KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 7 урок > Работа с грязными данными

> Оглавление

> Альтернативный способ создания списка

Как мы знаем, `List comprehension` — это способ, который часто используется как лаконичная (и ускоренная) замена циклов, где заполняется список:

```
xs = [i + 3 for i in range(10)]
```

Аналогично:

```
xs = []
for i in range(10):
    xs.append(i + 3)
```

В таком способе можно прописать условия и даже вложенные циклы. Однако не стоит сильно их нагружать: читаемость кода — одно из его самых важных качеств, а вложенные конструкции делают код сложнее для понимания.

```
# Получаем все чётные числа от 0 до 10 (не включая правую гра
ницу)
evens = [i for i in range(10) if i % 2 == 0]
```

```
# Аналогично
even = []
for i in range(10):
    if i % 2 == 0:
        even.append(i)
```

Больше информации тут.

> Конвертация типов в датафрэйме

Нередкой является ситуация, когда тип данных в колонке не соответствует желаемому. Почему это вообще важно? Для разных типов определены разные операции — `'0.15' * 100` не переведёт дробь в проценты.

Чтобы это исправить, есть метод `astype`, в который можно передать словарь, где ключи — названия колонок, а значения — новые типы для них. Метод возвращает новый датафрэйм с изменёнными типами:

```
# Приведём колонку money к типу данных float
df = df.astype({'money': 'float'})
```

Для конвертации типов колонок есть более простой вариант — передайте желаемый тип при вызове `astype` от колонки:

```
# Аналогично привели колонку height к типу данных float
df.height = df.height.astype('float')
```

Документация

> Удаление колонок

Чтобы убрать часть колонок из датафрэйма, воспользуйтесь методом `drop`, куда можно передать список из названий, которые нужно убрать. Метод также позволяет убирать строки по индексу.

Для указания измерения, в котором мы работаем, используется аргумент `axis` (`0` — строки, `1` — колонки).

Лучше использовать более понятные `columns`/`index`. Возвращается новый датафрэйм:

```
# Из датафрейма df убираем колонку Date
df = df.drop(columns = 'Date')
```

```
# Из датафрейма df убираем строку с индексом 350
df = df.drop(index = 350)
```

Документация

> Открывание файлов

Рассмотренный способ для открытия файлов с помощью `pd.read_csv` не единственный и не первый в питоне. Как нам известно, любой файл открывается с помощью функции `open()`, принимающей путь к файлу.

Например, у нас имеется файл `text.txt` с содержимым:

```
Строка 1
Строка 2
Строка 3
```

Откроем его с помощью функции `open()`:

```
file = open('text.txt')
```

Если мы посмотрим на тип данных переменной `file` после выполнения этой команды, то увидим следующее:

```
type(file)
_io.TextIOWrapper
```

`_io.TextIOWrapper` — это класс в Python, который представляет собой обертку для работы с текстовыми файлами. Он обеспечивает интерфейс для чтения и записи текстовых данных в файлы.

Когда производится открытие файла с помощью функции `open()` в режиме чтения текстового файла, Python создает объект `_io.TextIOWrapper`, который предоставляет возможность использовать методы для чтения данных из файла, перемещения указателя файла и других операций, связанных с текстовыми файлами.

У `file` есть различные методы на чтение содержимого, например, метод `readlines().`

Метод `readlines()` — это метод объекта файла в Python, который используется для чтения всех строк из файла и возвращения их в виде списка строк. Каждый элемент списка представляет одну строку из файла.

```
file.readlines()
```

Output:

```
['Строка 1\n', 'Строка 2\n', 'Строка 3']
```

Для того, чтобы прочитать и вывести на экран все строки из файла `text.txt`, можно воспользоваться циклом `for`:

```
for line in file.readlines():
    print(line)
```

Output:

```
Строка 1

Строка 2

Строка 3
```

В конце работы с файлом (то есть когда он вам больше не понадобится), его нужно закрыть, для чего используется метод `close`:

```
file.close()
```

Больше информации тут.

> Просмотр содержимого папок

Просмотр папок и многие другие операции, связанные с файлами и папками, выполняются с помощью библиотеки `os`.

Модуль `os` в Python предоставляет набор функций для работы с операционной системой, включая управление процессами, файлами и каталогами.

