Attestation Questions & Answers

Теория

Опыт работы с СУБД, хранилищами данныхд

1. Транзакции, ACID, MVCC, WAL.
    1. ***Q*** : Перечислить требования ACID к транзакционной системе;

       A :

       A - **Atomicity** (атомарность) - никакая транзакция не будет зафиксирована частично.
       C - **Consistency** (согласованность) - транзакция фиксирующая свои результаты сохраняет согласованность базы данных.
       I - **Isolation** (изолированность) - во время выполнения транзакции параллельные транзакции не оказывают влияния на ее результат.
       D - **Durability** (долговечность) - независимо от проблем, изменения, сделанные завершенной транзакцией, должны остаться сохраненными после возвращения системы в работу.

    2. Каким средством реализовано выполнение этих требований в PostgreSQL?
       Наверно, речь идет о MVCC и WAL, лучше уточнить //◢
       Если правильно помню разговор с Денисом, надо рассказать про MVCC. Вот тут кратенько и понятно описано: Принципы работы СУБД. MVCC / ХабрахбрХабрахабр
       Если кратко: у PG есть глобальный реестр всех транзакций с их статусами, каждая транзакция пронумерована. Все это сбрасывается при каждом vacuum, во избежание рпереполнения 4-байтового целого, который используется для номера. У каждой записи в таблице есть 2 поля: Xmin - номер транзакции, которая ее создала, Xmax - номер транзакции которая ее удалила. Так же есть еще Cmin и Cmax, которые работают так же, только в рамках одной транзакции (номера операций). Для проверки валидности используется сложное условие завязанное на этих 5 параметрах (например запись не валидна, если Xmax больше номера текущей транзакции и транзакция с Xmax не откатилась).
       WAL(Write-Ahead Log) - это журнал, в который пишутся все операции на своем, внутреннем, языке (судя по всему типа «стереть строку 50 в таблице 64822») и из которого потом можно воссоздать состояние системы после сбоя. Пишутся всегда до того, как что-то сделать, и потом ставится определенная пометка.

    3. ***Q*** : Перечислить уровни изоляции транзакций. Отличие Read Committed от Repeatable Read. В каких случаях необходимо использовать Repeatable Read?
       A : **Read uncommitted** - В PostgreSQL фактически представляет из себя Read committed. **Read committed** - уровень по-умолчанию, SELECT-запрос видит только те данные, которые были закоммичены *перед* его запуском, плюс те данные, которые были добавлены *внутри* текущей транзакции. Т.е. SELECT-запрос *не видит* данные,

которые другие транзакции еще *не закоммитили*. **Repeatable read** - SELECT-запрос видит только те данные, которые были закоммичены *перед* стартом текущей транзакции, плюс те, которые были добавлены *внутри* текущей транзакции. Отличие от Read committed в том, что в случае Repeatable read SELECT-запросы не видят данные, которые были закоммичены *параллельными* транзакциями *после старта текущей*. **Serializable** - моделируется последовательное выполнение всех зафиксированных транзакций, как если бы транзакции выполнялись одна за другой, последовательно, а не параллельно.
https://postgrespro.ru/docs/postgrespro/9.5/transactioon-iso#TRANSACTION-ISO

4. ***Q*** : cap-теорема

   A : Утверждение о том, что в любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств:
   Consistency: **согласованность данных** - во всех вычислительных узлах в один момент времени данные не противоречат друг другу.
   Availability: **доступность** - любой запрос к распределённой системе завершается корректным откликом.
   Partition tolerance: **устойчивость к разделению** - расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

2. ***Q*** : Потеряются ли данные вследствие сбоя системы (например, отключение электричества). Какой механизм предоставляет PostgreSQL для предотвращения потери данных?

   A : При "стандартных" настройках данные не потеряются, т.к. транзакция будет считаться примененной только после записи данных на диск. См. WAL.

3. Индексы
   1. Для чего нужны индексы?
   2. ***Q*** : Будет ли прочитана вся таблица в случае запроса?

      ```
      SELECT * FROM table_name ORDER BY name LIMIT 1
      ```

      A : Зависит от того, есть ли по полю name индекс.

   3. ***Q*** : Что необходимо сделать, чтобы прочитана была только одна запись?

      A : Создать индекс по полю name.

   4. ***Q*** : Какие виды индексов существуют в PostgreSQL?

      A :

      B-tree - древовидный индекс, позволяет выполнять быстрый поиск по `<, <=, =, >=, >` . По-умолчанию создается именно этот индекс. Также может быть полезен при `IS NULL, IS NOT NULL` . При сопоставлении с образцом (`LIKE`) используется только если

шаблон привязан к началу строки, т.е. для `LIKE "abc%"` будет использован, при `LIKE "%abc"` - не будет. При сопоставлении с `ILIKE` (без учета регистра) будет использован, если строка начинается не с алфавитного символа (цифры, знаки).
Hash - индекс, позволяющий выполнять поиск за время O(1). Работает только с равенством (=). После аварийной остановки может быть поврежден и давать неверные результаты.
GiST, GIN - фактически представляет из себя не индекс, а инфраструктуру для индексов. Используются для геометрических данных, полнотекстового поиска, массивов.
BRIN - Фактически представляет из себя не индекс, а некие метаданные о данных в столбце внутри одного блока. Разработан для больших таблиц. При использовании индекса некоторые блоки, заведомо не имеющие подходящих данных, не будут сканироваться.

5. ***Q*** : Почему мы не используем по умолчанию индекс HASH у которого сложность поиска элемента O(const) а используем B-tree со сложностью O(log n)?
   A :
   1. Hash полезен только для оператора равенства (=).
   2. Hash требует пересборки после падения БД и может давать неверные результаты.

6. ***Q*** : Что из себя представляет B-tree индекс?
   A : B-дерево: сбалансированное сильно ветвистое дерево. Сбалансированность - длина любых путей от корня до листьев совпадает. Ветвистость - свойство каждого узла ссылаться на большое количество узлов-потомков. *// TODO: перенести в структуры данных --* ◢

4. Сборка мусора
   1. ***Q*** : Что делает операция VACUUM?
      A : VACUUM высвобождает пространство, занимаемое "мёртвыми" кортежами. При удалении и обновлении записей, кортежи физически не удаляются из таблицы. Они сохраняются в ней, пока не будет выполнен VACUUM.

