Шпаргалка по Pandas / Pandas Cheatsheet (XeLaTeX)

Создано с использованием LaTeX шаблона

March 30, 2025

Contents

1 Основные Структуры / Core Structures

Series (1D) и DataFrame (2D)

Series: Одномерный индексированный массив. Аналог столбца таблицы или словаря Python. ● **DataFrame**: Двумерная табличная структура с индексированными строками и столбцами. Основной объект для анализа данных. Состоит из объектов Series.

Создание Series и DataFrame

```python
import pandas as pd
import numpy as np

# Series (индекс по умолчанию)
s = pd.Series([10, 20, np.nan, 40])

# DataFrame из словаря Python
data = {'col_A': [1, 2, 3], 'col_B': ['X', 'Y', 'Z']}
dates_index = pd.date_range('20240101', periods=3)
df = pd.DataFrame(data, index=dates_index)
# print(s)
# print(df)
```

2 Загрузка и Сохранение / IO

Чтение и запись данных

Ключевые функции для взаимодействия с файлами.

`pd.read_csv('f.csv', sep=',')`

Чтение CSV файла. Важные параметры: ‘sep‘, ‘header‘, ‘index_col‘, ‘usecols‘, ‘parse_dates‘, ‘dtype‘.

`df.to_csv('out.csv', index=False)`

Запись DataFrame в CSV. Важные параметры: ‘sep‘, ‘index‘, ‘header‘, ‘columns‘.

`pd.read_excel('f.xlsx', sheet_name=0)`

Чтение из файла Excel. Важные параметры: ‘sheet_name‘, ‘header‘, ‘index_col‘.

`df.to_excel('out.xlsx', sheet_name='Data')`

Запись в Excel. Важные параметры: ‘sheet_name‘, ‘index‘, ‘header‘.

`pd.read_sql(query, conn)`

Чтение из SQL базы данных (требует ‘sqlalchemy‘ или аналог).

`pd.read_json('f.json', orient='records')`

Чтение из JSON файла/строки. ‘orient‘ важен для структуры.

3 Первичный Осмотр / Basic Inspection

Первичный анализ DataFrame (‘df‘)

Получение общего представления о данных.

Команды для осмотра ‘df‘

```python
df.head(3)      # Первые N строк (по умолч. 5)
df.tail(3)      # Последние N строк (по умолч. 5)
df.shape        # Размеры (строки, столбцы) - кортеж
df.info()       # Сводка: индекс, столбцы, non-null count, типы, память
# df.info(memory_usage='deep') # Более точная оценка памяти
df.describe()   # Статистика для числовых столбцов (count, mean, std, min, max, ...)
# df.describe(include='object') # Статистика для object/string (count, unique, top, freq)
# df.describe(include='all') # Статистика для всех типов
df.columns      # Список названий столбцов
df.index        # Индекс строк
df.dtypes       # Типы данных в каждом столбце
s.value_counts() # Подсчет уникальных значений в Series (столбце)
# s.value_counts(dropna=False) # Включая NaN
df['col_A'].nunique() # Количество уникальных значений в столбце
```

4 Очистка и Предобработка / Cleaning & Preprocessing

Подготовка данных к анализу

Обработка аномалий и приведение данных к нужному формату.

Работа с пропусками (NaN)

```
df.isnull()      # Boolean DataFrame: True где NaN
df.isnull().sum() # Количество NaN в каждом столбце
df.notnull()     # Boolean DataFrame: True где НЕ NaN
df.dropna()      # Удалить строки с любым NaN
# df.dropna(axis=1) # Удалить столбцы с любым NaN
# df.dropna(subset=['col_A']) # Удалить строки с NaN только в 'col_A'
df.fillna(0)     # Заменить все NaN на 0
# df['col_A'].fillna(df['col_A'].mean()) # Замена средним по столбцу
```

