Copy of page at:<https://realpython.com/pandas-python-explore-dataset/>

**Using Pandas and Python to Explore Your Dataset**

by Reka Horvath Jan 06, 2020

Table of Contents

* Setting Up Your Environment
* Using the Pandas Python Library
* Getting to Know Your Data
  + Displaying Data Types
  + Showing Basics Statistics
  + Exploring Your Dataset
* Getting to Know Pandas’ Data Structures
  + Understanding Series Objects
  + Understanding DataFrame Objects
* Accessing Series Elements
  + Using the Indexing Operator
  + Using .loc and .iloc
* Accessing DataFrame Elements
  + Using the Indexing Operator
  + Using .loc and .iloc
* Querying Your Dataset
* Grouping and Aggregating Your Data
* Manipulating Columns
* Specifying Data Types
* Cleaning Data
  + Missing Values
  + Invalid Values
  + Inconsistent Values
* Combining Multiple Datasets
* Visualizing Your Pandas DataFrame
* Conclusion

Do you have a large dataset that’s full of interesting insights, but you’re not sure where to start exploring it? Has your boss asked you to generate some statistics from it, but they’re not so easy to extract? These are precisely the use cases where **Pandas** and Python can help you! With these tools, you’ll be able to slice a large dataset down into manageable parts and glean insight from that information.

**In this tutorial, you’ll learn how to:**

* **Calculate** metrics about your data
* **Perform** basic queries and aggregations
* **Discover** and handle incorrect data, inconsistencies, and missing values
* **Visualize** your data with plots

You’ll also learn about the differences between the main data structures that Pandas and Python use.

**Setting Up Your Environment**

There are a few things you’ll need to get started with this tutorial. First is a familiarity with Python’s built-in data structures, especially lists and dictionaries. For more information, check out Lists and Tuples in Python and Dictionaries in Python.

The second thing you’ll need is a working Python environment. You can follow along in any terminal that has Python 3 installed. If you want to see nicer output, especially for the large NBA dataset you’ll be working with, then you might want to run the examples in a Jupyter notebook.

**Note:** If you don’t have Python installed at all, then check out Python 3 Installation & Setup Guide. You can also follow along online in a try-out Jupyter notebook.

The last thing you’ll need is the Pandas Python library, which you can install with pip:

$ python -m pip install pandas

You can also use the Conda package manager:

$ conda install pandas

If you’re using the Anaconda distribution, then you’re good to go! Anaconda already comes with the Pandas Python library installed.

**Note:** Have you heard that there are multiple **package managers** in the Python world and are somewhat confused about which one to pick? pip and conda are both excellent choices, and they each have their advantages.

If you’re going to use Python mainly for data science work, then conda is perhaps the better choice. In the conda ecosystem, you have two main alternatives:

1. If you want to get a stable data science environment up and running quickly, and you don’t mind downloading 500 MB of data, then check out the Anaconda distribution.
2. If you prefer a more minimalist setup, then check out the section on installing Miniconda in Setting Up Python for Machine Learning on Windows.

The examples in this tutorial have been tested with Python 3.7 and Pandas 0.25.0, but they should also work in older versions. Let’s get started!

1. **Using the Pandas Python Library**

Now that you’ve installed Pandas, it’s time to have a look at a dataset. In this tutorial, you’ll analyze NBA results provided by FiveThirtyEight in a 17MB CSV file. Create a script download\_nba\_all\_elo.py to download the data:

import requests

download\_url = "https://raw.githubusercontent.com/fivethirtyeight/data/master/nba-elo/nbaallelo.csv"

target\_csv\_path = "nba\_all\_elo.csv"

response = requests.get(download\_url)

response.raise\_for\_status() # Check that the request was successful

with open(target\_csv\_path, "wb") as f:

f.write(response.content)

print("Download ready.")

When you execute the script, it will save the file nba\_all\_elo.csv in your current working directory.

**Note:** You could also use your web browser to download the CSV file.

However, having a download script has several advantages:

* You can tell where you got your data.
* You can repeat the download anytime! That’s especially handy if the data is often refreshed.
* You don’t need to share the 17MB CSV file with your co-workers. Usually, it’s enough to share the download script.

