**Pandas**

План лекции:

1. Введение в Pandas.
2. Структуры данных в Pandas.
3. Работа с данными в Pandas.
4. Продвинутые возможности Pandas.
5. **Введение в Pandas**

Pandas – это мощная библиотека для анализа данных, построена поверх языка программирования Python. Она предоставляет высокоуровневые структуры данных и инструменты для эффективной обработки и анализа данных. Pandas является одной из наиболее популярных библиотек для работы с данными в Python, и широко используется в индустрии и научном сообществе.

Pandas предоставляет две основные структуры данных: Series и DataFrame. Series – это одномерная индексированная структура данных, которая может содержать различные типы данных, включая числа, строки и объекты Python. DataFrame – это двумерная структура данных, представляющая таблицу с рядами и столбцами. DataFrame может быть рассмотрен как аналог таблицы в реляционной базе данных или электронной таблицы Excel.

Основная цель Pandas – упростить работу с данными. Она предлагает множество функций и методов для чтения, записи, фильтрации, агрегирования, группирования и визуализации данных. Благодаря своим мощным функциональным возможностям, Pandas позволяет быстро и эффективно манипулировать большими объемами данных.

Давайте рассмотрим несколько ключевых возможностей Pandas:

**Чтение и запись данных**. Pandas может считывать данные из различных источников, таких как файлы CSV, Excel, SQL-базы данных и другие форматы данных. Она также может записывать данные в различные форматы. Функции чтения и записи данных в Pandas предоставляют гибкость и простоту в использовании.

**Индексирование и выборка данных**. Pandas предоставляет мощные инструменты для индексации и выборки данных. Вы можете использовать индексы для доступа к определенным значениям, столбцам или строкам в DataFrame. Вы также можете фильтровать данные с использованием условий и логических операторов.

**Манипуляции с данными**. Pandas предлагает широкий набор функций для манипулирования данными. Вы можете изменять значения в столбцах, добавлять или удалять столбцы и строки, выполнять агрегацию данных, объединять и разделять DataFrame, сортировать данные и многое другое. Это делает Pandas очень гибкой библиотекой для обработки и подготовки данных перед анализом.

**Группировка и агрегация данных**. Pandas позволяет группировать данные по определенным критериям и выполнять агрегацию по группам. Вы можете вычислять средние значения, суммы, медианы и другие статистические показатели для каждой группы данных. Это особенно полезно при анализе больших наборов данных и выявлении паттернов и тенденций.

**Визуализация данных**. Pandas интегрируется с другой популярной библиотекой Python - Matplotlib, что позволяет легко создавать графики и визуализации данных. Вы можете построить диаграммы рассеяния, столбчатые диаграммы, линейные графики и многое другое, чтобы исследовать и представить ваши данные.

**Установка Pandas.**

Убедитесь, что у вас установлен Python на вашем компьютере. Вы можете загрузить и установить его с официального сайта Python (https://www.python.org/), следуя инструкциям для вашей операционной системы.

Откройте командную строку или терминал на вашем компьютере.

Введите следующую команду для установки Pandas с использованием pip (установщика пакетов Python):

pip install pandas

Нажмите Enter, чтобы выполнить команду. Pip начнет загрузку и установку Pandas и его зависимостей.

После завершения установки Pandas будет готов к использованию.

**Импорт Pandas в проект**:

Откройте свой Python-скрипт или среду разработки.

Добавьте следующую строку в начало своего скрипта, чтобы импортировать Pandas:

Здесь мы используем сокращение "pd" для Pandas, чтобы обращаться к его функциям и объектам.

Теперь вы можете использовать Pandas в своем проекте, вызывая его функции и методы через префикс "pd".

Например, вы можете создать DataFrame, используя функцию pd.DataFrame(), или применить методы для обработки данных, такие как df.head() для просмотра первых строк DataFrame.

1. **Структуры данных в Pandas**

В Pandas, структура данных Series представляет собой одномерный массив с метками (индексами) для каждого элемента. Она может содержать данные различных типов, включая числа, строки, булевы значения и объекты Python.

Чтобы создать Series в Pandas, вы можете использовать функцию pd.Series(). Вот пример:

import pandas as pd

# Создание Series из списка чисел

numbers = [1, 2, 3, 4, 5]

series = pd.Series(numbers)

print(series)

В этом примере мы создали Series из списка чисел [1, 2, 3, 4, 5]. Результат будет выглядеть примерно так:

0 1

1 2

2 3

3 4

4 5

dtype: int64

Здесь каждый элемент списка стал значением в Series, а слева от значений отображаются их индексы (0, 1, 2, 3, 4). Тип данных в этом примере – int64.

