

Композиция

На предыдущем уроке переменные объекта были ссылками на другие объекты (прямоугольник состоял из точек, точки из x и y координат).

Отношение "состоит из" - композиция.

Автомобиль *состоит из* 4 колес, мотора, корпуса, руля и так далее.

Наследование

Другое отношение расширяет свойства другого объекта, отношение "is a" (это).

Мотоцикл - это велосипед с мотором. К функциональности велосипеда добавили мотор (и управление им).

Расширяется функциональность

Поля (переменные и методы):

- добавляются (появился мотор);
- сохраняются старыми без изменения (колеса, сиденье);
- старые методы могут измениться (ехать - теперь начинается с "завести мотор" и не нужно крутить педали).

Термины

- **Базовый класс** (родительский, родитель) - на его основе делаем новый класс.
- **Производный класс** (унаследованный класс, дочерний класс) - расширили старый класс и получили новый.

Простой пример наследования

```
import math
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def dist0(self):
        """ расстояние до точки (0,0) """
        return math.hypot(self.x, self.y)

class Circle(Point):
    # класс Circle расширяет функциональность класса Point
    def __init__(self, x=0, y=0, r=0):
        super().__init__(x, y) # вызов конструктора родительского класса
        self.r = r # добавление еще одного поля self.r - радиуса


    def area(self):
        # определение нового метода area
        return math.pi * self.r**2

    def edge_dist0(self):
        # кроме расстояния от центра окружности до (0,0) введем расстояние от окружности до (0,0)
        return abs(self.dist0() - self.r) # доступен метод базового класса self.dist0()
```

```
# закончились определения классов
p = Point(3, 4)
c = Circle(4, 3, 2.5)


print(p.dist0())    # 5
print(c.dist0())    # 5, у объекта типа Circle тоже есть метод dist0

print(c.area())     # 19.625
print(p.area())     # ОШИБКА, у объекта класса Point нет метода area.
```

 Старайтесь писать в 1 файле 1 класс, имя файла - совпадает с именем класса.

Методы `area()` и `circumference()` достаточно очевидны.

Метод `edge_dist0()` в ходе производимых вычислений вызывает метод `dist0()`. Так как класс `Circle` не реализует свой метод `dist0()`, интерпретатор найдет и будет использовать метод базового класса `Point`.

 Сначала вызовите `init` базового класса, а потом доопределяйте поля наследника.

Вдруг поля наследника зависят от полей базового класса (вычисляются по ним). Поля базового класса вряд ли зависят от наследника (мы не закладывали в базовый класс дар предвидения).

Чей атрибут?

В базовом классе и в наследнике в `init` методе запишем `self.y`. Это единая переменная на оба класса или два разных атрибута? Проверим:

```
class A:
    def __init__(self, x=[]):
        self.x = x;
        self.y = 1

class B(A):
    def __init__(self, x=[]):
        super().__init__(x)
        self.x.append('banana')
        self.y = 2

b = B(['apple'])
print(b.x)          # ['apple', 'banana']
print(b.y)          # 2
print(dir(b))

a = A(['apple'])
print(a.x)          # ['apple']
print(a.y)          # 1
print(dir(a))
```

Вывод: переменная одна, обращаемся в любом классе мы к ней по `self`.

`dir` тоже дают идентичный результат.

```
dir(b)
['_class_', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
```

```

['__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', 'x', 'y']
['apple']
dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', 'x', 'y']

```

Полиморфизм

(Цитата из Саммерфилда) Полиморфизм подразумевает, что любой объект данного класса может использоваться, как если бы это был объект любого из базовых его классов. По этой причине, когда создается подкласс, нам требуется реализовать только необходимые дополнительные методы и переопределить только те существующие методы, которые нам хотелось бы заменить. Переопределяя методы, мы можем в случае необходимости использовать реализацию базовых классов, применяя функцию `super()` внутри переопределяемых методов.

На пальцах: если объект типа B, то вызываться будет функция именно класса B (если она переопределена в классе B).

Чтобы добраться до методов именно базового класса, надо специально указывать какой именно класс.

