

Шпаргалка по Pandas, Numpy, Matplotlib / Полезные методы Cheatsheet (XeLaTeX)

Краткий справочник по основным операциям
April 2, 2025

Contents

- 1 NumPy: Математика и Массивы (np) 1
- 2 Pandas: Таблицы Данных для Анализа (pd) 1
- 3 Чтение и Запись Файлов (Базовый Python) 3
- 4 Визуализация: Matplotlib и Seaborn 3

1 NumPy: Математика и Массивы (np)

Зачем NumPy?

Это фундамент для числовых вычислений в Python. Представь его как **супер-оптимизированную таблицу Excel** для работы с числами, особенно с большими наборами данных. Все операции выполняются очень быстро. Основной объект - **ndarray** (n-мерный массив).

Создание Массивов

Основные способы "завести" себе массив.

Создание NumPy массивов

```
1 import numpy as np
2
3 # Из Python списка
4 arr1 = np.array([1, 2, 3, 4, 5])
5
6 # Массив нулей (форма: 2 строки, 3 столбца)
7 zeros = np.zeros((2, 3))
8
9 # Массив единиц
10 ones = np.ones((3, 2))
11
12 # Последовательность чисел (как range)
13 seq1 = np.arange(0, 10, 2) # Старт, стоп (не вкл.), шаг
14
15 # Заданное количество чисел в интервале
16 seq2 = np.linspace(0, 1, 5) # Старт, стоп (вкл.), кол-во
```

Базовые Операции

NumPy позволяет делать математику сразу со всем массивом.

Операции с массивами

```
1 a = np.array([1, 2, 3])
2 b = np.array([4, 5, 6])
3
4 # Поэлементные операции
5 c = a + b # -> array([5, 7, 9])
6 d = a * 2 # -> array([2, 4, 6])
7 e = a ** 2 # -> array([1, 4, 9])
8
9 # Математические функции
10 f = np.sin(a)
11 g = np.exp(a)
12
13 # Матричное умножение (для 1D - скалярное)
14 dot_product = np.dot(a, b) # 1*4 + 2*5 + 3*6 = 32
15
16 # Сравнения
17 bool_arr = a > 1 # -> array([False, True, True])
```

Аналогия с **широковещанием (broadcasting)**: NumPy "растягивает" массивы меньшей размерности (например, число '2' в 'a * 2'), чтобы их формы совпали для операции. Как если бы ты красил стену валиком (операция), а краска (число) сама распределялась по всей ширине.

Форма и Размер (shape, reshape)

Важно понимать "габариты" массива.

Атрибуты и методы формы

```
1 data = np.array([[1, 2, 3], [4, 5, 6]])
2
3 # Форма (кортеж: строки, столбцы, ...)
4 print(data.shape) # -> (2, 3)
5
6 # Количество измерений
7 print(data.ndim) # -> 2
8
9 # Общее количество элементов
10 print(data.size) # -> 6
11
12 # Изменение формы (кол-во элементов должно совпадать!)
13 reshaped = data.reshape((3, 2))
14 # [[1, 2],
15 #  [3, 4],
16 #  [5, 6]]
17
18 # "Вытягивание" в 1D массив
19 flattened = data.flatten() # или data.reshape(-1)
20 # [1, 2, 3, 4, 5, 6]
```

Важно: shape - это "чертеж" массива (например, 2 строки на 3 столбца), а size - общее число "кирпичиков" (элементов).

Индексирование и Срезы

Доступ к элементам массива.

Доступ к элементам NumPy

```
1 arr = np.array([[10, 20, 30], [40, 50, 60]])
2
3 # Первый элемент (первая строка, первый столбец)
4 print(arr[0, 0]) # -> 10
5 # или
6 print(arr[0][0]) # -> 10
7
8 # Вся первая строка
9 print(arr[0, :]) # -> array([10, 20, 30])
10 # или
11 print(arr[0]) # -> array([10, 20, 30])
12
13 # Весь второй столбец
14 print(arr[:, 1]) # -> array([20, 50])
15
16 # Срез: первые две колонки первой строки
17 print(arr[0, 0:2]) # -> array([10, 20])
18
19 # Boolean индексирование
20 print(arr[arr > 30]) # -> array([40, 50, 60])
```

2 Pandas: Таблицы Данных для Анализа (pd)

Зачем Pandas?

