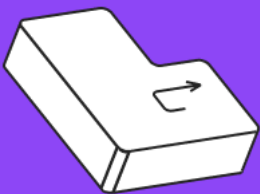


Лекция 11.

ООП. Особенности

Python

Погружение в Python



Оглавление

На этой лекции мы	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции	
1. Создание и удаление	4
Шаблон Одиночка, Singleton	7
Задание	9
2. Строка документации	10
Хранение документации в <code>__doc__</code>	11
3. Представления экземпляра	12
Представление для пользователя, <code>__str__</code>	13
4. Математика и логика в классах	17
Сдвиг вправо, <code>__rshift__</code>	19
Right методы	20
Умножение текста на “продвинутый текст” методом <code>__rmul__</code>	20
In place методы	21
Вычисление процентов вместо нахождения остатка от деления, <code>__imod__</code>	21
Задание	22
5. Сравнение экземпляров класса	23
Сравнение на идентичность, <code>__eq__</code>	23
Сравнение на больше и меньше	25
Неизменяемые экземпляры, хеширование дандер <code>__hash__</code>	26
Простейшая реализация хэша	28
Задание	29
6. Обработка атрибутов	29
Получение значения атрибута, <code>__getattr__</code>	30

Присвоение атрибуту значения, <code>__setattr__</code>	31
Обращение к несуществующему атрибуту, <code>__getattr__</code>	32
Удаление атрибута, <code>__delattr__</code>	33

На этой лекции мы

1. Разберёмся с созданием и удалением классов
2. Узнаем о документировании классов
3. Изучим способы представления экземпляров
4. Узнаем о возможностях переопределения математических операций
5. Разберёмся со сравнением экземпляров
6. Узнаем об обработке атрибутов

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с объектно-ориентированным программированием в Python.
2. Изучили особенности инкапсуляции в языке
3. Узнали о наследовании и механизме разрешения множественного наследования.
4. Разобрались с полиморфизмом объектов.

Термины лекции

- **Дандер** — имя переменной, начинающейся и заканчивающейся двумя подчеркиваниями. В Python такие переменные используются для создания специальных свойств и методов объекта, влияющих на его поведение.

Подробный текст лекции

1. Создание и удаление

Разберём процессы, которые происходят при создании и удалении. Python позволяет контролировать создание класса и экземпляра класса, а также удаление экземпляра. Забегая вперёд, задачи контроля создания класса и удаления экземпляра более высокого уровня. В обычной практике они используются редко. Но общие знания лишними не будут для понимания классов Python в целом.

Создание экземпляра класса, `__init__`

За создание экземпляра класса отвечает дандер метод `__init__`. С ним вы уже знакомы по прошлому занятию. Всё дело в том, что это самый частый дандер метод в ООП Python

```
class User:
    def __init__(self, name: str, equipment: list = None):
        self.name = name
        self.equipment = equipment if equipment is not None else []

        self.life = 3
        # принтим только в учебных целях, а не для реальных задач
        print(f'Создал {self} со свойствами:\n'
              f'{self.name = },\t{self.equipment = },\t{self.life'
              = }')

print('Создаём первый раз')
u_1 = User('Спенглер')
print('Создаём второй раз')
u_2 = User('Венкман', ['протонный ускоритель', 'ловушка'])
print('Создаём третий раз')
u_3 = User(equipment=['ловушка', 'прибор ночного видения'],
            name='Стэнц')
```

Класс User дважды вызывается для создания экземпляров u_1, u_2 и u_3. Вызов класса - это указание его имени с круглыми скобками после. Такой вызов автоматически перенаправляется в метод `__init__`. По сути мы вызвали его, следовательно возможность передавать аргументы при создании экземпляра прописывается параметрами “инит”.

В нашем примере `name` является обязательным параметром, а `equipment` - не обязателен. При этом значения можно передавать как позиционно, так и по ключу. В этом плане методы классов работают аналогично обычным функциям.

Внутри метода каждый аргумент присваивается переменной экземпляра. Она начинается с `self` — названия первого параметра метода. При этом для “оборудования” делается проверка на `None`. Если список не передан, создаётся новый для каждого экземпляра. Кроме того свойство `life` получает значение 3. Подобный подход является предпочтительным. У каждого экземпляра будет своё свойство `life` изменяющееся внутри и не зависящее от других экземпляров и от самого класса User.

Вывод на печать позволяет отследить порядок выполнения действий. Пока не отработает инициализация экземпляра, дальше код не выполняется.



Важно! Постарайтесь разобраться в работе дандер метода `__init__`. Он встречается чаще других и определяет структуру экземпляров. В подавляющем большинстве (если речь не идёт о метапрограммировании) внутри метода прописываются все свойства экземпляра.