Now you can use the Pandas Python library to take a look at your data:

>>> import pandas as pd

>>> nba = pd.read\_csv("nba\_all\_elo.csv")

>>> type(nba)

<class 'pandas.core.frame.DataFrame'>

Here, you follow the convention of importing Pandas in Python with the pd alias. Then, you use .read\_csv() to read in your dataset and store it as a DataFrame object in the variable nba.

**Note:** Is your data not in CSV format? No worries! The Pandas Python library provides several similar functions like read\_json(), read\_html(), and read\_sql\_table(). To learn how to work with these file formats, check out Reading and Writing Files With Pandas or consult the docs.

You can see how much data nba contains:

>>> len(nba)

126314

>>> nba.shape

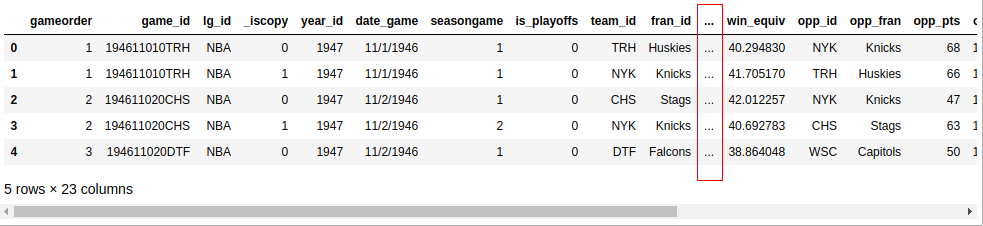
(126314, 23)

You use the Python built-in function len() to determine the number of rows. You also use the .shape attribute of the DataFrame to see its **dimensionality**. The result is a tuple containing the number of rows and columns.

Now you know that there are 126,314 rows and 23 columns in your dataset. But how can you be sure the dataset really contains basketball stats? You can have a look at the first five rows with .head():

>>> nba.head()

If you’re following along with a Jupyter notebook, then you’ll see a result like this:



Unless your screen is quite large, your output probably won’t display all 23 columns. Somewhere in the middle, you’ll see a column of ellipses (...) indicating the missing data. If you’re working in a terminal, then that’s probably more readable than wrapping long rows. However, Jupyter notebooks will allow you to scroll. You can configure Pandas to display all 23 columns like this:

>>> pd.set\_option("display.max.columns", None)

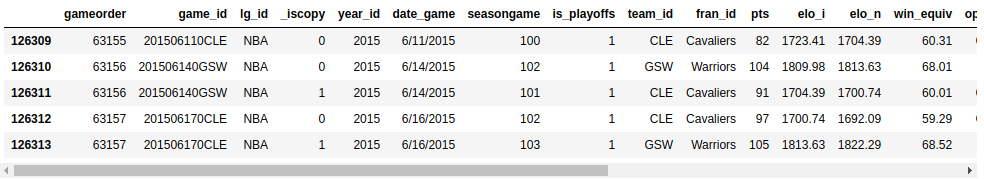
While it’s practical to see all the columns, you probably won’t need six decimal places! Change it to two:

>>> pd.set\_option("display.precision", 2)

To verify that you’ve changed the options successfully, you can execute .head() again, or you can display the last five rows with .tail() instead:

>>> nba.tail()

Now, you should see all the columns, and your data should show two decimal places:



You can discover some further possibilities of .head() and .tail() with a small exercise. Can you print the last three lines of your DataFrame? Expand the code block below to see the solution:

Similar to the Python standard library, functions in Pandas also come with several optional parameters. Whenever you bump into an example that looks relevant but is slightly different from your use case, check out the official documentation. The chances are good that you’ll find a solution by tweaking some optional parameters!

**2. Getting to Know Your Data**

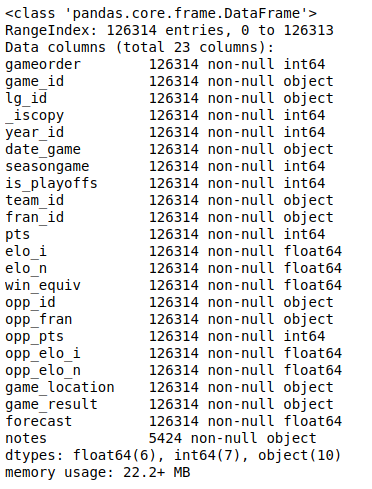
You’ve imported a CSV file with the Pandas Python library and had a first look at the contents of your dataset. So far, you’ve only seen the size of your dataset and its first and last few rows. Next, you’ll learn how to **examine your data** more systematically.

