом деле создаём объект. Причём тип данных определяет набор свойств и методов (структуру) созданного объекта.

Например, если объявить числовую переменную `my_num`, мы на самом деле создадим объект с большим количеством различных свойств (“`denominator`”, “`imag`”, “`numerator`”…) и методов (“`bit_length`”, “`__abs__`”, “`from_bytes`”…). Все эти свойства и методы Python сам определил для нашего объекта, поскольку автоматически установил тип данных «int».



Мы не можем переопределить, добавить или удалить свойства или методы полученного таким образом объекта. В данном случае, говорят, что у этого объекта **строгая структура**.

Практически все объекты, создаваемые от базовых типов данных Python, обладают строгой структурой. Но есть исключения. Например, функции. Python позволяет определять для функций новые свойства и методы (удалять и переопределять уже имеющиеся нельзя).

```
def say_hello():
  print('Привет!')

say_hello.description='Функция, которая здоровается' # добавили свойство "description"

def function_introduce():
  print('Я функция')

say_hello.introduce=function_introduce # добавили метод "introduce"

print(say_hello.description) # будет выведено
                            # 'Функция, которая здоровается'

say_hello.introduce() # будет выведено
                      # 'Я функция'
```

Таким образом, мы добавили функции `say_hello` новое свойство `description` и новый метод `introduce`.

Но что, если мы хотим сами определить структуру объекта? А если нужно создать много объектов с собственной структурой? Неужели для этого нужно каждый раз создавать функцию и к ней дописывать новые свойства и методы? Нет!

Классы

Python позволяет определять собственные типы данных с помощью классов. Класс — это описание того, какие данные и функциональность будут у объекта. Это своего рода чертёж, на основании которого создают объекты — экземпляры данного класса. Класс устанавливает в свои экземпляры свойства (данные) и методы (функциональность).

На самом деле все базовые типы данных Python тоже являются классами. Это можно увидеть, если вывести тип переменной в консоль.

```
a=123
print(type(a)) # Будет выведено
               # <class 'int'>
```

Чтобы объявить класс, используем ключевое слово `class`. После него напишем имя класса — переменную, к которой обратимся, если захотим его использовать. В языках программирования классы принято называть с заглавной буквы.

```
class My_class:
    def __init__(self, object_name):
        self.name = object_name
        self.surname = None

    def introduce(self): # тут параметр self
        print('Я пользовательский класс')


my_object = My_class( 'мой первый собственный объект' )
my_object.introduce() # тут вызвали метод без аргументов
```

`Self` — это ключевое слово, которое доступно внутри любого класса. `Self` хранит ссылку на текущий объект.

Любой класс содержит метод `__init__`, который вызывается при создании нового объекта этого класса. Этот метод называется **конструктором** класса. Он нужен для определения действий, которые должны производиться при создании объекта. Часто используется, чтобы заполнить будущий объект данными.

Чтобы задать метод, который будет у всех объектов описываемого класса, нужно при определении этого метода первым параметром обязательно указать `self`. Однако при вызове метода этот параметр не учитывается.

Запись `My_class('мой первый собственный объект')` возвращает новый объект, который мы заботливо сохраняем в переменную `my_object`.

У полученного таким образом объекта будет **не строгая структура**. Т. е. мы можем переопределить, добавить или удалить свойства или методы этого объекта.

```
def say_bye():
    print('Пока!')

class My_class:
    def __init__(self, object_name):
        self.name = object_name
        self.surname = None

    def introduce(self):
        print('Я пользовательский класс')

my_object = My_class('мой первый собственный объект')

my_object.introduce()  # Будет выведено
                       # 'Я пользовательский класс'
my_object.introduce = say_bye # Переопределили метод
my_object.introduce()  # Будет выведено
                       # 'Пока'

my_object.surname = 'У объекта нет фамилии' # Переопределили свойство
print(my_object.surname) # Будет выведено
                         # 'У объекта нет фамилии'

del my_object.name # Удалили свойство
print(my_object.name) # Будет ошибка, так как
                      # такого свойства у объекта больше нет
```

Использование self

Как было сказано ранее, `self` — это ссылка на текущий объект. Используя ключевое слово `self`, мы можем в внутри класса обращаться к свойствам и методам объекта. В методе `__init__` с его помощью мы определили и задали значения свойствам `name` и `surname`. Используя `self`, также можно обратиться к свойству `name` внутри метода `introduce`.

```
class My_class:
    def __init__(self, object_name):
        self.name = object_name
        self.surname = None

    def introduce(self):
        print('Привет! Я', self.name)

my_object = My_class( 'Курлык' )
my_object.introduce() # Будет выведено
                      # 'Привет! Я Курлык'
```

Наследование

Наследование в ООП — это возможность создать класс на основе других классов.

