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

Краткий справочник по основным операциям

Содержание

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

```python
import numpy as np

# Из Python списка
arr1 = np.array([1, 2, 3, 4, 5])

# Массив нулей (форма: 2 строки, 3 столбца)
zeros = np.zeros((2, 3))

# Массив единиц
ones = np.ones((3, 2))

# Последовательность чисел (как range)
seq1 = np.arange(0, 10, 2) # Старт, стоп (не вкл.), шаг

# Заданное количество чисел в интервале
seq2 = np.linspace(0, 1, 5) # Старт, стоп (вкл.), кол-во
```

Базовые Операции

NumPy позволяет делать математику сразу со всем массивом.

Операции с массивами

```python
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Поэлементные операции
c = a + b  # -> array([5, 7, 9])
d = a * 2  # -> array([2, 4, 6])
e = a ** 2 # -> array([1, 4, 9])

# Математические функции
f = np.sin(a)
g = np.exp(a)

# Матричное умножение (для 1D - скалярное)
dot_product = np.dot(a, b) # 1*4 + 2*5 + 3*6 = 32

# Сравнения
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

```python
data = np.array([[1, 2, 3], [4, 5, 6]])

# Форма (кортеж: строки, столбцы, ...)
print(data.shape)  # -> (2, 3)

# Количество измерений
print(data.ndim)   # -> 2

# Общее количество элементов
print(data.size)   # -> 6

# Изменение формы (кол-во элементов должно совпадать!)
reshaped = data.reshape((3, 2))
# [[1, 2],
#  [3, 4],
#  [5, 6]]

# "Вытягивание" в 1D массив
flattened = data.flatten() # или data.reshape(-1)
# [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

```
arr = np.array([[10, 20, 30], [40, 50, 60]])

# Первый элемент (первая строка, первый столбец)
print(arr[0, 0]) # -> 10
# или
print(arr[0][0]) # -> 10

# Вся первая строка
print(arr[0, :]) # -> array([10, 20, 30])
# или
print(arr[0])    # -> array([10, 20, 30])

# Весь второй столбец
print(arr[:, 1]) # -> array([20, 50])

# Срез: первые две колонки первой строки
print(arr[0, 0:2]) # -> array([10, 20])

# Boolean индексирование
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 супермассив чисел, то Pandas - это **умная электронная таблица**, которую можно программировать.

Чтение Данных (CSV)

Самый частый способ загрузить данные.

Чтение CSV

```
import pandas as pd

# Предположим, есть файл 'your_data.csv'
# df = pd.read_csv('your_data.csv')

# Частые параметры:
# sep=';' - если разделитель точка с запятой
# header=None - если в файле нет заголовков
# names=['col1', 'col2'] - задать имена столбцов
# usecols=['col1', 'col3'] - прочитать только нужные

# Создадим пример DataFrame для демонстрации
df_data = {'col_A': [1, 2, 3, 4], 'col_B': ['x', 'y', 'x', 'z'], 'col_C': [10, 20, 30, 40]}
df = pd.DataFrame(df_data) # Используем этот df далее

# Посмотреть первые/последние строки
print(df.head())    # Первые 5 строк
print(df.tail(3))   # Последние 3 строки

# Информация о DataFrame (типы, пропуски)
df.info()

# Базовые статистики для числовых столбцов
print(df.describe())
```

DataFrame и Series

Основные структуры данных Pandas.

Создание и доступ

```
# Создание Series (один столбец)
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'], name='MySeries')

# Создание DataFrame (таблица) из словаря
data = {'col_A': [1, 2, 3], 'col_B': ['x', 'y', 'z']}
df_example = pd.DataFrame(data)

# Доступ к столбцу (возвращает Series)
col_a_series = df_example['col_A']
# или (если имя без пробелов/спецсимволов)
col_b_series = df_example.col_B