Series обладает множеством полезных методов и атрибутов, которые могут быть применены к нему для выполнения различных операций и анализа данных. Некоторые из них:

series.values: возвращает массив значений в Series.

series.index: возвращает индексы в Series.

series.head(n): возвращает первые n элементов в Series.

series.tail(n): возвращает последние n элементов в Series.

series.describe(): вычисляет основные статистические показатели о Series, такие как среднее, стандартное отклонение, минимум, максимум и другие.

series.unique(): возвращает уникальные значения в Series.

series.value\_counts(): считает количество появлений каждого значения в Series.

Также можно обращаться к элементам Series по их индексу и выполнять различные операции фильтрации и преобразования данных.

Создание и индексирование Series в Pandas. Рассмотрим несколько способов.

**Создание Series с пользовательскими индексами**:

import pandas as pd

data = [10, 20, 30, 40, 50]

index = ['a', 'b', 'c', 'd', 'e']

series = pd.Series(data, index=index)

print(series)

В этом примере мы создаем Series с пользовательскими индексами, заданными в списке ['a', 'b', 'c', 'd', 'e'].

**Индексирование Series**:

import pandas as pd

data = [10, 20, 30, 40, 50]

index = ['a', 'b', 'c', 'd', 'e']

series = pd.Series(data, index=index)

print(series['b']) # Доступ к элементу по индексу

print(series[2]) # Доступ к элементу по позиции

print(series[1:4]) # Индексация по диапазону

В этом примере мы показываем различные способы доступа к элементам Series. Мы можем использовать индекс 'b' для доступа к конкретному элементу, позицию 2 для доступа к элементу по позиции и диапазон [1:4] для получения среза элементов.

**Фильтрация элементов в Series**:

import pandas as pd

data = [10, 20, 30, 40, 50]

index = ['a', 'b', 'c', 'd', 'e']

series = pd.Series(data, index=index)

filtered\_series = series[series > 30]

print(filtered\_series)

Мы можем фильтровать элементы в Series с помощью условных выражений. В этом примере мы фильтруем элементы, оставляя только те, которые больше 30.

**Изменение элементов в Series**:

import pandas as pd

data = [10, 20, 30, 40, 50]

index = ['a', 'b', 'c', 'd', 'e']

series = pd.Series(data, index=index)

series['c'] = 35 # Изменение значения элемента по индексу

series[1:4] = 0 # Изменение значения элементов по позиции

print(series)

Мы можем изменять значения элементов в Series, присваивая новые значения по индексу или позиции.

**Удаление элементов из Series**:

import pandas as pd

data = [10, 20, 30, 40, 50]

index = ['a', 'b', 'c', 'd', 'e']

series = pd.Series(data, index=index)

new\_series = series.drop('c') # Удаление элемента по индексу

print(new\_series)

Мы можем удалять элементы из Series с помощью метода drop(), указывая индекс элемента, который мы хотим удалить.

Теперь давайте перейдем к более мощной структуре данных Pandas, которая называется DataFrame.

DataFrame – это двумерная структура данных, представляющая таблицу с рядами и столбцами. DataFrame позволяет нам организовывать и манипулировать данными более сложным образом. Он имеет много функций для чтения, записи, фильтрации, агрегирования и визуализации данных.

**Создание DataFrame**.

Самый простой способ создания DataFrame – использовать словарь или двумерный массив NumPy. Вот несколько примеров:

import pandas as pd

# Создание DataFrame из словаря

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']}

df1 = pd.DataFrame(data)

print(df1)

# Создание DataFrame из двумерного массива NumPy

import numpy as np

numpy\_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

df2 = pd.DataFrame(numpy\_array, columns=['A', 'B', 'C'])

print(df2)

В первом примере мы создаем DataFrame из словаря, где ключи словаря становятся названиями столбцов, а значения словаря становятся данными столбцов. Во втором примере мы создаем DataFrame из двумерного массива NumPy и задаем названия столбцов.

**Индексация в DataFrame**.

Доступ к данным в DataFrame можно получить с использованием различных методов. Вот несколько примеров:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']}

df = pd.DataFrame(data)

print(df['Имя']) # Выбор столбца по названию

print(df.Возраст) # Выбор столбца через атрибут

print(df.loc[1]) # Выбор строки по индексу

print(df.loc[1, 'Город']) # Выбор значения по индексу строки и названию столбца

print(df.iloc[0]) # Выбор строки по позиции

print(df.iloc[0, 1]) # Выбор значения по позиции строки и столбца

Мы можем использовать оператор квадратных скобок или атрибуты для выбора столбцов в DataFrame. Для выбора строк мы можем использовать методы .loc[] или .iloc[], где .loc[] используется для индексирования по меткам, а .iloc[] - по позициям.

