Модуль 2. Наследование (4 ак. ч.)

* Принцип утиной типизации.
* Понятия базового класса и производного класса.
* Функция isinstance и ее применение.
* Практикум: Создание производного класса, применение экземпляров базового и производного класса.

[Введение 2](#_Toc146842057)

[Введение в наследование 3](#_Toc146842058)

[Супер-класс и класс-наследник 3](#_Toc146842059)

[Применение Enum 9](#_Toc146842060)

[Добавление метода 10](#_Toc146842061)

[Перегрузка операторов 14](#_Toc146842062)

[Практикум 15](#_Toc146842063)

[Решение 16](#_Toc146842064)

[Выводы 19](#_Toc146842065)

[Дополнительный материал 19](#_Toc146842066)

[Дескрипторы (data descriptor и non-data descriptor) 20](#_Toc146842067)

[Магический метод \_\_call\_\_. Функторы и классы-декораторы 26](#_Toc146842068)

[Классы-декораторы 29](#_Toc146842069)

[Объект как функция 30](#_Toc146842070)

[Магические методы \_\_eq\_\_ и \_\_hash\_\_ 30](#_Toc146842071)

[Магический метод \_\_bool\_\_ определения правдивости объектов 33](#_Toc146842072)

[Магические методы \_\_getitem\_\_, \_\_setitem\_\_ и \_\_delitem\_\_ 35](#_Toc146842073)

[Магические методы \_\_iter\_\_ и \_\_next\_\_ 37](#_Toc146842074)

[Наследование от встроенных типов данных 41](#_Toc146842075)

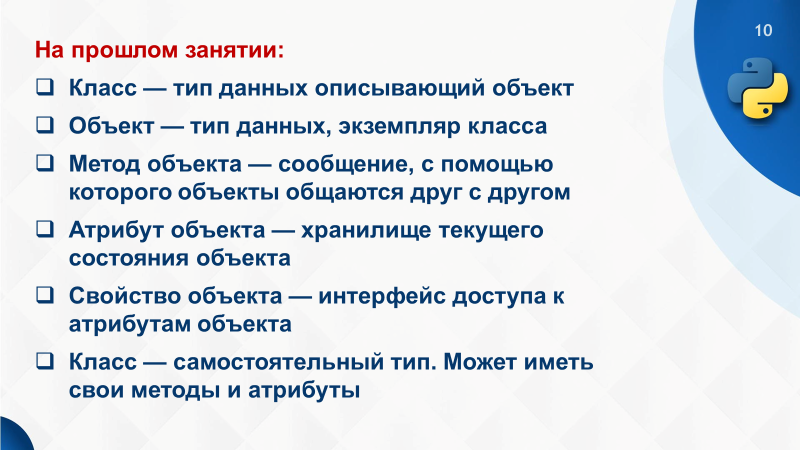
[Наследование. Атрибуты private и protected 42](#_Toc146842076)

[Атрибуты private и protected на уровне класса 45](#_Toc146842077)

[Принцип утиной типизации 46](#_Toc146842078)

# Введение

Вспомним предыдущее занятие



Используя класс Lamp – мы отработали первый принцип ООП – инкапсуляцию. Первый способ – через функцию property, второй способ через декоратор property. Мне удобнее через функцию property.

class Point:  
 \_\_count = 0  
  
 @classmethod  
 def get\_count\_points(cls):  
 return cls.\_\_count  
  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
 Point.\_\_count += 1  
  
 def move\_to(self, x, y):  
 self.x += x  
 self.y += y  
  
 def \_\_repr\_\_(self):  
 return f'Я точка: {self.x} x {self.y}'

Обратимся к предыдущему коду

p = Point(10,20)

p

Я точка: 10 x 20

А теперь мы обратимся к методу

p.move\_to(100,200)

p

Я точка: 110 x 220

p.move\_to

<bound method Point.move\_to of Я точка: 110 x 220>

или при обращении к свойству x получаем его значение

p.x

110

для отладки используется функция getattr

prop = getattr(Point, 'x')

prop

<property object at 0x00000203E6738EF0>

prop.fget

<function Point.get\_x at 0x00000203E88640E0>

Мы видим, что геттером этого свойства является функция get\_x

а сеттером

prop.fset

<function Point.set\_x at 0x00000203E8864180>

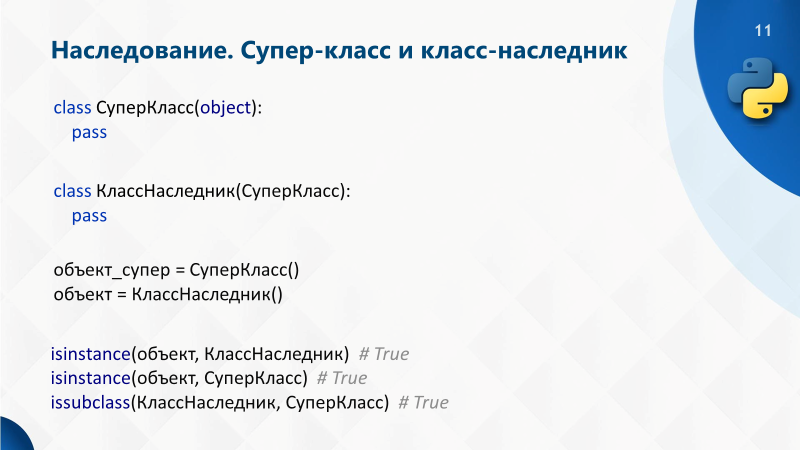
Функция set\_x

Т.е. в плане отладки мы можем получить информацию об этих методах. Скажем так неглубоко, мы видим поднаготную этих методов.

# Введение в наследование

## Супер-класс и класс-наследник

Посмотрим на слайд. Наследование.



Если мы возьмем классы, которые мы рассматривали. К примеру, что общего у списка, кортежа, строки? Они иттерабельны. У них есть длина – метод len. Но все остальное у них разное. Методы/

Смысл в следующем – одно и тоже глупо описывать несколько раз. Правда.

В этом деле нам помогает наследование.

Давайте вспомним, что в одной из ситуаций мы переопределили встроенный метод. Какой, думаем. Метод \_\_repr\_\_. Он возвращал объект (тип его). А мы заставили его писать нужную нам информацию.

Все классы состоят в некой иерархии. СуперКласс Животные, в нем подкласс Млекопитающие, Земноводные, Птицы, Рыб и т.п.

Млекопитающие делятся дальше живородящие и однопроходные. Живородящие на низшие и высшие.

Тоже самое в Пайтоне. Есть некий класс object от которого происходят (или наследуются) все остальные классы.

Но мы на этом не останавливаемся. Сделаем заготовку для изучения второго принципа ООП.

class Point:  
  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
 Point.\_\_count += 1  
  
 def move\_to(self, x, y):  
 self.x = x  
 self.y = y  
  
  
  
 def \_\_repr\_\_(self):  
 return f'Я точка: {self.x} x {self.y}'

На самом деле наш класс унаследован от того самого object

class Point(object):  
  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y

посмотрим на его начинку

pprint(object.\_\_dict\_\_)

Т.е. класс object – это Адам и Ева в одном флаконе. Это Суперкласс, в котором уже описаны базовые свойственные другим классам Методы

Там есть и \_\_repr\_\_и \_\_str\_\_ и \_\_setattr\_\_ и \_\_init\_\_ и много чего интересного.

Это означает, что наш класс Point получает по наследству всё, что там написано. Все этим магические, волшебные, дандер методы, которые там описаны.

Обычно класс, с которого всё начинается, так и называется Суперкласс (слайд). И для всех дочерних классов он таковым и является.

Если мы хотим описать какой-то класс наследник, то мы так и пишем, указывая класс в скобках класс-наследник

class СуперКласс(object):  
 pass  
  
  
class КлассНаследник(СуперКласс):  
 pass  
  
  
объект\_супер = СуперКласс()  
объект = КлассНаследник()  
  
isinstance(объект, КлассНаследник) *# True*isinstance(объект, СуперКласс) *# True*issubclass(КлассНаследник, СуперКласс)

И теперь с помощью функции isinstance мы можем вычислить всех родителей данного объекта.

Т.е. и СуперКласс и КлассНаследник являются родителями нашего объекта

Видим True, в т.ч. в цепочке наследования.

Является ли кошка экземпляром класса Кошачьи – да, а экземпляром класс Животные. Тоже – да.

А может быть экземпляром класса Домашние животные. Да, легко.

Чтобы определить принадлежность класса подклассу используется функция issubclass.

issubclass(Point, object)

Давайте создадим

class Point3D(Point):  
 pass

и создадим экземпляр этого класса

p3d = Point3D(10,20)  
print(p3d.x, p3d.y)  
print (p3d)

Вы видим, что объект класса Point3D полностью отнаследовал всё свойства и методы класса Point.

Как это произошло? При вызове конструктора он пошел в класс Point3D, там он инициализатор init не нашел, поэтому потом он полез в родительский класс Point3D и успешно проинициализировался.

Тоже самое произошло и с остальными методами. Покажи мне x и y. Покажи мне repr. А этажем выше есть. Давайте уберем repr.

Позовем isinstance и issubclass для проверки.

А теперь добавим уникальные метод для Point3dD

class Point3D(Point):  
 def move\_by(self, x, y):  
 self.x += x  
 self.y += y

Теперь Point3D имеет допонительно метод move\_by

Попробуем вызвать метод move\_by у двумерной точки

p = Point(1,2)  
p3d = Point3D(10, 20)  
p.move\_to(100,200)  
p.move\_by(100,200)  
p3d.move\_by(300,400)

видим ошибку, т.к. move\_by у point нет.

Т.е мы просто расширили поведение базового класса. Это первое. Расширение.