Твой основной инструмент для работы с **табличными данными** (как в Excel или SQL). Строит свою работу поверх NumPy. Два главных объекта: **Series** (один столбец) и **DataFrame** (таблица, коллекция Series). Аналогия: Если NumPy - это супер-массив чисел, то Pandas - это **умная электронная таблица**, которую можно программировать.

Чтение Данных (CSV)

Самый частый способ загрузить данные.

Чтение CSV

```
1 import pandas as pd
2
3 # Предположим, есть файл 'your_data.csv'
4 # df = pd.read_csv('your_data.csv')
5
6 # Частые параметры:
7 # sep=';' - если разделитель точка с запятой
8 # header=None - если в файле нет заголовков
9 # names=['col1', 'col2'] - задать имена столбцов
10 # usecols=['col1', 'col3'] - прочитать только
    нужные
11
12 # Создадим пример DataFrame для демонстрации
13 df_data = {'col_A': [1, 2, 3, 4], 'col_B': ['x',
    'y', 'x', 'z'], 'col_C': [10, 20, 30, 40]}
14 df = pd.DataFrame(df_data) # Используем этот df
    далее
15
16 # Посмотреть первые/последние строки
17 print(df.head()) # Первые 5 строк
18 print(df.tail(3)) # Последние 3 строки
19
20 # Информация о DataFrame (типы, пропуски)
21 df.info()
22
23 # Базовые статистики для числовых столбцов
24 print(df.describe())
```

DataFrame и Series

Основные структуры данных Pandas.

Создание и доступ

```
1 # Создание Series (один столбец)
2 s = pd.Series([10, 20, 30], index=['a', 'b', 'c'],
    name='MySeries')
3
4 # Создание DataFrame (таблица) из словаря
5 data = {'col_A': [1, 2, 3], 'col_B': ['x', 'y',
    'z']}
6 df_example = pd.DataFrame(data)
7
8 # Доступ к столбцу (возвращает Series)
9 col_a_series = df_example['col_A']
10 # или (если имя без пробелов/специфических символов)
11 col_b_series = df_example.col_B
12
13 # Доступ к нескольким столбцам (возвращает
    DataFrame)
14 subset_df = df_example[['col_A', 'col_B']]
```

Выборка: .loc vs .iloc (Важно!)

Частая путаница у новичков! Это два основных способа выбрать строки/столбцы.

- .loc[]: Выбирает по **МЕТКАМ** (именам) индекса и столбцов. **Включает** правую границу среза.
- .iloc[]: Выбирает по **ЦЕЛОЧИСЛЕННЫМ ПОЗИЦИЯМ** (номерам, начиная с 0). **НЕ включает** правую границу среза (как в Python).

Аналогия: Представь библиотеку. .loc - это поиск книги по названию и автору

(метки). .iloc - это взять "пятую книгу с третьей полки" (позиции).

Примеры .loc и .iloc (используем df из блока "Чтение")

```
1 # df имеет стандартный числовой индекс [0, 1, 2,
    3]
2
3 # --- .loc (по МЕТКАМ индекса и столбцов) ---
4 # Строка по метке индекса (здесь метка=число)
5 print(df.loc[0])
6 # Срез строк по меткам (включительно!)
7 print(df.loc[0:2]) # Строки с индексами 0, 1, 2
8 # Строки и столбцы по меткам
9 print(df.loc[[0, 3], ['col_A', 'col_C']])
10 # Конкретная ячейка
11 print(df.loc[1, 'col_B'])
12
13 # --- .iloc (по ПОЗИЦИЯМ) ---
14 # Строка по номеру (первая)
15 print(df.iloc[0])
16 # Срез строк по номерам (НЕ включительно!)
17 print(df.iloc[0:2]) # Строки 0 и 1 (т.е. с
    индексами 0 и 1)
18 # Строки и столбцы по номерам
19 print(df.iloc[[0, 3], [0, 2]]) # 1я и 4я строки,
    1й и 3й столбцы
20 # Конкретная ячейка
21 print(df.iloc[1, 1]) # Элемент на пересечении 2й
    строки, 2го столбца
```

Фильтрация Данных

Отбор строк по условиям (используем 'df' из блока "Чтение").

