KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 5 урок > PYTHON

> Оглавление 5 урока

14. Парсинг строковых колонок и фильтрация колонок по названию

> Открывание сжатых файлов

У функции `pd.read_csv` есть аргумент compression, который принимает строчку типа компрессии и открывает заархивированный файл:

```
# Here type of compression is zip
ads_data = pd.read_csv('ads_data.csv.zip', compression='zip')
```

Больше информации

> Формат UNIX time

Время может быть указано в разном формате, один из них — число секунд, прошедших с 1970 года.

Кажется странным? (*Мне — да*)

Зато удобно — времена представляются как целые числа, которые легко вычитать и сравнивать, а при необходимости можно сконвертировать в human-readable формат:

```
pd.to_datetime(1554076848, unit='s')
```

Больше информации

> Атрибуты времени

Временные серии обладают атрибутом `dt`, в котором находится множество атрибутов и методов для доступа ко времени. Давайте посмотрим на часть из них:

```
df.start_at
```

```
0       2010-11-16 16:44:00
1       2010-06-01 00:34:00
2       2010-05-31 05:01:00
3       2010-06-01 00:29:00
4       2010-09-11 23:55:00
               ...
23106   2010-07-31 13:15:00
23107   2010-10-02 05:26:00
23108   2010-09-21 09:56:00
23109   2010-04-29 04:30:00
23110   2010-03-16 19:58:00
Name: start_at, Length: 23111, dtype: datetime64[ns]
```

Микросекунды

`dt.microsecond` — сколько микросекунд в указанном времени (то есть, если время 5 минут, 0 секунд и 3 микросекунды, то он вернёт 3, а не 5 * 60 * 106)

```
df.start_at.dt.microsecond
```

```
0        0
1        0
2        0
3        0
4        0
        ..
23106    0
23107    0
23108    0
23109    0
23110    0
Name: start_at, Length: 23111, dtype: int64
```

Секунды

```
df.start_at.dt.second
```

```
0        0
1        0
2        0
3        0
4        0
        ..
23106    0
23107    0
23108    0
23109    0
23110    0
Name: start_at, Length: 23111, dtype: int64
```

Минуты

```
df.start_at.dt.minute
```

```
0        44
1        34
2         1
3        29
4        55
        ..
23106    15
23107    26
23108    56
23109    30
23110    58
Name: start_at, Length: 23111, dtype: int64
```

Час

```
df.start_at.dt.hour
```

```
0        16
1         0
2         5
3         0
4        23
        ..
23106    13
23107     5
23108     9
23109     4
23110    19
Name: start_at, Length: 23111, dtype: int64
```

День месяца

```
df.start_at.dt.day
```

```
0        16
1         1
2        31
3         1
4        11
         ..
23106    31
23107     2
23108    21
23109    29
23110    16
Name: start_at, Length: 23111, dtype: int64
```

Номер дня в недели

```
df.start_at.dt.weekday
```

```
0        1
1        1
2        0
3        1
4        5
        ..
23106    5
23107    5
23108    1
23109    3
23110    1
Name: start_at, Length: 23111, dtype: int64
```

Имя дня в недели

```
df.start_at.dt.day_name()
```

```
0         Tuesday
1         Tuesday
2          Monday
3         Tuesday
4        Saturday
           ...
23106    Saturday
23107    Saturday
23108     Tuesday
23109    Thursday
23110     Tuesday
Name: start_at, Length: 23111, dtype: object
```

Номер недели в году

```
df.start_at.dt.week
```

```
0        46
1        22
2        22
3        22
4        36
        ..
23106    30
23107    39
23108    38
23109    17
23110    11
Name: start_at, Length: 23111, dtype: int64
```

Номер месяца

```
df.start_at.dt.month
```

```
0        11
1         6
2         5
3         6
4         9
        ..
23106     7
23107    10
23108     9
23109     4
23110     3
Name: start_at, Length: 23111, dtype: int64
```