Контроль создания класса через `__new__`

Метод `__new__` срабатывает раньше `__init__` и определяет что должен вернуть класс в качестве себя - класса. Рассмотрим вначале общий пример.

```
class User:
    def __init__(self, name: str):
        self.name = name
        print(f'Создал {self.name = }')

    def __new__(cls, *args, **kwargs):
        instance = super().__new__(cls)
        print(f'Создал класс {cls}')
        return instance

print('Создаём первый раз')
u_1 = User('Спенглер')
```

```
print('Создаём второй раз')
u_2 = User('Венкман')
print('Создаём третий раз')
u_3 = User(name='Стэнц')
```

Метод `__new__` принимает в качестве первого параметра сам себя. Обычно используют слово `cls` — сокращение от `class`. Так понятно, что мы работаем с классом, а не с его экземпляром. Параметры `*args`, `**kwargs` нужны для правильной работы метода `__init__` и попадания в него любых аргументов.

Внутри `__new__` необходимо вызвать аналогичный родительский метод. Он возвращает класс, который можно модифицировать прежде чем вернуть из метода.

Расширение неизменяемых классов

Один из вариантов использования дандер `__new__` — расширение функциональности уже имеющихся неизменяемых классов Python. Например мы хотим использовать переменную целого типа, которая дополнительно хранит присвоенное числу имя.

```
class NamedInt(int):
    def __new__(cls, value, name):
        instance = super().__new__(cls, value)
        instance.name = name
        print(f'Создал класс {cls}')
        return instance

print('Создаём первый раз')
a = NamedInt(42, 'Главный ответ жизни, Вселенной и вообще')
print('Создаём второй раз')
b = NamedInt(73, 'Лучшее просто число')
print(f'{a = }\t{a.name = }\t{type(a) = }')
print(f'{b = }\t{b.name = }\t{type(b) = }')
c = a + b
print(f'{c = }\t{type(c) = }')
```

Параметр `value` нужен для передачи значения в родительский класс `int`. Далее к целому числу добавляется параметр `name` с переданным значением. После создания объекта он возвращается для присваивания переменной.

Обратите внимание, что наш класс унаследовал всё, что умеет класс `int`. Мы смогли сложить два числа и получить обычное целое число без свойства `name`.

Шаблон Одиночка, Singleton

Ещё один вариант использования дандре `__new__` — паттерн Singleton. Он позволяет создавать лишь один экземпляр класса. Вторая и последующие попытки будут возвращать ранее созданный экземпляр.



Внимание! Это не единственный вариант создания паттерна Одиночка в Python, а версия через `__new__`

```
class Singleton:
    _instance = None

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

    def __init__(self, name: str):
        self.name = name

a = Singleton('Первый')
print(f'{a.name = }')
b = Singleton('Второй')
print(f'{a is b = }')
print(f'{a.name = }\n{b.name = }')
```

Защищенная переменная `_instance` хранит `None` и при создании первого экземпляра получает на него ссылку. Благодаря `*args, **kwargs` в методе `__new__` аргумент “Первый” проваливается в метод `__init__` и попадает в параметр `name`. При создании экземпляра второй раз возвращается его первая версия и у неё заменяется свойство `name`. Переменные `a` и `b` ссылаются на один и тот же класс, а следовательно их свойство `name` общее.

Удаление экземпляра класса, `__del__`

Дандер метод `__del__` также редко используется в обычной практике ООП. Он отвечает за действия при удалении экземпляра класса. Если быть более точным, метод срабатывает при достижении нуля счётчиком ссылок на объект, но перед его удалением из памяти сборщиком мусора. Рассмотрим простой пример.

```
class User:
    def __init__(self, name: str):
        self.name = name
        print(f'Создал {self.name = }')

    def __del__(self):
        print(f'Удаление экземпляра {self.name}')

u_1 = User('Спенглер')
u_2 = User('Венкман')
```

После создания двух экземпляров программа завершилась. Начался процесс освобождения памяти и перед удалением экземпляров был вызван их метод `__del__`. Обычно метод `__del__` используют для явного освобождения других ресурсов. Например экземпляр использует соединения с базой данных и перед удалением его необходимо корректно завершить.



Важно! Python в любом случае удалит бы экземпляры и освободил память, даже если дандер `__del__` не прописан.

Команда del

Зарезервированное слово `del` не удаляет объекты в Python. Оно разрывает связь между переменной и объектом, на который переменная указывает. После строки с `del` к переменной нельзя обратиться, а у объекта на единицу уменьшается счётчик ссылок.

```
import sys

class User:
    def __init__(self, name: str):
        self.name = name
```



```

print(f'Создал {self.name = }')

def __del__(self):
    print(f'Удаление экземпляра {self.name}')

u_1 = User('Спенглер')
print(sys.getrefcount(u_1))
u_2 = u_1
print(sys.getrefcount(u_1), sys.getrefcount(u_2))
del u_1
print(sys.getrefcount(u_2))
print('Завершение работы')
```

В момент первого вызова метода `getrefcount` имеем одно значение счётчика ссылок. Когда переменная `u_2` получает ссылку на тот же объект, счётчик ссылок увеличивается на единицу. Команда `del` уменьшает счётчик на единицу. А удаление объекта происходит после завершения кода.



