**1. План урока**

## На уроке

1. Познакомитесь с **форматом файлов CSV**. Это текстовый формат хранения табличных данных. Научитесь читать и записывать CSV-файлы с помощью **библиотеки**

csv

.

1. Изучите **библиотеку**

pandas

 — инструмент для анализа данных в Python.

pandas

 предоставляет удобные и простые в использовании инструменты для обработки и анализа данных. Эти инструменты помогают легко читать и записывать данные из различных источников, работать с пропущенными значениями и выполнять множество других операций.

1. Научитесь использовать

DataFrame

 и

Series

. Это основные объекты

pandas

 для работы с данными. Узнаете, как создавать, индексировать и изменять

DataFrame

, а также как осуществлять основные операции с ним, такие как фильтрация, группировка, агрегация и сортировка данных.

## Про домашку

Продолжите работать с проектом. Добавите в свой код несколько типов входных данных о транзакциях из CSV- или XLSX-файлов.

# 2. Библиотека csv

**Библиотека**

csv

**в Python** — предоставляет удобные средства для работы с файлами в формате CSV.

**Формат CSV (Comma-Separated Values — значения, разделенные запятыми)** — это текстовый формат, широко используемый для хранения табличных данных, где каждая строка файла соответствует строке таблицы, а ячейки в строке разделены специальным символом-разделителем, чаще всего запятой.

Пример содержимого CSV-файла:

Name,Age,Gender

Alice,25,Female

Bob,30,Male

Charlie,35,Male

## Чтение из CSV-файла

Для начала работы с CSV в Python необходимо импортировать библиотеку

csv

. Затем можно использовать функцию

csv.reader

.

**Функция**

csv.reader

 — используется для чтения содержимого файла. В качестве аргумента функции чтения передается файл.

Пример кода для чтения CSV-файла:

import csv

with open('file.csv') as file:

reader = csv.reader(file)

for row in reader:

print(row)

Этот код последовательно выводит каждую строку из файла

file.csv

.

Если в файле в качестве разделителя используется не запятая, а другой символ, например точка с запятой, можно указать это при вызове

csv.reader

 в параметре

delimiter

:

with open('file.csv') as file:

reader = csv.reader(file, delimiter=';')

for row in reader:

print(row)

## Запись данных в CSV-файл

**Функция**

csv.writer

 — используется, чтобы записать данные в CSV-файл.

Пример создания файла и записи данных в него:

import csv

rows = [['Name', 'Age', 'Gender'],

['Alice', '25', 'Female'],

['Bob', '30', 'Male'],

['Charlie', '35', 'Male']]

with open('file.csv', 'w', newline='') as file:

writer = csv.writer(file)

writer.writerows(rows)

Для изменения разделителя, как и при чтении файла CSV, используйте аргумент

delimiter

 функции

csv.writer

.

## Работа с CSV как со словарями

**Функция**

DictReader

 — позволяет читать CSV-файл и создавать словари из строк файла. Ключами в словаре будут названия столбцов, а значениями — значения в соответствующих ячейках.

Пример чтения с

DictReader

:

import csv

with open('file.csv') as file:

reader = csv.DictReader(file)

for row in reader:

print(row['Name'], row['Age'], row['Gender'])

**Функция**

DictWriter

 — позволяет записывать словари в CSV-файл.

Для этого нужно указать названия столбцов (ключи в словаре) в качестве аргумента

fieldnames

 функции

DictWriter

, а затем вызывать метод

writeheader()

 для записи заголовка и метод

writerow()

 для записи строк.

Пример записи с

DictWriter

:

import csv

rows = [{'Name': 'Alice', 'Age': '25', 'Gender': 'Female'},

{'Name': 'Bob', 'Age': '30', 'Gender': 'Male'},

{'Name': 'Charlie', 'Age': '35', 'Gender': 'Male'}]

with open('file.csv', 'w', newline='') as file:

fieldnames = ['Name', 'Age', 'Gender']

writer = csv.DictWriter(file, fieldnames=fieldnames)

writer.writeheader()

for row in rows:

writer.writerow(row)

Этот пример показывает, как создать CSV-файл и записать в него данные, используя модуль

csv

 в Python. Давайте разберем код шаг за шагом:

1. **Импортируем модуль**

csv

:

import csv

1. **Создаем список словарей, каждый из которых представляет собой строку данных**:

rows = [{'Name': 'Alice', 'Age': '25', 'Gender': 'Female'},

{'Name': 'Bob', 'Age': '30', 'Gender': 'Male'},

{'Name': 'Charlie', 'Age': '35', 'Gender': 'Male'}]

Здесь

rows

 — это список, содержащий три словаря. Каждый словарь представляет одну строку данных с ключами

Name

,

Age

 и

Gender

.

1. **Открываем файл для записи**:

with open('file.csv', 'w', newline='') as file:

Открываем файл с именем

file.csv

 в режиме записи (

'w'

). Параметр

newline=''

 используется, чтобы избежать добавления лишних пустых строк между строками данных на ОС Windows.

1. **Определяем заголовки столбцов**:

fieldnames = ['Name', 'Age', 'Gender']

fieldnames

 содержит список заголовков столбцов, которые будут использоваться в CSV-файле.

1. **Создаем объект**

DictWriter

**и записываем заголовок столбцов**:

writer = csv.DictWriter(file, fieldnames=fieldnames)

writer.writeheader()

DictWriter

 создается с указанием файла и заголовков столбцов. Метод

writeheader()

 записывает строку заголовков в CSV-файл.

