

Лекция 5: Очистка и подготовка данных

Как превратить «грязную» таблицу в данные, которым можно доверять

Азамат Ботирович Кучкаров

Tashkent International University
Кафедра точных наук и цифровых технологий

5 октября 2026 г.

- Почему очистка данных занимает большую часть времени аналитика
- Находить пропуски (`NaN`), в том числе скрытые: `"N/A"`, `"?"`, `""`
- Выбирать стратегию: удаление (`dropna`) или заполнение (`fillna`, `ffill`, `interpolate`)
- Находить и удалять дубликаты (`duplicated`, `drop_duplicates`)
- Исправлять типы данных (`astype`, `pd.to_numeric`, `pd.to_datetime`)
- Находить выбросы методами IQR и z-оценки и решать, что с ними делать
- Очищать текст: `strip`, `lower`, `replace`, регулярные выражения
- Проверять согласованность данных: диапазоны, категории, логические правила

Связь с предыдущими лекциями

Лекция 3 — `DataFrame`, `.loc`, фильтрация. Лекция 4 — загрузка CSV, Excel, JSON. Теперь: загруженные данные почти никогда не готовы к анализу.

1. Зачем нужна очистка данных
2. Пропущенные значения: поиск
3. Стратегии: `dropna` и `fillna`
4. Дублирующиеся строки
5. Исправление типов данных
6. Выбросы (outliers)
7. Очистка строк: `strip`, `replace`, `regex`
8. Проверка согласованности
9. Полный конвейер очистки
10. Заключение

1. Принцип GIGO: Garbage In — Garbage Out

«Мусор на входе — мусор на выходе». Лучшая модель и красивый график дадут неверный результат на ошибочных данных.

- **Одна опечатка в цене** (1200000 вместо 12000) — средняя цена растёт в 10+ раз
- **Город записан по-разному** (Ташкент, ташкент , Tashkent) — в отчёте три города вместо одного
- **Дублирующийся заказ** — завышенная выручка
- **Пропуск принят за ноль** — заниженный средний чек

Известный факт

На поиск, очистку и подготовку данных уходит **60–80%** рабочего времени аналитика.

Проблема	Пример	Инструмент pandas
Пропуски	Пустая ячейка, <code>None</code> , <code>"N/A"</code>	<code>isna</code> , <code>dropna</code> , <code>fillna</code>
Дубликаты	Один заказ записан дважды	<code>duplicated</code> , <code>drop_duplicates</code>
Неверный тип	Цена как текст <code>"4 500"</code>	<code>astype</code> , <code>pd.to_numeric</code>
Выбросы	Цена хлеба 1 200 000 сум	<code>quantile</code> , <code>clip</code>
Грязный текст	<code>" Алишер "</code> , <code>"алишер"</code>	<code>.str.strip</code> , <code>.str.lower</code> , <code>regex</code>
Несогласованность	Количество <code>-1</code> , доставка до заказа	Логические проверки, <code>assert</code>

Сквозной пример: «грязные» данные о продажах

```
df_raw = pd.DataFrame({
    'order_id': [101, 102, 103, 103, 104, 105, 106, 107, 108, 109],
    'mijoz': [' Алишер ', 'Дилноза', 'Бобур', 'Бобур', 'алишер', None, 'Камола', 'Жасур',
    ↪ 'Нодира', 'Шахзод'],
    'shahar': ['Ташкент', 'ташкент ', 'Самарканд', 'Самарканд', 'Tashkent', 'Бухара',
    ↪ 'самарканд', 'N/A', 'Бухара', 'Ташкент'],
    'narx': ['12000', '4 500', '8000', '8000', '15000', '9000', '?', '7000', '1200000',
    ↪ '6500'],
    'soni': [2, 1, np.nan, np.nan, 3, 1, 2, -1, 1, 4],
    'sana': ['2026-03-01', '2026-03-01', '2026-03-02', '2026-03-02', '2026-03-03',
    ↪ '2026-03-03', '2026-03-04', '2026-03-05', '2026-03-05', None],
})
df = df_raw.copy() # исходник df_raw сохраняем нетронутым
```

Сквозной пример: смотрим на данные

```
print(df)
```

	order_id	mijoz	shahar	narx	soni	sana
0	101	Алишер	Ташкент	12000	2.0	2026-03-01
1	102	Дилноза	ташкент	4 500	1.0	2026-03-01
2	103	Бобур	Самарканд	8000	NaN	2026-03-02
3	103	Бобур	Самарканд	8000	NaN	2026-03-02
4	104	алишер	Tashkent	15000	3.0	2026-03-03
5	105	NaN	Бухара	9000	1.0	2026-03-03
6	106	Камола	самарканд	?	2.0	2026-03-04
7	107	Жасур	N/A	7000	-1.0	2026-03-05
8	108	Нодира	Бухара	1200000	1.0	2026-03-05
9	109	Шахзод	Ташкент	6500	4.0	NaN

Задание: найдите все проблемы

Порядок работы

Сначала **смотрим** на данные, потом **исправляем**, затем **проверяем** результат.
Никогда не очищайте данные «вслепую».