Допустим мы описали класс `Person` (Человек). У объектов этого класса есть такие свойства как: `name`, `surname`, `age` (имя, фамилия, возраст).

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

Далее нам нужно описать класс `Student` (Студент). Но любой студент ведь тоже является человеком! Он обладает всеми теми свойствами и методами, что и объекты класса `Person`. Однако, помимо всех свойств человека, у студента ещё должны быть свойства: `university`, `faculty` (университет, факультет). Чтобы не копировать все свойства и методы, описанные в классе `Person`, при описании класса `Student` пользуются наследованием.

```
class Student(Person):
    def __init__(self, name, surname, age, university, faculty):
        super().__init__(name, surname, age)
        self.university = university
        self.faculty = faculty

    def introduce(self):
        print('Я студент!')
```

Наследование позволяет объектам дочернего класса получить все те же свойства и методы, которые доступны объектам родительском.

В данном случае класс `Person` является **родительским** по отношению к `Student` (в то же время `Student` является **дочерним** по отношению к `Person`).

Чтобы объекты дочернего класса получали все свойства и методы, описанные для объектов родительского класса, после объявления имени дочернего класса в круглых скобках указывают имя родительского класса.

В дочернем классе мы можем вызывать ключевое слово `super()`– ссылку на объект родительский класс. Благодаря нему в конструкторе класса `Student` мы можем вызвать метод `__init__` (описанный в классе `Person`). Аналогичным образом мы могли бы обратиться к любому другому свойству или методу, описанному в родительском классе.

Переопределение свойств и методов объектов родительского класса

Но что будет, если при описании новых методов и свойств в дочернем классе, мы дадим им те же имена, что уже были использованы при описании родительского класса? Ответ прост! Они перезапишутся!

```
class Student(Person):
    def __init__(self, name, surname, age, university, faculty):
        super().__init__(name, surname, age)
        self.university = university
        self.faculty = faculty

    def introduce(self):
        print('Я студент!')


class BaumanStudent(Student):
    def __init__(self, name, surname, age, faculty):
        super().__init__(name, surname, age, 'МГТУ', faculty)

    def introduce(self):
        print('Я инженер!')


some_student = Student('Вася', 'Пупкин', 20, 'МГУ', 'ВМК')
some_student.introduce() # Будет выведено
                         # 'Я студент!'

bauman_student=BaumanStudent('Митя', 'Иванов', 20, 'РТ')
bauman_student.introduce() # Будет выведено
                           # 'Я инженер!'
```

В данном случае в дочернем классе `BaumanStudent` мы переопределили методы `__init__` и `introduce`.

Множественное наследование

Python поддерживает множественное наследование.

При объявлении дочернего класса в круглых скобках можно указать несколько родительских классов через запятую. Тогда объекты дочернего класса будут наследовать все свойства и методы объектов родительских классов.

```
class Employee:
    def work(self):
        print("Employee works")

class Student:
    def study(self):
        print("Student studies")


class WorkingStudent(Employee, Student):
    pass

tom = WorkingStudent()
tom.work()    # Будет выведено 'Employee works'
tom.study()   # Будет выведено 'Student studies'
```

Множественное наследование может быть удобным, однако стоит учитывать, что если в родительских класса были описаны методы или свойства с одинаковыми именами, то объекты дочернего класса получают свойства и методы того родителя, который был указан **первым**.

```
class Employee:
    def do(self):
        print("Employee works")


class Student:
    def do(self):
        print("Student studies")


# class WorkingStudent(Student,Employee):
class WorkingStudent(Employee, Student):
    pass


tom = WorkingStudent()
tom.do()   # Будет выведено "Employee works"
```

Таким образом, объекты дочернего класса всегда получают конструктор объектов первого родителя. Но что делать, если в дочернем классе мы хотим, например, определить новый `__init__`, в котором вызывается конструктор каждого из родителей?

