Pandas

Содержание

1 Pandas: основы (часть 1)

1.1 Структуры данных: `Series` и `DataFrame`

`Series` (1D). `Series` — одномерный массив значений с *индексом* (метками строк). Держит:

- **values**: вектор значений произвольного типа (числа, строки, даты, объекты);
- **index**: метки элементов (по умолчанию `0..N-1`);
- **name**: имя серии (опц.).

Конструктор `Series(data, index=None, dtype=None, name=None, copy=False)`.

- `data` — список/массив/`dict`/скаляр;
- `index` — явные метки; если `data`=`dict`, неуказанные ключи будут `NaN`;
- `dtype` — целевой тип;
- `name` — имя серии;
- `copy` — принудительно копировать `data`.

Примеры создания `Series`.

```
import pandas as pd
import numpy as np

s1 = pd.Series([10, 20, 30], index=["a", "b", "c"], name="score")
s2 = pd.Series({"NY": 1.2, "LA": 0.9, "SF": 1.7}, name="weights")
arr = np.array([3.14, 2.71, 1.61], dtype=np.float64)
s3 = pd.Series(arr, name="phi")  # из питру массива
```

Обращение к элементам `Series`.

```
s = pd.Series([10, 20, 30], index=["a", "b", "c"])
s["a"]         # по метке -> 10
s.loc["b"]     # по метке -> 20
s.iloc[2]      # по позиции -> 30
s[0:2]         # срез по позиции -> a,b
s[s > 15]      # булева фильтрация -> b,c
```

`DataFrame` (2D). `DataFrame` — таблица ($N \times d$) с **строковым индексом** и **именами столбцов**. Столбцы — это `Series`, собранные по общему индексу.

Конструктор `DataFrame(data, index=None, columns=None, dtype=None, copy=False)`.

- `data` — `dict` столбцов, список `dict`, 2D `ndarray`/список списков, другой `DataFrame`;

- `index` — метки строк;
- `columns` — имена столбцов;
- `dtype` — общий тип для всех столбцов (если совместимо);
- `copy` — явная копия данных.

Создание `DataFrame` из словаря списков.

```
df = pd.DataFrame({
    "city": ["NY", "LA", "NY", "SF"],
    "age":  [ 25,   33,  41,  29],
    "pay":  [3.2,  4.1,  5.5, 3.9]
}, index=["u1","u2","u3","u4"])
df.index    # Index(['u1','u2','u3','u4'], dtype='object')
df.columns  # Index(['city','age','pay'], dtype='object')
```

Список словарей → `DataFrame`.

```
rows = [{"city":"NY","age":25}, {"city":"LA","age":33}]
df2  = pd.DataFrame(rows)
```

2D-массив → `DataFrame`.

```
mat = np.array([[1,2,3],[4,5,6]], dtype=np.int64)
df3 = pd.DataFrame(mat, columns=["A","B","C"])
```

`info()` и быстрый обзор.

```
df.info()      # типы столбцов, непустые значения, память
df.head(3)     # первые 3 строки
df.describe()  # числовая сводка
```

1.2 Доступ к данным: основные приёмы

Часто используемые способы.

- `df[col]` — выбрать столбец (или список столбцов);
- `df.loc[row_labels, col_labels]` — *по меткам* индекса/столбцов;
- `df.iloc[row_pos, col_pos]` — *по позициям* (целочисленные индексы);
- `df[0:4]` — срез строк по позициям $[0, 4)$;
- `df[bool_vector]` — фильтрация по булевой маске.

Примеры выборок.

```
df["age"]                  # Series
df[["city","pay"]]         # DataFrame с 2 столбцами

df.loc["u2", "pay"]        # доступ по меткам: одна ячейка
df.loc[["u1","u3"], ["city","age"]]   # подтаблица

df.iloc[0, 2]              # по позициям: 1-я строка, 3-й столбец
df.iloc[0:2, 1:3]          # срезы по позициям

df[ df["age"] >= 30 ]      # булева фильтрация
```

`loc` **vs** `iloc`. `loc` работает *по меткам* индекса/столбцов, поддерживает списки меток, срезы *включительно* по конечной метке, булевы маски. `iloc` — *по позициям* (целые числа), срезы *не включают* правую границу, булевы маски такой же длины, как число строк.

1.3 Сэмплирование строк/столбцов: `sample()`

Примеры `sample()`.

```
df.sample(n=3, random_state=7)             # 3 случайные строки
df.sample(frac=0.30, random_state=7)       # 30% строк
w = np.array([0.1, 0.2, 0.3, 0.4])         # веса строк
df.sample(n=3, weights=w, replace=True, random_state=7)

# выбор случайных столбцов:
df.sample(n=2, axis=1, random_state=7)
```

1.4 Индексация логическими выражениями

С `Series`.

```
s = pd.Series([10, 25, 40, 15], index=list("abcd"))
s[s > 30]                     # элементы > 30
s[(s > 15) & (s % 2 == 1)]    # и то, и другое
```

С `DataFrame`.

```
df[ df["age"] > 30 ]
df[ (df["age"] > 30) & (df["city"].isin(["NY","SF"])) ]
```

Логика через `map/apply/lambda`. `map` применяет функцию к `Series` поэлементно, `apply` — поэлементно к `Series` или постолбцово/построчно к `DataFrame` (в зависимости от `axis`).

```
# признаки "прибрежных" городов (пример)
coastal = {"NY": True, "LA": True, "SF": True}
mask = df["city"].map(coastal).fillna(False)
df_coast = df[mask]

# сложное условие с lambda
mask2 = df["pay"].apply(lambda v: (v >= 3.5) and (v <= 5.0))
df_mid = df[mask2]
```

`isin` — проверка на вхождение. Возвращает булеву маску принадлежности значений столбца множеству.

```
df[ df["city"].isin(["NY","SF"]) ]
df[ ~df["city"].isin({"LA"}) ]  # отрицание через ~
```

1.5 Типы данных, `info()`, `dtypes`, расширенные типы

`df.info()` и `df.dtypes`. `info()` показывает непустые значения, типы столбцов, память; `dtypes` возвращает `Series` с типами. Частые типы: `int64/int32`, `float64`, `bool`, `object` (строки/объекты), `datetime64[ns]`, `timedelta64[ns]`, `category`. Расширенные типы (*extension dtypes*): `Int64`, `boolean`, `string`, `Float64`, `pd.CategoricalDtype` — поддерживают `NA` на уровне столбца.

`astype(self, dtype, copy=True, errors="raise")`.

- `dtype` — целевой тип (`"float64"`, `"Int64"`, `"string"`, `"category"`, маппинг {`col: dtype`});
- `copy` — при `True` возвращает копию (если возможно);
- `errors` — `"raise"` (по умолчанию) или `"ignore"`.

Примеры:

```
df["age"] = df["age"].astype("Int64")            # целые с поддержкой NA
df = df.astype({"pay":"float64", "city":"string"})
```

1.6 Загрузка и приведение типов: `read_csv`, парсеры и конвертеры

Чтение CSV + предобработка столбца «Температура».

```python
import pandas as pd

df = pd.read_csv("sensors.csv", sep=",", encoding="utf-8")
temp_converter = lambda x: x.replace("C", "").strip() if isinstance(x, str)
    else x
df["Температура"] = (df["Температура"]
                     .apply(temp_converter)
                     .pipe(pd.to_numeric, errors="coerce")
                     .astype("float64"))
```

Как это работает: `apply` поэлементно убирает символы и пробелы; `to_numeric` переводит в число (невалидные → `NaN` при `errors="coerce"`); `astype("float64")` задаёт финальный тип. (Корректный тип — `float64`, а не «`float63`».)

Парсеры чисел/дат/интервалов.

- `pd.to_numeric(x, errors="raise"/"coerce"/"ignore")` — безопасное приведение к числам;
- `pd.to_datetime(x, errors="coerce utc=False, dayfirst=False, format=None)` — парсинг дат;
- `pd.to_timedelta(x, errors="coerce unit=None)` — парсинг длительностей (`"1 days 02:00:00"`, `"120s"` и т.п.).

Пример:

```python
df["when"] = pd.to_datetime(df["when"], errors="coerce", utc=True)
df["dur"]  = pd.to_timedelta(df["duration_sec"], unit="s")
df["num"]  = pd.to_numeric(df["maybe_number"], errors="coerce")
```

1.7 Выбор столбцов по типам: `select_dtypes`

Сигнатура и параметры. `DataFrame.select_dtypes(include=None, exclude=None)` — выбирает столбцы с указанными типами.

- `include` — тип или список типов (`"number"`, `"datetime"`, `"category"`, `np.number`, `"string"`, `"boolean"`, ...);
- `exclude` — тип(ы), которые нужно исключить.