   2. ***Q*** : Чем отличается VACUUM от VACUUM FULL?
      A : При VACUUM FULL происходит "полная" очистка, которая может освободить больше пространства, но выполняется гораздо дольше и запрашивает исключительную блокировку таблицы. Этот режим также требует дополнительное место на диске, так как он записывает *новую копию* таблицы и не освобождает старую до завершения операции. Обычно это следует использовать, только *когда требуется высвободить значительный объём пространства*, выделенного таблице. При использовании простого VACUUM пустое пространство, как правило, не возвращается операционной системе, а только помечается свободным для дальнейшего использования.

5. План запроса
    1. ***Q*** : Какую информацию можно увидеть при выполнении EXPLAIN запроса?

       A : "Стоимость" запроса, предполагаемое количество обработанных строк, предполагаемый размер строки. План запроса - каким образом будет происходить выборка данных.

    2. ***Q*** : Что означают cost, rows, width?

       A :

       Cost: два числа: 1) Приблизительная стоимость запуска. Это время, которое проходит, прежде чем начнётся этап вывода данных, например для сортирующего узла это время сортировки. 2) Приблизительная общая стоимость. Она вычисляется в предположении, что узел плана выполняется до конца, то есть возвращает все доступные строки. На практике родительский узел может досрочно прекратить чтение строк дочернего (LIMIT).
       Rows: Ожидаемое число строк, которое должен вывести узел плана.
       Width: Ожидаемый средний размер строк, выводимых узлом плана.

    3. ***Q*** : Что такое Seq Scan?

       A : Поиск непосредственно по таблице (вместо использования индексов).

    4. ***Q*** : Чем Seq Scan отличается от Index Scan?

       A : При Seq Scan происходит поиск данных непосредственно в таблице, при этом индекс не используется. При Index Scan выполняется поиск данных в индексе. В некоторых случаях даже при наличии индексов планировщик предпочитает Seq Scan, например при малом размере таблицы.

    5. ***Q*** : Для чего нужен параметр ANALYZE?

       A : При использовании ANALYZE помимо построения плана происходит выполнение самого запроса. В вывод EXPLAIN при этом добавляется статистика по реальному запросу (Actual time, execution time).

Структуры данных и алгоритмы

1. Структуры данных
    1. массивы
    2. связные списки
    3. словари
2. Поиск - привести примеры 2-х, 3-х алгоритмов рассказать как они работают.
3. Сортировка - привести пример 2-х, 3-х алгоритмов сортировки
4. Рассказать отличия алгоритмов обхода дерева в глубину и ширину
5. Временная и пространственная сложность алгоритма
    1. Оценка сложности произвольного алгоритма.
    2. Провести оценку сложности произвольного алгоритма, на выбор.
6. Графы
    1. Какие бывают разновидности графов?
    2. Какие алгоритмы могут применяться для поиска на графах?
7. Хеш-таблицы
    1. Как устроена Хеш-таблица?
    2. Что такое коллизии поему они могут возникать, и как разрешаются?
8. Деревья
    1. Что такое куча?
    2. Что такое сбалансированное дерево?
    3. Для чего могут быть нужны красно-черные деревья?

Навыки проектирования информационных систем

1. ООП
    1. Что такое ООП?
    2. Чем ООП отличается от других парадигм программирования?
    3. Какие парадигмы кроме ООП вы знаете и используете в своей практике?
    4. Инкапсуляция, полиморфизм, наследование
    5. Конструкторы, деструкторы
    6. SOLID
        1. Рассказать о принципе единственной обязанности
        2. Что такое инверсия зависимостей и для чего она нужна?
    7. Что такое слабая связность? Какие методы используются для достижения слабой связности в приложении.
    8. Что такое интерфейсы?
    9. Чем интерфейс отличается от абстрактного класса?
    10. Паттерны
        1. Какие обобщающие классы паттернов вы можете назвать?
        2. Что такое фабричный метод?
        3. Что возвращает абстрактная фабрика?
        4. Чем шаблон Наблюдатель отличается от шаблона Издатель-Подписчик?
        5. Какие паттерны применяются для организации доступа к данным?
    11. Как работает архитектура MVC
    12. Какие еще варианты реализации архитектурного каркаса приложения вы знаете?

Веб-протоколы

Криптография

Практика

Знание технологий и платформ

Python

1. ***Q*** : **Какие виды типизации существуют?**

    A : Python

    1. Языки программирования по типизации принято делить на два больших лагеря — *типизированные* и *нетипизированные* (*бестиповые*). К первому например относятся C, Python, Scala, PHP и Lua, а ко второму — язык ассемблера, Forth и Brainfuck.
    // Сюда можно внести shell? Там тоже нет типов, все данные строки. // ▲
// Наверное можно//

    Так как «бестиповая типизация» по своей сути — проста как пробка, дальше она ни на какие другие виды не делится. А вот типизированные языки разделяются еще на несколько пересекающихся категорий:

- ***Статическая*** / ***динамическая*** типизация. Статическая определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции. Т.е. уже компилятор на 100% уверен, какой тип где находится. В динамической типизации все типы выясняются уже во время выполнения программы.

  Примеры:
  Статическая: C, Java, C#;
  Динамическая: Python, JavaScript, Ruby.
- ***Сильная*** / ***слабая*** типизация (также иногда говорят ***строгая / нестрогая***). Сильная типизация выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество. Языки со слабой типизацией выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно.

  Примеры:
  Сильная: Java, Python, Haskell, Lisp;
  Слабая: C, JavaScript, Visual Basic, PHP.
- ***Явная*** / ***неявная*** типизация. Явно-типизированные языки отличаются тем, что тип новых переменных / функций / их аргументов нужно задавать явно. Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор.

  Примеры:
  Явная: C++, D, C#
  Неявная: PHP, Lua, JavaScript, Python

Также нужно заметить, что все эти категории пересекаются, например язык C имеет *статическую слабую явную* типизацию, а язык ***Python — динамическую сильную неявную***.

Ликбез по типизации в языках программирования /...

2. ***Q*** : Какой вид динамической типизации используется в **Python**?
   A : ***Динамическая / Сильная / Неявная***

3. ***Q*** : Перечислить встроенные типы данных.
   A : В Python имеется множество встроенных типов данных. Вот наиболее важные из них:
   1. None
   2. **Логический**, может принимать одно из двух значений — ***True*** (истина) или ***False*** (ложь).