Задание: найдите все проблемы

1. Заказ 103 записан дважды
2. Лишние пробелы и разный регистр в mijoz
3. Явные пропуски в mijoz, soni, sana
4. **Скрытые** пропуски: "N/A", "?"
5. Один город — четыре написания
6. narx — текст, "4 500" с пробелом
7. Цена 1200000 — вероятный выброс
8. Количество -1 — невозможно
9. sana — текст, а не дата

Порядок работы

Сначала **смотрим** на данные, потом **исправляем**, затем **проверяем** результат. Никогда не очищайте данные «вслепую».

2. Что такое NaN

Обозначение	Где встречается
<code>np.nan</code>	Числовые столбцы (<code>float64</code>)
<code>None</code>	Python; превращается в NaN
<code>pd.NA</code>	Типы <code>Int64</code> , <code>boolean</code> , <code>string</code>
<code>pd.NaT</code>	Пропуск в датах (Not a Time)

```
print(np.nan == np.nan) # False
print(type(np.nan))     # float
```

Внимание

NaN не равен ничему, даже самому себе. Проверять пропуски через `==` **нельзя** — только `isna()` / `notna()`.

Почему `son1` имеет тип `float64`?

NaN — число с плавающей точкой. Один пропуск в целочисленном столбце превращает весь столбец во `float64`.

Обнаружение пропусков: info()

Первое, что стоит запустить. Сравните `Non-Null Count` с числом строк.

```
df.info()
```

```
RangeIndex: 10 entries, 0 to 9
#   Column      Non-Null Count  Dtype
---  -
0   order_id    10 non-null      int64
1   mijoz        9 non-null       str
2   shahar       10 non-null      str
3   narx         10 non-null      str
4   soni         8 non-null       float64
5   sana         9 non-null       str
```

Версии pandas

В pandas 3.x текстовые столбцы имеют тип `str`, в версиях 1.x и 2.x — `object`.

Количество и доля пропусков

```
# Количество пропусков
```

```
df.isna().sum()
```

```
# Доля пропусков, %
```

```
(df.isna().mean() * 100).round(1)
```

```
# Строки хотя бы с одним пропуском
```

```
mask = df.isna().any(axis=1)
```

```
mask.sum()          # 4
```

```
df[mask]
```

order_id	0	0.0
mijoz	1	10.0
shahar	0	0.0
narx	0	0.0
soni	2	20.0
sana	1	10.0

Приём

`isna()` возвращает таблицу `True/False`. Так как `True = 1`, `.sum()` считает количество пропусков, а `.mean()` — их долю.

По `isna()` в `shahar` и `narx` пропусков нет, но там есть "N/A" и "?".

Типичные маскировки:

- Текст: "N/A", "-", "?", "нет", "null", "", " "
- Числа-заглушки: 0, -1, 999, 99999
- Даты-заглушки: 1900-01-01, 1970-01-01

```
df['shahar'].value_counts(dropna=False)
```

shahar	
Ташкент	2
Самарканд	2
Бухара	2
ташкент	1
Tashkent	1
самарканд	1
N/A	1

Способ 1: заменить маркеры на NaN после загрузки

```
df = df.replace({'N/A': np.nan, '?': np.nan, '-': np.nan})  
df.isna().sum()  
# order_id 0 | mijoz 1 | shahar 1 | narx 1 | soni 2 | sana 1
```

Способ 2 (лучше): указать маркеры при загрузке файла

```
df = pd.read_csv('savdo.csv', na_values=['N/A', '?', '-', 'нет'])
```

Числовые заглушки

0 в столбце «возраст» или -1 в «количестве» — не всегда пропуск. Решение принимается только после того, как вы поняли **смысл** столбца.

Почему данные пропущены?

Тип	Значение	Пример	Что делать
MCAR	Полностью случайно	Сбой датчика, потерянная анкета	Удалять или заполнять — смещения нет
MAR	Зависит от других столбцов	Молодые реже указывают доход	Заполнять с учётом групп
MNAR	Зависит от самого значения	Люди с высоким доходом его скрывают	Осторожно! Отметить пропуск флагом

Практический вывод

Пропуск — это тоже информация. Иногда сам факт, что клиент **не указал** телефон, важнее самого телефона.

3. Удаление: dropna

```
df.dropna()           # строки с ХОТЯ БЫ ОДНИМ пропуском
df.dropna(how='all')  # строки, где ВСЕ значения пропущены
df.dropna(subset=['soni']) # смотреть только столбец soni
df.dropna(thresh=5)   # оставить строки с >= 5 заполненными
df.dropna(axis=1)     # удалить СТОЛБЦЫ с пропусками
```

Вызов	Результат	Комментарий
<code>df.dropna()</code>	4 строки	Потеряли 60% данных!
<code>df.dropna(subset=['soni'])</code>	8 строк	Удалены только заказы без количества
<code>df.dropna(thresh=5)</code>	10 строк	В каждой строке ≤ 1 пропуска
<code>df.dropna(axis=1)</code>	1 столбец	Остался только <code>order_id</code>

Главная опасность `dropna()`

Даже при 10% пропусков в каждом столбце удаление всех неполных строк может уничтожить половину таблицы. Проверяйте `shape` до и после.