Способы фильтрации

```
1 # 1. Boolean Indexing (основной способ)
2 # Условие: выбрать строки, где значение в 'col_A'
    > 2
3 filter1 = df[df['col_A'] > 2]
4 print("Filter 1:\n", filter1)
5
6 # Несколько условий: & (И), | (ИЛИ), ~ (НЕ)
7 # Обязательно скобки вокруг каждого условия!
8 filter2 = df[(df['col_A'] > 1) & (df['col_B'] ==
    'x')]
9 print("Filter 2:\n", filter2)
10
11 # Использование .isin()
12 filter3 = df[df['col_B'].isin(['x', 'y'])]
13 print("Filter 3:\n", filter3)
14
15 # 2. Метод .query() (удобно для сложных условий)
16 # Строка запроса похожа на SQL WHERE
17 filter4 = df.query('col_A > 2 and col_B == "z"')
18 print("Filter 4:\n", filter4)
19 # Можно использовать переменные с @
20 threshold = 15
21 filter5 = df.query('col_C > @threshold')
22 print("Filter 5:\n", filter5)
```

Группировка и Агрегация (groupby() .agg())

Разделяй (по группам), Властвуй (применяй функцию), Объединяй (результаты). Аналогия: Разложить все фрукты по корзинам (группировка по типу фрукта), затем посчитать вес яблок, средний размер апельсинов и т.д. (агрегация), и записать результаты для каждой корзины.

Пример GroupBy (используем df из блока "Чтение")

```
1 # Сгруппировать по 'col_B' (категориальный
    столбец),
2 # посчитать сумму 'col_C' (числовой)
3 # и среднее/количество для 'col_A' (числовой)
4 agg_results = df.groupby('col_B').agg(
5     total_C=('col_C', 'sum'), # Новый
        столбец = ('старый', 'функция')
6     average_A=('col_A', 'mean'),
7     count_A=('col_A', 'count')
8 )
9 print("Aggregation Results:\n", agg_results)
10 # agg_results будет DataFrame с 'col_B' в индексе
11
12 # Можно группировать по нескольким столбцам
13 # multi_group = df.groupby(['cat1',
    'cat2']).size() # размер групп
```

Объединение Таблиц (merge, join, concat)

Склеивание данных из разных источников.

- 'pd.merge(df1, df2, on='key, how='...)': Похоже на SQL JOIN. Объединяет по общим столбцам (on='key') или индексам. 'how' определяет тип: "inner" (только общие ключи), "outer" (все ключи), "left", "right".
- 'pd.concat([df1, df2], axis=...)': Простое **склеивание** таблиц. 'axis=0' (по умолчанию) - добавить строки df2 под df1. 'axis=1' - добавить столбцы df2 справа от df1 (требует совпадения индексов или аккуратности).
- 'df1.join(df2)': Удобный метод для объединения по **индексу** (похож на 'merge' с 'left_index=True', 'right_index=True').

Аналогия: 'merge' - как найти общих друзей (ключи) в двух списках контактов. 'concat' - как приклеить один список контактов под другим (axis=0) или положить их рядом (axis=1).

Примеры Merge и Concat

```
1 # Создадим два DataFrame для примеров
2 df1 = pd.DataFrame({'user_id': ['u1', 'u2', 'u3'],
    'value1': [10, 20, 30]})
3 df2 = pd.DataFrame({'user_id': ['u2', 'u3', 'u4'],
    'value2': [55, 66, 77]})
4 df3 = pd.DataFrame({'user_id': ['u5', 'u6'],
    'value1': [40, 50]}) # Для concat
5
6 # Merge (Inner Join по 'user_id')
7 df_merged = pd.merge(df1, df2, on='user_id',
    how='inner')
8 print("Merged DF:\n", df_merged)
9
10 # Concat (склеить строки df1 и df3)
11 df_concatenated = pd.concat([df1, df3], axis=0,
    ignore_index=True)
12 print("Concatenated DF (rows):\n",
    df_concatenated)
13 # ignore_index=True сбрасывает исходные индексы
```

Применение Функций (.apply())

Для сложных операций, которые нельзя сделать стандартными методами (используем 'df' из блока "Чтение").