3. **Числа (Numbers)**, могут быть целыми `int` (1 и 2), с плавающей точкой `float` (1.1 и 1.2), дробными (1/2 и 2/3), и даже комплексными `complex`.
4. **Строка (String)** — последовательности символов Юникода, например, HTML-документ.
5. **Байты** и **массивы байтов**, например, файл изображения в формате JPEG.
6. **Список (List)** — упорядоченные последовательности значений.
7. **Кортеж (Tuple)** — упорядоченные неизменяемые последовательности значений.
8. **Множество (Set)** — неупорядоченные наборы значений.
9. **Словарь (Dictionary)** — неупорядоченные наборы пар вида ключ-значение.

4. ***Q*** : Привести примеры изменяемых и неизменяемых типов, атомарных и ссылочных.

   A : Многие из предопределённых типов данных Python — это типы ***неизменяемых***(***immutable***) объектов:

   - числовые данные (**int** , **float** , **complex**);
   - символьные строки (**<class> 'str'**);
   - кортежи (**tuple**);
   - // есть еще такая штука как frozenset, тот же set только неизменяемый //

   Другие типы определены как ***изменяемые***(***mutable***):

   - списки (**list**);
   - множества (**set**);
   - словари (**dict**);

   Вновь определяемые пользователем типы (классы) могут быть определены как неизменяемые или изменяемые.
   В Python также выделяют ***атомарные*** и ***структурные*** (***или ссылочные***) типы данных. К ***атомарным*** типам относятся:

   - числа;
   - строки

   Ссылочные типы - это:

   - списки (**list**);
   - кортежи (**tuple**);
   - словари (**dict**);
   - функции;
   - классы и экземпляры классов;

5. ***Q*** : Что в результате будет выведено на экран?

```
def foo(a=[]):
    a.append(1)
    print(a)

foo()

foo()

foo()
```

A : Так как аргумент *a* *mutable* и *ссылочный:*

```
[1]
[1, 1]
[1, 1, 1]
```

6. **Словарь (dict)**
    1. ***Q*** : Как удалить элемент из словаря?

       A :

       ```
       my_dict = {'a': 1, 'b': 2}

       del my_dict['a']

       print my_dict

       # {'b': 2}
       ```

    2. ***Q*** : Какие типы данных могут выступать в роли ключей словаря?

       A : Словари — один из встроенных сложно структурированных типов (коллекций) Python (на ряду со списками, кортежами, множествами). Но словари в Python имеют особое, исключительное значение, так как сам интерпретатор Python весь основывается на словарях, а текущее пространство имён в точке исполнения — это словарь, ключами которого являются имена объектов.
       Ключами элементов словаря могут быть численные значения, строки, кортежи (**tuple**) и объекты *собственных классов* или даже функции, как показано ниже:

       ```
       def foo( i ):
           return i * i

       d = { 1:"111", 'two':"222", (3, 5, 7):"333", foo:"444" }

       print( d )
       ```

       Ключом словаря не может быть unhashable тип (увидим TypeError), иными словами все изменяемые структуры.
       // Почему unhashable класс может быть ключом? //
    3. ***Q*** : Можно ли использовать объекты пользовательских классов в качестве ключей?

A : Да, можно.

4. ***Q*** : Какие методы должны быть реализованы у hashable объектов?

   A : Объект *hashable* если у него есть значение *hash* которое не меняется в течении жизни объекта (должен иметь метод **__hash__()**), и может быть сравнен с другим объектом по значению hash (должен иметь метод **__eq__()** или **__cmp__()**)

5. ***Q*** : Сколько объектов будет содержать словарь?

```
class C(object):

    def __hash__(self):
        return 42

    def __eq__(self, other):
        return True

d = {C(): 1, C(): 2, C(): 3}
```

   A : Один

```
{<__main__.C object at 0x102589350>: 3}
```

7. ***Q*** : Какой объект будет создан в результате выполнения?

   A :

```
print set('abracadabra')

set(['a', 'r', 'b', 'c', 'd'])


print set([1, (1, 2)])

set([(1, 2), 1])


print set([1, {1, 2}])

TypeError: unhashable type: 'set'
```

8. ***Q*** : Почему вариант со вложенным множеством не работает?

   A : Потому что второй элемент в массиве - массив, а он не является *hashable* объектом.

9. ***Q*** : Написать **код**, удаляющий дубликаты из списка.

   A :

```
my_list = [1, 1, 2, 3, 4, 4, 4, 5]
my_list = list(set(my_list))

print my_list

# [1, 2, 3, 4, 5]
```

10. ***Q*** : В каких случаях нельзя использовать следующий **код** для решения предыдущей задачи?

```
mylist = list(set(mylist))
```

A : Когда необходимо сохранить очередность элементов

11. ***Q*** : Есть набор данных, над которым требуется выполнять операции проверки на вхождение заданного элемента в набор. Какую структуру данных лучше всего использовать для хранения набора и почему?
A : Все зависит от типа данных и от других операций, которые будут выполняться в дальнейшем. Простейшим вариантом видится множество (**set**), так как он уже реализует проверку на уникальность.

12. ***Q*** : Преобразовать строку `'12345432'` в список из цифр, используя list comprehension и/или функциональный подход:

```
# '12345432' -> [1, 2, 3, 4, 5, 4, 3, 2]
```

A :

```
[int(i) for i in '12345432']
```

13. ***Q*** : Что в результате будет выведено на экран:

```
x = 10

def foo():
    print x
    x += 1

foo()
```

A : Ошибка.

```
UnboundLocalError: local variable 'x' referenced before assignment
```

14. ***Q*** : Почему предыдущий **код** не работает? Что нужно сделать, чтобы он заработал?

A : Потому что у функции нет доступа к переменной снаружи.

Область видимости в Python

15.
```
x = 10

def foo():
    global x
    print x
    x += 1

foo()  # 10
foo()  # 11
```

16. ***Q*** : Как получить итератор по списку?

    A : list_iterator = iter([1, 2, 3, 4])

    Итерируемый объект, итератор и генератор / Хабр

    1. ***Q*** : Написать генератор счетчика без использования itertools и с использованием itertools .

       A :

```
def my_counter(start=0, step=1):
    while True:
        yield start
        start += step

ccc = my_counter(1, 2)

print ccc.next()
print ccc.next()
print ccc.next()


from itertools import count
cc = count()

print cc.next()
print cc.next()
print cc.next()
```

17. ***Q*** : Что входит в интерфейс генератора? Зачем нужен метод __send__() ?