**Displaying Data Types**

The first step in getting to know your data is to discover the different data types it contains. While you can put anything into a list, the columns of a DataFrame contain values of a specific data type. When you compare Pandas and Python data structures, you’ll see that this behavior makes Pandas much faster!

You can display all columns and their data types with .info():

>>> nba.info()

This will produce the following output:  
You’ll see a list of all the columns in your dataset and the type of data each column contains. Here, you can see the data types int64, float64, and object. Pandas uses the NumPy library to work with these types. Later, you’ll meet the more complex categorical data type, which the Pandas Python library implements itself.

The object data type is a special one. According to the *Pandas Cookbook*, the object data type is “a catch-all for columns that Pandas doesn’t recognize as any other specific type.” In practice, it often means that all of the values in the column are strings.

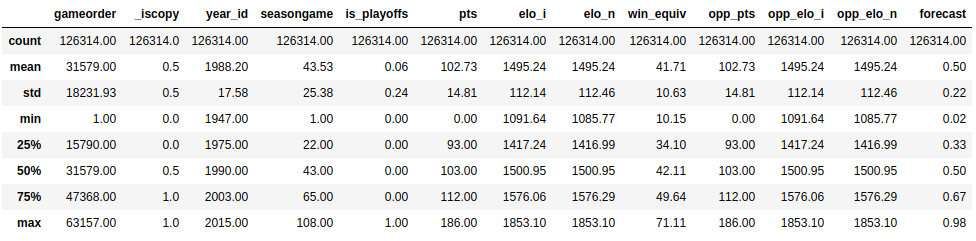
Although you can store arbitrary Python objects in the object data type, you should be aware of the drawbacks to doing so. Strange values in an object column can harm Pandas’ performance and its interoperability with other libraries. For more information, check out the official getting started guide.

**Showing Basics Statistics**

Now that you’ve seen what data types are in your dataset, it’s time to get an overview of the values each column contains. You can do this with .describe():

>>> nba.describe()

This function shows you some basic descriptive statistics for all numeric columns:

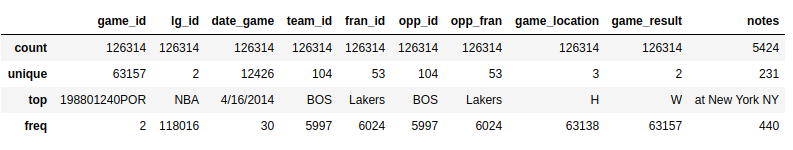


.describe() only analyzes numeric columns by default, but you can provide other data types if you use the include parameter:

>>> import numpy as np

>>> nba.describe(include=np.object)

.describe() won’t try to calculate a mean or a standard deviation for the object columns, since they mostly include text strings. However, it will still display some descriptive statistics:



Take a look at the team\_id and fran\_id columns. Your dataset contains 104 different team IDs, but only 53 different franchise IDs. Furthermore, the most frequent team ID is BOS, but the most frequent franchise ID Lakers. How is that possible? You’ll need to explore your dataset a bit more to answer this question.

**Exploring Your Dataset**

**Exploratory data analysis** can help you answer questions about your dataset. For example, you can examine how often specific values occur in a column:

>>> nba["team\_id"].value\_counts()

BOS 5997

NYK 5769

LAL 5078

...

SDS 11

>>> nba["fran\_id"].value\_counts()

Name: team\_id, Length: 104, dtype: int64

Lakers 6024

Celtics 5997

Knicks 5769

...