Название месяца

```
df.start_at.dt.month_name()
```

```
0          November
1              June
2               May
3              June
4         September
             ...
23106         July
23107      October
23108    September
23109        April
23110        March
Name: start_at, Length: 23111, dtype: object
```

Год

```
df.start_at.dt.year
```

```
0        2010
1        2010
2        2010
3        2010
4        2010
        ...
23106    2010
23107    2010
23108    2010
23109    2010
23110    2010
Name: start_at, Length: 23111, dtype: int64
```

Число дней в текущем месяце

```
df.start_at.dt.daysinmonth
```

```
0        30
1        30
2        31
3        30
4        30
        ..
23106    31
23107    31
23108    30
23109    30
23110    31
Name: start_at, Length: 23111, dtype: int64
```

Разность времени

Timedelta — это тип данных, соответствующий разнице двух времён, то есть, какая-то продолжительность времени.

```
df.wait_time
```

```
0       -1 days +23:42:00
1                     NaT
2                     NaT
3                     NaT
4                00:05:00
               ...
23106            00:00:00
23107   -1 days +23:47:00
23108   -1 days +23:51:00
23109            00:07:00
23110                 NaT
Name: wait_time, Length: 23111, dtype: timedelta64[ns]
```

Компоненты

Все единицы измерения времени можно извлечь сразу с помощью атрибута components

```
df.wait_time.dt.components
```

	days	hours	minutes	seconds	milliseconds	microseconds	nanoseconds
0	-1.0	23.0	42.0	0.0	0.0	0.0	0.0
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	0.0	0.0	5.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...
23106	0.0	0.0	0.0	0.0	0.0	0.0	0.0
23107	-1.0	23.0	47.0	0.0	0.0	0.0	0.0
23108	-1.0	23.0	51.0	0.0	0.0	0.0	0.0
23109	0.0	0.0	7.0	0.0	0.0	0.0	0.0
23110	NaN	NaN	NaN	NaN	NaN	NaN	NaN

23111 rows × 7 columns

Больше информации

> quantile

Метод для поиска определённых перцентилей. Принимает число от 0 до 1, обозначающее перцентиль в виде доли:

- 0 — нулевой перцентиль
- 0.1 — десятый перцентиль
- 0.75 — 75-й перцентиль (также третий квартиль)

```
df
```

	values
0	1
1	2
2	3
3	4
4	5

```
df.quantile(q=0.75)
```

```
values    4.0
Name: 0.75, dtype: float64
```

Также в `q` можно передать список всех желаемых перцентилей:

```
df.quantile(q=[0.5, 0.7])
```

	values
0.5	3.0
0.7	3.8

Что делать со значениями, не попадающими в перцентиль?

Если ровно по заданному перцентилю в датафрейме нет значения, то по умолчанию метод линейно выведет его. Поменять это поведение можно с помощью параметра *interpolation*. Вариант 'higher' берёт большую точку из смежных:

```
df.quantile(q=[0.5, 0.7], interpolation='higher')
```

	values
0.5	3
0.7	4

P.S.: Краткость — сестра таланта, но не в создании названий параметров в одну букву. Здесь сложно перепутать, так как название метода намекает, но когда будете создавать свои функции и методы, называйте всё осмысленно.

Документация

> Сводные таблицы

Сводные таблицы — удобный способ преобразовать данные, с возможностью применения к ним агрегирующей функции. В pandas есть 2 функции, различающиеся только тем, проводится ли агрегация.

Обе принимают 3 аргумента:

- `index` — название колонки, значения из которой станут индексами.
- `columns` — название колонки, значения из которой станут колонками.
- `values` — название колонки, значения из которой распределяться по сформированным группам.

