Введение и основы синтаксиса

Переменные и типы данных

Создать и использовать переменные

In
```
english = 378.2
russian = 153.9
german = 76.0
chinese = 908.7

top3_total = english+russian+german

print(chinese - top3_total)
```

Out
```
300.6
```

Узнать тип переменной

In
```
russian_web_part = 0.061
print(type(russian_web_part))
```

Out
```
<class 'float'>
```

Прокомментировать код

In
```
# число сайтов с китайским языком
# популярных сайтов среди 10 млн самых
print(0.017 * 10000000)
```

Out
```
170000.0
```

Преобразовать float в int и наоборот

In
```
russian_web_popular_2 = int(russian_web_popular)
english_native_2 = float(english_native)
```

Вывод на экран

Напечатать текст на экране

In
```
print("Исследование распространённости языков.")
```

Out
```
Исследование распространённости языков.
```

Вывести дробь функцией format()

In
```
print("Индекс проникновения в интернет: {:.2f}".format(2.31))
```

Out
```
Индекс проникновения в интернет: 2.31
```

Вывести проценты функцией format()

In
```
print("Доля сайтов с языком: {:.1%}".format(0.061))
```

Out
```
Доля сайтов с языком: 6.1 %
```

Списки и циклы

Списки, строки и циклы

Распечатать или изменить элемент списка

In
```
emojixpress = [2.26, 19.1, 25.6, 233.0, 15.2]

# распечатать элемент списка emojixpress с индексом 0
print(emojixpress[0])

# присвоить элементу списка emojixpress с индексом 4 новое значение
emojixpress[4] = 100500.0 # значение выбрано произвольно для примера
```

Out
```
2.26
```

Просуммировать элементы списка

In
```
emojixpress = [
2.26, 19.1, 25.6, 233.0, 15.2, 22.7, 64.6, 87.5, 6.81, 6.0,
4.72, 24.7, 21.7, 10.0, 118.0, 3.31, 23.1, 1.74, 4.5, 0.0333
]

total = 0
for count in emojixpress:
    total += count

print("{:.2f}".format(total))
```

Out
```
694.57
```

Вычислить длину списка или строки

In
```
emojixpress = [2.26, 6.8, 25.6, 233.0,
15.2, 22.7, 64.6, 87.5, 19.1, 3.31]

print(len(emojixpress))
```

Out
```
10
```

In
```
message = "I love you"


print(len(message))
```

Out
```
10
```

Форматирование

Выровнять текст

In
```
print("|{: <20}|".format("Ухмыляюсь"))
print("|{: >20}|".format("Ухмыляюсь"))
print("|{: ^20}|".format("Ухмыляюсь"))
```

Out
```
|Ухмыляюсь           |
|          Ухмыляюсь|
|     Ухмыляюсь      |
```

Выровнять и вывести с заданной точностью

In
```
print("|{: <20.2f}|".format(233.0))
print("|{: >20.1f}|".format(2270.0))
print("|{: ^20.1%}|".format(0.61))
```

Out
```
|233.00              |
|              2270.0|
|       61.0%        |
```

Операции с таблицами

Списки

Получить из списка диапазон

In
```
digits_names = [
'ноль', 'один', 'два', 'три', 'четыре', 'пять', 'шесть', 'семь', 'восемь', 'девять'
]

# указываем обе границы диапазона (правая не включается)
print(digits_names[4:7])

# опускаем левую границу — идём с начала списка
print(digits_names[:5])

# опускаем правую границу — идём до конца списка
print(digits_names[7:])
```

Out
```
['четыре', 'пять', 'шесть']
['ноль', 'один', 'два', 'три', 'четыре']
['семь', 'восемь', 'девять']
```

Добавить к списку элемент в конец

In
```
emoji = ['Ухмыляюсь', 'Сияю от радости', 'Катаюсь от смеха', 'Слёзы радости']

print(emoji)

emoji.append('Подмигиваю')

print(emoji)
```

Out
```
['Ухмыляюсь', 'Сияю от радости', 'Катаюсь от смеха', 'Слёзы радости']
['Ухмыляюсь', 'Сияю от радости', 'Катаюсь от смеха', 'Слёзы радости', 'Подмигиваю']
```

Отсортировать таблицу (список списков) по столбцу

In
```
data.sort(key=lambda row: row[1], reverse=True)
```

Циклы

Получить диапазон чисел или повторить код

In
```
for element in range(5):
    print(element)
```

Out
```
0
1
2
3
4
```

In
```
for i in range(3):
    print("*****")
```

Out
```
*****
*****
*****
```

Изменить список в цикле

In
```
...

for i in range(len(data)):
  part = data[i][1]/emojixpress_total
  data[i].append(part)

...
```

Выровнять и вывести с заданной точностью

In
```
print('Анализ ', end='')
print('эмодзи')
```

Out
```
Анализ эмодзи
```

Функции, переменные и условия

Проверить условие

In
```
def check_if_recent(year):
    if year < 2008:
        print("Фильм был снят давно.")
    else:
        print("Фильм свежий.")
```

Проверить наличие элемента в списке

In
```
if "криминал" in genres:
    ...
```

Создать функцию с аргументом и возвращаемым значением

In
```
def dollars_to_rubles(dollars):
    rubles = dollars*rubles_for_dollar
    return rubles
```

Объединить или изменить условия

In
```
# условие должно быть НЕ выполнено
if not year > 2007:
    ...
```

In
```
# должно быть выполнено
# одно условие И ЕЩЁ другое
if year > 2000 and "история" in genres:
    ...
```

In
```
# должно быть выполнено
# одно условие ИЛИ ХОТЯ БЫ условие
if "фантастика" in genres or
   "фэнтези" in genres:
    ...
```

Создать константу

In
```
RUBLES_FOR_DOLLAR = 67.01
```

Pandas для анализа данных

Вызов библиотеки pandas

Вызов библиотеки pandas

```
In  import pandas
    import pandas as pd
```

Конструктор DataFrame() для создания таблицы

```
In  pd.DataFrame(data = data, columns =
    columns)
    # аргумент data – список с данными,
    # аргумент columns – список с
    # названиями столбцов
```

Метод tail() для вывода последних строк таблицы

```
In  df.tail() # последние 5 строк
    df.tail(15) # последние 15 строк
```

Метод read_csv() для чтения файлов формата CSV

```
In  df = pd.read_csv('путь к файлу')
```

Метод head() для вывода первых строк таблицы

```
In  df.head() # первые 5 строк
    df.head(10) # первые 10 строк
```

Атрибут columns для вывода названий столбцов

```
In  df.columns
```

Атрибут shape для вывода размера таблицы

```
In  df.shape
```

Атрибут dtypes для получения информации о типах данных в таблице

```
In  df.dtypes
```

Метод info() для просмотра сводной информации о таблице

```
In  df.info()
```

Атрибут loc[строка, столбец] даёт доступ к элементу в DataFrame по строке и столбцу

```
In  df.loc[:, 'column']
```

```
Out
```

Вид	Реализация
Одна ячейка	`.loc[7, 'column']`
Один столбец	`.loc[:, 'column']`
Несколько столбцов	`.loc[:, ['column_1', 'column_4']]`
Несколько столбцов подряд (срез)	`.loc[:, 'column_5': 'column_8']`
Одна строка	`.loc[1]`
Все строки, начиная с заданной	`.loc[1:]`
Все строки до заданной	`.loc[:3]`
Несколько строк подряд (срез)	`.loc[2:5]`

Логическая индексация для получения элементов по определенному условию

```
Out
```