Работа с дубликатами

```
df.duplicated() # Boolean Series: True для дублирующихся строк (кроме первого вхождения)
# df.duplicated(subset=['col_A', 'col_B']) # Проверка дубликатов по подмножеству столбцов
df.drop_duplicates() # Удалить дублирующиеся строки
# df.drop_duplicates(keep='last') # Оставить последнее вхождение
```

Изменение типов и переименование

```
# Изменение типа столбца
# df['col_A'].astype(float)
# df['date_col'] = pd.to_datetime(df['date_col'], format='%Y-%m-%d')
# df['category_col'].astype('category') # Экономия памяти для категориальных

# Переименование столбцов/индекса
# df.rename(columns={'old_name': 'new_name'}, index={'old_idx': 'new_idx'})
```

5 Выборка и Фильтрация / Selection & Filtering

Доступ к данным

Индексация:
`[]` (столбцы), `.loc[]` (метки), `.iloc[]` (позиции).

Выбор столбцов и подмножеств

```
df['col_A']      # Выбор одного столбца (Series)
df.col_A         # Альтернатива (если имя валидно)
df[['col_A', 'col_B']] # Выбор нескольких столбцов (DataFrame)
```

Выбор строк/значений по МЕТКАМ (.loc)

```
# df.loc[label]         # Строка по метке индекса
# df.loc[start:end]     # Срез строк по меткам (включая end)
# df.loc[:, 'col_A']    # Все строки, столбец 'col_A'
# df.loc[label, 'col_A']# Конкретное значение
```

Выбор строк/значений по ПОЗИЦИЯМ (.iloc)

```
# df.iloc[0]          # Первая строка (позиция 0)
# df.iloc[0:2]        # Первые две строки (срез до 2, НЕ включая 2)
# df.iloc[:, 0]       # Все строки, первый столбец (позиция 0)
# df.iloc[0, 0]       # Значение в [0, 0]
# df.iloc[[0, 2], [0, 1]] # Выбор по спискам позиций строк/столбцов
```

Фильтрация по условию (Boolean Indexing)

Мощный способ выбора строк на основе логических условий.

Примеры Boolean Indexing

```
df[df['col_A'] > 10] # Строки, где значение в col_A > 10
# Условия: & (И), | (ИЛИ), ~ (НЕ). Скобки обязательны!
df[(df['col_A'] > 10) & (df['col_B'] == 'X')]
df[df['col_B'].isin(['X', 'Y'])] # Строки, где col_B равно 'X' или 'Y'
# df[df['col_A'].between(10, 20)] # Значения между 10 и 20 включительно

# Использование .loc с булевым массивом (предпочтительнее для явности)
# df.loc[df['col_A'] > 10, ['col_B', 'col_C']] # Фильтр + выбор столбцов
```

6 Сортировка / Sorting

Сортировка данных

Упорядочивание строк DataFrame. **По значениям** ('sort_values') и **по индексу** ('sort_index').

Примеры сортировки

```
# Сортировка по значениям одного или нескольких столбцов
df.sort_values(by='col_A') # По возрастанию
df.sort_values(by='col_A', ascending=False) # По убыванию
# df.sort_values(by=['col_A', 'col_B']) # По нескольким столбцам

# Сортировка по индексу
df.sort_index(ascending=False)

# Важно: сортировка возвращает НОВЫЙ DataFrame.
# Для изменения на месте: inplace=True
```

7 Трансформация и Применение Функций / Transformation & Applying Functions

Изменение и создание данных

Применение функций, работа со специальными типами, создание новых признаков.