Удаление оправдано, если:

- Пропусков мало (меньше ~5% строк) и они случайны (MCAR)
- Пропущено ключевое значение (`order_id`, целевая переменная в ML)
- Столбец почти пуст (заполнен на 5%) — проще удалить его целиком

Константой:

```
df['soni'].fillna(0) # пропуск -> 0
df['mijoz'].fillna('Неизвестно') # отдельная категория
df.fillna({'mijoz': 'Неизвестно', 'soni': 0}) # словарь: столбец -> значение
```

Осторожно с нулём

«Купили 0 раз» $\neq$ «не знаем, сколько купили». Ноль занижает среднее и сумму.

Статистикой:

```
df['soni'].mean() # 1.625
df['soni'].median() # 1.5
df['soni'] = df['soni'].fillna(df['soni'].median()) # числа
df['shahar'] = df['shahar'].fillna(df['shahar'].mode()[0]) # категории
```

Какую статистику выбрать

Статистика	Когда использовать
Среднее (<code>mean</code>)	Числа с симметричным распределением, без выбросов
Медиана (<code>median</code>)	Числа с выбросами или «перекосом» (цены, доходы, зарплаты) — выбор по умолчанию
Мода (<code>mode</code>)	Категории (город, категория товара)

Почему `mode()[0]`?

`mode()` возвращает `Series`: самых частых значений может быть несколько. `[0]` берёт первое.

Пропущенную цену логичнее заполнить медианой её **категории**, а не общей.

```
m = pd.DataFrame({
    'kategoriya': ['Продукты']*3
                  + ['Электроника']*3,
    'narx': [5000, np.nan, 7000,
            300000, 500000, np.nan],
})
m['narx'] = m['narx'].fillna(
    m.groupby('kategoriya')['narx']
    .transform('median'))
```

	kategoriya	narx
0	Продукты	5000.0
1	Продукты	6000.0
2	Продукты	7000.0
3	Электроника	300000.0
4	Электроника	500000.0
5	Электроника	400000.0

Общая медиана была бы **150 000** — абсурдно и для хлеба, и для телефона.

`transform` возвращает `Series` той же длины: каждой строке — медиана её группы.

Заполнение по соседям: ffill, bfill, interpolate

Для **временных рядов** (курс валют, температура, остатки на складе):

```
s = pd.Series([10, np.nan, np.nan, 16, np.nan, 20])

s.ffill()      # [10, 10, 10, 16, 16, 20]    последнее известное
s.bfill()      # [10, 16, 16, 16, 20, 20]    следующее известное
s.interpolate() # [10, 12, 14, 16, 18, 20]    прямая между точками
```

Внимание

Перед `ffill/bfill/interpolate` данные должны быть **отсортированы по времени**.

Устаревший синтаксис

`df.fillna(method='ffill')` удалён в современных версиях pandas — используйте `df.ffill()`.

Если факт пропуска может быть важен (MNAR), сохраните его **до** заполнения:

```
df['soni_nan'] = df['soni'].isna()      # True, если значение было пропущено
df['soni'] = df['soni'].fillna(df['soni'].median())
```

	soni	soni_nan
0	2.0	False
1	1.0	False
2	1.5	True
3	1.5	True

Информация не потеряна: позже можно проверить, отличаются ли такие строки от остальных.

Ситуация	Стратегия
Пропусков < 5%, случайные	<code>dropna()</code>
Пропущен ID или ключевое поле	<code>dropna(subset=[...])</code>
Столбец пуст на 70%+	Удалить столбец
Числа без выбросов	<code>fillna(mean)</code>
Числа с выбросами (цены, доходы)	<code>fillna(median)</code>
Категории	<code>fillna(mode)</code> или 'Неизвестно'
Значение зависит от группы	<code>groupby(...).transform(...)</code>
Временной ряд	<code>ffill()</code> или <code>interpolate()</code>
Пропуск сам по себе информативен	Флаг-индикатор + заполнение

Не используйте inplace

```
# Плохо: в pandas 3 это НЕ изменит df (Copy-on-Write)  
df['soni'].fillna(0, inplace=True)
```

```
# Хорошо: явное присваивание  
df['soni'] = df['soni'].fillna(0)
```

Copy-on-Write

Начиная с pandas 3.0, `df['soni']` — отдельный объект; изменение «на месте» не затрагивает исходную таблицу. Привыкайте писать `df[...] = ....`