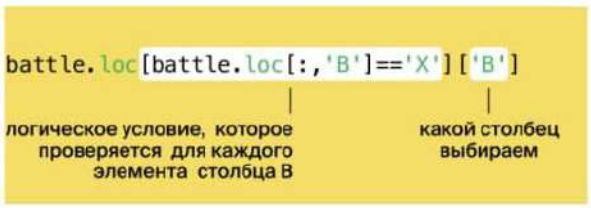
Вид	Реализация	Сокращенная запись
Все строки, удовлетворяющие условию	`'df.loc[df.loc[:,'column'] == 'X']'`	`'df[df['column'] == 'X']'`
Столбец, удовлетворяющий условию	`'df.loc[df.loc[:,'column'] == 'X']['column']'`	`'df[df['column'] == 'X']['column']'`
Применение метода	`'df.loc[df.loc[:,'column'] == 'X']['column'].count()'`	`'df[df['column'] == 'X']['column'].count()'`

Индексация в Series

```
Out
```

Вид	Реализация	Сокращенная запись
Один элемент	`'df.loc[7]'`	`'df[7]'`
Несколько элементов	`'df.loc[[5, 7, 10]]'`	`'df[[5, 7, 10]]'`
Несколько элементов подряд (срез)	`'df.loc[5:10]'` включая 10	`'df[5:10]'` не включая 10
Все элементы, начиная с заданного	`'df.loc[1:]'`	`'df[1:]'`
Все элементы до заданного	`'df.loc[:3]'` включая 3	`'df[:3]'` не включая 3

ВИД	РЕАЛИЗАЦИЯ	СОКРАЩЁННАЯ ЗАПИСЬ
Одна ячейка	`.loc[7, 'genre']`	-
Один столбец	`.loc[:, 'genre']`	`df['genre']`
Несколько столбцов	`.loc[:, ['genre', 'Artist']]`	`df [['genre', 'Artist']]`
Несколько столбцов подряд (срез)	`.loc[:, 'user_id': 'genre']`	-
Одна строка	`.loc[1]`	-
Все строки, начиная с заданной	`.loc[1:]`	`df[1:]`
Все строки до заданной	`.loc[:3]` включая 3	`df[:3]` не включая 3
Несколько строк подряд (срез)	`.loc[2:5]` включая 5	`df[2:5]` не включая 5



ВИД	РЕАЛИЗАЦИЯ	СОКРАЩЁННАЯ ЗАПИСЬ
Все строки, удовлетворяющие условию	`battle.loc[battle.loc[:,'B'] == 'X']`	`battle[battle['B'] == 'X']`
Столбец, удовлетворяющий условию	`battle.loc[battle.loc[:,'B'] == 'X']['B']`	`battle[battle['B'] == 'X']['B']`
Применение метода	`battle.loc[battle.loc[:,'B'] == 'X']['B'].count()`	`battle[battle['B'] == 'X']['B'].count()`

Посчитать в каждом столбце отсутствующие значения можно методом `.isnull()`. Если значение элемента не существует, `.isnull()` возвращает True, а иначе — False. Суммируют эти True вызовом метода `sum()`, который в этом случае возвращает общее число элементов без определённых значений.

PYTHON

```
print(cholera.isnull().sum())
```

Также подойдёт метод `isna()`, подсчитывающий пустые значения. В таблице по холере пропущенные значения качественные, так что этот метод отыщет их все.

PYTHON

```
print(cholera.isna().sum())
```

Чтобы не лишиться строк с важными данными, заполним значения NaN в столбце *'imported_cases'* нулями.

Для этого лучше всего использовать метод `fillna()`, где в качестве аргумента выступает заменитель отсутствующих значений.

От строк с нулевыми значениями избавляются методом `dropna()`. Он удаляет любую строку, где есть хоть одно отсутствующее значение.

У этого метода есть аргументы:

1. `subset = [ ]`. Его значением указывают названия столбцов, где нужно искать пропуски.
2. Уже знакомый нам *inplace*.

```
cholera.dropna(subset = ['total', 'deaths', 'case_fatality_rate'], inplace = True)
```

название таблицы

название столбцов, где нужно искать пропуски

PYTHON

```
cholera.dropna(subset = ['total_cases', 'deaths', 'case_fatality_rate' ], inplace = True)
```

Теперь удалим правый столбец с пропущенными значениями. Снова вызываем метод `dropna()`. Как и `set_axis()`, он имеет ещё и аргумент *axis*. Если этому аргументу присвоить значение `'columns'`, он удалит любой столбец, где есть хоть один пропуск.

название таблицы

```
cholera.dropna(axis='columns', inplace = True)
```

удаление столбца, где есть хоть одно пропущенное значение

PYTHON

```
cholera.dropna(axis = 'columns', inplace = True)
```

Грубые дубликаты — повторы — обнаруживают методом `duplicated()`. Он возвращает `Series` со значением True при наличии дубликатов, и False, когда их нет.

```python
print(df.duplicated())
```

Чтобы посчитать количество дубликатов в наборе данных, нужно вызвать метод `sum()`:

```python
print(df.duplicated().sum())
```

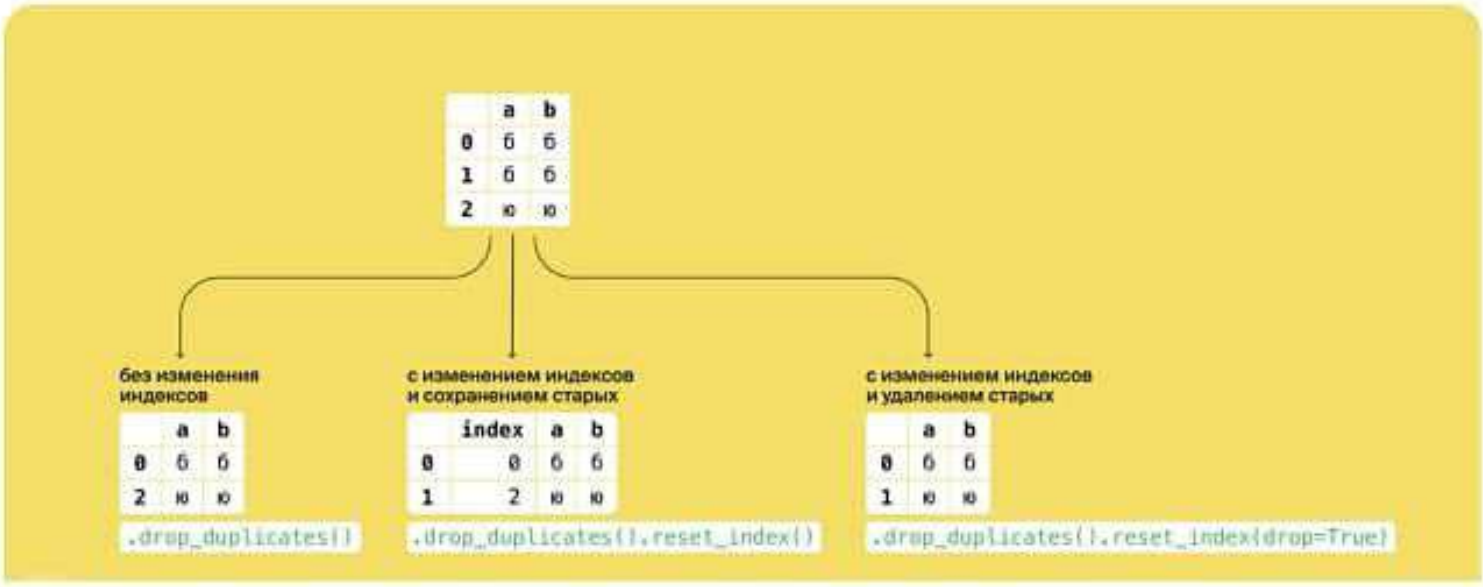
Для удаления дубликатов есть метод `drop_duplicates()`:

```python
df.drop_duplicates(inplace = True)
```

При вызове метода `drop_duplicates()` вместе с повторяющимися строками удаляются их индексы.

Последовательность индексов нарушается: после 0 следует 2 и т.д.