# Доступ к нескольким столбцам (возвращает DataFrame)
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")

```
# df имеет стандартный числовой индекс [0, 1, 2, 3]

# --- .loc (по МЕТКАМ индекса и столбцов) ---
# Строка по метке индекса (здесь метка=число)
print(df.loc[0])
# Срез строк по меткам (включительно!)
print(df.loc[0:2]) # Строки с индексами 0, 1, 2
# Строки и столбцы по меткам
print(df.loc[[0, 3], ['col_A', 'col_C']])
# Конкретная ячейка
print(df.loc[1, 'col_B'])

# --- .iloc (по ПОЗИЦИЯМ) ---
# Строка по номеру (первая)
print(df.iloc[0])
# Срез строк по номерам (НЕ включительно!)
print(df.iloc[0:2]) # Строки 0 и 1 (т.е. с индексами 0 и 1)
# Строки и столбцы по номерам
print(df.iloc[[0, 3], [0, 2]]) # 1я и 4я строки, 1й и 3й столбцы
# Конкретная ячейка
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

```python
# 1. Boolean Indexing (основной способ)
# Условие: выбрать строки, где значение в 'col_A' > 2
filter1 = df[df['col_A'] > 2]
print("Filter 1:\n", filter1)

# Несколько условий: & (И), | (ИЛИ), ~ (НЕ)
# Обязательно скобки вокруг каждого условия!
filter2 = df[(df['col_A'] > 1) & (df['col_B'] == 'x')]
print("Filter 2:\n", filter2)

# Использование .isin()
filter3 = df[df['col_B'].isin(['x', 'y'])]
print("Filter 3:\n", filter3)

# 2. Метод .query() (удобно для сложных условий)
# Строка запроса похожа на SQL WHERE
filter4 = df.query('col_A > 2 and col_B == "z"')
print("Filter 4:\n", filter4)
# Можно использовать переменные с @
threshold = 15
filter5 = df.query('col_C > @threshold')
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”)

```python
# Сгруппировать по 'col_B' (категориальный столбец),
# посчитать сумму 'col_C' (числовой)
# и среднее/количество для 'col_A' (числовой)
agg_results = df.groupby('col_B').agg(
    total_C=('col_C', 'sum'),       # Новый столбец = ('старый', 'функция')
    average_A=('col_A', 'mean'),
    count_A=('col_A', 'count')
)
print("Aggregation Results:\n", agg_results)
# agg_results будет DataFrame с 'col_B' в индексе

# Можно группировать по нескольким столбцам
# 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

```python
# Создадим два DataFrame для примеров
df1 = pd.DataFrame({'user_id': ['u1', 'u2', 'u3'], 'value1': [10, 20, 30]})
df2 = pd.DataFrame({'user_id': ['u2', 'u3', 'u4'], 'value2': [55, 66, 77]})
df3 = pd.DataFrame({'user_id': ['u5', 'u6'], 'value1': [40, 50]}) # Для concat

# Merge (Inner Join по 'user_id')
df_merged = pd.merge(df1, df2, on='user_id', how='inner')
print("Merged DF:\n", df_merged)

# Concat (склеить строки df1 и df3)
df_concatenated = pd.concat([df1, df3], axis=0, ignore_index=True)
print("Concatenated DF (rows):\n", df_concatenated)
# ignore_index=True сбрасывает исходные индексы
```

Применение Функций (`.apply()`)

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

Примеры .apply()

```python
# Применить функцию к каждому элементу столбца (Series)
df['C_percent'] = df['col_C'].apply(lambda x: x / df['col_C'].sum() * 100)
print("DF with C_percent:\n", df)

# Применить функцию к каждой строке (axis=1)
def custom_logic(row):
    # row - это Series, представляющий строку
    return row['col_A'] * 10 + row['col_C'] if row['col_B'] == 'x' else row['col_C']

df['result'] = df.apply(custom_logic, axis=1)
print("DF with result:\n", df)
```

Осторожно: ‘.apply()‘ может быть медленным на больших данных. По возможности используй 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

```python
# Запись в файл (перезапишет, если существует)
lines_to_write = ["Строка 1\n", "Строка 2\n"]
try: # Добавим try-except для случая, если файл недоступен
    with open('my_output.txt', 'w', encoding='utf-8') as f:
        f.write("Одна строка\n")
        f.writelines(lines_to_write)
    print("Файл 'my_output.txt' успешно записан.")
except IOError as e:
    print(f"Ошибка записи файла: {e}")

# Чтение из файла
try:
    with open('my_output.txt', 'r', encoding='utf-8') as f:
        print("\nЧитаем файл 'my_output.txt':")
        # content_str = f.read()      # Прочитать весь файл в строку
        # content_list = f.readlines() # Прочитать все строки в список
        for line in f:              # Читать построчно (эффективно)
            print(line.strip())
except FileNotFoundError:
    print("Файл 'my_output.txt' не найден.")
except IOError as e:
     print(f"Ошибка чтения файла: {e}")


