[**TODO TABLE** 3](#_Toc110029480)

[***Python*** 4](#_Toc110029481)

[**Иммутабельность типов данных** 4](#_Toc110029482)

[***if \_\_name\_\_ == ‘\_\_main\_\_’*** 4](#_Toc110029483)

[***List/dict/set comprehension*** 4](#_Toc110029484)

[**Контекстный менеджер** 5](#_Toc110029485)

[**Создание своего контекстного менеджера** 5](#_Toc110029486)

[**Отличия кортежа от списка** 5](#_Toc110029487)

[**Функция *map*** 5](#_Toc110029488)

[**Функция *filter*** 5](#_Toc110029489)

[**Различия методов *type()* и *isinstance()*** 6](#_Toc110029490)

[**Различия *\_\_new\_\_* и *\_\_init\_\_*** 6](#_Toc110029491)

[**Протокол итерации (итерируемый объект)** 7](#_Toc110029492)

[**Итераторы** 7](#_Toc110029493)

[**Генераторы** 8](#_Toc110029494)

[**Отличия генераторов от итераторов** 9](#_Toc110029495)

[**\*args и \*\*kwargs** 10](#_Toc110029496)

[**Виртуальное окружение** 11](#_Toc110029497)

[***GIL*** 11](#_Toc110029498)

[***is, id, ==*** 11](#_Toc110029499)

[**Индексатор класса** 12](#_Toc110029500)

[**Поверхностное и глубокое виды копирования объектов** 12](#_Toc110029501)

[**Пакеты и модули** 13](#_Toc110029502)

[**Чистая функция** 13](#_Toc110029503)

[**Именованные кортежи (*namedtuple*)** 14](#_Toc110029504)

[**Магические атрибуты и методы** 14](#_Toc110029505)

[**Миксины** 15](#_Toc110029506)

[***MRO* (порядок разрешения методов)** 15](#_Toc110029507)

[***classmethod, staticmethod*** 16](#_Toc110029508)

[**Абстрактный класс** 17](#_Toc110029509)

[**Метаклассы** 17](#_Toc110029510)

[**Хеш-таблицы (словари) в *Python*** 19](#_Toc110029511)

[**Множества в *Python*** 19](#_Toc110029512)

[***pep8*** 19](#_Toc110029513)

[***pylint*** 19](#_Toc110029514)

[***Django*** 20](#_Toc110029515)

[***Model-View-Template* (*MVT*, архитектурный подход *Django*)** 20](#_Toc110029516)

[**Оптимизация запросов (*select\_related*, *prefetch\_related*)** 20](#_Toc110029517)

[***Middleware*** 21](#_Toc110029518)

[**Наследование моделей** 21](#_Toc110029519)

[**Базы данных** 22](#_Toc110029520)

[**Схемы (зачем нужны)** 22](#_Toc110029521)

[**Виды соединений таблиц** 22](#_Toc110029522)

[**Индексы** 23](#_Toc110029523)

[**Оптимизация запросов** 24](#_Toc110029524)

[***ACID* (свойства транзакционной СУБД)** 27](#_Toc110029525)

[**Уровни изолированности транзакций** 27](#_Toc110029526)

[**Репликация** 29](#_Toc110029527)

[**Денормализация** 31](#_Toc110029528)

[**REST** 32](#_Toc110029529)

[**Идемпотентность *HTTP* методов** 32](#_Toc110029530)

**TODO TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| Docker | Poetry | Django+debug toolbar+celery+redis |  |
| миксины | Магические методы | GIL описать | Asyncio + корутины |
| SOLID | Тестирование (pytest и mock) | Метаклассы | REST/restful/DRF |
| Брокеры сообщений | Расписать важные вопросы на собесах по видосам на ютубе | CI/CD | секционирование шардирование |

***Python***

## **Иммутабельность типов данных**

Тип переменной *a* **иммутабелен**, если после перезаписи значения переменной изменился ее идентификатор (*id(a)*). Изменение идентификатора указывает на то, что после перезаписи значения создался новый объект, следовательно, старый объект нельзя было изменить.

## ***if \_\_name\_\_ == ‘\_\_main\_\_’***

Данное условие позволяет определить точку входа. По идее это делается для того, чтобы при импорте различных модулей их код не вызывался в текущем модуле (операция импорта инициирует выполнение кода импортируемого модуля). С проверкой внутри каждого из импортируемых модулей в текущем модуле вызовется только код текущего модуля.

Наглядный пример реализован в учебном проекте.

## ***List/dict/set comprehension***

***Comprehension*** – *pythonic* способ генерирования создания списков, множеств и словарей.

**Примеры**

list = [i \* i for i in range(10) if i % 2 == 0]

print(type(list))

dict = {k: k + 5 for k in range(5)}

print(type(dict))

set = {k for k in range(5)}

print(type(set))

>> <class 'list'>

>> <class 'dict'>

>> <class 'set'>

## **Контекстный менеджер**

<https://devpractice.ru/python-lesson-21-context-manager/>

## **Создание своего контекстного менеджера**

Нужно реализовать протокол менеджера контекста одним из способов:

1. реализация методов \_\_*enter*\_\_ и \_\_*exit*\_\_.
2. использование модуля *contextlib* (через декоратор *@contextmanager*).

