Лекция 15. Введение в ООП, ОПП в скриптовых языках

Коновалов А. В.

5 декабря 2022 г.

Объектно-ориентированное программирование

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — парадигма программирования, представляющая программу как множество взаимодействующих объектов — сущностей, сочетающих в себе хранимые данные и какое-то своё поведение.

В «чистом» ООП объекты взаимодействуют между собой путём посылки *сообщений,* когда объект принимает сообщение, он вызывает *метод* для его обработки. Сейчас редко говорят о посылке сообщений, чаще их называют просто *вызовами методов* — метод понимается как функция, связанная с объектом.

Класс

Класс в «чистом» ООП — множество объектов, имеющих общее (сходное) поведение, каждый объект принадлежит какому-то классу.

Объекты, принадлежащие некоторому классу, называют **экземплярами** этого класса.

Иерархия классов

Классы образуют *иерархию,* т.е. могут вкладываться друг в друга. Продолжая аналогию со множествами, если класс вкладывается в другой класс, т.е. является его частным случаем, то он называется **подклассом** (или **производным классом**). Если класс `D` является подклассом класса `B`, то объекты класса `D` также являются объектами класса `B`. Соответственно, **надклассом** (или **базовым классом**) некоторого класса называется класс, в который он вкладывается.

Т.к. подклассы создаются из надклассов, то подклассы часто называют производными классами или классами-потомками, а надклассы — базовыми классами или классами-предками, родительскими классами.

Три принципа ООП

Парадигма ООП базируется на «трёх китах» — большинство реализаций ООП в разных объектно-ориентированных языках программирования эти принципы поддерживают:

1. Инкапсуляция
2. Наследование
3. Полиморфизм

Инкапсуляция

Инкапсуляция — объект скрывает своё внутренее устройство от других объектов, извне к содержимому объекта обращаться нельзя (можно только к тому, что сам объект разрешит), можно только вызывать методы объекта.

Если нужно изменить какое-то свойство объекта,
то в соответствии с принципом инкапсуляции доступ к свойству осуществляется при помощи двух методов: «геттера» (от слова «get») и «сеттера» (от слова «set»). Их ещё называют, соответственно, аксессор (от слова «access») и модификатор. Свойство с доступом на чтение реализуется с помощью одного геттера без соответствующего сеттера.

Наследование (1)

Наследование — возможность строить новые классы на основе существующих, т.е. создавать подклассы. Подкласс содержит те же данные, которые содержит класс-предок, но кроме того может содержать и какие-то новые данные, характерные для потомка — подкласс *наследует данные* предка. Подкласс также *наследует поведение* предка, т.е. методы, которые были у предка, по умолчанию наследуются потомком. Однако, потомок может *переопределять* методы предка, т.е. предоставлять для них новую реализацию, т.е. другое поведение на те же сообщения. Также потомок может реализовывать новые методы, которых не было у предка.

Наследование (2)

Интерфейс объекта — множество сообщений, которые объект может получать (иначе говоря, каким образом можно взаимодействовать с объектом). В соответствии с принципом наследования интерфейс у объекта тоже наследуется — те методы, которые можно было вызвать у предка, можно вызывать и у потомка, они либо унаследованы, либо переопределены.

Полиморфизм

Полиморфизм — возможность взаимозаменяемо использовать объекты различных классов с общим интерфейсом. В частности, если некоторый код работает с объектами базового класса, то он будет без изменений работать и с любыми его потомками, поскольку потомки интерфейс наследуют.

В языках со статической типизацией (вроде Java или C++) интерфейсы определяются явно как перечни методов, для классов нужно явно указывать базовый класс и реализуемые интерфейсы, для переменных нужно явно указывать или класс, или интерфейс. В языках с динамической типизацией это делать не обязательно.

Утиная типизация

Утиная типизация — случай, когда интерфейс объекта определяется неявно, как множество контекстов использования данного объекта. Происходит от английской идиомы «утиный тест»: «если что-то плавает как утка, крякает как утка, то, скорее всего, это утка». Утиная типизация характерна для динамически типизированных языков.