1. **Записываем строки данных в файл**:

for row in rows:

writer.writerow(row)

Цикл проходит по каждому словарю в

rows

 и записывает его в CSV-файл с помощью метода

writer.writerow(row)

.

В результате выполнения этого кода создается файл

file.csv

, содержащий следующие данные:

Name,Age,Gender

Alice,25,Female

Bob,30,Male

Charlie,35,Male

## Пример работы с CSV-файлом

Рассмотрим решение следующей задачи. Необходимо прочитать CSV-файл и вывести на экран информацию о студентах, у которых средний балл больше 4.5.

Файл с данными для задачи: [students.csv](https://drive.google.com/file/d/1s3s69_u85sLoWa3nc-IIFx8OWzka2Dq3/view?usp=sharing)

Решение:

import csv

with open('students.csv') as f:

    reader = csv.reader(f, delimiter=',')

    next(reader)  # Пропускаем заголовок таблицы

    for row in reader:

        name, age, avg\_grade = row

        if float(avg\_grade) > 4.5:

            print(f'{name} ({age} лет) - средний балл: {avg\_grade}')

Разберем, что происходит в этом коде:

1. **Импортируем модуль**

csv

:

import csv

1. **Открываем CSV-файл.**

with open('students.csv') as f

 — открывает файл

students.csv

 в режиме чтения.

1. **Создаем объект для чтения CSV.**

reader = csv.reader(f, delimiter=',')

 — создает объект для чтения CSV, который будет считывать файл с разделителем «запятая».

1. **Пропускаем заголовок таблицы.**

next(reader)

 — пропускает первую строку файла, которая обычно содержит заголовки столбцов.

1. **Читаем строки из файла.**

Цикл

for row in reader

 проходит по каждой строке в файле.

1. **Извлекаем данные из строки.**

name, age, avg\_grade = row

 — извлекает значения из текущей строки и присваивает их переменным

name

,

age

 и

avg\_grade

.

1. **Проверяем средний балл.**

Условие

if float(avg\_grade) > 4.5

 проверяет, превышает ли средний балл 4.5.

1. **Выводим информацию.**

Если условие выполняется, выводится строка с именем, возрастом и средним баллом студента.

print(f'{name} ({age} лет) - средний балл: {avg\_grade}')

# 3. Создание и чтение данных в pandas

pandas

 — это библиотека Python для обработки и анализа данных. Ее часто используют для работы с табличными данными, такими как данные из файлов Excel, CSV или SQL.

## Установка pandas

Для установки библиотеки

pandas

 с помощью poetry необходимо выполнить следующую команду:

poetry add pandas

## Импорт библиотеки

Чтобы использовать библиотеку

pandas

 в своем коде, ее нужно импортировать. Обычно

pandas

 импортируют с сокращенным именем

pd

:

import pandas as pd

## Основные объекты pandas

В

pandas

 существует два основных объекта —

DataFrame

 и

Series

. Рассмотрим каждый объект подробнее.

### DataFrame

DataFrame

 — это двумерная таблица, в которой каждая запись соответствует строке и столбцу. Она позволяет хранить и обрабатывать данные в табличной форме.

Пример создания

DataFrame

:

df = pd.DataFrame({'Yes': [50, 21], 'No': [131, 2]})

print(df)

Результат:

Yes

No

0

50

131

1

21

2

В этом примере значение в ячейке

0, "No"

 равно 131, а в ячейке

0, "Yes"

 равно 50.

Записи в

DataFrame

 могут быть различными, в том числе строками:

df = pd.DataFrame({

'Bob': ['Мне это понравилось.', 'Это было ужасно.'],

'Sue': ['Довольно хорошо.', 'Без вкуса.']

})

print(df)

Результат:

Bob

Sue

0

Мне это понравилось.

Довольно хорошо.

1

Это было ужасно.

Без вкуса.

По умолчанию каждой строке в таблице

DataFrame

 присваивается метка (идентификатор), значение которой возрастает от 0 (0, 1, 2, 3, ...). Если числовую метку неудобно использовать для взаимодействия с данными, значение метки можно поменять.

Список меток строк, используемых в

DataFrame

, называется **Index**. Мы можем назначить ему значения с помощью параметра

index

, который указывается при создании

DataFrame

.

Пример:

df = pd.DataFrame({

'Bob': ['Мне это понравилось.', 'Это было ужасно.'],

'Sue': ['Довольно хорошо.', 'Без вкуса.']

}, index=['Товар A', 'Товар B'])

print(df)

В качестве параметра мы указали «Товар А» и «Товар В» и получили следующий результат:

Bob

Sue

Товар A

Мне это понравилось.

Довольно хорошо.

Товар B

Это было ужасно.

Без вкуса.

### Series

Series

 — это одномерная последовательность значений данных. Напоминает столбец в таблице

DataFrame

.

Пример создания

Series

:

s = pd.Series([1, 2, 3, 4, 5])

print(s)

Результат:

0 1

1 2

2 3

3 4

4 5

dtype: int64

Последняя строка

dtype: int64

 указывает на тип данных, хранящихся в

Series

.

Мы можем задавать метки строк для

Series

 с помощью параметра

index

. Однако у

Series

 нет имени столбца, у него только одно общее имя, которое указывается в параметре

name

.

s = pd.Series([30, 35, 40], index=['Продажи 2015 года', 'Продажи 2016 года', 'Продажи 2017 года'], name='Товар A')

print(s)

Результат:

Продажи 2015 года 30

Продажи 2016 года 35

Продажи 2017 года 40

Name: Товар A, dtype: int64

## Чтение данных из файлов

Часто данные уже существуют и хранятся в различных форматах, таких как CSV и Excel. Мы можем использовать функцию

pd.read\_csv()

 для чтения данных из файла CSV и

pd.read\_excel()

 для чтения данных из Excel.