Примеры:

```python
num_cols = df.select_dtypes(include="number")
obj_cols = df.select_dtypes(include="object")
no_dates = df.select_dtypes(exclude=["datetime", "datetimetz"])
```

1.8 Категории (category): как и зачем

Что это. `category` — дискретный тип с фиксированным набором *категорий* (словари меток), экономит память и ускоряет сравнения/группировки. Можно задать порядок категорий (для сортировки и сравнений).

Создание и работа.

```
# простое приведение:
df["city_cat"] = df["city"].astype("category")

# явное описание категорий и их порядка:
cat_dtype = pd.api.types.CategoricalDtype(
    categories=["LA", "NY", "SF"], ordered=True
)
df["city_ord"] = df["city"].astype(cat_dtype)

# доступ к списку категорий:
df["city_ord"].cat.categories      # Index(['LA','NY','SF'])
df["city_ord"].cat.reorder_categories(["NY","LA","SF"], ordered=True,
    inplace=False)
```

Где полезно.

- столбцы со строковыми повторами (города, сегменты, коды);
- сортировки по логике («Low» < «Medium» < «High»);
- уменьшение памяти на больших таблицах.

2 Pandas: пропуски, изменение и объединение данных (часть 2)

2.1 Проверка и обработка пропусков

Что считается пропуском. В `pandas` пропуски представлены как `NaN` (числовые), `NaT` (даты/времена), а также пустые значения `None` в объектных столбцах. Функции-синонимы: `pd.isna` ≡ `pd.isnull`, `pd.notna` ≡ `pd.notnull`.

Диагностика пропусков.

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

df = pd.DataFrame({
    "city": ["NY", "LA", None, "SF"],
    "age":  [25, np.nan, 41, 29],
    "pay":  [3.2, 4.1, np.nan, 3.9]
})

pd.isnull(df)           # булева таблица пропусков
df.isnull().sum()       # число NaN по столбцам
df.isnull().mean()      # доля NaN по столбцам
df["age"].isna().sum() # пропуски в одном столбце
```

Заполнение пропусков: `fillna`. `DataFrame.fillna(value, method=None, axis=None, limit=None, inplace=False)`:

- `value`: скаляр/словарь {`col: value`}/`Series`/`DataFrame`;
- `method`: `"ffill"`/`"pad"` (протянуть вперёд), `"bfill"`/`"backfill"` (назад);
- `axis`: 0 — по строкам, 1 — по столбцам (для методов);
- `limit`: макс. число подряд заполняемых пропусков;
- `inplace`: менять на месте.

```python
# скаляром
df["age_filled"] = df["age"].fillna(df["age"].median())

# по словарю значений для разных столбцов
df2 = df.fillna({"city": "UNK", "pay": df["pay"].mean()})

# протяжка значений вперёд/назад
df_sorted = df.sort_values("age")
df_sorted["pay_ffill"] = df_sorted["pay"].fillna(method="ffill")
df_sorted["pay_bfill"] = df_sorted["pay"].fillna(method="bfill", limit=1)
```

Интерполяция: `interpolate`. `Series/DataFrame.interpolate(method="linear` `axis=0, limit=None, limit_direction="forward` `inplace=False, ...)`:

- `method`: `"linear"` (по индексу), `"time"` (если индекс — даты), `"index"`, `"pad"`/`"ffill"`, `"bfill"`, `"polynomial"`/`"spline"` (нужен `order`);
- `limit`: макс. длина непрерывного блока NaN для заполнения;
- `limit_direction`: `"forward"`/`"backward"`/`"both"`.

```python
s = pd.Series([1.0, np.nan, np.nan, 4.0], index=[0, 1, 2, 3])
s_lin = s.interpolate(method="linear")        # 1.0, 2.0, 3.0, 4.0
# временная интерполяция:
ts = pd.Series([1.0, np.nan, 3.0],
               index=pd.to_datetime(["2024-01-01","2024-01-02","2024-01-04"]))
ts_time = ts.interpolate(method="time")       # по временной шкале
```

2.2 Удаление пропусков: `dropna`

Сигнатура. `DataFrame.dropna(axis=0, how="any thresh=None, subset=None, inplace=False):`

- `axis`: 0 — удалить строки, 1 — столбцы;
- `how`: `"any"` — удалить, если есть хотя бы один NaN; `"all"` — удалить, если все NaN;
- `thresh`: оставить только те, где *число непустых* $\geq$ `thresh`;
- `subset`: ограничить проверку набором столбцов;
- `inplace`: менять на месте.

```
# удалить строки, где есть хотя бы один NaN
d1 = df.dropna()

# удалить столбцы, если все значения в них NaN
d2 = df.dropna(axis=1, how="all")

# оставить строки, где непустых значений >= 2
d3 = df.dropna(thresh=2)

# удалять строки, проверяя только столбцы ["age", "pay"]
d4 = df.dropna(subset=["age", "pay"])

# на месте:
df.dropna(subset=["age"], inplace=True)
```

2.3 Добавление и удаление элементов

Добавление/обновление столбцов.

- Присваивание выравнивается по индексу, длины должны совпадать (или скаляр заполняет весь столбец).
- `assign` создаёт новый `DataFrame` (удобно для пайплайнов).
- `insert(loc, column, value)` вставляет столбец в позицию `loc`.

```
df["age2"] = df["age"] ** 2
df = df.assign(is_adult = lambda t: t["age"] >= 18)

df.insert(1, "city_len", df["city"].fillna("").str.len())  # столбец на позицию
    1
```

Добавление строк. **Важно:** `DataFrame.append` устарел (удалён в pandas 2.0). Используйте `pd.concat` или `loc` с новым индексом.

```python
# через loc (если индекс уникален)
df.loc["u5"] = {"city":"NY", "age":37, "pay":4.8}

# несколькими строками через concat
new_rows = pd.DataFrame([{"city":"LA","age":30,"pay":4.0},
                         {"city":"SF","age":27,"pay":3.6}],
                        index=["u6","u7"])
df = pd.concat([df, new_rows], axis=0)
```

Удаление строк/столбцов: `drop`. `DataFrame.drop(labels=None, axis=0, index=None, columns=None, errors="raise` `inplace=False)`:

- `axis`: 0 — строки, 1 — столбцы;
- `index`/`columns`: альтернативная запись меток для удаления;
- `errors="ignore"` — не падать, если метки не найдены.

```python
# удалить столбцы
df = df.drop(columns=["age2", "city_len"], errors="ignore")

# удалить строки по меткам индекса
df = df.drop(index=["u1", "u3"])

# Series.drop
s = pd.Series([10,20,30], index=["a","b","c"])
s = s.drop(["b"], inplace=False)
```

2.4 Объединение таблиц: `concat`, `merge`, `join`

`pd.concat` склеивает объекты по оси. `pd.concat(objs, axis=0, join="outer` `ignore_index=False` `keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True)`:

- `objs`: список/словарь `Series`/`DataFrame`;
- `axis`: 0 — вертикально (по строкам), 1 — горизонтально (по столбцам);
- `join`: `"outer"` (объединение) или `"inner"` (пересечение) наборов столбцов/индексов;
- `ignore_index`: переиндексация строк `0..N-1`;
- `keys`: добавляет внешний уровень индекса (иерархия) при склейке;
- `verify_integrity`: ошибка при повторе индексов (axis=0) или дубликатах столбцов (axis=1).

Примеры `concat`.

```python
a = pd.DataFrame({"id":[1,2], "x":[10,20]})
b = pd.DataFrame({"id":[3,4], "x":[30,40]})

# 1) Вертикально, переиндексация
ab = pd.concat([a, b], axis=0, ignore_index=True)

# 2) Горизонтально (по столбцам) с выравниванием по индексу
c = pd.DataFrame({"y":[100,200]}, index=[1,3])
axc = pd.concat([a.set_index("id"), c], axis=1, join="outer")

# 3) Внешний уровень индекса (иерархия)
ab_h = pd.concat({"partA": a, "partB": b}, axis=0)

# 4) Жёсткая проверка уникальности индексов
# pd.concat([a.set_index("id"), b.set_index("id")], verify_integrity=True)
```

`pd.merge` — реляционное объединение (JOIN). `pd.merge(left, right, how="inner on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes= copy=True, indicator=False, validate=None)`:

- `how`: `"inner"`, `"left"`, `"right"`, `"outer"`;
- `on`: общий столбец(ы) (если имена совпадают);
- `left_on`/`right_on`: имена столбцов в левой/правой таблице;
- `left_index`/`right_index`: джойн по индексам;
- `suffixes`: суффиксы для одинаковых имён столбцов;
- `indicator=True`: добавить столбец `_merge` (`"left_only"`, `"right_only"`, `"both"`);
- `validate`: контроль кардинальности (`"1:1"`, `"1:m"`, `"m:1"`, `"m:m"`).

Примеры `merge`.