Второе, что мы можем сделать – это переопределить метод базового класса.

def \_\_repr\_\_(self):  
 return f'Я 3D точка: {self.x} x {self.y} x {self.z}'

Но у нас же 3d-точка. Как нам это сделать. У нас же нет конструктора для 3d-точки. Автоматом вызывается первый конструктор. Но в нем 2 параметра, а мы хотим передать три.

Самый простой способ – переопределить конструктор.

def \_\_init\_\_(self, x, y , z):  
 self.x = x  
 self.y = y  
 self.z = z

и скорректировать метод move\_by

def move\_by(self, x, y, z):  
 self.x += x  
 self.y += y  
 self.z += z

Но, однако неудобно, тупо переписано? скопировано В чем засада? А если мы хотим добавит точке четвертое измерение. Мы же не будем глупо копипастить.

Как быть, что пробрасывать, а что дописать.

Подходим к понятию конструктор суперкласса

class Point3D(Point):  
  
 def \_\_init\_\_(self, x, y, z):  
 self.z = z  
 Point.\_\_init\_\_(self, x, y)

т.о. мы переопредели конструктор, точнее его всего один параметр. Теперь из производного класса мы вызовем родительский конструктор, в который пробрасываем два параметра x и y.

Всё работает отлично, ничего не сломалось, ошибок нет.

Т.е. мы все лишь дописали нужную нам часть.

Оптимизируем таким же образом метод \_\_repr\_\_

def \_\_repr\_\_(self):  
 s = Point.\_\_repr\_\_(self)  
 return f'{s} x {self.z}'

Тоже самое с move\_to

def move\_to(self, x,y, z):  
 self.z = z  
 Point.move\_to(self,x,y)

Однако, совершенству нет предела. Каждый раз мы явно пишем имя класс Point.

А теперь будем использовать волшебное слово super ддя обращения к родительскому (или супер-классу).

def move\_to(self, x,y, z):  
 self.z = z  
 *# Point.move\_to(self,x,y)* super().move\_to(x,y)

Что-то я забыл ? self ? Он здесь не нужен!

Хотя технически мы можем его туда передать таким образом

super(Point3D,self).move\_to(x,y)

у любого объекта есть магический (волшебный атрибут) \_\_class\_

p3d.\_\_class\_\_

<class '\_\_main\_\_.Point3D'>

Который возвращает родительский класс

А у волшебного метода \_\_class\_\_ имеется еще более волшебный метод \_\_name\_\_

p3d.\_\_class\_\_.\_\_name\_\_

'Point3D'

p.\_\_class\_\_.\_\_name\_\_

'Point'

name дает имя.

Исходя из этого мы можем снова переписать метод move\_to

*#super().move\_to(x, y)  
#super(Point3D,self).move\_to(x,y)*super(self.\_\_class\_\_, self).move\_to(x, y)

Это всё одно и тоже

Также мы можем хакнуть и класс нашей лампочки

lamp1.\_\_class\_\_.count

Иногда класс используется просто как коробка методов

class Math3D:  
 PI = 3.14  
 @classmethod  
 def cos(n):  
 pass  
  
 @classmethod  
 def sin(n):  
 pass

Math3D.cos(2)

т.е когда мы будем вызывать методы, без создания экземпляров данного класса. Там могут просто лежать некие алгоритмы.

В конечном итоге наш класс Point3D мы должны переписать с использованием волшебных супер-методов

class Point3D(Point):  
  
 def \_\_init\_\_(self, x, y, z):  
 self.z = z  
 super().\_\_init\_\_(x, y)  
  
 def move\_to(self, x,y, z):  
 self.z = z  
 super().move\_to(x, y)  
  
  
 def \_\_repr\_\_(self):  
 s = super().\_\_repr\_\_()  
 return f'{s} x {self.z}'

## Применение Enum

Модуль enum содержит в себе тип для перечисления значений с возможностью итерирования и сравнения.



Его можно использовать для создания понятных обозначений вместо использования чисел (для которых приходится помнить, какое число что обозначает) или строк (в которых легко опечататься и не заметить).

from enum import Enum  
  
  
class Color(Enum):  
 RED = 1  
 GREEN = 2  
 BLUE = 3  
   
  
print(Color.RED)

Чем это отличается от такой записи?

class Color:  
 RED = 1  
 GREEN = 2  
 BLUE = 3  
  
  
print(Color.RED)

В этом случае мы получили бы просто значение атрибутов. А как достучаться до значения ‘red’.

А вот так

print(Color.RED.value)

А какой смысл, если мы можем, как угодно поменять значение енуменатора

class Color(Enum):  
 RED = 'red'

Кроме того этот класс стал итерируемым, т.е. мы можем пройтись по его атрибутам

for color in Color:  
 print(color.name, color.value)

Класс без методов, класс выполняющий специальную функцию

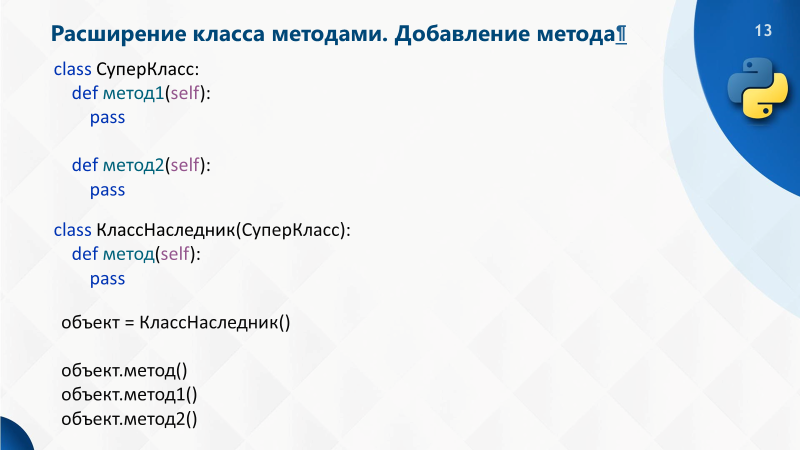
Color.RED

<Color.RED: 'red'>

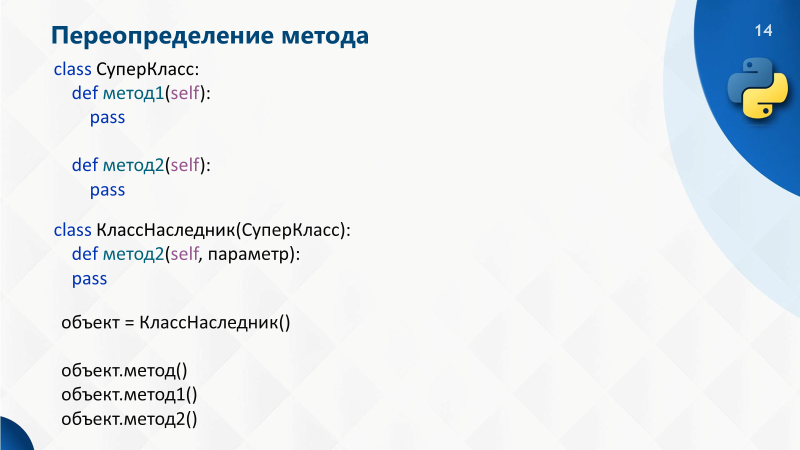
type(1)

<class 'int'>

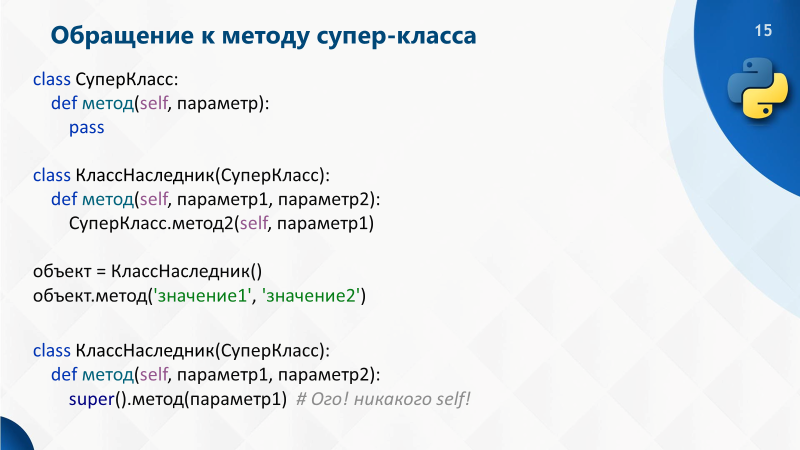
## Добавление метода



class СуперКласс:  
 def метод1(self):  
 pass  
  
 def метод2(self):  
 pass  
  
  
class КлассНаследник(СуперКласс):  
 def метод(self):  
 pass  
  
  
объект = КлассНаследник()  
  
объект.метод()  
объект.метод1()  
объект.метод2()



class СуперКласс:  
 def метод1(self):  
 pass  
  
 def метод2(self):  
 pass  
  
  
class КлассНаследник(СуперКласс):  
 def метод2(self, параметр):  
 pass  
  
  
объект = КлассНаследник()  
  
объект.метод1()  
объект.метод2('значение')



class СуперКласс:  
 def метод(self, параметр):  
 pass  
  
  
class КлассНаследник(СуперКласс):  
 def метод(self, параметр1, параметр2):  
 СуперКласс.метод2(self, параметр1)  
  
  
объект = КлассНаследник()  
  
объект.метод('значение1', 'значение2')  
  
  
class КлассНаследник(СуперКласс):  
 def метод(self, параметр1, параметр2):  
 super().метод(параметр1) *# Ого! никакого self!*

Далее разберем жизненный пример. Домовая книга, загс, регистрация граждан

class Human:  
 def \_\_init\_\_(self, name, sex):  
 self.name = name  
 self.sex = sex.upper()  
 self.status = None  
  
  
  
Создадим двух объектов класс Human.

john = Human('John', 'm')  
ann = Human('Ann', 'w')

А теперь наша задача поженить их между собой. Пишем метод

def marry(self, human):  
 if isinstance(human, Human):  
 self.status = human  
 human.status = self  
 else:  
 raise TypeError('Объект не человек')

john.marry(ann)  
  
print(ann.status)

А теперь они захотели иметь детей. Т.к. мы живем в России – однополые браки у нас воспрещены, и дети у однополых родителей не д.б. Давайте станем еще категоричнее, это у нас учебный пример. Дети должны рождаться только в браке и только в любви.