### Чтение данных из CSV-файла

**Функция**

pd.read\_csv()

 — используется для чтения данных из CSV-файла.

Скачайте и распакуйте zip-архив: [winemag-data-130k-v2.csv.zip](https://drive.google.com/file/d/1YwWPEqu-JXuMi35OtBHq2OzdERx1d4wf/view?usp=sharing). Используйте полученный CSV-файл для закрепления материала на практике.

Пример чтения CSV-файла:

wine\_reviews = pd.read\_csv("winemag-data-130k-v2.csv")

Чтобы получить информацию о размерности

DataFrame

 или

Series

, то есть о количестве строк и столбцов, используется **атрибут**

shape

.

Когда вы используете атрибут

shape

, он возвращает кортеж, где первый элемент — это количество строк, а второй элемент — это количество столбцов.

print(wine\_reviews.shape)

Результат:

(129971, 14)

Мы видим, что наш

DataFrame

 содержит почти 130 000 строк, разбитых на 14 разных столбцов.

Чтобы ознакомиться со структурой таблицы, можно вывести первые записи из таблицы.

**Метод**

head()

 — по умолчанию выводит первые пять строк таблицы. Вы можете указать другое количество строк в аргументе.

Пример, как вывести первые пять строк таблицы (по умолчанию):

print(wine\_reviews.head())

**Результат (вывод первых пяти строк таблицы):**



Чтобы вывести конкретное количество строк, передайте нужное количество строк в метод

head()

:

print(wine\_reviews.head(3))

**Результат:**



### Чтение данных из Excel-файла

**Функция**

pd.read\_excel()

 — используется для чтения данных из файла Excel. Этот метод обладает дополнительными параметрами, такими как имя листа или указание столбца, который следует использовать в качестве индекса.

Для корректной работы с Excel-файлами в

pandas

 необходимо дополнительно установить библиотеку

openpyxl

:

poetry add openpyxl

Пример чтения Excel-файла, в котором есть 10 строк и 5 столбцов:

excel\_data = pd.read\_excel("example\_data.xlsx")

print(excel\_data.shape)

print(excel\_data.head())

Результат строки

print(excel\_data.shape)

:

(10, 5) # 10 строк и 5 столбцов

Результат строки

print(excel\_data.head())

:

Column1

Column2

Column3

Column4

Column5

0

1

A

10.5

2022-01-01

True

1

2

B

13.6

2022-01-02

False

2

3

C

11.2

2022-01-03

True

3

4

D

9.3

2022-01-04

True

4

5

E

8.1

2022-01-05

False

Мы можем выбрать конкретный лист из Excel-файла при его чтении. Для этого нужно указать название листа в параметре

sheet\_name

:

excel\_data\_specific\_sheet = pd.read\_excel("example\_data.xlsx", sheet\_name='Sheet2')

print(excel\_data\_specific\_sheet.head())

Также можно указать столбец, который следует использовать в качестве индекса. Для этого необходимо указать в параметре

index\_col

 номер колонки, которая будет использоваться в качестве индекса:

excel\_data\_with\_index = pd.read\_excel("example\_data.xlsx", index\_col=0)

print(excel\_data\_with\_index.head())

Ссылка на Excel-файл, который используется при решении задач в видео: [winemag-data-130k-v2.xlsx](https://docs.google.com/spreadsheets/d/10ipksqUsQhu9n-VP-R5MQQ106IBh96aH/edit?usp=sharing&ouid=106765172812768673646&rtpof=true&sd=true).

# 4. Индексирование, выбор и присваивание

В этом разделе продолжаем работать с файлом [winemag-data-130k-v2.csv.zip](https://drive.google.com/file/d/1YwWPEqu-JXuMi35OtBHq2OzdERx1d4wf/view?usp=sharing).

Давайте разберем, как получать доступ к данным в датафрейме и модифицировать их.

Для начала загрузим необходимые данные:

import pandas as pd

reviews = pd.read\_csv("winemag-data-130k-v2.csv", index\_col=0)

Датасет, который мы загрузили, содержит информацию о винах различных сортов, например рейтинги, описания и регион происхождения.

## Доступ к столбцам таблицы

Столбцы таблицы можно получить двумя основными способами:

1. **Обращение через точку**, если имя столбца соответствует правилам именования переменных в Python:

reviews.country

1. **Через квадратные скобки и имя столбца в виде строки**:

reviews['country']

Оба способа вернут

Series

, содержащий значения столбца

country

.

Чтобы получить конкретное значение из столбца, можно воспользоваться индексом:

reviews['country'][0] # Способ 1

reviews.country[0] # Способ 2

## Обращение к данным по строкам через индексацию

**Индексация** — это процесс доступа к данным в датафрейме с использованием их позиции (номеров строк и столбцов) или меток (имен строк и столбцов). Индексация позволяет эффективно выбирать, модифицировать и анализировать данные, содержащиеся в датафрейме.

Существует два основных метода индексации в

pandas

:

1. iloc

**(Integer Location)** — позволяет осуществлять доступ к данным по их числовым индексам или позициям.

1. loc

**(Label Location)** — позволяет осуществлять доступ к данным по их меткам или именам.

### Индексация с помощью iloc

**Метод**

iloc

 — используется для индексации по числовым значениям, т. е. по порядковому номеру строки или столбца. Этот метод применяется, когда вам необходимо обращаться к данным по их позиции в датафрейме.