**Фильтрация в DataFrame**.

Мы можем фильтровать данные в DataFrame, используя условные выражения. Вот пример:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']}

df = pd.DataFrame(data)

filtered\_df = df[df['Возраст'] > 30]

print(filtered\_df)

Мы фильтруем DataFrame, оставляя только те строки, в которых значение в столбце 'Возраст' больше 30.

**Изменение и удаление данных в DataFrame**.

Мы можем изменять значения в DataFrame и удалять столбцы или строки. Вот несколько примеров:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']}

df = pd.DataFrame(data)

df['Город'] = 'Санкт-Петербург' # Изменение значений в столбце

df.loc[1, 'Возраст'] = 31 # Изменение значения по индексу строки и названию столбца

df.drop('Возраст', axis=1, inplace=True) # Удаление столбца

df.drop(0, inplace=True) # Удаление строки

print(df)

В этом примере мы изменяем значения в столбце, изменяем отдельное значение по индексу строки и названию столбца, а также удаляем столбец и строку из DataFrame.

Работа с пропущенными данными (NaN, None) является важной частью анализа данных. Pandas предоставляет функции и методы для обнаружения, обработки и заполнения пропущенных данных в DataFrame. Вот некоторые методы, которые можно использовать:

**Обнаружение пропущенных данных**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', None, 'Чарли'],

'Возраст': [25, None, 35, 40],

'Город': ['Москва', 'Нью-Йорк', 'Лондон', None]}

df = pd.DataFrame(data)

print(df.isnull()) # Возвращает DataFrame с булевыми значениями, указывающими на пропущенные данные

print(df.isnull().sum()) # Суммирует количество пропущенных данных по столбцам

Метод .isnull() позволяет обнаружить пропущенные данные в DataFrame. Он возвращает DataFrame той же формы, что и исходный, но заполняет ячейки с пропущенными данными значением True, а в остальных ячейках - False. С помощью метода .sum() мы можем посчитать количество пропущенных данных по каждому столбцу.

**Удаление пропущенных данных**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', None, 'Чарли'],

'Возраст': [25, None, 35, 40],

'Город': ['Москва', 'Нью-Йорк', 'Лондон', None]}

df = pd.DataFrame(data)

df.dropna(how=’any’, inplace=True) # Удаляет строки с пропущенными данными

df.dropna(axis=1) # Удаляет столбцы с пропущенными данными

Метод .dropna() позволяет удалить строки или столбцы, содержащие пропущенные данные. По умолчанию, метод удаляет строки, но с помощью параметра axis=1, мы можем удалить столбцы.

**Заполнение пропущенных данных**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', None, 'Чарли'],

'Возраст': [25, ‘None’, 35, 40],

'Город': ['Москва', 'Нью-Йорк', 'Лондон', None]}

df = pd.DataFrame(data)

df['Возраст'].fillna(df['Возраст'].mean(), inplace=True) # Заполняет пропущенные данные средним значением столбца

df.fillna('Неизвестно') # Заполняет пропущенные данные заданным значением

Метод .fillna() позволяет заполнить пропущенные данные заданным значением. Мы можем указать значение в качестве аргумента метода. Также можно заполнить пропущенные данные средним значением или любым другим вычисленным значением, например, медианой или модой столбца.

В Pandas, агрегирование данных в DataFrame позволяет объединять данные на основе определенных критериев и вычислять агрегатные функции для каждой группы. Это полезно при анализе данных и получении сводной информации. Давайте рассмотрим некоторые методы для группировки и применения агрегатных функций:

Группировка данных:

import pandas as pd

data = {'Город': ['Москва', 'Москва', 'Санкт-Петербург', 'Санкт-Петербург', 'Москва'],

'Год': [2018, 2019, 2018, 2019, 2019],

'Продажи': [100, 150, 200, 120, 180]}

df = pd.DataFrame(data)

grouped = df.groupby('Город') # Группировка по столбцу 'Город'

print(grouped.groups) # Вывод групп и их индексов

Метод .groupby() позволяет сгруппировать данные по определенному столбцу или нескольким столбцам. Он возвращает объект GroupBy, который содержит информацию о группах и их индексах.