    A : Методы __next__() и __iter__(), а так же выражение yield вместо return. Метод send() позволяет передавать генератору данные.

18. ***Q*** : Что такое сопроцедуры? Как реализовать сопроцедуры при помощи генераторов?

    A : Сопроцедуры или сопрограммы (**coroutine**) это частный случай генераторов.

    В отличие от генераторов сопроцедуры принимают на вход аргументы. Для этого используется метод **send()** . Чаще всего сопроцедуры используются при асинхронном

программировании или dataflow-программировании. ***Важно:*** при написании сопроцедур, ключевое слово **yield** имеет особую форму записи **(yield)**
Ниже приведена простая реализация генератора-сопроцедуры, который может складывать два аргумента, хранить историю результатов и выводить историю:

```
def calc():
    history = []
    while True:
        x, y = (yield)
        if x == 'h':
            print history
            continue
        result = x + y
        print result
        history.append(result)

c = calc()

print type(c)  # <type 'generator'>

c.next()  # Необходимая инициация. Можно написать c.send(None)
c.send((1,2))  # Выведет 3
c.send((100, 30))  # Выведет 130
c.send((666, 0))  # Выведет 666
c.send(('h',0))  # Выведет [3, 130, 666]
c.close()  # Закрываем генератор
```

Т.е. мы создали генератор, проинициализировали его и подаём ему входные данные. Он, в свою очередь, эти данные обрабатывает и ***сохраняет своё состояние между вызовами до тех пор пока мы его не закрыли***. После каждого вызова генератор ***возвращает управление туда, откуда его вызвали***.
Ниже приведен пример декоратора, который избавляет от необходимости инициализировать генератор:

```
def coroutine(f):
    def wrap(*args, **kwargs):
        gen = f(*args, **kwargs)
        gen.send(None)
        return gen

    return wrap


@coroutine
def calc():
    history = []
    while True:
        x, y = (yield)
        if x == 'h':
            print history
            continue
        result = x + y
        print result
        history.append(result)
```

Сопрограммы в Python

19. ***Q*** : Что такое замыкание? Приведите пример реализации замыкания в python.

    A : Замыкание (англ. closure) — функция, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем **коде** и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своём контексте

    Пример:

```
def outer_func(x):
    def inner_func(y):
        # inner_func замкнуло в себе x
        return y + x
    return inner_func
```

20. ***Q*** : Написать декоратор, который считает количество вызовов произвольной функции.

    A :

```
def counter(func):
    """
    Декоратор, считающий и выводящий количество вызовов
    декорируемой функции.
    """
    def wrapper(*args, **kwargs):
        wrapper.count += 1
        res = func(*args, **kwargs)
        print("{0} была вызвана: {1}x".format(func.__name__, wrapper.count))
        return res
    wrapper.count = 0
    return wrapper
```

Декораторы

21. ***Q*** : Что будет выведено на экран?

```
def bar(func):
    def wrapper():
        """wrapper function docstring"""
        return func()
    return wrapper

@bar
def foo():
    """foo function docstring"""
    print(foo.__name__)
    print(foo.__doc__)
```

A :

```
wrapper
wrapper function docstring
```

22. ***Q*** : Как изменить декоратор из предыдущего примера, чтобы сохранить все метаданные декорированной функции.

A : Для этого можно использовать функцию **wraps()** из модуля **functools**

```
import functools

def bar(func):
    @functools.wraps(func)
    def wrapper():
        """wrapper function docstring"""
        return func()
    return wrapper

@bar
def foo():
    """foo function docstring"""
    print(foo.__name__)
    print(foo.__doc__)

foo()
```

23. ***Q*** : Можно ли написать декоратор класса? Что в этом случае передается в параметры? Как можно использовать эту возможность?

    A : Да, можно. Передается объект класса. Можно например добавить метод в класс.

    Пример:

```
def class_dec(cls):
    class Class(cls):
        def method(self, *args, **kwargs):
            print args, kwargs
    return Class

@class_dec
class Foo():
    pass

>>> foo = Foo()
>>> foo.method(10, 11, a=1, b=2)
(10, 11) {'a': 1, 'b': 2}
```

24. ***Q*** : Привести примеры декораторов из стандартной библиотеки.

    A : `@classmethod` , `@staticmethod` , `@property`

25. ***Q*** : Чем отличаются декораторы `@classmethod` и `@staticmethod` ?

    A : **Статические методы** являются синтаксическими аналогами статических функций. Они не получают ни экземпляр (`self`), ни класс (`cls`) первым параметром. Для создания статического метода (только классы нового типа могут иметь статические методы) используется декоратор `@staticmethod`

    Методы класса занимают промежуточное положение между статическими и обычными. В то время как обычные методы получают первым параметром экземпляр класса, а статические не получают ничего, в классовые методы *передается класс*. Возможность

создания классовых методов является одним из следствий того, что в Python классы также являются объектами. Для создания классового (только классы нового типа могут иметь классовые методы) метода можно использовать декоратор `@classmethod`

26. ***Q*** : Для чего используется декоратор @property

    A : Для вычисляемых полей. В общем случае property() принимает четыре аргумента: get_function, set_function, del_function и строка документации. Т.е. мы можем определить поведение не только получения, но и добавления и удаления вычисляемого поля.

27. ***Q*** : Привести пример тернарной условной операции.

    A :

    Общий синтаксис: `a if condition else b` . Например:

    ```
    return 'true' if expr else 'false'
    ```

28. ***Q*** : Почему стоит избегать использование логических операторов для реализации тернарной условной операции?

    ```
    1                                   [expression] and    [on_true] or    [on_false]
    ```

    A : Это будет работать неочевидным образом в случае `on_true == 0`.