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

class Person_With_Surname:
    def __init__(self, surname):
        self.surname = surname

class Person_With_Age:
    def __init__(self, age):
        self.age = age


class Person(Person_With_Name, Person_With_Surname, Person_With_Age):
    def __init__(self, name, surname, age):

        # super() - хранит ссылку на объект первого родителя
        super().__init__(name)

        # super(Person_With_Name, self) - хранит ссылку на объект того класса,
        # который при наследовании указан следующим после Person_With_Name
        super(Person_With_Name, self).__init__(surname)

        # super(Person_With_Surname, self) - хранит ссылку на объект того класса,
        #  который при наследовании указан следующим после Person_With_Name
        super(Person_With_Surname, self).__init__(age)

some_person = Person('Вася', 'Пупкин', 21)


class Employee:
    def work(self):
        print("Employee works")


class Student:
    def study(self):
        print("Student studies")


class WorkingStudent(Employee, Student):
    pass


tom = WorkingStudent()
tom.work()    # Будет выведено 'Employee works'
tom.study()   # Будет выведено 'Student studies'
```

Свойства и методы класса

Мы научились определять свойства и методы объектов класса. Но сам класс ведь тоже является объектом. А значит тоже обладает методами и свойствами. Такие свойства и методы называют **свойствами и методами класса**. Не путайте с **свойствами и методами объектов (экземпляров) класса**!

```
class Person:
    type = "Person"

    @staticmethod
    def say_type():
        print('This class type is:', Person.type)


print(Person.type) # Будет выведено 'Person'
Person.say_type() # Будет выведено 'This class type is: Person'
```

Свойства класса определяют сразу под именем класса без использования ключевого слова `self`. Методы класса определяют с использованием аннотации `@staticmethod` и без использования `self` в качестве первого параметра. Такие методы называются **статическими**.

Подобные свойства и методы являются общими для всех объектов класса:

```
class Person:
    type = "Person"

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


tom = Person("Tom")
bob = Person("Bob")
print(tom.type)   # Будет выведено 'Person'
print(bob.type)   # Будет выведено 'Person'

# изменим атрибут класса
Person.type = "Class Person"
print(tom.type)  # Будет выведено 'Class Person'
print(bob.type)  # Будет выведено 'Class Person'
```

Свойства класса обычно применяются, когда надо определить некоторые общие данные для всех объектов класса, независимые от конкретного экземпляра.

Аналогично статические методы обычно определяют общее поведение, которое не зависит от конкретного экземпляра.

Объявление класса и создание объекта

```
class My_class:
    def __init__(self, object_name):
        self.name = object_name
        self.surname = None

    def introduce(self):
        print('Привет! Я', self.name)


my_object = My_class( 'мой объект' )
my_object.introduce() # тут вызвали метод
                      # без аргументов
```

Проверка типа объекта

Функция isinstance() позволяет проверить, является ли объект экземпляром какого-либо класса.

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

a = My_class('мой  объект')
print(isinstance(a, My_class))
# Будет выведено 'True'
```

Свойства и статические методы классов

```
class Person:
    type = "Person"

    @staticmethod
    def say_type():
        print('Type is:', Person.type)

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


tom = Person("Tom")
bob = Person("Bob")
print(tom.type)   # Будет выведено
'Person'
print(bob.type)   # Будет выведено
'Person'

# изменим атрибут класса
Person.type = "Class Person"
print(tom.type)  # Будет выведено
                 # 'Class Person'
print(bob.type)  # Будет выведено
                 # 'Class Person'

tom.say_type() # Будет выведено
               # 'type is: Class Person'
bob.say_type() # Будет выведено
               # 'type is: Class Person'
```

Наследование

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

    def say_hello(self):
        print('Привет! Я', self.name)


class Student(Person):
    def __init__(self, name, surname, age,
university):
        super().__init__(name, surname, age)
        self.university = university
        self.faculty = faculty
        super().say_hello()


some_student = Student('Вася', 'Пупкин', 20, 'МГТУ')
# Будет выведено 'Привет! Я Вася'
```

Множественное наследование

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


class Surname:
    def __init__(self, surname):
        self.surname = surname

class Age:
    def __init__(self, age):
        self.age = age


class Person(Name, Surname, Age):
    def __init__(self, name, surname, age):
        super().__init__(name)
        super(Name, self).__init__(surname)
        super(Surname, self).__init__(age)

some_person = Person('Вася', 'Пупкин', 12)
```