Применение агрегатных функций:

import pandas as pd

data = {'Город': ['Москва', 'Москва', 'Санкт-Петербург', 'Санкт-Петербург', 'Москва'],

'Год': [2018, 2019, 2018, 2019, 2019],

'Продажи': [100, 150, 200, 120, 180]}

df = pd.DataFrame(data)

grouped = df.groupby('Город')

print(grouped.sum()) # Сумма значений для каждой группы

print(grouped.mean()) # Среднее значение для каждой группы

print(grouped.max()) # Максимальное значение для каждой группы

print(grouped.min()) # Минимальное значение для каждой группы

print(grouped.count()) # Количество элементов в каждой группе

print(grouped.describe()) # Основные статистические показатели для каждой группы

Мы можем применять различные агрегатные функции, такие как sum(), mean(), max(), min(), count(), describe(), к объекту GroupBy. Эти функции вычисляют агрегатные значения для каждой группы и возвращают результаты в виде нового DataFrame.

Комбинированная группировка и агрегирование:

import pandas as pd

data = {'Город': ['Москва', 'Москва', 'Санкт-Петербург', 'Санкт-Петербург', 'Москва'],

'Год': [2018, 2019, 2018, 2019, 2019],

'Продажи': [100, 150, 200, 120, 180]}

df = pd.DataFrame(data)

grouped = df.groupby(['Город', 'Год']) # Группировка по нескольким столбцам

print(grouped.sum()) # Сумма значений для каждой комбинации 'Город' и 'Год'

Мы можем также группировать данные по нескольким столбцам, указав список столбцов в методе groupby(). Это позволяет создавать более детальные группы и вычислять агрегатные значения для каждой комбинации столбцов.

В Pandas, для соединения и объединения DataFrame используются различные методы. Эти операции позволяют комбинировать данные из нескольких DataFrame, основываясь на определенных условиях или просто объединяя их вместе. Давайте рассмотрим некоторые из этих методов:

Соединение по столбцам (Merge):

import pandas as pd

data1 = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35]}

data2 = {'Имя': ['Алиса', 'Боб', 'Дэвид'],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']}

df1 = pd.DataFrame(data1)

df2 = pd.DataFrame(data2)

merged = pd.merge(df1, df2, on='Имя') # Соединение по общему столбцу 'Имя'

print(merged)

Метод .merge() позволяет соединить два DataFrame по указанному столбцу или нескольким столбцам. В этом примере мы соединяем df1 и df2 по столбцу 'Имя', и получаем DataFrame, который объединяет данные из обоих DataFrame.

Объединение по строкам (Concatenate):

import pandas as pd

data1 = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35]}

data2 = {'Имя': ['Дэвид', 'Эмили', 'Филипп'],

'Возраст': [28, 32, 37]}

df1 = pd.DataFrame(data1)

df2 = pd.DataFrame(data2)

concatenated = pd.concat([df1, df2]) # Объединение по строкам

print(concatenated)

Метод .concat() позволяет объединить несколько DataFrame по строкам. В данном примере мы объединяем df1 и df2 в один DataFrame, где данные просто присоединяются друг за другом.

Объединение по индексу (Join):

import pandas as pd

data1 = {'Возраст': [25, 30, 35]}

data2 = {'Зарплата': [50000, 60000, 70000]}

df1 = pd.DataFrame(data1, index=['Алиса', 'Боб', 'Чарли'])

df2 = pd.DataFrame(data2, index=['Алиса', 'Боб', 'Чарли'])

joined = df1.join(df2) # Объединение по индексу

print(joined)

Метод .join() позволяет объединить два DataFrame по индексу. В этом примере мы объединяем df1 и df2 по индексу, что позволяет нам объединить соответствующие данные из обоих DataFrame.

Объединение по условию (Merge с параметром how):

import pandas as pd

data1 = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']}

data2 = {'Имя': ['Алиса', 'Боб', 'Дэвид'],

'Возраст': [25, 30, 35]}

df1 = pd.DataFrame(data1)

df2 = pd.DataFrame(data2)

merged\_inner = pd.merge(df1, df2, on='Имя', how='inner') # Внутреннее объединение

merged\_outer = pd.merge(df1, df2, on='Имя', how='outer') # Внешнее объединение

merged\_left = pd.merge(df1, df2, on='Имя', how='left') # Левое объединение

merged\_right = pd.merge(df1, df2, on='Имя', how='right') # Правое объединение

print(merged\_inner)

print(merged\_outer)

print(merged\_left)

print(merged\_right)

Метод .merge() позволяет указать параметр how, который определяет тип объединения: inner (внутреннее объединение), outer (внешнее объединение), left (левое объединение) или right (правое объединение). В результате мы получаем объединенные DataFrame в соответствии с заданным типом объединения.