4. Дублирующиеся строки: откуда и как найти

Причины: двойное нажатие «Оформить заказ», объединение пересекающихся файлов, повторный запуск парсера, ввод одного клиента дважды.

```
df.duplicated().sum()          # 1
df[df.duplicated(keep=False)] # показать ВСЕ копии
```

	order_id	mijoz	shahar	narx	soni	sana
2	103	Бобур	Самарканд	8000	NaN	2026-03-02
3	103	Бобур	Самарканд	8000	NaN	2026-03-02

Параметр keep	Что помечается как дубликат
'first' (по умолчанию)	Все копии, кроме первой
'last'	Все копии, кроме последней
False	Все копии

Дубликаты по ключу и удаление

Строки могут отличаться в мелочах, но по смыслу быть одним объектом:

```
df.duplicated(subset=['order_id']).sum()      # номер заказа уникален
df.duplicated(subset=['mijoz', 'sana']).sum()  # клиент + время
df['order_id'].is_unique                       # False
```

Удаление:

```
df = df.drop_duplicates()                    # полные дубликаты
df = df.drop_duplicates(subset=['order_id']) # по ключу, первая
df = df.drop_duplicates(subset=['order_id'], keep='last') # самая свежая
df = df.drop_duplicates().reset_index(drop=True) # + перенумерация
```

Какую копию оставлять?

Если данные отсортированы по времени, `keep='last'` сохраняет **самую актуальную** версию записи.

Неявные дубликаты

`uplicated()` сравнивает **посимвольно**: "Алишер", " алишер ", "АЛИШЕР" — три разных человека.

```
n = pd.DataFrame({'mijoz': ['Алишер', ' алишер ', 'АЛИШЕР', 'Дилноза'],
                  'tel':   ['901', '901', '901', '902']})
n.duplicated().sum()           # 0 — дубликатов как будто нет

n['mijoz'] = n['mijoz'].str.strip().str.title()
n.duplicated().sum()           # 2 — теперь видно
n.drop_duplicates()
```

	mijoz	tel
0	Алишер	901
3	Дилноза	902

Вывод

Сначала очищайте строки, потом ищите дубликаты.

5. Почему тип важен

order_id	int64
mijoz	str
shahar	str
narx	str
soni	float64
sana	str

- `narx` — текст: `sum()` склеит строки, а `df['narx'] > 5000` вызовет ошибку
- `sana` — текст: нельзя извлечь месяц, посчитать разницу дат, правильно отсортировать
- `soni` — `float64`, хотя количество — целое

astype — преобразование в указанный тип:

```
pd.Series(['1', '2']).astype(int)      # str -> int64
pd.Series([1.7, 2.2]).astype(int)      # [1, 2] - дробь ОТБРАСЫВАЕТСЯ!
df['shahar'].astype('category')        # str -> category
```

astype строгий: ошибки преобразования

```
pd.Series(['12000', '4 500', '8000']).astype(int)
# ValueError: invalid literal for int() with base 10: '4 500'

pd.Series([1.0, 2.0, np.nan]).astype(int)
# IntCastingNaNError: Cannot convert non-finite values (NA or inf) to integer
```

pd.to_numeric — безопасное преобразование:

```
s = pd.Series(['12000', '4 500', '8000', '?', '1200000'])
pd.to_numeric(s, errors='coerce')
# [12000.0, NaN, 8000.0, NaN, 1200000.0]
```

`errors='raise'` (по умолчанию) — ошибка; `errors='coerce'` — неверные значения → NaN.

"4 500" — корректная цена с пробелом, но `coerce` молча превратил её в `NaN`.
Сначала очистите строку, затем преобразуйте:

```
pd.to_numeric(s.str.replace(' ', ''), errors='coerce')  
# [12000.0, 4500.0, 8000.0, NaN, 1200000.0]
```

Хорошая привычка

После `coerce` посмотрите, какие значения не распознались:

```
converted = pd.to_numeric(s, errors='coerce')  
print(s[converted.isna() & s.notna()])  # что не удалось преобразовать
```

```
df['sana'] = pd.to_datetime(df['sana'], errors='coerce')
df['sana'].dt.month           # месяц
df['sana'].dt.day_name()     # день недели
df['sana'].max() - df['sana'].min() # длительность периода
```

Осторожно со смешанными форматами — ошибка без предупреждения:

```
d = pd.Series(['2026-03-01', '2026/03/03',
               '05.03.2026', 'нет даты'])
pd.to_datetime(d, errors='coerce',
               format='mixed', dayfirst=True)
```

0	2026-01-03	<- 1 марта
1	2026-03-03	стало
2	2026-03-05	3 января!
3	NaT	

Правило

Если формат известен — укажите его: `format='%d.%m.%Y'`. После преобразования проверяйте `min()` и `max()`.