Пример есть в учебном проекте.

## **Отличия кортежа от списка**

Элементы списка можно редактировать, а кортежа – нет.

## **Функция *map***

**Функция** *map* применяет функцию к каждому элементу последовательности и **возвращает итератор** с результатами.

**Пример**

list\_of\_str = ['1', '2', '5', '10']

list = list(map(int, list\_of\_str))

print(list)

>> [1, 2, 5, 10]

Вместо *map* можно использовать *list comprehension*, но следует учитывать, что *map* **возвращает итератор**, а *list comprehension* – **список**. Если нужно сгенерировать большое количество элементов, то использование *map* выглядит уместнее в целях экономии памяти.

## **Функция *filter***

Функция *filter* применяет функцию ко всем элементам последовательности и **возвращает итератор** с теми объектами, для которых функция вернула *True*. Например, вернуть только те строки, в которых находятся числа:

list\_of\_strings = ['one', 'two', 'list', '', 'dict', '100', '1', '50']

list(filter(str.isdigit, list\_of\_strings))

>> ['100', '1', '50']

Из списка чисел оставить только нечетные:

list(filter(lambda x: x % 2 == 1, [10, 111, 102, 213, 314, 515]))

>> [111, 213, 515]

## **Различия методов *type()* и *isinstance()***

В отличие от *type*, функция *isinstance* **возвращает** не тип данных аргумента, а **булево значение**, говорящее о том, принадлежит объект к определенному классу или нет:

num = 4.44

print(isinstance(num, float))

>> True

Также отличие от *type* состоит в том, что *isinstance* **"знает" о наследовании**. Функция воспринимает объект производного класса, как объект базового. Поэтому нельзя использовать *type* для проверок наследующихся типов.

**Пример**

class BaseExample:

pass

class DerivedExample(BaseExample):

pass

test = DerivedExample()

print(isinstance(test, BaseExample))

>> True

## **Различия *\_\_new\_\_* и *\_\_init\_\_***

*\_\_new\_\_* **создает и возвращает объект класса**, он выполняется в первую очередь. В качестве первого параметра принимает ссылку на класс (*cls*).

*\_\_init\_\_* **отвечает лишь за инициализацию объекта класса**, он выполняется после *\_\_new\_\_* **и ничего не возвращает***.* В качестве первого параметра принимает имеет ссылку на объект (*self*).

## **Протокол итерации (итерируемый объект)**

**Итерируемый объект (*iterable*)** – это объект, возвращающий элементы по одному, а именно объект, **реализующий** **один из** методов: *\_\_iter\_\_* **или** *\_\_getitem\_\_* и метод *\_\_next\_\_.*

## **Итераторы**

**Итератор** – это объект, который может возвращать элементы последовательности по одному.

Технически же это любой объект, **реализующий метод** *\_\_next\_\_*, который должен вернуть **следующий элемент** или исключение *StopIteration,* если перечислены все элементы.

Также итератор **реализует метод** *\_\_iter\_\_*, **возвращающий итератор**. Если этого метода нет, функция *iter()* проверяет, нет ли метода *\_\_getitem\_\_* – метода, который позволяет получать элементы по индексу. Если метод *\_\_getitem\_\_* есть, **возвращается итератор**, который проходится по элементам, используя индекс (начиная с 0).  Если не реализован ни один из этих методов, тогда будет **вызвано исключение** *TypeError*.

**Пример**

numbers = [1, 2]

i = iter(numbers)

next(i)

>> 1

next(i)

>> 2

next(i)

>> StopIteration Traceback (most recent call last) in () ----> 1 next(i) StopIteration:

Для того, чтобы итератор снова начал возвращать элементы, его надо **заново создать**.

**Итераторы полезны** тем, что они **отдают элементы по одному**. Например, при работе с файлом или любой последовательностью это полезно тем, что **в памяти будет находиться** не вся последовательность, а **только одно текущее значение** итератора.

## **Генераторы**

В языках программирования есть такие понятия, как ленивые/отложенные вычисления (*lazy evaluation*). Генераторы можно считать реализацией механизма отложенного вычисления.

Генератор генерирует значения, возвращающиеся по запросу, и после возврата одного значения выполнение функции-генератора приостанавливается до запроса следующего значения. Между запросами **генератор сохраняет свое состояние**.

Python позволяет **создавать** генераторы **двумя способами**:

1. **генераторное выражение**;
2. **функция-генератор**.

**Генераторное выражение** использует такой же синтаксис, как *list comprehension* (но используются скобки круглые), но **возвращает итератор**, а не список:

genexpr = (x\*\*2 for x in range(1,5))

next(genexpr)

>> 1

next(genexpr)

>> 4

next(genexpr)

>> 9

**Генераторные функции** – это функции, где есть хотя бы одно выражение *yield*. При запуске генератора, функция выполняется до первого выражения *yield*. Генератор при этом встанет «на паузу» до следующей итерации. При следующей итерации выполнение генератора продолжится до очередного *yield*:

def f\_gen(m):

s = 1

for n in range(1,m):

yield n\*\*2 + s

s += 1

a = f\_gen(5)

for i in a:

print(i)

>> 2

>> 6

>> 12

>> 20

Еще одна особенность, о которой нужно помнить: в генератор можно посылать значения с помощью метода *send*:

def gen():

while True:

x = yield

if x == 0:

yield 99

continue

print(x)

g = gen()

next(g)

g.send(10)

print(g.send(0))

>> 10

>> 99

## **Отличия генераторов от итераторов**

**Итератор** – более общая концепция, чем генератор. Можно сказать, что генератор является итератором, но не наоборот.

Главной целью **итератора** является предоставление интерфейса для поочередного доступа к элементам коллекций и потоков данных.