# Режимы: 'r' - чтение, 'w' - запись (перезапись), 'a' - дозапись в конец
# encoding='utf-8' - важно для русского языка!
```

Pandas для Структурированных Файлов

Для CSV, Excel, JSON, SQL и др. используй встроенные функции Pandas (используем ‘df‘ из блока ”Применение Функций”).

Запись в CSV/Excel с Pandas

```python
try:
    # Сохранить DataFrame в CSV без индекса
    df.to_csv('output_data.csv', index=False, encoding='utf-8')
    print("\nDataFrame успешно сохранен в 'output_data.csv'")

    # Сохранить DataFrame в Excel (раскомментируйте, если нужен Excel)
    # df.to_excel('output_data.xlsx', index=False, sheet_name='MyData')
    # print("DataFrame успешно сохранен в 'output_data.xlsx'")
except Exception as e:
    print(f"Ошибка сохранения DataFrame: {e}")
```

4 Визуализация: Matplotlib и Seaborn

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.

Примеры Matplotlib

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd # Нужен для примера scatter

# Данные для графиков
x_lin = np.linspace(0, 10, 100)
y1_lin = np.sin(x_lin)
y2_lin = np.cos(x_lin)
data_hist = np.random.randn(1000)
data_box = [np.random.normal(0, std, 100) for std in range(1, 4)]
# DataFrame для scatter
df_scatter = pd.DataFrame({'feature1': np.random.rand(50) * 10,
                           'feature2': np.random.rand(50) * 10})

# --- Создание фигур ---
# Линейный график
plt.figure(figsize=(8, 4)) # Размер фигуры (опционально)
plt.plot(x_lin, y1_lin, label='sin(x)')
plt.plot(x_lin, y2_lin, label='cos(x)', linestyle='--')
plt.title('Линейный график (Matplotlib)')
plt.xlabel('Ось X')
plt.ylabel('Ось Y')
plt.legend() # Показать легенду
plt.grid(True) # Добавить сетку

# Диаграмма рассеяния (Scatter plot)
plt.figure()
plt.scatter(df_scatter['feature1'], df_scatter['feature2'], alpha=0.5) # alpha - прозрачность
plt.title('Диаграмма рассеяния (Matplotlib)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Гистограмма
plt.figure()
plt.hist(data_hist, bins=30, color='skyblue', edgecolor='black')
plt.title('Гистограмма (Matplotlib)')
plt.xlabel('Значение')
plt.ylabel('Частота')

# Ящик с усами (Box plot)
plt.figure()
plt.boxplot(data_box, labels=['Gr1', 'Gr2', 'Gr3'])
plt.title('Ящик с усами (Matplotlib)')
plt.ylabel('Значение')

# ВАЖНО: Отображение графиков plt.show() будет в конце секции
```

Seaborn (`sns`) для Статистики

Более красивые и статистически ориентированные графики, часто одной строкой.

Примеры Seaborn

```python
import seaborn as sns
import matplotlib.pyplot as plt # Часто используется для донастройки sns графиков

# Загрузка примера данных из seaborn
tips = sns.load_dataset("tips") # DataFrame с данными о чаевых

# --- Создание фигур ---
# Линейный график (с доверительным интервалом по умолчанию)
plt.figure() # Можно управлять размером через plt
sns.lineplot(x="total_bill", y="tip", data=tips)
plt.title('Линейный график (Seaborn)')

# Диаграмма рассеяния (с возможностью раскраски по категории)
plt.figure()
sns.scatterplot(x="total_bill", y="tip", hue="time", data=tips)
plt.title('Диаграмма рассеяния (Seaborn)')

# Гистограмма (с оценкой плотности KDE)
plt.figure()
sns.histplot(data=tips, x="total_bill", kde=True)
plt.title('Гистограмма (Seaborn)')

# Ящик с усами (удобно для сравнения по категориям)
plt.figure()
sns.boxplot(x="day", y="total_bill", data=tips)
plt.title('Ящик с усами (Seaborn)')

# ВАЖНО: Отображение графиков plt.show() будет в 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

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