Специальные типы: Int64, category, bool

Целые числа с пропусками — тип `Int64` (с заглавной буквы):

```
pd.Series([1, None, 3], dtype='Int64')      # [1, <NA>, 3]  dtype: Int64
```

Категориальный тип — экономия памяти при малом числе уникальных значений:

```
c = pd.Series(['Ташкент', 'Бухара', 'Самарканд'] * 10000)
c.memory_usage(deep=True)                # 2 420 132 байт
c.astype('category').memory_usage(deep=True)  # 30 421 байт (~80 раз меньше)
```

«Да/Нет» → `True/False`:

```
b = pd.Series(['ha', "yo'q", 'Ha', 'yes'])
b.str.lower().map({'ha': True, "yo'q": False, 'yes': True})
# [True, False, True, True]
```

Задача	Инструмент
Чистый текст → число	<code>astype(int) / astype(float)</code>
«Грязный» текст → число	очистка + <code>pd.to_numeric(errors='coerce')</code>
Текст → дата	<code>pd.to_datetime(format=...)</code>
Целые числа с пропусками	<code>astype('Int64')</code>
Мало уникальных значений	<code>astype('category')</code>
«Да/Нет» → bool	<code>.map({...})</code>

6. Что такое выброс (outlier)

Выброс — значение, которое сильно отличается от большинства остальных.

Вид	Пример	Что делать
Ошибка	Цена хлеба 1 200 000, возраст 250, рост 17 м	Исправить или удалить
Реальное редкое событие	Оптовый заказ на 500 шт., зарплата директора, «Чёрная пятница»	Оставить — это важная информация

Внимание

Ни один статистический метод не скажет, ошибка это или нет. Методы лишь **находят кандидатов**. Решение принимает аналитик, понимающий предметную область.

Как выброс влияет на статистику

```
narx = pd.Series([12000, 4500, 8000,
                  8000, 15000, 9000, 7000,
                  1200000, 6500, 11000])

narx.mean()      # 128100.0
narx.median()    # 8500.0

narx.plot(kind='box')      # boxplot
narx.plot(kind='hist')     # гистограмма
```

```
narx.describe()
count      1.000000e+01
mean       1.281000e+05
std        3.766393e+05
min        4.500000e+03
25%        7.250000e+03
50%        8.500000e+03
75%        1.175000e+04
max        1.200000e+06
```

Одно значение увеличило **среднее в 15 раз**; медиана почти не изменилась — она **устойчива** (робастна). Разрыв между **75%** и **max** в 100 раз — повод насторожиться.

Метод IQR (межквартильный размах)

1. Q_1 — 25-й процентиль, Q_3 — 75-й
2. $IQR = Q_3 - Q_1$
3. Выбросы — вне $[Q_1 - 1.5 \cdot IQR, Q_3 + 1.5 \cdot IQR]$

Коэффициент 1.5 предложил Джон Тьюки. Для **экстремальных** выбросов используют 3.

```
q1 = narx.quantile(0.25) # 7250.0
q3 = narx.quantile(0.75) # 11750.0
iqr = q3 - q1            # 4500.0
```

```
lower = q1 - 1.5 * iqr   # 500.0
upper = q3 + 1.5 * iqr   # 18500.0
```

```
narx[(narx < lower) | (narx > upper)]
```

```
7      1200000
```

$$z = \frac{x - \bar{x}}{\sigma}, \quad \text{выброс, если } |z| > 3$$

```
z = (narx - narx.mean()) / narx.std()  
# [-0.31, -0.33, -0.32, -0.32, -0.3, -0.32, -0.32, 2.85, -0.32, -0.31]
```

Неожиданно!

Для 1 200 000 получилось $z = 2.85 < 3$ — метод **не нашёл** очевидный выброс. Сам выброс раздул среднее и σ и «спрятался» за ними (особенно на малых выборках).

Робастная версия — медиана и MAD вместо среднего и σ :

```
med = narx.median()  
mad = (narx - med).abs().median()  
robust_z = 0.6745 * (narx - med) / mad      # для 1 200 000: 357.19
```

Метод	Плюсы	Минусы
IQR	Устойчив, не требует нормальности	На «перекошенных» данных находит слишком много
z-score	Простой, понятный	Предполагает нормальность; выброс искажает <code>mean</code> , <code>std</code>
Робастный z	Устойчив к выбросам	Сложнее объяснить заказчику
Бизнес-правила	Самый надёжный, основан на смысле	Требует знания предметной области

```
df[(df['yosh'] < 0) | (df['yosh'] > 110)]    # возраст  
df[df['narx'] > 100_000]                    # в продуктовом магазине нет товаров дороже
```

Что делать с выбросами

1. Удалить строки

```
narx_clean = narx[(narx >= lower) & (narx <= upper)]
```

2. Заменить на NaN и обработать как пропуск

```
narx = narx.where((narx >= lower) & (narx <= upper))
```

3. Ограничить (clip / винзоризация)

```
narx.clip(lower=lower, upper=upper)
```

[12000, 4500, 8000, 8000, 15000, 9000, 7000, 18500, 6500, 11000]

4. Исправить, если причина известна (лишние нули)

5. Оставить и анализировать отдельно (оптовые заказы)

Документируйте решения

Сколько выбросов найдено, каким методом и что с ними сделано. Через месяц этого никто не вспомнит.