Главная задача **генератора** – хранить закономерность, по которой генерируется последовательность.

## **\*args и \*\*kwargs**

***\*args*** – кортеж параметров (является типом *tuple*), позволяющий передавать в функцию неопределенное количество аргументов:

def getsum(\*numbers):

sum = 0

for num in numbers:

sum += num

return sum

print(add(2, 3))

print(add(2, 3, 5))

>> 5

>> 10

***\*\*kwargs*** – то же самое, что и \*args, но обязывает проставлять имена параметров и является типом *dictionary*. Позволяет передавать параметры в произвольном порядке:

def total\_fruits(\*\*fruits):

total = 0

for amount in fruits.values():

total += amount

return total

print(total\_fruits(banana=5, mango=7, apple=8))

print(total\_fruits(banana=5, mango=7, apple=8, oranges=10))

## **Виртуальное окружение**

Виртуальное окружение помогает изолировать зависимости проекта в отдельном пространстве (вместо глобального окружения ОС), привязанном к конкретному проекту. Это позволяет не конфликтовать зависимостям разных версий при наличии множества больших проектов на одной машине.

Также виртуальное окружение можно использовать для хранения приватных данных, необходимых для работы проекта (*API*-ключи, различные переменные, пароли и т.д.). Для этого можно создавать **переменные окружения** и получать их значения в конфигах приложения.

Поддержка виртуальных окружений есть в Python по умолчанию (начиная с версии 3.3) в виде *venv*.

**Примеры** использования на видео:

<https://www.youtube.com/watch?v=Y9MRCxq4DIc&ab_channel=%D0%94%D0%B8%D0%B4%D0%B6%D0%B8%D1%82%D0%B0%D0%BB%D0%B8%D0%B7%D0%B8%D1%80%D1%83%D0%B9%21>

## ***GIL***

<https://tproger.ru/translations/global-interpreter-lock-guide/>

## ***is, id, ==***

Ниже приведен наглядный **пример**

# определим три одинаковых списка

list\_1 = ['a', 'b', 'c']

list\_2 = list\_1

list\_3 = list(list\_1)

print(list\_1) # >> ['a', 'b', 'c']

print(list\_2) # >> ['a', 'b', 'c']

print(list\_3) # >> ['a', 'b', 'c']

# list\_1 равен list\_2 и по значению и как объект, т.к. оба указывают на один и тот же адрес в памяти

# (это можно проверить оператором is или сравнить id (эквивалентные способы))

print(list\_1 == list\_2, id(list\_1) == id(list\_2), list\_1 is list\_2)

>> True True True

# list\_2 равен list\_3 только ПО ЗНАЧЕНИЮ (==). А как объекты они разные

print(list\_2 == list\_3, id(list\_2) == id(list\_3), list\_2 is list\_3)

>> True False False

# добавим элемент в list\_1 и посмотрим, что станет с list\_2

list\_1.append('n')

print(list\_1)

>> ['a', 'b', 'c', 'n']

# list\_2 также изменится, потому что является ссылкой на list\_1

print(list\_2)

>> ['a', 'b', 'c', 'n']

# а list\_3 не изменится

print(list\_3)

>> ['a', 'b', 'c']

## **Индексатор класса**

Нужно реализовать метод *\_\_getitem\_\_* (см. пример в учебном проекте).

## **Поверхностное и глубокое виды копирования объектов**

*copy.copy(x)* – возвращает поверхностную копию *x*.

*copy.deepcopy(x)* – возвращает глубокую копию *x*.

Разница между **поверхностным** и **глубоким** копированием существенна только для **составных** объектов, содержащих изменяемые объекты (например, список списков, или словарь, в качестве значений которого – списки или словари).

**Пример**

import copy

test\_1 = [0, [1, 2]]

test\_copy = copy.copy(test\_1) # поверхностная копия

print(test\_1, test\_copy)

>>> [0, [1, 2]] [0, [1, 2]]

test\_copy[1].append(3)

print(test\_1, test\_copy)

>> [0, [1, 2, 3]] [0, [1, 2, 3]]

# тройка добавилась и туда и туда, т.к. поверхностное копирование

не проводит копирование вложенных объектов

test\_1 = [0, [1, 2]]

test\_deepcopy = copy.deepcopy(test\_1) # глубокая копия

test\_deepcopy[1].append(3)

print(test\_1, test\_deepcopy)

>> [0, [1, 2]] [0, [1, 2, 3]]

## **Пакеты и модули**

**Модуль** представляет собой **один файл** *Python*, который можно импортировать.

**Пакет** состоит из нескольких файлов/модулей *Python*. По сути это директория, которая может содержать поддиректории и модули. Особенностью является также то, что **пакет** **должен** **содержать пустой файл** с названием *\_\_init\_\_.py.*

## **Чистая функция**

Чистая функция – функция, удовлетворяющая следующим критериям:

1. **всегда возвращает одно** и то же **значение** **для одних и тех же** значений **входных** **параметров**;
2. **не имеет побочных эффектов** (не влияет на код, содержащийся вне функции. Например, не меняет значения глобальных переменных, не использует потоки ввода-вывода).