Синтаксис применения:

dataframe.iloc[индексы строк, индексы столбцов]

Где:

* индексы строк

 — здесь можно указать конкретные индексы (например,

0

 или

1

), диапазон (например,

0:3

), список индексов (например,

[0, 1, 2]

) либо пропустить, что означает выбор всех строк (

:

).

* индексы столбцов

 — указываются аналогично индексам строк: можно указать конкретную позицию столбца, диапазон, список индексов или пропустить для выбора всех столбцов.

Пример:

reviews.iloc[0] # Первая строка датафрейма

reviews.iloc[:, 0] # Первый столбец датафрейма

reviews.iloc[:3, 0] # Первые три строки первого столбца

reviews.iloc[1:3, 0] # Вторая и третья строки первого столбца

reviews.iloc[[0, 1, 2], 0] # Первые три строки первого столбца

reviews.iloc[-5:] # Последние пять строк датафрейма

### Индексация с помощью loc

**Метод**

loc

 — используется для индексации по меткам. Этот метод удобен, когда вы знаете метки строк или столбцов, к которым хотите получить доступ.

Синтаксис применения:

dataframe.loc[метки строк, метки столбцов]

Где:

* метки строк

 — здесь можно указать конкретные метки строк (например,

0

 или

'index\_label'

), диапазон (

'start':'end'

), список меток (например,

[label1, label2]

) либо пропустить для выбора всех строк (

:

).

* метки столбцов

 — указываются аналогично меткам строк: можно указать конкретные метки столбцов, диапазон, список меток или пропустить для выбора всех столбцов.

Пример:

reviews.loc[0, 'country'] # Значение первой строки в столбце `country`

reviews.loc[:, 'country'] # Все значения в столбце `country`

reviews.loc[:, ['taster\_name', 'taster\_twitter\_handle', 'points']] # Все значения в указанных столбцах

## Управление индексом

Управление индексом может быть особенно полезным в ряде ситуаций, когда нужно оптимизировать работу с данными или повысить удобство их анализа.

Когда данные имеют уникальные идентификаторы, такие как имена или номера, можно использовать эти идентификаторы в качестве индекса. Это позволяет легко и быстро выбирать конкретные строки. Например, если у вас есть датафрейм с информацией о студентах, установка их уникальных идентификаторов (например, номеров студенческих билетов) в качестве индекса облегчит поиск данных о конкретном студенте.

**Метод**

set\_index()

 в

pandas

 — позволяет управлять индексом датафрейма и используется для назначения одного или нескольких столбцов в качестве индекса датафрейма.

Синтаксис:

dataframe.set\_index(keys, drop=True, append=False, inplace=False, verify\_integrity=False)

Где:

* keys

 — строка или список строк, определяющих имя (имена) столбцов, которые будут назначены индексом.

* drop

 — логический параметр (по умолчанию

True

). Указывает, следует ли удалять использованный для индекса столбец из датафрейма. Если

False

, столбец будет сохранен.

* append

 — логический параметр (по умолчанию

False

). Указывает, нужно ли добавлять новый индекс к существующему многомерному индексу.

* inplace

 — логический параметр (по умолчанию

False

). Указывает, следует ли производить операцию на месте, не создавая новый датафрейм.

* verify\_integrity

 — логический параметр (по умолчанию

False

). Если

True

, проверяет, что новый индекс не содержит дубликатов, это гарантирует целостность данных.

Пример:

import pandas as pd

data = {

'student\_id': ['001', '002', '003'],

'name': ['Alice', 'Bob', 'Charlie'],

'grade': [87, 92, 78]

}

df = pd.DataFrame(data)

# Установить 'student\_id' в качестве индекса

df.set\_index('student\_id', inplace=True)

# Легкий доступ к данным конкретного студента

print(df.loc['001'])

## Отбор данных по условию

Библиотека

pandas

 позволяет выбирать строки или столбцы на основе условий. Синтаксис использования похож на использование операторов

if

.

Чтобы выбрать строки, где значение в столбце

country

 равно

"Italy"

, запишем:

italy\_reviews = reviews.loc[reviews.country == 'Italy']

print(italy\_reviews)

## Объединение условий с помощью логических операторов

Операторы логического объединения —

&

 (и) и

|

 (или) — позволяют гибко создавать сложные запросы.

### Использование логического оператора & (и)

Чтобы выбрать строки, где значение в столбце

country

 равно

"Italy"

**и** значение в столбце

points

 больше или равно 90, используйте оператор

&

:

high\_rating\_italy\_reviews = reviews.loc[(reviews.country == 'Italy') & (reviews.points >= 90)]

print(high\_rating\_italy\_reviews)

Обратите внимание: условия должны быть заключены в круглые скобки, иначе вы получите ошибку синтаксиса.

### Использование логического оператора | (или)

Чтобы выбрать строки, где значение в столбце

country

 равно

"Italy"

**или** значение в столбце

points

 больше или равно 90, используйте оператор

|

:

italy\_or\_high\_rating\_reviews = reviews.loc[(reviews.country == 'Italy') | (reviews.points >= 90)]

print(italy\_or\_high\_rating\_reviews)

# 5. Дополнительные операции для выборки

## Использование метода isin

**Метод**

isin

 — проверяет, содержится ли значение в определенном столбце в заданном списке значений.

