В центре ООП находится понятие объекта.

* **Объект** — это сущность, экземпляр класса, которая имеет свои атрибуты и методы, которая может на реагировать на события, используя свои данные.
* **Ме́тод** — это функция или процедура, принадлежащая какому-то классу или объекту. **Методы** - вещи, которые может сделать объект; «глаголы» объектов.
* **Атрибуты** - вещи, которые описывают объект; «прилагательные» объектов.
* **Событие** — это сообщение, которое возникает в различных точках исполняемого кода при выполнении определённых условий. События предназначены для того, чтобы иметь возможность предусмотреть реакцию программного обеспечения. **Событие** - силы, внешние по отношению к объекту, на которые этот объект может реагировать.

В объектно-ориентированном программировании (ООП) – **КЛАСС** - это основной элемент, в рамках которого осуществляется конструирование программ. Класс содержит в себе данные и код, который управляет этими данными.

Класс зачастую описывает объект реального мира. Как и реальный объект, класс содержит свой набор параметров и характеристик. Каждый такой параметр называется поле класса (очень похоже на обычные переменные). Также класс способен манипулировать своими характеристиками (полями) с помощью методов класса (похожи на функции в процедурных языках).

* **Класс** - обеспечивает способ создания новых объектов на основе «метаопределения» объекта.
* **Инкапсуляция - м**еханизм объединения данных (переменных) и кода, действующего на данные (методы), в единое целое (класс).
* **Наследование -** форма повторного использования программного обеспечения, в которой вы создаете класс, который поглощает данные и поведение существующего класса и дополняет их новыми возможностями.
* **Полиморфизм** - способность метода делать разные вещи на основе объекта, на который он действует (полиморфизм позволяет вам определять один интерфейс и иметь несколько реализаций) 2+2 //// ‘he’+’llo’.
* Класс можно определить как шаблон / план, который описывает поведение / состояние, поддерживаемое объектом его типа
* Определение класса: class ClassName [(суперкласс)]:[атрибуты (данные и методы)]
* Определения классов, как и определения функций (инструкции def), должны быть описаны до того, как они будут вызваны.
* Когда вводится определение класса, создается новое пространство имен и используется в качестве локальной области - таким образом, все назначения локальных переменных попадают в это новое пространство имен.
* Когда определение класса завершено, создается объект класса.
* Класс создает новое пространство имен

**Объекты**

• Объекты - это экземпляры класса (объекты имеют состояния и поведение)

• Все в Python является объектом

• Создание экземпляра класса использует нотацию функции:

x = ClassName () # создает новый экземпляр класса и назначает этот объект локальной переменной x.

• Атрибуты вызова:

* object.attribute
* object.method ()

• Все объекты экземпляра содержат:

* уникальный идентификатор (целое число, возвращаемое id (x))
* тип (возвращаемый type (x))

• Python автоматически удаляет ненужные объекты, чтобы освободить пространство памяти. Процесс, с помощью которого Python периодически восстанавливает блоки памяти, которые больше не используются, называется «Сбор мусора»

**Конструктор класса**

Конструктор класса позволяет задать определенные параметры объекта при его создании. Конструктор класса вызывается автоматически, при создании объекта. Таким образом появляется возможность создавать объекты с уже заранее заданными атрибутами. Конструктором класса является метод:

\_\_init\_\_(self)

Первым аргументом (в скобках после имени метода) всегда должен идти “пустой” аргумент, который общепринято называть “self“.

Он служит для связи экземпляра (объекта) класса, который вызывает этот метод, с самим классом.

Например, для того, чтобы иметь возможность задать цвет, длину и ширину прямоугольника при его создании, добавим к классу Rectangle следующий конструктор:

class Rectangle:

def \_\_init\_\_(self, color="green", width=100, height=100):

self.color = color

self.width = width

self.height = height

def square(self):

return self.width \* self.height

rect1 = Rectangle()

print(rect1.color)

print(rect1.square())

rect1 = Rectangle("yellow", 23, 34)

print(rect1.color)

print(rect1.square())

**Инкапсуляция** в Python работает лишь на уровне соглашения между программистами о том, какие атрибуты являются общедоступными, а какие — внутренними. В Python отсутствуют модификаторы доступа к атрибутам, такие как **public, protected или private**. Все атрибуты являются общедоступными. Но существует соглашение для определения защищенных и частных интерфейсов.

Одиночное подчеркивание в начале имени атрибута говорит о том, что переменная или метод не предназначен для использования вне методов класса, однако атрибут доступен по этому имени.