7. Аксессор .str

Через `.str` строковые методы Python применяются **ко всему столбцу**, без циклов. `NaN` пропускаются автоматически.

```
s = pd.Series([' Алишер ', 'алишер', 'АЛИШЕР', 'Дилноза ', None])
s.str.strip().str.title()
# ['Алишер', 'Алишер', 'Алишер', 'Дилноза', NaN]
```

Почему лишние пробелы опасны:

```
pd.Series(['Ташкент', 'Ташкент ']).unique()           # 2 - «два города»
pd.Series(['Ташкент', 'Ташкент ']).str.strip().unique() # 1
```

Метод	Что делает	Пример
<code>.str.strip()</code>	Пробелы по краям	' Ташкент ' → 'Ташкент'
<code>.str.lower() / .upper()</code>	Регистр	'ТАШКЕНТ' → 'ташкент'
<code>.str.title()</code>	Заглавная в словах	'ташкент' → 'Ташкент'
<code>.str.replace(a, b)</code>	Замена подстроки	'4 500' → '4500'
<code>.str.contains(p)</code>	Содержит ли подстроку	→ True/False
<code>.str.startswith(p)</code>	Начинается ли с ...	
<code>.str.len()</code>	Длина строки	
<code>.str.split(sep)</code>	Разбить строку	'Ташкент, Чиланзар' → список
<code>.str[a:b]</code>	Срез строки	<code>.str[-9:]</code> — последние 9 символов

Рецепт нормализации категорий

strip → lower → словарь замен → title

```
sh = pd.Series(['Ташкент', 'ташкент ', 'Tashkent',  
               'г. Ташкент', 'Самарканд',  
               ↪ 'самарканд'])  
clean = (sh.str.strip()  
         .str.lower()  
         .str.replace(r'^г\.\s*', '', regex=True)  
         .replace({'tashkent': 'ташкент'})  
         .str.title())  
clean.value_counts()
```

Ташкент	4
Самарканд	2

Обратите внимание на разницу

`Series.str.replace('a', 'b')` — заменяет **подстроку** внутри строки.

`Series.replace({'a': 'b'})` — заменяет **значение ячейки целиком**.

Регулярные выражения (regex): минимальный набор

Шаблон	Значение
<code>\d</code>	Любая цифра
<code>\D</code>	Всё, кроме цифр
<code>\s</code>	Пробельный символ
<code>\w</code>	Буква, цифра или <code>_</code>
<code>.</code>	Любой символ
<code>\.</code>	Буквальная точка

Шаблон	Значение
<code>+</code>	Один или больше раз
<code>*</code>	Ноль или больше раз
<code>{n}</code>	Ровно n раз: <code>\d{9}</code>
<code>^ / \$</code>	Начало / конец строки
<code>[...]</code>	Символ из набора
<code>(...)</code>	Группа (что извлечь)

Совет

Пишите шаблоны в «сырых» строках `r'...'`, чтобы Python не обрабатывал `\`. Проверяйте шаблоны на regex101.com.

Regex: нормализация телефонных номеров

```
tel = pd.Series(['+998 90 123-45-67', '998901234568', '90 123 45 69',
                '+998(91)1112233', '93-444-55-66', '12-34'])

# Шаг 1: удалить всё, кроме цифр
digits = tel.str.replace(r'\D', '', regex=True)
# ['998901234567', '998901234568', '901234569', '998911112233', '934445566',
  ↪ '1234']

# Шаг 2: последние 9 цифр, если длина корректна (9 или 12), иначе NaN
local = digits.str[-9:].where(digits.str.len().isin([9, 12]))

# Шаг 3: единый формат
tel_clean = '+998' + local
# ['+998901234567', '+998901234568', '+998901234569',
  # '+998911112233', '+998934445566', NaN]
```

Некорректный 12-34 стал NaN — его легко найти и проверить вручную.

Число из текста — `.str.extract` возвращает содержимое группы (...):

```
price = pd.Series(['12 000 сум', '4500 so'm', '8,000', 'цена: 9000',  
↪ 'бесплатно'])  
(price.str.extract(r'(\d[\d\s,]*)')[0]  
  .str.replace(r'[\s,]', '', regex=True)  
  .pipe(pd.to_numeric, errors='coerce'))  
# [12000.0, 4500.0, 8000.0, 9000.0, NaN]
```

Проверка формата email — `.str.match`:

```
em = pd.Series(['ali@mail.uz', 'bobur.mail.uz', 'kamola@gmail.com ', 'test@'])  
em.str.strip().str.match(r'^[\w.+~]+@[~\w-]+\.[\w.]+$')  
# [True, False, True, False]
```

Лишние пробелы внутри: `.str.replace(r'\s+', ' ', regex=True)`

8. Проверка согласованности данных

Даже без пропусков и дубликатов данные могут быть **логически неверными**.

```
# 1. Диапазон
df[df['soni'] <= 0]                # количество > 0
df[~df['yosh'].between(0, 110)]    # возраст 0..110
df[df['sana'] > pd.Timestamp('today')] # дата не в будущем

# 2. Допустимые значения
allowed = ['Ташкент', 'Самарканд', 'Бухара', 'Неизвестно']
df[~df['shahar'].isin(allowed)]

# 3. Уникальность
df['order_id'].is_unique
```

Знак ~ — логическое «НЕ»: строки, где условие **не** выполняется.

