# Магические методы galore

dunder-методы (от англ. сокращения double underscore)

```
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
```

```
def set_bound(self, left):
        self.MIN_COORD = left
```

локальное свойство в экземпляре класса.

```
@classmethod
    def set_bound(cls, left):
        cls.MIN_COORD = left
```

значение атрибута изменяется в самом классе

__setattr__(self, key, value)__ – автоматически вызывается при изменении свойства key класса;

__getattribute__(self, item) – автоматически вызывается при получении свойства класса с именем item;

__getattr__(self, item) – автоматически вызывается при получении несуществующего свойства item класса;

__delattr__(self, item) – автоматически вызывается при удалении свойства item (не важно: существует оно или нет).

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

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

    def __getattribute__(self, item):
        print("__getattribute__")
        return object.__getattribute__(self, item)
```

```
print(pt1.MIN_COORD)

print(pt1._Point__x)
```

```
def __getattribute__(self, item):
        if item == "_Point__x":
            raise ValueError("Private
attribute")
        else:
            return object.__getattribute__(self,
item)
```

```
def __setattr__(self, key, value):
    print("__setattr__")
    object.__setattr__(self, key, value)
```

После запуска видим несколько сообщений «__setattr__». Это связано с тем, что в момент создания экземпляров класса в инициализаторе __init__ создавались локальные свойства __x и __y. В этот момент вызывался данный метод. Также в переопределенном методе __setattr__ нужно вызвать соответствующий метод из базового класса object, иначе локальные свойства в экземплярах создаваться не будут.

Запретим создание локального свойства с именем z:

```
def __setattr__(self, key, value):
    if key == 'z':
        raise AttributeError("недопустимое имя атрибута")
    else:
        object.__setattr__(self, key, value)
```

# Внимание! Если

```
def __setattr__(self, key, value):
    if key == 'z':
        raise AttributeError("недопустимое имя атрибута")
    else:
        self.__x = value
```

то метод __setattr__ начнет выполняться по рекурсии, пока не возникнет ошибка достижения максимальной глубины рекурсии.

__getattr__ автоматически вызывается, если идет обращение к несуществующему атрибуту:

```
def __getattr__(self, item):
    print("__getattr__: " + item)
```

Зачем он может понадобиться? Например, нам необходимо определить класс, в котором при обращении к несуществующим атрибутам возвращается значение False, а не генерируется исключение.

__delattr__ вызывается в момент удаления какого-либо атрибута из экземпляра класса:

```
def __delattr__(self, item):
        print("__delattr__: "+item)
```

Добавим новое локальное свойство в экземпляр pt1:

```
pt1.a = 10
```

затем выполним команду его удаления:

```
del pt1.a
```

Увидим, что действительно был вызван метод __delattr__, но сам атрибут удален не был.

Это из-за того, что внутри этого метода нужно вызвать соответствующий метод класса object, который и выполняет непосредственное удаление:

```
def __delattr__(self, item):
    object.__delattr__(self, item)
```

# __call__

При создании экземпляра класса используются круглые скобки:

c = Counter() — это оператор вызова. В действительности, когда происходит вызов класса, то автоматически запускается магический метод __call__ , и в данном случае он создает новый экземпляр этого класса.

Схема реализации метода _call__ — 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».

```
class Counter:
    def __init__(self):
        self.__counter = 0


    def __call__(self, *args, **kwargs):
        print("__call__")
        self.__counter += 1
        return self.__counter
```

Так можно вызывать экземпляры (c()).

Классы, экземпляры которых можно вызывать подобно функциям, получили название **функторы**.

Параметры *args, **kwargs — означают, что при вызове объектов им можно передать произвольное количество аргументов.

# Применение __call__

1. Использование класса с методом __call__ вместо замыканий функций

```
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)
```

```
s1 = StripChars("?:!.; ")
s2 = StripChars(" ")
```

Объект s2 уже отвечает только за удаление пробелов, тогда как s1 — и некоторых других символов.

Решение задачи, где нам требуется сохранять символы для удаления.

# 2. Классы-декораторы

```
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

def df_sin(x):
    return math.sin(x)
```

Можно задать декоратор явно:

```
df_sin = Derivate(df_sin)
```

или через @:

```
@Derivate
def df_sin(x):
    return math.sin(x)
```

Результат одинаков — 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__.

__str__() – магический метод для отображения информации об объекте класса для пользователей (например, для функций print, str);

__repr__() – магический метод для отображения информации об объекте класса в режиме отладки (для разработчиков).

Класс для описания кошек:

```
class Cat:
    def __init__(self, name):
        self.name = name
cat = Cat('Васька')
```

<ex1.Cat object at 0x0495D028>

Если нам нужно ее как-то переопределить и отобразить в другом виде (формате), то, как раз для этого используются магические методы __str__ и __repr__. Давайте для начала переопределим метод __repr__ и посмотрим, как это отразится на выводе служебной информации о классе:

```
def __repr__(self):
        return f"{self.__class__}: {self.name}"
```

При создании экземпляра видим:

<class 'ex1.Cat'>: Васька

Добавим второй магический метод __str__ :

```
def __str__(self):
        return f"{self.name}"
```

Снова переопределим класс Cat, создадим его экземпляр и при отображении ссылки:

cat

по-прежнему будем видеть служебную информацию от метода __repr__. Однако, если выполнить отображение экземпляра класса через print или str, то будет срабатывать уже второй метод __str__. Вот в этом отличие этих двух магических методов.

# Магические методы __len__ и __abs__

__len__() – позволяет применять функцию len() к экземплярам класса;

__abs__() - позволяет применять функцию abs() к экземплярам класса.

```
class Point:
    def __init__(self, *args):
        self.__coords = args
```

Определимразмерность координат с помощью функции len():

```
p = Point(1, 2)
print(len(p))
```

Нужно добавить

```
def __len__(self):
    return len(self.__coords)
```

— магический метод __len__ указал, что нужно возвращать, в момент применения функции len() к экземпляру класса.

Следующий магический метод __abs__ работает аналогичным образом, только активируется в момент вызова функции abs для экземпляра класса.

```
def __abs__(self):
    return list( map(abs, self.__coords) )
```

# Методы для работы с арифметическими операторами

__add__() – для операции сложения;

__sub__() – для операции вычитания;

__mul__() – для операции умножения;

__truediv__() – для операции деления.

```
class Clock:
    __DAY = 86400   # число секунд в одном дне
    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY
```

Далее в этом же классе пропишем метод get_time для получения текущего времени в виде форматной строки:

```
def get_time(self):
        s = self.seconds % 60            # секунды
        m = (self.seconds // 60) % 60    # минуты
        h = (self.seconds // 3600) % 24  # часы
        return
f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:{self.__get_formatted(s)}"


    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")
```

Как добавить 100 секунд?

c1.seconds = c1.seconds + 100

Но хочется:

c1 = c1 + 100

Добавим

```
def __add__(self, other):
        if not isinstance(other, int):
            raise ArithmeticError("Правый операнд должен быть типом int")


        return Clock(self.seconds + other)
```

```
c1 = Clock(1000)

c1 = c1.__add__(100)
```

В результате активируется метод __add__ и параметр other принимает целочисленное значение 100. Проверка проходит и формируется новый объект класса Clock со значением секунд 1000+100 = 1100. Этот объект возвращается методом __add__, и переменная c1 начинает ссылаться на этот новый экземпляр класса. На прежний уже не будет никаких внешних ссылок, поэтому он будет автоматически удален сборщиком мусора.

А если мы хотим складывать два разных объекта класса Clock?

```
c1 = Clock(1000)

c2 = Clock(2000)

c3 = c1 + c2

print(c3.get_time())
```

По умолчанию будет ошибка, но изменим реализацию

```
def __add__(self, other):
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или объектом Clock")

        sc = other if isinstance(other, int) else other.seconds
        return Clock(self.seconds + sc)
```

— и теперь можно складывать и отдельные целые числа, и объекты Clock.

Важный нюанс — порядок записи.

c1 = c1 + 100 => ок

c1 = 100 + c1 => ошибка, потому что получится 100.__add__(c1).

Как решить проблему? Специальные методы в Python с добавлением буквы r: __radd__() — будет вызван, если не может быть вызван метод __add__().

```
def __radd__(self, other):
    return self + other
```

Тогда будет вызван метод __add__, но с правильным порядком типов данных, поэтому сложение пройдет без ошибок.

# Модификация с первой i

_iadd__() вызывается для команды: c1 += 100

Если запустить сейчас программу, то никаких ошибок не будет и отработает метод __add__(). Но в методе __add__ создается новый объект класса Clock, тогда как при операции += этого делать не обязательно.