Поэтому вызов `drop_duplicates()` соединяют в цепочку с вызовом метода `reset_index()`. Тогда создаётся новый `DataFrame`, где старые индексы превращаются в обычный столбец под названием *index*, а индексы всех строк снова следуют в естественном порядке. Если же мы не хотим создавать новый столбец *index*, то при вызове `reset_index()` передаётся аргумент `drop` со значением True. Все индексы переписываются в порядке возрастания, без пропусков.



Вот такой код сохраняет в переменной *df* таблицу, очищенную от дубликатов, с новой индексацией.

```
df = df.drop_duplicates().reset_index(drop=True)
```

Предобработка данных

Синтаксис

Метод set_axis() для изменения названий столбцов

```
In  df.set_axis(['a','b','c'],axis = 'columns',inplace = True)

    # аргументы – список новых названий столбцов,
    # axis со значением columns для изменений в столбцах,
    # inplace со значением True для изменения структуры данных
```

Методы isnull() и isna() для определения пропущенных значений

```
In  df.isnull()
    df.isna()

    # В сочетании с методом sum() –
    # подсчёт пропущенных значений
    df.isnull().sum()
    df.isna().sum()
```

Метод fillna() для заполнения пропущенных значений

```
In  df = df.fillna(0)
    # аргумент – значение, на которое
    # будут заменены пропущенные значения
```

Метод dropna() для удаления пропущенных значений

```
In  df.dropna()
    # удаление всех строк, где есть
    # хотя бы одно пропущенное значение
```

```
In  df.dropna(subset = ['a','b','c'],
    inplace = True)
    # аргумент subset – названия столбцов,
    # где нужно искать пропуски
```

```
In  df.dropna(axis = 'columns',
    inplace = True)
    # аргумент axis со значением 'columns'
    # для удаления столбцов с хотя бы
    # одним пропущенным значением
```

Метод duplicated() для нахождения дубликатов

```
In  df.duplicated()

    # В сочетании с методом sum() –
    # возвращает количество дубликатов
    df.duplicated().sum()
```

Метод unique() для просмотра всех уникальных значений в столбце

```
In  df['column'].unique()
```

Метод drop_duplicates() для удаления дубликатов

```
In  df.drop_duplicates().reset_index(drop
     = True)
    # аргумент drop со значением True,
    # чтобы не создавать столбец со
    # старыми значениями индексов

    '''
    При вызове метода drop_duplicates()
    вместе с повторяющимися строками
    удаляются их индексы, поэтому
    используется с методом reset_index()
    '''
```

Метод replace() для замены значений в таблице или столбце

```
In  df.replace('first_value', 'second_value')

    # первый аргумент – текущее значение
    # второй аргумент – новое значение
```

В Pandas для группировки данных есть метод `groupby()`. Он принимает как аргумент название столбца, по которому нужно группировать. В случае с делением экзопланет по годам открытия:

```
print(exoplanet.groupby('discovered'))

exo_number = exoplanet.groupby('discovered')['radius'].count()
print(exo_number)

exo_radius_sum = exoplanet.groupby('discovered')['radius'].sum()
print(exo_radius_sum)
```

Поиск необычного в группе — что среди планет, что среди меломанов — это прежде всего поиск чемпионов: объектов с выдающимися показателями по разным статьям. Как всю таблицу, так и отдельные группы изучают, сортируя строки по какому-либо столбцу. В Pandas для этой операции есть метод `sort_values()`. У него два аргумента:

- *by* = 'имя столбца' — имя столбца, по которому нужно сортировать;
- *ascending* : по умолчанию True. Для сортировки по убыванию установите значение False.


```
print(exoplanet.sort_values(by = 'radius').head(30))
```

А чтобы не тратить время на лишнее, поставим оба условия сразу. Для этого в Pandas есть логический оператор **&**, подобный оператору *and* языка Python. Напомним, его смысл на русском языке можно передать словами «и ещё»:

```
# экзопланеты меньше Земли __ и ещё __ открытые в 2014 году
exo_small_14 = exoplanet[ (exoplanet['radius']<1) & (exoplanet['discovered']==2014)]
print(exo_small_14)
```

От строк с нулевыми значениями избавляются методом `dropna()`. Он удаляет любую строку, где есть хоть одно отсутствующее значение.

У этого метода есть аргументы:

1. `subset = [ ]`. Его значением указывают названия столбцов, где нужно искать пропуски.
2. Уже знакомый нам *inplace*.

```
cholera.dropna(subset = ['total', 'deaths', 'case_fatality_rate'], inplace = True)
```

название таблицы

название столбцов, где нужно искать пропуски

PYTHON

```
cholera.dropna(subset = ['total_cases', 'deaths', 'case_fatality_rate' ], inplace = True)
```

Анализ данных и оформление результатов

Синтаксис

Метод groupby ('название столбца') для группировки данных

In
```
df.groupby('название столбца')

# группировка по столбец_1
# и вывод столбец_2
df.groupby('столбец_1')['столбец_2']

# подсчёт количества в группе
df.groupby('название столбца').count()

# подсчёт суммы в группе
df.groupby('название столбца').sum()
```

Метод sort_values (by = 'название столбца') для сортировки таблицы по указанному столбцу

In
```
# сортировка по возрастанию
# (значение по умолчанию)
df.sort_values(by = 'название столбца')

# сортировка по убыванию
df.sort_values(by = 'название столбца',
    ascending = False)
```

Метод max ( ) для определения максимального значения

In
```
df['название столбца'].max()
# максимальное значение в столбце
```

Метод min ( ) для определения минимального значения

In
```
df['название столбца'].min()
# минимальное значение в столбце
```

Метод mean ( ) для расчёта среднего арифметического

In
```
df['название столбца'].mean()
# среднее значение по столбцу
```

Метод median ( ) для расчёта медианы

In
```
df['название столбца'].median()
# медиана по столбцу
```

Работа с пропусками

Уникальные значения столбца и их количество

```
In  data['column'].value_counts()
```

Арифметические операции со столбцами

```
In  # NB!: столбцы должны иметь числовой тип

    data['column1'] = data['column2'] + data['column3']
    data['column1'] = data['column2'] - data['column3']
    data['column1'] = data['column2'] * data['column3']
    data['column1'] = data['column2'] / data['column3']
```

Расчёт параметров числовых столбцов

```
In  data['column'].sum()    # сумма значений
    data['column'].min()    # минимум
    data['column'].max()    # максимум
    data['column'].mean()   # среднее значение
    data['column'].median() # медиана в столбце
```

```
In  data['column'].count()  # количество значений в столбце
```

Применение нескольких операций к столбцу при группировке

```
In  data.groupby('column1').agg({'column2': ['count', 'sum'], 'column3': ['min', 'max']})
```

Вызвав к столбцу `'source'` метод `value_counts()`, который возвращает уникальные значения и количество их упоминаний, определим, сколько раз источник трафика был пропущен.

```
import pandas as pd

logs = pd.read_csv('/datasets/logs.csv')
print(logs['source'].value_counts())
```

Вызовем метод **agg()**, указывающий, какие именно функции применить к столбцу *'purchase'*. Название столбца и сами функции запишем в особую структуру данных — **словарь**. Словарь состоит из **ключа** и **значения**:

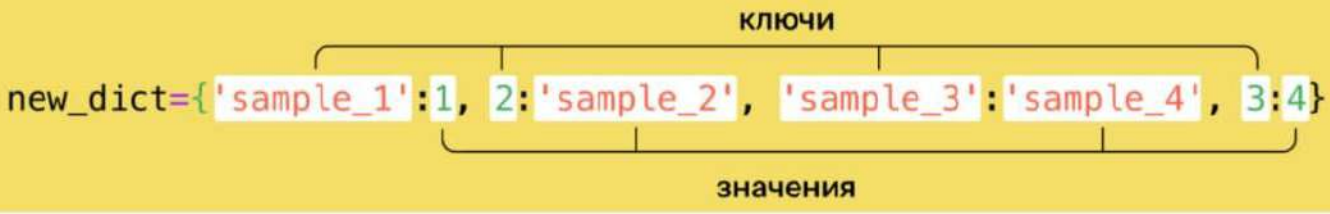
Скопировать код PYTHON

```
{'purchase':['count','sum']}
```

Здесь ключ — это название столбца, к которому нужно применить функции, а значением выступает список с названиями функций.