Теперь разберем сами функции:

pivot

Преобразует датафрейм в таблицу, где значения использованных колонок становятся новыми индексами и колонками:

```
events_by_day
```

	date_day	event	events
0	2019-04-01	click	881
1	2019-04-01	view	41857
2	2019-04-02	click	1612
3	2019-04-02	view	165174
4	2019-04-03	click	1733
5	2019-04-03	view	224843
6	2019-04-04	click	1447
7	2019-04-04	view	107098
8	2019-04-05	click	581790
9	2019-04-05	view	2050279

```
events_by_day.pivot(index='date_day', columns='event', values='events')
```

event	click	view
date_day		
2019-04-01	881	41857
2019-04-02	1612	165174
2019-04-03	1733	224843
2019-04-04	1447	107098
2019-04-05	581790	2050279

Документация

pivot_table

Всё как в предыдущем методе, только можно произвести агрегацию, получая одно значение из группы с одинаковыми значениями в новых индексах и колонках. По умолчанию берётся среднее от группы значений.

```
df
```

	date_day	event	events
0	2019-04-01	click	10
1	2019-04-02	click	20
2	2019-04-02	click	30
3	2019-04-01	view	15
4	2019-04-03	view	20
5	2019-04-03	view	45

```
df.pivot_table('events', index='date_day', columns='event')
```

event	click	view
date_day		
2019-04-01	10.0	15.0
2019-04-02	25.0	NaN
2019-04-03	NaN	32.5

```
df.pivot_table('events', index='date_day', columns='event', aggfunc='max')
```

event	click	view
date_day		
2019-04-01	10.0	15.0
2019-04-02	30.0	NaN
2019-04-03	NaN	45.0

Документация

> Альтернативный способ создания колонок

Колонки в датафрейме можно также создать с помощью метода `assign`. Он возвращает исходный датафрейм с добавленными колонками — нужно перезадать переменную, чтобы изменить датафрейм.

В метод передаются аргументы формата `название колонки = её содержимое` — как название параметра и его значение при вызове функции. Здесь название колонок нужно писать без кавычек.

Так, мы задаём новую колонку `loyality`, значения в которой являются отношением значений в колонке `max_orders` к значениям в колонке `orders`:

```
users_data = users_data.assign(loyality = users_data.max_orders / users_data.orders)
```

Больше информации

> Замена пропущенных значений

Замена пропущенных значений — часто возникающая задача в некоторых сферах. Одно из простых решений — заменить все на одно значение (например, 0). Это можно сделать с помощью метода `fillna`, принимающего значение, на которое будут заменены пропущенные значения:

```
retention_pivot.head(3)
```

date	2019-04-01	2019-04-02	2019-04-03	2019-04-04	2019-04-05	2019-04-06
start_date						
2019-04-01	1.0	0.6	0.4	0.3	0.2	0.1
2019-04-02	NaN	1.0	0.6	0.4	0.3	0.2
2019-04-03	NaN	NaN	1.0	0.6	0.4	0.3

```
retention_pivot.head(3).fillna(0)
```

date start_date	2019-04-01	2019-04-02	2019-04-03	2019-04-04	2019-04-05	2019-04-06
2019-04-01	1.0	0.6	0.4	0.3	0.2	0.1
2019-04-02	0.0	1.0	0.6	0.4	0.3	0.2
2019-04-03	0.0	0.0	1.0	0.6	0.4	0.3

Документация

> loc

До этого при отборе определённых строк мы пользовались методом query или использовали `[]`. Также существует метод `loc` (и его собрат `iloc`), позволяющий выбрать поднабор строк и колонок из датафрейма. В некоторых случаях `loc` удобнее, но обычно запись с ним более громоздкая и он работает медленнее `query`

```
df
```

	1	2	3	4	5	6	7	8	9	10
0	1.0	1.0	NaN	1.0	NaN	NaN	1.0	0.0	0.0	NaN
1	NaN	1.0	0.0	1.0	NaN	1.0	NaN	NaN	0.0	0.0
2	NaN	1.0	NaN	1.0	NaN	1.0	0.0	0.0	NaN	NaN
3	1.0	1.0	NaN	NaN	NaN	NaN	NaN	1.0	0.0	NaN

Эта запись отберёт все строки из датафрейма, где значения в колонке `'6'` равны 1, и колонки от `'1'` до `'6'`

```
df.loc[df['6'] == 1, '1':'6']
```