```python
users = pd.DataFrame({"uid":[1,2,3],
                      "city":["NY","LA","SF"]})

pays  = pd.DataFrame({"uid":[1,1,3,4],
                      "pay":[10.0, 12.0, 9.5, 7.0]})

# 1) INNER JOIN по столбцу uid
j1 = pd.merge(users, pays, how="inner", on="uid")
# оставит uid={1,3}

# 2) LEFT JOIN + индикатор источника
j2 = pd.merge(users, pays, how="left", on="uid", indicator=True)
# uid=2 попадёт с NaN в pay; _merge покажет "left_only"

# 3) JOIN по индексам
L = users.set_index("uid")
R = pays.set_index("uid")
j3 = pd.merge(L, R, left_index=True, right_index=True, how="outer")

# 4) Разные имена ключей + суффиксы
left  = pd.DataFrame({"user_id":[1,2], "score":[0.7, 0.9]})
right = pd.DataFrame({"uid":[1,2], "score":[10, 12]})
j4 = pd.merge(left, right, how="inner",
              left_on="user_id", right_on="uid",
              suffixes=("_prob", "_amount"))
```

`DataFrame.join` — удобная обёртка для джойна по индексам или именам столбцов (для правой таблицы можно передать `on`).

```python
L = users.set_index("uid")
R = pays.set_index("uid")
j = L.join(R, how="left")      # left join по индексу
```

2.5 Ещё полезные операции (для качества данных)

`reindex` и `align`. `reindex` подгоняет таблицу под новый набор меток; `align` выравнивает два объекта по общей оси(сям).

```python
s = pd.Series([1,2,3], index=["a","b","c"])
s2 = s.reindex(["b","c","d"], fill_value=0)  # b:2, c:3, d:0

x = pd.Series([1,2], index=["a","b"])
y = pd.Series([10,20], index=["b","c"])
xa, ya = x.align(y, join="outer", fill_value=0)  # выравнивание индексов
```

`combine_first` и `update`. `combine_first` дополняет пропуски из другого объекта; `update` обновляет значения inplace.

```python
a = pd.Series([1, np.nan, 3], index=["x","y","z"])
b = pd.Series([9, 8, 7],       index=["y","z","w"])
c = a.combine_first(b)  # x:1, y:8, z:3, w:7

dfA = pd.DataFrame({"v":[1,np.nan,3]})
dfB = pd.DataFrame({"v":[10,20,30]})
dfA.update(dfB)         # заменит непустыми из B по совместимым индексам
```

where/mask. where сохраняет значения там, где условие истинно, иначе подставляет замену; mask — обратное поведение.

```python
s = pd.Series([10, 20, 30, 40])
s1 = s.where(s >= 25, other=np.nan)  # значения <25 -> NaN
s2 = s.mask(s >= 25, other=0)        # где >=25 -> 0
```

drop_duplicates/duplicated.

- duplicated(subset=None, keep="first"/"last"/False) — булева маска дублей;
- drop_duplicates(subset=None, keep="first"/"last"/False, inplace=False).

```python
df_dups = pd.DataFrame({"a":[1,1,2,2], "b":[3,3,3,4]})
mask = df_dups.duplicated(subset=["a","b"], keep="first")
df_nd = df_dups.drop_duplicates(subset=["a","b"], keep="last")
```

Замечания по inplace. inplace=True модифицирует объект без создания нового. В современных практиках часто предпочитают функциональный стиль без inplace (лучше для читаемости и компоновки пайплайнов), а также потому, что inplace не всегда экономит память.

Итог. В этом разделе собраны типовые приёмы: *диагностика/заполнение/интерполяция* пропусков (isna/fillna/interpolate), *удаление* строк/столбцов (dropna/drop), *добавление* столбцов/строк (assign/insert/loc/concat), а также *объединение* таблиц (concat/merge/join) и служебные операции выравнивания/обновления (reindex/align/combine_fir
Эти инструменты покрывают подавляющее большинство сценариев подготовки и склейки данных в pandas.

3 Pandas: работа с внешними источниками и базовые операции

3.1 CSV: чтение, запись, управление ресурсами и чанки

Как «открываются/закрываются» CSV в pandas.

- Если вы передаёте *путь/URL* строкой в `pd.read_csv(...)` или `DataFrame.to_csv(...)` (например, `"data.csv"` или `"s3://bucket/file.csv"`), то **pandas сам откроет и закроет** файловый дескриптор.
- Если вы передаёте *уже открытый файловый объект* (например, `open("data.csv "r")`), то **закрыть его должны вы** (через `f.close()` или `with`).

Чтение CSV: `pd.read_csv`. Ключевые параметры (самые используемые):

- `filepath_or_buffer`: путь/URL/файловый объект. Поддерживаются компрессии `.gz`, `.bz2`, `.zip`, `.xz` (авто `compression="infer"`).
- `sep` (`delimiter`): разделитель (по умолчанию `"`). Для TSV: `sep="\t"`.
- `header`: строка(и) заголовка (индекс с 0) или `None`, если заголовка нет.
- `names`: список имён столбцов (используйте совместно с `header=None`, если в файле нет заголовка).
- `index_col`: столбец(ы), который станет индексом (имя или позиция).
- `usecols`: подмножество столбцов (имена или позиции) — экономит память/время.
- `dtype`: словарь `{col: dtype}` или тип по умолчанию (ускоряет и снижает расход памяти).
- `na_values`, `keep_default_na`: свои маркеры пропусков и поведение дефолтных `NaN`.
- `parse_dates`: `True`/список столбцов/словарь — парсинг дат, `dayfirst`, `format`, `cache_dates`.
- `encoding`: например, `"utf-8"`, `"cp1251"`; `encoding_errors` — стратегия при ошибках.
- `on_bad_lines`: `"error"` / `"warn"` / `"skip"` — поведение при «битых» строках.
- `engine`: `"c"` (быстро), `"python"` (гибко), `"pyarrow"` (для больших данных; зависит от версии).
- `chunksize`: целое — читать **пакетами** (итерируемый объект `TextFileReader`).
- `iterator=True`: вернуть итератор (аналогично `chunksize`, но без явного размера чанка).
- `storage_options`: параметры для удалённых файловых систем (S3, GCS, WebHDFS и т.п. через `fsspec`).

```python
import pandas as pd

# Базовое чтение
df = pd.read_csv("data.csv")

# Без заголовка, с именами колонок и кастомным разделителем
df = pd.read_csv("data_no_header.tsv",
                 sep="\t", header=None,
                 names=["user_id", "age", "city"])

# Только часть столбцов + явные типы
df = pd.read_csv("big.csv",
                 usecols=["user_id", "age", "pay"],
                 dtype={"user_id":"int64", "age":"float32", "pay":"float32"})

# Пропуски и кодировка
df = pd.read_csv("data_cp1251.csv",
                 encoding="cp1251",
                 na_values=["NA", "None", ""],
                 keep_default_na=True)

# Разбор дат из нескольких столбцов в один datetime
df = pd.read_csv("logs.csv",
                 parse_dates={"ts": ["date", "time"]}, dayfirst=True)

# Чтение из сжатого файла (auto infer) и из URL
df_gz = pd.read_csv("events.csv.gz")
df_url = pd.read_csv("https://example.com/data.csv")
```

Чтение большими файлами по чанкам.

- `chunksize=N` возвращает *итератор*, выдающий куски по `N` строк как `DataFrame`.
- Паттерн: «прочитал чанк → локально агрегировал → накапливаю результат».

```python
totals = {}
for chunk in pd.read_csv("huge.csv", chunksize=100_000,
                         usecols=["city", "pay"], dtype={"pay":"float32"}):
    agg = chunk.groupby("city")["pay"].sum()
    for k, v in agg.items():
        totals[k] = totals.get(k, 0.0) + float(v)

# Финальный результат
totals_df = pd.Series(totals, name="pay_sum").to_frame().sort_values("pay_sum",
    ascending=False)
```

Запись CSV: `DataFrame.to_csv`. Ключевые параметры:

- `path_or_buf`: путь/буфер/URL; `compression` — как при чтении (`"gzip"`, `"zip"`, `"xz"`...).
- `sep`: разделитель, `index`: писать ли индекс (по умолчанию `True`).
- `header`: писать заголовок, `columns`: порядок/подмножество колонок.
- `na_rep`: как выводить `NaN`, `float_format`: формат чисел, `date_format` для дат.
- `quoting`, `quotechar`, `line_terminator`, `mode` (например, `"w"`/`"a"`).

```python
# Запись без индекса в UTF-8
df.to_csv("out.csv", index=False, encoding="utf-8")

# Сжать в gzip
df.to_csv("out.csv.gz", index=False, compression="gzip")

# Кастомные представления пропусков и формат чисел
df.to_csv("out_pretty.csv", index=False, na_rep="NA", float_format="%.3f")
```

Буфер/контекст-менеджер.