    ```
    x = 0
    x == 0 and 0 or 1  # 1
    x = 1
    x == 0 and 0 or 1  # 1
    ```

29. context manager

    она же конструкция with some_function() as p:

    Запись вида:

    ```
    with open('some_file', 'w') as opened_file:
        opened_file.write('Hola!')
    ```

    эквивалентна:

    ```
    file = open('some_file', 'w')
    try:
        file.write('Hola!')
    finally:
        file.close()
    ```

    Можно написать собственный класс, который будет реализовавыть context manager, для этого надо реализовать два метода: __enter__(self) и __exit__(self, type, value, traceback). __enter__ должен вернуть объект, с которым мы будем работать, __exit__ завершить работу, при этом он принимает параметрами тип, значение и трейсбэк ошибки, которая произойдет (или нет) во время исполнения обернутого **кода**. Пример:

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        self.file_obj.close()
```

30.
```
with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')
```

Так же в contextlib есть декоратор @contextmanager, который тоже позволит создать менеджер контекста. Декорируемая функция должна быть генератором, которая в yield вернет ровно одно значение, которое будет подставленно в переменную (после слова as)

31. Q: Какие методы должен реализовать дескриптор данных?
    A: Если говорить в общем, то дескриптор — это атрибут объекта со связанным поведением (*англ.*binding behavior), т.е. такой, чьё поведение при доступе переопределяется методами протокола дескриптора. Эти методы: `__get__` , `__set__` и `__delete__` . Если хотя бы один из этих методов определён для объекта, то он становится дескриптором.
    Если объект определяет сразу и `__get__` , и `__set__` , то он считается дескриптором данных (*англ.*data descriptor).

32. Q: В какой момент будут вызваны методы объекта __getattr__, __getattribute__?
    A: __getattribute__ вызывается при доступе к любому из атрибутов объекта. __getattr__ вызывается при доступе к несуществующему атрибуту или же, если __getattribute__ по каким-то причинам выкинул ошибку, вызовется __getattr__

33. ООП
    1. Q: Что такое интерфейсы и чем они отличаются от абстрактных классов?
       A:

    2. Q: Есть ли в python интерфейсы?
       A: В Питоне нет, но есть проектик который их сделал
       // вроде как смысла в этом нет Абстрактные классы и интерфейсы в Питоне / ХабрахабрХабрахабр//

    3. Q: Как реализовать абстрактный класс в python?
       A:

```
from abc import ABCMeta

class Movable():
    __metaclass__=ABCMeta
```

4. Q: Есть ли в python множественное наследование?
   A: Да

5. Q: Как происходит поиск метода в классе при множественном наследовании?
   A: В глубину с самого первого родителя, если дошли до конца и не нашли переходим ко второму
6. Q: Что такое `mixin` ?
   A: Класс-добавка. Обычно содержит небольшой функционал и не имеет объектов, используется для добавления функционала к классам

34. Метаклассы
    1. Q: Создать приведенный ниже класс с помощью метакласса type

```
1  class MyClass(MyBaseClass):
2      bar = True
```

    A:
    type('MyClass', ('MyBaseClass',), {'bar': True})
    2. Q: Привести пример использования метаклассов в Django
    A: // модели? //

35. ***Q*** : GIL, Какие особенности есть у реализации многопоточности в Питоне?
    A : GIL (***G****lobal* ***I****nterpereter* ***L****ock*) - это способ синхронизации потоков, который используется в некоторых интерпретируемых языках программирования, например в Python и Ruby. Является самым простым способом избежать конфликтов при одновременном обращении разных потоков к одним и тем же участкам памяти. GIL ограничивает параллельность вычислений, т.е. потоки не будут исполняться параллельно на разных CPU.

36. Q: garbage collector
    A: Сборщик мусора имеет три поколения (счёт начинается с нуля). При создании объекта он попадает в нулевое поколение.
    У каждого поколения есть счётчик и порог. Работает эта пара так:
    - При добавлении объекта в поколение счётчик увеличивается.
    - При выбывании из поколения счётчик уменьшается.
    - Когда счётчик превысит пороговое значение — по всем объектам из поколения пройдётся сборщик мусора. Кого найдёт — удалит.
    - Все выжившие в поколении объекты перемещаются в следующее (из нулевого в первое, из первого во второе). Из второго поколения объекты никуда не попадают и остаются там навечно.
    - Перемещённые в следующее поколение объекты меняют соответствующий счетчик, и операция может повториться уже для следующего поколения.
    - Счётчик текущего поколения сбрасывается.

Объекты, подлежащие уничтожению но имеющие переопределённый метод `__del__` не могут быть собраны. Причина проста: эти объекты могут ссылаться друг на друга. Python не способен определить безопасный порядок вызова `__del__` . Если вызывать деструкторы в произвольном порядке, то можно получить ситуацию вида:

- Деструктор объекта *a* для работы требует объект *b*.
- Последний в своём деструкторе обращается к объекту *a*.
- Если вызовем `__del__` у *a*, то деструктор *b* не сможет отработать нормально. Ссылка на *a* будет иметь значение *None*.

Чтобы не заставлять программиста корректно разрешать такие ситуации было принято решение не уничтожать подобные объекты а просто перемещать их в `gc.garbage` — и дальше программист пусть сам разбирается что делать с этим мусором.

37. Q: Как можно организовать динамический импорт модулей в Питоне?
    A: Первый вариант использовать функцию __import__(), которая принимает аргументом строку с путем до модуля (черевато иногда ошибками) и затем использовать getattr(module, class) для получения класса.
    Более изящный вариант пакет imp, метод load_module(). Но перед эти вызывается find_module()
38. Q: Как получить список импортированных модулей?
    A:
    sys.modules - словарь с испортированными модулями,
    sys.modules.keys() - "чистый" список модулей
39. Модуль `itertools`
    1. Q:Есть два списка A и B, необходимо получить результат AxB?
       A: product(A, B)
    2. Q: Дана строка S, необходимо вывести все возможные перестановки размером n алфавитном порядке?
       A: pnermutations('string', n) // вернет turples из букв
    3. Q: Дана строка S, необходимо вывести все возможные комбинации размером n алфавитном порядке?
       A: combinations('string', n) // вернет turples из букв
40. Модуль `collections`
    1. defaultdict

       A: collections. **defaultdict** ([*default_factory*,]...)

       Тип данных, который практически в точности повторяет функциональные возможности словарей, за исключением способа обработки обращений к несуществующим ключам.
       Когда происходит обращение к несуществующему ключу, вызывается функция, которая передается в аргументе default_factory. Эта функция должна вернуть значение по умолчанию, которое затем сохраняется как значение указанного ключа.
       Остальные аргументы функции defaultdict() в точности те же самые, что передаются встроенной функции dict().

2. Counter

A: Счетчик

class collections. **Counter**

```
counter = Counter(['spam', 'spam', 'eggs', 'spam'])
# Counter({'spam': 3, 'eggs': 1})

counter2 = Counter(['eggs', 'eggs', 'bacon'])
# Counter({'eggs': 2, 'bacon': 1})

counter + counter2
# Counter({'bacon': 1, 'eggs': 3, 'spam': 3})

counter2 - counter
# Counter({'bacon': 1, 'eggs': 1})

counter & counter2
# Counter({'eggs': 1})

counter | counter2
# Counter({'bacon': 1, 'eggs': 2, 'spam': 3})
```

most_common ([*count*])

возвращает n наиболее часто встречающихся элементов, в порядке убывания встречаемости. Если n не указано, возвращаются все элементы.