**class** **A**:

**def** \_private(self):

print("Это приватный метод!")

>>> a = A()

>>> a.\_private()

Это приватный метод!

Двойное подчеркивание в начале имени атрибута даёт большую защиту: атрибут становится недоступным по этому имени.

**>>> class** **B**:

**...**  **def** \_\_private(self):

**...**  **print**("Это приватный метод!")

**...**

**>>>** b = B()

**>>>** b.\_\_private()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'B' object has no attribute '\_\_private'

Однако полностью это не защищает, так как атрибут всё равно остаётся доступным под именем \_ИмяКласса\_\_ИмяАтрибута:

**>>>** b.\_B\_\_private()

Это приватный метод!

**self** - ссылка на экземпляр объекта

• self используется для доступа к членам экземпляра (атрибуты, методы).

• Мы могли бы выбрать любое другое имя, но self - это конвенция

self - это ни в коем случае не зарезервированное слово. Это просто название переменной.

В методах класса первый параметр функции по соглашению именуют self, и это ссылка на сам объект этого класса. Но это именно соглашение. Вы вольны называть параметры как угодно.

1. В чем будет отличие, если мы вместо

class Student :

def f (self, n, y) :

self.name=n

self.year=y

print(self.name, "is on the", self.year, "-th year")

напишем

class Student :

def f (self, n, y) :

name=n

year=y

print(name, "is on the", year, "-th year")

В теле методов нет неявного присваивания атрибутам объекта.

self.name = 1 — присваивает значение 1 атрибуту объекта.

name = 1 — присваивает значение 1 локальной переменной (даже если присваивание происходит в методе класса).

2. Почему если мы уберем self из описания класса

class Student :

def f (n,y) :

name = n

year = y

print(name, "is on the", year, "-th year")

s = S()

то при вызове s.f("Vasya", 5) получим ошибку

TypeError: f() takes 2 positional arguments but 3 were given

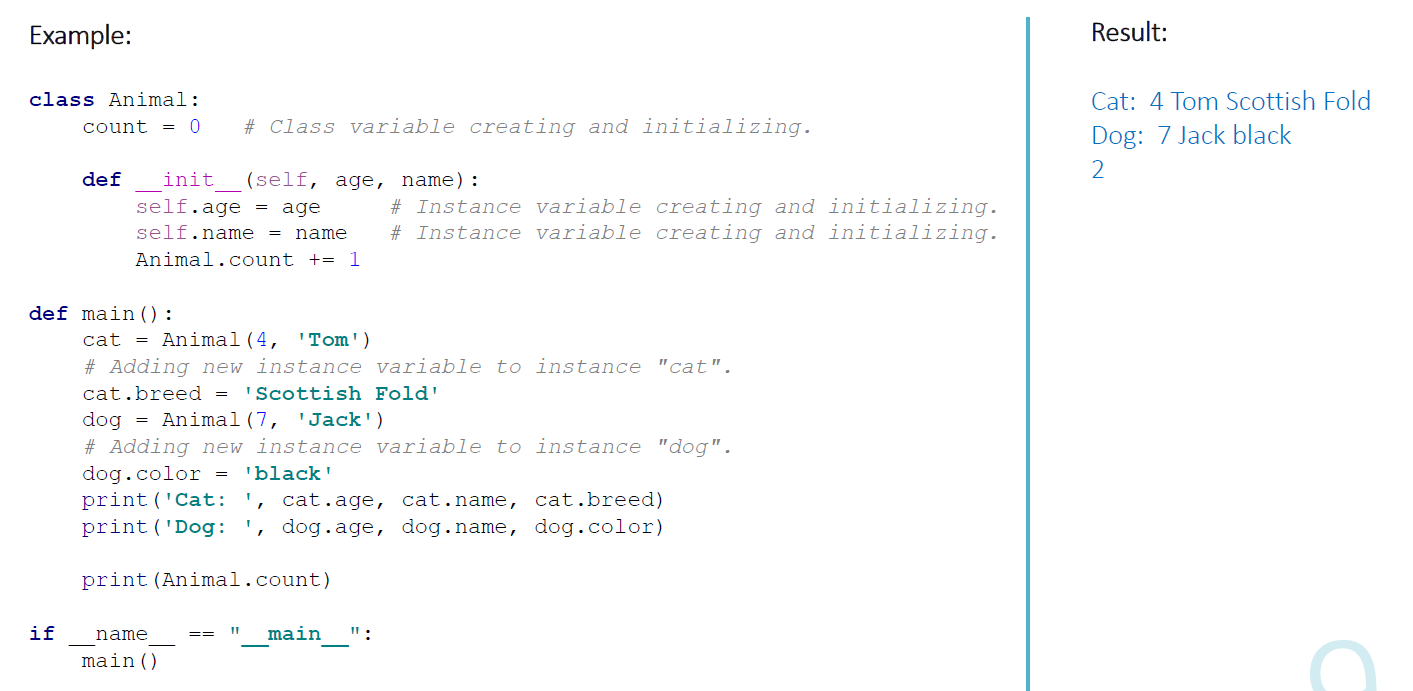
Какой третий параметр был передан?