Поэтому пишем метод делать детей с учетом проверки половой разницы родителей.

def make\_children(self, partner):  
 if self.status == partner and self.sex != partner.sex:  
 return Human(random.choice(['Оля', 'Коля', 'Толя', 'Поля']), random.choice(['M', 'W']))  
 else:  
 raise Exception('Нельзя иметь детей')

Ну и имя наследника хотим увидеть

def \_\_repr\_\_(self):  
 return self.name

john = Human('Джон', 'm')  
mike = Human('Миша', 'm')  
ann = Human('Аня', 'w')  
elise = Human('Лиза', 'w')  
  
john.marry(ann)  
baby = ann.make\_children(john)  
print(baby)

print(ann.\_\_dict\_\_)  
ann.make\_children(mike) # искл  
ann.make\_children(elise) # искл

А теперь давайте будем женить наших персонажей с помощью обычных арифметических операций

К примеру john + ann

Переходим к понятию перегрузка операторов

В питоне для каждого оператора (к примеру математического) есть своя магия, волшебство, экспектопатронус.

Для этого вместо marry напишем магию \_\_add\_\_

Теперь пишем john + ann и всё работает.

john + ann

Проверим их статусы

print(john.status)  
print(ann.status)

А размножаться они будут с помощью оператора умножения

def \_\_mul\_\_(self, partner):

baby = ann \* john  
print(baby)

Мы видим, что у Ани и Джона родился Коля

Аня

Джон

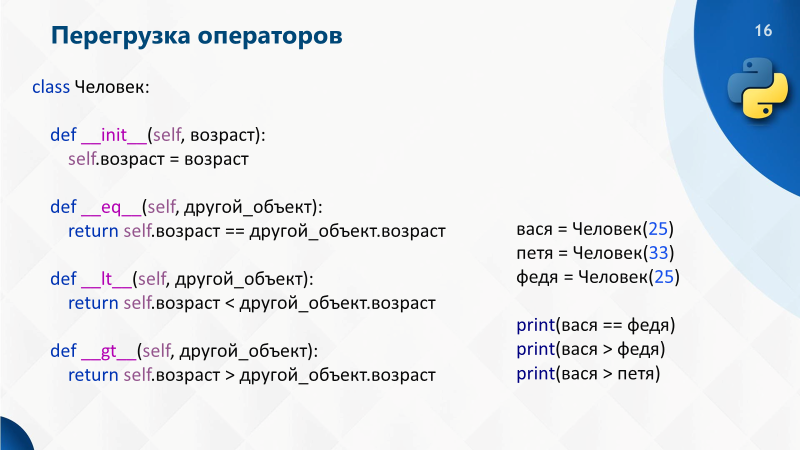
Коля

Магическая магия – умножение \_\_mul\_\_

Перегружать мы можем всё-что угодно. Почти. К примеру.

Eq = eqil, lt = less then, gt = D

## Перегрузка операторов



class Человек:  
  
 def \_\_init\_\_(self, возраст):  
 self.возраст = возраст  
  
 def \_\_eq\_\_(self, другой\_объект):  
 return self.возраст == другой\_объект.возраст  
  
 def \_\_lt\_\_(self, другой\_объект):  
 return self.возраст < другой\_объект.возраст  
  
 def \_\_gt\_\_(self, другой\_объект):  
 return self.возраст > другой\_объект.возраст  
  
  
вася = Человек(25)  
петя = Человек(33)  
федя = Человек(25)  
  
print(вася == федя)  
print(вася > федя)  
print(вася > петя)

Таким образом мы разобрали второй кит – ООП – наследование. Закрепим.

## Практикум

Создайте класс User и и его наследника класс SuperUser, которые описывают пользователя и супер-пользователя

В классе User необходимо описать:

* конструктор, который принимает в качестве параметров значения для атрибутов name, login и password
* свойства для изменения и получения значений атрибутов
* метод show\_info, который *печатает* в произвольном формате значения атрибутов name и login
* атрибут класса count для хранения количества созданных экземпляров класса User

Необходимые условия, которые надо учесть:

После создания объекта

* атрибут name доступен и для чтения, и для изменения
* атрибут login доступен только для чтения
* атрибут password доступен только для изменения

В классе SuperUser необходимо описать:

* конструктор, который принимает в качестве параметров значения для атрибутов name, login, password и role
* свойство для изменения и получения значения атрибута role
* метод show\_info, который *печатает* в произвольном формате значения атрибутов name, login и role
* атрибут класса count для хранения количества созданных экземпляров класса SuperUser

Как это должно работать

user1 = User('Paul McCartney', 'paul', '1234')

user2 = User('George Harrison', 'george', '5678')

user3 = User('Richard Starkey', 'ringo', '8523')

admin = User('John Lennon', 'john', '0000', 'admin')

user1.show\_info() *# Например: Name: Paul McCartney, Login: paul*

admin.show\_info() *# Например: Name: John Lennon, Login: john*

users = User.count

admins = SuperUser.count

print(f'Всего обычных пользователей: {users}') *# Всего обычных пользователей: 3*

print(f'Всего супер-пользователей: {admins}') *# Всего супер-пользователей: 1*

user3.name = 'Ringo Star'

print(user3.name) *# Ringo Starr*

print(user2.login) *# george*

user2.login = 'geo' *# Должна быть ошибка*

user1.password = 'Pa$$w0rd'

print(user2.password) *# Должна быть ошибка*

## Решение

class User:  
 count = 0  
  
 def \_\_init\_\_(self, n, l, p):  
 self.\_\_n = n  
 self.\_\_l = l  
 self.\_\_p = p  
 User.count += 1  
  
 def get\_name(self):  
 return self.\_\_n  
  
 def set\_name(self, n):  
 self.\_\_n = n  
  
 name = property(get\_name, set\_name)  
  
 def get\_login(self):  
 return self.\_\_l  
  
 def set\_login(self, value):  
 raise AttributeError('Нельзя менять значение')  
  
 login = property(get\_login, set\_login)  
  
 def get\_passw(self):  
 return '\*\*\*\*\*\*'  
  
 def set\_passw(self, value):  
 self.\_\_p = value  
  
 password = property(get\_passw, set\_passw)  
  
 def show\_info(self):  
 print(f'Name: {self.\_\_n}')  
 print(f'Login: {self.\_\_l}')  
  
  
class SuperUser(User):  
 count = 0  
  
 def \_\_init\_\_(self, n, l, p, r):  
 super().\_\_init\_\_(n, l, p)  
 self.\_\_role = r  
 SuperUser.count +=1  
 @property  
 def role(self):  
 return self.\_\_role  
 @role.setter  
 def role(self, value):  
 self.\_\_role = value  
  
 def show\_info(self):  
 super().show\_info()  
 print(f'Role: {self.\_\_role}')  
  
  
u = User(1, 2, 3)  
u.show\_info()  
  
s1 = SuperUser(1,2,3,4)  
s1.show\_info()

Результат:

Name: 1

Login: 2

Name: 1

Login: 2

Role: 4

Тестовые значение

user1 = User('Paul McCartney', 'paul', '1234')  
user2 = User('George Harrison', 'george', '5678')  
user3 = User('Richard Starkey', 'ringo', '8523')  
admin = SuperUser('John Lennon', 'john', '0000', 'admin')  
  
user1.show\_info() *# Например: Name: Paul McCartney, Login: paul*admin.show\_info() *# Например: Name: John Lennon, Login: john*users = User.count  
admins = SuperUser.count  
  
print(f'Всего обычных пользователей: {users}') *# Всего обычных пользователей: 3*print(f'Всего супер-пользователей: {admins}') *# Всего супер-пользователей: 1*user3.name = 'Ringo Star'  
print(user3.name) *# Ringo Starr*print(user2.login) *# george  
# user2.login = 'geo' # Должна быть ошибка*user1.password = 'Pa$$w0rd'  
*# print(user2.password) # Должна быть ошибка*

Результат:

Name: Paul McCartney

Login: paul

Name: John Lennon

Login: john

Role: admin

Всего обычных пользователей: 4

Всего супер-пользователей: 1

Ringo Star

george

Вопрос: Как банки воруют миллионы? Где ошибка

Засада count считает в двух местах

Мы должны СуперЮзера вычесть этот count обратно

def \_\_init\_\_(self, n, l, p, r):  
 super().\_\_init\_\_(n, l, p)  
 self.\_\_role = r  
 SuperUser.count += 1  
 super().count -= 1

Работает. Напоминаю как работает супер

super().count -= 1 🡺 User.count-=1

Проверим работу:

Всего обычных пользователей: 3

Всего супер-пользователей: 1

или более правильно динамические обратиться через магию \_\_class\_\_

self.\_\_class\_\_.count -= 1 #так не работает!!! Не показывать

## Выводы

Классы могут расширяться путём наследования

Наследования является одним из важных средств повторного использования готового кода в ООП

Наследованный класс содержит в себе всё, что указано для всех его родительских классов

Переопределение (override) метода — одна из возможностей ООП, позволяющая подклассу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов

Объект super используется для обращения к методу суперкласса из подкласса

Перегрузка (overload) операторов — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

# Дополнительный материал

Если в общих чертах, то идея наследования состоит в том, что новый класс · создается не на "пустом месте", а на основе уже существующего класса. Предположим, что один класс создается на основе другого класса. Тот класс, что создается, будем называть производным классом. Тот класс, на основе которого создается производный класс, будем называть базовым классом. Таким образом, по определению производный класс создается на основе базового класса.