Huskies 60

Name: fran\_id, dtype: int64

It seems that a team named "Lakers" played 6024 games, but only 5078 of those were played by the Los Angeles Lakers. Find out who the other "Lakers" team is:

>>> nba.loc[nba["fran\_id"] == "Lakers", "team\_id"].value\_counts()

LAL 5078

MNL 946

Name: team\_id, dtype: int64

Indeed, the Minneapolis Lakers ("MNL") played 946 games. You can even find out when they played those games:

>>> nba.loc[nba["team\_id"] == "MNL", "date\_game"].min()

'1/1/1949'

>>> nba.loc[nba["team\_id"] == "MNL", "date\_game"].max()

'4/9/1959'

>>> nba.loc[nba["team\_id"] == "MNL", "date\_game"].agg(("min", "max"))

min 1/1/1949

max 4/9/1959

Name: date\_game, dtype: object

It looks like the Minneapolis Lakers played between the years of 1949 and 1959. That explains why you might not recognize this team!

You’ve also found out why the Boston Celtics team "BOS" played the most games in the dataset. Let’s analyze their history also a little bit. Find out how many points the Boston Celtics have scored during all matches contained in this dataset. Expand the code block below for the solution:

You’ve got a taste for the capabilities of a Pandas DataFrame. In the following sections, you’ll expand on the techniques you’ve just used, but first, you’ll zoom in and learn how this powerful data structure works.

**3. Getting to Know Pandas’ Data Structures**

While a DataFrame provides functions that can feel quite intuitive, the underlying concepts are a bit trickier to understand. For this reason, you’ll set aside the vast NBA DataFrame and build some smaller Pandas objects from scratch.

**Understanding Series Objects**

Python’s most basic data structure is the list, which is also a good starting point for getting to know **pandas.Series** objects. Create a new Series object based on a list:

>>> revenues = pd.Series([5555, 7000, 1980])

>>> revenues

0 5555

1 7000

2 1980

dtype: int64

You’ve used the list [5555, 7000, 1980] to create a Series object called revenues. A Series object wraps two components:

1. A sequence of **values**
2. A sequence of **identifiers**, which is the index

You can access these components with .values and .index, respectively:

>>> revenues.values

array([5555, 7000, 1980])

>>> revenues.index

RangeIndex(start=0, stop=3, step=1)

revenues.values returns the values in the Series, whereas revenues.index returns the positional index.

**Note:** If you’re familiar with NumPy, then it might be interesting for you to note that the values of a Series object are actually n-dimensional arrays:

>>> type(revenues.values)

<class 'numpy.ndarray'>

If you’re not familiar with NumPy, then there’s no need to worry! You can explore the ins and outs of your dataset with the Pandas Python library alone. However, if you’re curious about what Pandas does behind the scenes, then check out Look Ma, No For-Loops: Array Programming With NumPy.

While Pandas builds on NumPy, a significant difference is in their **indexing**. Just like a NumPy array, a Pandas Series also has an integer index that’s implicitly defined. This implicit index indicates the element’s position in the Series.

However, a Series can also have an arbitrary type of index. You can think of this explicit index as labels for a specific row:

>>> city\_revenues = pd.Series(

... [4200, 8000, 6500],

... index=["Amsterdam", "Toronto", "Tokyo"]

... )

>>> city\_revenues

Amsterdam 4200

Toronto 8000

Tokyo 6500

dtype: int64

Here, the index is a list of city names represented by strings. You may have noticed that Python dictionaries use string indices as well, and this is a handy analogy to keep in mind! You can use the code blocks above to distinguish between two types of Series:

1. **revenues:** This Series behaves like a Python list because it only has a positional index.
2. **city\_revenues:** This Series acts like a Python dictionary because it features both a positional and a label index.

Here’s how to construct a Series with a label index from a Python dictionary:

>>> city\_employee\_count = pd.Series({"Amsterdam": 5, "Tokyo": 8})

>>> city\_employee\_count

Amsterdam 5

Tokyo 8

dtype: int64

The dictionary keys become the index, and the dictionary values are the Series values.

Just like dictionaries, Series also support .keys() and the in keyword:

>>> city\_employee\_count.keys()

Index(['Amsterdam', 'Tokyo'], dtype='object')

>>> "Tokyo" in city\_employee\_count

True

>>> "New York" in city\_employee\_count

False

You can use these methods to answer questions about your dataset quickly.