Чтобы выбрать строки, где значение в столбце

country

 равно

"Italy"

 или

"France"

, используйте метод

isin

:

italy\_france\_reviews = reviews.loc[reviews.country.isin(['Italy', 'France'])]

print(italy\_france\_reviews)

В этом примере

reviews.country.isin(['Italy', 'France'])

 создает логический массив, где каждое значение будет

True

, если соответствующее значение в столбце

country

 равно

"Italy"

 или

"France"

, и

False

 — в противном случае.

Затем метод

loc

 используется для выбора строк, где условие

True

. В результате

italy\_france\_reviews

 будет содержать все строки из

reviews

, где значение в столбце

country

 равно

"Italy"

 или

"France"

.

## Использование метода notnull

**Метод**

notnull

 — проверяет, является ли значение в столбце не равным

NaN

 (неопределенным или отсутствующим).

Чтобы выбрать строки, где значение в столбце

price

 не равно NaN, т. е. строка содержит значение, используйте метод

notnull

:

not\_null\_price\_reviews = reviews.loc[reviews.price.notnull()]

print(not\_null\_price\_reviews)

В этом примере

reviews.price.notnull()

 создает логический массив, где каждое значение будет

True

, если соответствующее значение в столбце

price

 не является

NaN

, и

False

, если оно является

NaN

.

Метод

loc

 затем используется для выбора строк, где условие

True

. В результате

not\_null\_price\_reviews

 будет содержать все строки из

reviews

, где значение в столбце

price

 присутствует, т. е. не равно

NaN

.

## Многократное объединение условий

Можно объединить несколько условий, используя операторы

&

 и

|

 для создания сложных фильтров.

Рассмотрим пример. Необходимо выбрать строки, где значение в столбце

country

 равно

"Italy"

 или

"France"

 и значение в столбце

points

 больше или равно 90:

complex\_condition\_reviews = reviews.loc[reviews.country.isin(['Italy', 'France']) & (reviews.points >= 90)]

print(complex\_condition\_reviews)

Другой пример — требуется выбрать строки, где значение в столбце

country

 равно

"Italy"

 или значение в столбце

points

 больше или равно 90, а значение в столбце

price

 не равно

NaN

:

complex\_condition\_reviews = reviews.loc[((reviews.country == 'Italy') | (reviews.points >= 90)) & (reviews.price.notnull())]

print(complex\_condition\_reviews)

## Присваивание данных

Для присваивания данных в датафрейме используйте обращение через квадратные скобки и имя столбца в виде строки:

# Присвоить значение 'everyone' всем значениям в столбце `critic`

reviews['critic'] = 'everyone'

# Создание нового столбца с обратным порядком индексов

reviews['index\_backwards'] = range(len(reviews), 0, -1)

# 6. Статистика по данным и отображения

Понимание и умение применять статистические методы и функции отображения данных позволяют делать более осознанные выводы о данных и принимать обоснованные решения на их основе.

## Методы describe, mean, unique и value\_counts

Познакомимся с базовыми методами анализа данных в таблицах. Эти методы позволяют быстро получить общее представление о данных.

1. **Основные статистические характеристики столбца**

Например, если у нас есть столбец

points

 в таблице

reviews

, то мы можем использовать метод

describe()

, чтобы получить статистику по этому столбцу: среднее значение, стандартное отклонение, медиану и квартили.

Терминология статистики

Получаем статистику по столбцу

points

:

reviews.points.describe()

Результат будет представлен в следующей форме:

count 129971.000000

mean 88.447138

...

75% 91.000000

max 100.000000

Name: points, Length: 8, dtype: float64

1. **Статистика по текстовым столбцам**

Например, для столбца с именами дегустаторов

taster\_name

 также можно использовать метод

describe()

 для получения информации о количестве уникальных значений и наиболее часто встречающемся имени.

reviews.taster\_name.describe()

Результат выглядит так:

count 120650

unique 19

top Roger Voss

freq 25514

Name: taster\_name, dtype: object

1. **Средние значения и уникальные значения**
   * Среднее значение столбца

points

 можно получить с помощью метода

mean()

:

reviews.points.mean()

* + Для получения списка уникальных значений столбца

taster\_name

 используйте

unique()

:

reviews.taster\_name.unique()

* + Чтобы узнать количество упоминаний каждого имени, используйте

value\_counts()

:

reviews.taster\_name.value\_counts()

Пример результата:

Roger Voss 25514

Michael Schachner 15134

...

Fiona Adams 27

Christina Pickard 6

Name: taster\_name, Length: 19, dtype: int64

## Пример работы с функцией map

**Функция**

map

**в Python** — позволяет применить функцию к каждому элементу списка.

В контексте анализа данных это часто используется для преобразования данных из одного формата в другой.

Рассмотрим пример.

Перепишем оценки вина из 100-балльной системы в систему, где среднее значение равно 0:

review\_points\_mean = reviews.points.mean()

reviews.points.map(lambda p: p - review\_points\_mean)

Добавим новый столбец с длиной описания вина:

reviews['description\_length'] = reviews.description.map(len)

Теперь можно провести анализ нового столбца:

reviews['description\_length'].describe()

## Метод apply

**Метод**

apply

**в библиотеке**

pandas

 — позволяет применять пользовательские функции к каждой строке или столбцу

DataFrame

.

Это полезный инструмент для выполнения сложных операций и преобразований над данными.

Использование метода

apply

 позволяет гибко и эффективно выполнять операции над данными, которые не могут быть выполнены с использованием встроенных методов

pandas

.

Параметры метода

apply

:

* func

 — функция, которую необходимо применить.