Примеры .apply()

```
1 # Применить функцию к каждому элементу столбца (Series)
2 df['C_percent'] = df['col_C'].apply(lambda x: x /
3 df['col_C'].sum() * 100)
4 print("DF with C_percent:\n", df)
5 # Применить функцию к каждой строке (axis=1)
6 def custom_logic(row):
7     # row - это Series, представляющий строку
8     return row['col_A'] * 10 + row['col_C'] if
9     row['col_B'] == 'x' else row['col_C']
10 df['result'] = df.apply(custom_logic, axis=1)
11 print("DF with result:\n", df)
```

Осторожно: 'apply()' может быть медленным на больших данных. По возможности используйте встроенные векторизованные операции NumPy/Pandas.

3 Чтение и Запись Файлов (Базовый Python)

Зачем?

Хотя Pandas отлично работает с CSV/Excel, иногда нужно работать с обычными текстовыми файлами (логи, конфиги, .txt). Важно делать это безопасно. *Аналогия:* Файл - это коробка. Открытие/закрытие - это работа с крышкой. Конструкция 'with open(...)' гарантирует, что ты **всегда закроешь коробку**, даже если что-то пойдет не так при упаковке/распаковке.

Стандартный способ Python (with open)

Рекомендуемый подход для работы с файлами.

Чтение и запись текстовых файлов

```
1 # Запись в файл (перезапишет, если существует)
2 lines_to_write = ["Строка 1\n", "Строка 2\n"]
3 try: # Добавим try-except для случая, если файл
4     # недоступен
5     with open('my_output.txt', 'w',
6 encoding='utf-8') as f:
7         f.write("Одна строка\n")
8         f.writelines(lines_to_write)
9         print("Файл 'my_output.txt' успешно записан.")
10    except IOError as e:
11        print(f"Ошибка записи файла: {e}")
12
13 # Чтение из файла
14 try:
15     with open('my_output.txt', 'r',
16 encoding='utf-8') as f:
17         print("\nЧитаем файл 'my_output.txt':")
18         # content_str = f.read() # Прочитать
19         # весь файл в строку
20         # content_list = f.readlines() # Прочитать
21         # все строки в список
22         for line in f: # Читать
23             # построчно (эффективно)
24             print(line.strip())
25    except FileNotFoundError:
26        print("Файл 'my_output.txt' не найден.")
27    except IOError as e:
28        print(f"Ошибка чтения файла: {e}")
29
30 # Режимы: 'r' - чтение, 'w' - запись (перезапись),
31 # 'a' - дозапись в конец
32 # encoding='utf-8' - важно для русского языка!
```

Pandas для Структурированных Файлов

Для CSV, Excel, JSON, SQL и др. используйте встроенные функции Pandas (используем 'df' из блока "Применение Функций").

Запись в CSV/Excel с Pandas

```
1 try:
2     # Сохранить DataFrame в CSV без индекса
3     df.to_csv('output_data.csv', index=False,
4 encoding='utf-8')
5     print("\nDataFrame успешно сохранен в
6 'output_data.csv'")
7
8     # Сохранить DataFrame в Excel
9     (раскомментируйте, если нужен Excel)
10    # df.to_excel('output_data.xlsx', index=False,
11    sheet_name='MyData')
12    # print("DataFrame успешно сохранен в
13    'output_data.xlsx'")
14 except Exception as e:
15    print(f"Ошибка сохранения DataFrame: {e}")
```

Зачем?

"Лучше один раз увидеть, чем сто раз услышать". Графики помогают понять данные, найти паттерны, выбросы и представить результаты.

- **Matplotlib ('plt')**: Низкоуровневая библиотека, дает полный контроль. *Аналогия:* Набор инструментов художника (холст, кисти, краски).
- **Seaborn ('sns')**: Высокоуровневая, построена на Matplotlib. Упрощает создание красивых статистических графиков, хорошо интегрируется с Pandas. *Аналогия:* Готовые шаблоны или трафареты для рисования стандартных фигур.

Основы Matplotlib (plt)

Базовые команды для создания простых графиков.