Параметр self подставляется автоматически, когда метод вызывается для объекта. Поэтому, при вызjве метода с двумя переменными, ему на самом деле передаётся три (и он должен уметь принимать три аргумента).

3. Когда стоит добавлять self в описание метода класса, а когда нет?

Когда объявляете метод, который будет вызываться для экземпляра класса.

**Переменные класса (class variable) и переменные экземпляра (instance variable)**



**Методы класса и экземпляра**

Метод - это просто объект функции, созданный командой def

Метод работает точно так же, как простая функция

**Методы экземпляра**

* метод экземпляра знает свой экземпляр
* def name (self, arg1, arg2, ...):

suite

* Метод экземпляра получает экземпляр как неявный первый аргумент (self)

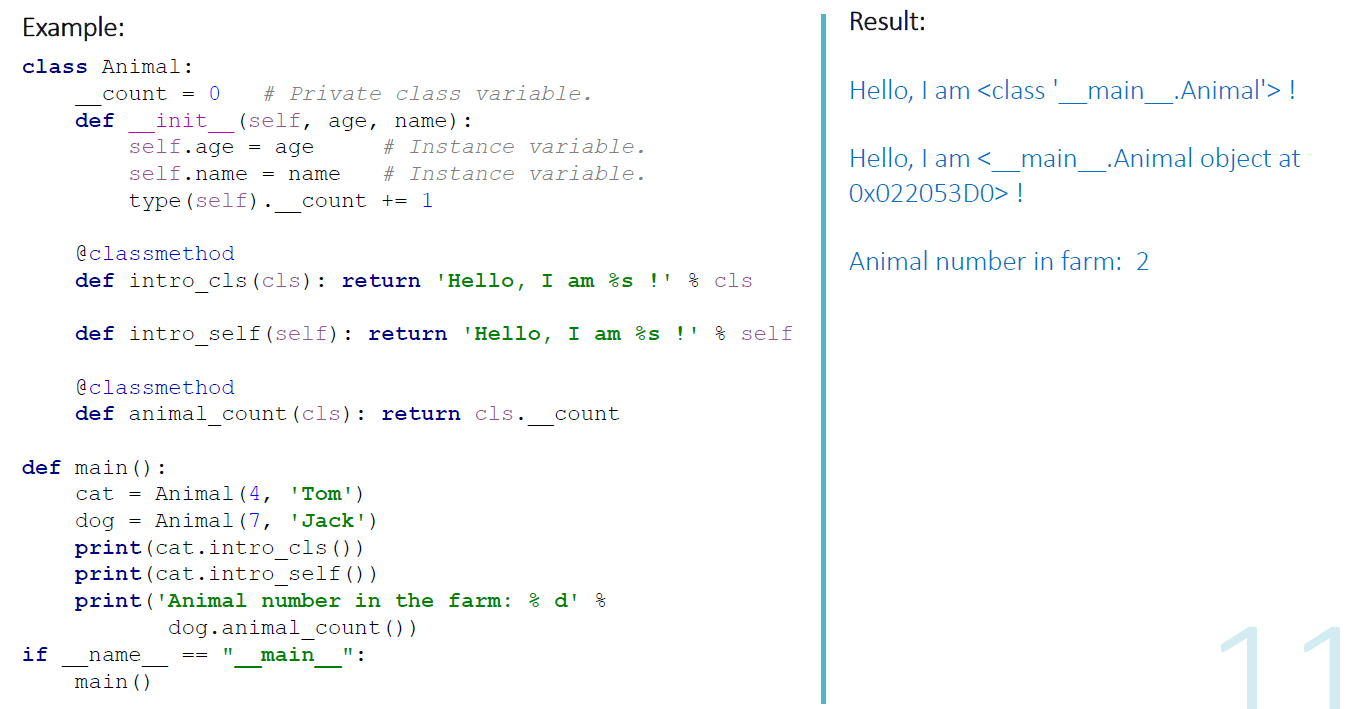
**Методы класса**

* метод класса знает свой класс
* @classmethod

def f (cls, arg1, arg2, ...):

suite

* метод класса получает класс как неявный первый аргумент (cls)



Поведение по умолчанию для доступа к атрибутам - это получение, установка или удаление атрибута из словаря объекта

**Дескриптор** — это атрибут объекта со связанным поведением (англ. binding behavior), т.е. такой, чьё поведение при доступе переопределяется методами протокола дескриптора. Эти методы: \_\_get\_\_, \_\_set\_\_ и \_\_delete\_\_. Если хотя бы один из этих методов определён для объекта, то он становится дескриптором.