* axis

 — параметр, который указывает, применять функцию к строкам (значение 1) или столбцам (значение 0). *Подробнее с этим параметром мы познакомимся далее.*

### Примеры использования apply

Предположим, у нас есть

DataFrame

 с несколькими столбцами, и мы хотим вычислить среднее значение для каждого столбца:

import pandas as pd

# Создадим DataFrame

df = pd.DataFrame({

'A': [1, 2, 3],

'B': [4, 5, 6],

'C': [7, 8, 9]

})

# Используем apply для вычисления среднего значения каждого столбца

result = df.apply(pd.Series.mean)

print(result)

Результат будет следующим:

A 2.0

B 5.0

C 8.0

dtype: float64

Также мы можем создать свою функцию, которая будет, например, вычислять разницу между максимумом и минимумом для каждого столбца, и применить ее с помощью метода

apply

 :

# Функция, вычисляющая разницу между максимумом и минимумом

def range\_of\_series(series):

return series.max() - series.min()

# Используем apply для применения функции

result = df.apply(range\_of\_series)

print(result)

Результат будет следующим:

A 2

B 2

C 2

dtype: int64

## Параметр axis

**Параметр**

axis

 — указывает направление применения метода по строкам или по столбцам:

* axis=0

 — функция применяется к каждому столбцу.

* axis=1

 — функция применяется к каждой строке.

### Примеры работы с параметром axis

1. **Применение по столбцам —**

axis=0

Давайте создадим

DataFrame

 и посмотрим, как суммируются значения по столбцам:

df = pd.DataFrame({

'A': [1, 2, 3],

'B': [4, 5, 6]

})

result = df.sum(axis=0)

print(result)

Результат будет следующим:

A 6

B 15

dtype: int64

1. **Применение по строкам —**

axis=1

:

Теперь применим ту же функцию

sum

, но по строкам:

result = df.sum(axis=1)

print(result)

Результат будет следующим:

0 5

1 7

2 9

dtype: int64

# 7. Группировка и сортировка

## Группировка данных

Часто при работе с данными возникает необходимость группировать их по определенным критериям. Например, если мы хотим узнать среднюю цену вина для каждой страны, мы можем использовать метод

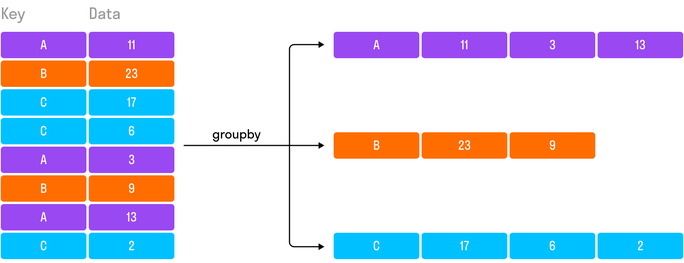
groupby()

.

**Метод**

groupby()

 — позволяет сгруппировать данные по одному или нескольким столбцам или по результатам какой-либо функции.



Синтаксис метода:

DataFrame.groupby(by=None, axis=0, level=None, as\_index=True, sort=True, group\_keys=True, squeeze=NoDefault.no\_default, observed=False, dropna=True)

Параметры метода:

* by

 — имя столбца или список имен столбцов для группировки, либо функция для группировки.

* axis

 — ось для группировки (по умолчанию

axis=0

, что означает группировку по строкам).

* sort

 — сортирует сгруппированные ключи (по умолчанию

True

).

* as\_index

 — определяет, будет ли результирующий объект

DataFrame

 или

Series

 содержать группировочные метки (уникальные значения или комбинации значений, которые используются для определения групп при выполнении группировки данных в

pandas

) как индекс. Если

as\_index=True

, метки групп будут использоваться в качестве индекса результирующего

DataFrame

. Если

False

, метки групп будут добавлены как обычные столбцы.

Другие параметры

### Примеры использования

1. **Группировка по одному столбцу:**

df.groupby('column\_name')

1. **Группировка по нескольким столбцам:**

df.groupby(['column1', 'column2'])

1. **Использование агрегирующих функций:**

df.groupby('column\_name').mean()

df.groupby(['column1', 'column2']).sum()

Здесь

df.groupby('column\_name')

 группирует строки

DataFrame

df

 по уникальным значениям в столбце

column\_name

. Метод

mean()

 вычисляет среднее значение для всех числовых столбцов в каждой группе. Результат — новый

DataFrame

, где индексом являются уникальные значения из

column\_name

, а значениями — средние значения для каждой группы.

1. **Группировка с использованием функции:**

df.groupby(lambda x: x // 10 \* 10)

Лямбда-функция берет индекс строки

x

, делит его на 10 с целочисленным делением (

x // 10

), затем умножает результат на 10 (

\* 10

). Это приводит к округлению индекса до ближайшего меньшего десятка. Например:

* + Если

x = 5

, то

5 // 10 \* 10

 будет 0.

* + Если

x = 15

, то

15 // 10 \* 10

 будет 10.

* + Если

x = 23

, то

23 // 10 \* 10

 будет 20.

Метод

groupby

 использует лямбда-функцию для определения групп. Каждая строка в

DataFrame

 будет отнесена к группе в зависимости от значения ее индекса, преобразованного этой функцией.

Таким образом мы получим:

* + **Группа с меткой 0** — индексы от 0 до 9.
  + **Группа с меткой 10** — индексы от 10 до 19.
  + **Группа с меткой 20** — индексы от 20 до 29, и так далее.