**Understanding DataFrame Objects**

While a Series is a pretty powerful data structure, it has its limitations. For example, you can only store one attribute per key. As you’ve seen with the nba dataset, which features 23 columns, the Pandas Python library has more to offer with its **DataFrame**. This data structure is a sequence of Series objects that share the same index.

If you’ve followed along with the Series examples, then you should already have two Series objects with cities as keys:

1. city\_revenues
2. city\_employee\_count

You can combine these objects into a DataFrame by providing a dictionary in the constructor. The dictionary keys will become the column names, and the values should contain the Series objects:

>>> city\_data = pd.DataFrame({

... "revenue": city\_revenues,

... "employee\_count": city\_employee\_count

... })

>>> city\_data

revenue employee\_count

Amsterdam 4200 5.0

Tokyo 6500 8.0

Toronto 8000 NaN

Note how Pandas replaced the missing employee\_count value for Toronto with NaN.

The new DataFrame index is the union of the two Series indices:

>>> city\_data.index

Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object')

Just like a Series, a DataFrame also stores its values in a NumPy array:

>>> city\_data.values

array([[4.2e+03, 5.0e+00],

[6.5e+03, 8.0e+00],

[8.0e+03, nan]])

You can also refer to the 2 dimensions of a DataFrame as **axes**:

>>> city\_data.axes

[Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object'),

Index(['revenue', 'employee\_count'], dtype='object')]

>>> city\_data.axes[0]

Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object')

>>> city\_data.axes[1]

Index(['revenue', 'employee\_count'], dtype='object')

The axis marked with 0 is the **row index**, and the axis marked with 1 is the **column index**. This terminology is important to know because you’ll encounter several DataFrame methods that accept an axis parameter.

A DataFrame is also a dictionary-like data structure, so it also supports .keys() and the in keyword. However, for a DataFrame these don’t relate to the index, but to the columns:

>>> city\_data.keys()

Index(['revenue', 'employee\_count'], dtype='object')

>>> "Amsterdam" in city\_data

False

>>> "revenue" in city\_data

True

You can see these concepts in action with the bigger NBA dataset. Does it contain a column called "points", or was it called "pts"? To answer this question, display the index and the axes of the nba dataset, then expand the code block below for the solution:

As you use these methods to answer questions about your dataset, be sure to keep in mind whether you’re working with a Series or a DataFrame so that your interpretation is accurate.

**4. Accessing Series Elements**

In the section above, you’ve created a Pandas Series based on a Python list and compared the two data structures. You’ve seen how a Series object is similar to lists and dictionaries in several ways. A further similarity is that you can use the **indexing operator** ([]) for Series as well.

You’ll also learn how to use two Pandas-specific **access methods**:

1. **.loc**
2. **.iloc**

You’ll see that these data access methods can be much more readable than the indexing operator.

**Using the Indexing Operator**

Recall that a **Series** has two indices:

1. **A positional or implicit index**, which is always a RangeIndex
2. **A label or explicit index**, which can contain any hashable objects

Next, revisit the city\_revenues object:

>>> city\_revenues

Amsterdam 4200

Toronto 8000

Tokyo 6500

dtype: int64

You can conveniently access the values in a Series with both the label and positional indices:

>>> city\_revenues["Toronto"]

8000

>>> city\_revenues[1]

8000

You can also use negative indices and slices, just like you would for a list:

>>> city\_revenues[-1]

6500

>>> city\_revenues[1:]

Toronto 8000

Tokyo 6500

dtype: int64

>>> city\_revenues["Toronto":]

Toronto 8000

Tokyo 6500

dtype: int64

If you want to learn more about the possibilities of the indexing operator, then check out Lists and Tuples in Python.

**Accessing Series' Elements via .loc and .iloc**

The indexing operator ([]) is convenient, but there’s a caveat. What if the labels are also numbers? Say you have to work with a Series object like this:

>>> colors = pd.Series(

... ["red", "purple", "blue", "green", "yellow"],

... index=[1, 2, 3, 5, 8]

... )

>>> colors

1 red

2 purple

3 blue

5 green

8 yellow

dtype: object

What will colors[1] return? For a positional index, colors[1] is "purple". However, if you go by the label index, then colors[1] is referring to "red".