1. **Работа с данными в Pandas**

В Pandas есть функциональность для чтения и записи данных в различных форматах, таких как CSV и Excel. Давайте рассмотрим некоторые методы для выполнения этих операций:

**Чтение данных из CSV**:

import pandas as pd

df = pd.read\_csv('data.csv')

print(df)

Метод read\_csv() позволяет читать данные из CSV-файла и создавать DataFrame на их основе. Вы можете указать путь к файлу в качестве аргумента метода. CSV-файл должен содержать данные, разделенные запятыми (или другим разделителем), и может иметь заголовок с названиями столбцов.

**Запись данных в CSV**:

import pandas as pd

df = pd.DataFrame({'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']})з

df.to\_csv('data.csv', index=False)

**Запись данных в Excel**:

Рекомендовано установить дополнительный модуль открытия Excel файлов: pip install openpyxl

import pandas as pd

df = pd.DataFrame({'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']})

df.to\_excel('data.xlsx', sheet\_name='Sheet1', index=False)

Метод to\_excel() позволяет записывать данные из DataFrame в файл Excel. Вы можете указать путь к файлу и имя листа Excel в качестве аргументов метода. Параметр index=False указывает на то, что индексы строк не должны быть записаны в файл.

**Чтение данных из Excel:**

import pandas as pd

df = pd.read\_excel('data.xlsx', sheet\_name='Sheet1')

print(df)

Метод read\_excel() позволяет читать данные из файла Excel и создавать DataFrame на их основе. Вы можете указать путь к файлу и имя листа Excel в качестве аргументов метода. Файл Excel должен содержать таблицу данных, и каждый лист может иметь свое имя.

Манипуляции с данными в Pandas позволяют выполнять различные операции по преобразованию и обработке данных в DataFrame. Давайте рассмотрим некоторые основные манипуляции с данными:

**Добавление новых столбцов**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35]}

df = pd.DataFrame(data)

df['Город'] = ['Москва', 'Нью-Йорк', 'Лондон'] # Добавление нового столбца

df['Зарплата'] = [50000, 60000, 70000] # Добавление нового столбца с данными

print(df)

Мы можем добавлять новые столбцы в DataFrame, присваивая им значения или список значений.

**Удаление столбцов или строк**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35],

'Город': ['Москва', 'Нью-Йорк', 'Лондон']}

df = pd.DataFrame(data)

df.drop('Город', axis=1, inplace=True) # Удаление столбца

df.drop(1, inplace=True) # Удаление строки по индексу

print(df)

Мы можем удалять столбцы или строки из DataFrame с помощью метода drop(). Параметр axis=1 указывает на удаление столбца, а axis=0 (по умолчанию) - на удаление строки.

**Преобразование значений в столбцах**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35]}

df = pd.DataFrame(data)

df['Возраст'] = df['Возраст'] + 1 # Увеличение значений столбца на 1

df['Возраст'] = df['Возраст'].apply(lambda x: x \* 2) # Применение функции к значениям столбца

print(df)

Мы можем преобразовывать значения в столбцах, выполняя арифметические операции или применяя пользовательские функции с помощью метода apply().

**Сортировка данных**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35]}

df = pd.DataFrame(data)

df.sort\_values(by='Возраст', ascending=False, inplace=True) # Сортировка по столбцу 'Возраст'

print(df)

Мы можем сортировать данные в DataFrame по значениям столбца с помощью метода sort\_values(). Параметр by указывает на столбец, по которому производится сортировка, а ascending=False указывает на сортировку по убыванию.

**Применение функций к столбцам**:

import pandas as pd

data = {'Возраст': [25, 30, 35]}

df = pd.DataFrame(data)

df['Возраст\_в\_годах'] = df['Возраст'].apply(lambda x: x / 12) # Применение функции к столбцу и создание нового столбца

print(df)

Мы можем применять пользовательские функции к столбцам DataFrame с помощью метода apply(). В этом примере мы создаем новый столбец, в котором значения столбца 'Возраст' преобразуются из месяцев в годы.

**Удаление полных дубликатов строк**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35, 30, 35]}

df = pd.DataFrame(data)

df.drop\_duplicates(inplace=True) # Удаление полных дубликатов строк

print(df)

Метод drop\_duplicates() позволяет удалить полные дубликаты строк из DataFrame. Параметр inplace=True указывает на изменение DataFrame без необходимости присваивания результата новой переменной.