	1	2	3	4	5	6
1	NaN	1.0	0.0	1.0	NaN	1.0
2	NaN	1.0	NaN	1.0	NaN	1.0

Кусок `df['6'] == 1` возвращает логическую серию, где напротив нужных значений стоит `True`. Вместо `df['6'] == 1` может быть любое выражение, которое даст коллекцию `True`/`False` размером с число строк в датафрейме (в нашем случае — 4)

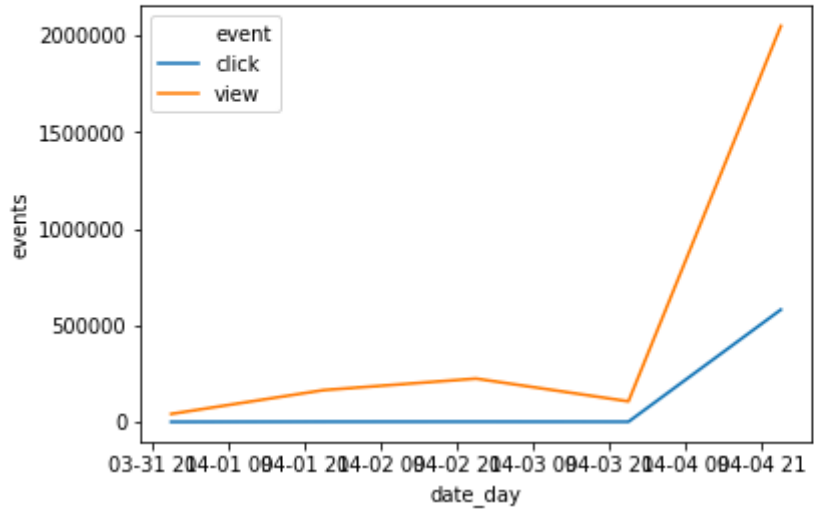
Документация

> Lineplot

Line chart – линейная диаграмма. По оси x и y откладываются значения точек, эти точки соединяются. Аргумент `hue` принимает имя колонки, по значениям которой идёт разделение на цвета:

```
sns.lineplot(x="date_day", y="events", hue="event", data=events_by_day)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fedf47bae80>
```



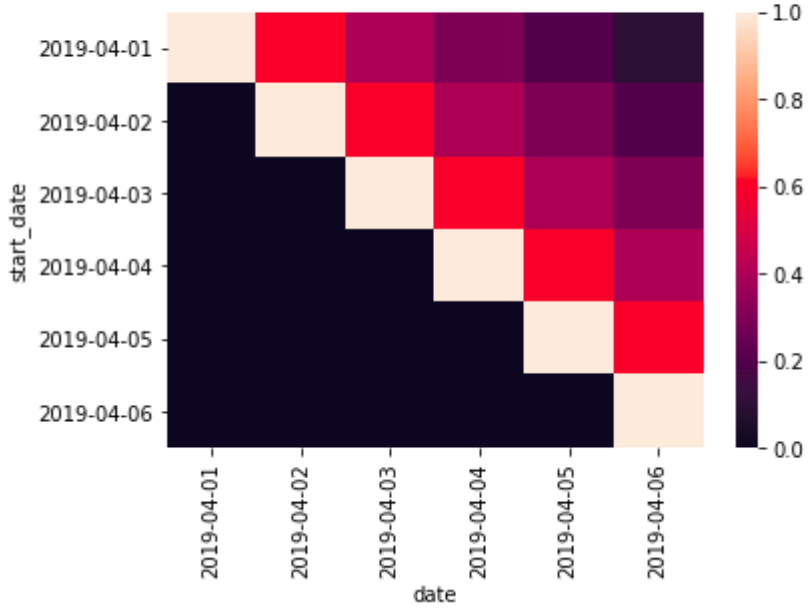
Документация

> Heatmap

Удобный тип графика, когда есть множество значений с двумя категориальными признаками (обычно это индекс и колонки в датафрейме). По осям откладываются значения этих категориальных переменных, каждая ячейка — значение, которое мы визуализируем. Интенсивность ячейки пропорциональна значению.

```
sns.heatmap(retention_pivot)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fedde4a6f98>
```