**• протокол дескриптора:**

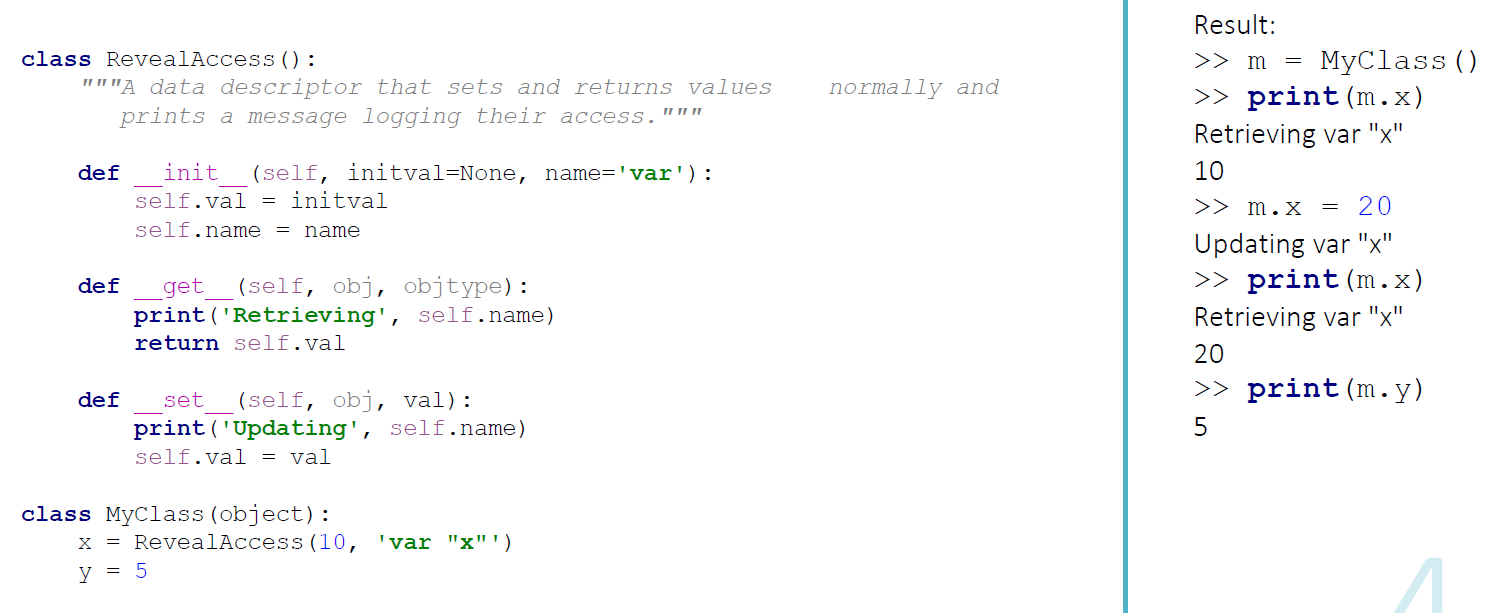
descr.\_\_get\_\_(self, obj, type=None) --> value

descr.\_\_set\_\_(self, obj, value) --> None

descr.\_\_delete\_\_(self, obj) --> None

Собственно это всё. Определите любой из этих методов и объект будет считаться дескриптором, и сможет переопределять стандартное поведение, если его будут искать как атрибут.

Следующий код создаёт класс, чьи объекты являются дескрипторами данных и всё, что они делают — это печатают сообщение на каждый вызов get или set.



Если убрать протоколы дескриптора, то я получу:

<\_\_main\_\_.A object at 0x0341CF70>

**Свойство**

Свойство представляет собой особый тип члена класса, промежуточный по функциональности между атрибутом и методом.

