Аналитики данных часто сталкиваются с **грязными данными,** которые могут существенно замедлить процесс анализа. Грязны данные – это *пропущенные значения*, *дубликаты*, *неконсистентные данные*. Пропущенные значения заставляют нас гадать, что же было замыслено нашим коллегой; дубликаты вводят в заблуждение, умножая одно и то же на количество их копий, а неконсистентные данные заставляют нас сомневаться в каждой цифре.

Очищать грязные данные можно c [Pandas](https://pandas.pydata.org/" \t "_blank). **Рассмотрим основные методы.**

Пропущенные значения

Пропущенные значения могут возникать по разным причинам. Вне зависимости от причины, **сначала их нужно обнаружить.**

isnull() в Pandas возвращает DataFrame или Series с булевыми значениями, где True указывает на пропущенное значение. Аналогично, notnull() возвращает True для элементов, где значение присутствует:

# удаляем строки, где есть хотя бы одно пропущенное значение  
cleaned\_df = df.dropna()  
  
# заполняем все пропущенные значения нулями  
filled\_df = df.fillna(0)  
  
print("Очищенный DataFrame:\n", cleaned\_df)  
prit("DataFrame с заполненными пропусками:\n", filled\_df)

Код выведет таблицу, где True обозначает пропущенные значения.

*- Что с ними делать дальше?* Один из подходов – **удаление строк или столбцов** с пропусками, но это может привести к потере какой-то информации. Другой подход – **заполнение пропусков.**

Для заполнения пропусков используем метод fillna():

# заполняем пропуски средним значением по столбцам  
df.fillna(df.mean(), inplace=**True**)  
  
print("df с заполненными пропусками средним:\n", df)

Для категориальных данных может быть разумно заполнить пропуски наиболее часто встречающимся значением:

# заполнение пропусков в зависимости от условий  
df['A'].fillna(value=df['A'].mean(), inplace=**True**)  
df['B'].fillna(value=df['B'].median(), inplace=**True**)  
  
# кастомная функция для заполнения  
**def** custom\_fill(series):  
 **return** series.fillna(series.mean())  
  
df.apply(custom\_fill)

Удаление с помощью dropna()

По умолчанию dropna() удаляет строки, которые содержат хотя бы одно пропущенное значение:

**import** pandas **as** pd  
**import** numpy **as** np  
  
df = pd.DataFrame({  
 'Name': ['ivan', 'artem', np.nan, 'diana', 'eva'],  
 'Age': [25, np.nan, np.nan, 27, 28],  
 'City': ['New York', 'Los Angeles', 'Chicago', np.nan, 'Miami']  
})  
  
# удаляем строки с пропущенными значениями  
df\_cleaned = df.dropna()  
print(df\_cleaned)

dropna() удаляет все строки, где есть хотя бы одно NaN.

Чтобы удалить столбцы, содержащие пропущенные значения можно юзать параметр axis=1:

df\_cleaned\_columns = df.dropna(axis=1)  
print(df\_cleaned\_columns)

Можно удалить строки, где все значения пропущены с помощью параметра how='all':

# добавляем строку, полностью состоящую из NaN  
df.loc[5] = [np.nan, np.nan, np.nan]  
  
# удаляем строки, где все значения являются NaN  
df\_cleaned\_all = df.dropna(how='all')  
print(df\_cleaned\_all)

Можно указать столбцы, для которых должна быть применена проверка на NaN, используя параметр subset.

# удаляем строки, где пропущены значения в определенных столбцах  
df\_cleaned\_subset = df.dropna(subset=['Name', 'City'])  
print(df\_cleaned\_subset)

В этом случае будут удалены только те строки, в которых отсутствуют значения в столбцах *Name* или *City*

thresh позволяет указать минимальное количество непропущенных значений, при котором строка или столбец сохраняется:

# удаляем строки, где менее двух непропущенных значений  
df\_cleaned\_thresh = df.dropna(thresh=2)  
print(df\_cleaned\_thresh)

Кастомные варианты

Часто юзаются для числовых данных, чтобы минимизировать влияние пропусков на распределение данных.