Важно! Счётчик ссылок возвращает значение больше ожидаемого, так как сама функция создаёт дополнительную ссылку на объект при вызове.

Если в коде дописать `del u_2` в предпоследней строке, удаление объекта произойдёт раньше завершения работы программы. Счётчик ссылок дошёл до нуля и сборщик мусора освободил память.

Задание

Перед вами несколько строк кода. Напишите что делает класс, не запуская код. У вас 3 минуты.

```

class Count:
    _count = 0
    _last = None

    def __new__(cls, *args, **kwargs):
        if cls._count < 3:
            cls._last = super().__new__(cls)
            cls._count += 1
```

```

        return cls._last

    def __init__(self, name: str):
        self.name = name

```

2. Строка документации

Как и при создании функции, при создании классов принято оставлять документацию к нему. Для этого достаточно использовать многострочный комментарий сразу после определения класса — строки `class ClassName:`. Посмотрите на пример

```

class User:
    """A User training class for demonstrating class
    documentation.
    Shows the operation of the help(cls) and the dander method
    __doc__"""

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

u_1 = User('Спенглер')
print('Справка класса User ниже', '*' * 50)
help(User)
print('Справка экземпляра u_1 ниже', '*' * 50)
help(u_1)

```

После заголовка класса оставили пару строк описания. Также добавлена строка документации для метода `__init__`. Далее вызываем справку для класс `User` и для его экземпляра. С первого взгляда они похожи и выводят куда больше информации. Помимо оставленной документации `help` собирает информацию о методах класса. Выводит первую строку метода и строку документации метода, если она задана многострочным комментарием.

Отличие справки для класса и экземпляра лишь в первой строке. Сравните:

Help on class User in module `__main__`:

Help on User in module `__main__` object:

Хранение документации в `__doc__`

Любая многострочная строка после заголовка класса и метода автоматически сохраняется в дандер переменную `__doc__`. Помимо вызова справки через функцию `help` можно прочитать отдельный многострочник напрямую обратившись к переменной.

```
class User:
    """A User training class for demonstrating class
    documentation.
    Shows the operation of the help(cls) and the dander method
    __doc__"""

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name
        print(f'Создал {self.name = }')

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

u_1 = User('Спенглер')
print(f'Документация класса: {User.__doc__ = }')
print(f'Документация экземпляра: {u_1.__doc__ = }')
print(f'Документация метода: {u_1.simple_method.__doc__ = }')
```

Как и в случае с `help` обращение через класс или через экземпляр не даёт разницы.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class MyClass:
    A = 42
```

```

"""About class"""

def __init__(self, a, b):
    """self.__doc__ = None"""
    self.a = a
    self.b = b

def method(self):
    """Documentation"""
    self.__doc__ = None

help(MyClass)

```

3. Представления экземпляра

При работе с классами, а точнее с их экземплярами бывает необходимо вывести их содержимое в консоль. С этим отлично справляется функция `print`, но есть одно но. Попробуем “запринтить” класс из примера выше.

```

class User:

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

user = User('Спенглер')
print(user)

```

В результат получили строку вида `<__main__.User object at 0x000001C1B4FA6B60>`. Число в конце — адрес объекта в оперативной памяти. Он может быть разным для разных ПК и даже при разных запусках программы. Но пользы от этой информации немного. Для получения читаемого описания необходимо переопределить как минимум один из дандер методов: `__str__` или `__repr__`.

Представление для пользователя, `__str__`

Дандер метод `__str__` используется для получения удобного пользователю описания экземпляра.

```
class User:

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

    def __str__(self):
        return f'Экземпляр класса User с именем "{self.name}"'

user = User('Спенглер')
print(user)
```

Метод `__str__` обязан вернуть строку `str`. Обычно это строка содержит информацию о свойствах класса для понимания что за экземпляр перед нами. Упор делается на удобство чтения. Но и о краткости забывать не стоит.

Представление для создания экземпляра, `__repr__`

Дандер метод `__repr__` аналогичен `__str__`, но возвращает максимально близкое к созданию экземпляра класса представление.

```
class User:

    def __init__(self, name: str):
```

```

        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

    def __repr__(self):
        return f'User({self.name})'

user = User('Спенглер')
print(user)

```

Если скопировать вывод метода `repr` и присвоить его переменной, должен получится ещё один экземпляр класса. Рассмотрим более сложный класс и его метод `__repr__`.

```

class User:

    def __init__(self, name: str, equipment: list = None):
        self.name = name
        self.equipment = equipment if equipment is not None else
[]
        self.life = 3

    def __repr__(self):
        return f'User({self.name}, {self.equipment})'

user = User('Венкман', ['протонный ускоритель', 'ловушка'])
print(user)

```

Мы снова получили строку, которую можно скопировать и создать экземпляр без внесения правок. При этом свойство `life` опущено в выводе, т.к. не влияет на создание экземпляра.

Приоритет методов

Добавим классу из примера выше метод `__str__` и посмотрим какой из них сработает.