В результате наследования все поля и функции из базового класса неявным образом "наследуются" в производном классе. С формальной точки зрения чтобы создать производный класс на основе базового при описании производного класса после имени класса в круглых скобках указывается имя базового класса.

## Дескрипторы (data descriptor и non-data descriptor)

В этом разделе занятии мы затронем довольно интересную, но не простую тему – дескрипторов классов. Я начну с одного явного недостатка объектов-свойств, о которых мы с вами уже подробно говорили. Давайте представим, что создаем класс для представления точек в трехмерном пространстве:

class Point3D:

    def \_\_init\_\_(self, x, y, z):

        self.\_x = x

        self.\_y = y

        self.\_z = z

Здесь у нас формируются защищенные локальные свойства для создаваемого объекта класса Point3D. Теперь представим, что согласно заданию координаты должны представляться исключительно целыми числами. Для этого я пропишу следующий метод проверки:

    @classmethod

    def verify\_coord(cls, coord):

        if type(coord) != int:

            raise TypeError("Координата должна быть целым числом")

А вызывать его будут в сеттерах соответствующих свойств класса:

    @property

    def x(self):

        return self.\_x

    @x.setter

    def x(self, coord):

        self.verify\_coord(coord)

        self.\_x = coord

    @property

    def y(self):

        return self.\_y

    @y.setter

    def y(self, coord):

        self.verify\_coord(coord)

        self.\_y = coord

    @property

    def z(self):

        return self.\_z

    @z.setter

    def z(self, coord):

        self.verify\_coord(coord)

        self.\_z = coord

Здесь все вам должно быть понятно. И теперь мы можем в инициализаторе использовать эти объекты-свойства для формирования локальных атрибутов экземпляров:

class Point3D:

    def \_\_init\_\_(self, x, y, z):

        self.x = x

        self.y = y

        self.z = z

Все, создавая объекты этого класса:

p = Point3D(1, 2, 3)

print(p.\_\_dict\_\_)

у нас автоматически будут формироваться нужные локальные атрибуты, и мы сможем с ними работать через объекты-свойства x, y, z.

Но, смотрите, в нашем классе Point3D получилось своеобразное дублирование: мы три раза прописывали свойства, фактически, с одинаковым функционалом. Менялись только названия методов и локальных атрибутов.

Представьте, во что превратится описание этого класса, если нужно будет задать 10 и более таких объектов-свойств! Программист во всем этом просто запутается, да и редактировать такую программу станет непросто. Как можно все это оптимизировать? Здесь нам на помощь как раз и приходят дескрипторы.

Вначале, что вообще такое дескрипторы? Это класс, который содержит или один магический метод \_\_get\_\_:

class A:

    def \_\_get\_\_(self, instance, owner):

        return ...

Или класс, в котором дополнительно прописаны методы \_\_set\_\_ и/или \_\_del\_\_:

class B:

    def \_\_get\_\_(self, instance, owner):

        return ...

    def \_\_set\_\_(self, instance, value):

        ...

    def \_\_del\_\_(self):

        ...

Первый (класс A) называется non-data descriptor (дескриптор не данных), а второй (класс B) – data descriptor (дескриптор данных). Это различие имеет смысл, но об этом позже.

Как вы уже догадались, эти магические методы, по сути, геттеры и сеттеры, а также делитер. Давайте, теперь посмотрим, как дескриптор может упростить наш программный код с обработкой координат точек.

Вначале я покажу все взаимодействие на схеме. Так как все координаты – целые числа, то интерфейс взаимодействия с ними мы определим через дескриптор с названием Integer (это имя мы, конечно же, придумываем сами):

class Integer:

    def \_\_set\_name\_\_(self, owner, name):

        self.name = "\_" + name

    def \_\_get\_\_(self, instance, owner):

        return instance.\_\_dict\_\_[self.name]

    def \_\_set\_\_(self, instance, value):

        print(f"\_\_set\_\_: {self.name} = {value}")

        instance.\_\_dict\_\_[self.name] = value

Пока не обращайте внимания на его содержимое, сейчас я все подробно объясню. Затем, в классе Point3D мы создадим три атрибута как объекты класса Integer:

class Point3D:

    x = Integer()

    y = Integer()

    z = Integer()

    def \_\_init\_\_(self, x, y, z):

        self.x = x

        self.y = y

        self.z = z

Эти атрибуты и есть дескрипторы данных, через которые будет проходить взаимодействие. Итак, когда мы создавали экземпляры классов Integer, то автоматически вызывался магический метод \_\_set\_name\_\_, в котором параметр self являлся ссылкой на создаваемый экземпляр класса; owner – ссылка на класс Point3D; name – имя атрибута (для первого объекта x, затем, y и z). В этом методе мы формируем локальное свойство с именем атрибута, добавляя перед ним одно нижнее подчеркивание (так принято делать при определении дескрипторов). В итоге, в экземплярах классов будут храниться имена \_x, \_y, \_z.

Зачем нам это нужно? Смотрите дальше. Предположим, мы создаем экземпляр класса Point3D:

pt = Point3D(1, 2, 3)

Сработает инициализатор, а в нем идет обращение к дескрипторам x, y, z. В частности, мы им присваиваем переданные значения. В этом случае, в классе Integer срабатывает сеттер (магический метод \_\_set\_\_), параметр self – это ссылка на объект дескриптора; instance – ссылка на объект pt, из которого произошло обращение к дескриптору; value – присваиваемое значение. В этом сеттере мы выводим в консоль сообщение, что был вызван данный метод и отображаем сохраненное имя и присваиваемое значение. Следующей строчкой через ссылку instance, то есть, на экземпляр класса pt, формируем в нем локальное свойство с именем self.name и присваиваем значение value. В результате, в объекте pt появляются локальные свойства \_x, \_y, \_z с соответствующими значениями.

Если затем, выполнить считывание данных через дескриптор, например, x, то автоматически сработает геттер (метод \_\_get\_\_), в котором self – это ссылка на объект Integer; instance – ссылка на экземпляр класса pt; owner – ссылка на класс Point3D. Мы здесь через ссылку instance обращаемся к словарю \_\_dict\_\_ и считываем значение нужного локального свойства, которое, затем, возвращается геттером. Это же значение автоматически возвращается и самим дескриптором.

Вот общая схема работы дескрипторов применительно к нашему классу Point3D. Теперь, сколько бы интерфейсов взаимодействия нам не понадобилось, мы легко их можем добавить в наш класс и все будет выглядеть понятно и компактно. На первый взгляд все это может показаться каким-то сложным и запутанным. Но, если внимательно во всем разобраться, то все предельно просто, только несколько громоздко. Именно громоздко, а не сложно. Поэтому, при необходимости, просто посмотрите несколько раз объяснение схемы и я уверен, каждый из вас поймет принцип ее работы. Ну а мы реализуем ее в нашей программе.

После создания экземпляра класса и вывода локальных свойств объекта:

pt = Point3D(1, 2, 3)

print(pt.\_\_dict\_\_)

увидим в консоли следующие строчки:

\_\_set\_\_: \_x = 1  
\_\_set\_\_: \_y = 2  
\_\_set\_\_: \_z = 3  
{'\_x': 1, '\_y': 2, '\_z': 3}

Последнее, что нужно прописать в дескрипторе – это проверку корректности данных. Для этого у нас уже есть метод verify\_coord, перенесем его в класс Integer и вызовем в сеттере:

class Integer:

    @classmethod

    def verify\_coord(cls, coord):

        if type(coord) != int:

            raise TypeError("Координата должна быть целым числом")

    def \_\_set\_name\_\_(self, owner, name):

        self.name = "\_" + name

    def \_\_get\_\_(self, instance, owner):

        return instance.\_\_dict\_\_[self.name]

    def \_\_set\_\_(self, instance, value):

        self.verify\_coord(value)

        instance.\_\_dict\_\_[self.name] = value

Теперь, если при формировании объекта указать неверный тип данных:

pt = Point3D('1', 2, 3)

то увидим сообщение об ошибке.

Еще в классе Integer я сделаю обращение к атрибутам экземпляра через стандартные функции getattr и setattr:

    def \_\_get\_\_(self, instance, owner):

        return getattr(instance, self.name)

    def \_\_set\_\_(self, instance, value):

        self.verify\_coord(value)

        setattr(instance, self.name, value)

Так будет правильнее, с точки зрения Python, чем обращение напрямую к специальной коллекции \_\_dict\_\_.

В итоге, мы с вами определили дескриптор данных (data descriptor) и на его основе создали три объекта x, y, z для интерфейса взаимодействия с координатами точки объектов класса Point3D.

Теперь вернемся к тому, что вначале я вам говорил, что важно различать дескрипторы данных и не данных. В чем разница? Ну, во-первых, очевидно, дескрипторы не данных не могут менять значения какого-либо свойства, так как не имеют сеттера и делитера. Они служат только для считывания информации. И есть второе важное отличие. Они имеют тот же приоритет доступа, что и обычные атрибуты класса. О чем здесь речь? Давайте я покажу это различие на примере. Добавим в программу еще один дескриптор, только не данных:

class ReadIntX:

    def \_\_set\_name\_\_(self, owner, name):

        self.name = "\_x"

    def \_\_get\_\_(self, instance, owner):

        return getattr(instance, self.name)

Он у нас будет считывать локальное свойство \_x. Определим его в классе Point3D:

    xr = ReadIntX()

И теперь можем использовать для считывания локального атрибута \_x:

print(pt.xr)

Как видите, все работает. Но, что будет, если мы запишем конструкцию:

pt.xr = 5

Произойдет ошибка? Нет! В экземпляре pt будет создано новое локальное свойство с именем xr и мы в этом можем убедиться:

print(pt.xr, pt.\_\_dict\_\_)

Кроме того, при обращении к pt.xr мы получаем значение 5, а не 1. Это, как раз и говорит о том, что приоритет доступа к локальным свойствам объекта и к дескриптору не данных одинаков.