Результат выполнения кода сохраним в переменной `logs_grouped`:

```
logs_grouped = logs.groupby('source').agg({'purchase': ['count', 'sum']})
```



Собрать все строки да сделать их числами! Для этого есть стандартный метод Pandas — **to_numeric()**. Он превращает значения столбца в числовой тип `float64` (вещественное число).

```python
transactions['amount'] = pd.to_numeric(transactions['amount'])
```

У метода *to_numeric()* есть параметр **errors**. От значений, принимаемых *errors*, зависят действия *to_numeric* при встрече с некорректным значением:

- `errors= 'raise' — дефолтное поведение: при встрече с некорректным значением выдаётся ошибка, операция перевода в числа прерывается;
- `errors= 'coerce' — некорректные значения *принудительно* заменяются на NaN;
- `errors= 'ignore' — некорректные значения *игнорируются*, но остаются.

```python
transactions['amount'] = pd.to_numeric(transactions['amount'], errors='coerce')
```

Методом `to_numeric()` мы не только превратим строки в числовой тип там, где это возможно, но и выясним, на каких значениях метод не работает.

Особенность метода `to_numeric()` в том, что при переводе все числа будут иметь тип данных *float*. Это подходит далеко не всем значениям. Но есть и хорошие новости: в нужный тип значения переводят методом `astype()`. Например, аргумент `('int')` метода `astype()` означает, что значение нужно перевести в целое число:

Методом **to_datetime()** превратим содержимое этого столбца в понятные для Python даты.

Для этого строку форматируют, обращаясь к специальной системе обозначений, где:

- %d — день месяца (от 01 до 31)
- %m — номер месяца (от 01 до 12)
- %Y — четырёхзначный номер года (например, 2019)
- Z — стандартный разделитель даты и времени
- %H — номер часа в 24-часовом формате
- %I — номер часа в 12-часовом формате
- %M — минуты (от 00 до 59)
- %S — секунды (от 00 до 59)

Преобразуем формат даты 01.04.2019Z11:03:00 из первой строчки датафрейма:

1. Сначала номер дня месяца. В соответствии с таблицей форматов заменяем его на **%d**: **%d**.04.2019Z11:03:00
2. Далее точка (ее оставляем без изменений), потом номер месяца: **%m**: **%d.%m**.2019Z11:03:00
3. После четырёхзначный номер года, заменяем 2019 на **%Y**: **%d.%m.%Y**Z11:03:00
4. Букву **Z** оставляем без изменений: **%d.%m.%YZ**11:03:00
5. Номер часа в 24-часовом формате заменим на **%H: %d.%m.%YZ%H:**03:00
6. Количество минут обозначим **%M: %d.%m.%YZ%H:%M:**00
7. Завершим форматирование обозначением секунд **%S: %d.%m.%YZ%H:%M:%S**

Вот такое выражение **'%d.%m.%YZ%H:%M:%S'** передают в аргумент *format* метода `to_datetime()` при переводе строковых значений столбца 'date' в формат *datetime:*

PYTHON
```
arrivals['date_datetime'] = pd.to_datetime(
    arrivals['date'], format='%d.%m.%YZ%H:%M:%S'
)
print(arrivals.head())
```

Часто приходится исследовать статистику по месяцам: например, узнать, на сколько минут сотрудник опаздывал в среднем. Чтобы осуществить такой расчёт, нужно поместить время в класс *DatetimeIndex* и применить к нему атрибут *month*:

PYTHON
```
arrivals['month'] = pd.DatetimeIndex(arrivals['date']).month
```

Объединим несколько таблиц в одну методом **merge()**).

`merge()` применяют к таблице, к которой присоединяют другую. У метода следующие аргументы:

- *right* — имя *DataFrame* или *Series*, присоединяемого к исходной таблице
- *on* — название общего списка в двух соединяемых таблицах: по нему происходит слияние
- *how* — чьи *id* включать в итоговую таблицу. Аргумент *how* может принять значение *left:* тогда в итоговую таблицу будут включены id из левой таблицы. Аргумент *right* включает *id* из правой таблицы.

Объединим таблицы *data* и *subcategory_dict* со следующими условиями:

- `data` — таблица, к которой будем присоединять другую таблицу
- `subcategory_dict` — таблица, которую присоединяем к *data*
- `'subcategory_id'` — общий столбец в двух таблицах, по нему будем объединять
- `how='left'` — *id* таблицы *data* включены в итоговую таблицу *data_subcategory*

PYTHON

```
data_subcategory = data.merge(subcategory_dict, on='subcategory_id', how='left')
print(data_subcategory.head(10))
```

В Pandas для подготовки сводных таблиц вызывают метод **pivot_table()**.

Аргументы метода:

- *index* — столбец или столбцы, по которым группируют данные (название товара)
- *columns* — столбец, по значениям которого происходит группировка (даты)
- *values* — значения, по которым мы хотим увидеть сводную таблицу (количество проданного товара)
- *aggfunc* — функция, применяемая к значениям (сумма товаров)

```
data_pivot = data_final.pivot_table(index=['category_name', 'subcategory_name'], columns='source', values='visits', aggfunc='sum')
print(data_pivot.head(10))
```

PYTHC

```
source                                direct  organic
category_name subcategory_name
Авто          Автоакустика              5915    15433
              Автомагнитолы             7783    18690
              Автомобильные инверторы    145      150
```

Как видно, основная категория включает в себя подкатегорию и это представлено в структуре датафрейма: категория отображена иерархически главной над подкатегорией. Такие датафреймы содержат в себе мультииндекс. Часто при работе с такими датафреймами мультииндекс убирают, чтобы категория была отображена на каждой строчке датафрейма:

PYTHON

```
data_pivot_with_reset_index = data_pivot.reset_index()
print(data_pivot_with_reset_index.head(10))
```

Для группировки данных также подходит изученная вами ранее комбинация методов `groupby()` и `agg()`, но с ними таблица будет выглядеть иначе.

Метод `groupby()` принимает один аргумент — столбец (или список столбцов), по которым группируют данные. В метод `agg()` передают словарь. Его ключ — это названия столбцов, а значение — функции, которые будут к этим столбцам применены (например, `sum` или `count`). Такие функции называются агрегирующие.

Решим ту же задачу по SEO-оптимизации методами `groupby()` и `agg()`:

PYTHON

```
data_grouped = data_final.groupby(['category_name','subcategory_name','source']).agg({'visits':'sum'})
print(data_grouped.head(10))
```

Изменение типов данных

Pandas (Dataset)

Чтение таблицы из файла Excel

In
```
df = pd.read_excel('file.xlsx', sheet_name='Лист 1')

# первый аргумент – строка с именем файла
# второй аргумент (sheet_name) – имя листа
```

In
```
df = pd.read_excel('file.xlsx')
# если второй аргумент пропущен, то будет прочитан первый по счёту лист
```

Слияние двух датасетов

In
```
data.merge(d, on, how)
# d   – датасет, с которым сливают
# on  – колонка, по значениям которой сливают
# how – тип слияния:

data.merge(data2, on='merge_column', how='left')
## left – обязательно присутствуют все значения из таблицы data,
##        вместо значений из data2 могут быть NaN

data.merge(data2, on='merge_column', how='right')
## right – обязательно присутствуют все значения из таблицы data2,
##         вместо значений из data могут быть NaN
```

Формирование сводной таблицы

In
```
data_pivot = data.pivot_table(index = ['column1', 'column2'], columns = 'source',
values = 'column_pivot', aggfunc = 'function')

# index   – столбец или столбцы, по которым происходит группировка данных
# columns – столбец по значениям которого будет происходить группировка
# values  – значения, по которым мы хотим увидеть сводную таблицу
# aggfunc – функция, которая будет применяться к значениям
```