**Удаление дубликатов по определенным столбцам**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35, 30, 35]}

df = pd.DataFrame(data)

df.drop\_duplicates(subset=['Имя'], inplace=True) # Удаление дубликатов по столбцу 'Имя'

print(df)

Мы можем указать столбцы, по которым нужно проверить наличие дубликатов, с помощью параметра subset в методе drop\_duplicates(). В этом примере мы удаляем дубликаты строк на основе столбца 'Имя'.

**Удаление дубликатов и сохранение первого вхождения**:

import pandas as pd

data = {'Имя': ['Алиса', 'Боб', 'Чарли', 'Боб', 'Чарли'],

'Возраст': [25, 30, 35, 30, 35]}

df = pd.DataFrame(data)

df.drop\_duplicates(keep='first', inplace=True) # Удаление дубликатов и сохранение первого вхождения

print(df)

Параметр keep в методе drop\_duplicates() позволяет указать, какое вхождение дубликата нужно сохранить. Значение 'first' (по умолчанию) указывает на сохранение первого вхождения, 'last' указывает на сохранение последнего вхождения, а False указывает на удаление всех вхождений дубликата.

Pandas предоставляет возможности для базовой визуализации данных, которые могут быть полезны при исследовании и анализе данных. Визуализация помогает наглядно представить информацию и выявить паттерны и тренды. Вот некоторые методы для визуализации данных с помощью Pandas:

**Гистограмма (Histogram)**:

import pandas as pd

import matplotlib.pyplot as plt

data = {'Возраст': [25, 25, 30, 30, 30, 35, 40, 40, 45, 50, 50, 55, 65]}

df = pd.DataFrame(data)

df['Возраст'].plot.hist() # Построение гистограммы

# Дополнительные параметры для настройки графика (название, метки осей, размер и т.д.)

plt.title('Распределение возраста')

plt.xlabel('Возраст')

plt.ylabel('Частота')

plt.show() # Отображение графика

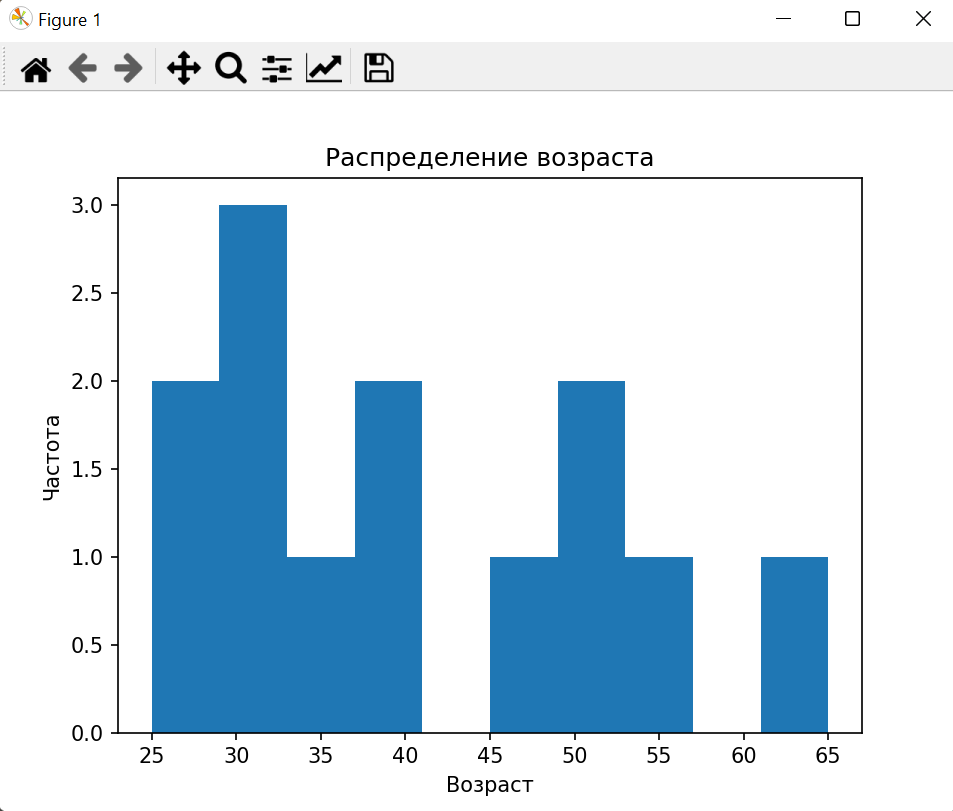


Figure Гистограмма

Метод plot.hist() позволяет построить гистограмму для выбранного столбца в DataFrame. Вы можете использовать дополнительные параметры, такие как title, xlabel, ylabel, для настройки графика.