В общих чертах это функция, возвращаемое значение которой зависит исключительно от входных данных.

**Преимущества**:

* читаемость;
* портативность;
* легкая тестируемость.

## **Именованные кортежи (*namedtuple*)**

Именованный кортеж – это то же самое, что и обычный кортеж, но с той разницей, что теперь к элементу кортежа можно будет обращаться по имени, что улучшает читабельность кода (фактически это будет аналогом перечисления *Enum* в *C#*).

**Пример**:

from collections import namedtuple

# 1 параметр – имя типа, далее – список имен элементов кортежа

States = namedtuple('States', 'In\_Progress Open')

states = States(1, 4)

print(states)

>> States(In\_Progress=1, Open=4)

## **Магические атрибуты и методы**

**Магические методы** – методы, вызывающиеся неявно при выполнении других определенных методов. Например, в случае приведения к строке объекта некоторого класса вызывается магический метод \_\_*str*\_\_, если он переопределен в классе. **Наименования магических методов обрамлены с обеих сторон двумя подчеркиваниями**.

\_\_***dict***\_\_ – словарь, хранящий пользовательские атрибуты объекта и его значения.

\_\_***slots***\_\_ – атрибут, позволяющий ограничить набор атрибутов класса.

Если \_\_*slots*\_\_ реализован, объект станет занимать меньше памяти, т.к. магический атрибут \_\_*dict*\_\_ не будет создаваться. Пример работы с этим атрибутом есть в учебном проекте.

\_\_***new***\_\_ – метод, создающий объект класса, принимаемого в качестве параметра. После выполнения отдает объект методу \_\_***init***\_\_, \_\_*init*\_\_ в свою очередь инициализирует атрибуты объекта.

## **Миксины**

**Миксины/классы-примеси** – концепция, облегчающая создание сложных иерархий наследования классов. Применение миксин возможно только при наличии множественного наследования в языке.

Идея заключается в том, что **класс-примесь предоставляет небольшие функциональные возможности** для другой сущности, при этом желательно, чтобы класс-примесь не наследовался от других сущностей. Это облегчает создание и поддержку иерархии наследования классов.

В *Python* **нет специального синтаксиса для поддержки миксинов**, поэтому визуально миксины представляют из себя обычные классы. Но главное в том, что *Python* позволяет реализовывать множественное наследование, что и позволяет наследоваться целевому классу от множества классов-примесей.

## ***MRO* (порядок разрешения методов)**

*MRO* представляет из себя **алгоритм построения линейного вида иерархии класса для разрешения конфликтных ситуаций при множественном наследовании классов**. Разумеется, что это актуально тогда, когда класс-наследник не имеет собственной реализации метода.

Чтобы увидеть линейный вид иерархии и понять приоритетность каждой сущности в иерархии наследования, можно распечатать результат вызова **у класса** метода *mro*() или магического атрибута \_\_*mro*\_\_. Как работает *MRO* можно рассмотреть на примере с комментариями ниже. Вкратце на типовые вопросы «в каком порядке выведутся принты в коде..» можно дать ответ, что они выведутся в обратном порядке *MRO*, то есть в обратном порядке по очереди наследования в коде (см. пример ниже).

**Пример**

class A:

def \_\_init\_\_(self):

print('class A')

class B(A):

def \_\_init\_\_(self):

super().\_\_init\_\_()

print('class B')

class C(A):

def \_\_init\_\_(self):

super().\_\_init\_\_()

print('class C')

class Test(B, C):

def \_\_init\_\_(self):

super().\_\_init\_\_()

print('class Test')

# вывод будет идти в обратном порядке по MRO, т.е. в обратном порядке по очереди наследования: Test(B, C)...A. Получаем вывод A -> C -> B -> Test

Test()

print(Test.mro())

>> class A

class C

class B

class Test

[<class '\_\_main\_\_.Test'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

Еще один пример с механикой работы *MRO* реализован в учебном проекте.

## ***classmethod, staticmethod***

Метод класса, объявленный с декоратором ***classmethod***, принимает на вход в качестве первого параметра *cls*, и может изменять только состояние полей, относящихся к классу (полей, объявленных вне инициализатора \_\_*init*\_\_). Т.е. этот метод связан с классом, а не объектом, но объект класса видит такие методы.

Метод класса, объявленный с декоратором ***staticmethod***, является обычным статическим методом, т.е. он не имеет доступа к классу, его объектам и его состоянию. Такой метод уместно внести в класс, если он имеет какое-то отношение к классу и к контексту сущности, описанной в классе.

Пример реализован в учебном проекте.

## **Абстрактный класс**

Абстрактный класс – класс, у которого нельзя создать объект..

Для реализации в *Python* нужно импортировать из модуля *abs* класс *ABC* и метод *abstractmethod*. Целевой класс нужно унаследовать от *ABC*, а абстрактные методы объявить с декоратором @*abstractmethod*.

