Published in Towards Data Science

Amanda Iglesias Moreno Follow

Sep 16, 2020 · 13 min read · ✨ · Listen

Data filtering in Pandas

The complete guide to clean data sets — Part 3

Filtering data from a data frame is one of the most common operations when cleaning the data. Pandas provides a wide range of methods for selecting data according to the position and label of the rows and columns. In addition, Pandas also allows you to obtain a subset of data based on column types and to filter rows with boolean indexing.

In this article, we will cover the most common operations for selecting a subset of data from a Pandas data frame: (1) selecting a single column by label, (2) selecting multiple columns by label, (3) selecting columns by data type, (4) selecting a single row by label, (5) selecting multiple rows by label, (6) selecting a single row by position, (7) selecting multiple rows by position, (8) selecting rows and columns simultaneously, (9) selecting a scalar value, and (10) selecting rows using Boolean selection.

Additionally, we will provide multiple coding examples! Now, let's get started :) ❤




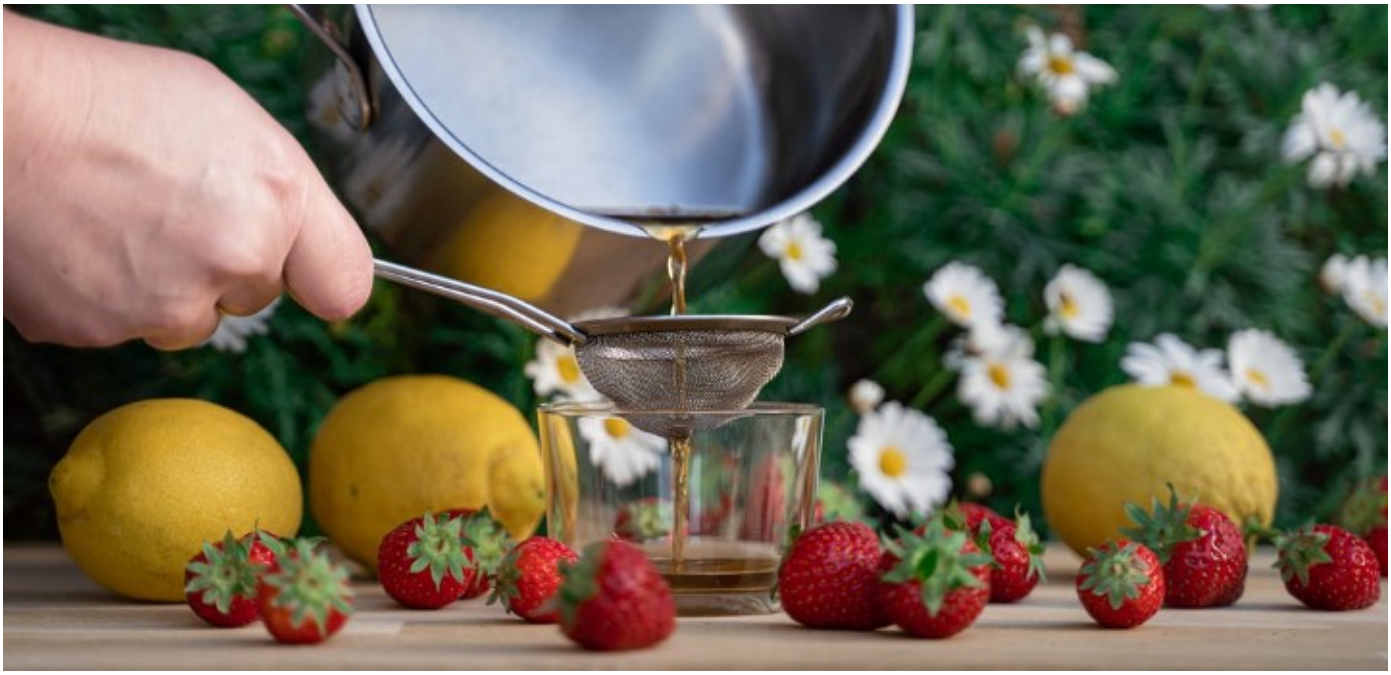

Photo by Daphné Be Frenchie on Unsplash

Data set

In this article, we use a small data set for learning purposes. In the real world, the data sets employed will be much larger; however, the procedures used to filter the data remain the same.