Однако, если в дескриптор добавить сеттер и превратить его в дескриптор данных:

class ReadIntX:

    def \_\_set\_name\_\_(self, owner, name):

        self.name = "\_x"

    def \_\_get\_\_(self, instance, owner):

        return getattr(instance, self.name)

    def \_\_set\_\_(self, instance, value):

        setattr(instance, self.name, value)

А создание локального атрибута xr в объекте pt мы сделаем через коллекцию \_\_dict\_\_:

pt.\_\_dict\_\_['xr'] = 5

то при выполнении:

print(pt.xr, pt.\_\_dict\_\_)

увидим значение 1, хотя в объекте существует свойство xr. Это произошло потому, что приоритет обращению к дескриптору данных выше, чем к локальным атрибутам экземпляра класса. То есть, здесь все работает ровно так, как и с доступом к объектам-свойствам, о которых мы говорили на прошлых занятиях.

На этом мы завершим с вами знакомство с этой непростой темой «дескрипторы». Постарался объяснить все, как можно проще. Если тема не до конца понятна, то возможно нужно посмотреть ее несколько раз, либо совсем пропустить, так как это относительно редко используется в практике. Но знать этот функционал нужно, чтобы при необходимости не изобретать велосипед, а использовать уже встроенные возможности языка Python.

## Магический метод \_\_call\_\_. Функторы и классы-декораторы

Познакомимся с очередным магическим методом \_\_call\_\_. Магические методы еще называют:

dunder-методы (от англ. сокращения double underscore)

В дальнейшем я буду говорить магические методы. Итак, когда вызывается метод \_\_call\_\_ и для чего он нужен? Как вы уже знаете, после объявления любого класса:

class Counter:

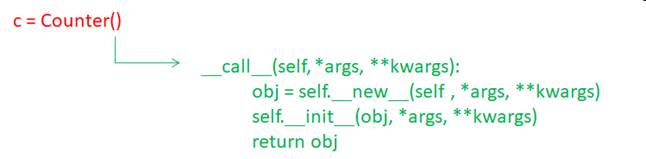
    def \_\_init\_\_(self):

        self.\_\_counter = 0

Мы можем создавать его экземпляры командой:

c = Counter()

Обратите внимание на круглые скобки после имени класса. В общем случае – это оператор вызова, например, так можно вызывать функции. Но, как видите, так можно вызывать и классы. В действительности, когда происходит вызов класса, то автоматически запускается магический метод \_\_call\_\_ и в данном случае он создает новый экземпляр этого класса:



Это очень упрощенная схема реализации метода \_\_call\_\_, в действительности, она несколько сложнее, но принцип тот же: сначала вызывается магический метод \_\_new\_\_ для создания самого объекта в памяти устройства, а затем, метод \_\_init\_\_ - для его инициализации. То есть, класс можно вызывать подобно функции благодаря встроенной для него реализации магического метода \_\_call\_\_. А вот экземпляры классов так вызывать уже нельзя. Если записать команду:

c()

то возникнет ошибка: «TypeError: 'Counter' object is not callable».

Как вы уже догадались, мы можем поправить этот момент, если явно в классе Counter пропишем магический метод \_\_call\_\_, например, так:

class Counter:

    def \_\_init\_\_(self):

        self.\_\_counter = 0

    def \_\_call\_\_(self, \*args, \*\*kwargs):

        print("\_\_call\_\_")

        self.\_\_counter += 1

        return self.\_\_counter

Здесь мы выводим сообщение, что был вызван данный метод, затем увеличиваем счетчик counter для текущего объекта на 1 и возвращаем его.

Запустим программу снова и теперь никаких ошибок нет, а в консоли отобразилась строка «\_\_call\_\_», что означает вызов магического метода \_\_call\_\_.

То есть, благодаря добавлению этого магического метода в наш класс, теперь можно вызывать его экземпляры подобно функциям через оператор круглые скобки. Классы, экземпляры которых можно вызывать подобно функциям, получили название функторы.

В нашем случае метод \_\_call\_\_ возвращает значение счетчика, поэтому с объектом можно работать, следующим образом:

c = Counter()

c()

c()

res = c()

print(res)

Мы здесь три раза вызвали метод \_\_call\_\_ и счетчик \_\_counter трижды увеличился на единицу. Поэтому в консоли мы видим значение 3. Мало того, если создать еще один объект-счетчик:

c = Counter()

c2 = Counter()

c()

c()

res = c()

res2 = c2()

print(res, res2)

То они будут работать совершенно независимо и подсчитывать число собственных вызовов.

Давайте еще раз посмотрим на определение метода \_\_call\_\_. Здесь записаны параметры \*args, \*\*kwargs. Это значит, что при вызове объектов мы можем передавать им произвольное количество аргументов. Например, в нашем случае можно указать значение изменения счетчика при текущем вызове. Для этого я перепишу метод \_\_call\_\_, следующим образом:

    def \_\_call\_\_(self, step=1, \*args, \*\*kwargs):

        self.\_\_counter += step

        return self.\_\_counter

Здесь появился в явном виде первый параметр step с начальным значением 1. То есть, можно вызывать объекты, например, так:

c(2)

c(10)

res = c()

res2 = c2(-5)

Вот общий принцип работы магического метода \_\_call\_\_. Но здесь остается, как всегда, один важный вопрос: зачем это нужно, где может пригодиться? Давайте я приведу несколько примеров его использования.

Первый пример – это использование класса с методом \_\_call\_\_ вместо замыканий функций. Смотрите, мы можем объявить класс StripChars, который бы удалял вначале и в конце строки заданные символы:

class StripChars:

    def \_\_init\_\_(self, chars):

        self.\_\_chars = chars

    def \_\_call\_\_(self, \*args, \*\*kwargs):

        if not isinstance(args[0], str):

            raise ValueError("Аргумент должен быть строкой")

        return args[0].strip(self.\_\_chars)

Для этого, в инициализаторе мы сохраняем строку \_\_chars – удаляемые символы, а затем, при вызове метода \_\_call\_\_ удаляем символы через строковый метод strip для символов \_\_chars. То есть, теперь можно создать экземпляр класса и указать те символы, которые следует убирать:

s1 = StripChars("?:!.; ")

А, затем, вызвать объект s1 подобно функции:

res = s1(" Hello World! ")

print(res)

В результате объект s1 будет отвечать за удаление указанных символов в начале и конце строки. Но нам ничто не мешает определять другие объекты этого класса с другим набором символов:

s1 = StripChars("?:!.; ")

s2 = StripChars(" ")

res = s1(" Hello World! ")

res2 = s2(" Hello World! ")

print(res, res2, sep='\n')

То есть, объект s2 уже отвечает только за удаление пробелов, тогда как s1 и некоторых других символов. Достаточно элегантное решение задачи, где нам требуется сохранять символы для удаления.

## Классы-декораторы

Второй пример – это реализация декораторов с помощью классов. Ранее мы с вами создавали декоратор для вычисления значения производной функции в определенной точке x. Я повторю эту реализацию, но с использованием класса. Вначале запишем следующий класс:

class Derivate:

    def \_\_init\_\_(self, func):

        self.\_\_fn = func

    def \_\_call\_\_(self, x, dx=0.0001, \*args, \*\*kwargs):

        return (self.\_\_fn(x + dx) - self.\_\_fn(x)) / dx

Здесь в инициализаторе сохраняем ссылку на функцию, которую декорируем, а в методе \_\_call\_\_ принимаем один обязательный параметр x – точку, где вычисляется производная и dx – шаг изменения при вычислении производной.

Далее, определим функцию, например, просто синус:

def df\_sin(x):

    return math.sin(x)

и вызове ее пока без декорирования:

print(df\_sin(math.pi/4))

После запуска программы увидим значение примерно 0.7071. Давайте теперь добавим декоратор. Это можно сделать двумя способами. Первый, прописать все в явном виде:

df\_sin = Derivate(df\_sin)

Теперь df\_sin – это экземпляр класса Derivate, а не исходная функция. Поэтому, когда она будет вызываться, то запустится метод \_\_call\_\_ и вычислится значение производной в точке math.pi/4.

Второй способ – это воспользоваться оператором @ перед объявлением функции:

@Derivate

def df\_sin(x):

    return math.sin(x)

Получим абсолютно тот же самый результат. Вот принцип создания декораторов функций на основе классов. Как видите, все достаточно просто – запоминаем ссылку на функцию, а затем, расширяем ее функционал в магическом методе \_\_call\_\_.

Надеюсь, из этого занятия вы поняли, как работает метод \_\_call\_\_ и где он может быть использован. Конечно, я привел всего два простых примера, чтобы продемонстрировать принцип его работы. В реальных задачах, проектах, вы должны сами, используя свои знания, решать, какие механизмы, приемы следует применять для решения текущих задач. Привести алгоритмы решений на все случаи жизни просто невозможно – это уже навык алгоритмизации, которым должен владеть каждый программист. И вырабатывается он, в основном, на решении практических задач. Так что больше практикуйтесь параллельно с изучением возможностей языка Python.