**Круговая диаграмма (Pie Chart):**

import pandas as pd

import matplotlib.pyplot as plt

data = {'Страна': ['Россия', 'США', 'Китай', 'Германия', 'Франция'],

'Население': [144, 331, 1444, 83, 67]}

df = pd.DataFrame(data)

df.plot.pie(y='Население', labels=df['Страна'], autopct='%1.1f%%') # Построение круговой диаграммы

plt.title('Доля населения по странам')

plt.ylabel('')

plt.show()

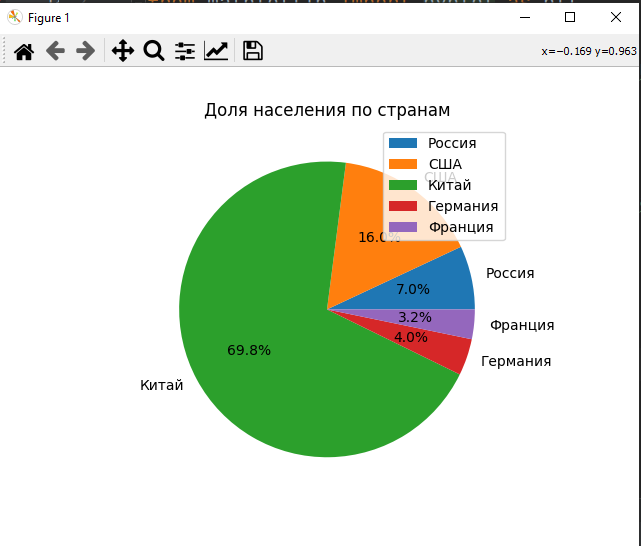


Figure Круговая диаграмма

Метод plot.pie() позволяет построить круговую диаграмму для выбранного столбца в DataFrame. Вы можете использовать параметры y для выбора столбца, содержащего данные, labels для меток секторов и autopct для отображения процентного значения в каждом секторе.

**Линейный график (Line Plot):**

import pandas as pd

import matplotlib.pyplot as plt

data = {'Год': [2015, 2016, 2017, 2018, 2019],

'Продажи': [100, 150, 200, 120, 180]}

df = pd.DataFrame(data)

df.plot(x='Год', y='Продажи') # Построение линейного графика

plt.title('Динамика продаж')

plt.xlabel('Год')

plt.ylabel('Продажи')

plt.show()

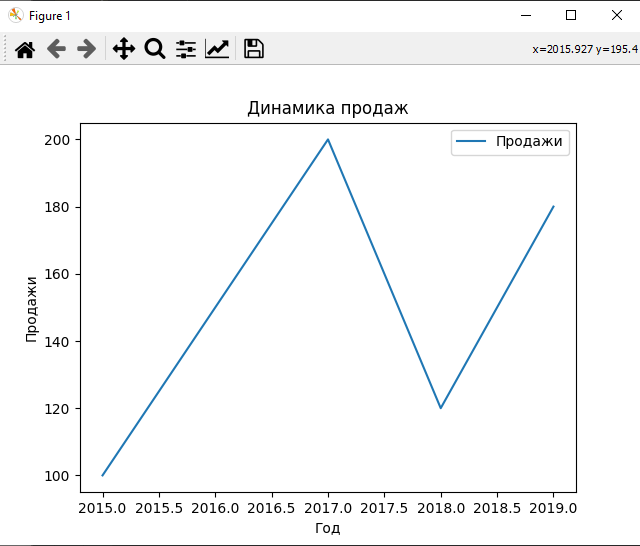


Figure Линейный график

Метод plot() с указанием столбцов x и y позволяет построить линейный график для выбранных столбцов в DataFrame. Вы можете использовать параметры title, xlabel, ylabel для настройки графика.