Создание новых столбцов

```
# На основе существующих
# df['new_col'] = df['col_A'] * 2
# df['col_C'] = df['col_A'] + df['col_B'] # Поэлементные операции

# С использованием np.where (аналог IF в SQL/Excel)
# df['flag'] = np.where(df['col_A'] > 10, 'High', 'Low')
```

Применение функций (apply, map, applymap)

```
# apply: применяет функцию к строкам (axis=1) или столбцам (axis=0)
# df.apply(np.sum, axis=0) # Сумма по каждому столбцу (Series)
# df.apply(lambda row: row['col_A'] * row.name.day, axis=1) # Пример сложной функции по строкам

# map: работает поэлементно на Series (для замены или преобразования)
# s.map({'X': 1, 'Y': 2}) # Замена значений по словарю
# s.map('Value: {}'.format) # Применение строковой функции

# applymap: работает поэлементно на DataFrame (применяет функцию к каждому элементу)
# df_numeric.applymap(lambda x: x**2) # Квадрат каждого элемента
# ВАЖНО: Старайтесь использовать векторизованные операции Pandas/NumPy вместо apply/applymap, если это возможно (они быстрее!).
```

Работа со строками (.str)

```
# Доступ к строковым методам через аксессор .str для Series
# s_text = pd.Series(['apple', ' Banana ', 'kiwi '])
# s_text.str.lower()      # -> ['apple', ' banana ', 'kiwi ']
# s_text.str.strip()      # -> ['apple', 'Banana', 'kiwi']
# s_text.str.contains('a')# -> [True, True, False]
# s_text.str.split('a')   # -> [['', 'pple'], [' B', 'n', 'n', ' '], ['kiwi ']]
# s_text.str.replace('a', 'X') # -> ['Xpple', ' BXnana ', 'kiwi ']
# s_text.str.len()        # Длина каждой строки
```

Работа с датами (.dt)

```
# Доступ к компонентам даты/времени через аксессор .dt для Series (типа datetime)
# s_dates = pd.to_datetime(pd.Series(['2024-01-05', '2024-02-10']))
# s_dates.dt.year         # -> [2024, 2024]
# s_dates.dt.month_name() # -> ['January', 'February']
# s_dates.dt.dayofweek    # -> [4, 5] (Понедельник=0, Воскресенье=6)
# s_dates.dt.date         # Только дата (без времени)
# (s_dates - pd.Timedelta(days=1)) # Вычитание временного интервала
```

8 Группировка и Агрегация / Grouping & Aggregation

Split-Apply-Combine (Разделяй-Применяй-Объединяй)

Основа 'groupby': 1. **Split**: Данные делятся на группы по ключу. 2. **Apply**: Функция применяется к каждой группе. 3. **Combine**: Результаты объединяются.

GroupBy: Группировка и агрегация

Расчет сводных статистик по группам.

Примеры GroupBy

```
# Группировка по одному или нескольким столбцам
grouped = df.groupby('key_col')
# grouped = df.groupby(['key1', 'key2']) # Группировка по нескольким ключам

# Применение агрегирующих функций
# grouped.mean() # Среднее для всех числовых столбцов в каждой группе
# grouped['data_col'].sum() # Сумма для конкретного столбца в каждой группе
# grouped.size() # Размер каждой группы (включая NaN в ключах)
# grouped.count() # Количество НЕ-NaN значений в каждой группе/столбце

# Несколько агрегаций сразу с .agg()
# grouped['data_col'].agg(['sum', 'mean', 'std'])
# grouped.agg({
#     'data_col1': ['mean', 'min', 'max'], # Несколько функций к одному столбцу
#     'data_col2': 'nunique',              # Одна функция к другому
#     'data_col3': lambda x: x.max() - x.min() # Пользовательская lambda-функция
# })

# Применение функции к группам без агрегации (.transform)
# Часто используется для заполнения пропусков средним по группе или для нормализации
# df['group_mean'] = df.groupby('key_col')['data_col'].transform('mean')
# df['normalized'] = df['data_col'] / df.groupby('key_col')['data_col'].transform('sum')

# Фильтрация групп (.filter)
# Оставить только те группы, где среднее по data_col > 10
# filtered_groups = df.groupby('key_col').filter(lambda g: g['data_col'].mean() > 10)
```

Сводные таблицы / Pivot Tables

Аналог сводных таблиц в Excel для агрегации и изменения формы данных.