The data frame contains information about 10 employees of a company: (1) id, (2) name, (3) surname, (4) division, (5) telephone, (6) salary, and (7) type of contract.

```
import pandas as pd

# information about employees
id_number = ['128', '478', '257', '299', '175', '328', '099', '457', '144', '222']
name = ['Patrick', 'Amanda', 'Antonella', 'Eduard', 'John', 'Alejandra', 'Layton', 'Melani
surname = ['Miller', 'Torres', 'Brown', 'Iglesias', 'Wright', 'Campos', 'Platt', 'Cavill',
division = ['Sales', 'IT', 'IT', 'Sales', 'Marketing', 'Engineering', 'Engineering', 'Sale
salary = [30000, 54000, 80000, 79000, 15000, 18000, 30000, 35000, 45000, 30500]
telephone = ['7366578', '7366444', '7366120', '7366574', '7366113', '7366117', '7366777',
type_contract = ['permanent', 'temporary', 'temporary', 'permanent', 'internship', 'interr

```

df_employees.py hosted with ❤ by GitHub view raw

	name	surname	division	salary	telephone	type_contract
128	Patrick	Miller	Sales	30000	7366578	permanent
478	Amanda	Torres	IT	54000	7366444	temporary
257	Antonella	Brown	IT	80000	7366120	temporary
299	Eduard	Iglesias	Sales	79000	7366574	permanent
175	John	Wright	Marketing	15000	7366113	internship
328	Alejandra	Campos	Engineering	18000	7366117	internship
099	Layton	Platt	Engineering	30000	7366777	permanent
457	Melanie	Cavill	Sales	35000	7366579	temporary
144	David	Lange	Engineering	45000	7366441	permanent
222	Lewis	Bellow	Sales	30500	7366440	permanent

1. Selecting a single column by label

To select **a single column** in Pandas, we can use both the **. operator** and the **[] operator.**

> Selecting a single column by label
>
> → df[string]

The following code access the salary column using both methods (dot notation and square braces).

```
# select the column (salary) using dot notation
salary = df_employees.salary

# select the column (salary) using square brackets
```

```
# <class 'pandas.core.series.Series'>

print(type(salary_2))
# <class 'pandas.core.series.Series'>

salary
```

single_column_label.py hosted with ❤ by GitHub view raw

```
128    30000
478    54000
257    80000
299    79000
175    15000
328    18000
099    30000
457    35000
144    45000
222    30500
Name: salary, dtype: int64
```

As shown above, when **a single column** is retrieved, the result is a **Series** object. To obtain a **DataFrame** object when selecting only one column, we need to pass in a list with a single item instead of just a string.

```
# obtain a Series object by passing in a string to the indexing operator
df_employees['salary']

# obtain a DataFrame object by passing a list with a single item to the indexing operator
df_employees[['salary']]
```

single_column_label_dataframe.py hosted with ❤ by GitHub view raw

478	54000
257	80000
299	79000
175	15000
328	18000
099	30000
457	35000
144	45000
222	30500

Besides, it is important to bear in mind that we can not use dot notation to access a specific column of a data frame when the column name contains spaces. If we do it, a **SyntaxError** is raised.

2. Selecting multiple columns by label

We can select **multiple columns** of a data frame by passing in a list with the column names as follows.

Selecting multiple columns by label

→ df[list_of_strings]

```
# select multiple columns by passing in a list with the column names to the indexing operat
df_employees[['division', 'salary']]
```

multiple_columns_label.py hosted with ❤ by GitHub view raw

478	IT	54000
257	IT	80000
299	Sales	79000
175	Marketing	15000
328	Engineering	18000
099	Engineering	30000
457	Sales	35000
144	Engineering	45000
222	Sales	30500

As shown above, the result is a **DataFrame** object containing only the columns provided in the list.

3. Selecting columns by data type

We can use the **pandas.DataFrame.select_dtypes(include=None, exclude=None)** method to select columns based on their data types. The method accepts either a list or a single data type in the parameters **include** and **exclude**. It is important to keep in mind that at least one of these parameters (include or exclude) must be supplied and they must not contain overlapping elements.