**Ср. значение:**

**import** pandas **as** pd  
**import** numpy **as** np  
  
df = pd.DataFrame({  
 'A': [1, 2, np.nan, 4, 5],  
 'B': [np.nan, 2, 3, np.nan, 5],  
 'C': [1, 2, 3, 4, np.nan]  
})  
  
# заполняем пропущенные значения в столбце A средним значением по этому столбцу  
df['A'].fillna(df['A'].mean(), inplace=**True**)

**Медиана**

Медиана может быть предпочтительнее среднего значения в случаях, когда данные содержат выбросы, которые могут исказить среднее:

# фулим пропущенные значения в столбце B медианой  
df['B'].fillna(df['B'].median(), inplace=**True**)

**Мода**

Для категориальных данных или данных с дискретными значениями можно использовать моду:

# C - категориальный столбец, заполняем пропущенные значения модой  
df['C'].fillna(df['C'].mode()[0], inplace=**True**)

Можно применять произвольные функции для более сложных заполнений. Например, можно заполнить пропуски значениями, основанными на других столбцах данных:

# Заполняем пропущенные значения в столбце A, используя кастомную логику  
df['A'].fillna(df.apply(**lambda** row: row['B'] **if** pd.isnull(row['A']) **else** row['A'], axis=1), inplace=**True**)

Когда заполнение должно учитывать группировку по какому-либо признаку, можно использовать apply() или transform() совместно с groupby():

# допустим, есть столбец группы, и мы хотим заполнить пропуски средним значением внутри каждой группы  
df['group'] = ['X', 'X', 'Y', 'Y', 'Z']  
df['A'] = df.groupby('group')['A'].transform(**lambda** x: x.fillna(x.mean()))

А что насчет дупликатов?

Дубликаты могут могут быть как полностью идентичными записями, так и частичными дубликатами, когда совпадают только некоторые поля. В любом случае, дубликаты вносят шум в данные, увеличивают их объем и могут привести к неверным аналитическим выводам.

drop\_duplicates() позволяет не только удалять полные дубликаты строк, но и предоставляет настройки для работы с частичными дубликатами, удалим полные дупликаты:

**import** pandas **as** pd  
  
df = pd.DataFrame({  
 'A': [1, 2, 2, 3, 3, 3],  
 'B': ['a', 'b', 'b', 'c', 'c', 'c'],  
 'C': [100, 200, 200, 300, 300, 300]  
})  
  
# удаляем дубликаты  
df\_unique = df.drop\_duplicates()  
  
print(df\_unique)

Код удалит все полные дубликаты строк, оставив уникальные комбинации значений по всем столбцам.

Часто нужно удалять дубликаты, основываясь не на всех столбцах, а только на некоторых. Для этого в drop\_duplicates() есть параметр subset:

# удаляем дубликаты, основываясь только на столбцах 'A' и 'B'  
df\_unique\_subset = df.drop\_duplicates(subset=['A', 'B'])  
  
print(df\_unique\_subset)

keep позволяет контролировать, какие дубликаты будут удалены: первый, последний или все:

# Удаляем все дубликаты, кроме последнего вхождения  
df\_keep\_last = df.drop\_duplicates(keep='last')  
  
print(df\_keep\_last)

Можно использовать комбинацию методов Pandas для предварительной обработки данных перед удалением дубликатов. Например, можно привести все строки к нижнему регистру, удалить пробелы, преобразовать даты в единый формат и т.д:

# преобразуем все строки в нижний регистр и удаляем пробелы  
df['B'] = df['B'].str.lower().str.replace(' ', '')  
  
# теперь удаляем дубликаты  
df\_preprocessed = df.drop\_duplicates()  
  
print(df\_preprocessed)

Категориальные данные

get\_dummies() используется для преобразования категориальных переменных в фиктивные/индикаторные переменные, кстати в ml юзается довольно часто:

**import** pandas **as** pd  
  
df = pd.DataFrame({  
 'Color': ['Red', 'Green', 'Blue', 'Green']  
})  
  
# преобразуем категориальные данные в индикаторные переменные  
dummies = pd.get\_dummies(df['Color'])  
print(dummies)

В результате каждое уникальное значение в столбце Color превращается в отдельный столбец с индикаторными переменными (0 или 1), указывающими на присутствие или отсутствие данной категории в каждой строке.

factorize() используется для получения числового представления категориальных данных, присваивая каждой уникальной категории целочисленный идентификатор:

# получаем числовое представление категориальных данных  
codes, uniques = pd.factorize(df['Color'])  
  
print(codes) # массив кодов  
print(uniques) # массив уникальных значений

str-методы позволяют выполнять такие операции, как преобразование регистра, поиск и замена подстрок, разбиение строк и т.п:

df\_text = pd.DataFrame({  
 'Text': ['This is a test.', 'Hello, World!', 'Otus!']  
})  
  
# преобразуем все тексты в нижний регистр  
df\_text['Text\_lower'] = df\_text['Text'].str.lower()  
  
# удаляем знаки пунктуации  
df\_text['Text\_clean'] = df\_text['Text'].str.replace('[^\w\s]', '', regex=**True**)  
  
print(df\_text)

Иногда можно использовать str.replace(), str.findall().

Преобразование текстовых данных в числовые часто требуется для аналитических и ml моделей. Например, после использования factorize() для категориальных данных можно обратно преобразовать числовые коды в текстовые категории, используя массив uniques:

# преобразование кодов обратно в текстовые категории  
df['Color\_restored'] = codes  
df['Color\_restored'] = df['Color\_restored'].map(**lambda** x: uniques[x])  
  
print(df)

## Как очистить данные: пошаговое руководство

Чтобы очистить данные, сперва их нужно правильно загрузить. В этом руководстве мы покажем базовые методы загрузки данных из CSV-файла. Больше вариантов чтения CSV вы найдете в [документации](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html).

**От редакции Pythonist.**На нашем сайте есть статья по этой теме — [«Чтение файлов в формате CSV в Python»](https://pythonist.ru/chtenie-formata-csv-v-python/).

**import pandas as pd**

**# 1. Чтение данных из csv - способ по умолчанию**

**df = pd.read\_csv('my\_file.csv')**

**# 2. Чтение данных из csv с использованием запятой в качестве разделителя**

**df = pd.read\_csv('my\_file.csv', delimiter=',')**

**# 3. Чтение данных из csv с использованием запятой в качестве разделителя и без заголовков**

**df = pd.read\_csv('my\_file.csv', delimiter=',', header=None)**

**# 4. Чтение данных из csv с использованием запятой в качестве разделителя и с пользовательскими заголовками**

**my\_headers = ['Id','Name', 'Type', 'Price']**

**df = pd.read\_csv('my\_file.csv', delimiter=',', header=0, names=my\_headers)**

### Удаление дублирующихся данных

Чтобы очистить данные, некоторые шаги нужно сделать обязательно. Один из таких шагов — удаление дубликатов. И тут не важно, идет речь о текстовых или числовых данных. Если у вас слишком много дубликатов, это увеличивает время обработки данных.

**# 1. Удаление дубликатов и возврат копии датафрейма**

**df = df.drop\_duplicates()**

**# 2. Удаление дубликатов в исходном датафрейме**

**df = df.drop\_duplicates(inplace=True)**

**# 3. Отбрасываем дубликаты, оставляя первое/последнее вхождение**

**df = df.drop\_duplicates(inplace=True, keep='last')**

**# 4. Для нахождения дубликатов учитываем только определенные столбцы**

**df = df.drop\_duplicates(subset=['Id', 'Price'], inplace=True, keep='last')**

### Удаление эмодзи

Зачастую нам не нужны эмодзи в текстовых наборах данных. Мы можем их удалить всего одной строчкой кода. Сниппет, приведенный ниже, будет удалять эмодзи из датафрейма pandas столбец за столбцом. Этот сниппет можно найти на [Stackoverflow](https://stackoverflow.com/questions/57514169/how-can-i-remove-emojis-from-a-dataframe/57514515" \l "57514515" \t "_blank).

**f = df.astype(str).apply(lambda x: x.str.encode('ascii', 'ignore').str.decode('ascii'))**

education.yandex.ru

Здесь мы переводим все данные в значения ASCII, а те, которые не могут быть переведены, игнорируем. После перевода в значения ASCII наш код переводит данные обратно. Так мы получаем все наши данные без эмодзи.

### Перевод данных в нижний регистр

Весьма вероятно, что вам придется изменить регистр данных. Здесь мы переводим их в нижний. Больше примеров можно найти в [документации](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.lower.html).