```
def __iadd__(self, other):
        print("__iadd__")
        if not isinstance(other, (int, Clock)):
            raise ArithmeticError("Правый операнд должен быть типом int или объектом Clock")
        sc = other if isinstance(other, int) else other.seconds
        self.seconds += sc
        return self
```

Так не создается новый объект, а меняется число секунд в текущем. Это логичнее, так как вызывать цепочкой операцию += не предполагается и, кроме того, она изменяет (по смыслу) состояние текущего объекта.

| Оператор | Метод оператора | Оператор | Метод оператора |
|---|---|---|---|
| x + y | __add__(self, other) | x += y | __iadd__(self, other) |
| x - y | __sub__(self, other) | x -= y | __isub__(self, other) |
| x * y | __mul__(self, other) | x *= y | __imul__(self, other) |
| x / y | __truediv__(self, other) | x /= y | __itruediv__(self, other) |
| x // y | __floordiv__(self, other) | x //= y | __ifloordiv__(self, other) |
| x % y | __mod__(self, other) | x %= y | __imod__(self, other) |

# Методы сравнений __eq__, __ne__, __lt__, __gt__ и т.д.

__eq__() – для равенства ==

__ne__() – для неравенства !=

__lt__() – для оператора меньше <

__le__() – для оператора меньше или равно <=

__gt__() – для оператора больше >

__ge__() – для оператора больше или равно >=

```
class Clock:
    __DAY = 86400   # число секунд в одном дне
    def __init__(self, seconds: int):
        if not isinstance(seconds, int):
            raise TypeError("Секунды должны быть целым числом")
        self.seconds = seconds % self.__DAY
    def get_time(self):
        s = self.seconds % 60            # секунды
        m = (self.seconds // 60) % 60    # минуты
        h = (self.seconds // 3600) % 24  # часы
        return f"{self.__get_formatted(h)}:{self.__get_formatted(m)}:{self.__get_formatted(s)}"
    @classmethod
    def __get_formatted(cls, x):
        return str(x).rjust(2, "0")
```

Изначально для класса реализован только один метод сравнения на равенство:

```
c1 = Clock(1000)

c2 = Clock(1000)

print(c1 == c2)
```

Но здесь объекты сравниваются по их id (адресу в памяти), а нужно, чтобы сравнивались секунды в каждом из объектов c1 и c2. Для этого переопределим магический метод __eq__():

```
def __eq__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")
        sc = other if isinstance(other, int) else other.seconds
        return self.seconds == sc
```

После запуска программы видим значение True, т.к. объекты содержат одинаковое время. Кроме того, можно выполнять проверку и на неравенство:

print(c1 != c2)

Если интерпретатор языка Python не находит определение метода ==, то он пытается выполнить противоположное сравнение с последующей инверсией результата. В данном случае находится оператор == и выполняется инверсия: not (a == b)

Добавим сравнение больше/меньше:

```
def __lt__(self, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")
        sc = other if isinstance(other, int) else other.seconds
        return self.seconds < sc => так будет дублирование класса, так что


@classmethod
    def __verify_data(cls, other):
        if not isinstance(other, (int, Clock)):
            raise TypeError("Операнд справа должен иметь тип int или Clock")
        return other if isinstance(other, int) else other.seconds
```

```
def __eq__(self, other):
        sc = self.__verify_data(other)
        return self.seconds == sc


    def __lt__(self, other):
        sc = self.__verify_data(other)
        return self.seconds < sc
```

Если сделать проверку на больше: print(c1 > c2), то сработает тот же метод меньше, но для объекта c2: c2 < c1

В отличие от оператора ==, где применяется инверсия, здесь меняется порядок операндов.

Можно определить:

```
def __gt__(self, other):
        sc = self.__verify_data(other)
        return self.seconds > sc
```

Меньше или равно:

```
def __le__(self, other):
        sc = self.__verify_data(other)
        return self.seconds <= sc
```

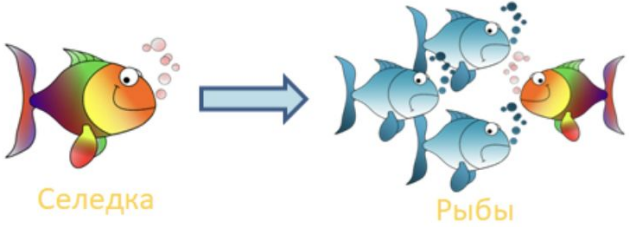
# ХЭШ!

hash(123)

hash("Python")

hash((1, 2, 3))

Формирует по определенному алгоритму целочисленные значения для неизменяемых объектов. Причем для равных объектов на выходе всегда должны получаться равные хэши, но равные хэши не гарантируют равенство объектов.



Свойства хеша:

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

Причем хэши можно вычислять только для неизменяемых объектов. Но зачем все это надо? Некоторые объекты в Python, например, словари используют хэши в качестве своих ключей. Ключ словаря должен относиться к неизменяемому типу данных:

```
d = {}

d[5] = 5

d["python"] = "python"

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

Это необходимо, чтобы можно было вычислить хеш объектов и ключи хранить в виде:

(хэш ключа, ключ)

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.