```python
# Явно управляем ресурсом (закроется автоматически)
with open("out.csv", "w", encoding="utf-8", newline="") as f:
    df.to_csv(f, index=False)
```

3.2 Excel: чтение/запись, несколько листов, движки

Чтение Excel: `pd.read_excel`. Ключевые параметры:

- `io`: путь/буфер/ExcelFile/URL.
- `sheet_name`: имя/индекс листа, список имён/индексов или `None` (прочитать все листы в словарь).
- `header`, `names`, `usecols`, `dtype`, `converters`, `skiprows`, `nrows`.
- `engine`: `"openpyxl"` для `.xlsx` (дефолт), `"odf"` для `.ods`. (Поддержка `xlrd` для `.xls` зависит от версии).
- `na_values`, `parse_dates`, `date_format`, `thousands`, `decimal`.

```python
# Один лист по имени
df = pd.read_excel("book.xlsx", sheet_name="Sheet1", dtype={"age":"Int64"})

# Несколько листов в dict[str, DataFrame]
sheets = pd.read_excel("book.xlsx", sheet_name=None, usecols="A:C")
df1 = sheets["Sheet1"]
df2 = sheets["Sheet2"]
```

Запись Excel: `DataFrame.to_excel` **и** `ExcelWriter`.

- Для одного листа: `df.to_excel("out.xlsx sheet_name="data index=False)`.
- Для нескольких листов используйте `pd.ExcelWriter` (контекст-менеджер), `engine="openpyxl"/"`
- `mode="a"` + `if_sheet_exists` (например, `"overlay"/"replace"/"new"`) — дозапись в существующий файл.

```python
# Один лист
df.to_excel("report.xlsx", sheet_name="Summary", index=False)

# Несколько листов
with pd.ExcelWriter("report_multi.xlsx", engine="openpyxl", mode="w") as wr:
    df1.to_excel(wr, sheet_name="Train", index=False)
    df2.to_excel(wr, sheet_name="Test", index=False)

# Дозапись в существующий файл (если поддерживается вашей версией pandas/openpyxl)
with pd.ExcelWriter("report.xlsx", engine="openpyxl", mode="a",
    if_sheet_exists="replace") as wr:
    df_new.to_excel(wr, sheet_name="Summary", index=False)
```

3.3 Арифметические операции (структуры с выравниванием)

Операторы и методы: выравнивание по индексам/колонкам.

- **Важно:** операции `+ - * /` и методы `.add/.sub/.mul/.div` *выравнивают* оси по меткам: сначала приводят индексы/колонки к объединению, затем применяют операцию элементwise. Несовпадающие места дают `NaN`, если не задан `fill_value`.
- Методы `.add/.sub/...` поддерживают `fill_value=` для «заполнения отсутствующих» перед операцией.

```python
A = pd.DataFrame({"x":[1,2], "y":[3,4]}, index=["a","b"])
B = pd.DataFrame({"y":[10,20], "z":[5,6]}, index=["b","c"])

A_plus_B = A + B                 # выравнивание по индексам и колонкам -> NaN где нет
    пары
A_add = A.add(B, fill_value=0)   # отсутствующее считаем нулём

# Операции со скаляром и векторизацией по столбцам/строкам
A2 = A * 10
A3 = A.div(A["x"], axis=0)       # делим построчно на столбец "x"
```

3.4 Логические операции, сравнения, `any`/`all`

Сравнения и булевы маски.

- Сравнения `>, >=, <, <=, ==, !=` работают поэлементно и возвращают `DataFrame`/`Series` булевого типа.
- Логика: используйте побитовые операторы `&`, `|`, `~` (NOT) **с обязательными скобками**.

```python
mask = (df["age"] >= 18) & (df["pay"] > 3.5)
adults = df[mask]

# any/all по осям (по умолчанию axis=0: по строкам, собирая столбцы)
has_nan_by_row = df.isna().any(axis=1)       # True, если есть хотя бы один NaN в стр
    оке
all_positive_by_col = (df.select_dtypes("number") > 0).all(axis=0)
```

3.5 Статистика: sum, mean, describe и др.

Частые агрегаты.

- sum, mean, median, min, max, std, var, sem, prod, quantile(q).
- Параметры: axis=0/1, skipna=True/False, numeric_only=True/False.
- Счётчики: count (ненулевых/не NaN), nunique(dropna=True), value_counts() (для Series).
- Сводка: describe() — базовая статистика; percentiles= чтобы настраивать квантили.

```
num = df.select_dtypes(include="number")
stats = num.agg(["count","mean","std","min","max"])
q = num.quantile([0.25, 0.5, 0.75], axis=0)

desc = df.describe(include="all", percentiles=[0.1, 0.5, 0.9])
vc = df["city"].value_counts(dropna=False, normalize=True)
```

Скользящие агрегаты (кратко). rolling(window, min_periods, center) → .mean(), .sum(), .std() и т.п.; expanding(), ewm() для экспоненциальных средних.

3.6 Функции высшего порядка: lambda, apply, pipe

lambda. Анонимные функции удобны для кратких преобразований: lambda x: .... Обычно применяются внутри apply/map/assign.

Series.map и Series.apply.

- map — для Series, применяет функцию/словарь/Series *к каждому значению* (часто быстрее и проще).
- apply на Series — тоже поэлементно, но принимает любую произвольную функцию.

```
# map со словарём (перекодировка)
city2code = {"NY":1, "LA":2, "SF":3}
df["city_code"] = df["city"].map(city2code).fillna(0).astype(int)

# apply с lambda по значениям Series
df["age_bin"] = df["age"].apply(lambda a: "adult" if a>=18 else "teen")
```

DataFrame.apply.

- apply(func, axis=0/1): axis=0 — применить к каждому *столбцу* (получаем Series на вход функции); axis=1 — применить к каждой *строке*.
- Возвращаемый тип зависит от результата функции: скаляр → Series, Series/dict → DataFrame.

```python
# Нормировать каждый столбец (axis=0)
norm = df.select_dtypes("number").apply(lambda s: (s - s.mean()) / s.std(ddof=0),
    axis=0)

# Скомбинировать из строк
def mk_label(row):
    return f"{row['city']}_{int(row['age'])}" if pd.notna(row["age"]) else
    row["city"]

df["label"] = df.apply(mk_label, axis=1)
```

`DataFrame.applymap` **(поэлементно).** Для поэлементного применения по всей таблице (на числовых колонках быстрее использовать векторизацию/NumPy).

```python
df_num = df.select_dtypes("number").applymap(lambda v: round(v, 2))
```

`pipe`: **композиция шагов.** `pipe(func, *args, **kwargs)` — передаёт объект в функцию первым аргументом (читабельные пайплайны).

```python
def clean_cols(t, cols):
    return t.dropna(subset=cols)

def add_ratio(t, num, den, name="ratio"):
    return t.assign(**{name: t[num] / t[den]})

res = (df
       .pipe(clean_cols, cols=["age","pay"])
       .pipe(add_ratio, num="pay", den="age", name="pay_per_age"))
```

3.7 Дополнительно: чтение/запись с удалённых/архивных источников

Компрессии и архивы.

- `compression="infer"` автоматически определяет по расширению (`.gz`, `.bz2`, `.zip`, `.xz`).
- Для ZIP: если внутри **один** файл CSV, `read_csv("file.zip")` обычно работает «из коробки». Если файлов несколько — используйте `zipfile` + `read_csv` по файловому объекту.

```python
import zipfile, io

with zipfile.ZipFile("multi.zip") as zf:
    with zf.open("data/a.csv") as f:
        df_a = pd.read_csv(f)
```

Облачные/удалённые хранилища. С помощью `fsspec` можно читать/писать `s3://`, `gs://`, `hdfs://` и т.п. через `storage_options`.

```
# Пример (псевдо): чтение из S3 с аутентификацией
df = pd.read_csv("s3://my-bucket/data.csv",
                 storage_options={"key":"...", "secret":"..."})
```

Резюме. Для CSV/Excel `pandas` берёт на себя открытие/закрытие файла, если вы даёте путь/URL. Для больших данных используйте `usecols`, `dtype`, `chunksize`. Для Excel — `ExcelWriter` при работе с несколькими листами. Арифметика и сравнения выполняются с *выравниванием* по меткам, для поэлементной логики и агрегаций используйте `apply`/`map`/`pipe`.

4 Pandas: агрегирование, трансформации, строковый аксессор, опции, оконные методы

4.1 Агрегирование данных: `.agg`

Идея. Агрегирование сводит множество значений к одному (или нескольким) показателям: `sum`, `mean`, `max`, пользовательские функции и т.п. Работает для `Series`/`DataFrame`, а также для `GroupBy`.

Пользовательская функция-агрегатор.