> Selecting columns by data type
>
> → df.select_dtypes(include=None, exclude=None)

In the example below, we select the numeric columns (both integers and floats) of the data frame by passing in the **np.number** object to the **include** parameter. Alternatively, we can obtain the same results by providing the string '**number**' as input.

As you can observe, the **select_dtypes() method** returns a **DataFrame** object including the dtypes in the **include parameter** and excluding the dtypes in the **exclude**

```
# select numeric columns - numpy object
numeric_inputs = df_employees.select_dtypes(include=np.number)

# check selected columns with the .columns attribute
numeric_inputs.columns
# Index(['salary'], dtype='object')

# the method returns a DataFrame object
print(type(numeric_inputs))
# <class 'pandas.core.frame.DataFrame'>

# select numeric columns - string
numeric_inputs_2 = df_employees.select_dtypes(include='number')

# check selected columns with the .columns attribute
numeric_inputs_2.columns
# Index(['salary'], dtype='object')

# the method returns a DataFrame object
print(type(numeric_inputs_2))
# <class 'pandas.core.frame.DataFrame'>

# visualize the data frame
numeric_inputs
```

columns_data_type.py hosted with ❤ by **GitHub** **view raw**

	salary
128	30000
478	54000
257	80000
299	79000
175	15000
328	18000
099	30000
457	35000
144	45000

types in Pandas.

Numpy object	String	Selection
np.number	number	Select both integers and floats
np.floating	floating	Select floats
np.integer	integer	Select integers
np.datetime64	datetime	Select datetimes
np.timedelta64	timedelta	Select timedeltas
np.int8	int8	Select 8-bit integers
np.int16	int16	Select 16-bit integers
np.int32	int32	Select 32-bit integers
np.in64	int64	Select 64-bit integers
np.float16	float16	Select 16-bit floats
np.float32	float32	Select 32-bit floats
np.float64	float64	Select 64-bit floats
np.float128	float128	Select 128-bit floats
np.object	object	Select objects

As a reminder, we can check the data types of the columns using **pandas.DataFrame.info** method or with **pandas.DataFrame.dtypes** attribute. The former prints a concise summary of the data frame, including the column names and their data types, while the latter returns a **Series** with the data type of each column.

```
1  # concise summary of the data frame, including the column names and their data types
2  df_employees.info()
```

employees_info.py hosted with ❤ by **GitHub** **view raw**




```
name             10 non-null object
surname          10 non-null object
division         10 non-null object
salary           10 non-null int64
telephone        10 non-null object
type contract    10 non-null object
dtypes: int64(1), object(5)
memory usage: 320.0+ bytes
```

```
# check the data types of the columns
df_employees.dtypes
```

employees_dtypes.py hosted with ❤ by GitHub view raw

```
name             object
surname          object
division         object
salary            int64
telephone        object
type contract    object
dtype: object
```

4. Selecting a single row by label

DataFrames and **Series** do not necessarily have numerical indexes. By default, the index is an integer indicating the row position; however, it can also be an alphanumeric string. In our current example, the index is the id number of the employee.

```
# we can check the indexes of the data frame using the .index method
df_employees.index
# Index(['128', '478', '257', '299', '175', '328', '099', '457', '144', '222'], dtype='obje
# the index is the id number of the employee (categorical variable - type object)
```

employee_index.py hosted with ❤ by GitHub view raw

→ df.loc[string]

The code below shows how to select the employee with id number 478.

```
# select the employee with id number 478 with the .loc[] indexer
df_employees.loc['478']
```

single_row_label.py hosted with ❤ by GitHub view raw

```
name              Amanda
surname           Torres
division              IT
salary             54000
telephone        7366444
type_contract    temporary
Name: 478, dtype: object
```

As shown above, when a single row is selected, the **.loc[] indexer** returns a **Series object.** However, we can also obtain a single-row **DataFrame** by passing a single-element list to the **.loc[]** method as follows.

```
# select the employee with id number 478 with the .loc[] indexer, providing a single-elemer
df_employees.loc[['478']]
```

single_row_label_dataframe.py hosted with ❤ by GitHub view raw

	name	surname	division	salary	telephone	type_contract
478	Amanda	Torres	IT	54000	7366444	temporary

5. Selecting multiple rows by label

We can select **multiple rows** with the **.loc[] indexer**. Besides a single label, the indexer

→ df.loc[list_of_strings]

→ df.loc[slice_of_strings]

Next, we obtain a subset of our data frame containing the employees with id number 478 and 222 as follows.