Что будет напечатано?

```

class A:
    def __init__(self, x=[]):
        self.x = x;
        self.y = 1

    def foo(self):
        return self.y

    def qqq(self):
        pass

class B(A):
    def __init__(self, x=[]):
        super().__init__(x) # вместо super() можно написать super(A,
self), указывая конкретный класс.
        self.x.append('banana')
        self.y = 2

    def foo(self):
        return -self.y

    def zzz(self):
        return A.foo(self) # так можно добраться до методов базовых классов

b = B(['apple'])

print(b.y) # 2
print(b.foo()) # -2, вызвали foo именно класса B (по умолчанию, ибо
объект класса B)

```

```

print(b.zzz())          # 2, к методу родительского класса тоже можно
                          обратиться
print(dir(b))

a = A(['apple'])
print(a.y)              # 1
print(a.foo())          # 1
print(dir(a))

```

Посмотрим, что возвращает dir()

```

['apple', 'banana']
2
-2
2
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', 'foo', 'qqq', 'x', 'y', 'zzz']
['apple']
1
1
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__weakref__', 'foo', 'qqq', 'x', 'y']

```

Заметим, что qqq есть и в объекте класса A, и в объекте класса B.

A zzz - только в объекте класса B. **Не надо переопределять методы в наследуемом классе, если вас устраивает метод базового класса.**

Если при обращении к объекту у него нет этого метода, то интерпретатор будет искать метод в базовом классе. А если не найдет, то в базовом базового класса, пока не найдет метод. Тогда метод будет выполнен.

Если нигде в предках не найдется метод, то возникнет исключение [AttributeError?](#).

Наследие предков. Методы класса object

Когда в питоне создается объект, то сначала вызывается метод `__new__`, который создает объект. Его обычно нам хватает и не надо переопределять.

Далее вызывается метод `__init__`, который инициализирует поля объекта. Его мы переопределяем, создавая новые поля. В этом методе надо писать инициализацию добавленных полей текущего класса. Для инициализации полей родительского класса нужно **явно** вызвать метод `__init__` родительского класса.

```

super().__init__()

```

Имена методов НЕ должны начинаться и заканчиваться `__`, если это не переопределение специальных методов.

В питоне операторам сравнения соответствуют методы

Специальный метод	Пример использования	Означает
<code>__lt__(self, other)</code>	$x < y$	Возвращает True, если x меньше, чем y
<code>__le__(self, other)</code>	$x \leq y$	Возвращает True, если x меньше или равно y
<code>__eq__(self, other)</code>	$x == y$	Возвращает True, если x равно y
<code>__ne__(self, other)</code>	$x != y$	Возвращает True, если x НЕ равно y
<code>__gt__(self, other)</code>	$x > y$	Возвращает True, если x больше, чем y
<code>__ge__(self, other)</code>	$x \geq y$	Возвращает True, если x больше или равно y

Все экземпляры классов поддерживают оператор `==` и операция сравнения всегда возвращает False, если не переопределен метод `__eq__(self, other)`

Переопределим этот метод в классе Point:

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

🔧 Если в классе реализован метод `__eq__`, то метод `__ne__` (not equal, не равно), отработает корректно (инвертирует результат `__eq__`).

Теперь мы можем сравнивать (равны или нет) объекты одного и того же класса.

Но мы не можем сравнить с объектом другого типа, например переменной x типа int. Получим исключение [AttributeError?](#).

Внимание: мы можем сравнить объектом другого типа, у которого тоже есть атрибут x. Как это запретить?

- `assert isinstance(other, Point)`
- `if not isinstance(other, Point): raise TypeError?`
- `if not isinstance(other, Point): return NotImplemented?`
 - самый правильный
 - вызывается метод `other.__eq__(self)`, может тип объекта other поддерживает сравнение с Point
 - если не поддерживает (нет метода `__eq__` или возвращает `NotImplemented?`), возбуждается исключение `TypeError?`
 - 🛠️ вернуть `NotImplemented?` может только переопределенный специальный метод сравнения из таблицы.

💡 **isinstance** принимает объект и класс (или кортеж классов) и возвращает True, если объект принадлежит данному классу (или одному из классов, перечисленных в кортеже) или одному из базовых классов указанного класса (или одного из классов, перечисленных в кортеже).

```
def __repr__(self):
    return "Point({0.x!r}, {0.y!r})".format(self)

def __str__(self):
    return "({0.x!r}, {0.y!r})".format(self)
```

Разница между str и repr

Задачи

В саду работают работники и собирают яблоки. Каждый свое количество яблок в час. Эти яблоки воруют вороны. Каждая ворона может украсть свое количество в час (разом).