```
counter.most_common(1)
# [('spam', 3)]
```

3. namedtuple

A: collections. **namedtuple** (*typename*,*fieldnames*[,*verbose*])

Возвращает именованный кортеж.
Именованные кортежи эффективнее расходуют память и поддерживают различные операции над кортежами, такие как распаковывание элементов (например, если имеется список именованных кортежей, эти кортежи можно будет распаковывать в цикле for, например: for name, shares, price in stockList).
Недостатком именованных кортежей является более низкая скорость операции получения значений атрибутов в сравнении с классами.

- typename - имя класса, возвращаемого объекта
- fieldnames - список имен атрибутов в виде строк.

  Имена в этом списке должны быть допустимыми идентификаторами Python. Они не должны начинаться с символа подчеркивания, а порядок их следования определяет порядок следования элементов кортежа, например [‘hostname’, ‘port’].
  Кроме того, допускается передавать строку, такую как ‘hostname port’ или ‘hostname, port’.

- verbose - булево, выводить определение класса в поток стандартного вывода.

Django

1. Сколько запросов к базе данных будет выполнено и почему?

```
users = Users.objects.filter(...)
messages = Messages.objects.filter(user__in=users)
messages[0]
```

A: 1 запрос при выполнении messages[0], т.к. Питон ленив и первые две строки просто создадут QS без запроса к БД.

```
qs = Messages.objects.filter(name='Fred')
x = qs[2]
x = qs[2]
```

A: 2 запроса на последних двух строках, т.к. вытаскивание по индексу не кэшируется

```
qs = Messages.objects.filter(name='Fred')
list(qs)
x = qs[2]
x = qs[2]
```

A: 1 запрос, т.к. list(qs) вытащит все и положит в кэш, а следующие два запроса будут доставаться из кэша

```
user = Users.objects.get(pk=1)
list(user.messages.all())
list(user.messages.all())
```

A: 3 запроса. get - возвращает объект, так что запрос точно будет. list() - вычисляет QS, а так как нету объекта, в который бы все закэшировалось, повторный вызов вычислит QS еще раз

2. Какой способ обращения предпочтительнее и почему?
    1. `message.user.id`
    2. `message.user_id`

    A: предпочтительнее второе, т.к. во втором используется поле, которое есть у объекта, а в первом будет сделан запрос в базу
3. Q: В какой момент `QuerySet` вычисляется?

   A:
    1. Во время первого итеррирования по QS (методы типа all(), iterator() исключение, см. ниже)
    2. Slicing. Если QS не был вычислен, то слайс вернет не вычисленный QS. QS вычисляется заново, если делаем слайс на вычисленный QS или если использовать "шаг" (третий параметр в слайсе). В случае использования шага QS, вычислится и вернется list.
    3. При применении функций repr(), len(), list() над QS
    4. При использовании в качестве bool() (if QS: НО: если не используется метод exists)
    5. Pickling/Caching

4. Q: Как посчитать количество записей в `QuerySet` ?

   A: Если нам в любом случае понадобятся данные всей выборки быстрее использовать питоновский len(), который выполнит запрос в QS и сохранит все в кэш.
   Используя count() выполнится SELECT COUNT(*) в бд. Поэтому, если нам не надо грузить все записи, а надо просто их посчитать, лучше всегда использовать count().
5. Q: Как проверить наличие хотя бы одной записи в таблице?

   A: через метод exists(), тогда будет проверено есть ли хоть одна строка, при этом результат выборки не засунется в кэш
6. Q: В каких случаях следует использовать метод `QuerySet.iterator()` ?

   A: если результат выборки огромен, т.к. итератор будет извлекать данные частями. НО: данные не будут помещаться в кэш
7. Будет ли помещены в кэш данные выборки?

```
messages = Messages.objects.filter(name='Fred')
for x in messages.iterator():
    pass
```

   A: нет

8. Что будет в результате, если запись с `id = 1` не существует?