222	Lewis	Bellow	Sales	30500	7366440	permanent

Notice that, the end index of **.loc[] method** is always included, meaning the selection includes the last label.

6. Selecting a single row by position

The **.iloc[] indexer** is used to index a data frame by position. To select a single row with the .iloc[] attribute, we pass in the row position (a single integer) to the indexer.

> Selecting a single row by position
>
> → df.iloc[integer]

In the following block of code, we select the row with index 0. In this case, the first row of the DataFrame is returned because in Pandas indexing starts at 0.




```
name              Patrick
surname            Miller
division            Sales
salary              30000
telephone         7366578
type_contract    permanent
Name: 128, dtype: object
```

Additionally, the **.iloc[] indexer** also supports **negative integers** (starting at -1) as relative positions to the end of the data frame.




```
name                Lewis
surname            Bellow
division            Sales
salary              30500
telephone         7366440
type_contract    permanent
Name: 222, dtype: object
```

As shown above, when a single row is selected, the **.iloc[] indexer** returns a **Series** object that has the column names as indexes. However, as we did with the **.loc[] indexer**, we can also obtain a **DataFrame** by passing a single-integer list to the indexer in the following way.




	name	surname	division	salary	telephone	type_contract
222	Lewis	Bellow	Sales	30500	7366440	permanent

Lastly, keep in mind that an **IndexError** is raised when trying to access an index that is out-of-bounds.




7. Selecting multiple rows by position

To extract multiple rows by position, we pass either a list or a slice object to the **.iloc[]** indexer.

> Selecting multiple rows by position
>
> → df.iloc[list_of_integers]
>
> → df.iloc[slice_of_integers]

The following block of code shows how to select the first five rows of the data frame using a list of integers.




	name	surname	division	salary	telephone	type_contract
128	Patrick	Miller	Sales	30000	7366578	permanent
478	Amanda	Torres	IT	54000	7366444	temporary
257	Antonella	Brown	IT	80000	7366120	temporary
299	Eduard	Iglesias	Sales	79000	7366574	permanent
175	John	Wright	Marketing	15000	7366113	internship

Alternatively, we can obtain the same results using slice notation.




	name	surname	division	salary	telephone	type_contract
128	Patrick	Miller	Sales	30000	7366578	permanent
478	Amanda	Torres	IT	54000	7366444	temporary
257	Antonella	Brown	IT	80000	7366120	temporary
299	Eduard	Iglesias	Sales	79000	7366574	permanent
175	John	Wright	Marketing	15000	7366113	internship

As shown above, Python slicing rules (half-open interval) apply to the **.iloc[] attribute**, meaning the first index is included, but not the end index.

8. Selecting rows and columns simultaneously

So far, we have learnt how to select rows in a data frame by label or position using the **.loc[]** and **.iloc[] indexers**. However, both indexers are not only capable of selecting rows, but also rows and columns simultaneously.

To do so, we have to provide the row and column labels/positions separated by a comma as follows:




→ df.loc[row_labels, column_labels]

→ df.iloc[row_positions, column_positions]

where **row_labels** and **column_labels** can be a single string, a list of strings, or a slice of strings. Likewise, **row_positions** and **column_positions** can be a single integer, a list of integers, or a slice of integers.

The following examples show how to extract rows and columns at once using the **.loc[]** and **.iloc[]** indexers.

- **Selecting a scalar value**

We select the salary of the employee with the id number 478 by position and label in the following manner.




In this case, the output of both indexers is an integer.

- **Selecting a single row and multiple columns**

We select the name, surname, and salary of the employee with id number 478 by passing a single value as the first argument and a list of values as the second argument, obtaining as a result a Series object.




```
name       Amanda
surname    Torres
salary      54000
Name: 478, dtype: object
```

- **Selecting disjointed rows and columns**

To select multiple rows and columns, we need to pass two list of values to both indexers. The code below shows how to extract the name, surname, and salary of employees with id number 478 and 222.




	name	surname	salary
478	Amanda	Torres	54000
222	Lewis	Bellow	30500

Unlike before, the output of both indexers is a DataFrame object.