The good news is, you don’t have to figure it out! Instead, to avoid confusion, the Pandas Python library provides two **data access methods**:

1. **.loc** refers to the **label index**.
2. **.iloc** refers to the **positional index**.

These data access methods are much more readable:

>>> colors.loc[1]

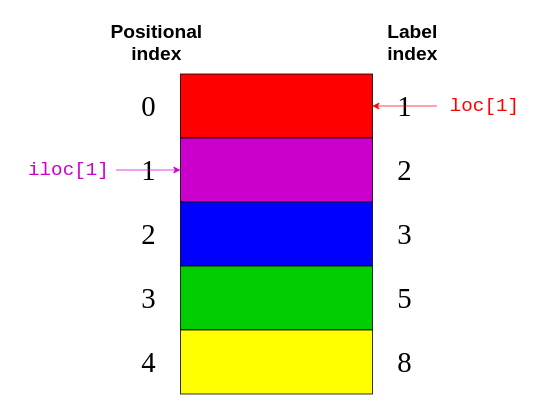
'red'

>>> colors.iloc[1]

'purple'

colors.loc[1] returned "red", the element with the label 1. colors.iloc[1] returned "purple", the element with the index 1.

The following figure shows which elements .loc and .iloc refer to:



Again, .loc points to the label index on the right-hand side of the image. Meanwhile, .iloc points to the positional index on the left-hand side of the picture.

It’s easier to keep in mind the distinction between .loc and .iloc than it is to figure out what the indexing operator will return. Even if you’re familiar with all the quirks of the indexing operator, it can be dangerous to assume that everybody who reads your code has internalized those rules as well!

**Note:** In addition to being confusing for Series with numeric labels, the Python indexing operator has some **performance drawbacks**. It’s perfectly okay to use it in interactive sessions for ad-hoc analysis, but for production code, the .loc and .iloc data access methods are preferable. For further details, check out the Pandas User Guide section on indexing and selecting data.

.loc and .iloc also support the features you would expect from indexing operators, like slicing. However, these data access methods have an important difference. While .iloc excludes the closing element, .loc includes it. Take a look at this code block:

>>> # Return the elements with the implicit index: 1, 2

>>> colors.iloc[1:3]

2 purple

3 blue

dtype: object

If you compare this code with the image above, then you can see that colors.iloc[1:3] returns the elements with the **positional indices** of 1 and 2. The closing item "green" with a positional index of 3 is excluded.

On the other hand, .loc includes the closing element:

>>> # Return the elements with the explicit index between 3 and 8

>>> colors.loc[3:8]

3 blue

5 green

8 yellow

dtype: object

This code block says to return all elements with a **label index** between 3 and 8. Here, the closing item "yellow" has a label index of 8 and is included in the output.

You can also pass a negative positional index to .iloc:

>>> colors.iloc[-2]

'green'

You start from the end of the Series and return the second element.

**Note:** There used to be an .ix indexer, which tried to guess whether it should apply positional or label indexing depending on the data type of the index. Because it caused a lot of confusion, it has been deprecated since Pandas version 0.20.0.

It’s highly recommended that you **do not use .ix** for indexing. Instead, always use .loc for label indexing and .iloc for positional indexing. For further details, check out the Pandas User Guide.

You can use the code blocks above to distinguish between two Series behaviors:

1. You can use .iloc on a Series similar to using [] on a **list**.
2. You can use .loc on a Series similar to using [] on a **dictionary**.

Be sure to keep these distinctions in mind as you access elements of your Series objects.

**5. Accessing DataFrame Elements**

Since a DataFrame consists of Series objects, you can use the very same tools to access its elements. The crucial difference is the additional **dimension** of the DataFrame. You’ll use the indexing operator for the columns and the access methods .loc and .iloc on the rows.

**Accessing DataFrame columns using the Indexing operator**

If you think of a DataFrame as a dictionary whose values are Series, then it makes sense that you can access its columns with the indexing operator:

>>> city\_data["revenue"]

Amsterdam 4200

Tokyo 6500

Toronto 8000

Name: revenue, dtype: int64

>>> type(city\_data["revenue"])

pandas.core.series.Series

Here, you use the indexing operator to select the column labeled "revenue".