Некоторые из наиболее часто используемых функций из библиотеки `os` включают:

`os.getcwd()`	возвращает текущую рабочую директорию (папку)
`os.listdir(path)`	принимает путь к папке и возвращает её содержимое в виде списка
`os.path.exists(path)`	проверяет, существует ли файл или папка по указанному пути

`os.path.join(path1, path2)`	объединяет два пути в один
`os.mkdir()`	создает новую папку
`os.rmdir(path)`	удаляет пустые папки по указанному пути
`os.path.exists(path)`	проверяет, существует ли файл или папка по указанному пути
`os.walk`	используется для рекурсивного обхода (обход дерева) папок и файлов в указанном пути, возвращая список содержимого каждой папки

Ознакомиться со всеми методами библиотеки `os` можно тут.

Метод `os.listdir()`

Метод `os.listdir()` в модуле `os` возвращает список имен файлов и папок в указанной директории (папке). Он принимает на вход один аргумент — путь к папке, затем возвращает её содержимое в виде списка:

```
import os
os.listdir('/home/jupyter-an.karpov/shared/')
```

Данный код выведет список файлов и папок, расположенных в ноутбуке пользователя `an.karpov` в папке `shared`.

Важно отметить, что метод `os.listdir()` не включает специальные записи `"."` и `".."` в возвращаемый список имен файлов и папок.

Если в метод `os.listdir()` не передать аргумент, то будет возвращен список имен файлов и папок в текущей рабочей директории. Текущая рабочая директория — это по сути папка, в которой программа была запущена или в которой она в настоящее время выполняется.

Названия файлов в папке вместе с путём к ней позволяют реконструировать полный путь и работать с этими файлами:

```
# Указываем путь до папки
path = 'data_folder'

# Объединяем переменную path, символ / и первый элемент из сп
```

```
иска имен файлов и подпапок в папке data_folder
path_to_file = path + '/' + os.listdir(path)[0]
```

Таким образом, в переменной `path_to_file` будет содержаться путь к первому файлу или подпапке в папке `data_folder`.

С помощью метода `os.listdir()` можно строить пути к файлам, находящимся в разных папках, но в рамках одной директории.

Предположим, у нас есть следующая структура хранения данных:

```
data_folder/
    ├── subfolder1/
    │   ├── file1.txt
    │   └── file2.txt
    └── subfolder2/
        ├── file3.txt
        └── file4.txt
```

Мы хотим создать список путей к файлам `file1.txt`, `file2.txt`, `file3.txt` и `file4.txt`. Это вполне можно реализовать с помощью метода `os.listdir()` и цикла `for` следующим образом:

```
import os

# Задаем переменную data_folder, содержащую путь к директории
data_folder в файловой    системе
data_folder= 'data_folder'

# Создаём пустой список file_paths, в который будут добавлять
ся пути к файлам.
file_paths = []

# Запускаем цикл для каждого элемента(подпапки subfolder1 и s
ubfolder2) из директории      data_folder
for folder in os.listdir(data_folder):
        # Строим путь к подпапке, объединяя переменную data_f
```

```
older и имя текущего элемента
    folder_path = os.path.join(data_folder, folder)
    # Запускаем вложенный цикл по элементам внутри подпапок(s
ubfolder1 и subfolder2)
    for file in os.listdir(folder_path):
            # Строим путь уже к самим файлам, объединяя путь
к подпапки и имя текущего файла
          file_path = os.path.join(folder_path, file)
        # Добавляем получившийся путь в список
        file_paths.append(file_path)
```

Таким образом, данный код рекурсивно проходит по всем подпапкам в директории (папке) `data_folder` и создает список путей ко всем файлам, находящимся в этих подпапках.

Документация

Метод `os.walk()`

При вложенности папок и необходимости добраться до дна можно использовать метод `os.walk()`, который предоставляет собой удобный способ рекурсивного обхода директорий (папок) в файловой системе.

Он возвращает генератор, который генерирует кортежи для каждой директории в дереве каталогов. Каждый кортеж содержит три элемента: путь к папке, список подпапок в текущей папке и список имен файлов в них, не являющихся папками:

```
import os

for path, dirs, files in os.walk('Res_Tree'):
    print(path, dirs, files)
```

Output:

```
Res_Tree ['F000545', 'M000547', 'F000570'] ['.DS_Store']
Res_Tree/F000545 ['res_2019.09.11_0.0_6493B6_Container-dat_21
25_91-105-165_F000545', 'res_2019.09.10_0.0_6493B6_Container-
```

```
dat_2110_84-99-223_F000545'] []
Res_Tree/F000545/res_2019.09.11_0.0_6493B6_Container-dat_2125
_91-105-165_F000545 [] ['meals_list_2.txt']
...
```

На каждой итерации (первый этап цикла) метод возвращает тройку из пути к нынешней папке, списков папок и файлов, хранящихся в этой папке.