Pandas (Column)

Перевод значений столбца из строкового типа str в вещественный тип float

In
```
pd.to_numeric(data['column'])
# первый аргумент – колонка из датафрейма
# второй аргумент (errors) – метод обработки ошибок


pd.to_numeric(data['column'], errors='raise' )
# если errors='raise' (значение по умолчанию), то при встрече с некорректным
# значением выдается ошибка, операция перевода в числа прерывается;

pd.to_numeric(data['column'], errors='coerce')
# если errors='coerce', то некорректные значения принудительно заменяются на NaN;

pd.to_numeric(data['column'], errors='ignore')
# если errors='ignore', то некорректные значения игнорируются, но остаются.
```

In
```
data['column'] = pd.to_numeric(data['column'])
# Возвращает новую колонку, не заменяя предыдущую.
# Для замены нужно выполнить присваивание.
```

Перевод значений столбца в другой тип данных

In
```
data['column'].astype('type') # например int для целых чисел, а str для строк
```

In
```
data['column'] = data['column'].astype('type')
# Возвращает новую колонку, не заменяя предыдущую.
# Для замены нужно выполнить присваивание.
```

Перевод из строки в дату и время

In
```
pd.to_datetime(data['date_time_column'], format='%d.%m.%Y %H:%M:%S')
# обязательный второй агрумент – строка формата

'''
Формат строится с использованием следующих обозначений для частей даты и времени:
• %d – день месяца (от 01 до 31)
• %m – номер месяца (от 01 до 12)
• %Y – год с указанием столетия (например, 2019)
• %H – номер часа в 24-часовом формате
• %I – номер часа в 12-часовом формате
• %M – минуты (от 00 до 59)
• %S – секунды (от 00 до 59)
'''

# например, если даты выглядят так:
20.03.2017 11:00:50    # то формат:
'%d.%m.%Y %H:%M:%S'
```

In
```
data['date_time_column'] = pd.to_datetime(data['date_time_column'],
format='%d.%m.%Y %H:%M:%S')
# Возвращает новую колонку, не заменяя предыдущую.
# Для замены нужно выполнить присваивание.
```

Получение отдельных частей даты и времени

In
```
# Получение из столбца с датой и временем...
pd.DatetimeIndex(data['time']).year   # года
pd.DatetimeIndex(data['time']).month  # месяца
pd.DatetimeIndex(data['time']).day    # дня
pd.DatetimeIndex(data['time']).hour   # часа
pd.DatetimeIndex(data['time']).minute # минуты
pd.DatetimeIndex(data['time']).second # секунды
```

Python

Обработка исключений

In
```
try:
    ... # код, где может быть ошибка

except:
    ... # действия если возникла ошибка
```

Способ 1. Во вводном курсе вы уже изучали метод `duplicated()`. В сочетании с методом `sum()` он возвращает количество дубликатов. Напомним, что если выполнить метод `duplicated()` без подсчёта суммы, на экране будут отображены все строки. Там, где есть дубликаты, будет значение `True`, где дубликата нет — `False`.

Способ 2. Вызвать метод `value_counts()`, который анализирует столбец, выбирает каждое уникальное значение и подсчитывает частоту его встречаемости в списке. Применяют метод к объекту *Series*. Результат его работы – список пар «значение-частота», отсортированные по убыванию. Все дубликаты, которые встречаются чаще других, оказываются в начале списка.

Найдём `'мух'` в цикле. Превратим набор слов в список — применим метод `split()`. Метод разбивает строки на части специальным разделителем в `аргументе` и собирает их в список:

Типа если у тебя строка из нескольких слов, то он их на одельные списки разбивает по слову,чтобы можно было анализировать каждое слово из строки

Python

Приведение строки к нижнему регистру

```
In  string.lower()
```

Подсчёт различных значений в списке

```
In  from collections import Counter

    Counter(lst)

    # используется коллекция Counter,
    # реализующая словарь для подсчёта
    # колличества неизменяемых объектов
```

Pandas

Приведение строк в колонке к нижнему регистру

```
In  data['column'].str.lower()
```

NLTK

Получение стеммера для русского языка

```
In  russian_stemmer =
                SnowballStemmer('russian')
```

Получение стема от слова на русском

```
In  russian_stemmer.stem(word)
```

PyMystem

Получение стеммера/лемматизатора для слов на русском

```
In  from pymystem3 import Mystem

    m = Mystem()
```

Лемматизация строки с русским текстом

```
In  m.lemmatize(text)
```

Категоризация данных

Применение метода к значениям в столбце или строке

In
```
data['column'].apply(method)

# чтобы применить метод к строкам,
# укажите параметр axis=1
data.apply(method, axis=1)
```

Функция для строки

In
```
def function(row):
    info1 = row['column1']
    info2 = row['column2']
    ...

# далее идёт обработка
# в соответствии со значениями
```

Словарь

Приёмы упорядочивания данных

Выделение словаря категорий
В словаре каждому названию категории даётся в соответствие некоторое число, которое потом используется в основном датасете

Категоризация
Объединение данных в категории. Она может проводиться по данным из:

- *одного столбца*
  тогда метод категоризации принимает только одно значение - значение соответствующе столбца
- *нескольких столбцов*
  тогда метод категоризации принимает строку из датафрейма целиком

Первые графики и выводы

Чтение данных из файла с использованием разделителей

In
```
data = read_csv('file.csv',  sep=';' , decimal=',')

# sep – разделитель столбцов
# decimal – разделитель десятичных знаков
```

Импорт библиотеки matpotlib

In
```
import matplotlib.pyplot as plt
```

Отображение диаграммы ящик с усами

In
```
data.boxplot(column='column')
plt.show()
```

Числовое описание данных для колонки

In
```
data['column'].describe()
```

Изменение осей графика

In
```
plt.xlim(x_min, x_max)
# мин и макс для оси X
```

In
```
plt.ylim(y_min, y_max)
# мин и макс для оси Y
```

Отображение гистограммы с n_bins, min_value, max_value

In
```
data['column'].hist(bins=n_bins, range=(min_value, max_value))
plt.show()

# отображение гистограммы с числом корзин n_bins и отображаемыми
# минимальным и максимальным значением min_value и max_value
```

Изучение срезов данных

Проверка наличия элементов списка lst в столбце

```
In  data['column'].isin(lst)
```

Работа с датой и временем

```
In  # Получить ...
    data['datetime'].dt.date    #дату
    data['datetime'].dt.year    #год
    data['datetime'].dt.weekday #день нед.
```

Быстрое получение срезов данных

```
In  data.query('column != "value"')
    data.query('column < column.mean()')
```

```
In  variable = 2
    data.query('column > @variable')
```

Поставить значение года на первое место

```
In  pd.to_datetime(df['datetime'],
                   yearfirst = True)
```

Сдвиг даты и времени

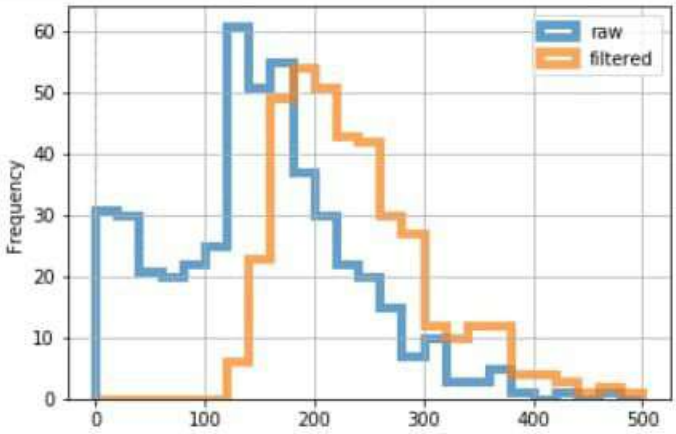
```
In  data['shifted_dt'] = data['datetime'] + pd.Timedelta(hours=10) # добавить 10 часов
```