Межстолбцовые (логические) правила

```
o = pd.DataFrame({
    'narx': [1000, 2000], 'soni': [2, 3], 'summa': [2000, 5000],
    'buyurtma_sana': pd.to_datetime(['2026-03-01', '2026-03-05']),
    'yetkazish_sana': pd.to_datetime(['2026-03-03', '2026-03-02'])})

o[o['narx'] * o['soni'] != o['summa']]      # сумма != цена x количество
o[o['yetkazish_sana'] < o['buyurtma_sana']]  # доставка раньше заказа
```

	narx	soni	summa	buyurtma_sana	yetkazish_sana
1	2000	3	5000	2026-03-05	2026-03-02

Строка 1 нарушает **оба** правила: $2000 \times 3 = 6000 \neq 5000$, доставка 2 марта — раньше заказа 5 марта.

Дробные числа

Вместо `!=` используйте допуск: `(a - b).abs() > 0.01`.

```
def check(df):
    problems = {
        'дубликаты order_id':
            df['order_id'].duplicated().sum(),
        'soni <= 0': (df['soni'] <= 0).sum(),
        'narx <= 0': (df['narx'] <= 0).sum(),
        'неизвестный город': (~df['shahar'].isin(
            ['Ташкент', 'Самарканд', 'Бухара',
             'Неизвестно'])).sum(),
        'дата в будущем':
            (df['sana'] > pd.Timestamp('today')).sum(),
        'пропуски (всего)': df.isna().sum().sum(),
    }
    return pd.Series(problems)
```

assert — «сигнализация»:
останавливает программу,
если условие ложно.

```
assert df['order_id'].is_unique
assert (df['soni'] > 0).all()
assert df['narx'].notna().all()
```

Данные обновляются: завтра
появится новый вид ошибки —
и вы узнаете об этом **сразу**, а
не после отчёта руководству.

9. Порядок действий

#	Шаг	Инструменты
1	Осмотр	<code>info()</code> , <code>describe()</code> , <code>value_counts()</code> , <code>isna().sum()</code>
2	Скрытые NaN	<code>"N/A"</code> , <code>"?"</code> , <code>""</code> → <code>NaN</code>
3	Строки	<code>strip</code> , <code>lower</code> , <code>replace</code> , <code>regex</code>
4	Типы	<code>to_numeric</code> , <code>to_datetime</code> , <code>astype</code>
5	Дубликаты	<code>drop_duplicates</code>
6	Выбросы и ошибки	невозможные значения → <code>NaN</code>
7	Пропуски	<code>dropna</code> / <code>fillna</code>
8	Проверка	<code>check()</code> , <code>assert</code>

Порядок важен

Дубликаты — **после** очистки строк. Пропуски — **после** обработки выбросов (иначе выброс исказит статистику заполнения).

Функция очистки (1/2): строки и типы

```
def clean(row):  
    df = row.copy()      # не изменяем исходные данные!  
  
    # 1. Скрытые пропуски  
    df = df.replace({'N/A': np.nan, '?': np.nan, '': np.nan})  
  
    # 2. Очистка строк  
    df['mijoz'] = df['mijoz'].str.strip().str.title()  
    df['shahar'] = (df['shahar'].str.strip().str.lower()  
                   .replace({'tashkent': 'ТАШКЕНТ'})  
                   .str.title())  
  
    # 3. Типы данных  
    df['narx'] = pd.to_numeric(df['narx'].str.replace(r'\s', '', regex=True),  
                              ↪ errors='coerce')  
    df['sana'] = pd.to_datetime(df['sana'], errors='coerce')
```

Функция очистки (2/2): дубликаты, выбросы, пропуски

4. Дубликаты

```
df = df.drop_duplicates(subset='order_id', keep='first')
```

5. Невозможные значения и выбросы -> NaN

```
df.loc[df['soni'] <= 0, 'soni'] = np.nan
```

```
df['narx'] = df['narx'].where(df['narx'] <= 100_000)
```

6. Пропуски

```
df['soni'] = df['soni'].fillna(df['soni'].median()).astype(int)
```

```
df['narx'] = df['narx'].fillna(df['narx'].median())
```

```
df['mijoz'] = df['mijoz'].fillna('Неизвестно')
```

```
df['shahar'] = df['shahar'].fillna('Неизвестно')
```

```
df['sana'] = df['sana'].ffill()
```