Для построения путей к файлам определенного типа (например, файлов с расширением `.txt`) в структуре папок, описанной выше, мы можем использовать `os.walk()` следующим образом:

```
import os

# Задаем переменную data_folder, содержащую путь к директории
data_folder в файловой    системе
data_folder= 'data_folder'

# Создаём пустой список file_paths, в который будут добавлять
ся пути к файлам.
file_paths = []
# Запускаем цикл for совместно с методом os.walk() по папке d
ata_folder
# В переменной path находится текущий путь (папка data_folde
r), dirs — список подпапок в текущей папке, files — список фа
йлов в них
for path, dirs, files in os.walk(data_folder):
        # Вложенный цикл для перебора файлов в текущей папке
    for file in files:
            # Проверка формата файла — текстовый файл должен
заканчиваться на '.txt'
        if file.endswith('.txt'):
                # Построение полного пути к файлу с помощью о
s.path.join(), объединяя       текущий путь path и имя файла
file
            file_path = os.path.join(path, file)
```

```
            # Добавляем получившийся путь в список
            file_paths.append(file_path)
```

Документация

Также есть ещё один вариант работы с файлами и папками — модуль pathlib.

> Регулярные выражения

Их также называют регэкспы или РЕ. При работе с текстовыми данными часто возникает необходимость их парсить (то есть извлекать нужные данные из всего текста). Возьмет такой пример: у нас есть данные о почтовых адресах пользователей, мы хотим узнать, с каких доменов (всё, что после @) у нас пользователей больше:

```
vasya@yandex.ru
katya_ivanova@gmail.com
sasha@karpov.courses.com
masha@gmail.com
```

Мы могли бы посчитать по доменным именам `value_counts`, если бы они были у нас в колонке в датафрейме. Но что делать, если нам даны целые мэйлы?

На помощь приходят регулярные выражения. **Регулярные выражения** — это специальный язык для описания низкого уровня языковой грамматики. Можно сказать, что РЕ позволяют вычленить из регулярного текста (его структура одинакова/почти одинакова на протяжении всего текста) нужные нам части.

Сначала разберём всё в простом питоне, а потом уже в пандасе. В данном примере с почтой мы можем просто воспользоваться строковыми методами питона:

- засплитить по @
- взять последнюю часть получившегося списка
- ...
- PROFIT

Но не на всех задачах встроенные методы так хорошо работают. Сначала посмотрим, как решить этот таск PE, а потом разберём что-нибудь посложнее. Решение и его объяснение:

```
import re

mail = 'vasya@yandex.ru'

pattern = re.compile('@([\w.]+)')

pattern.findall(mail)
['yandex.ru']
```

- `import re` — импортируем модуль re для работы с регулярными выражениями, в чистом питоне их нет
- `pattern = re.compile('@([\w.]+)')` — с помощью функции `compile` из модуля `re` создаём паттерн (образец), который будем искать в тексте, помещаем его в переменную `pattern`. Паттерн создаётся при помощи строки — о том, с чем совпадает (что мэтчит) этот паттерн мы поговорим дальше. Паттерн обладает набором методов (также, как у датафрейма есть методы), один из которых мы и используем
- `pattern.findall(mail)` — применяем метод `findall` на строке с почтой. Метод `findall` возвращает список со всеми встречаниями паттерна (`pattern`) в строке, где мы ищем (`mail`). В результате мы получили список с одним мэтчем — `['yandex.ru']`

На первый взгляд кажется, что способ непонятный (и неудивительно — мы ещё не обсуждали как описывается паттерн) и бессмысленный, ведь есть `split`. Однако у этого способа на чуть более сложной задаче уже есть плюсы:

```
text = '''We have several emails - vasya@yandex.ru, katya_iva
nova@gmail.com,
sasha@karpov.courses.com and also masha@gmail.com'''
```

```
pattern.findall(text)
['yandex.ru', 'gmail.com', 'karpov.courses.com', 'gmail.com']
```

Одним питоновским сплитом мы бы тут не отделались! В следующей главе поговорим об описании паттерна.