```

class User:

```

```

def __init__(self, name: str, equipment: list = None):
    self.name = name
    self.equipment = equipment if equipment is not None else
[]
    self.life = 3

def __str__(self):
    eq = 'оборудованием: ' + ', '.join(self.equipment) if
self.equipment else 'пустыми руками'
    return f'Перед нами {self.name} с {eq}. Количество жизней
- {self.life}'

def __repr__(self):
    return f'User({self.name}, {self.equipment})'

user = User('Венкман', ['протонный ускоритель', 'ловушка'])
print(user)

```

При вызове функции print сработал метод __str__. Как же получить вывод от __repr__ при наличии двух методов? Есть несколько способов вывода на печать:

```

class User:
    ...

user = User('Венкман', ['протонный ускоритель', 'ловушка'])
print(user)
print(f'{user}')

print(repr(user))
print(f'{user = }')

```

В первых двух вариантах срабатывает дандер __str__. Далее мы явно передаём в print результат встроенной функции repr, которая обращается к одноимённому методу. Так же при использовании f-строк символ равенства выводит имя переменной слева от знака равно и repr справа от него.

Печать коллекций

Однако метод __repr__ оказывается более приоритетным, если на печать выводится не один элемент, а коллекция элементов.

```

class User:

    def __init__(self, name: str, equipment: list = None):
        self.name = name
        self.equipment = equipment if equipment is not None else []
        self.life = 3

    def __str__(self):
        eq = 'оборудованием: ' + ', '.join(self.equipment) if self.equipment else 'пустыми руками'
        return f'Перед нами {self.name} с {eq}. Количество жизней - {self.life}'

    def __repr__(self):
        return f'User({self.name}, {self.equipment})'

u_1 = User('Спенглер')
u_2 = User('Венкман', ['протонный ускоритель', 'ловушка'])
u_3 = User(equipment=['ловушка', 'прибор ночного видения'], name='Стэнц')
ghostbusters = [u_1, u_2, u_3]

print(ghostbusters)
print(f'{ghostbusters}')
print(repr(ghostbusters))
print(f'{ghostbusters = }')

print(*ghostbusters, sep='\n')

```

В приведённом примере список из трёх экземпляров при печати возвращает герг представление во всех четырёх рассмотренных способах. И только при распаковке списка через звёздочку функция print получает экземпляры напрямую и вызывает их дандер `__str__`.

Задание

Перед вами несколько строк кода. Что в нём неправильно. У вас 3 минуты.

```

class MyClass:

    def __init__(self, a, b):

```



```

self.a = a
self.b = b
self.c = a + b

def __str__(self):
    return f'MyClass(a={self.a}, b={self.b}, c={self.c})'

def __repr__(self):
    return str(self.a) + str(self.b) + str(self.c)

```

4. Математика и логика в классах

Под математикой стоит понимать переопределение дандер методов, которые позволяют производить операции сложения, вычитания, умножения и т.п. с использованием математических символов. Что касается логики, речь идёт о логических операциях “и”, “или” и т.п. над объектами. Рассмотрим возможности Python в таблице.

Операция в Python	Основной метод	Right метод	In place метод
+	<code>__add__(self, other)</code>	<code>__radd__(self, other)</code>	<code>__iadd__(self, other)</code>
-	<code>__sub__(self, other)</code>	<code>__rsub__(self, other)</code>	<code>__isub__(self, other)</code>
*	<code>__mul__(self, other)</code>	<code>__rmul__(self, other)</code>	<code>__imul__(self, other)</code>
@	<code>__matmul__(self, other)</code>	<code>__rmatmul__(self, other)</code>	<code>__imatmul__(self, other)</code>
/	<code>__truediv__(self, other)</code>	<code>__rtruediv__(self, other)</code>	<code>__itruediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>	<code>__rfloordiv__(self, other)</code>	<code>__ifloordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>	<code>__rmod__(self, other)</code>	<code>__imod__(self, other)</code>
<code>divmod()</code>	<code>__divmod__(self, other)</code>	<code>__rdivmod__(self, other)</code>	<code>__idivmod__(self, other)</code>
<code>**</code> , <code>pow()</code>	<code>__pow__(self, other[, modulo])</code>	<code>__rpow__(self, other[, modulo])</code>	<code>__ipow__(self, other[, modulo])</code>
<<	<code>__lshift__(self, other)</code>	<code>__rlshift__(self, other)</code>	<code>__ilshift__(self, other)</code>
>>	<code>__rshift__(self, other)</code>	<code>__rrshift__(self, other)</code>	<code>__irshift__(self, other)</code>
&	<code>__and__(self, other)</code>	<code>__rand__(self, other)</code>	<code>__iand__(self, other)</code>

^	__xor__(self, other)	__rxor__(self, other)	__ixor__(self, other)
	__or__(self, other)	__ror__(self, other)	__ior__(self, other)

Переопределение перечисленных в таблице методов позволяет использовать указанные в первом столбце операции для вычисления результата. Рассмотрим некоторые из них на примерах.

Обычные методы

Начнём с методов из второго столбца. Если Python встречает два экземпляра класса с одним из знаков между ними, ищется соответствующий знаку дандер метод для вызова. Если метод не определён, возвращается ошибка. При этом метод должен возвращать новый экземпляр класса без изменения исходных.

Сложение через __add__

Создадим класс вектор и научим вектора складываться.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

a = Vector(2, 4)
b = Vector(3, 7)
c = a + b
print(f'{a = }\t{b = }\t{c = }')
```