7. Новый столбец

```
df['summa'] = df['narx'] * df['soni']
```

```
return df.reset_index(drop=True)
```

```
clean_df = clean(df_raw)
```

	order_id	mijoz	shahar	narx	soni	sana	summa
0	101	Алишер	Ташкент	12000.0	2	2026-03-01	24000.0
1	102	Дилноза	Ташкент	4500.0	1	2026-03-01	4500.0
2	103	Бобур	Самарканд	8000.0	2	2026-03-02	16000.0
3	104	Алишер	Ташкент	15000.0	3	2026-03-03	45000.0
4	105	Неизвестно	Бухара	9000.0	1	2026-03-03	9000.0
5	106	Камола	Самарканд	8000.0	2	2026-03-04	16000.0
6	107	Жасур	Неизвестно	7000.0	2	2026-03-05	14000.0
7	108	Нодира	Бухара	8000.0	1	2026-03-05	8000.0
8	109	Шахзод	Ташкент	6500.0	4	2026-03-05	26000.0

order_id	int64
mijoz	str
shahar	str
narx	float64
soni	int64
sana	datetime64[us]
summa	float64

```
check(clean_df)
дубликаты order_id    0
soni <= 0              0
narx <= 0              0
неизвестный город     0
дата в будущем        0
пропуски (всего)      0
```

Было	Стало
10 строк, 1 дубликат	9 уникальных заказов
6 пропусков (включая скрытые)	0 пропусков
Город записан 4 способами	3 города + Неизвестно
<code>narx</code> , <code>sana</code> — текст	<code>float64</code> , <code>datetime64</code>
Цена 1 200 000	Заменена медианой 8 000
Количество —1	Заменено медианой 2

Для обсуждения

Правильно ли заменять 1 200 000 медианой? Возможно, верная цена — 12 000 (лишние нули). Если бы мы знали это наверняка, лучше было бы **исправить** значение. Очистка данных — это **набор решений**, и их нужно уметь обосновать.

10. Основные выводы

1. **Очистка — обязательный этап** анализа. GIGO: из грязных данных нельзя получить верные выводы.
2. Пропуски бывают **явными** (NaN) и **скрытыми** ("N/A", "?", 0, -1). Ищите через `isna().sum()` и `value_counts(dropna=False)`.
3. `dropna` легко уничтожает данные — проверяйте `shape`. Медиана — для чисел, мода — для категорий, `ffill/interpolate` — для рядов.
4. **Дубликаты** ищите после очистки строк.
5. `pd.to_numeric / pd.to_datetime` с `errors='coerce'` — но проверяйте, что стало NaN.
6. **Выбросы**: IQR (устойчивый) или z-оценка (чувствительна к самим выбросам). Метод находит кандидатов — решает аналитик.
7. Текст: аксессор `.str`: `strip` → `lower` → `replace` → `regex`.
8. **Проверки согласованности** и `assert` защищают от будущих ошибок.

8 распространённых ошибок новичков

1. Вызывать `df.dropna()` без параметров и терять половину таблицы
2. Заполнять нулём там, где ноль имеет реальный смысл
3. Заполнять пропуски средним **до** удаления выбросов
4. Не замечать скрытых пропусков ("`N/A`", "`-`", пустых строк)
5. Искать дубликаты до очистки пробелов и регистра
6. Использовать `errors='coerce'` и не проверять, что стало `NaN`
7. Автоматически удалять все выбросы, не разобравшись в их природе
8. Изменять исходную таблицу вместо копии (`raw.copy()`)

1. **Диагностика.** Загрузите `savdo.csv` (или датасет с Kaggle). Составьте «паспорт качества»: тип, доля пропусков, число уникальных значений, скрытые пропуски и подозрительные значения по каждому столбцу.
2. **Пропуски.** Сравните `dropna`, `fillna(mean)` и `fillna(median)` для одного столбца. Как изменились среднее, медиана и размер? Что выберете и почему?
3. **Выбросы.** Найдите выбросы методами IQR и z-оценки. Совпали ли результаты? Примените `clip` и сравните `describe()` до и после.
4. **Строки и regex.** 10 телефонных номеров в разных форматах → `+998XXXXXXXXXX`; некорректные → `NaN`.
5. **Конвейер.** Напишите `clean(df)` и `check(df)` (не менее 5 проверок). Покажите `check()` до и после очистки, добавьте `assert`.

- **pandas — Working with missing data:**
pandas.pydata.org/docs/user_guide/missing_data.html
- **pandas — Working with text data:**
pandas.pydata.org/docs/user_guide/text.html
- **Книга:** *Python for Data Analysis* (Уэс Маккинни), глава 7 «Data Cleaning and Preparation»
- **Тренажёр регулярных выражений:** regex101.com
- **Интерактивный курс:** Kaggle Learn — *Data Cleaning*

Спасибо за внимание! Вопросы?