Урок 25. Основы ООП, классы и объекты. Инкапсуляция, Полиморфизм, Наследование, Абстракция

Основы ООП, классы и объекты

В языке Python объектно-ориентированное программироване (ООП) играет ключевую роль. Даже программируя в рамках структурной парадигмы, вы все равно пользуетесь объектами и классами, пусть даже встроенными в язык, а не созданными лично вами.

Ключевыми понятиями ООП являются **класс** и **объект**. То есть код программы или проекта в целом базируется на классах и объектах.

Класс — это тип (можно сказать шаблон или макет), который определяет как ведет себя объект. Класс можно сравнить с чертежом, по которому создаются объекты. **В Python существует 1 стандартный суперкласс (класс-предок) — object** (не путать с Объектом!), все классы наследуют его методы.

Объект — это экземпляр класса (в Python принято называть **instance**).

```
# простой класс на Python
class Foo:
    pass
foo_instance = Foo() # экземпляр класса Foo
```

Метод — это **функция**, которая определена внутри класса.

```
class Baz:
    def check(self):
        print("Вызван метод check класса Baz")
Baz().check() # вызов метода check после инициализации класса Baz
baz_instance = Baz() # создание экземпляра класса Baz
baz_instance.check() # вызов метода check для текущего экземпляра
```

Метакласс — это класс класса, метакласс определяет поведение класса. Класс является экземпляром метакласса. **В Python существует 1 стандартный метакласс — type**, он лежит в основе Python и его логика полностью написана на языке программирования **C**, как и большинство встроенных пакетов в Python.

> На Python можно написать всё, на C можно написать Python. © Джейсон Стейтем

```
class Bar(object, metaclass=type):
    pass
# идентичный с Foo класс, стандартная реализация,
# которая происходит автоматически при создании любого класса
```

Для проверки, является ли экземпляр *instance1* экземпляром класса *Class1* существует встроенная функция: `isinstance(instance1, Class1)` которая возвращает булевое значение результата проверки.

P.S. эта функция поддерживает такой синтаксис: `isinstance(instance1, (Class2, Class3))`, что будет эквивалентно с `isinstance(instance1, Class2) or isinstance(instance1, Class3)`

> **P.S.** В Python **всё является подклассом object** - и строки, и списки, и словари, и всё остальное (и даже сам метакласс **type**).

```
def my_def(x): return x
my_types = (
    1, 1.0, "1", ["1"], (1,), {1}, {1: 1}, None,
    lambda x: x, my_def, type, object
)
if all(isinstance(item, object) for item in my_types):
    print("Everything is instance of object superclass")
```

> Поэтому любое выражение `isinstance(<everything>, object)` в Python реализации всегда будет возвращать **True**

Инкапсуляция

Инкапсуляция — ограничение доступа к **атрибутам** экземпляра класса. К атрибутам класса относят атрибуты данных и атрибуты-методы.

Python предоставляет 3 уровня доступа к атрибутам:

1. **Публичный** (public):
   - синтаксис: нет особого синтаксиса ();
   - пример имени: `common_attr`;
   - описание: *"Используйте на здоровье."*
2. **Защищенный** (protected):
   - синтаксис: одно нижнее подчеркивание (_) в начале названия;
   - пример имени: `_protected_attr`;
   - описание: *"Можете стрелять, пока это не будет вашей ногой..."*
3. **Приватный** (private),:

- синтаксис: два нижних подчеркивания (__) в начала названия;
- пример имени: `__private_attr`);
- описание: *"Не лезь, оно тебя сожрёт!!!"*

Инкапсуляция в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними.

В итоге инкапсуляция для Python Developers выглядит примерно так:

```
class Foo:
    class_attr = "class_attr value"
    _class_attr = "_class_attr value"
    __class_attr = "__class_attr value"
    def class_method(self):
        return "class_method result"
    def _class_method(self):
        return "_class_method result"
    def __class_method(self):
        return "__class_method result"
foo_instance = Foo()
print("foo_instance.class_attr =>", foo_instance.class_attr)
print("foo_instance._class_attr =>", foo_instance._class_attr)
# AttributeError: 'Foo' object has no attribute '__class_attr'
# print(foo_instance.__class_attr)
print("foo_instance._Foo__class_attr =>", foo_instance._Foo__class_attr)
print("\nfoo_instance.class_method() =>", foo_instance.class_method())
print("foo_instance._class_method() =>", foo_instance._class_method())
# AttributeError: 'Foo' object has no attribute '__class'
```

```
# print(foo_instance.__class_method())
print("foo_instance._Foo__class_method() =>", foo_instance._Foo__class_method())
```

Полиморфизм

Полиморфизм — определение **метода** с одним названием **в разных классах**.