Утиная типизация (пример)

Пример утиной типизации. Функция `mysum()`:

```
def mysum(xs):
    res = xs[0]
    for x in xs[1:]:
        res += x
    return res
```

складывает элементы непустого списка, используя операцию +. Она будет применима и к спискам чисел (где + означает сложение), и к спискам строк, списков или кортежей (где + означает конкатенацию), и к спискам любых других объектов, для которых каким-то образом определена операция +. Более того, она определена не только для списков, но и вообще для любых итерируемых объектов. Т.е. функция `mysum` определена для итерируемых объектов, поддерживающих операцию +.

ООП в скриптовых языках на примере Python

Объектно-ориентированное программирование в Python

Python — динамически типизированный объектно-ориентированный язык, в частности, поддерживающий утиную типизацию. В Python все типы данных являются классами, в частности, встроенные типы вроде `int`, `float`, `str`, `list` и т.д. — тоже классы. Привычные нам операции вроде + являются неявным вызовом методов, например + — метод `__add__`, * — метод `__mul__` и т.д.

В Python есть корневой базовый класс `object`, который является предком (прямым или косвенным) всех типов данных.

Объекты в Python'е содержат как данные, так и методы. Данные, хранящиеся в экземплярах (объектах), называются *полями* или *атрибутами.*

Синтаксис определения класса (1)

Классы в Python’е определяются следующим образом:

```
class ‹имя-класса›:
    ‹определения›

class ‹имя-класса›(‹имя-предка›, ‹имя-предка›…):
    ‹определения›
```

Если предок явно не указан, подразумевается предок `object`. Предков в Python может быть несколько (т.н. множественное наследование). Пользоваться множественным наследованием не рекомендуется.

Синтаксис определения класса (2)

Определения — это присванивания переменным (переменным класса) и определения функций (методов). Переменные, описанные в определении класса, являются общими для всех экземпляров этого класса, поэтому их называют *переменными класса,* в отличие от переменных (атрибутов или полей) экземпляра, которые у каждого объекта свои.

Если определений нет (иногда нужен просто пустой класс), то вместо определений пишется ключевое слово `pass`.

Синтаксис определения класса (3)

Метод объекта определяется как обычная функция при помощи ключевого слова `def`, однако, обязана принимать не менее одного параметра. Первым параметром метода всегда является ссылка на экземпляр класса, для которого этот метод вызван, этот параметр принято называть `self`.

Классы в Python передаются **по ссылке,** т.е. экземпляры как-то «плавают» в памяти, а сами переменные хранят не значения, а ссылки, т.е. адреса, указывающие на местоположения объектов в памяти. Присвоение значения переменной приводит не к копированию значения, а к копированию ссылки. Адрес можно увидеть, вызвав встроенную функцию `id(obj)` — если два адреса равны, значит, это один и тот же объект, если не равны — два разных.

Конструктор объекта

Конструктор объекта — это операция создания объекта. Выглядит она как вызов функции, имя которой является именем класса:

```
‹имя-класса›(‹параметры›…)
```

Конструктор объекта вызывает специальный метод-инициализатор `__init__`, который принимает ссылку на создаваемый объект и параметры конструктора. Часто метод `__init__` также называют конструктором объекта. Если параметры конструктора не соответствуют параметрам метода `__init__`, то мы получаем ошибку.

Обращения к атрибутам объектов (1)

Для создания или модификации атрибута объекта нужно написать `‹объект›.‹имя-атрибута›`. Если атрибут у объекта уже был, то он получит новое значение, если не было, то он будет создан (точно также, как и в случае присваивания переменной). Добавлять новые атрибуты встроенным типам (`int`, `str`, `list`, `object`…) нельзя.

```
class C:
    common = 10


x = C()
y = C()
print(C.common)              # 10
print(x.common)              # 10
print(y.common)              # 10
```

Обращения к атрибутам объектов (2)

```
C.common = 20
print(x.common)                 # 20
print(y.common)                 # 20

x.field = 30
print(x.field)                  # 30

y.field = 40
print(x.field)                  # 30
print(y.field)                  # 40

x.common = 50
print(x.common)                 # 50
print(y.common)                 # 20
```