Округление времени

```
In  data['datetime'] = data['datetime'].dt.round('1H')  # округлить до 1 часа
    data['datetime'] = data['datetime'].dt.round('1D')  # округлить до 1 дня
    data['datetime'] = data['datetime'].dt.round('5T')  # округлить до 5 минут
    data['datetime'] = data['datetime'].dt.round('10S') # округлить до 10 секунд
    data['datetime'] = data['datetime'].dt.floor('1H')  # округлять в меньшую сторону
    data['datetime'] = data['datetime'].dt.ceil('1H')   # округлять в большую сторону
```

Построение графиков по датафрейму

```
In  data.plot(x='column1',    # столбец значений для горизонтальной оси
              y='column2',    # столбец значений для вертикальной оси
              style='o-',     # стиль заполнения:'o'(точечный) или  'o-'(точечно-линейный)
              xlim=(0, 30),   # границы по оси X
              ylim=(30, 0),   # границы по оси Y
              figsize=(4, 5), # размеры картинки: (x_size, y_size)
              grid=True)      # отображать сетку или нет
```



Выглядят непривычно. Всё из-за ряда новых параметров:

- **histtype** (от англ. *the type of histogram*, «тип гистограммы»). В параметре указывают тип гистограммы, по умолчанию — это столбчатая (закрашенная). Значение *'step'* (англ. «шаг») чертит только линию.
- **linewidth** (от англ. *width of line*, «толщина линии»). Задаёт толщину линии графика в пикселях.
- **alpha** (от термина «альфа-канал»). Назначает густоту закраски линии. 1 — это 100% закраска; 0 — прозрачная линия. С параметром 0.7 линии чуть прозрачны, так виднее их пересечения.
- **label** (англ. «ярлык, этикетка»). Название линии.
- **ax** (от англ. *axis* — «ось»). Метод *plot()* возвращает оси, на которых был построен график. Чтобы обе гистограммы расположились на одном графике, сохраним оси первого графика в переменной *ax*, а затем передадим её значение параметру *ax* второго *plot()*. Так, сохранив оси одной гистограммы и построив вторую на осях первой, мы объединили два графика.
- **legend** (англ. «легенда»). Выводит легенду — список условных обозначений на графике. На нашем графике вы можете найти её в верхнем правом углу.

Работа с несколькими источниками данных

Использование списков, словарей, серий, датафреймов для получения срезов

Проверяет, есть ли значение в списке

```
In  our_list = [1, 2, 3, 4]
    data.query('column in @our_list')
```

Проверяет, есть ли значение среди ключей словаря

```
In  our_dict = {0: 1, 4: 82, 71: 1414}
    data.query('column in @our_dict')
```

Проверяет, есть ли значение среди индексов серии

```
In  our_series = pd.Series([81, 12, 64])
    data.query('column in @our_series.index')
```

Проверяет, есть ли значение среди значений серии

```
In  our_series = pd.Series([81, 12, 64])
    data.query('column in @our_series')
```

Проверяет, есть ли значение среди индексов датафрейма

```
In  our_dataframe = pd.DataFrame({
        'column1': [0, 1, 10, 11, 12],
        'column2': [5, 4, 3, 2, 1],
        })

    data.query(
        'column in @our_dataframe.index'
        )
```

Проверяет, есть ли значение среди значений колонки датафрейма

```
In  our_dataframe = pd.DataFrame({
        'column1': [0, 1, 10, 11, 12],
        'column2': [5, 4, 3, 2, 1],
        })

    data.query(
        'column in @our_dataframe.column2'
        )
```

Построение гистограммы с дополнительными параметрами

```
In  data.plot(kind='hist',
              y='column',
              histtype='step',          # тип диаграммы
              range=(y_min, y_max),
              bins=n_bins,
              linewidth=our_linewidth, # толщина линий графика в пикселях
              alpha=our_alpha,         # густота закраски, число от 0 до 1
              label='label',           # название линии
              ax=our_ax,               # оси, по которым строится график
              grid=True,
              legend=True)             # выводить ли легенду к графику
```

Возврат крайних значений группы

```
In  df.pivot_table(index='index_column', values='values_column', aggfunc='first')#первый
    df.pivot_table(index='index_column', values='values_column', aggfunc='last') #последний
```

Взаимосвязь данных

Построение точечной диаграммы (диаграммы рассеяния)

In
```
data.plot(x='column_x', y='column_y', kind='scatter')
```

Построение попарных точечных диаграмм для столбцов датафрейма

In
```
pd.plotting.scatter_matrix(data)
```

Построение ячеечной диаграммы

In
```
data.plot(x='column_x', y='column_y', kind='hexbin', gridsize=20, sharex=False)

# gridsize – число ячеек по горизонтальной оси
```

Вычисление коэффициента корреляции Пирсона

In
```
print(data['column_1'].corr(data['column_2']))
# или
print(data['column_2'].corr(data['column_1']))

# Коэффициент не зависит от порядка рассчёта
```

Коэффициент корреляции Пирсона между всеми парами столбцов

In
```
data.corr()
```

Валидация результатов

Построение столбчатой диаграммы

```
In  data.plot(y='column', kind='bar')
```

Построение круговой диаграммы

```
In  data.plot(y='column', kind='pie')
```

Выборочное изменение значения

```
In  data['column'].where(s > control_value, default_value)

    # если не выполняется условие - первый параметр,
    # то значение заменяется на второй параметр
```

Срезы по значениям столбца

```
In  # column_value - значение столбца,
    # column_slice - срез данных, в котором значение столбца - column_value

    for column_value, column_slice in data.groupby('column'):
        # do something
```

Теория вероятностей

Объявление таблицы в numpy array

```
In  table = np.array([[2,3,4,5,6,7],
                      [3,4,5,6,7,8],
                       ...
                      [7,8,9,10,11,12]])
```

Получение списка ключей и значений словаря

```
In  dictionary.keys()   # список ключей
    dictionary.values() # список значений
```

Вычисление факториала числа

```
In  from math import factorial

    x = factorial(5)
```

Задание нормального распределения по математическому ожиданию и стандартному отклонению

```
In  from scipy import stats as st

    distr = st.norm(1000, 100)
```

Вычисление вероятности

```
In  result = distr.cdf(x)                    # вероятность получить значение не больше x

    result = distr.cdf(x2) - distr.cdf(x1) # вероятность получить значение между x1 и x2

    result = distr.ppf(p1)                   # значение по вероятности
```

Проверка гипотез

Проверка гипотезы о равенстве среднего генеральной совокупности некоторому значению

In
```
from scipy import stats as st

results = st.ttest_1samp(array, interested_value)
# array - выборка
# interested_value - предполагаемое среднее, на равенство которому мы делаем тест

print('p-значение: ', results.pvalue)
```

Проверка гипотезы о равенстве среднего двух генеральных совокупностей по взятым из них выборкам

In
```
from scipy import stats as st

sample_1 = [...] # sample_1 - выборка из первой генеральной совокупности
sample_2 = [...] # sample_2 - выборка из второй генеральной совокупности

results = st.ttest_ind(sample_1, sample_2, equal_var = True)
# equal_var - считать ли равными дисперсии выборок, по умолчанию имеет значение True

print('p-значение: ', results.pvalue)
```

Проверка гипотезы о равенстве средних двух генеральных совокупностей для зависимых (парных) выборок

In
```
from scipy import stats as st

results = st.ttest_rel(before, after)
# pair_1 - первая парная выборка
# pair_2 - вторая парная выборка

print('p-значение: ', results.pvalue)
```