Помимо уже привычных методов `__init__` и `__repr__` определили метод `__add__`. В предпоследней строке пытаемся сложить вектора. Без метода `__add__` получили бы ошибку вида: `TypeError: unsupported operand type(s) for +: 'Vector' and 'Vector'`.

В самом методе используем два параметра — `self` для обращения к элементам экземпляра и `other` для обращения к элементам другого объекта, стоящего справа от знака плюс. Получив значения `x`, `y` для нового вектора метод возвращает его — новый экземпляр класса `Vector`.

Сдвиг вправо, `__rshift__`

Переопределение методов не обязательно должно быть для чисел. Напишем класс, который генерирует шкаф с одеждой и выбрасывает указанное количество вещей при правом сдвиге. Не забудем, что дандер метод должен возвращать новый экземпляр.

```
from random import choices

class Closet:
    CLOTHES = ('брюки', 'рубашка', 'костюм', 'футболка',
               'перчатки', 'носки', 'туфли')

    def __init__(self, count: int, storeroom=None):
        self.count = count
        if storeroom is None:
            self.storeroom = choices(self.CLOTHES, k=count)
        else:
            self.storeroom = storeroom

    def __str__(self):
        names = ', '.join(self.storeroom)
        return f'Осталось вещей в шкафу {self.count}: \n{names}'

    def __rshift__(self, other):
        shift = self.count if other > self.count else other
        self.count -= shift
        return Closet(self.count, choices(self.storeroom,
                                           k=self.count))

storeroom = Closet(10)
print(storeroom)
```

```
for _ in range(4):  
    storeroom = storeroom >> 3  
    print(storeroom)
```

Константа CLOTHES хранит доступный список одежды. Из него будем выбирать count предметов. Внутри __rshift__ сделали проверку на оставшееся количество вещей, чтобы не выбросить больше, чем уже имеется. Метод возвращает новый экземпляр, где count уменьшился на сдвиг, а второй аргумент содержит выборку из уже лежащих в шкафу вещей.

Создав экземпляр на 10 вещей и последовательно удаляем по три предмета в цикле. Но уйти в минус не удаётся, обрабатывает наша защита.

Right методы

Right методы срабатывают в том случае, если у левого аргумента в выражении метод не был найден. Например при записи `x + y` вначале производится поиска дандер метода `x.__add__`. Если он не найден, вызываем `y.__radd__`.

Умножение текста на “продвинутый текст” методом __rmul__

Создадим класс на основе `str` с методом `__rmul__`. Если слева оказывается обычная строка, будем между словами добавлять текст из “продвинутой строки”, перемножим их.

```
class StrPro(str):  
    def __new__(cls, *args, **kwargs):  
        instance = super().__new__(cls, *args, **kwargs)  
        return instance  
  
    def __rmul__(self, other: str):  
        words = other.split()  
        result = self.join(words)  
        return StrPro(result)
```

```

text = 'Каждый охотник желает знать где сидит фазан'
s = StrPro(' (=^.^=) ')
print(f'{text = }\n{s = }')
print(text * s)
print(s * text)  # TypeError: 'str' object cannot be interpreted
                  as an integer

```

Метод `__new__` позволили нам наследоваться от класса `str` и забрать все свойства и методы, определённые в нём. Мы добавили лишь `__rmul__` где делим строку стоящую слева от знака умножить - `other` на отдельные слова. Далее собираем новую строку с добавлением `self`- строки справа от знака умножения.

При умножении `str` на `StrPro` получаем ожидаемый результат. Если же поменять значения местами, получаем ошибку. Обычную строку можно умножить на целое число, но не другой экземпляр.

In place методы

In place методы используются при короткой записи математического символа слитно со знаком равенства: `a += b`. Такая запись подразумевает внесение изменений в исходный объект, а не возврат нового экземпляра. Возвращать надо самого себя — `self`.