## Объект как функция

Объект класса может имитировать стандартную функцию, то есть при желании его можно "вызвать" с параметрами. За эту возможность отвечает специальный метод \_\_call\_\_:

class Multiplier:

def \_\_call\_\_(self, x, y):

return x\*y

multiply = Multiplier()

multiply(19, 19) # 361

# то же самое

multiply.\_\_call\_\_(19, 19) # 361

## Магические методы \_\_eq\_\_ и \_\_hash\_\_

Затронем тему вычисления хеша для объектов классов. Вначале что это такое и зачем нужно? В Python имеется специальная функция:

hash(123)

hash("Python")

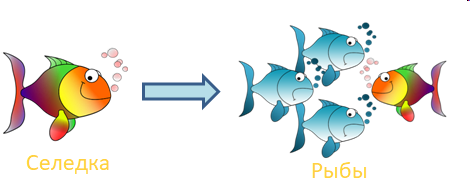
hash((1, 2, 3))

которая формирует по определенному алгоритму целочисленные значения для неизменяемых объектов. Причем, для равных объектов на выходе всегда должны получаться равные хэши:

hash("Python")

hash((1, 2, 3))

А вот обратное утверждение делать нельзя: равные хэши не гарантируют равенство объектов. Это, как в известном выражении: селедка – это рыба, но не каждая рыба селедка. С хэшами все то же самое.



Однако, если хеши не равны, то и объекты точно не равны. Получаем следующие свойства для хеша:

1. Если объекты a == b (равны), то равен и их хэш.
2. Если равны хеши: hash(a) == hash(b), то объекты могут быть равны, но могут быть и не равны.
3. Если хеши не равны: hash(a) != hash(b), то объекты точно не равны.

Причем, обратите внимание, хэши можно вычислять только для неизменяемых объектов. Например, для списков:

hash([1, 2, 3])

получим ошибку «unhashable type» - не хэшируемый объект.

Итак, мы увидели, что для любого неизменяемого объекта можно вычислять хэш с помощью функции hash(), но зачем все это надо? В действительности некоторые объекты в Python, например, словари используют хэши в качестве своих ключей. Вспомним, когда мы у словаря указываем ключ, то он должен относиться к неизменяемому типу данных:

d = {}

d[5] = 5

d["python"] = "python"

d[(1, 2, 3)] = [1, 2, 3]

В действительности, это необходимо, чтобы можно было вычислить хеш объектов и ключи хранить в виде:

(хэш ключа, ключ)

Для чего это понадобилось? Дело в том, что первоначально нужная запись в словаре ищется по хэшу, так как существует быстрый алгоритм поиска нужного значения хэша. А, затем, для равных хешей (если такие были обнаружены), отбирается запись с указанным в ключе объекте. Такой подход значительно ускоряет поиск значения в словаре.

Думаю, теперь вы представляете себе, что такое хэш и зачем он нужен. Давайте теперь рассмотрим простой класс Point для представления координат на плоскости:

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Для экземпляров этого класса:

p1 = Point(1, 2)

p2 = Point(1, 2)

мы совершенно спокойно можем вычислять хеш:

print(hash(p1), hash(p2), sep='\n')

Обратите внимание, несмотря на то, что координаты точек p1 и p2 равны, их хэши разные. То есть, с точки зрения функции hash() – это два разных объекта. Но как она понимает, равные объекты или разные? Все просто. Если оператор сравнения:

print(p1 == p2)

дает True, то объекты равны, иначе – не равны. Соответственно, для разных объектов будут получаться и разные хэши. Но раз это так, что будет, если мы переопределим поведение этого оператора сравнения с помощью магического метода \_\_eq\_\_()? Давайте попробуем:

    def \_\_eq\_\_(self, other):

        return self.x == other.x and self.y == other.y

Теперь у нас объекты с одинаковыми координатами будут считаться равными. Но при запуске программы возникает ошибка «unhashable type», то есть, наши объекты стали не хэшируемыми. Да, как только происходит переопределение оператора ==, то начальный алгоритм вычисления хэша для таких объектов перестает работать. Поэтому, нам здесь нужно прописать свой способ вычисления хэша объектов через магический метод \_\_hash\_\_(), например, так:

    def \_\_hash\_\_(self):

        return hash((self.x, self.y))

Мы здесь вызываем функцию hash для кортежа из координат точки. Этот кортеж относится к неизменяемому типу, поэтому для него можно применить стандартную функцию hash(). То есть, мы подменили вычисление хэша объекта класса Point на вычисление хэша от координат точки. Теперь, после запуска программы видим, что объекты равны и их хэши также равны.

Что это в итоге означает? Смотрите, если взять пустой словарь:

d = {}

А, затем, сформировать записи через объекты p1 и p2:

d[p1] = 1

d[p2] = 2

print(d)

то они будут восприниматься как один и тот же ключ, так как объекты равны и их хэши тоже равны. А вот если магические методы в классе Point поставить в комментарии и снова запустить программу, то увидим, что это уже разные объекты, которые формируют разные ключи словаря. Вот для чего может понадобиться тонко настраивать работу функции hash() для объектов классов. И теперь вы знаете, как это делается.

## Магический метод \_\_bool\_\_ определения правдивости объектов

На этом занятии мы поговорим о способах настройки и определения правдивости объектов классов. Что такое правдивость? Это когда к экземпляру явно или неявно применяется функция bool(). С ней мы с вами уже знакомы и применяли к обычным типам данных:

bool(123)

bool(-1)

bool(0)

bool("python")

bool("")

bool([])

В стандартном поведении она возвращает True для непустых объектов и False – для пустых. Давайте посмотрим, что она будет выдавать для экземпляров классов. Я возьму класс из предыдущего занятия:

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Создадим его объект:

p = Point(3, 4)

и применим к нему функцию bool():

print(bool(p))

Увидим значение True. В действительности, эта функция всегда возвращает True для любых объектов пользовательского класса. Получается, что смысла в ней особого нет, применительно к экземплярам наших классов? Не совсем. Мы можем переопределить ее поведение либо через магический метод \_\_len\_\_(), либо через метод \_\_bool\_\_():

* \_\_len\_\_() – вызывается функцией bool(), если не определен магический метод \_\_bool\_\_();
* \_\_bool\_\_() – вызывается в приоритетном порядке функцией bool().

Вначале я пропишу магический метод \_\_len\_\_() в классе Point, следующим образом:

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_len\_\_(self):

        print("\_\_len\_\_")

        return self.x \* self.x + self.y \* self.y

В этом методе я вычисляю и возвращаю квадрат длины радиус-вектора с координатами (x; y). Запустим программу и видим значение True, а также сообщение «\_\_len\_\_». То есть, действительно был вызван метод \_\_len\_\_() и, так как он вернул не нулевое значение, то функция bool() интерпретировала его как True.

Давайте в экземпляре класса пропишем нулевые координаты, чтобы длина вектора была нулевой:

p = Point(0, 0)

Теперь видим ожидаемое значение False.

Конечно, если нам нужно явно описать алгоритм работы функции bool() применительно к нашим экземплярам класса, то следует использовать магический метод \_\_bool\_\_(). Я запишу его в таком виде:

    def \_\_bool\_\_(self):

        print("\_\_bool\_\_")

        return self.x == self.y

Теперь, объект будет считаться правдивым (истинным), если его координаты равны. Запускаем программу и видим, что для нулей отображается значение True. Если же прописать не равные координаты:

p = Point(10, 20)

то получаем значение False. Конечно, такая реализация магического метода \_\_bool\_\_() – это лишь учебный пример, чтобы вы поняли принцип его работы. В реальности, мы можем в этом методе прописывать любую логику. Единственное условие, чтобы данный метод возвращал булево значение True или False. Указывать в операторе return другие типы данных запрещено.

Все это хорошо, но где это используется? Чаще всего в условных конструкциях. Например, если прописать вот такое условие:

if p:

    print("объект p дает True")

else:

    print("объект p дает False")

Здесь происходит неявный вызов функции bool() при проверке условия. Поэтому в программах, где требуется описать собственные проверки истинности или ложности объектов, то пользуются или магическим методом \_\_len\_\_(), но чаще всего, магическим методом \_\_bool\_\_().

## Магические методы \_\_getitem\_\_, \_\_setitem\_\_ и \_\_delitem\_\_

Продолжаем знакомиться с магическими методами классов. Здесь речь пойдет о следующем их наборе:

* \_\_getitem\_\_(self, item) – получение значения по ключу item;
* \_\_setitem\_\_(self, key, value) – запись значения value по ключу key;
* \_\_delitem\_\_(self, key) – удаление элемента по ключу key.

Давайте разберемся для чего они нужны и как их можно использовать. Предположим, что мы создаем класс для представления студентов:

class Student:

    def \_\_init\_\_(self, name, marks):

        self.name = name

        self.marks = list(marks)

Его экземпляр можно сформировать, следующим образом:

s1 = Student('Сергей', [5, 5, 3, 2, 5])

В объекте s1 имеется локальное свойство marks со списком студентов. Мы можем к нему обратиться и выбрать любую оценку:

print(s1.marks[2])

Но что если мы хотим делать то же самое, но используя только ссылку на объект s1:

print(s1[2])

Если сейчас запустить программу, то увидим сообщение об ошибке, что наш класс (объект) не поддерживает такой синтаксис. Как вы, наверное, уже догадались, поправить это можно с помощью магического метода \_\_getitem\_\_. Запишем его в нашем классе, следующим образом:

    def \_\_getitem\_\_(self, item):

        return self.marks[item]

Теперь ошибок нет и на экране видим значение 3. Однако, если указать неверный индекс:

print(s1[20])

то получим исключение IndexError, которое сгенерировал список marks. При необходимости, мы можем сами контролировать эту ошибку, если в методе \_\_getitem\_\_ пропишем проверку:

    def \_\_getitem\_\_(self, item):

        if 0 <= item < len(self.marks):

            return self.marks[item]

        else:

            raise IndexError("Неверный индекс")

При запуске программы видим наше сообщение «Неверный индекс». Также можно сделать проверку на тип индекса:

print(s1['abc'])

для списков он должен быть целым числом. Поэтому дополнительно можно записать такую проверку:

    def \_\_getitem\_\_(self, item):

        if not isinstance(item, int):

            raise TypeError("Индекс должен быть целым числом")

       if 0 <= item < len(self.marks):

            return self.marks[item]

        else:

            raise IndexError("Неверный индекс")

То есть, здесь возможны самые разные вариации обработки и проверки исходных данных, прежде чем обратиться к списку marks и вернуть значение.