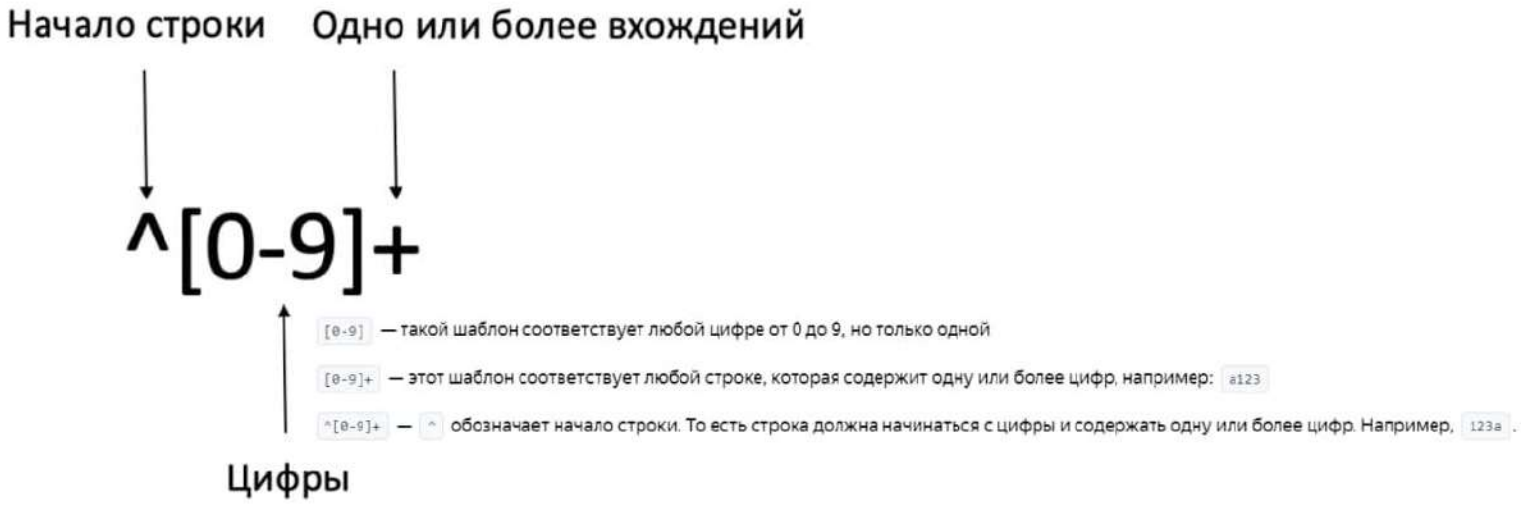
Основные шаблоны регулярных выражений

В таблице приведены простейшие шаблоны регулярных выражений. Сложные регулярные выражения состоят из их комбинаций.

РЕГУЛЯРНОЕ ВЫРАЖЕНИЕ	ОПИСАНИЕ	ПРИМЕР	ПОЯСНЕНИЕ
`[]`	Один из символов в скобках	`[a-]`	a или -
`[^_]`	Отрицание	`[^a]`	любой символ кроме «a»
`-`	Интервал	`[0-9]`	интервал: любая цифра от 0 до 9
`.`	Один любой символ, кроме перевода строки	`a.`	as, a1, a_
`\d` (аналог `[0-9]`)	Любая цифра	`a\d | a[0-9]`	a1, a2, a3
`\w`	Любая буква, цифра или `_`	`a\w`	a_, a1, ab
`[A-z]`	Любая латинская буква	`a[A-z]`	ab
`[А-я]`	Любая буква кириллицы	`a[А-я]`	ая
`?`	Ноль или одно вхождение	`a?`	a или ничего
`+`	Одно и более вхождений	`a+`	a или aa, или aaa
`*`	Ноль и более вхождений	`a*`	ничего или a, или aa
`^`	Начало строки	`^a`	a1234, abcd
`$`	Конец строки	`a$`	1a, ba

Как создать шаблон регулярного выражения

Напишем шаблон сложного регулярного выражения, которое ищет строку, начинающуюся с числа.



`[0-9]` — такой шаблон соответствует любой цифре от 0 до 9, но только одной

`[0-9]+` — этот шаблон соответствует любой строке, которая содержит одну или более цифр, например: `a123`

`^[0-9]+` — `^` обозначает начало строки. То есть строка должна начинаться с цифры и содержать одну или более цифр. Например, `123a`.

Регулярные выражения в Python

Самые распространённые задачи аналитика:

- найти подстроку в строке
- разбить строки на подстроки на основании шаблона
- заменить части строки на другую строку

Вот какие методы библиотеки *re* для этого понадобятся:

search(pattern, string) (англ. «поиск») ищет шаблон *pattern* в строке *string*. Хотя *search()* ищет шаблон во всей строке, возвращает он только первую найденную подстроку:

```
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.search('\w+', string))

<re.Match object; span=(1, 8), match='Генерал'>
```

Метод *search()* возвращает объект типа **match** (англ. «соответствовать»). Параметр *span* (англ. «диапазон») указывает диапазон индексов, подходящих под шаблон. В нашем случае открывающая кавычка `«` не отвечает правилу, которое игнорирует знаки препинания. Вот потому индексы идут с 1 по 8: от буквы «Г» до буквы «л». В параметре *match* указано само значение подстроки.

Шаблону `'\w+'` соответствует любая подстрока, содержащая одну и более букв, цифр или символ нижнего подчёркивания _ . Метод *search()* нашёл, что этому шаблону соответствует первое слово в строке. Так как под правило `'\w+'` не подходит пробел, метод вернул всё, что идёт до первого пробела.

Если нам не нужны дополнительные сведения о диапазоне, выведем только найденную подстроку методом *group()*:

```
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.search('\w+', string).group())

'Генерал'
```

Как добыть информацию между определенными словами? Достанем данные, содержащие русские буквы или пробел между символами `"«"` и `"»"`

```
import re

string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.search('«[А-я ]+»', string).group())

'«Генерал Слокам»'
```

Обратите внимание, что мы учли все символы, содержащиеся в нужной подстроке, в том числе — пробел.

split(pattern, string) (англ. «расщеплять, разбивать») разделяет строку *string* по границе шаблона *pattern*.

```python
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.split('\d+', string))

['«Генерал Слокам» ', ' июня ', ' года Ист-Ривер Человеческий фактор']
```

Строка разделена на три части. Границы деления строки проходят там, где метод встретил указанный в аргументе шаблон. В нашем случае шаблону регулярного выражения `'\d+'` соответствует одна и более цифр. Поэтому строка поделилась натрое в тех местах, где *split()* обнаружил подстроки из цифр — `15` и `1904`.

Количеством делений строки можно управлять. За это отвечает параметр **maxsplit** метода *split()* (по умолчанию равен 0).

```python
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.split('\d+', string, maxsplit = 1))

['«Генерал Слокам» ', ' июня 1904 года Ист-Ривер Человеческий фактор']
```

Строка разделилась один раз по первому найденному шаблону.

sub(pattern, repl, string) (от англ. *substring*, «подстрока») ищет подстроку *pattern* в строке *string* и заменяет его на подстроку **repl** (от англ. *replace* — «заменить»).

```python
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.sub('\d+', '', string)) #ищем числа
# и заменяем на пустоту

'«Генерал Слокам»  июня  года Ист-Ривер Человеческий фактор'
```

Все подстроки с числами заменены на пустоту.

Метод **findall(pattern, string)** возвращает список *всех подстрок* в *string*, удовлетворяющих шаблону *pattern*. А не только первую подходящую подстроку, как *search()*. Найдём все слова, заканчивающиеся на `"мя"`:

```python
import re
mya = "Вовремя подняли знамя и бросились в пламя"
print(re.findall('[А-я]+мя', mya))

['Вовремя', 'знамя', 'пламя']
```

split(pattern, string) (англ. «расщеплять, разбивать») разделяет строку *string* по границе шаблона *pattern*.

```python
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.split('\d+', string))

['«Генерал Слокам» ', ' июня ', ' года Ист-Ривер Человеческий фактор']
```

Строка разделена на три части. Границы деления строки проходят там, где метод встретил указанный в аргументе шаблон. В нашем случае шаблону регулярного выражения `'\d+'` соответствует одна и более цифр. Поэтому строка поделилась натрое в тех местах, где *split()* обнаружил подстроки из цифр — `15` и `1904`.