Если у абстрактного метода не объявить вышеупомянутый декоратор, то данный метод необязательно переопределять в наследниках (работает как *virtual*-методы в *C#*). А если декоратор прописан, то метод обязательно должен быть переопределен в наследниках. Пример реализован в учебном проекте.

## **Метаклассы**

Для начала нужно понимать, что абсолютно **все сущности** в языке *Python* **являются** **объектами**, включая классы.

Для определения типа класса можно вызвать метод *type* и увидеть, что метод возвращает <class 'type'>. То есть, **любой объект в *Python* имеет тип *type***.

Можно сказать, что **метакласс – это сущность, объекты которой являются классами**. Например, *type* является метаклассом и может работать в двух режимах, в первом из которых *type* принимает 1 аргумент – тип объекта, и возвращает его базовый тип, а во втором *type* принимает 3 аргумента: *type(name, bases, attrs)*, где:

* ***name***: название класса;
* ***base***: кортеж родительских классов (может быть пустым);
* ***attrs***: словарь, содержащий имена и значения атрибутов.

Используя *type* во втором виде, можно динамически создавать классы с атрибутами.

Также можно создать свой метакласс, наследуясь от *type*, и в определении целевого класса указать metaclass=<имя метакласса> (пример есть в учебном проекте).

На практике с метаклассом можно столкнуться, например, создавая класс собственной модели в *Django*.

В сообществе *Python* бытует высказывание от одного эксперта, смысл которого состоит в том, что обычный программист вряд ли когда-нибудь столкнется с необходимостью написания собственных метаклассов.

Примеры приведены в учебном проекте.

## **Хеш-таблицы (словари) в *Python***

Для каждого ключа вычисляется хеш с помощью встроенной функции *hash*. Вычисленный хеш используется для определения места (индекса) в хеш-таблице. Таким образом алгоритмическая сложность получения любого значения по ключу составляет *O(1)*.

**Ключами словаря** в *Python* **могут являться только хешируемые объекты** (**значения неизменяемых типов данных**).

## **Множества в *Python***

Множество в *Python* – структура неупорядоченных данных. Особенности множества:

* элементы множества не индексируются;
* множество не содержит дубликаты элементов;
* множество содержит только значения неизменяемых типов данных.

## ***pep8***

Пакет, чекающий код на наличие ошибок согласно код-стайлу *pep8*.

**Установка**: *pip install pep8*

**Проверить** **код (нужно находиться в директории файла)**: *pep8 file.py*

## ***pylint***

**Установка**: *pip install pylint*

**Интеграция с *PyCharm*:** переходим в *Settings-Tools-External Tools* и добавляем новый тулз. ***Name***: *pylint*, в ***Program*** указываем путь к .*exe*-файлу *pylint* из папки виртуального окружения *venv/Scripts*. Задаем значения для ***Parameters***: *$FileName$*, для ***Working*** ***directory***: *$FileDir$*.

Использовать можно при помощи ПКМ на файле – *External Tools* – *pylint*.

***Django***

## ***Model-View-Template* (*MVT*, архитектурный подход *Django*)**

*MVT* включает в себя три составляющие (не путать с аналогичными в модели *MVC*):

* ***M*** – модели, описывающие сущности приложения. Представляют собой классы, соответствующие таблица в БД;
* ***V*** – представления (или контроллеры из *MVC*). Данный слой связывает слои *Model* и *Template*. Конкретное представление должно быть связано с соответствующим маршрутом (*url*), при необходимости оно может обращаться к слою Model, также представление решает какой темплейт нужно отрендерить в браузере по конкретному запросу.
* ***T*** – темплейты, представляющие визуальный слой приложения *HTML*-страницы, фронтенд).

*Django* не предполагает наличия отдельного слоя бизнес-логики, поэтому этот слой лучше реализовать самостоятельно в виде отдельных пакетов и модулей.

## **Оптимизация запросов (*select\_related*, *prefetch\_related*)**

Рассмотрим пример: есть две связанные модели – **жанр** и **книга**, связь *1-N*. Есть **страница, выводящая информацию об *N* книгах** из БД **по определенному жанру**. На странице мы хотим видеть название жанра около каждой книги.

По умолчанию приложение отправило бы *N* запросов к таблице жанров для получения названия жанра, что очень неоптимально.

Эту проблему и решает *select\_related*. С ним мы делаем 1 запрос и используем информацию из него *N* раз, а без него делаем *N* запросов. **Его можно использовать с моделями, не связанными связью *N-N***.

*prefetch\_related* нужен в той же ситуации, что и *select\_related*, но в случае наличия связи *N-N.*

Влияние на количество запросов от данных методов наглядно видно в секции *SQL DebugToolbar.*

## ***Middleware***

*Middleware* – **промежуточный программный слой**, позволяющий обрабатывать запросы из браузера, прежде чем они достигнут слоя представления *Django*, а также ответы от представлений до того как они попадут в браузер.

Путь запроса от браузера до *view*: ***Browser – M1 – M2 … M..N – View***.

Путь запроса от *view* до браузера: ***View – M..N … M2 – M1 – Browser***.

## **Наследование моделей**

*Django* позволяет реализовать 4 вида наследования: **абстрактная модель, мультитабличное наследование, модель-прокси** и **множественное наследование**.