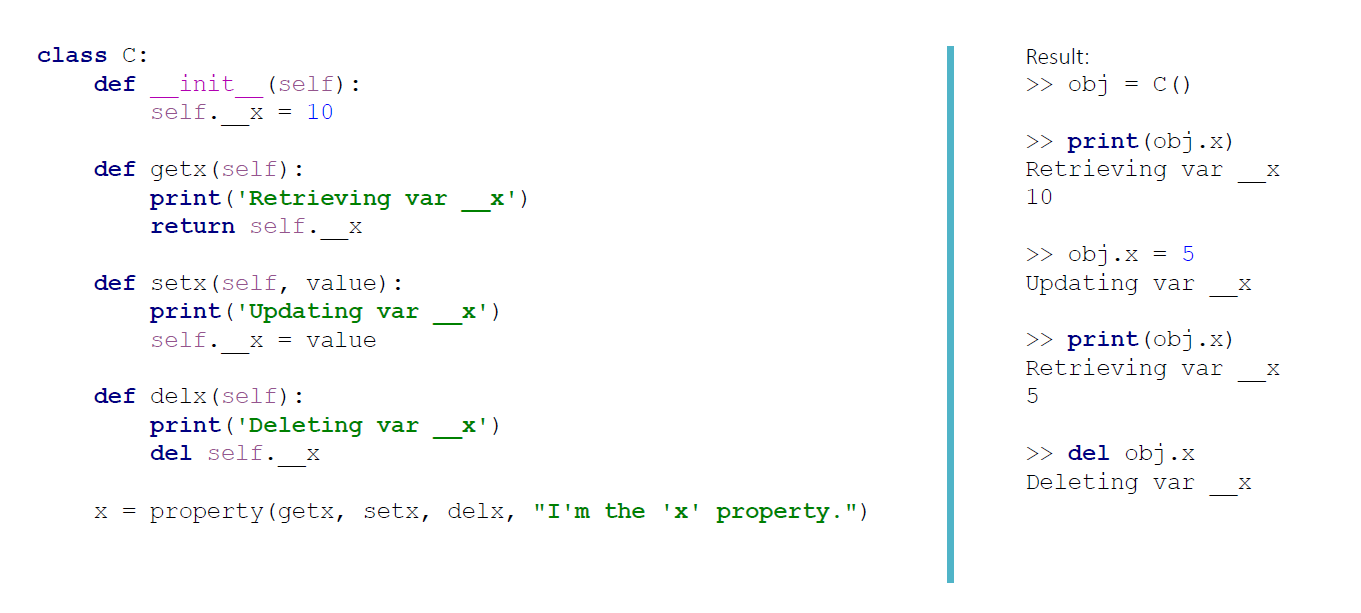
Доступ к защищенным / закрытым атрибутам достигается за счет атрибутов **property**

class property(fget=None, fset=None, fdel=None, doc=None) →

возвращает свойства атрибутов

* fget - метод для получения значения атрибута.
* fset - это функция для установки значения атрибута.
* fdel - это функция для удаления значения атрибута.
* doc создает docstring для атрибута.

Вызова property() достаточно, чтобы создать дескриптор данных, который вызывает нужные функции во время доступа к атрибуту.



• **Decorator** - это функция, возвращающая другую функцию, обычно применяемую как преобразование функции с использованием синтаксиса @ wrapper

Помните, что синтаксис @decorator - это просто синтаксический сахар; синтаксис:

@property

def foo(self): return self.\_foo

действительно означает то же самое, что

def foo(self): return self.\_foo

foo = property(foo)

Наследование - это форма повторного использования программного обеспечения, в которой вы создаете класс, который поглощает данные и поведение существующего класса и улучшает их с помощью новых возможностей. Существующий класс называется базовым классом (суперкласс), а новый класс называется производным классом (подклассом)

**Типы наследования**

• Одиночное наследование - где подклассы наследуют функции одного суперкласса

• Множественное наследование - где один класс может иметь более одного суперкласса и наследовать функции от всех родительских классов

• Многоуровневое наследование - где подкласс унаследован от другого подкласса

**Синтаксис одиночного наследования Python**

class DerivedClassName(BaseClassName):

<statement-1>

.

.

.

<statement-N>

**Синтаксис множественного наследования Python**

class DerivedClassName (Base1, Base2, Base3):

<statement-1>

,

,

,

< statement-N>

**super ([type [, object-or-type]]) -** возвращает прокси-объект, который делегирует вызовы метода родительскому или родному классу типа

(полезно для доступа к унаследованным методам, которые были переопределены в классе)

* Типичный вызов суперкласса выглядит следующим образом:

class B(A):

def method(self, arg):

super( ).method(arg)