Теперь давайте предположим, что хотели бы иметь возможность менять оценки студентов, используя синтаксис:

s1[2] = 4

print(s1[2])

Сейчас, после запуска программы будет ошибка TypeError, что объект не поддерживает операцию присвоения, так как в классе не реализован метод \_\_setitem\_\_. Давайте добавим и его:

    def \_\_setitem\_\_(self, key, value):

        if not isinstance(key, int) or key < 0:

            raise TypeError("Индекс должен быть целым неотрицательным числом")

        self.marks[key] = value

Однако, если мы сейчас укажем несуществующий индекс:

s1[6] = 4

то операция присвоения новой оценки приведет к ошибке. Если предполагается использовать такую возможность, то реализовать ее можно, следующим образом:

    def \_\_setitem\_\_(self, key, value):

        if not isinstance(key, int) or key < 0:

            raise TypeError("Индекс должен быть целым неотрицательным числом")

        if key >= len(self.marks):

            off = key + 1 - en(self.marks)

            self.marks.extend([None]\*off)

        self.marks[key] = value

Если индекс превышает размер списка, то мы расширяем список значениями None до нужной длины (с помощью метода extend), а затем, в последний элемент записываем переданное значение value. Теперь, при выполнении команд:

s1[10] = 4

print(s1.marks)

Увидим список:

[5, 5, 3, 2, 5, None, None, None, None, None, 4]

То есть, он был расширен до 10 элементов и последним элементом записано 4. И так можно прописывать любую нужную нам логику при записи новых значений в список marks.

Наконец, последний третий магический метод \_\_delitem\_\_ вызывается при удалении элемента из списка. Если сейчас записать команду:

del s1[2]

то в консоли увидим сообщение: «AttributeError: \_\_delitem\_\_». Здесь явно указывается, что при удалении вызывается метод \_\_delitem\_\_. Добавим его в наш класс:

    def \_\_delitem\_\_(self, key):

        if not isinstance(key, int):

            raise TypeError("Индекс должен быть целым числом")

        del self.marks[key]

Теперь оценки успешно удаляются, если указан верный индекс.

Вот общие возможности данных магических методов. Теперь, если в программе вам потребуется реализовать подобную логику поведения экземпляров классов, я думаю, вы легко со всем справитесь.

## Магические методы \_\_iter\_\_ и \_\_next\_\_

* \_\_iter\_\_(self) – получение итератора для перебора объекта;
* \_\_next\_\_(self) – переход к следующему значению и его считывание.

Давайте разберемся для чего они нужны и как их можно использовать. Вы все знаете, как работает функция range(). Она выдает значения арифметической прогрессии, например:

list(range(5))

дает последовательность целых чисел от 0 до 4. Перебрать значения объекта range также можно через итератор:

a = iter(range(5))

next(a)

next(a)

…

В конце генерируется исключение StopIteration. Так вот, мы можем создать подобный объект, используя магические методы \_\_iter\_\_ и \_\_next\_\_. Давайте это сделаем для объекта frange, который будет выдавать последовательность вещественных чисел арифметической прогрессии. Для этого я объявлю класс:

class FRange:

    def \_\_init\_\_(self, start=0.0, stop=0.0, step=1.0):

        self.start = start

        self.stop = stop

        self.step = step

        self.value = self.start - self.step

Здесь в инициализатор мы передаем начальное значение прогрессии, конечное и шаг изменения. Также формируем свойство value, которое будет представлять собой текущее значение для считывания.

Для перебора элементов добавим в этот класс метод, который будет соответствовать магическому методу \_\_next\_\_:

    def \_\_next\_\_(self):

        if self.value + self.step < self.stop:

            self.value += self.step

            return self.value

        else:

            raise StopIteration

В этом методе мы увеличиваем значение value на шаг step и возвращаем до тех пор, пока не достигли значения stop (не включая его). При достижении конца генерируем исключение StopIteration, ровно так, как это делает объект range.

Сформируем объект этого класса:

fr = FRange(0, 2, 0.5)

и четыре раза вызовем метод \_\_next\_\_()

print(fr.\_\_next\_\_())

print(fr.\_\_next\_\_())

print(fr.\_\_next\_\_())

print(fr.\_\_next\_\_())

Видим четыре значения нашей арифметической прогрессии. Если вызвать \_\_next\_\_() еще раз:

print(fr.\_\_next\_\_())

получим исключение StopIteration. В целом получился неплохой учебный пример. В действительности, благодаря определению магического метода \_\_next\_\_ в классе FRange, мы можем применять функцию next() для перебора значений его объектов:

fr = FRange(0, 2, 0.5)

print(next(fr))

print(next(fr))

print(next(fr))

print(next(fr))

Здесь функция next() вызывает метод \_\_next\_\_ и возвращенное им значение, возвращается функцией next(). При этом, в качестве аргумента мы ей передаем экземпляр самого класса. То есть, объект класса выступает в роли итератора. В нашем случае так и задумывалось. Однако, перебрать объект fr с помощью цикла for не получится:

for x in fr:

    print(x)

Появится ошибка, что объект не итерируемый. Почему? Ведь мы прописали поведение функции next()? Этого не достаточно. Необходимо еще, чтобы объект возвращал итератор при вызове функции iter:

it = iter(fr)

Для этого в классе нужно прописать еще один магический метод \_\_iter\_\_. В нашем примере он будет выглядеть, так:

    def \_\_iter\_\_(self):

        self.value = self.start - self.step

        return self

Мы здесь устанавливаем начальное значение value и возвращаем ссылку на объекта класса, так как этот объект в нашем примере и есть итератор – через него вызывается магический метод \_\_next\_\_.

Теперь, после запуска программы у нас не возникает никаких ошибок и цикл for перебирает значения объекта fr. То же самое мы можем сделать и через next():

fr = FRange(0, 2, 0.5)

it = iter(fr)

print(next(it))

print(next(it))

print(next(it))

print(next(it))

Как вы помните, цикл for именно так и перебирает итерируемые объекты, сначала неявно вызывает функцию iter(), а затем, на каждой итерации – функцию next(), пока не возникнет исключение StopIteration. Кроме того, благодаря магическому методу \_\_iter\_\_ мы теперь можем обходить значения объекта fr много раз с самого начала, например:

it = iter(fr)

print(next(it))

print(next(it))

print(next(it))

print(next(it))

it = iter(fr)

print(next(it))

print(next(it))

print(next(it))

print(next(it))

Таким образом, сформировали класс FRange, который воспринимается как итерируемый объект с возможностью перебора функцией next() или циклом for.

В заключение этого занятия я приведу пример еще одного класса FRange2D для формирования таблиц значений:

class FRange2D:

    def \_\_init\_\_(self, start=0.0, stop=0.0, step=1.0, rows=5):

        self.fr = FRange(start, stop, step)

        self.rows = rows

Здесь в инициализаторе создается одномерный объект FRange, который будет формировать строки таблицы. Параметр rows – число строк. Далее, пропишем два магических метода \_\_iter\_\_ и \_\_next\_\_, следующим образом:

    def \_\_iter\_\_(self):

        self.value\_row = 0

        return self

    def \_\_next\_\_(self):

        if self.value\_row < self.rows:

            self.value\_row += 1

            return iter(self.fr)

        else:

            raise StopIteration

Обратите внимание, что метод \_\_next\_\_ возвращает не конкретное значение, а итератор на объект класса FRange. Сейчас вы поймете почему так. Создадим объект класса FRange2D:

fr = FRange2D(0, 2, 0.5, 4)

и для перебора его значений нам понадобятся два цикла for:

for row in fr:

    for x in row:

        print(x, end=" ")

    print()

Первый цикл перебирает первый итератор – объект класса FRange2D и на каждой итерации возвращает итератор объекта класса FRange. Именно поэтому мы в методе \_\_next\_\_ класса FRange2D возвращаем иетратор, иначе бы не смогли перебирать объект row во вложенном цикле for.

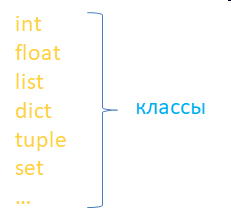
После запуска программы увидим на экране следующую таблицу чисел:

0.0 0.5 1.0 1.5  
0.0 0.5 1.0 1.5  
0.0 0.5 1.0 1.5  
0.0 0.5 1.0 1.5

Вот общий принцип создания итерируемых объектов. Надеюсь, эти примеры вам были понятны и вы теперь знаете, как и для чего используются магические методы \_\_iter\_\_ и \_\_next\_\_.

## Наследование от встроенных типов данных

Интересный факт языка Python, что все стандартные типы данных являются классами:



Мы в этом легко можем убедиться, если выполним для них функцию issubclass():

issubclass(int, object)

issubclass(list, object)

Всюду увидим True. А мы знаем, что эта функция работает исключительно с классами, а не объектами, поэтому данный факт подтверждает, что эти типы являются классами языка Python.