**Абстрактная модель** позволяет разместить внутри себя некоторый общий код и использовать его в моделях-наследниках. **Таблицы в БД создаются ТОЛЬКО для моделей-наследников**. Данный вид наследования реализуется с помощью указания в подклассе *Meta* атрибута *abstract* = *True*.

**Мультитабличное наследование** позволяет создать полноценную иерархию наследования, в рамках которой д**ля ВСЕХ моделей будет создана таблица** **в БД**. Между моделями-наследниками и родительской моделью создается неявная связь вида *OneToOneField*. Данный вид наследования реализуется аналогично наследованию обычных классов *Python*.

**Модель-прокси**, унаследования от какой-либо другой модели, **позволяет дополнить поведение родительской модели, но не ее структуру**. Для этого в подклассе *Meta* дочерней модели нужно указать атрибут *proxy = True*, чтобы эта модель стала прокси. **Для моделей-прокси таблицы в БД не создаются**.

**Множественное наследование** моделей реализуется также как и множественное наследование обычных классов *Python*.

**Базы данных**

## **Схемы (зачем нужны)**

Есть несколько возможных объяснений, для чего стоит применять схемы:

* чтобы объединить объекты БД в логически связанные группы для облегчения управления ими.
* чтобы в одной базе сосуществовали разные приложения, и при этом не возникало конфликтов имён.

## **Виды соединений таблиц**

1. *INNER JOIN* – **внутреннее** соединение. Объединяет из соединяемых таблиц только те строки, для которых выполняется условие *ON*.
2. *LEFT JOIN* – **левостороннее** соединение. Строки из **левой** таблицы в любом случае попадут в результирующую выборку.
3. *RIGHT JOIN* – **правостороннее** соединение. Строки из **правой** таблицы в любом случае попадут в результирующую выборку.
4. *CROSS JOIN* – **перекрестное** соединение. Каждая запись левой таблицы объединяется с каждой записью правой таблицы (декартово произведение).

## **Индексы**

Индекс БД – объект, создаваемый с целью повышения производительности поиска данных. Индекс хранит ссылки на страницы, в которых лежат записи.

Индексы могут наоборот уменьшить производительность, если применяются в таблицах, где часто происходят операции *INSERT*, *DELETE* и *UPDATE*, т.к. нужно актуализировать состояние самих индексов.

Также нужно помнить о том, что **индексы занимают дополнительный объем памяти**, т.к. хранятся на диске.

|  |  |
| --- | --- |
| **Индекс** | **Кейсы для применения** |
| ***btree*** – **сбалансированное дерево**. Является индексом по умолчанию*.* Наиболее часто применяемый вид индекса | Любые данные, с которыми можно проводить операции сравнения |
| ***hash*-индексы** – хеширование значений индексированного поля таблицы | Работает только с простыми условиями равенства  Используется *PostgreSQL* для соединения таблиц, если у таблиц нет индексов. СУБД временно создает хеш-индексы для таблиц, соединяет их и затем удаляет хеш-индексы. |
| ***GiST (R-Tree)*** | Позволяет индексировать значения, перекрывающиеся между собой. Это могут быть значения многомерных структур хранения данных типа координат в многомерном пространстве  Индексация географических типов данных |
| ***GIN* (инвертированный индекс)** | Полнотекстовый поиск, поиск по *JSON*-полям |
| **Частичный индекс** | Применяется, когда нет цели индексировать данные целиком.  **Пример**: *CREATE INDEX idx\_table\_field ON table (field) WHERE field IS NOT NULL*  **Недостаток**: чтобы индекс применялся, необходимо в запросы «таскать» выражение, указанное при создании индекса (в примере это выражение *IS NOT NULL*). |
| **Функциональный индекс** | Применяется, когда есть необходимость проиндексировать не сами значения, а результаты функций от этих значений.  **Пример**: *CREATE INDEX idx\_table\_field ON table (LOWER(field))*  **Недостаток**: чтобы индекс применялся, необходимо в запросы «таскать» выражение, указанное при создании индекса (в примере это выражение *LOWER(field)*). |
| **Кластерный индекс** | Кластеризация сортирует данные по указанному индексу. То есть она применяется к таблице и ранее созданному индексу  Кластеризация является одноразовой операцией: последующие изменения в таблице нарушают порядок кластеризации |

## **Оптимизация запросов**

Сначала нужно понять к каким данным СУБД обращается чаще всего и какие запросы выполняются дольше всего.

**Долгое выполнение запросов может быть связано со внешними факторами**: например, в момент выполнения запроса выполнялось резервное копирование или в то же время сервер обрабатывал другие тяжелые запросы. **То есть дело может быть не в самом запросе.**

**1 шаг (статистика)**: собрать статистику выполнения запросов, например, с помощью расширения ***pg\_stat\_statements***. Сначала нужно активировать расширение в файле конфигурации ***postgresql.conf***, добавив в него строки:

shared\_preload\_libraries = ‘pg\_stat\_statements’

# **соберет статистику по 1000 наиболее используемых запросов**

pg\_stat\_statements.max = 1000

pg\_stat\_statements.track = all

**Обращение к статистике**:

SELECT query, calls, total\_time, rows,

100.0 \* shared\_blks\_hit / nullif(shared\_blks\_hit + shared\_blks\_read, 0) AS hit\_percent