Вычисление процентов вместо нахождения остатка от деления, `__imod__`

Создадим простой класс `Money`, который будет увеличивать значение на указанный процент при записи `Money %= float | int`

```

from decimal import Decimal

class Money:
    def __init__(self, value: int | float):
        self.value = Decimal(value)

```

```
def __repr__(self):
    return f'Money({self.value:.2f})'

def __imod__(self, other):
    self.value = self.value * Decimal(1 + other / 100)
    return self
```

```
m = Money(100)
print(m)
m %= 50
print(m)
m %= 100
print(m)
```

Для точности вычислений используется класс `Decimal`. Поэтому при увеличении на указанный процент используем дополнительное обёртывание правого значения в `Decimal`.



Важно! Не забывайте `return self` при работе с `in place` дандер методами.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class MyClass:

    def __init__(self, data):
        self.data = data

    def __and__(self, other):
        return MyClass(self.data + other.data)

    def __str__(self):
        return str(self.data)

a = MyClass((1, 2, 3, 4, 5))
b = MyClass((2, 4, 6, 8, 10))
print(a & b)
```

5. Сравнение экземпляров класса

Числа сравниваются по значению, строки посимвольно. Но при желании можно сравнивать любые объекты Python реализовав перечисленные ниже дандер методы.

- `__eq__` - равно, `==`
- `__ne__` - не равно, `!=`
- `__gt__` - больше, `>`
- `__ge__` - не больше, меньше или равно, `<=`
- `__lt__` - меньше, `<`
- `__le__` - не меньше, больше или равно, `>=`

Перечисленные методы попарно противоположны. Обратите внимание на приставку не в списке. Реализовав один из пары, второй Python попытается получить инвертируя значение. Не истина — это ложь, а не ложь — это истина.

При реализации метода обычно принято возвращать `True` или `False`. Если возвращается другое значение в конструкциях вида `if x == y:`, Python применит функцию `bool()` к результату для получения `True` или `False`.

Сравнение на идентичность, `__eq__`

Создадим класс треугольник, который хранит длины трёх сторон. В первом варианте не будем прописывать дандер `__eq__` и попробуем сравнить экземпляры.

```
class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __str__(self):
        return f'Треугольник со сторонами: {self.a}, {self.b}, {self.c}'

one = Triangle(3, 4, 5)
two = one
three = Triangle(3, 4, 5)
print(one == two)
print(one == three)
```

Переменные one и two равны, т.к. ссылаются на один и тот же объект в памяти. А вот треугольники one и three считаются разными хоть и имеют одинаковые длины сторон. Дело в том, что Python по умолчанию добавляет метод `__eq__` следующего вида.

```
def __eq__(self, other):  
    return self is other
```

Как вы помните `is` сравнивает адреса объектов в памяти. Следовательно проверка по умолчанию это: `True if id(self) == id(other) else False`.

А теперь напишем свою проверку на идентичность. Допустим возможность переворачивания треугольника перед сравнением. Например треугольники со сторонами 3, 4, 5 и 4, 3, 5 будем считать равными.

```
class Triangle:  
    def __init__(self, a, b, c):  
        self.a = a  
        self.b = b  
        self.c = c  
  
    def __str__(self):  
        return f'Треугольник со сторонами: {self.a}, {self.b}, {self.c}'  
  
    def __eq__(self, other):  
        first = sorted((self.a, self.b, self.c))  
        second = sorted((other.a, other.b, other.c))  
        return first == second  
  
one = Triangle(3, 4, 5)  
two = one  
three = Triangle(3, 4, 5)  
four = Triangle(4, 3, 5)  
print(f'{one == two    = }')  
print(f'{one == three = }')  
print(f'{one == four  = }')  
print(f'{one != one   = }')
```

Функция `sorted` получает кортеж из трёх сторон и возвращает их в упорядоченном виде. Сравнив оба списка поэлементно определяем равны треугольники или нет. Обратите внимание на последнюю строку. Проверка на неравенство не вызвала ошибку. Python вызвал дандер `__eq__`, а к результату применил команду `not`.

Сравнение на больше и меньше

При необходимости сравнивать объекты не только на равенство, можно реализовать дополнительные методы сравнения. Например для работы сортировки используется метод `__lt__`, проверяющий какой из пары элементов меньше.

Доработаем класс треугольника и будем сравнивать их по площади. Для вычисления площади напишем отдельный метод, использующий [формулу Герона](#).

```
from math import sqrt

class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __str__(self):
        return f'Треугольник со сторонами: {self.a}, {self.b}, {self.c}'

    def __repr__(self):
        return f'Triangle({self.a}, {self.b}, {self.c})'

    def __eq__(self, other):
        first = sorted((self.a, self.b, self.c))
        second = sorted((other.a, other.b, other.c))
        return first == second

    def area(self):
        p = (self.a + self.b + self.c) / 2
        _area = sqrt(p * (p - self.a) * (p - self.b) * (p - self.c))
        return _area

    def __lt__(self, other):
        return self.area() < other.area()

one = Triangle(3, 4, 5)
two = Triangle(5, 5, 5)
print(f'{one} имеет площадь {one.area():.3f} y.e.2')
print(f'{two} имеет площадь {two.area():.3f} y.e.2')
print(f'{one > two} = {\n{one < two} = }')
```