Раз это так, то что нам мешает наследоваться от них и расширять функционал по мере необходимости? Ничего, поэтому можно сделать, например, такую реализацию:

class Vector(list):

    def \_\_str\_\_(self):

        return " ".join(map(str, self))

v = Vector([1, 2, 3])

print(v)

Мы здесь переопределили магический метод \_\_str\_\_ для вывода списка в виде набора данных через пробел. Мало того, теперь тип данных нашего списка стал не list, а Vector:

print(type(v))

увидим:

<class '\_\_main\_\_.Vector'>

Конечно, стандартные типы данных редко расширяют с помощью пользовательских классов, но понимать, что они представляют собой классы и что такая возможность в принципе существует, важно.

Итак, из этого материала вы должны были узнать, что все классы по умолчанию наследуются от базового класса object, как работает функция issubclass() и что из себя представляют встроенные типы данных языка Python.

## Наследование. Атрибуты private и protected

Мы продолжаем изучение темы «наследование». На этом занятии мы увидим, как влияет режим доступа private и protected атрибутов при наследовании классов.

Ранее мы с вами об этом уже говорили и в частности отмечали, что:

* \_attribute (с одним подчеркиванием) – режим доступа protected (служит для обращения внутри класса и во всех его дочерних классах)
* \_\_attribute (с двумя подчеркиваниями) – режим доступа private (служит для обращения только внутри класса).

Давайте посмотрим, как ведут себя атрибуты с этими режимами доступа при наследовании. Возьмем пример из предыдущего занятия с двумя классами:

class Geom:

    name = 'Geom'

    def \_\_init\_\_(self, x1, y1, x2, y2):

        print(f"инициализатор Geom для {self.\_\_class\_\_}")

        self.\_\_x1 = x1

        self.\_\_y1 = y1

        self.\_\_x2 = x2

        self.\_\_y2 = y2

class Rect(Geom):

    def \_\_init\_\_(self, x1, y1, x2, y2, fill='red'):

        super().\_\_init\_\_(x1, y1, x2, y2)

        self.\_\_fill = fill

Здесь мы пытаемся в инициализаторе базового класса Geom сформировать приватные локальные свойства с координатами прямоугольника. Дополнительно в инициализаторе самого класса создается приватное свойство \_\_fill.

Ниже создадим объект класса Rect:

r = Rect(0, 0, 10, 20)

и выведем все его локальные атрибуты в консоль:

print(r.\_\_dict\_\_)

После запуска программы увидим следующие строчки:

инициализатор Geom для <class '\_\_main\_\_.Rect'>  
{'\_Geom\_\_x1': 0, '\_Geom\_\_y1': 0, '\_Geom\_\_x2': 10, '\_Geom\_\_y2': 20, '\_Rect\_\_fill': 'red'}

Смотрите, локальные свойства с координатами имеют префикс \_Geom, то есть, префикс того класса, в котором они непосредственно были прописаны.

Несмотря на то, что параметр self является ссылкой на объект класса Rect. Это особенность поведения (формирования) приватных атрибутов в базовых классах. У них всегда добавляется префикс именно базового класса, а не класса объекта self. А вот последнее свойство \_\_fill имеет ожидаемый префикс \_Rect, так как оно было создано в классе Rect.

Что из этого следует? Во-первых, мы, конечно же, не можем обратиться в свойствам-координатам в дочернем классе Rect. Если в нем прописать метод get\_coords():

    def get\_coords(self):

        return (self.\_\_x1, self.\_\_y1, self.\_\_x2, self.\_\_y2)

а, затем, вызвать через объект класса Rect:

r.get\_coords()

то увидим ошибку AttributeError. Но если перенести этот метод в базовый класс Geom, то все сработает без ошибок, так как приватным свойствам будет добавлен правильный префикс \_Geom.

Возможно, вам кажется это немного запутанным? Но давайте вспомним, а для чего вообще нужны и когда используются приватные атрибуты.

Мы говорили, что это закрытые от внешнего вмешательства свойства или методы текущего класса, доступные только внутри этого класса и недоступные из других, в том числе и из дочерних классов.

Именно поэтому приватные атрибуты жестко привязываются к текущему классу, в котором они создаются, так как по логике предполагается их использовать только внутри этого класса и больше нигде.

Если же нам нужно определить закрытые атрибуты, доступные в текущем классе и во всех его дочерних классах, то для этого следует использовать метод определения protected – одно нижнее подчеркивание. Поэтому правильнее было бы создавать свойства-координаты в базовом инициализаторе в режиме protected:

class Geom:

    name = 'Geom'

    def \_\_init\_\_(self, x1, y1, x2, y2):

        print(f"инициализатор Geom для {self.\_\_class\_\_}")

        self.\_x1 = x1

        self.\_y1 = y1

        self.\_x2 = x2

        self.\_y2 = y2

class Rect(Geom):

    def \_\_init\_\_(self, x1, y1, x2, y2, fill='red'):

        super().\_\_init\_\_(x1, y1, x2, y2)

        self.\_fill = fill

    def get\_coords(self):

        return (self.\_x1, self.\_y1, self.\_x2, self.\_y2)

Тогда никаких проблем с доступом уже не возникает:

r = Rect(0, 0, 10, 20)

print(r.\_\_dict\_\_)

r.get\_coords()

После запуска программы увидим следующие строчки:

инициализатор Geom для <class '\_\_main\_\_.Rect'>  
{'\_x1': 0, '\_y1': 0, '\_x2': 10, '\_y2': 20, '\_fill': 'red'}

Опять же, как я ранее отмечал, режим доступа protected в реальности никак не ограничивает доступ к атрибутам объектов класса или самого класса. Например, мы можем обратиться к координатам напрямую через экземпляр класса:

print(r.\_x1)

Никаких ошибок не будет. Нижнее подчеркивание лишь предупреждает (сигнализирует) программиста о защищенном атрибуте, к которому напрямую лучше не обращаться. Этот атрибут был создан для внутренней логики работы алгоритма в классе и не предназначен для обращения извне. В дальнейшем это может привести к проблемам, например, при изменении версии класса, в котором такого атрибута уже не будет, или он будет играть другую роль и т.п.

## Атрибуты private и protected на уровне класса

Все также работает и с атрибутами уровня класса. Например, сейчас мы совершенно спокойно можем обратиться к свойству name класса Geom через объект класса Rect:

print(r.name)

Добавив одно нижнее подчеркивание, функционал останется прежним, мы лишь отметим, что к этой переменной извне лучше не обращаться:

print(r.\_name)

Но, если прописать два подчеркивания, то доступ будет закрыт всюду, кроме самого класса Geom:

print(r.\_\_name)

или так:

class Rect(Geom):

    def \_\_init\_\_(self, x1, y1, x2, y2, fill='red'):

        super().\_\_init\_\_(x1, y1, x2, y2)

        self.\_fill = fill

        self.\_name = self.\_\_name

Но в Geom мы можем к ней обращаться:

class Geom:

    \_\_name = 'Geom'

    def \_\_init\_\_(self, x1, y1, x2, y2):

        print(f"инициализатор {self.\_\_name}")

        self.\_x1 = x1

        self.\_y1 = y1

        self.\_x2 = x2

        self.\_y2 = y2

Те же ограничения доступа можно накладывать и на методы. Если в базовом классе Geom определить приватный метод, например, для проверки корректности значений координат:

class Geom:

    ...

    def \_\_verify\_coord(self, coord):

        return 0 <= coord <= 100

то он будет доступен только внутри этого класса и вызвать его, скажем, в дочернем классе Rect уже не получится:

class Rect(Geom):

    def \_\_init\_\_(self, x1, y1, x2, y2, fill='red'):

        super().\_\_init\_\_(x1, y1, x2, y2)

        super().\_\_verify\_coord(x1)

        self.\_fill = fill

Кстати, этот пример также показывает, что приватность запрещает переопределение методов в дочерних классах. Если же у метода прописать только одно подчеркивание, то его можно будет вызывать во всех дочерних классах.

Я, надеюсь, вы теперь знаете, как работают режимы доступа атрибутов при наследовании классов и сможете их грамотно применять в своих программах.

## Принцип утиной типизации

В новых версиях Python аннотации типов получают всё большую поддержку, всё чаще и чаще используются в библиотеках, фреймворках, и проектах на Python. Помимо дополнительной документированности кода, аннотации типов позволяют таким инструментам, как mypy, статически произвести дополнительные проверки корректности программы и выявить возможные ошибки в коде. В этой статье пойдет речь об одной, как мне кажется, интересной теме, касающейся статической проверки типов в Python – протоколах, или как сказано в PEP-544, статической утиной типизации.

Часто, когда речь заходит о Python, всплывает фраза *утиная типизация*, или даже что-нибудь вроде:

Если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка

[Утиная типизация](https://docs.python.org/3/glossary.html#term-duck-typing) – это концепция, характерная для языков программирования с динамической типизацией, согласно которой конкретный тип или класс объекта не важен, а важны лишь свойства и методы, которыми этот объект обладает. Другими словами, при работе с объектом его тип не проверяется, вместо этого проверяются свойства и методы этого объекта. Такой подход добавляет гибкости коду, позволяет полиморфно работать с объектами, которые никак не связаны друг с другом и могут быть объектами разных классов. Единственное условие, чтобы все эти объекты поддерживали необходимый набор свойств и методов.

>>> class Meter:

... def \_\_len\_\_(self):

... return 1\_000

...

>>> len([1, 2, 3])

3

>>> len("Duck typing...")

14

>>> len(Meter())

1000

В примере выше функции len не важен тип аргумента, а важно лишь то, что у объекта можно вызвать метод \_\_len\_\_().

Но именно эта гибкость и усложняет раннее обнаружение ошибок типизации. Корректность использования объектов определяется динамически, в момент выполнения программы, и зачастую тестирование – единственный способ отловить подобные ошибки. Статическая проверка типов и корректности программы в данном случае представляет значительную сложность.