- **Selecting continuous rows and columns**

We can extract continuous rows and columns of the data frame by using slice notation. The following code snippet shows how to select the name, surname, and salary of employees with id number 128, 478, 257, and 299.




	name	surname	salary
128	Patrick	Miller	30000
478	Amanda	Torres	54000
257	Antonella	Brown	80000
299	Eduard	Iglesias	79000

As shown above, we only employ slice notation to extract the rows of the data frame since the id numbers we want to select are continuous (indexes from 0 to 3).

It is important to remember that the **.loc[] indexer** uses a **closed interval**, extracting both the start label and the stop label. On the contrary, the **.iloc[] indexer** employs a **half-open interval**, so the value at the stop index is not included.

9. Selecting a scalar value using the .at[] and .iat[] indexers

As mentioned above, we can select a scalar value by passing two strings/integers separated by a comma to the **.loc[]** and **.iloc[] indexers.** Additionally, Pandas provides two optimized functions to extract a scalar value from a data frame object: the **.at[]** and **.iat[] operators**. The former extracts a single value by label, while the latter access a single value by position.

Selecting a scalar value by label and position

- df.at[string, string]

The code below shows how to select the salary of the employee with the id number 478 by label and position with the **.at[]** and **.iat[] indexers**.

We can use the **%timeit magic function** to calculate the execution time of both Python statements. As shown below, the **.at[]** and **.iat[] operators** are much faster than the




```
12.5 µs ± 46 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
7.84 µs ± 22.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```




```
15.4 µs ± 765 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
9.63 µs ± 167 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Lastly, it is important to remember that the **.at[]** and **.iat[]** indexers can only be used to access a single value, raising a type error when trying to select multiple elements of the data frame.




10. Selecting rows using Boolean selection

So far, we have filtered rows and columns in a data frame by label and position. Alternatively, we can also select a subset in Pandas with boolean indexing. Boolean selection consists of selecting rows of a data frame by providing a boolean value (True or False) for each row.

In most cases, this array of booleans is calculated by applying to the values of a single or multiple columns a condition that evaluates to True or False, depending on whether or not the values meet the condition. However, it is also possible to manually create an array of booleans using among other sequences, Numpy arrays, lists, or Pandas Series.

Then, the sequence of booleans is placed inside square brackets [], returning the rows associated with a True value.




Boolean selection according to the values of a single column

The most common way to filter a data frame according to the values of a single column is by using a comparison operator.

A comparison operator evaluates the relationship between two operands (a and b) and returns True or False depending on whether or not the condition is met. The following table contains the comparison operators available in Python.

Operator	Description
==	True if a is equal to b else False
!=	True if a is not equal to b else False
>	True if a is greater than b else False
<	True if a is less that b else False
>=	True if a is greater than or equal to b else False
<=	True if a is less than or equal to b else False

These comparison operators can be used on a single column of the data frame to obtain a sequence of booleans. For instance, we determine whether the salary of the employee is greater than 45000 euros by using the greater than operator as follows.




```
128    False
478     True
257     True
299     True
175    False
328    False
099    False
457    False
144    False
222    False
Name: salary, dtype: bool
```

The output is a Series of booleans where salaries higher than 45000 are True and those less than or equal to 45000 are False. As you may notice, the Series of booleans has the same indexes (id number) as the original data frame.

This Series can be passed to the indexing operator [] to return only the rows where the result is True.




	name	surname	division	salary	telephone	type_contract
478	Amanda	Torres	IT	54000	7366444	temporary
257	Antonella	Brown	IT	80000	7366120	temporary
299	Eduard	Iglesias	Sales	79000	7366574	permanent

As shown above, we obtain a data frame object containing only the employees with a salary higher than 45000 euros.

Boolean selection according to the values of multiple columns

Previously, we have filtered a data frame according to a single condition. However, we can also combine multiple boolean expression together using logical operators. In Python, there are three logical operators: and, or, and not. However, these keywords are not available in Pandas for combining multiple boolean conditions. Instead, the




|	if one of the operands is True than it returns True	or
~	it returns True if the operand is False	not

The code below shows how to select employees with a salary greater than 45000 and a permanent contract combining two boolean expressions with the logical operator &.