**df['Type'] = df['Type'].str.lower()**

**df['Name'] = df['Name'].str.lower()**

### Удаление множественных пробелов, табов и символов перевода строки

В каждом наборе данных присутствуют ненужные пробелы, табы и переходы на новую строку. Проблема в том, что табы и абзацы хорошо видно, тогда как пробелы можно не заметить, а они повлияют на обучение моделей.

**df['Type'] = df['Type'].str.replace('\n', '')**

**df['Type'] = df['Type'].str.replace('\t', ' ')**

**df['Type'] = df['Type'].str.replace(' {2,}', ' ', regex=True)**

**df['Type'] = df['Type'].str.strip()**

Первые две строки кода заменяют табы и символы перевода строки на пустые строки. Третья строка ищет два и более пробелов при помощи регулярного выражения, а затем заменяет их на один пробел. Наконец, последняя строка обрезает данные с обоих концов, удаляя пробелы.

### Удаление URL-адресов

Для получения данных многие пользуются опросами. Но люди невнимательно заполняют поля, и порой в этих данных встречаются URL-адреса. Регулярное выражение, приведенное ниже, удаляет URL. Вы можете использовать и любой другой regex-паттерн для поиска URL. Здесь найденные адреса заменяются пустой строкой.

**df['Type'] = df['Type'].replace(r'http\S+', '', regex=True).replace(r'www\S+', '', regex=True)**

### Отбрасываем строки с пустыми данными

После выполнения предыдущих шагов по очистке в столбцах остаются пустые данные. От этих пустых строк нужно избавиться, иначе это создаст неопределенность при обучении модели. Чтобы удалить все строки с пустыми данными, мы используем два метода:

**df.dropna()**

**df['Type'].astype(bool)**

**df = df[df['Type'].astype(bool)]**

Первая строка кода удаляет все строки, содержащие np.nan, pd.NaT и None. Остальные удаляют строки, содержащие символы пустой строки. Второй метод быстрый, но если в столбце будет хотя бы пробел, он не сработает. Это еще одна причина обрезать лишние пробелы заранее.

### Дальнейшая обработка данных

Иногда, чтобы очистить данные, бывает нужно отбросить некоторые столбцы, создать новый столбец из существующих данных или удалить строки, не содержащие определенных данных.

**import numpy as np**

**df = df.drop(['Id', 'Name'], axis=1)**

**df = df[df['Type'].str.contains('frozen') | df['Type'].str.contains('green')]**

**def detect\_price(row):**

**if row['Price'] > 15.50:**

**return 'High'**

**elif row['Price'] > 5.50 and row['Price'] <= 15.50:**

**return 'Medium'**

**elif row['Price'] > 0.0 and row['Price'] <= 5.50:**

**return 'Low'**

**else:**

**return np.NaN**

**df['Range'] = df.apply (lambda row: detect\_price(row), axis=1)**

Здесь в третьей строке мы отбрасываем два столбца с именами Id и Name и возвращаем копию нового датафрейма.

Четвертая строка проверяет, содержит ли столбец Type строку frozen или green, затем возвращает True и сохраняет эту строку.

Строки с 7 по 17 создают новый столбец с именем Range на основе данных столбца Price. Используя [лямбда-функцию](https://pythonist.ru/lyambda-funkczii-v-python-primery/), мы передаем каждую строку в функцию detect\_price и возвращаем значение на основе цены. Затем возвращаемое значение присваивается новому столбцу в строке, переданной в функцию. Мы используем np.NaN, чтобы потом иметь возможность удалить эти строки при помощи df.dropna().

[**Главная страница**](https://pythonist.ru/) » [**Вопросы с собеседования**](https://pythonist.ru/category/voprosy-s-sobesedovaniya-python/) » [**Функции**](https://pythonist.ru/category/voprosy-s-sobesedovaniya-python/funkczii/) » Лямбда-функции в Python: примеры

# Лямбда-функции в Python: примеры

[**Функции**](https://pythonist.ru/category/voprosy-s-sobesedovaniya-python/funkczii/)

Лямбда-функции – это анонимные функции, которые содержат только одно выражение.

Вы можете подумать, что лямбда-функции – это что-то сложное, доступное только продвинутым программистам. Но из этой статьи вы узнаете, как легко начать использовать их в своём коде.