Документация

> 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 пользователях, и 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. Не углубляясь в определения, 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 строковые методы 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 РЕ, 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` — импотиртируем библиотеку 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, и помещаем его в переменную pattern. Паттерн создаётся при помощи строки — о том, с чем совпадает (что матчит) этот паттерн мы поговорим дальше.Паттерн обладает набором методов (также, как у датафрэйма 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 непонятно (и неудивительно — мы ещё не обсуждали как описывается паттерн) и бессмысленно, ведь есть `split`. Однако у этого способа уже на чутьболее сложной задаче есть плюсы:

```
text = '''We have several emails - vasya@yandex.ru, katya_ivanova@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.

Регэксповая песочница

> Азбука регулярных выражений

Как уже говорилось, РЕ это язык о регулярном языке. Для его описания используется набор символов. Рассмотрим наиболее частые из них:

Буквы и цифры

Все буквы и цифры в паттерне обозначают буквы. То есть, 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 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 этоЕщё 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, ч
тобы человеку был причинён вред.
Робот должен повиноваться всем приказам, которые даёт человек, кроме тех случаев, когд
а эти приказы противоречат Первому Закону.
Робот должен заботиться о своей безопасности в той мере, в которой это не противоречит
Первому или Второму Законам.'''


pattern = re.compile('в...')
pattern.findall(asimov)
```

```
['вред', 'веку', 'воим', 'вием', 'веку', 'вред', 'вино', 'вать', '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 заматчившегося 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 матча. Это позволяет нам извлечь нужную группу из каждого матча (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 матчим в тексте 3 цифры, минус, 3 цифры. Ео есть он должен быть в тексте, чтобы заматчить, 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 (0 — бесконечность)
- `+` — 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 где есть

```
asimov = '''Робот не может причинить вред человеку или своим бездействием допустить, ч
тобы человеку был причинён вред.
```

```
Робот должен повиноваться всем приказам, которые даёт человек, кроме тех случаев, когд
а эти приказы противоречат Первому Закону.
Робот должен заботиться о своей безопасности в той мере, в которой это не противоречит
Первому или Второму Законам.'''


pattern = re.compile('человек.*')
pattern.findall(asimov)
['человеку или своим бездействием допустить, чтобы человеку был 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 хочется искать `\d` (то есть идущие друг за другом `\` и `d`) или просто `\`? Заэкранировать их ещё одним `\`! Однако, стоит помнить, что в питоне `\` тоже специальный символ, поэтому придётся добавлять ещё один `\` и 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 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 специальный атрибут `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')
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.

Или перевод всех значений к верхнему регистру:

```
df.name.str.upper()
```

```
0    ARISTOTLE
1        ZENON
2         KANT
3         HUME
4    HEIDEGGER
Name: name, dtype: object
```

Слайсинг

Когда мы пишем `str`, то получаем доступ к строкам в колонке, и как бы вызываем от них строковый метод. Также 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]
'Arist'
```

Сплит строк

По аналогии со сплитом обычной строки, мы можем засплиттить 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]
'David Hume'
```

Документация

> Парсинг строковых колонок в пандас

Чтобы извлечь данные 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+` — матчит буквы/цифры/подчёркивания, которые встречаются один или больше раз подряд

- `, \(` — запятая, пробел и скобочка, которые идут после первой группы. `\`, потому что символ скобки имеет специальное значение в PE.
- `(?P<data>.+)` — другая именованная группа
  - `?P` — опять же, это идентификатор группы
  - `<data>` — имя группы *data*
  - `.+` — берёт любой символ один или больше раз подряд
- `\)` — 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 датафрэйм 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 PE возвращает все колонки с названиями, матчимящимися 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 перевозкахОтберём 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