```
def strange(x):
    return (x.sum()) ** 0.5

# как lambda + читаемое имя в выходе:
strange = lambda x: (x.sum()) ** 0.5
strange.__name__ = "super_strange"
```

Агрегирование `DataFrame.agg`.

- Можно передавать список строк-имен, список функций, смешивать: `["sum "max strange]`.
- Можно задать разные агрегаторы для разных столбцов через словарь: `{"col1":["mean "std"], "col2":[strange]}`.
- Результат при списке агрегаторов — **многоуровневые колонки** (уровень 1 — столбец, уровень 2 — имя агрегатора).

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

df = pd.DataFrame({
    "group": ["A","A","B","B","B"],
    "x": [1, 2, 3, 4, 5],
    "y": [10, 20, 30, 40, 50]
})

# 1) Обычное агрегирование по всем числовым столбцам:
out1 = df[["x","y"]].agg(["sum", "max", strange])
# Доступ к элементам многоуровневых колонок:
x_sum = out1[("x", "sum")]
y_super = out1[("y", "super_strange")]

# 2) Разные агрегаторы для разных столбцов:
out2 = df.agg({"x": ["mean", "std"], "y": ["min", strange]})
```

Агрегирование `GroupBy.agg`.

- Тот же синтаксис, плюс `NamedAgg` для наименования столбцов.

```python
g = df.groupby("group")
# Список агрегаторов:
gb1 = g[["x","y"]].agg(["mean", "max", strange])

# NamedAgg: сразу получаем плоские имена колонок
gb2 = g.agg(
    x_mean=("x", "mean"),
    x_std =("x", "std"),
    y_super=("y", strange)
)

# Обращение к элементам:
val_A_mean = gb2.loc["A", "x_mean"]
```

Агрегирование `Series.agg`.

```python
s = df["x"]
s_out = s.agg(["sum", "mean", strange])   # Series -> Series
mean_x = s_out["mean"]
```

Как обращаться к элементам результата.

- Если результат с многоуровневыми колонками: `res[("col "agg")]` или `res.loc[:, ("col "agg")]`.
- Если `NamedAgg`: обычные плоские имена столбцов `res["x_mean"]`.
- Для строк — стандартные `.loc[index_label]` / `.iloc[pos]`.

4.2 Трансформации: `.transform`

Идея. `transform` применяет функцию к данным *с сохранением формы* (индекса) исходного объекта: на каждый исходный элемент возвращается соответствующий результат. Часто используется с `GroupBy` для нормализации/стандартизации «внутри группы».

Примеры.

```
# Z-score по всей колонке x (результат той же длины, что df)
z = df["x"].transform(lambda s: (s - s.mean()) / s.std(ddof=0))

# Нормализация внутри групп: вычтем среднее по группе
df["x_centered"] = df.groupby("group")["x"].transform(lambda s: s - s.mean())

# Несколько функций (DataFrame.transform): словарь по столбцам
df_num = df[["x","y"]]
tf = df_num.transform({"x": np.log1p, "y": lambda s: s / s.max()})
```

Доступ к элементам.

- Результат `transform` такой же длины: обращаемся как к обычному столбцу/таблице (`tf["x"]`, `tf.iloc[0]`).
- В `GroupBy.transform` индексы совпадают с исходными — можно `assign` в исходный `DataFrame`.

Сравнение `agg` vs `transform`.

- `agg` → *сжатие*: меньше строк/столбцов (сводные показатели).
- `transform` → *размещение по месту*: длина/индекс сохраняются.

4.3 Строковый аксессор: `.str`

Идея. `Series.str` — векторизованные строковые операции; работает с объектами типа `string`/`object` (строки), часто быстрее, чем `apply` с `lambda`.

Частые операции (цепочки применений).

```python
s = pd.Series(["  new YORK ", None, "sAn-FrAn  "])

# Нормализация формы записи
clean = (s
         .str.strip()            # обрезать пробелы по краям
         .str.lower()            # в нижний регистр
         .str.replace("-", " ", regex=False)
         .str.title()            # Title Case
        )

# Поиск подстроки, регулярки, извлечение
mask = clean.str.contains(r"^San", na=False, regex=True)
parts = clean.str.split(" ", expand=True)      # в DataFrame
num = pd.Series(["id=42", "id=7"]).str.extract(r"id=(\d+)",
    expand=False).astype("Int64")

# Заполнение/выравнивание
padded = pd.Series(["7","42"]).str.zfill(3)    # -> ["007","042"]
```

Полезные методы. `.len()`, `.replace()`, `.contains()`, `.startswith()`, `.endswith()`, `.slice()`, `.pad()`, `.zfill()`, `.get`, `.wrap`, `.get_dummies()`.

4.4 Опции pandas: `get_option`, `set_option`, `reset_option`, `option_context`

Идея. Глобальные настройки отображения/поведения: формат чисел, число видимых строк/колонок, ширина столбцов, вывод дат и т.д.

Чтение/установка/сброс опций.

```python
import pandas as pd

# Посмотреть значение опции
pd.get_option("display.max_rows")

# Установить опцию
pd.set_option("display.max_rows", 200)
pd.set_option("display.max_colwidth", 120)
pd.set_option("display.float_format", lambda v: f"{v:,.3f}")

# Сбросить конкретную опцию (или все через "all")
pd.reset_option("display.max_rows")
# pd.reset_option("all")   # осторожно: сбросит все настройки
```

Контекстный менеджер `option_context`. Временная смена опций (возвращаются по выходу из блока).

```python
with pd.option_context("display.max_rows", 5,
                       "display.max_columns", 8,
                       "display.precision", 2):
    print(df.describe(include="all"))
# Вне блока опции вернутся к прежним значениям
```

Частые префиксы опций. `display.*`, `mode.*` (например, `mode.chained_assignment`), `use_inf_as_na` и др.

4.5 Оконные методы: скользящее и расширяющееся окна

Скользящее окно: `.rolling(window, min_periods, center, ...)`

- Ставит «окно» длины `window` на последовательность и вычисляет агрегат внутри окна.
- `min_periods`: минимум наблюдений внутри окна для вычисления (иначе `NaN`).
- `center=True`: окно центрируется (иначе правое-выровненное).
- После `.rolling(...)` доступны `.mean()`, `.sum()`, `.std()`, `.median()`, `.quantile()`, `.apply(func)`.

```python
s = pd.Series([1, 3, 5, 7, 9, 11])

roll_mean = s.rolling(window=3, min_periods=1).mean()
# 0: mean(1) -> 1.0
# 1: mean(1,3) -> 2.0
# 2: mean(1,3,5) -> 3.0
# 3: mean(3,5,7) -> 5.0
# ...

# Центрированное окно на DataFrame-колонке:
df["x_roll_std"] = df["x"].rolling(window=5, center=True, min_periods=3).std()
```

Расширяющееся окно: `.expanding(min_periods)`

- «Нарастающая» агрегация с начала ряда: сначала по 1-му элементу, потом по первым двум и т.д.
- Доступны те же агрегаты, что и у `rolling`.

```python
exp_sum = s.expanding(min_periods=1).sum()
# -> [1, 1+3, 1+3+5, ...]
df["y_exp_mean"] = df["y"].expanding(min_periods=2).mean()
```

Экспоненциальные окна (вкратце). `.ewm(span=, halflife=, com=)` — экспоненциально взвешенные средние и дисперсии; полезно для временных рядов:

```python
df["x_ewm"] = df["x"].ewm(span=10, adjust=False).mean()
```

4.6 Итоги: когда что применять

- **Агрегирование** (`agg`) — свести данные к компактным метрикам (по столбцам/-группам).
- **Трансформация** (`transform`) — вернуть результат *той же формы*, например, стандартизация по группам.
- **Строковый аксессор** (`str`) — быстрые векторные операции над текстовыми столбцами.
- **Опции** — управлять тем, как `pandas` показывает данные, и делать это локально через `option_context`.
- **Окна** — локальные/нарастающие агрегаты по последовательности (скользящие, расширяющиеся, экспоненциальные).

5 Визуализация данных (`pandas` + `matplotlib`)

Быстрый обзор. У `Series`/`DataFrame` есть метод `.plot(...)`. Ключевой параметр `kind` задаёт тип графика: `"line"`, `"bar"`, `"barh"`, `"hist"`, `"box"`, `"kde"`, `"area"`, `"scatter"`, `"hexbin"`, `"pie"`. Частые параметры: `figsize=(w,h)`, `title=`, `xlabel=`, `ylabel=`, `grid=True`, `legend=True`, `color=` (или `colors=[...]`) и `style=` (например, `-"`, `."`, `"o-"`).