Нужно написать классы **Worker** и **Crow** для моделирования количества яблок в корзине с учетом сбора яблок и краж.

Пример классов:

```
class Crow:
    def __init__(self, basket, apple=0):
        self.basket = basket      # корзина из которой ворона ворует яблоки
        self.apple = apple        # сколько яблок в 1 час сможет своровать
        ворона

    def steal(self):
        """ Возвращает сколько украла яблок из корзины"""
        # надо дописать код
        return 0

class Worker:
    def __init__(self, basket, apple=0):
        self.basket = basket      # корзина в которую работник кладет яблоки
        self.apple = apple        # сколько яблок в 1 час собирает работник

    def work(self):
        """Кладет яблоки в корзину, возвращает сколько яблок собрал работник
за 1 час."""
        # надо дописать код
        return 0
```

Видно, что функциональность классов сильно совпадает. Определим класс **Person** с полями **apple** и **basket** и общими для работника и ворон функциями.

```
class Person:
    def __init__(self, basket, apple=0):
        print('Появился кто-то')
        self.basket = basket      # корзина, куда кладут или откуда берут
        self.apple = 0            # переменная, где будет храниться сколько
        яблок в час может собрать или украсть эта персона
        self.set_apple(apple)      # сделаем присвоение с проверкой

    def set_apple(self, apple):
        if apple < 0:
            self.apple = 0
        else:
            self.apple = apple % 100
        print(self)
```

```
def __str__(self):
    s = 'кто-то, будет %d яблок в час.' % (self.apple)
    s = s + '\n' + str(self.basket)
    return s
```

Теперь можно написать новый класс **Worker** - наследник класса **Person**

У нового класса **Worker** уже сразу будут функции: `__str__()`, `set_apple()` и `__init__` от класса **Person**.

Объекты класса наследника уже "умеют" то, что предоставил им класс-родитель.

```
class Worker(Person):
    def __init__(self, basket, apple=0):
        super().__init__(basket, apple) # вызываем конструктор базового
        класса
        print('Я работник, я собираю')

    def work(self):
        """Кладет яблоки в корзину, возвращает сколько яблок собрал работник
        за 1 час."""
        self.basket.add(self.apple)
        print('Работник положил %d яблок' % self.apple)
        print(self.basket)

        return self.apple
```

Пример использования **Worker**

```
if __name__ == '__main__':
    # можно корзина будет целым числом (класс int)?
    # b = 0
    b = Basket() # класс Basket напишите сами.

    # можно сделать персоны, но они не умеют ни работать, ни красть:
    p1 = Person()
    p1 = Person(220)
    p2 = Person(-30)

    # делаем рабочих
    w1 = Worker(b, 10)
    w2 = Worker(b, 30)
    print()

    # рабочие работают
    w1.work()
    w2.work()
    w2.work()

    print()
    print(w1)
    print(w2)
```

Вороны тоже пользуются корзиной и "знают" про яблоки. Можно написать класс **Crow** как наследник класса **Person**.



Задача 0

Реализуйте класс **Basket** для хранения яблок.

Добавьте в класс **Person** поле `name` и сделайте печать такую, чтобы удобно было различать разных работников.

Допишите и запустите пример с классом **Worker**

Задача 1

Реализовать и проверить класс **Crow** (Имя вороны может быть "Краа").

Задача 2

Работники и вороны в саду 8 часов. Написать программу-модель, чтобы посмотреть сколько яблок в корзине каждый час. Использовать классы-наследники **Worker** и **Crow**

Задача 3

В саду можно собирать яблоки и не получать за них зарплату (про зарплату в коде ничего не пишем), а платить за них хозяину сада.

Написать класс **Customer** который может и собирать яблоки, и покупать их из корзины по цене 2 монеты 1 яблоко.

Когда покупатель покупает яблоки, они убираются из корзины. Ворона их красть больше не может. Платить за еще не собранные яблоки покупатель не может.

Нужно считать деньги, которые покупатель заплатит за собранные яблоки. Ворона красть деньги не может.

Смоделировать 8 часов работы с покупками. В конце напечатать, сколько денег заработал хозяин сада и сколько яблок еще не продано.

-- [TatyanaDerbysheva](#) - 05 Nov 2017