```
data = [Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4, 4, 4),
Triangle(3, 5, 3)]
result = sorted(data)
print(result)
print(', '.join(f'{item.area():.3f}' for item in result))
```

Рассмотрим получившийся код:

- дандер `__init__` и `__str__` остались без изменений
- дандер `__repr__` нужен для вывода читаемого списка экземпляров
- дандер `__eq__` также не изменился
- метод `area` вычисляет площадь треугольника по формуле Герона:
 - находим p как половину суммы длин сторон
 - вычисляем площадь как корень из произведения p на разность p с каждой из сторон
- дандер `__lt__` вызывает для каждого из сравниваемых экземпляров метод `area` и возвращает результат сравнения площадей: `True` или `False`.

В основном коде создали пару треугольников, посмотрели на их площадь и убедились, что сравнения на больше так же работает и возвращает обратное от сравнения на меньше.

Далее создали список треугольников и отсортировали их. Визуальная проверка площадей подтверждает, что треугольники были упорядочены именно на основе их сравнения.

Неизменяемые экземпляры, хеширование дандер `__hash__`

Как вы помните ключом `dict` и элементами `set` и `frozenset` могут быть только неизменяемые типы данных. А для проверки на неизменяемость используется функция `hash()`. Она должна возвращать целое число в 4 или 8 байт в зависимости от разрядности интерпретатора Python. И это число должно быть неизменным на всём протяжении работы программы.

Попробуем сложить наши треугольники из примера выше в множество не изменяя код.

```
from math import sqrt

class Triangle:
    ...
```

```
triangle_set = {Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4,
4, 4), Triangle(3, 5, 3)}
print(triangle_set)
```

Получаем ошибку `TypeError: unhashable type: 'Triangle'` ведь дандер `__hash__` у нас не реализован. Прежде чем приступить к написанию кода попробуем закомментировать метод проверки на равенство и запустит код снова.

```
from math import sqrt

class Triangle:
    ...
    # def __eq__(self, other):
    #     first = sorted((self.a, self.b, self.c))
    #     second = sorted((other.a, other.b, other.c))
    #     return first == second
    ...

triangle_set = {Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4,
4, 4), Triangle(3, 5, 3)}
print(triangle_set)
print(', '.join(f'{hash(item)}' for item in triangle_set))
```

Работает! Экземпляры треугольника стали хэшируемыми. Правило следующее.

- нет `__eq__`, нет `__hash__` - неизменяемый объект. Python сам реализует оба дандера
- есть `__eq__`, нет `__hash__` - изменяемый объект. Python устанавливает `__hash__ = None`
- есть `__eq__`, есть `__hash__` - неизменяемый объект реализованный разработчиком
- нет `__eq__`, есть `__hash__` - запрещённая комбинация! Разработчик допустил ошибку

Если вы хотите явно отключить поддержку хэширования, в определение класса добавляется строка `__hash__ = None`

```
class Triangle:
    __hash__ = None
    ...
```

Простейшая реализация хэша

При желании реализовать собственный метод `__hash__` рекомендуется сделать все свойства класса неизменяемыми. Внутри дандер метода возвращается результат работы функции `hash()`. На вход функция получает кортеж из всех свойств класса.

```
from math import sqrt

class Triangle:
    def __init__(self, a, b, c):
        self._a = a
        self._b = b
        self._c = c

    def __str__(self):
        return f'Треугольник со сторонами: {self._a}, {self._b}, {self._c}'

    def __repr__(self):
        return f'Triangle({self._a}, {self._b}, {self._c})'

    def __eq__(self, other):
        first = sorted((self._a, self._b, self._c))
        second = sorted((other._a, other._b, other._c))
        return first == second

    def area(self):
        p = (self._a + self._b + self._c) / 2
        _area = sqrt(p * (p - self._a) * (p - self._b) * (p - self._c))
        return _area

    def __lt__(self, other):
        return self.area() < other.area()

    def __hash__(self):
        return hash((self._a, self._b, self._c))

triangle_set = {Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4, 4, 4), Triangle(3, 5, 3)}
print(triangle_set)
print(', '.join(f'{hash(item)}' for item in triangle_set))
```

Свойства класса получили символ подчёркивания в начале имени. Сообщаем коллегам по коду, что они защищённые и не должны изменяться.



Важно! Как вы помните это договорённость, которую можно обойти. Философия Python надеется на разумного человека, который понимает что делать, а что нет.

Сам дандер `__hash__` возвращает результат вычисления хэша для кортежа из трёх элементов — длин сторон.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class MyClass:

    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.c = a + b

    def __str__(self):
        return f'MyClass(a={self.a}, b={self.b}, c={self.c})'

    def __eq__(self, other):
        return (sum((self.a, self.b)) - self.c) == (sum((other.a,
other.b)) - other.c)

x = MyClass(42, 2)
y = MyClass(73, 3)
print(x == y)
```

6. Обработка атрибутов

Python имеет четыре дандер метода, которые позволяют контролировать обращения к атрибуту экземпляра. Разберём их на простом примере класса вектор.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

a = Vector(2, 4)
```

Получение значения атрибута, __getattr__

Дандер __getattr__ вызывается при любой попытке обращения к атрибутам экземпляра.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y


    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не в пространстве')
        return object.__getattr__(self, item)
```

```
a = Vector(2, 4)
print(a.z) # AttributeError: У нас вектор на плоскости, а не в пространстве
print(f'{a = }')
```

В параметр `item` попадает имя атрибута, к которому пытаются обратиться в виде `str`. Мы прописали проверку имён и если это третья координата `z`, вызываем ошибку `AttributeError`.