Пример pivot_table

```
# pd.pivot_table(df,
#                values='Value_Column', # Столбец для агрегации
#                index='Row_Index_Column', # Столбец(ы) для индекса строк
#                columns='Column_Index_Column', # Столбец(ы) для названий столбцов
#                aggfunc=np.sum, # Функция агрегации (sum, mean, count, ...)
#                fill_value=0) # Значение для заполнения NaN после агрегации
#
# pd.crosstab(df['col_A'], df['col_B']) # Таблица сопряженности (частот)
```

Часто результатом groupby или pivot_table является DataFrame с MultiIndex!

9 Объединение / Merging & Joining

Объединение DataFrames

Комбинирование данных из разных источников. **merge()**

(SQL JOIN), **join()**

(по индексам), **concat()**

(склеивание).

Примеры объединения

```
# df1, df2 - примеры DataFrame
# merge: аналог SQL JOIN (по столбцам)
# pd.merge(df1, df2, on='key_col', how='inner') # Inner join по ключу
# pd.merge(df1, df2, left_on='key1', right_on='key2', how='left') # Left join по разным ключам
# how: 'inner' (по умолч.), 'outer', 'left', 'right'

# concat: склеивание таблиц по оси (строк или столбцов)
# pd.concat([df1, df2], axis=0) # Склеить строки (ось 0)
# pd.concat([df1, df2], axis=1) # Склеить столбцы (ось 1) - важно совпадение индексов

# join: объединение по индексам (или индекс с ключом)
# df1.join(df2.set_index('key_col'), on='key_col', how='inner')
```

10 Визуализация / Visualization

Быстрая визуализация

Простые графики для первичного анализа с помощью `.plot()`. Использует Matplotlib под капотом.

Пример простого графика

```
# Требуется matplotlib: pip install matplotlib
import matplotlib.pyplot as plt

# df['numeric_col'].hist(bins=30) # Гистограмма
# df.plot(kind='scatter', x='col_A', y='col_B') # Диаграмма рассеяния
# df.groupby('category_col')['value_col'].mean().plot(kind='bar') # Столбчатая диаграмма средних по группам
# plt.show() # Отобразить график (часто не нужно в Jupyter)
# plt.savefig('img/my_plot.png') # Сохранить график
# plt.close()
```

11 Полезные ссылки и советы / Links & Tips

Полезные ресурсы

`Pandas Documentation`
Официальная документация[a]

`Stack Overflow [pandas]`
Вопросы и ответы сообщества

`Pandas GitHub`
Исходный код библиотеки

[a] Pandas Development Team 2024.

Цитата / Quote

> "There should be one– and preferably only one –obvious way to do it."
> – *The Zen of Python (import this)*

Советы по производительности и Best Practices

- **Векторизация > Итерация:** Используйте встроенные функции Pandas/NumPy вместо циклов `for` или `.iterrows()`. Векторизованные операции работают на порядки быстрее.
- **Тип `'category'`:** Для столбцов с небольшим количеством уникальных строковых значений используйте `df['col'].astype('category')`. Это значительно экономит память и ускоряет операции (особенно `groupby`).
- **`.loc` vs `[]` для присваивания:** При изменении данных используйте `.loc` для избежания `SettingWithCopyWarning`. Например: `df.loc[mask, 'col'] = value`.
- **Работа с MultiIndex:** `groupby` и `pivot_table` часто возвращают MultiIndex. Изучите методы работы с ним (`.reset_index()`, `.unstack()`, `.stack()`, выборка через кортежи).
- **Оценка памяти:** Используйте `df.info(memory_usage='deep')` для более точной оценки занимаемой памяти DataFrame, особенно при наличии строковых данных.

References

[1] Pandas Development Team. *pandas documentation*. `https://pandas.pydata.org/docs/`. Accessed: 2025-03-30. 2024.