```
Users.objects.get(id=1)
```

   A: DoesNotExist

9. Q: Как с помощью Django ORM можно построить запрос с операцией OR?
   A: использовать Q(), в котором | будет означать OR
10. Q: В таблице есть числовое поле `count` . Как увеличить значение `count` на единицу одним запросом?
    A: Используя класс F(), который означает "само поле"
    Model.objects.filter(pk=id).update(count=F('count') + 1)
11. Q:Чем отличаются `commit_on_success()` и `atomic()` ?

    A: `commit_on_success()` - начиная с 1.6 устаревший вариант работы с транзакциями.
    Это декоратор, который оборачивает все действия функции в транзакцию. Если транзакция завершается успешно, изменения фиксируются, иначе откатываются. Существуют проблемы с вложенными кусками, т.к. вложенный кусок может отдельно закоммититься, тогда как внешний может откатиться
    atomic() - новая версия работы с транзакциями, введена с 1.6. atomic блоки могут быть вложенными. В этом случае, если вложенный блок выполнился успешно, изменения в базе данных, которые он произвел, могут быть отменены при ошибке во внешнем блоке **кода**.
    atomic может быть как декоратором, так и менеджером контекста
    // commit_on_success тоже, инфа 100%. -- ▲
12. Q: QuerySet.select_related(), QuerySet.prefetch_related()
    A: select_related создает запрос SQL объединяющий связанные таблицы и включая

дополнительные поля в SELECT. По этой причине, select_related получает связанные объекты в том же запросе. Однако, что бы избежать большого количества возвращаемых данных при обработке “множественных” связей, select_related работает только со связями возвращающими один объект - внешний ключ и связь один-к-одному.

prefetch_related, с другой стороны, выбирает данные для каждой связи отдельно, и выполняет “объединение” на уровне Python. Благодаря этому могут быть обработаны связи многое-ко-многим и многое-к-одному, которые не обрабатывает select_related, в том числе и внешние ключи и связь один-к-одному поддерживаемые select_related.

13. Q: QuerySet.values() and values_list()
    A: values() возвращает вместо модели словарь, где ключи это названия полей
    values_list() вернет листы со значениями полей. Если указывается только одно поле, можно указать дополнительный параметр flat=True, тогда вместо множества листов длины 1, вернется один лист
14. Q: QuerySet.defer() and only()
    A: в defer() указывается какие поля надо исключить из запроса (например если они очень тяжелые и мы не уверены нужны ли они нам). Причем если мы обратимся потом к полю, которое было в defer() каждое обращение будет создавать запрос в бд
    при использовании only() будут выбраны только поля указанные в only()
15. Q: QuerySet.annotate() and QuerySet.aggregate()
    A: Обе функции осуществляют агрегацию (Count(), Min(), Max(), Sum() и прочее)
    annotate() - добавит поле к модели. Если в агрегации используется только одно поле, агрегация может быть не именнованной, тогда имя составляется как <имя поля>__<имя функции> (entry__count). Если поле не одно, обязательно именование
    aggregate() - в отличие от предыдущего метода вернет словарь
16. mptt
    // django-mptt, если речь о нем, помогает реализовать хранение дерева в таблице. Данный метод хранения называется NestedSet. Метод хорош при доставании данных, т.к. это можно вытащить все одним запросом, но плох при обновлении данных, т.к. хранит в каждом узле самый правый и самый левый ключи и приходится их каждый раз индексировать заново //
17. middleware
    A: самый адекватный русский эквивалент "промежуточный слой". Указываются в MIDDLWARE_CLASSES при выполнении запроса сначала выполнятся process_request(request) и process_view(requet, view_func, view_args, view_kwargs) в прямом порядке (сверху вниз, как указанно в настройке). Эти методы вызываются до вызова view, причем если они возвращаются HTTPResponse, то вызов на этом завершается. Затем, после вызова метода view, вызываются process_exception(request, exception) если запрос завершился ошибкой или же process_template_response(request, response) или process_response(request, response) в обратном порядке (снизу вверх).
18. Q: Для чего используется пакет django-south?

A: миграции

SQL (в postgres)

1. Есть две таблицы: пользователи и сообщения. Таблица пользователей содержит имя, таблица сообщений - ссылку на пользователя и текст сообщения.

I users	**Таблица пользователей**
id	Первичный ключ
first_name	Имя пользователя
last_name	Фамилия пользователя
messages	**Таблица сообщений**
id	Первичный ключ
user_id	Ссылка на пользователя
text	Текст сообщения

   Требуется:
   1. Q: вывести текст сообщения и имя пользователя, создавшего, сообщение;
      A: SELECT messages.text, users.last_name FROM messages JOIN users on messages.user_id=users.id;
   2. Q: вывести имена пользователей и количество сообщений каждого пользователя, в том числе пользователей, у которых нет сообщений;
      A: SELECT users.last_name, COUNT(messages.id) as count FROM users LEFT JOIN messages ON messages.user_id=users.id GROUP BY users.id
   3. Q: вывести тех, у кого количество сообщений более 10;
      A: SELECT users.last_name, COUNT(messages.id) as count FROM users JOIN messages ON messages.user_id=users.id GROUP BY users.id HAVING COUNT(messages.id) > 10
   4. Q: вывести список однофамильцев
      A: SELECT * FROM users WHERE last_name in (SELECT last_name FROM users GROUP BY last_name HAVING COUNT(id) > 1)
   5. Q: вывести только третью запись из таблицы.
      A: SELECT * FROM users LIMIT 1 OFFSET 2
2. Q: Перечислить способы хранения древовидных (иерархических) структур в реляционной базе данных.
   A: Неплохая статья с подробным описанием классических трех вариантов на примере Doctrine Иерархические структуры данных и Doctrine / ХабрахабрХабрахабр
   Adjacency List - самый простой способ. Храним узел с ссылку на родителя. Сложно доставать данные (чаще всего требуется множество запросов либо большой объем постобработки), но легко добавлять и изменять
   Nested Set - используется в mptt. Легок в чтении, но сложен в изменении
   Materialized Path - в этом случае мы храним отдельным полем не ссылку на родителя, а

путь до него. по сравнению с Nested Set, он более поддается изменениям. В то же время остается достаточно удобным для выборки деревьев целиком и их частей. Но, и он не идеален. Особенно по части поиска предков ветки.

3. Q: Для чего нужна конструкция `UNION`, чем отличается от `UNION ALL`?

   A: Операция UNION объединяет результаты нескольких запросов в один (query1 UNION query2) при этом удаляя дубликаты. UNION ALL дубликаты не удаляет

4. `SELECT FOR UPDATE`

   1. Q: В каких случаях используется?

      A: Когда нужна гарантия того, что прочитанные данные не изменятся в рамках транзакции

   2. Q: Какие операции будут заблокированы таким запросом?

      A: Будут заблокированны операции изменения данных (UPDATE, DELETE, SELECT FOR UPDATE)

5. Структура `SELECT`-запроса

   1. Q: Как задать перечень колонок для выборки?

      A: SELECT <перечень колонок через запятую>

   2. Q: Что такое `DISTINCT`, `DISTINCTROW`?

      A: DISTINCT исключает повторяющиеся строки из выборки и возвращает только "первую" (итог непредсказуем, если нет сортировки)

   3. Q: Для чего используется `WHERE`?

      A: Условия выборки

   4. Q: Как отсортировать выборку?

      A: ORDER BY

   5. Q: Каким образом можно сгруппировать данные выборки?

      A: GROUP BY

   6. Q: Можно ли применить `WHERE` после группировки?

      A: Если нужны условия после группировки, они пишутся после ключевого слова HAVING

6. Объединение таблиц

   1. Q: Как можно присоединить таблицу?

      A: JOIN, UNION, перечислить через запятую (работает как cross join)

   2. Q: Какие виды присоединения существуют?

      A: //если речь про виды джоинов то {INNER | {LEFT | RIGHT | FULL} OUTER | CROSS } JOIN
      INNER (можно не упоминать это имя) проверяет условие и выбирает только то, что соответствует условию
      OUTER выбирает по условию, плюс прибавляет то, что в "главной" таблице (в случае LEFT главная таблица левая, RIGHT - правая, FULL - обе) подставляя вместо несуществующих полей NULL
      CROSS на каждую строку из левой таблицы добавляет значения правой (декардово произведение)//

M3

1. Основные отличия m3 от django.
2. Путь запроса/ответа от клиента/клиенту в приложении с использованием m3.
3. Что такое Controller, Pack, Action в терминах m3?
4. Где логичнее всего расположить проверку прав на доступ к Action?

Javascript (ExtJS)

1. Q: Как можно создать объект.

   A:

   Оператор new