В Python мы обычно создаём [функции](https://pythonist.ru/funkczii-i-ih-argumenty-v-python-3/) так:

**def my\_func(a):**

**# Тело функции**

Они объявляются при помощи ключевого слова def, им можно давать название, добавлять список аргументов в скобках. В теле функции может быть много строчек кода, с любым количеством выражений. Но иногда вам может понадобиться функция с одним-единственным выражением внутри.

Например, функция, удваивающая аргумент:

**def double(x):**

**return x\*2**

Такую функцию, например, можно использовать в методе map.

**def double(x):**

**return x\*2**

**my\_list = [1, 2, 3, 4, 5, 6]**

**new\_list = list(map(double, my\_list))**

**print(new\_list) # [2, 4, 6, 8, 10, 12]**

Именно здесь целесообразно использовать лямбда-функцию, т.к. её можно создать прямо в том же месте, где она используется. А это означает меньше строк кода. Кроме того, нам не придётся создавать функцию с именем, которая будет занимать место в памяти и пригодится лишь однажды.

## Как использовать лямбда-функции в Python

Как вы уже поняли, лямбды применяются там, где нужна маленькая функция на короткое время – например, в функциях высшего порядка, таких как map или filter.

Синтаксис лямбда-функции имеют следующий:

**lambda args: expression**

Сперва пишется слово lambda, затем идет одинарный пробел, за ним — список аргументов через запятую, следом – двоеточие и само выражение, являющееся телом функции.

Обратите внимание, что лямбда-функциям нельзя дать имя, потому что они анонимные (без имени) по определению.

Аргументов в них может быть сколько угодно, но выражение в теле обязательно одно.

[python\_ad\_block]

### Пример 1

Можно написать лямбду, которая удваивает свой аргумент: lambda x: x\*2, и использовать её в функции map, чтобы удвоить все элементы в списке:

**my\_list = [1, 2, 3, 4, 5, 6]**

**new\_list = list(map(lambda x: x\*2, my\_list))**

**print(new\_list) # [2, 4, 6, 8, 10, 12]**

Заметьте разницу между этим кодом и функцией double, которую мы написали выше: лямбда гораздо компактнее.

### Пример 2

А ещё можно создать лямбда-функцию, которая ищет числа больше нуля: lambda x: x > 0 и использовать в filter, чтобы создать список исключительно положительных чисел.

**my\_list = [18, -3, 5, 0, -1, 12]**

**new\_list = list(filter(lambda x: x > 0, my\_list))**

**print(new\_list) # [18, 5, 12]**

Лямбда-функция определяется там, где используется. Таким образом, в памяти не придётся хранить функцию с именем. Поэтому для одноразового использования целесообразно писать лямбда-функции, чтобы избежать загромождения.

Несколько способов **очистить данные от шума** в Python:

* **Использовать фильтры**. [2](https://www.geeksforgeeks.org/noise-removal-using-lowpass-digital-butterworth-filter-in-scipy-python/) Для этого можно применить функцию lfilter из библиотеки scipy.signal. [1](https://stackoverflow.com/questions/37598986/reducing-noise-on-data)
* **Применить метод LOWESS**. Для этого нужно использовать реализацию statsmodels. Пример кода: [1](https://stackoverflow.com/questions/37598986/reducing-noise-on-data)

import statsmodels.api as sm

y\_lowess = sm.nonparametric.lowess(y, x, frac = 0.3) # 30% сглаживание LOWESS [1](https://stackoverflow.com/questions/37598986/reducing-noise-on-data)

plt.plot(y\_lowess[:, 0], y\_lowess[:, 1], 'b') # часть шума удалена [1](https://stackoverflow.com/questions/37598986/reducing-noise-on-data)

plt.show()

Скопировать

* **Использовать среднее скользящее**. Например, с помощью pandas с размером окна 30: [1](https://stackoverflow.com/questions/37598986/reducing-noise-on-data)

import pandas as pd

df = pd.DataFrame(y, x)

df\_mva = df.rolling(30).mean() # скользящее среднее с размером окна 30 [1](https://stackoverflow.com/questions/37598986/reducing-noise-on-data)

df\_mva.plot(legend = False)

Скопировать

* **Применить интерполяцию**. Для снижения шума через сглаживание можно использовать интерполяцию с радиальной функцией основы. [1](https://stackoverflow.com/questions/37598986/reducing-noise-on-data)

Выбор метода зависит от типа данных и задачи очистки.