 **Важно!** Строка `return object.__getattr__(self, item)` является обязательной. Без неё может возникнуть ошибка переполнения стека.

Присвоение атрибуту значения, `__setattr__`

Дандер `__setattr__` срабатывает каждый раз, когда в коде есть операция присвоения. Слева от знака равно экземпляр со свойством - `key`, справа значение - `value`.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не в пространстве')
        return object.__getattr__(self, item)

    def __setattr__(self, key, value):
        if key == 'z':
            raise AttributeError('У нас вектор на плоскости, а не в пространстве')
```

```

в пространстве')
    return object.__setattr__(self, key, value)

a = Vector(2, 4)
print(a.z)    # AttributeError: У нас вектор на плоскости, а не в
               пространстве
print(f'{a = }')
a.z = 73      # AttributeError: У нас вектор на плоскости, а не в
               пространстве
a.x = 3
print(f'{a = }')

```

Дандер `__setattr__` запрещает присваивать значение свойству. Как и в случае с `__getattr__` важная последняя строка. Она позволяет избежать рекурсии и присвоить значение свойству, которое мы не обработали ранее.

Обращение к несуществующему атрибуту, `__getattr__`

Если свойство отсутствует, в первую очередь вызывается дандер `__getattr__`. В случае возврата им ошибки `AttributeError` вызывается метод `__getattr__`. Он также может поднять ошибку. А может как-то иначе обработать запрос.

```

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

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
        return object.__getattr__(self, item)

```



```

def __setattr__(self, key, value):
    if key == 'z':
        raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
    return object.__setattr__(self, key, value)

def __getattr__(self, item):
    return None

a = Vector(2, 4)
print(a.z)  # None
print(f'{a = }')

```

Мы пропасаила возврат None для любого свойства, которое не удалось найти. Метод возвращает None и для свойства z, перехватывая исключение.

Удаление атрибута, __delattr__

При попытке удалить атрибут командой del можно использовать дандер __delattr__ для изменения логики.

```

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

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
        return object.__getattr__(self, item)

    def __setattr__(self, key, value):

```

```

        if key == 'z':
            raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
        return object.__setattr__(self, key, value)

    def __getattr__(self, item):
        return None

    def __delattr__(self, item):
        if item in ('x', 'y'):
            setattr(self, item, 0)
        else:
            object.__delattr__(self, item)

a = Vector(2, 4)
a.s = 73
print(f'{a = }, {a.s = }')
del a.x
del a.s
print(f'{a = }, {a.s = }')

```

В нашем классе при попытке удалить `x` или `y`, значение не удаляется. Вместо этого свойству присваивается ноль.

Обратите внимание на свойство `s`. Мы смогли присвоить ему значение 73. Дандер `__setattr__` контролирует только имя `z`. При удалении свойства `z` сработала ветка `else` и свойство было удалено. Однако мы не получили ошибки, обращаясь к несуществующему свойству, сработал дандер `__getattr__`.

Функции `setattr()`, `getattr()` и `delattr()`

В примере выше мы вызвали функцию `setattr` для присвоения у объекта `self` свойству `item` значения 0. В Python есть функции, которые позволяют делать тоже самое, что и рассмотренные выше дандер методы. Разница лишь в том, что метод реагирует на событие в коде, а функцию вы вызываете в тот момент, когда вам это нужно.

- `setattr(object, name, value)` — аналог `object.name = value`
- `getattr(object, name[, default])` — аналог `object.name` or `default`
- `delattr(object, name)` — аналог `del object.name`

Вывод

На этой лекции мы:

1. Разобрались с созданием и удалением классов
2. Узнали о документировании классов
3. Изучили способы представления экземпляров
4. Узнали о возможностях переопределения математических операций
5. Разобрались со сравнением экземпляров
6. Узнали об обработке атрибутов

Краткий анонс следующей лекции

1. Разберёмся в превращении объекта в функцию
2. Изучим способы создания итераторов
3. Узнаем о создании менеджеров контекста
4. Разберемся в превращении методов в свойства
5. Изучим работу дескрипторов
6. Узнаем о способах экономии памяти

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий и попробуйте перенести переменные и функции в класс. Добавьте к ним дандер методы из лекции для решения изначальной задачи.