**Диаграмма разброса (Scatter Plot):**

import pandas as pd

import matplotlib.pyplot as plt

data = {'Возраст': [25, 30, 35, 40, 45, 50, 55, 60, 65],

'Зарплата': [50000, 60000, 70000, 80000, 90000, 100000, 110000, 120000, 130000]}

df = pd.DataFrame(data)

df.plot.scatter(x='Возраст', y='Зарплата') # Построение диаграммы разброса

plt.title('Связь между возрастом и зарплатой')

plt.xlabel('Возраст')

plt.ylabel('Зарплата')

plt.show()

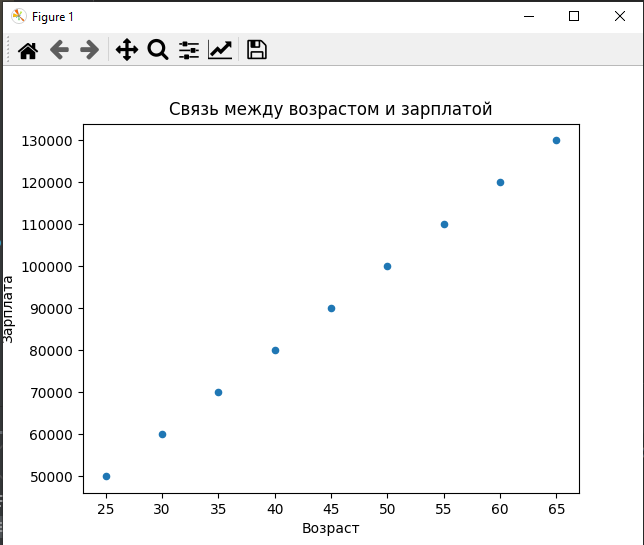


Figure Диаграмма разброса

Метод plot.scatter() позволяет построить диаграмму разброса для выбранных столбцов в DataFrame. Вы можете использовать параметры title, xlabel, ylabel для настройки графика.

1. **Продвинутые возможности Pandas**

**Работа с текстовыми данными:**

import pandas as pd

data = {'Название': ['Apple', 'Banana', 'Cherry'],

'Описание': ['Яблоко красное', 'Банан желтый', 'Вишня кислая']}

df = pd.DataFrame(data)

# Использование методов для работы с текстовыми данными

df['Название\_в\_верхнем\_регистре'] = df['Название'].str.upper()

df['Описание\_длина'] = df['Описание'].str.len()

print(df)

Методы, начинающиеся с .str, позволяют выполнять операции со строками в столбцах, содержащих текстовые данные. В данном примере мы использовали метод .upper() для преобразования названий фруктов в верхний регистр и метод .len() для вычисления длины описаний.

**Работа с категориальными данными:**

import pandas as pd

data = {'Фрукт': ['Яблоко', 'Банан', 'Яблоко', 'Вишня', 'Банан']}

df = pd.DataFrame(data)

# Преобразование столбца в категориальный тип данных

df['Фрукт'] = df['Фрукт'].astype('category')

print(df['Фрукт'].dtype)

print(df['Фрукт'].cat.categories)

Мы можем преобразовать столбец в категориальный тип данных с помощью метода .astype('category'). Это может быть полезно, когда у нас есть ограниченное количество уникальных значений в столбце. В примере мы проверяем тип данных столбца и выводим доступные категории.

**Создание своих функций для применения к данным:**

import pandas as pd

data = {'Зарплата': [50000, 60000, 70000, 80000]}

df = pd.DataFrame(data)

# Создание пользовательской функции и ее применение к столбцу

def double\_salary(salary):

return salary \* 2

df['Удвоенная\_зарплата'] = df['Зарплата'].apply(double\_salary)

print(df)

Мы можем создавать свои пользовательские функции и применять их к столбцам с помощью метода .apply(). В данном примере мы создаем функцию double\_salary(), которая удваивает значение зарплаты для каждой строки.

**Работа с многомерными индексами (MultiIndex):**

import pandas as pd

# Генерация большого DataFrame

data = {'ID': range(1000000),

'Зарплата': range(1000000)}

df = pd.DataFrame(data)

# Использование типа данных с фиксированной точностью

df['Зарплата'] = df['Зарплата'].astype('int32')

print(df.dtypes)

Мы можем оптимизировать производительность работы с большими объемами данных, устанавливая подходящие типы данных. В данном примере мы использовали тип данных 'int32' для столбца 'Зарплата', чтобы уменьшить использование памяти.

В заключение, мы рассмотрели основные возможности библиотеки pandas и узнали, как она может быть полезна при работе с данными. Мы изучили основные структуры данных, такие как Series и DataFrame, и научились выполнять на них различные операции, включая фильтрацию, сортировку и группировку данных.

Мы также рассмотрели способы обработки пропущенных значений и работы с временными рядами. Pandas предоставляет мощные инструменты для анализа данных, и мы изучили некоторые из них, такие как агрегирование данных, применение функций к группам данных и объединение данных из разных источников.

Важным аспектом работы с pandas является эффективная обработка больших объемов данных. Мы рассмотрели некоторые методы оптимизации производительности, такие как векторизация операций и использование индексов.