Обращения к атрибутам объектов (3)

Здесь определён класс `C` и два его экземпляра `x` и `y`. Атрибут `common` — это атрибут самого класса `C`, виден также у обоих экземпляров. Затем мы каждому экземпляру создали по атрибуту (полю) `field`, которым присвоили разные значения. Это два атрибута разных экземпляров, их можно менять независимо. Можно создать атрибут, имя которого совпадает с атрибутом класса, тогда атрибут экземпляра сокроет собой одноимённый атрибут класса (последние две строчки вывода).

Обычно поля объекта создаются в методе `__init__`, создавать новые поля (т.е. им делать первое присваивание) в других методах и вообще вне класса не принято (хотя и возможно, см. код выше по тексту).

Пример: класс Point (1)

Рассмотрим пример. Определим класс Point, представляющий точку в декартовой системе координат. У него будут атрибуты x и y и метод dist(p), вычисляющий расстояние до другой точки. Атрибуты будут инициализироваться при создании точки.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def dist(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return (dx**2 + dy**2)**0.5

p = Point(10, 10)
q = Point(14, 13)
print(p.dist(q))                # Будет выведено 5.0
```

Пример: класс `Point` (2)

В инициализаторе в переменной `self` находится ссылка на создаваемый объект, этому объекту мы инициализируем атрибуты `x` и `y`. Функция `__init__` принимает три параметра, поэтому конструктор вызывается с двумя параметрами, т.к. первый передаётся неявно.

При создании точки `p` объект, который будет присвоен переменной `p`, передаётся как `self`, параметры `x` и `y` получают значения `10` и `10` соответственно.

При вызове `p.dist(q)` значение `p` передаётся как параметр `self`, значение `q` — как параметр `other`. Т.е. если в методе мы определили N+1 параметров, то при вызове мы передаём N параметров, т.к. первый параметр передаётся неявно — ссылка на объект перед точкой в вызове метода.

Инкапсуляция в Python’е (1)

Вообще, реализация инкапсуляции в языке программирования — это языковое средство, запрещающее программисту напрямую добраться к содержимому объекта. При попытке нарушить инкапсуляцию программист получает либо ошибку компиляции, либо ошибку во время выполнения. Но, как правило, инкапсуляцию можно «обойти» или «взломать» — это разновидность «защиты от дурака».

В Python инкапсуляция обходится легко, надо правильно понимать, что это «защита от дурака».

Инкапсуляция в Python’е (2)

Для того, чтобы сокрыть атрибут или метод класса от доступа извне, имя нужно начать с двух знаков прочерка __. В этом случае Python переименует это поле.

В примере с классом `Point` выше мы могли обратиться к полям точки напрямую:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def dist(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return (dx**2 + dy**2)**0.5
```

Инкапсуляция в Python’е (3)

```
p = Point(10, 10)
q = Point(14, 13)
print(p.dist(q))                # 5.0
print(q.x, q.y)                 # 14 13
```

Инкапсуляция в Python’е (4)

Если мы переименуем атрибуты точки в __x и __y, то напрямую к ним обратиться не сможем:

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def dist(self, other):
        dx = self.__x - other.__x
        dy = self.__y - other.__y
        return (dx**2 + dy**2)**0.5

p = Point(10, 10)
q = Point(14, 13)
print(p.dist(q))
print(q.__x, q.__y)
```

Инкапсуляция в Python’е (5)

```
5.0
Traceback (most recent call last):
  File "D:/…/class_test.py", line 14, in <module>
    print(q.__x, q.__y)
AttributeError: 'Point' object has no attribute '__x'
```

На первый взгляд странно, что не содержит — ведь мы внутри `__init__` пишем `self.__x = x`, т.е. как бы создаём поле.
Почему так получается?

Инкапсуляция в Python’е (6)

Внутри определения класса (т.е. в блоке кода под `class ‹имя-класса›:`) имена вида `__‹имя-поля›` неявно переименовываются в `_‹имя-класса›__‹имя-поля›`.

Т.е. атрибуты точки на самом деле получают имена `_Point__x` и `_Point__y`. По этим именам к ним уже можно обратиться:

```
class Point:
    ‹…пропущено…›

p = Point(10, 10)
q = Point(14, 13)
print(p.dist(q))                    # 5.0
print(q._Point__x, q._Point__y)     # 14 13
```

Инкапсуляция в Python’е (7)

Если внутри класса по ошибке обратиться к несуществующему полю, то в сообщении об ошибке мы увидим уже переименованное имя. Для примера вместо `__y` напишем в методе `dist` `__z`:

```
…
    def dist(self, other):
        dx = self.__x - other.__x
        dy = self.__z - other.__y
        return (dx**2 + dy**2)**0.5
…
```

Инкапсуляция в Python’е (8)

Получим сообщение, в котором фигурирует не `__z`, а `_Point__z`:

```
Traceback (most recent call last):
  File "D:/…/class_test.py", line 13, in <module>
    print(p.dist(q))
  File "D:/…/class_test.py", line 8, in dist
    dy = self.__z - other.__y
AttributeError: 'Point' object has no attribute '_Point__z'
```

Наследование в Python’е (1)

Мы уже говорили, что для того, чтобы унаследовать один класс от другого, нужно указать имена базовых классов в скобках после имени класса. Рассмотрим пример:

```
class Base:
    def f(self, x):
        return x*x

    def g(self, x, y):
        return x + y


class Derived(Base):
    def g(self, x, y):
        return x * y

    def h(self, x, y):
        return x ** y
```

Наследование в Python’е (2)

В классе `Base` определяются два метода `f` и `g`, первый принимает один параметр и возводит его в квадрат, второй принимает два параметра и их складывает.

Метод `Derived` наследует от `Base` метод `f` без изменений, переопределяет метод `g`, что он уже перемножает параметры и добавляет метод `h`, возводящий один аргумент в степень другого. Таким образом, у `Derived` 3 метода.

Наследование в Python’е (3)

```
>>> b = Base()
>>> b.f(15)
225
>>> b.g(3, 5)
8
>>> b.h(3, 5)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    b.h(3, 5)
AttributeError: 'Base' object has no attribute 'h'
>>> d = Derived()
>>> d.f(15)
225
>>> d.g(3, 5)
15
>>> d.h(3, 5)
243
```

Наследование в Python’е (4)

Метод `__init__`, как и любые другие методы, может наследоваться. Т.е. если у нас есть метод `__init__` в базовом классе, а в производном классе `__init__` не определён, то будет вызываться метод `__init__` базового класса при создании производного, конструктор производного класса будет принимать те же параметры, что и конструктор базового класса.

Наследование в Python’е (5)

Если инициализатор производного класса определяется, то в нём, как правило, нужно вызывать инициализатор базового класса.

Его можно вызвать двумя способами:

- либо использовать вызов `‹имя-суперкласса›.__init__(self, ‹параметры›)`,
- либо `super().__init__(‹параметры›)`.

В первом случае мы явно вызываем метод как переменную класса, хранящую функцию и передаём в него `self` и необходимые параметры. Во втором случае вызываем встроенную функцию `super()`, которая строит объект-заместитель, через который можно вызвать метод базового класса, даже если он переопределён в потомке.

Наследование в Python’е (6)

Определим класс `Base` с атрибутами `x` и `y` и его потомок `Derived` с новым атрибутом `z`.

Пример использования первого способа:

```
class Base:
    def __init__(self, x, y):
        self.x = x
        self.y = y


class Derived(Base):
    def __init__(self, x, y, z):
        Base.__init__(self, x, y)
        self.z = z
```

Наследование в Python’е (7)

Пример второго способа:

```
class Base:
    def __init__(self, x, y):
        self.x = x
        self.y = y


class Derived(Base):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z
```

Наследование в Python’е (8)

Результат в обоих случаях будет одинаковым:

```
>>> b = Base(1, 2)
>>> b.x
1
>>> b.y
2
>>> d = Derived(3, 4, 5)
>>> d.x
3
>>> d.y
4
>>> d.z
5
```

Наследование и инкапсуляция в Python’е (1)

Какой метод предпочесть — зависит от предпочтений программиста. Главное — не забывать его вызвать, чтобы часть объекта, соответствующая базовому классу была правильно инициализирована.

Если мы пользуемся инкапсуляцией, то поля базового и производного класса будут переименованы по-разному:

```
class Base:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y


class Derived(Base):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.__z = z
```

```
>>> d = Derived(3, 4, 5)
>>> d._Base__x
3
>>> d._Base__y
4
>>> d._Derived__z
5
```

Наследование и инкапсуляция в Python’е (2)

Таким образом, если мы скрываем поля, добавляя в их начало __, то в базовом и производном классах можно полям давать одинаковые имена, к конфликту это не приведёт, т.к. они переименуются по-разному:

```
class Base:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
        self.__a = 10


class Derived(Base):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.__z = z
        self.__a = 20
```

```
>>> d = Derived(3, 4, 5)
>>> d._Base__a
10
>>> d._Derived__a
20
```

Полиморфизм в Python’е

В Python’е используется утиная типизация, т.е. интерфейс определяется контекстом использования, так что полиморфизм изначально в языке есть и не связан с наследованием.

Многие встроенные операции полиморфные, т.е. применимы к объектам разных типов, например, функция `len(x)` применима и строкам, и к спискам, и ко множествам и к прочим стандартным контейнерам. Операция + тоже применима к объектам разных типов.

Магические методы (1)

Часто, чтобы встроенные операции можно было применить к собственным объектам, нужно у объекта определить т.н. **магические методы** (это официальная терминология) — методы, имена которых начинаются и заканчиваются на два прочерка.

- `__init__` — инициализатор.
- `__str__` — получение строкового представления объекта, вызывается функцией `str(obj)` (т.е. конструктором строки), функцией `print()` и т.д.
- `__repr__` — получение «питоновского» образа объекта, т.е. строки, которую можно написать в программе, чтобы построить данный объект.
- `__add__`, `__sub__`, `__mul__` и много других — разнообразные арифметические операции.
- `__eq__` — сравнение на равенство. Если не определена, то объекты сравниваются по ссылке.
- `__get_item__(self, i)`, `__set_item__(self, i, x)` — обеспечивают возможность индексации вида `obj[i]` и `obj[i] = x`

Магические методы (2): пример

Большой пример на волшебные методы и полиморфные встроенные операции:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def dist(self, other):
        return abs(self - other)

    def __str__(self):
        return '<' + str(self.x) + '; ' + str(self.y) + '>'

    def __repr__(self):
        return 'Point(' + repr(self.x) + ', ' + repr(self.y)

    ‹пропущено›
```

Магические методы (3): пример

```
class Point:
    ‹пропущено›

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)

    def __abs__(self):
        return (self.x**2 + self.y**2)**0.5

‹пропущено›
```

Магические методы (4): пример

```
p = Point(10, 10)
q = Point(13, 14)
r = Point(10, 10)
print(p, q, r)
points = [p, q, r]
print(points)
print(p == q, p == r)
print(p + q, q - p, abs(q - p), q.dist(p))
```

Результат:

```
<10; 10> <13; 14> <10; 10>
[Point(10, 10), Point(13, 14), Point(10, 10)]
False True
<23; 24> <3; 4> 5.0 5.0
```

Магические методы (5): пример

Пояснения:

- Методы `__add__` и `__sub__` реализуют сложение и вычитание.
- Метод `__eq__` сравнивает на равенство две точки.
- Метод `__abs__` вызывается встроенной `abs(x)`, вычисляет расстояние до начала координат (если считать точку вектором — модуль вектора).
- Функция `print()` вызывает `__str__` для строкового представления объекта, точку мы решили записывать как координаты в угловых скобках.
- Список `list` при печати своего строкового представления вызывает `__repr__` для своих элементов. `__repr__` для точки печатает, как нужно вызвать конструктор для построения этой точки.