Количеством делений строки можно управлять. За это отвечает параметр **maxsplit** метода *split()* (по умолчанию равен 0).

```python
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.split('\d+', string, maxsplit = 1))

['«Генерал Слокам» ', ' июня 1904 года Ист-Ривер Человеческий фактор']
```

Строка разделилась один раз по первому найденному шаблону.

sub(pattern, repl, string) (от англ. *substring*, «подстрока») ищет подстроку *pattern* в строке *string* и заменяет его на подстроку **repl** (от англ. *replace* — «заменить»).

```python
import re
string = "«Генерал Слокам» 15 июня 1904 года Ист-Ривер Человеческий фактор"
print(re.sub('\d+', '', string)) #ищем числа
# и заменяем на пустоту

'«Генерал Слокам»  июня  года Ист-Ривер Человеческий фактор'
```

Все подстроки с числами заменены на пустоту.

Метод **findall(pattern, string)** возвращает список *всех подстрок* в *string*, удовлетворяющих шаблону *pattern*. А не только первую подходящую подстроку, как *search()*. Найдём все слова, заканчивающиеся на `"мя"`:

```python
import re
mya = "Вовремя подняли знамя и бросились в пламя"
print(re.findall('[А-я]+мя', mya))

['Вовремя', 'знамя', 'пламя']
```

Импортируем библиотеку и создадим объект *BeautifulSoup*:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(req.text, 'lxml')
```

Первый аргумент — это данные, из которых будет собираться древовидная структура. Второй аргумент — синтаксический анализатор, или парсер. Он отвечает за то, как именно из кода веб-страницы получается «дерево». Парсеров много, они создают разные структуры из одного и того же HTML-документа. За высокую скорость работы мы выбрали анализатор **lxml**. Есть и другие, например, *html.parser*, *xml* или *html5lib*.

Поиск по дереву

Мы превратили код в дерево. Время извлекать данные.

Первый метод поиска называется **find()** (англ. «найти»). В HTML-документе он находит первый элемент, имя которого ему передали в качестве аргумента, и возвращает его весь, с тегами и контентом. Найдём, к примеру, первый заголовок второго уровня:

```
heading_2=soup.find('h2')
print(heading_2)
```

```
<h2>Крупнейшие морские катастрофы XX века</h2>
```

Чтобы посмотреть контент без тега, вызывают метод **text**. Результат возвращается в виде строки:

```
print(heading_2.text)
```

Существует и другой метод поиска — **find_all** (англ. «найти всё»). В отличие от предыдущего метода, *find_all()* находит *все* вхождения определённого элемента в HTML-документе и возвращает список:

```
    paragraph=soup.find_all('p') # напомним, p - это параграф, текст между тегами <p> и </p>
    print(paragraph)
```

```
[<p>Благодаря массовой культуре морские катастрофы чаще всего ассоциируются с «Титаником». Однако в начале XX века столкновение пар
```

Методом *text* вычленим только контент из параграфов:

```
for paragraph in soup.find_all('p'):
    print(paragraph.text)
```

```
Благодаря массовой культуре морские катастрофы чаще всего ассоциируются с «Титаником». Однако в начале XX века
    столкновение парохода «Титаник» с айсбергом было не единственным кораблекрушением.
Теперь посмотрим на тех, кто был рядом с «Титаником»:
```

У методов *find()* и *find_all()* есть дополнительный фильтр поиска элементов страницы — параметр **attrs** (от англ. *attributes*, **«атрибуты»).

Это он охотится на идентификаторы и классы. Их имена уточняют в панели разработчика:

Параметру *attrs* передают словарь с именами и значениями атрибутов. Вот разыскивается элемент с идентификатором `'ten_years_first'`:

```python
print(soup.find('table',attrs={'id': 'ten_years_first'}))
```

Напомним, что открывающий тег `<table>` указывает начало таблицы, а закрывающий `</table>` — её конец. Внутри теги строк — `<tr>`; ячеек — `<td>` и заголовков столбцов — `<th>`.

Создадим пустой список *heading_table*, где сохраним названия столбцов. В цикле методом *find_all()* найдём все элементы `th`. Методом *text* добудем их контент и добавим его в список *heading_table*:

```python
heading_table = [] # Список, в котором будут храниться названия столбцов
for row in table.find_all('th'): # Названия столбцов прячутся в элементах th,
# поэтому будем искать все элементы th внутри table и пробегать по ним в цикле
        heading_table.append(row.text) # Добавляем контент из тега th в список heading_table методом append()
print(heading_table)
```

```python
['Название корабля', 'Дата катастрофы', 'Место катастрофы', 'Причина катастрофы']
```

Создадим пустой список *content*, сохраним там данные таблицы. В цикле обратимся к каждой строке по имени элемента `tr`.

Отдельно отметим, что самая первая строка таблицы с заголовками в тегах `<th> </th>`, нас не интересует. Потому перед тем, как в цикле добавлять значения в пустой список, укажем, что это не касается строки с заголовками: `if not row.find_all('th')`.

Применим метод *find_all()* к элементам *td*. Методом *text* очистим полученные ячейки от тегов и сложим в список *content*.

```python
content=[] # Список, в котором будут храниться данные из таблицы
for row in table.find_all('tr'):
# Каждая строка обрамляется тегом tr, необходимо пробежаться в цикле по всем строкам
    if not row.find_all('th'):
# Эта проверка необходима, чтобы пропустить первую строку таблицы с заголовками
            content.append([element.text for element in row.find_all('td')])
            # В каждой строке контент ячейки обрамляется тегами <td> </td>
            # Необходимо пробежаться в цикле по всем элементам td, вычленить контент из ячеек и добавить его в список
            # Затем добавить каждый из списков в список content
print(content)
```

```python
[['«Генерал Слокам»', '15 июня 1904 года', 'Ист-Ривер', 'Человеческий фактор'], ['«Каморта»', '6 мая 1902 года', 'Бенгальский залив
```

В результате мы получили 2 списка. В списке *heading_table* сохранили названия столбцов. В *content* — наполнение таблицы в виде двумерного массива.

Извлечение данных из веб-ресурсов

Получение информации с веб-страницы с адресом url

In
```
import requests

req = requests.get(URL)
print(req.text)         # вывод содержимого веб-страницы
print(req.status_code)  # вывод кода возврата
```

Поиск первого вхождения подстроки, соответствующей регулярному выражению pattern, в строку string

In
```
import re
print(re.search(pattern, string).group())
```

Разделение строки string на подстроки, границы которых определяются регулярным выражением pattern

In
```
import re
print(re.split(pattern, string, maxsplit=num_split))

# maxsplit - максимальное число делений, по умолчанию maxsplit = 0
```

Поиск подстроки по шаблону pattern в строке string и замена её на подстроку repl

In
```
import re
print(re.sub(pattern, repl, string))
```

Поиск всех подстрок по шаблону pattern в строке string

In
```
import re
print(re.findall(pattern, string))
```

Формирование древовидной структуры веб-страницы

In
```
from bs4 import BeautifulSoup
soup = BeautifulSoup(req.text, parser)
```

Поиск первого тега tag

In
```
# Возвращает строку с тегом, атрибутами и содержимым
# attrs - словарь атрибутов тега

tag_content = soup.find(tag, attrs={"attr_name": "attr_value"})
print(tag_content.text) # контент без тега
```

Выполнение операций со всеми тегами tag

In
```
# attrs - словарь атрибутов тега

for tag_content in soup.find_all(tag, attrs={"attr_name": "attr_value"}):
    # do something
```