1. **Пример с несколькими параметрами:**

df.groupby(['column1', 'column2'], as\_index=False, sort=False)

Этот пример демонстрирует, как группировать строки

DataFrame

df

 по значениям в двух столбцах (

column1

 и

column2

) с дополнительными параметрами

as\_index=False

 и

sort=False

.

### Пример

Рассмотрим пример с группировкой данных и вычислением средней цены:

import pandas as pd

# Предположим, у нас есть DataFrame df с данными о винах

df = pd.DataFrame({

'country': ['France', 'Italy', 'Spain', 'France', 'Italy', 'Spain'],

'price': [20, 15, 30, 22, 18, 25]

})

# Группировка данных по странам

country\_grouped = df.groupby('country')

# Вычисление средней цены для каждой страны

mean\_price\_by\_country = country\_grouped['price'].mean()

print(mean\_price\_by\_country)

Что мы делаем в примере:

1. **Разделение данных по странам** — используем функцию

groupby

, чтобы разделить данные в таблице по столбцу

country

.

1. **Вычисление средней цены** — применяем функцию

mean

 к столбцу

price

 для каждой группы, чтобы посчитать среднюю цену.

## Сортировка данных

Сортировка данных позволяет упорядочить строки по значениям определенного столбца.

**Метод**

sort\_values()

 — используется, чтобы отсортировать данные по цене в порядке убывания.

**name**

**region**

**sales**

**expense**

Markus

South

34 000

44 000

Edward

West

42 000

38 000

William

East

50 000

42 000

Emma

North

52 000

43 000

Sofia

East

90 000

50 000

Синтаксис метода:

DataFrame.sort\_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na\_position='last', ignore\_index=False, key=None)

Параметры метода:

* by

 — название одного столбца или список из нескольких названий столбцов, по которым будет выполняться сортировка.

* axis

 — ось для сортировки (

0

 или

'index'

 — для строк;

1

 или

'columns'

 — для столбцов).

* ascending

 — сортировка по возрастанию (по умолчанию

True

); можно также указать список для задания порядка сортировки для нескольких столбцов.

* kind

 — метод сортировки; по умолчанию

'quicksort'

. Другие варианты:

'mergesort'

,

'heapsort'

,

'stable'

.

Подробнее про сортировки

Другие параметры

### Примеры использования метода

1. **Сортировка по одному столбцу:**

df.sort\_values(by='column\_name')

1. **Сортировка по нескольким столбцам:**

df.sort\_values(by=['column1', 'column2'])

1. **Сортировка по убыванию:**

df.sort\_values(by='column\_name', ascending=False)

1. **Сортировка «на месте»:**

df.sort\_values(by='column\_name', inplace=True)

1. **Сортировка с использованием определенного метода сортировки:**

df.sort\_values(by='column\_name', kind='mergesort')

Этот пример демонстрирует, как отсортировать строки

DataFrame

 по значениям в столбце

column\_name

, используя алгоритм сортировки слиянием —

mergesort

.

1. **Сортировка с размещением NaN-значений в начале:**

df.sort\_values(by='column\_name', na\_position='first')

1. **Сортировка с применением пользовательской функции:**

df.sort\_values(by='column\_name', key=lambda x: x.str.lower())

### Примеры

1. **Сортировка по цене в порядке возрастания:**

import pandas as pd

data = {

'country': ['France', 'Italy', 'Spain', 'France', 'Italy', 'Spain'],

'price': [20, 15, 30, 22, 18, 25]

}

df = pd.DataFrame(data)

# Сортировка по столбцу 'price' в порядке возрастания

sorted\_by\_price = df.sort\_values(by='price')

print(sorted\_by\_price)

1. **Сортировка по цене в порядке убывания:**

# Сортировка по столбцу 'price' в порядке убывания

sorted\_by\_price\_desc = df.sort\_values(by='price', ascending=False)

print(sorted\_by\_price\_desc)

1. **Сортировка по нескольким столбцам:**

data = {

'country': ['France', 'Italy', 'Spain', 'France', 'Italy', 'Spain'],

'region': ['Bordeaux', 'Tuscany', 'Rioja', 'Bordeaux', 'Sicily', 'Rioja'],

'price': [20, 15, 30, 22, 18, 25]

}

df = pd.DataFrame(data)

# Сортировка по столбцам 'country' и 'price'

sorted\_by\_country\_price = df.sort\_values(by=['country', 'price'])

print(sorted\_by\_country\_price)

1. **Сортировка с размещением NaN-значений в начале:**

data = {

'country': ['France', 'Italy', 'Spain', 'France', 'Italy', 'Spain', None],

'price': [20, 15, 30, 22, 18, 25, 17]

}

df = pd.DataFrame(data)

# Сортировка по столбцу 'country' с размещением NaN в начале

sorted\_with\_nan\_first = df.sort\_values(by='country', na\_position='first')

print(sorted\_with\_nan\_first)

1. **Сортировка с применением пользовательской функции:**

data = {

'country': ['FRANCE', 'italy', 'Spain', 'france', 'ITALY', 'spain'],

'price': [20, 15, 30, 22, 18, 25]

}

df = pd.DataFrame(data)

# Сортировка по столбцу 'country' с использованием лямбда-функции

sorted\_case\_insensitive = df.sort\_values(by='country', key=lambda x: x.str.lower())

print(sorted\_case\_insensitive)

## Одновременная группировка и сортировка данных

Методы

groupby()

 и

sort\_values()

 можно комбинировать для группировки и сортировки данных.