Тестирование регулярных выражений

Азбука регулярных выражений

Чтобы описать регулярное выражение, используется набор символов. Рассмотрим наиболее частые из них:

Буквы и цифры

Буквы и цифры в паттерне обозначаются соответственно буквами и цифрами. Если мы напишем:

```
import re


text = 'the gray fox jumps over the lazy dog'

pattern = re.compile('ox')
pattern.findall(text)
['ox']  # from the fox
```

То мы найдём все `'ox'` в тексте. С числами такая же история. Да и со многими знаками типа `@`.

Метасимволы

Специальные символы для модуля `re`, обозначающие группу значений:

- `\d` — любая цифра (`digits`)
- `\D` — всё, что угодно, кроме цифры
- `\s` — любой пробельный символ (`spaces`)
- `\S` — всё, что угодно, кроме пробельного символа

- `\w` — любая буква, цифра или _ (`words`)
- `\W` — всё, что угодно, кроме буквы, цифр или `_`

То есть нижний регистр — хотим это, верхний регистр — хотим не это.

Ещё есть:

- `.` — любой символ

Пара примеров:

Тройки цифр:

```
text = '+7-921-000-00-00 +7-981-555-55-55'

pattern = re.compile('\d\d\d')

pattern.findall(text)
['921', '000', '981', '555']
```

Фрагменты из 4-х знаков, начинающиеся с `'в'` :

```
asimov = '''Робот не может причинить вред человеку или своим
бездействием допустить, чтобы человеку был причинён вред.
Робот должен повиноваться всем приказам, которые даёт челове
к, кроме тех случаев, когда эти приказы противоречат Первому
Закону.
Робот должен заботиться о своей безопасности в той мере, в ко
торой это не противоречит Первому или Второму Законам.'''

pattern = re.compile('в...')

pattern.findall(asimov)
['вред', 'веку', 'воим', 'вием', 'веку', 'вред', 'вино', 'ват
ь', 'всем', 'век,', 'в, к', 'воре', 'вому', 'воей', 'в то',
'в ко', 'воре', 'вому']
```

Группы

Скобочки (`()`) имеют особое значение — они обозначают группы символов в паттерне. Благодаря этому мы можем извлечь кусочки из замэтчившегося паттерна. Например, достанем только код из телефонного номера:

```
text = '+7-921-000-00-00 +7-981-555-55-55'

pattern = re.compile('(\d\d\d)-(\d\d\d)')

pattern.findall(text)
[('921', '000'), ('981', '555')]
```

Обратите внимание, что мы получаем кортежи, где каждый элемент — группа из одного мэтча. Это позволяет нам извлечь нужную группу из каждого мэтча (хотя бы просто циклом по `pattern.findall(...)` с извлечением 0-го элемента). Раньше мы получали все тройки цифр сплошняком.

Другое наблюдение — минус в паттерне никак не отображается: мы мэтчим в тексте 3 цифры, минус, 3 цифры. То есть он должен быть в тексте, чтобы замэтчить, но мы можем убрать его из аутпута.

Квантификаторы

Квантификаторы — это символы, позволяющие специфицировать, сколько раз нужно повторить то, что идёт до них. Вот их виды:

- `*` — сколько угодно раз (от 0 до бесконечности)
- `+` — 1 или больше раз
- `?` — 0 или 1 раз (то есть или предыдущий символ будет, или нет)
- `{}` — в скобочках можно указать точное время или диапазон, читайте подробнее о них и других символах в документации

Квантификаторы можно ставить после символа или группы. К примеру, отберём весь текст, начинающийся со слов человек:

```
asimov = '''Робот не может причинить вред человеку или своим
бездействием допустить, чтобы человеку был причинён вред.
Робот должен повиноваться всем приказам, которые даёт челове
```

```
к, кроме тех случаев, когда эти приказы противоречат Первому
Закону.
Робот должен заботиться о своей безопасности в той мере, в ко
торой это не противоречит Первому или Второму Законам.'''


pattern = re.compile('человек.*')

pattern.findall(asimov)
['человеку или своим бездействием допустить, чтобы человеку б
ыл причинён вред.',
 'человек, кроме тех случаев, когда эти приказы противоречат
Первому Закону.']
```

`*` и `+` стараются сожрать как можно больше символов в паттерн (почти как Уроборос).

