Магические методы: введение

свойства, декораторы, дескрипторы

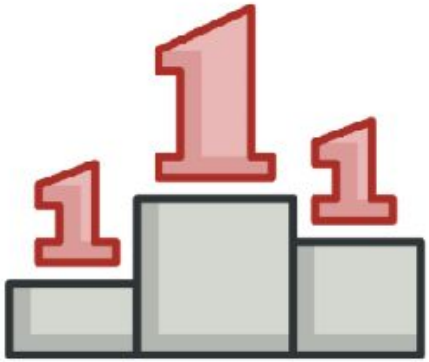
Какие уже знаем

__init__ (self) – инициализатор объекта класса

__new__(self) — ???

__del__(self) – финализатор класса

__dict__ — атрибуты экземпляра/класса



Одиночка

Singleton

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Пример для паттерна Singleton (Одиночка)

```
class DataBase:
    def __init__(self, user, psw, port):
        self.user = user
        self.psw = psw
        self.port = port

    def connect(self):
        print(f"соединение с БД: {self.user}, {self.psw},
{self.port}")

    def close(self):
        print("закрытие соединения с БД")

    def read(self):
        return "данные из БД"

    def write(self, data):
        print(f"запись в БД {data}")
```

Нужен класс для работы с БД, через который можно будет подключаться к СУБД, читать и записывать информацию, закрывать соединение. В программе должен существовать только один экземпляр этого класса в каждый момент ее работы. То есть, одновременно два объекта класса DataBase быть не должно => паттерн Singleton.

Реализация

Специальный атрибут:

__instance = None

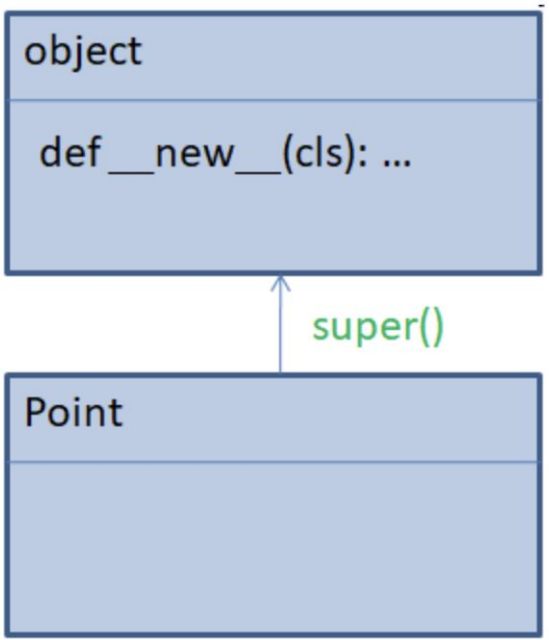
который будет хранить ссылку на экземпляр этого класса. Если экземпляра нет, то атрибут будет принимать значение None.

Чтобы гарантировать создание строго одного экземпляра, добавим в класс магический метод __new__:

```
def __new__(cls, *args, **kwargs):
    if cls.__instance is None:
        cls.__instance = super().__new__(cls)
    return cls.__instance
```

А что за super?

Начиная с версии Python 3, все классы автоматически и неявно наследуются от базового класса object:



Работает этот метод __new__ очевидным образом. Мы проверяем атрибут класса __instance. Причем для обращения к нему используем параметр cls – ссылку на текущий класс. Далее проверяем, если значение равно None, то вызываем метод __new__ базового класса и тем самым разрешаем создание объекта. Иначе просто возвращаем ссылку на ранее созданный экземпляр.

И пропишем еще один магический метод – финализатор __del__, который будет обнулять атрибут __instance перед уничтожением объекта, чтобы мы могли, при необходимости, создать новый.

Промежуточный вариант Singleton

```
class DataBase:
    __instance = None

    def __new__(cls, *args, **kwargs):
        if cls.__instance is None:
            cls.__instance = super().__new__(cls)
        return cls.__instance

    def __init__(self, user, psw, port):
        self.user = user
        self.psw = psw
        self.port = port

    def connect(self):
        print(f"соединение с БД: {self.user}, {self.psw}, {self.port}")

    def close(self):
        print("закрытие соединения с БД")

    def read(self):
        return "данные из БД"

    def write(self, data):
        print(f"запись в БД {data}")

    def __del__(cls, *args, **kwargs):
        cls.__instance = None
        return cls.__instance
```

Почему промежуточный?

Если попробовать создать два экземпляра:

db = DataBase('root', '1234', 80)

db2 = DataBase('root2', '5678', 40)

print(id(db), id(db2))

то их id ожидаемо будут равны. То есть, ссылки db и db2 действительно ведут на один объект. Но если выполнить метод:

db.connect()

db2.connect()

то увидим значения: 'root2', '5678', 40 – аргументы при повторном создании класса. По идее, если объект не создается, то и локальные свойства его также не должны меняться.

Почему так произошло? Мы здесь на самом деле видим первый объект. Но при повторном вызове DataBase() также был вызван магический метод __init__ с новым набором аргументов и локальные свойства изменили свое значение. Конечно, мы можем здесь поставить «костыль» и дополнительно в классе прописать флаговый атрибут, например:

__is_exist = False

специально для метода __init__, чтобы не выполнять его если объект уже создан.

Но а) костыли это плохо, б) есть для этого и специальный магический метод — __call__

Итог

__new__ — должен возвращать адрес нового созданного объекта, вызывается непосредственно перед созданием объекта класса

__init__ вызывается после создания объекта

Почему 2? В практике программирования встречаются самые разнообразные задачи и иногда нужно что-то делать и до создания объектов.