Например, чтобы отсортировать вина по цене в каждой стране, используем их вместе:

# Группировка данных по странам и сортировка по цене внутри каждой группы

sorted\_by\_price\_and\_country = df.groupby('country').apply(lambda x: x.sort\_values('price', ascending=False))

print(sorted\_by\_price\_and\_country.head())

Объясняем, что мы делаем в этом примере:

1. **Группировка** — сначала группируем данные по странам.

df.groupby('country')

1. **Сортировка внутри групп** — с помощью функции

apply

 применяем сортировку по цене внутри каждой группы.

df.groupby('country').apply(lambda x: x.sort\_values('price', ascending=False))

## Фильтрация данных

**Фильтрация** позволяет выбрать строки, которые удовлетворяют определенным условиям.

Например, чтобы выбрать вина, которые стоят более 100 долларов, используем оператор фильтрации —

df['price'] > 100

:

# Фильтрация вин, чья цена больше 100 долларов

expensive\_wines = df[df['price'] > 100]

print(expensive\_wines.head())

Что происходит в этом примере:

1. **Условие отбора** — строим логическое выражение

df['price'] > 100

 для фильтрации.

1. **Применение фильтра** — получаем строки, где цена вина больше 100 долларов.

## Группировка и фильтрация данных

Можно комбинировать

groupby()

 с фильтрацией данных.

Например, чтобы выбрать вина из Италии, которые стоят больше 100 долларов:

# Фильтрация вин из Италии

italian\_wines = df[df['country'] == 'Italy']

# Фильтрация дорогих итальянских вин

expensive\_italian\_wines = italian\_wines[italian\_wines['price'] > 100]

print(expensive\_italian\_wines.head())

Объясняем, что мы делаем в этом примере:

1. **Фильтрация по стране** — сначала выбираем все вина из Италии.
2. **Фильтрация по цене** — из отобранных вин выбираем только те, которые стоят больше 100 долларов.

## Группировка и агрегация данных

**Агрегация данных** — это процесс обобщения и свертывания данных с целью их анализа, упрощения и получения ключевой информации. Этот процесс включает различные операции для создания сводных показателей из более детализированных данных.

К агрегации относятся, например, суммирование, вычисление среднего значения, нахождение максимума и минимума, подсчет количества.

Для агрегации данных используется метод

agg()

.

**Метод**

agg

 (сокращение от aggregate) — позволяет выполнять несколько операций агрегации на различных столбцах

DataFrame

 одновременно.

Это полезно при анализе и обработке данных, когда необходимо применить разные функции агрегации к различным столбцам или несколько функций к одному столбцу. Например, чтобы вычислить среднюю цену и рейтинг для каждой страны.

Синтаксис:

DataFrame.agg(func, axis=0, \*args, \*\*kwargs)

Параметры метода:

* func

 — функция или словарь функций для агрегации. Возможные типы

func

:

* + Один оператор агрегации, например

'mean'

,

'sum'

.

* + Список операторов агрегации для применения к каждому столбцу.
  + Словарь, где ключи — это имена столбцов, а значения — функции агрегации или список функций для конкретного столбца.
* axis

 — ось для применения функции агрегации:

0

 или

'index'

 — для строк;

1

 или

'columns'

 — для столбцов.

* \*args

,

\*\*kwargs

 — дополнительные аргументы, передаваемые в функцию агрегации.

### Примеры использования метода

1. **Применение одной функции ко всем столбцам:**

df.agg('mean')

1. **Применение команды на несколько функций ко всем столбцам:**

df.agg(['mean', 'sum'])

1. **Применение функции к конкретным столбцам:**

df.agg({'column1': 'mean', 'column2': 'sum'})

1. **Применение нескольких функций к одному и нескольким столбцам:**
2. df.agg({
3. 'column1': ['mean', 'sum'],
4. 'column2': ['min', 'max']

})

### Пример

Рассмотрим пример, в котором сгруппируем данные по странам и вычислим среднюю цену и рейтинг:

# Группировка данных по странам и вычисление средней цены и рейтинга

mean\_price\_and\_rating\_by\_country = df.groupby('country').agg({'price': 'mean', 'points': 'mean'})

print(mean\_price\_and\_rating\_by\_country.head())

Что мы делаем в этом примере:

1. **Группировка** — группируем данные по странам.
2. **Агрегация** — применяем функцию

agg

 для вычисления среднего значения по столбцам

price

 и

points

.

## Перебор по cтрочкам или cтолбцам

### Перебор по строкам

**Метод**

iterrows()

 — используется для перебора строк в

DataFrame

.

Пример:

# Перебор строк DataFrame

for index, row in df.iterrows():

print(f'Index: {index}, Country: {row["country"]}, Price: {row["price"]}')

Объяснение примера:

1. **Метод**

iterrows()

 — возвращает итератор, который позволяет проходить по строкам

DataFrame

.

1. **Перебор строк** — внутри цикла можно использовать

.loc

 или

.at

 для доступа к значениям каждой строки.

### Перебор по столбцам

**Метод**

iteritems()

 — используется для перебора столбцов в

DataFrame

.

Пример:

# Перебор столбцов DataFrame

for column\_name, column\_data in df.iteritems():

print(f'Column: {column\_name}, Data: {column\_data.tolist()}')

Пояснение к примеру:

1. **Метод**

iteritems()

 — возвращает итератор, который позволяет проходить по столбцам

DataFrame

.

1. **Перебор столбцов** — внутри цикла можно использовать

.tolist()

, чтобы преобразовать столбец в список и работать с его значениями.