FROM pg\_stat\_statements

ORDER BY total\_time DESC LIMIT 10;

**1 шаг (альтернативный способ: лог)**: подключить в ***postgresql.conf*** логирование запросов, время выполнения которых превышает заданную величину в миллисекундах:

log\_duration = on

log\_lock\_waits = on

# время выполнения (в мс.), превысив которое запрос попадает в лог

log\_min\_duration\_statement = 100

log\_filename = ‘postgresql-%Y-%m-%d\_%H%M%S’

log\_directory = ‘/var/log/postgresql’

log\_destination = ‘csvlog’

logging\_collector = on

Просматривать отчеты, сделанные на основе логов, можно с помощью утилиты ***pgbadger***, которую необходимо установить отдельно.

**2 шаг:** посмотреть план запроса с помощью операторов ***EXPLAIN ANALYZE* (**их можно использовать как вместе, так и раздельно, см. далее**)**.

***EXPLAIN*** построит и покажет только план выполнения, но не выполнит запрос.

***ANALYZE*** соберет статистику используемых таблиц, выполнит запрос и покажет время его выполнения.

**Пример**

EXPLAIN SELECT \* FROM tenk1;

QUERY PLAN

-------------------------------------------------------------

Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

Числа, перечисленные в скобках (слева направо), имеют следующий смысл:

* приблизительная **стоимость запуска** (время, спустя которое начнётся этап вывода данных);
* приблизительная **общая стоимость (*cost*). 1 *cost* = 1 операция чтения страницы с диска**;
* ожидаемое **число строк**, которое должен вывести этот **узел плана**;
* ожидаемый **средний размер строк**, выводимых этим узлом плана (в байтах).

Если требуется проанализировать с помощью *EXPLAIN* *ANALYZE* запросы типа *DELETE*, *UPDATE* и *INSERT*, то необходимо обернуть их в транзакцию, т.к. *ANALYZE* выполняет запрос:

BEGIN;

EXPLAIN ANALYZE <query>;

ROLLBACK;

Важно помнить о том, что при выполнении запроса **планировщик выбирает план выполнения**, **опираясь на** **статистику**. Следовательно, при анализе запроса путем использования оператора *EXPLAIN* **нужно убедиться, что статистика не устарела**.

**Собрать статистику** можно, вызвав *ANALYZE* с именем таблицы, или просто *ANALYZE*, чтобы проанализировать все таблицы БД.

**Шаг 3: анализ**: при анализе нужно обращать внимание на следующие моменты:

* операции с большой ***cost***;
* узлы плана с последовательным чтением (***seq scan***), часто указывающие на отсутствие необходимых индексов;
* ***sort*** – операции сортировки являются дорогими.

**Дальнейшие возможные шаги**:

* на сервере **можно запускать** **автоочистку и автоанализ** (***autovacuum***) для удаления неиспользуемых записей (мертвых кортежей);
* **денормализация** – привнесение избыточности;
* **использование** **табличных пространств** – управление размещением объектов по физическим устройствам ввода/вывода. Например, активно используемые данные хранить на *SSD* дисках, а архивные – на более медленных *HDD*.
* **секционирование/партиционирование** – разделение большой таблицы на множество более мелких;
* **замена подзапросов в запросах на соединения**.

## ***ACID* (свойства транзакционной СУБД)**

***Atomicity*** – **атомарность** операций. Гарантирует, что никакая [транзакция](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) не будет зафиксирована в системе частично. Будут выполнены все её операции, либо не выполнено ни одной.

***Consisitency*** – **консистентность, согласованность**. Гарантия целостности данных, выполнения определенных правил и ограничений.

***Isolation*** – **изолированность**. Выполнение параллельных транзакций не должно влиять на итоговый результат другой транзакции. На самом деле это очень дорогое требование, поэтому оно частично и по-разному выполняется в соответствии с уровнями изоляции транзакций.

***Durability*** – **долговечность**. Это свойство гарантирует устойчивость к ошибками. Изменения успешно завершенной транзакции не должны пропасть в случае сбоев.

## **Уровни изолированности транзакций**

Стандарт *SQL* определяет **4 уровня изоляции** транзакций.

Пусть *T1* – **первая транзакция**, а *T2* – **вторая** **транзакция**.

***L1. READ UNCOMMITTED*** («грязное» чтение) – *T1* **видит изменения**, выполненные **незавершенной** *T2*. Этот уровень не реализован в *PostgreSQL*, т.к. не представляет практической ценности.

***L2****\*****. READ COMMITTED*** (чтение зафиксированных данных) – *T1* увидит изменения, выполненные *T2*, после коммита (фиксации) *T2*.

Попытка изменения одних и тех же данных в *T2* до коммита изменений *T1* будет заблокирована. \***Является уровнем по умолчанию в *PostgreSQL***.

***L3. REPEATABLE READ*** (повторяемое чтение) – *T1* увидит изменения, выполненные *T2*, только после коммита *T2* и *T1*.

Попытка изменения одних и тех же данных в *T2* до коммита изменения в *T1* будет заблокирована.