If the column name is a string, then you can use attribute-style accessing with dot notation as well:

>>> city\_data.revenue

Amsterdam 4200

Tokyo 6500

Toronto 8000

Name: revenue, dtype: int64

city\_data["revenue"] and city\_data.revenue return the same output.

There’s one situation where accessing DataFrame elements with dot notation may not work or may lead to surprises. This is when a column name coincides with a DataFrame attribute or method name:

>>> toys = pd.DataFrame([

... {"name": "ball", "shape": "sphere"},

... {"name": "Rubik's cube", "shape": "cube"}

... ])

>>> toys["shape"]

0 sphere

1 cube

Name: shape, dtype: object

>>> toys.shape

(2, 2)

The indexing operation toys["shape"] returns the correct data, but the attribute-style operation toys.shape still returns the shape of the DataFrame. You should only use attribute-style accessing in interactive sessions or for read operations. You shouldn’t use it for production code or for manipulating data (such as defining new columns).

**Accessing DataFrame Rows With .loc and .iloc**

Similar to Series, a DataFrame also provides .loc and .iloc **data access methods**. Remember, .loc uses the label and .iloc the positional index:

>>> city\_data.loc["Amsterdam"]

revenue 4200.0

employee\_count 5.0

Name: Amsterdam, dtype: float64

>>> city\_data.loc["Tokyo": "Toronto"]

revenue employee\_count

Tokyo 6500 8.0

Toronto 8000 NaN

>>> city\_data.iloc[1]

revenue 6500.0

employee\_count 8.0

Name: Tokyo, dtype: float64

Each line of code selects a different row from city\_data:

1. **city\_data.loc["Amsterdam"]** selects the row with the label index "Amsterdam".
2. **city\_data.loc["Tokyo": "Toronto"]** selects the rows with label indices from "Tokyo" to "Toronto". Remember, .loc is inclusive.
3. **city\_data.iloc[1]** selects the row with the positional index 1, which is "Tokyo".

Alright, you’ve used .loc and .iloc on small data structures. Now, it’s time to practice with something bigger! Use a data access method to display the second-to-last row of the nba dataset. Then, expand the code block below to see a solution:

For a DataFrame, the data access methods .loc and .iloc also accept a second parameter. While the first parameter selects rows based on the indices, the second parameter selects the columns. You can use these parameters together to select a **subset** of rows and columns from your DataFrame:

>>> city\_data.loc["Amsterdam": "Tokyo", "revenue"]

Amsterdam 4200

Tokyo 6500

Name: revenue, dtype: int64

Note that you separate the parameters with a comma (,). The first parameter, "Amsterdam" : "Tokyo," says to select all rows between those two labels. The second parameter comes after the comma and says to select the "revenue" column.

It’s time to see the same construct in action with the bigger nba dataset. Select all games between the labels 5555 and 5559. You’re only interested in the names of the teams and the scores, so select those elements as well. Expand the code block below to see a solution:

With data access methods like .loc and .iloc, you can select just the right subset of your DataFrame to help you answer questions about your dataset.

**6. Querying Your Dataset**

You’ve seen how to access subsets of a huge dataset based on its indices. Now, you’ll select rows based on the values in your dataset’s columns to **query** your data. For example, you can create a new DataFrame that contains only games played after 2010:

>>> current\_decade = nba[nba["year\_id"] > 2010]

>>> current\_decade.shape

(12658, 23)

You still have all 23 columns, but your new DataFrame only consists of rows where the value in the "year\_id" column is greater than 2010.

You can also select the rows where a specific field is not null:

>>> games\_with\_notes = nba[nba["notes"].notnull()]

>>> games\_with\_notes.shape

(5424, 23)

This can be helpful if you want to avoid any missing values in a column. You can also use .notna() to achieve the same goal.

You can even access values of the object data type as str and perform string methods on them:

>>> ers = nba[nba["fran\_id"].str.endswith("ers")]

>>> ers.shape

(27797, 23)

You use .str.endswith() to filter your dataset and find all games where the home team’s name ends with "ers".