As you may know, in Python, the comparison operators have a higher precedence than the logical operators. However, it does not apply to Pandas where logical operators have higher precedence than comparison operators. Therefore, we need to wrap each boolean expression in parenthesis to avoid an error.

Boolean selection using Pandas methods

Pandas provides a wide range of built-in functions that return a sequence of booleans, being an appealing alternative to more complex boolean expressions that combine comparison and logical operators.

- **The isin method**

The **pandas.Series.isin** method takes a sequence of values and returns True at the positions within the Series that match the values in the list.

This method allows us to check for the presence of one or more elements within a column without using the logical operator or. The code below shows how to select employees with a permanent or temporary contract using both the logical operator or and the isin method.




	name	surname	division	salary	telephone	type_contract
128	Patrick	Miller	Sales	30000	7366578	permanent
478	Amanda	Torres	IT	54000	7366444	temporary
257	Antonella	Brown	IT	80000	7366120	temporary
299	Eduard	Iglesias	Sales	79000	7366574	permanent
099	Layton	Platt	Engineering	30000	7366777	permanent
457	Melanie	Cavill	Sales	35000	7366579	temporary
144	David	Lange	Engineering	45000	7366441	permanent
222	Lewis	Bellow	Sales	30500	7366440	permanent

As you can see, the isin method comes in handy for checking multiple or conditions in the same column. Additionally, it is faster!




```
1.71 ms ± 140 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
1 ms ± 200 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

- **The between method**

The **pandas.Series.between** method takes two scalars separated by a comma which represent the lower and upper boundaries of a range of values and returns True at the positions that lie within that range.

The following code selects employees with a salary higher than or equal to 30000 and less than or equal to 80000 euros.




	name	surname	division	salary	telephone	type_contract
128	Patrick	Miller	Sales	30000	7366578	permanent
478	Amanda	Torres	IT	54000	7366444	temporary
257	Antonella	Brown	IT	80000	7366120	temporary
299	Eduard	Iglesias	Sales	79000	7366574	permanent
099	Layton	Platt	Engineering	30000	7366777	permanent
457	Melanie	Cavill	Sales	35000	7366579	temporary
144	David	Lange	Engineering	45000	7366441	permanent
222	Lewis	Bellow	Sales	30500	7366440	permanent

As you can observe, both boundaries (30000 and 80000) are included. To exclude them, we have to pass the argument **inclusive=False** in the following manner.




	name	surname	division	salary	telephone	type_contract
478	Amanda	Torres	IT	54000	7366444	temporary
299	Eduard	Iglesias	Sales	79000	7366574	permanent
457	Melanie	Cavill	Sales	35000	7366579	temporary
144	David	Lange	Engineering	45000	7366441	permanent
222	Lewis	Bellow	Sales	30500	7366440	permanent

As you may noticed, the above code is equivalent to writing two boolean expressions and evaluate them using the logical operator and.




- **String methods**

Additionally, we can also use boolean indexing with string methods as long as they return a sequence of booleans.

For instance, the **pandas.Series.str.contains** method checks for the presence of a substring in all the elements of a column and returns a sequence of booleans that we can pass to the indexing operator to filter a data frame.

	name	surname	division	salary	telephone	type_contract
128	Patrick	Miller	Sales	30000	7366578	permanent
299	Eduard	Iglesias	Sales	79000	7366574	permanent
457	Melanie	Cavill	Sales	35000	7366579	temporary

While the **contains** method evaluates whether or not a substring is contained in each element of a Series, the **pandas.Series.str.startswith** function checks for the presence




The following code shows how to select employees whose name starts with 'A'.

	name	surname	division	salary	telephone	type_contract
478	Amanda	Torres	IT	54000	7366444	temporary




data from a Pandas data frame. Additionally, we have provided multiple usage examples. Now! it is the time to put in practice those techniques when cleaning your own data! ✋

Besides data filtering, the data cleaning process involves many more operations. If you are still interested in knowing more about data cleaning, take a look at these articles.

Data normalization with Pandas and Scikit-Learn
The complete guide to clean datasets — Part 1
towardsdatascience.com

Identify Outliers With Pandas, Statsmodels, and Seaborn
The complete guide to clean data sets — Part 2
medium.com

Thanks for reading

Amanda ❤