***L4. SERIALIZABLE*** (сериализуемость) – уровень, на котором моделируется последовательное выполнение всех зафиксированных транзакций. Если транзакции обращаются к одной и той же таблице, зафиксируются изменения только той транзакции, которая зафиксируется быстрее всех.

Шаблон для наглядного представления и работы с уровнями изоляции (нужна заранее созданная БД с таблицей *test* и еще одним пользователем помимо *postgres*):

*T1* (сессия пользователя *postgres*):

-- READ COMMITTED

-- REPEATABLE READ

-- SERIALIZABLE

DELETE FROM test;

INSERT INTO test VALUES (1), (2);

SELECT \* FROM test

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

UPDATE test SET id = id + 10 WHERE id = 1;

COMMIT;

*T2* (сессия пользователя *test*):

SELECT \* FROM test

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

--- можно поменять id на 1, чтобы увидеть блокировки

UPDATE test SET id = id + 10 WHERE id = 2;

COMMIT;

## **Репликация**

**Репликация** – механизм синхронизации нескольких наборов данных.

Существуют два вида репликации: **физическая** и **логическая**. Оба вида будут описаны ниже.

Также существуют **гарантии** **репликации**: **синхронная** и **асинхронная.**

**Синхронная** репликация подразумевает, что мастер-сервер фиксирует транзакцию только после момента подтверждения получения данных репликой.

**Асинхронная** же подразумевает обратное: **мастер-сервер не ждет подтверждения данных** от реплики.

**По виду взаимодействия** репликация может быть **мастер-слейв**и **мастер-мастер.**

**Мастер-слейв репликация** предполагает главного (мастер) сервера и подчиненного (слейв) сервера. **Слейв повторяет состояние мастера** и не может менять данные самостоятельно.

**Репликация** **мастер-мастер предполагает равенство серверов**. Оба сервера могут обрабатывать запросы на чтение и изменение данных.

**Применение** репликации:

* **балансировка нагрузки**;
* **обеспечение отказоустойчивости** (переход на реплику в случае сбоя основного сервера);
* **тестирование новых версий *PostgreSQL***.

**Физическая** (**потоковая**) репликация – вид репликации, в основе которого лежит **передача записей журнала транзакций** от мастера к реплике. **Реплика применяет изменения**, принятые от мастера, **чисто механически**, без «понимания смысла», работая **в режиме постоянного восстановления**. Поэтому данный вид репликации можно применить только в условиях полной программно-аппаратной совместимости мастер и слейв серверов (одна архитектура *CPU* и версия *PostgreSQL*).

**Недостатки** физической репликации:

* мастер может отдать слейву испорченные данные, например, в случае сбоев *RAM*;
* на реплике не может быть локальных изменений схемы данных;
* требовательность к платформе и версии *PostgreSQL*;
* невозможность репликации мастер-мастер.

В основе **логической** репликации лежит принцип ***publisher*-*subscriber***, при этом у серверов нет выделенных ролей: **один и тот же сервер может публиковать изменения и быть подписчиком**.

Сервер-публикатор читает журнальные записи, но пересылает их подписчиком «как есть» (как при физической репликации), а предварительно декодирует в «логический», платформо-независимый вид. Поэтому логическая репликация не требует жесткой совместимости серверов, реплика должна лишь понимать протокол репликации.

**Логическая репликация не реплицирует** команды *DDL* (все изменения схемы данных нужно переносить вручную), значения последовательностей.

**Плюсы** логической репликации:

* репликация остановится, если мастер имеет аппаратные неисправности;
* нетребовательность к программной совместимости;
* возможность использования различающихся схем данных на мастере и слейве;
* возможность настройки мастер-мастер репликации.

## **Денормализация**

Под денормализацией понимают **намеренное нарушение требований нормализации**.

Денормализация обычно проводится путем добавления данных, которые согласно требования нормальных форм должны выноситься в отдельную таблицу (избыточность данных).

**Применение**: в случае долгого выполнения соединений в запросе можно **внести дополнительное поле** из присоединяемой таблицы в нужную таблицу.

Также можно построить так называемые **вертикальные таблицы**, в которых строки будут использоваться для хранения наименований полей и их значений:



Денормализация приводит к избыточности и дублированию данных, поэтому выигрыш от нее должен оправдывать затраты.

**REST**

## **Идемпотентность *HTTP* методов**

Метод *HTTP* является **идемпотентным**, если один и тот же неоднократно выполненный запрос оказывает один и тот же эффект, не изменяющий состояние сервера. Другими словами, идемпотентный метод не должен иметь никаких побочных эффектов (*side-effects*), кроме сбора статистики или подобных операций.

Методы [***GET***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/GET)***,***[***HEAD***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/HEAD)***,***[***PUT***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/PUT), [***DELETE***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/DELETE) и ***OPTIONS*** **идемпотентны** в отличие от методов [***POST***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/POST)и ***PATCH***.

Для идемпотентности нужно рассматривать только изменение фактического внутреннего состояния сервера, а возвращаемые запросами коды статуса могут отличаться: первый вызов [***DELETE***](https://developer.mozilla.org/ru/docs/Web/HTTP/Methods/DELETE) вернёт код [**200**](https://developer.mozilla.org/ru/docs/Web/HTTP/Status/200), а следующие вызовы вернут код [404](https://developer.mozilla.org/ru/docs/Web/HTTP/Status/404).