4 Визуализация: Matplotlib и Seaborn

Примеры Matplotlib

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd # Нужен для примера scatter
4
5 # Данные для графиков
6 x_lin = np.linspace(0, 10, 100)
7 y1_lin = np.sin(x_lin)
8 y2_lin = np.cos(x_lin)
9 data_hist = np.random.randn(1000)
10 data_box = [np.random.normal(0, std, 100) for std
11             in range(1, 4)]
12 # DataFrame для scatter
13 df_scatter = pd.DataFrame({'feature1':
14                            np.random.rand(50) * 10,
15                            'feature2':
16                                np.random.rand(50) *
17                                10})
18
19 # --- Создание фигур ---
20 # Линейный график
21 plt.figure(figsize=(8, 4)) # Размер фигуры
22 # (опционально)
23 plt.plot(x_lin, y1_lin, label='sin(x)')
24 plt.plot(x_lin, y2_lin, label='cos(x)',
25          linestyle='--')
26 plt.title('Линейный график (Matplotlib)')
27 plt.xlabel('Ось X')
28 plt.ylabel('Ось Y')
29 plt.legend() # Показать легенду
30 plt.grid(True) # Добавить сетку
31
32 # Диаграмма рассеяния (Scatter plot)
33 plt.figure()
34 plt.scatter(df_scatter['feature1'],
35            df_scatter['feature2'], alpha=0.5) # alpha -
36            прозрачность
37 plt.title('Диаграмма рассеяния (Matplotlib)')
38 plt.xlabel('Feature 1')
39 plt.ylabel('Feature 2')
40
41 # Гистограмма
42 plt.figure()
43 plt.hist(data_hist, bins=30, color='skyblue',
44          edgecolor='black')
45 plt.title('Гистограмма (Matplotlib)')
46 plt.xlabel('Значение')
47 plt.ylabel('Частота')
48
49 # Ящик с усами (Box plot)
50 plt.figure()
51 plt.boxplot(data_box, labels=['Gr1', 'Gr2',
52                               'Gr3'])
53 plt.title('Ящик с усами (Matplotlib)')
54 plt.ylabel('Значение')
55
56 # ВАЖНО: Отображение графиков plt.show() будет в
57 конце секции
```

Seaborn (sns) для Статистики

Более красивые и статистически ориентированные графики, часто одной строкой.

Примеры Seaborn

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt # Часто
3     используется для донастройки sns графиков
4
5 # Загрузка примера данных из seaborn
6 tips = sns.load_dataset("tips") # DataFrame с
7     данными о чаевых
8
9 # --- Создание фигур ---
10 # Линейный график (с доверительным интервалом по
11     умолчанию)
12 plt.figure() # Можно управлять размером через plt
13 sns.lineplot(x="total_bill", y="tip", data=tips)
14 plt.title('Линейный график (Seaborn)')
15
16 # Диаграмма рассеяния (с возможностью раскраски по
17     категории)
18 plt.figure()
19 sns.scatterplot(x="total_bill", y="tip",
20                hue="time", data=tips)
21 plt.title('Диаграмма рассеяния (Seaborn)')
22
23 # Гистограмма (с оценкой плотности KDE)
24 plt.figure()
25 sns.histplot(data=tips, x="total_bill", kde=True)
26 plt.title('Гистограмма (Seaborn)')
27
28 # Ящик с усами (удобно для сравнения по
29     категориям)
30 plt.figure()
31 sns.boxplot(x="day", y="total_bill", data=tips)
32 plt.title('Ящик с усами (Seaborn)')
33
34 # ВАЖНО: Отображение графиков plt.show() будет в
35 конце секции
```

Seaborn часто автоматически подписывает оси и создает легенды, используя имена столбцов DataFrame.

Когда Какой График Использовать?

Краткий гид по выбору типа визуализации:

- **Линейный график (plot/lineplot):** Показать **тренд** или изменение показателя во времени (или по другой непрерывной оси). Сравнение трендов нескольких групп.
- **Диаграмма рассеяния (scatter/scatterplot):** Посмотреть **взаимосвязь** между двумя **числовыми** переменными. Помогает найти корреляции, кластеры, выбросы.
- **Гистограмма (hist/histplot):** Понять **распределение** одной **числовой** переменной. Как часто встречаются те или иные значения? Есть ли пики? Симметрично ли распределение?
- **Ящик с усами (boxplot):** Сравнить распределения **числовой** переменной по нескольким **категориям**. Показывает медиану, квартили, разброс и потенциальные выбросы в каждой группе.

Для отображения всех созданных выше графиков Matplotlib/Seaborn, выполните:

Показать все графики

```
1 # Эта команда должна быть вызвана один раз после всех
2 команд plt.figure()/sns.*plot()
3
4 # В средах типа Jupyter Notebook/Lab графики могут
5 отображаться автоматически.
6
7 # В обычных Python скриптах plt.show() обязателен.
8 plt.show()
```