> **P.S.** при этом метод может полностью повторять логику (осторожно DRY) или изменять её, в пределах, нужных текущему классу.

```
from random import choice
class PregnancyTest:
    def make(self):
        # Возвращает элемент (по индексу 0 или 1) из кортежа ("|", "||")
        # который генерируется функцией choice из модуля random
        return ("|", "||")[choice((0, 1))]
class BrokenPregnancyTest:
    def make(self):
        # Не возвращает результат вовсе или показывает ошибку
        return ("", "|||")[choice((0, 1))]
# Повторите эти строчки по несколько раз
print(PregnancyTest().make())
print(BrokenPregnancyTest().make())
```

Наследование

Наследование — возможность дочерних классов наследовать **атрибуты** родительских. Дочерние классы также могут переопредять атрибуты и добавлять свои.

Для проверки, является ли *Class1* подклассом (наследован от) класса *Class2* существует встроенная функция: `issubclass(Class1, Class2)` которая возвращает булевое значение результата проверки.

P.S. эта функция поддерживает такой синтаксис: `issubclass(Class1, (Class2, Class3))`, что будет эквивалентно с `issubclass(Class1, Class2) or issubclass(Class1, Class3)`

Для получения списка всех атрибутов для экземпляра класса *instance1* существует встроенная функция `dir(instance1)`.

Для получения списка пути поиска атрибутов (*method resolution order*) в экземпляре класса *instance1* или в неком классе *Class1* можно вызвать встроенный метод `instance1.mro()` - для экземпляра, `Class1.mro()` - для класса.

```
class MainClass:
    __test = "MainClass test"
```

```
    def get_test(self):
        print("Вызов из класса:", self.__class__)
        return self.__test
class SubClass(MainClass): # SubClass наследован от класса MainClass
    __test = "SubClass test"
    # Переопределение метода test (родительского класса SubClass)
    # внутри подкласса SubClass
    # def get_test(self):
    #     print("Вызов из класса:", self.__class__)
    #     return self.__test
# Проверка на наследование MainClass от SubClass и наоборот
print("issubclass(MainClass, SubClass) =>", issubclass(MainClass, SubClass))
print(
    "issubclass(SubClass, MainClass) =>",
    issubclass(SubClass, MainClass),
    end="\n\n"
)
# Получение списка пути поиска атрибутов для классов MainClass и SubClass
print("MainClass.mro() =>", MainClass.mro())
print("SubClass.mro() =>", SubClass.mro(), end="\n\n")
# Вызов метода get_test() для MainClass и SubClass
print(MainClass().get_test())
print(SubClass().get_test())
```

Множественное наследование

При множественном наследовании у класса может быть более одного класса-предка.

> Множественное наследование — потенциальный источник ошибок, которые могут возникнуть *из-за наличия одинаковых имен методов в предках* (то есть Полиморфизма).

В Python есть возможность множественного наследования (поиск атрибутов всё также происходит по списку, который возвращает **object.mro()**):

```
class A:
    pass
class B:
    pass
# Множественное наследование (класс C одновременно наследован от класса A и B)
# (порядок имеет значение и будет влиять на .mro())
class C(B, A):
    pass
# class C(A, B):
#     pass
print(C.mro())
```

Абстракция

Абстракция — это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы.

Асбтрактный класс — это класс, который обычно предоставляет неполную функциональность и содержит один или несколько **абстрактных методов**.

Абстрактные методы — это методы, которые, как правило, не имеют никакой реализации, их оставляют на *обязательную* реализацию внутри **подклассов**.

Для создания абстрактных классов в Python используется модуль **abc**

```
from abc import ABC, abstractmethod
# AbstractClass наследован от класса ABC
# идентично записи AbstractClass(metaclass=ABCMeta) - реализация Python 3.0+
class AbstractClass(ABC):  # реализация Python 3.4+
    # Создание абстрактного метода required_method при помощи
    # декоратора @abstractmethod
    @abstractmethod
    def required_method(self): pass
# MainClass наследован от абстрактного класса AbstractClass и должен реализовать
# все абстрактные методы
class MainClass(AbstractClass):
    pass  # Приведёт к ошибке
    # def required_method(self): pass
# TypeError: Can't instantiate abstract class MainClass
# with abstract methods required_method
main_instance = MainClass()
```

```
class AbstractClass2:
    def required_method(self): raise NotImplementedError
class MainClass2(AbstractClass2):
    pass
    # def required_method(self): pass
# Создние экземпляра произойдёт успешно
main_instance2 = MainClass2()
# Попытка вызвать метод
# raise NotImplementedError
main_instance2.required_method()
```