Классовые и статические методы

Статические @staticmethod — метод не требует доступа к состоянию класса или экземпляра и служит просто для выполнения некоторой логики.

Методы класса @classmethod:

- Первым параметром идет cls – ссылка на класс, а не self – ссылка на объект класса. Это означает, что данный метод может обращаться только к атрибутам текущего класса, но не к локальным свойствам его экземпляров.
- Можно напрямую вызывать из класса, не передавая ссылку на экземпляр, как это было при вызове обычных методов через класс. Но «платой» за это является ограниченность метода: он может работать только с атрибутами класса, но не объекта, что, в общем то, естественно, так как у него изначально нет ссылки на объект. Во всем остальном этот метод работает абсолютно также, как и любой другой метод, объявленный в классе.
- Может использоваться для создания альтернативных конструкторов или для работы с атрибутами класса, которые могут быть изменены при наследовании — нужно взаимодействовать с атрибутами класса или метод должен быть переопределяемым в дочерних классах.

Декоратор @property

```
class Person:
    def __init__(self, name, old):
        self.__name = name
        self.__old = old


def get_old(self):
        return self.__old


    def set_old(self, old):
        self.__old = old
```

```
p = Person('Сергей', 20)
p.set_old(35)
print(p.get_old())
…
old = property(get_old, set_old)
```

Этот атрибут является объектом property. Данный объект так устроен, что при считывании данных он вызывает первый метод get_old, этот метод возвращает значение приватного локального свойства __old экземпляра класса p и именно это значение дальше возвращается атрибутом old.

Если же мы обращаемся к атрибуту класса old и присваиваем ему какое-то значение:

p.old = 35

то автоматически вызывается второй метод set_old и в локальное свойство __old заносится значение, указанное после оператора присваивания. В итоге, в текущем объекте p меняется локальное свойство __old на новое.

Но! Нужен один интерфейс взаимодействия со свойством.

Функции-декораторы

Декоратор – это функция, которая расширяет функционал другой функции.
То есть, вот эту строчку:

old = property(get_old, set_old)

можно переписать и так:

old = property()

old = old.setter(set_old)

old = old.getter(get_old)

Можно использовать эти декораторы, чтобы сразу нужный метод класса превратить в объект-свойство property. Делается это очень просто. Перед геттером (обратите внимание, именно перед геттером, а не сеттером или делитером) прописывается декоратор:

```
@property
def get_old(self):
    return self.__old
```

Но пока присваивание не работает:

```
p.get_old = 35
```

так как мы не прописали декоратор для сеттера.

Декоратор для сеттера

```
@get_old.setter
def get_old(self, old):
    self.__old = old
```

Метод, который вызывается при удалении свойства:

```
 @old.deleter
    def old(self):
        del self.__old
```

```
class Person:
    def __init__(self, name, old):
        self.__name = name
        self.__old = old

    @property
    def old(self):
        return self.__old

    @old.setter
    def old(self, old):
        self.__old = old
```

Этот вариант эквивалентен предыдущему варианту с тем лишь отличием, что теперь напрямую вызывать сеттер или геттер для локального свойства __old не получится. Остался один интерфейс взаимодействия – объект-свойство old. Именно так чаще всего делают на практике.

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):
        ...
```

Первый (класс A) называется non-data descriptor (дескриптор не данных), а второй (класс B) – data descriptor (дескриптор данных).

Класс для представления точек в 3D

Дескриптор с названием Integer:

```
class Integer:

    def __set_name__(self, owner, name):
        self.name = "_" + name


    def __get__(self, instance, owner):
        return getattr(instance, self.name)


    def __set__(self, instance, value):
        self.verify_coord(value)
        setattr(instance, self.name, value)
```

```
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 — дескрипторы данных, через которые будет проходить взаимодействие.

Метод для проверки корректности

```
@classmethod
    def verify_coord(cls, coord):
        if type(coord) != int:
            raise TypeError("Координата должна быть целым числом")
```

Отличия дескрипторов не-данных

1. Дескрипторы не данных не могут менять значения какого-либо свойства, так как не имеют сеттера и делитера. Они служат только для считывания информации.
2. Они имеют тот же приоритет доступа, что и обычные атрибуты класса (а приоритет обращению к дескриптору данных выше).

Работа с атрибутами класса

```
class Point:
    MAX_COORD = 100
    MIN_COORD = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def set_coord(self, x, y):
        self.x = x
        self.y = y
```

Эти атрибуты остаются в пространстве имен класса, не копируются в экземпляры. Но из экземпляров мы можем совершенно спокойно к ним обращаться, так как пространство имен объектов содержит ссылку на внешнее пространство имен класса:

```
print(pt1.MAX_COORD)
```

То есть атрибуты и методы класса – это общие данные для всех его экземпляров.

Обращение к атрибутам класса внутри методов

Нельзя просто прописать имена как:

```
  def set_coord(self, x, y):
        if MIN_COORD <= x <= MAX_COORD:
            self.x = x
            self.y = y
```

Нужно либо указать перед ними ссылку на класс:

```
if Point.MIN_COORD <= x <= Point.MAX_COORD:
```

но лучше через self:

```
if self.MIN_COORD <= x <= self.MAX_COORD:
```

Рис. 5.2. Типы интерфейсов

Критерии оценки интерфейса пользователем

- простота освоения и запоминания операций системы - конкретно определяется время освоения и продолжительность сохранения информации в памяти;
- скорость достижения результатов при использовании системы - определяется количеством вводимых или выбираемых мышью команд и настроек;
- субъективная удовлетворённость при эксплуатации системы (удобство работы, утомляемость и т.д.).