Подготовка данных (примерный датасет)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(7)

# Примерные данные
idx = pd.date_range("2024-01-01", periods=12, freq="M")
df = pd.DataFrame({
    "sales_A": np.random.randint(50, 120, size=12),
    "sales_B": np.random.randint(30, 100, size=12),
    "costs":   np.random.randint(20, 80,  size=12)
}, index=idx)
s = df["sales_A"]
```

Линейные графики (`kind="line"`; по умолчанию)

- **Один ряд:** `Series.plot()`.
- **Несколько рядов:** `DataFrame.plot()` рисует все числовые столбцы.
- Цвет и стиль: `color="C0"` или `color="#1f77b4"`, `style=-"`, маркеры `marker="o"`.

```python
# 1) Series -> одна линия
ax = s.plot(kind="line", figsize=(7, 3.5), title="Sales A (line)",
            color="tab:blue", style="o-", grid=True)
ax.set_xlabel("Month"); ax.set_ylabel("Units")
plt.show()

# 2) DataFrame -> несколько линий
ax = df[["sales_A","sales_B"]].plot(figsize=(7, 3.5), title="Sales A & B",
                                    style=["-o", "--s"], grid=True)
ax.set_xlabel("Month"); ax.set_ylabel("Units")
plt.show()
```

Столбчатые диаграммы: `bar`, `barh`

- `kind="bar"` — вертикальные столбцы, `"barh"` — горизонтальные.
- Для `DataFrame`: `stacked=True` для накопления.

```python
# Вертикальные столбцы (суммарные по месяцам для A и B)
ax = df[["sales_A", "sales_B"]].plot(kind="bar", figsize=(8, 3.8),
                                     title="Sales by month (bar)",
                                     grid=True)
ax.set_xlabel("Month"); ax.set_ylabel("Units")
plt.tight_layout(); plt.show()

# Горизонтальные столбцы (накопленные)
ax = df[["sales_A", "sales_B"]].plot(kind="barh", stacked=True, figsize=(8,
    3.8),
                                     title="Stacked barh: A+B", grid=True)
ax.set_xlabel("Units"); ax.set_ylabel("Month")
plt.tight_layout(); plt.show()
```

Гистограммы и KDE: `hist`, `kde`

- `hist`: распределение значений (параметры: `bins=`, `alpha=`).
- `kde`: оценка плотности (ядровая), сглаженная кривая.

```python
# Гистограмма продаж A
ax = s.plot(kind="hist", bins=10, alpha=0.7, figsize=(6, 3.2),
            title="Histogram: sales_A", color="tab:orange", grid=True)
ax.set_xlabel("Units"); plt.show()

# KDE (плотность) для sales_B
ax = df["sales_B"].plot(kind="kde", figsize=(6, 3.2),
                        title="KDE: sales_B", color="tab:green", grid=True)
ax.set_xlabel("Units"); plt.show()
```

Ящиковые диаграммы: `box`

- Показывают медиану, квартили и возможные выбросы.

- Удобно сравнивать распределения разных столбцов.

```
ax = df[["sales_A","sales_B","costs"]].plot(kind="box", figsize=(6.5, 3.5),
                                            title="Box plots", grid=True)
plt.show()
```

Графики с заливкой: `area`

- `kind="area"` — площадь под кривой; `stacked=True` по умолчанию.

```
ax = df[["sales_A","sales_B"]].plot(kind="area", alpha=0.4, figsize=(7, 3.5),
                                    title="Area chart: A & B", grid=True,
                                    color=["#1f77b4", "#ff7f0e"])
ax.set_xlabel("Month"); ax.set_ylabel("Units")
plt.show()
```

Точечные графики: `scatter`

- Требует указать оси: `x=` и `y=` (для `DataFrame`).
- Можно задавать цвет/размер по столбцу: `c=df["costs"]`, `s=...`, `cmap="viridis"`.

```
ax = df.plot(kind="scatter", x="sales_A", y="sales_B", figsize=(6.5, 3.6),
             title="Scatter: sales_A vs sales_B",
             c=df["costs"], cmap="viridis", s=60, alpha=0.9, grid=True)
ax.set_xlabel("sales_A"); ax.set_ylabel("sales_B")
plt.show()
```

Диаграмма из шестиугольников: `hexbin`

- Хорошо для больших облаков точек: показывает плотность (кол-во точек в ячейке).
- Параметры: `gridsize` (число ячеек по оси), `cmap`.

```
ax = df.plot(kind="hexbin", x="sales_A", y="sales_B", gridsize=15,
             figsize=(6.5, 3.6), cmap="magma", title="Hexbin density")
ax.set_xlabel("sales_A"); ax.set_ylabel("sales_B")
plt.show()
```

Круговая диаграмма: `pie`

- Для `Series`: `Series.plot.pie(...)`; для `DataFrame` — указать `y=` столбец.
- Параметры: `autopct="%.1f%%"`, `startangle=90`, `legend=True`.

```
# Суммы по каналам за год
totals = df[["sales_A","sales_B","costs"]].sum()
ax = totals.plot(kind="pie", figsize=(5.2, 5.2), autopct="%.1f%%",
                 startangle=90, title="Share of totals",
                 colors=["#4c78a8", "#f58518", "#54a24b"])
ax.set_ylabel("")  # убрать подпись оси Y
plt.tight_layout(); plt.show()
```

Цвета и стили линий

- **Цвета:** строковые имена (`"red"`, `"tab:blue"`), HEX (`"#1f77b4"`), RGB-кортежи (`(0.2,0.4,0.6)`).
- **Стили:** `style=` (пунктир), `"."`, `":"`; маркеры `marker="o"`, `"s"`, `"`.
- Прозрачность: `alpha=0.5`; толщина: `linewidth=2` или `lw=2`.
- Палитры: `colormap=` / `cmap=` (например, `"viridis"`, `"magma"`, `"plasma"`).

```
ax = df[["sales_A","sales_B"]].plot(figsize=(7, 3.5),
                                    color=["#1f77b4", "#ff7f0e"],
                                    style=["-o", "--s"], linewidth=2,
                                    title="Custom colors & styles", grid=True)
plt.show()
```

Полезные приёмы

- **Легенда и сетка:** `legend=True`, `grid=True`; управление через `ax.legend()`, `ax.grid(...)`.
- **Подписи:** `ax.set_title()`, `ax.set_xlabel()`, `ax.set_ylabel()`.
- **Несколько графиков:** `df.plot(subplots=True, layout=(r,c))`.
- **Сохранение:** `plt.savefig("plot.png dpi=200, bbox_inches="tight")`.

6 Оформление таблиц в pandas: Styler (цвета, формат, выделение)

Быстрый ориентир. Любой `DataFrame` имеет атрибут `.style`, возвращающий объект `Styler`. Он позволяет:

- **форматировать значения** (`.format()`, `.format_index()`, `na_rep`, `precision`, `thousands`);
- **подсвечивать** ячейки/строки/колонки по правилу (`.applymap()`, `.apply()`, `.highlight_*`, `.background_gradient()`, `.bar()`);
- **править внешний вид таблицы** (`.set_properties()`, `.set_caption()`, `.hide()`, `.set_table_styles()`).

Важно: функции, передаваемые в `.applymap()` и `.apply()`, должны возвращать CSS-строки (например, `"background-color: #ffeeba"`).

Данные для примеров

```
import pandas as pd
import numpy as np
np.random.seed(7)

df = pd.DataFrame({
    "product": ["A","B","C","D","E"],
    "price":   [12.5, 9.99, 15.0, 7.25, 13.4],
    "qty":     [10, 0, 7, 20, 3],
    "margin":  [0.15, 0.32, np.nan, -0.05, 0.18]
})
```

Форматирование чисел: `.format()`

Идея. Управляем отображением чисел и пропусков (не меняя сами данные).

```
# Формат по столбцам (словарём) + общий формат пропусков
styled = (df.style
    .format({
        "price":  "{:.2f} $",
        "margin": "{:.1%}"
    }, na_rep="--")
    .format_index(lambda s: s.upper(), axis="columns")  # формат заголовков
)
```

Замечания.

- `na_rep="-"` задаёт отображение `NaN`.
- Можно использовать `precision=` и `thousands=` (например, `thousands=`) глобально через `pd.options.display`.

Покоэлементная подсветка: `.applymap()`

Идея. Применить функцию к *каждой ячейке* и вернуть CSS-строку.

```
def color_negatives_red(v):
    try:
        return "color: red" if (pd.notna(v) and v < 0) else ""
    except Exception:
        return ""

styled = df.style.applymap(color_negatives_red, subset=["margin"])
```

Когда использовать. Если правило зависит только от *значения одной ячейки.*

Подсветка по строкам/колонкам: `.apply()`

Идея. Функция получает `Series` (строку или колонку) и должна вернуть `list`/`Series` CSS той же длины.