Эскапирование (экранирование)

Что делать, если не хочется искать `\d` (то есть идущие друг за другом `\` и `d`) или просто `\`? Заэкранировать их ещё одним `\`! Стоит помнить, что в питоне `\` тоже специальный символ, поэтому придётся добавлять ещё один `\`, в результате паттерн будет выглядеть захламлённым. Чтобы этого не происходило, используйте `raw` строки, то есть ставьте букву `r` перед строкой с паттерном.

Разумеется, это далеко не всё, но этого хватит, чтобы начать.

Регэкспы просты для базового освоения и сложны для использования на уровне мастера, но при этом бывают очень полезны в рутине. Но сразу предостерегаем вас — если есть готовая библиотека для парсинга специфичного текста, то воспользуйтесь ею (html — beautiful soap, json — json), так как регэкспы в большинстве случаев — это решение, которое подходит, чтобы быстро решить задачку с текстом без определённого формата или с простым форматом (регулярным).

Документация

> Строковые методы пандаса

Для строковых колонок датафрейма есть аксессор `str`, содержащий множество методов работы со строками (по сути, векторизованные питоновские методы для строк). Вызов самого по себе `str` ничего особо не даёт:

```
df
```

	name
0	Aristotle
1	Zenon
2	Kant
3	Hume
4	Heidegger

```
df.name.str
```

```
<pandas.core.strings.StringMethods at 0x7fa8c58257b8>
```

Применение

Строковые методы из `str` применяются к каждой ячейке колонки. Например, проверим, начинаются ли значения колонки `name` на `'A'`:

```
df.name.str.startswith('A')
```

```
0     True
1    False
2    False
3    False
4    False
Name: name, dtype: bool
```

Что произошло: после обращения к атрибуту `str` мы вызвали метод `startswith`, в который передали строку `'A'`. Это аналогично вызову типа:

```
'Aristotle'.startswith('A')

Output:
True
```

Только мы делаем это со всей колонкой. В результате получаем такую же колонку булиновских значений — `True`, если начинается на `'A'`, и `False`, если не начинается. Исходная колонка не меняется, возвращается новая.

Слайсинг

Когда мы пишем `str`, то получаем доступ к строкам в колонке и как бы вызываем от них строковый метод. Еще мы можем делать срезы, например, возьмём первые 5 букв:

```
df.name.str[:5]
```

```
0    Arist
1    Zenon
2     Kant
3     Hume
4    Heide
Name: name, dtype: object
```

Что аналогично:

```
'Aristotle'[:5]

Output:
'Arist'
```

Сплит строк

По аналогии со сплитом обычной строки мы можем засплитить строки в серии и получить на каждую ячейку по списку. Наш датафрейм немного изменился:

```
df
```

	Info
0	Aristotle, (384 BC-322 BC)
1	Zeno of Elea, (c. 495 BC-c. 430 BC)
2	Immanuel Kant, (1724–1804)
3	David Hume, (1711–1776)
4	Martin Heidegger, (1889–1976)

```
df['info'].str.split(',')
```

```
0              [Aristotle,  (384 BC-322 BC)]
1    [Zeno of Elea,  (c. 495 BC-c. 430 BC)]
2              [Immanuel Kant,  (1724–1804)]
3                 [David Hume,  (1711–1776)]
4           [Martin Heidegger,  (1889–1976)]
Name: info, dtype: object
```

Здесь мы получили списки с двумя элементами.

Колонки со списками

При работе с такими колонками по спискам можно индексироваться и слайситься также при помощи атрибута `str`:

```
df['info'].str.split(',').str[0]
```

```
0           Aristotle
1        Zeno of Elea
2       Immanuel Kant
3          David Hume
4    Martin Heidegger
Name: info, dtype: object
```

Что при работе с одним списком аналогично:

```
['David Hume', ' (1711–1776)'][0]

Output:
'David Hume'
```

Документация

> Парсинг строковых колонок в пандас

Для извлечения данных из строк в пандас есть специальный метод — `extract`. Он принимает паттерн RE, позволяющий вытащить нужные куски из текста в отдельные колонки.

```
df
```

	info
0	Aristotle, (384 BC-322 BC)
1	Zeno of Elea, (c. 495 BC-c. 430 BC)
2	Immanuel Kant, (1724–1804)
3	David Hume, (1711–1776)
4	Martin Heidegger, (1889–1976)

Извлечём отсюда информацию об имени и даты жизни:

```
df['info'].str.extract('(?P<name>\w+), \((?P<data>.+)\)')
```

	name	data
0	Aristotle	384 BC-322 BC
1	Elea	c. 495 BC-c. 430 BC
2	Kant	1724–1804
3	Hume	1711–1776
4	Heidegger	1889–1976