```
//Создаем наш объект
var MyObject = new Object();
//Переменные
MyObject.id = 5; //Число
MyObject.name = "Sample"; //Строка
//Функции
MyObject.getName = function()
{
    return this.name;
}
```

   Литеральная нотация

```
//Создаем наш объект с использованием литеральной нотации
MyObject = {
    id : 1,
    name : "Sample",
    boolval : true,
    getName : function()
    {
        return this.name;
    }
}
```

   Конструкторы объектов

```
function MyObject(id,name)
{
   this._id = id;
   this._name = name;
   this.defaultvalue = "MyDefaultValue";

   //Получение текущего значения
   this.getDefaultValue = function()
   {
      return this.defaultvalue;
   }

   //Установка нового значения
   this.setDefaultValue = function(newvalue)
   {
      this.defaultvalue = newvalue;
   }

   //Произвольная функция
   this.sum = function(a, b)
   {
      return (a+b);
   }
}

5var MyFirstObjectInstance = new MyObject(5,"Sample");
```

Ассоциативные массивы

```
var MyObject = new Number();
MyObject["id"] = 5;
MyObject["name"] = "SampleName";

for (MyElement in MyObject)
{
  //Код обхода
  //В MyElement - идентификатор записи
  //В MyObject[MyElement] - содержание записи
}
```

Подробненько о подводных камнях: Создание объектов в Javascript / ХабрахабрХабрахабр

2. Q: От чего наследуется простой объект `{}` ?
   A: Используя объектный литерал — нотацию `{}` — можно создать простой объект. Новый объект наследуется от `Object.prototype` и не имеет собственных свойств
3. Q: Как можно обратиться к свойствам объекта?
   A: Получить доступ к свойствам объекта можно двумя способами: используя либо точечную нотацию, либо запись квадратными скобками.

```
var foo = {name: 'kitten'}
foo.name; // kitten
foo['name']; // kitten
```

4.
```
var get = 'name';
foo[get]; // kitten
```

5.
```
foo.1234; // SyntaxError
foo['1234']; // работает
```

6. Q: Как удалить свойство объекта?
   A: Единственный способ удалить свойство у объекта — использовать оператор `delete`; устанавливая свойство в `undefined` или `null`, вы только заменяете связанное с ним *значение*, но не удаляете *ключ*.
   Замечание: Если ссылок на значение больше нет, то сборщиком мусора удаляется и само значение, но ключ объекта при этом всё так же имеет новое значение.

```
var obj = {
    bar: 1,
    foo: 2,
    baz: 3
};
obj.bar = undefined;
obj.foo = null;
delete obj.baz;

for(var i in obj) {
    if (obj.hasOwnProperty(i)) {
        console.log(i, '' + obj[i]);
    }
}
```

7. Q: Всё ли является объектами в javascript
   A: В JavaScript всё ведет себя, как объект, лишь за двумя исключениями — `null` и `undefined`.
   в JavaScript есть 6 *базовых типов* данных — это Undefined (обозначающий отсутствие значения), Null, Boolean (булев тип), String (строка), Number (число) и Object (объект). При этом первые 5 являются *примитивными* типами данных, а Object — нет. Кроме того, условно можно считать, что у типа Object есть «подтипы»: массив (Array), функция (Function), регулярное выражение (RegExp) и другие.
   Это несколько упрощенное описание, но на практике обычно достаточное.

   Кроме того, примитивные типы String, Number и Boolean определенным образом связаны с не-примитивными «подтипами» Object: String, Number и Boolean соответственно.
   Это означает, что строку 'Hello, world', например, можно создать и как примитивное значение, и как объект типа String.
8. Q: Можно вызвать метод у числового литерала?
   A: Неверно считать, что числовые литералы нельзя использовать в качестве объектов —

это распространённое заблуждение. Его причиной является упущение в парсере JavaScript, благодаря которому применение *точечной нотации* к числу воспринимается им как литерал числа с плавающей точкой.

```
2.toString(); // вызывает SyntaxError
```

Есть несколько способов обойти этот недостаток и любой из них можно использовать для того, чтобы работать с числами, как с объектами:

```
2..toString(); // вторая точка распознаётся корректно
2 .toString(); // обратите внимание на пробел перед точкой
(2).toString(); // двойка вычисляется заранее
```

9. В чем отличие строкового литерала от объекта типа String. Как сработает следующий **код** и почему так происходит?

```
var str = new String('test');
str.text = 'test'
console.log(str.text);

var str = 'test';
str.text = 'test';
console.log(str.text);
```

A: Объекты `String`, заданные через кавычки (и называемые "примитивными" строками), немного отличаются от объектов `String`, созданных с помощью оператора new. Так, например, типом (typeof) данных объекта, созданного при помощи `new`, является `'object'`, а не `'string'`. И такому объекту можно напрямую назначать дополнительные свойства и методы. В остальном - интерпретатор автоматически превращает примитивные строки в объекты.

10. Что будет выведено в консоль

```
var a = 42;

function foo() {
    a = 21;
}

foo();
console.log(a);
```

A: 21, т.к. переменная глобальна

11. Как вызвать функцию foo, чтобы this указывал на объект a?

```
var a = {}
function foo(b) {
    this.b = b;
}

A:
foo.call(a, 1)
foo.apply(a, [1])
```

Deploy (gunicorn/uwsgi/nginx)

Deploy (postgres)

Инструменты разработки

Git

1. Q: Как создать ветку?
   A: git branch <name> - просто создание
   git checkout -b <name> - создать и переключиться на новую ветку
2. Q: Как перенести коммит из одной ветки в другую?
   A: git cherry-pick <commit>
3. Q: Какие существуют способы включения изменений из одной ветки в другую?
   A: merge, rebase
4. Q: Как поставить метку на коммит?
   A: git tag -a <name>

IDE