```
# Подсветить максимумы в каждом столбце
def highlight_col_max(col: pd.Series):
    is_max = col == col.max(skipna=True)
    return ["background-color: #d1e7dd" if m else "" for m in is_max]

styled = df.style.apply(highlight_col_max, axis=0, subset=["price","qty"])

# Подсветить всю строку, если qty == 0
def highlight_row_if_zero(row: pd.Series):
    color = "background-color: #fff3cd" if row["qty"] == 0 else ""
    return [color] * len(row)

styled2 = df.style.apply(highlight_row_if_zero, axis=1)
```

Когда использовать. Если правило опирается на *всю строку/весь столбец* (сравнения, агрегаты).

Встроенные хелперы: `highlight_max/min/null/between`, `background_gradient`, `bar`

Готовые методы. Ускоряют типовые сценарии.

```
# Максимумы/минимумы
s1 = (df.style
      .highlight_max(subset=["price","qty"], axis=0, color="#c7f5d9")
      .highlight_min(subset=["price","qty"], axis=0, color="#fbd5d5"))

# Пропуски
s2 = df.style.highlight_null(null_color="#ffd6a5")

# Градиент фона (по колонкам, colormap из matplotlib)
s3 = df.style.background_gradient(cmap="Blues", subset=["price","qty"])

# "Полоска-прогресс" в ячейке по столбцу qty
s4 = df.style.bar(subset=["qty"], color="#ffe08a", vmin=0)
```

Пояснения.

- `axis=0` — ищем максимум/минимум в *каждом столбце*; `axis=1` — в каждой строке.
- `background_gradient` подкрашивает интенсивнее большие значения.
- `bar` рисует внутри ячейки горизонтальную полоску (min/max берутся из данных или задаются `vmin/vmax`).

Настройка свойств и структуры: `.set_properties()`, `.set_caption()`, `.hide()`

Единые CSS для подтаблицы.

```
styled = (df.style
    .set_caption("Demo table")
    .set_properties(subset=pd.IndexSlice[:, ["price","qty"]],
                    **{"text-align": "right", "font-weight": "600"})
    .hide(axis="index")   # скрыть индекс (pandas >= 1.4)
)
```

Прочее.

- `.set_table_styles()` — сложные табличные стили (границы, заголовки).
- `.to_html("table.html")` — экспорт отформатированной таблицы в HTML.

Комбинирование: формат + подсветка + градиент

Последовательность вызовов имеет значение. Обычно: `format` → подсветки → прочие стили.

```
styled = (df.style
    .format({"price": "{:,.2f} $", "margin": "{:.1%}"}, na_rep="--")
    .highlight_max(subset=["price","qty"], axis=0, color="#d1e7dd")
    .applymap(lambda v: "color: red" if (pd.notna(v) and v < 0) else "",
              subset=["margin"])
    .background_gradient(subset=["qty"], cmap="Greens")
    .set_caption("Combined styling")
)
```

Функции для `applymap()`: выделение ячеек

Примеры простых функций.

```
def highlight_if_zero(v):
    return "background-color: #fff3cd" if (pd.notna(v) and v == 0) else ""

def bold_if_high_price(v):
    return "font-weight: 700" if (pd.notna(v) and v >= 12) else ""

styled = (df.style
    .applymap(highlight_if_zero, subset=["qty"])
    .applymap(bold_if_high_price, subset=["price"])
)
```

Функции для `apply()`: выделение строк/колонок

Подсветка строки по условию нескольких полей.

```python
def highlight_row_complex(row: pd.Series):
    cond = (row["qty"] == 0) or (pd.notna(row["margin"]) and row["margin"] < 0)
    color = "background-color: #fde2e4" if cond else ""
    return [color] * len(row)

styled = df.style.apply(highlight_row_complex, axis=1)
```

Советы и типичные ошибки

- **Функции для стилера не должны менять данные:** возвращайте только CSS-строки, не делайте `inplace`-модификаций `df`.
- **Размерность результата:** `apply()` обязана вернуть массив той же длины, что и входная `Series`; `applymap()` — строку на каждую ячейку.
- **Производительность:** сложные `apply`/`applymap` на большие таблицы могут быть медленными; рассмотреть встроенные `highlight_*`, `background_gradient`, `bar`.
- **Экспорт/рендеринг:** стили — это HTML/CSS; в текстовые форматы (CSV/Parquet) они не сохраняются.

7 Частые приёмы в pandas: «шпаргалка» с примерами

Подготовка данных для примеров

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

df = pd.DataFrame({
    "name": ["Ann","Bob","Bob","Cory","Ann","Dana","Eli"],
    "age":  [23, 35, 35, 29, 41, np.nan, 35],
    "mark": [80, 91, 75, 60, 92, 74, 60],
    "grp":  ["A","A","B","B","A","B","B"],
    "date": pd.to_datetime([
        "2021-01-01","2021-01-01","2021-01-03",
        "2021-01-04","2021-01-05","2021-01-05","2021-01-07"
    ])
})
```

7.1 `describe()`: сводная статистика

Что делает. Вычисляет базовые статистики по столбцам (count/mean/std/min/квантили/max) для числовых признаков (или для всех при `include="all"`).

```python
df.describe()                    # только числовые
df.describe(include="all")       # все столбцы (в т.ч. object/categorical)
df["mark"].describe(percentiles=[.05,.5,.95])  # управлять квантилями
```

7.2 `head()`: первые строки

Что делает. Показывает верхнюю часть таблицы (по умолчанию 5 строк).

```
df.head()        # 5 строк
df.head(3)       # 3 строки
df.tail(2)       # последние 2
```

7.3 Между значениями: `Series.between`

Что делает. Булев фильтр по интервалу.

```
start, end = 30, 40
# pandas >= 1.3: inclusive принимает 'both'|'neither'|'left'|'right'
mask = df["age"].between(start, end, inclusive="both")
print(df[mask])

# старый синтаксис (устаревший): inclusive=True/False
# df["age"].between(start, end, inclusive=True)
```

7.4 `groupby`: ключевые приёмы

Почему «двойные скобки».

- `df.groupby("grp")["mark"]` даёт `SeriesGroupBy` (одна колонка).
- `df.groupby("grp")[["mark"]]` даёт `DataFrameGroupBy` (таблица).

От этого зависит форма результата после агрегации (серия vs датафрейм), имена столбцов и удобство последующей работы.

Базовые операции.

```
# Среднее по одной колонке (SeriesGroupBy -> Series)
g1 = df.groupby("grp")["mark"].mean()

# Среднее по колонке, но сохранить как DataFrame
g2 = df.groupby("grp")[["mark"]].mean()

# Несколько агрегаций (имена в кавычках или функции)
g3 = df.groupby("grp")[["mark","age"]].agg(["mean","max","count"])

# Пользовательская функция + переименование столбцов
def iqr(x): return x.quantile(.75) - x.quantile(.25)
g4 = (df.groupby("grp")[["mark"]]
        .agg(avg=("mark","mean"), iqr=("mark", iqr)))

# Обращение к строкам/столбцам после groupby:
g2.loc["A"]                 # строка группы 'A'
g3["mark"]["mean"]          # столбец 'mark'->'mean' (мультииндекс по столбцам)
g3.xs(("mark","mean"), axis=1)  # срез по мультииндексу столбцов

# Получить сырой поднабор группы
df.groupby("grp").get_group("A")
```

Счётчики.

```
df.groupby("grp").size()            # размер групп (включая NaN)
df.groupby("grp")["name"].count()   # не-NaN в 'name' по группам
```

7.5 value_counts: частоты значений

Что делает. Считает частоты уникальных значений (по убыв.).

```
df["name"].value_counts()                          # counts
df["name"].value_counts(normalize=True)            # доли
df["name"].value_counts(dropna=False, sort=True)   # управляем NaN/сортировкой
```

7.6 sort_values: сортировка

```
df.sort_values("age")                        # по одному столбцу
df.sort_values(["grp","mark"], ascending=[True, False])  # по нескольким
```

7.7 apply и map

Для Series:

```
s = df["name"]
s.map({"Ann":"ANN"})                 # map: подстановка по словарю/функции
s.map(lambda x: x.lower())

df["mark"].apply(lambda x: f"{x} pts")  # apply: произвольная функция
```

Для DataFrame:

```
# apply по строкам (axis=1) или по столбцам (axis=0)
df.apply(lambda col: col.isna().sum(), axis=0)       # NaN по колонкам
df.apply(lambda row: row["mark"] - (row["age"] or 0), axis=1)
```

Замечания: `Series.map` удобен для *замены* значений по словарю/функции. `Series.apply` универсальнее. Для `DataFrame` нет `.map`, но есть `.apply` и `.applymap` (поэлементно).