Итак,

- `df['info'].str` — обращаемся к атрибуту со строковыми методами
- `extract` — вызываем метод, достающий части текста
- `(?P<name>\w+)` — это именованная группа, она как группа, только к ней можно обращаться по имени
- `(?P...)` — говорит питону, что это именованная группа
  - `<name>` — имя группы, в данном случае *name*

    - `\w+` — мэтчит буквы/цифры/подчёркивания, которые встречаются один или больше раз подряд
- `, \(` — запятая, пробел и скобочка, которые идут после первой группы. `\`, потому что символ скобки имеет специальное значение в RE
- `(?P<data>.+)` — другая именованная группа
    - `?P` — опять же, это идентификатор группы
    - `<data>` — имя группы *data*
    - `.+` — берёт любой символ один или больше раз подряд
- `\)` — скобочка после 2-й группы

Найдя в ячейке текст, подходящий под такое описание, `extract` вытащит его, разобьёт на указанные группы и поместит в новые колонки с именами, как в указанных группах. Данный паттерн не самый оптимальный, но не использует новых метасимволов.

`extract` возвращает новый датафрейм с экстрагированным текстом.

Документация

Отбор колонок по названию

В пандасе есть удобный метод отбора колонок или строк по их названию — `filter`. Кроме строк он также может работать с регэкспами, что позволяет гибко отбирать колонки.

```
pd.DataFrame.filter(items/like/regex, axis)
```

- `items` — принимает список с названиями колонок или строк, особой разницы по сравнению с `loc` нет
- `like` — принимает строку и возвращает все колонки, где в названии содержится строка, переданная в `like`
- `regex` — принимает строку, означающую паттерн RE, возвращает все колонки с названиями, которые мэтчатся на паттерн
- `axis` — параметр для обозначения того, отбираем мы колонки или строки, принимает `'columns'` или `'index'`, по умолчанию фильтрует колонки

Посмотрим на примере данных о перевозках.

Отберём все колонки с `'id'` в названии:

```
taxi.filter(like='id')
```

	journey_id	user_id	driver_id	taxi_id	rider_score
0	23a1406fc6a11d866e3c82f22eed4d4c	0e9af5bbf1edfe591b54ecdfd7e91e26	583949a89a9ee17d19e3ca4f137b6b4c	b12f4f09c783e29fe0d0ea624530db56	5.0
1	dd2af4715d0dc16eded53afc0e243577	a553c46e3a22fb9c326aeb3d72b3334e	NaN	NaN	NaN
2	dd91e131888064bf7df3ce08f3d4b4ad	a553c46e3a22fb9c326aeb3d72b3334e	NaN	NaN	NaN
3	dd2af4715d0dc16eded53afc0e2466d0	a553c46e3a22fb9c326aeb3d72b3334e	NaN	NaN	NaN
4	85b7eabcf5d84e42dc7629b7d27781af	56772d544fdfa589a020a1ff894a86f7	d665fb9f75ef5d9cd0fd89479380ba78	0accdd3aa5a322f4129fa20b53278c69	5.0

Как видите, мы получили только колонки с `'id'` в названии — все колонки с идентификаторами и `rider_score`.

А теперь возьмём по паттерну только колонки с идентификаторами:

```
taxi.filter(regex='_id')
```

	journey_id	user_id	driver_id	taxi_id
0	23a1406fc6a11d866e3c82f22eed4d4c	0e9af5bbf1edfe591b54ecdfd7e91e26	583949a89a9ee17d19e3ca4f137b6b4c	b12f4f09c783e29fe0d0ea624530db56
1	dd2af4715d0dc16eded53afc0e243577	a553c46e3a22fb9c326aeb3d72b3334e	NaN	NaN
2	dd91e131888064bf7df3ce08f3d4b4ad	a553c46e3a22fb9c326aeb3d72b3334e	NaN	NaN
3	dd2af4715d0dc16eded53afc0e2466d0	a553c46e3a22fb9c326aeb3d72b3334e	NaN	NaN
4	85b7eabcf5d84e42dc7629b7d27781af	56772d544fdfa589a020a1ff894a86f7	d665fb9f75ef5d9cd0fd89479380ba78	0accdd3aa5a322f4129fa20b53278c69

Документация