7.8 replace: замены (с/без inplace)

```
# Без inplace: возвращает копию
df2 = df.replace({"name": {"Ann":"Anne", "Bob":"Robert"}})

# С inplace: изменяет на месте
df.replace({"grp": {"A":"Alpha","B":"Beta"}}, inplace=True)

# Регулярные выражения (в строковых столбцах)
df["name"] = df["name"].replace(r"^A.*", "A*", regex=True)
```

7.9 `idxmax()`: индекс максимума

Что делает. Возвращает метку индекса, где достигается максимум.

```
df["mark"].idxmax()          # индекс строки с макс. mark
# Для DataFrame по столбцам: получим индексы строк максимумов для каждого столбца
df[["age","mark"]].idxmax(axis=0)

# Пара настроек отображения + быстрый просмотр
pd.set_option("display.max_columns", 10)
pd.set_option("display.width", 120)
print(df.head())
```

7.10 `groupby mean` и повторно `value_counts`

```
df.groupby(["grp"])[["mark"]].mean()
df["grp"].value_counts()
```

7.11 `.index`, `.values` (`.to_numpy()`)

```
idx  = df.index              # объект Index (метки строк)
vals = df.values             # numpy-массив (устаревшая форма)
arr  = df.to_numpy()         # предпочтительнее: указывает dtype и copy
```

7.12 Удобно «менять»: имена, порядок, типы

```
# Переименование столбцов
df_ren = df.rename(columns={"mark":"score", "grp":"group"})

# Массовая правка: функция над именами
df2 = df.copy()
df2.columns = [c.upper() for c in df2.columns]

# Перестановка столбцов
df3 = df[["name","grp","mark","age","date"]]

# Смена типа
df["mark"] = df["mark"].astype("float64")
```

7.13 `assign`: создание вычисляемых столбцов

Идея. Передаём лямбда-функции, принимающие `DataFrame` и возвращающие столбец.

```
pred = df["mark"] >= 80
true = df["name"].isin(["Ann","Bob"])  # игрушечный пример

out = df.assign(
    y_true = true.astype(int),
    y_pred = pred.astype(int),
```

```
    correct = lambda d: (d["y_true"] == d["y_pred"]).astype(int)
)
```

Группировка: нельзя «по тому, чего нет» & как исправлять

Проблема. Если в какой-то день событий не было, то в исходных данных нет ни одной строки с этой датой. Простая `groupby` не покажет «нулевой» день.

Решения.

- **Через** `pivot_table` + `reindex` с полным датным индексом.
- **Через** `groupby` + `resample` (если `DatetimeIndex`).

```
# Счётчик событий по датам + заполнение пропусков нулями
daily = (df.groupby("date")
           .size()
           .reindex(pd.date_range(df["date"].min(),
                                  df["date"].max(), freq="D"),
                    fill_value=0)
           .rename("n_events"))

# То же через pivot_table
pt = (pd.pivot_table(df, index="date", values="name",
                     aggfunc="count")   # count/mean/sum/len и т.п.
        .reindex(pd.date_range(df["date"].min(),
                               df["date"].max(), freq="D"),
                 fill_value=0)
        .rename(columns={"name":"n_events"}))
```

Агрегация и имена методов. `.agg(...)` и `.aggregate(...)` — *синонимы*. В `.agg` допустимо:

- строки-имена `"mean"`, `"sum"` (быстро и кратко),
- функции `np.mean`, `np.sum` или свои функции,
- словари для переименования результата: `{new_name:(col, func)}`.

Приведение к «плоскому» виду.

```
g = df.groupby("grp")[["mark","age"]].agg(["mean","max"])
g_reset = g.reset_index()          # убрать групповой индекс в колонку
g_flat  = g_reset.copy()
g_flat.columns = ["grp","mark_mean","mark_max","age_mean","age_max"]
```

Дата-время: частая ошибка с `weekday`

Правильно: `weekday` — *атрибут* `.dt`, а не вызываемая функция.

```
df["Date"] = pd.to_datetime(df["date"])
df["week_days"] = df["Date"].dt.weekday      # 0=Mon .. 6=Sun   (без скобок)
df["day_names"] = df["Date"].dt.day_name()   # метод -> со скобками
```

7.14 read_csv(..., parse_dates=...)

Что делает. Парсит столбцы в datetime64.

```
# Одна колонка-дату
t = pd.read_csv("events.csv", parse_dates=["date"])

# Несколько дат
t = pd.read_csv("events.csv", parse_dates=["start_dt","end_dt"])

# Комбинированные даты из нескольких столбцов
t = pd.read_csv("events.csv",
                parse_dates={"dt": ["date","time"]})  # объединить 'date'+'time'
```

7.15 append (устар.), concat, merge

append (устарело): используйте pd.concat.

```
# Было: df = df.append(row_df, ignore_index=True)
# Стало:
df = pd.concat([df, row_df], ignore_index=True)
```

concat: склейка по оси.

```
df1 = df.iloc[:3]
df2 = df.iloc[3:]

# По строкам (ось 0)
v = pd.concat([df1, df2], axis=0, ignore_index=True)

# По столбцам (ось 1) с выравниванием по индексу
left  = df[["name","grp"]].set_index(df.index)
right = df[["mark","age"]].set_index(df.index)
h = pd.concat([left, right], axis=1, join="inner")  # inner/outer
```

merge: SQL-подобные соединения.

```
users = pd.DataFrame({"name":["Ann","Bob","Cory"], "uid":[1,2,3]})
scores = pd.DataFrame({"uid":[1,1,2,4], "score":[80,90,75,60]})

# Виды join: how="inner"|"left"|"right"|"outer"
m = users.merge(scores, on="uid", how="left")

# Соединение по разным именам ключей
left = pd.DataFrame({"user":["Ann","Bob"], "x":[1,2]})
right= pd.DataFrame({"name":["Ann","Cory"], "y":[10,20]})
m2 = left.merge(right, left_on="user", right_on="name", how="inner")
```

7.16 (Пропущено автором списка)

7.17 21. Временные ряды: индексы, частоты, оффсеты

Полезные конструкции.

```
start = pd.Timestamp("2020-01-01")
end   = pd.Timestamp("2020-12-01")

# Полный месячный индекс, приведённый к первому дню месяца:
full_index = (pd.period_range(start=start, end=end, freq="M")
              .to_timestamp("M")             # последний день месяца
              - pd.offsets.MonthEnd(1)       # сместиться назад на 1 месяц к концу
              + pd.offsets.MonthBegin(1))    # перейти к началу месяца

# Ещё короче: просто первый день каждого месяца
mi = pd.date_range(start, end, freq="MS")    # MS = Month Start

# Ресемплинг (если индекс = DatetimeIndex):
ts = (df.set_index("date")
        .assign(n=1)
        .resample("D")["n"].sum()            # события в день
        .reindex(pd.date_range(df["date"].min(),
                               df["date"].max(), freq="D"),
                 fill_value=0))
```

Ещё полезное: пропуски, удаление/добавление строк

Пропуски (`isnull`, `fillna`, `interpolate`).

```
df.isnull()                  # булева маска
df.isnull().sum()            # NaN по столбцам
df.fillna({"age": df["age"].median()}, inplace=True)

# Интерполяция по числовым колонкам
s = pd.Series([1, np.nan, 3, np.nan, 5])
s.interpolate(method="linear")   # 1.0, 2.0, 3.0, 4.0, 5.0
```

Удаление строк/столбцов: `dropna`, `drop`.

```
df.dropna()                      # удалить строки с NaN
df.dropna(axis=1)                # удалить столбцы с NaN
df.dropna(thresh=3)              # оставить строки, где >=3 ненулевых значений
df.dropna(subset=["age","mark"])  # учитывать только указанные столбцы

# Удаление элементов по ярлыкам/позициям
df = df.drop(index=[0, 2])       # убрать строки 0 и 2
df = df.drop(columns=["age"])    # убрать столбец
```

Добавление: новые столбцы, строки.

```
df["passed"] = (df["mark"] >= 80).astype(int)   # новый столбец

# Добавить строки (рекомендуется через concat)
new_rows = pd.DataFrame({"name":["Zoe"], "age":[22], "mark":[88], "grp":["A"],
                         "date":[pd.Timestamp("2021-01-02")]})
df = pd.concat([df, new_rows], ignore_index=True)
```

Индексирование и логические выражения

Базовые способы доступа.

```
df["mark"]                # столбец как Series
df[["name","mark"]]       # несколько столбцов
df.loc[0:3, ["name","mark"]]        # по меткам
df.iloc[0:3, 0:2]                    # по позициям
df[0:4]                              # срез строк
df[df["mark"] > 80]                  # булев фильтр
```

Логические выражения с map/lambda.

```
# Сложные условия
f = (df["grp"].map({"A":1,"B":0}).fillna(0) == 1) & (df["mark"] >= 80)
df[f]

# Несколько условий
df[(df["mark"] >= 80) & (df["age"].between(20, 40, inclusive="both"))]

# isin для множеств
df[df["name"].isin(["Ann","Cory"])]
```

Отображение опций

```
pd.set_option("display.max_columns", 10)
pd.set_option("display.width", 120)
print(df.head())
```