You can combine multiple criteria and query your dataset as well. To do this, be sure to put each one in parentheses and use the logical operators | and & to separate them.

**Note:** The operators and, or, &&, and || won’t work here. If you’re curious as to why, then check out the section on how the Pandas Python library uses Boolean operators in Python Pandas: Tricks & Features You May Not Know.

Do a search for Baltimore games where both teams scored over 100 points. In order to see each game only once, you’ll need to exclude duplicates:

>>> nba[

... (nba["\_iscopy"] == 0) &

... (nba["pts"] > 100) &

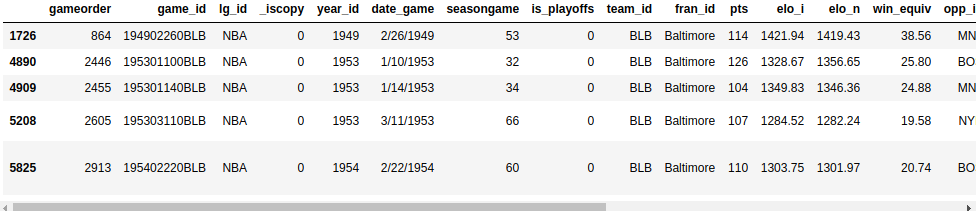
... (nba["opp\_pts"] > 100) &

... (nba["team\_id"] == "BLB")

... ]

Here, you use nba["\_iscopy"] == 0 to include only the entries that aren’t copies.

Your output should contain five eventful games:



Try to build another query with multiple criteria. In the spring of 1992, both teams from Los Angeles had to play a home game at another court. Query your dataset to find those two games. Both teams have an ID starting with "LA". Expand the code block below to see a solution:

When you know how to query your dataset with multiple criteria, you’ll be able to answer more specific questions about your dataset.

**7. Grouping and Aggregating Your Data**

You may also want to learn other features of your dataset, like the sum, mean, or average value of a group of elements. Luckily, the Pandas Python library offers **grouping and aggregation functions** to help you accomplish this task.

A Series has more than twenty different methods for calculating descriptive statistics. Here are some examples:

>>> city\_revenues.sum()

18700

>>> city\_revenues.max()

8000

The first method returns the total of city\_revenues, while the second returns the max value. There are other methods you can use, like .min() and .mean().

Remember, a column of a DataFrame is actually a Series object. For this reason, you can use these same functions on the columns of nba:

>>> points = nba["pts"]

>>> type(points)

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

>>> points.sum()

12976235

A DataFrame can have multiple columns, which introduces new possibilities for aggregations, like **grouping**:

>>> nba.groupby("fran\_id", sort=False)["pts"].sum()

fran\_id

Huskies 3995

Knicks 582497

Stags 20398

Falcons 3797

Capitols 22387

...

By default, Pandas sorts the group keys during the call to .groupby(). If you don’t want to sort, then pass sort=False. This parameter can lead to performance gains.

You can also group by multiple columns:

>>> nba[

... (nba["fran\_id"] == "Spurs") &

... (nba["year\_id"] > 2010)

... ].groupby(["year\_id", "game\_result"])["game\_id"].count()

year\_id game\_result

2011 L 25

W 63

2012 L 20

W 60

2013 L 30

W 73

2014 L 27

W 78

2015 L 31

W 58

Name: game\_id, dtype: int64

You can practice these basics with an exercise. Take a look at the Golden State Warriors’ 2014-15 season (year\_id: 2015). How many wins and losses did they score during the regular season and the playoffs? Expand the code block below for the solution:

In the examples above, you’ve only scratched the surface of the aggregation functions that are available to you in the Pandas Python library. To see more examples of how to use them, check out Pandas GroupBy: Your Guide to Grouping Data in Python.

**8. Manipulating Columns**

You’ll need to know how to **manipulate** your dataset’s columns in different phases of the data analysis process. You can add and drop columns as part of the initial data cleaning phase, or later based on the insights of your analysis.

Create a copy of your original DataFrame to work with:

>>> df = nba.copy()

>>> df.shape

(126314, 23)

You can define new columns based on the existing ones:

>>> df["difference"] = df.pts - df.opp\_pts

>>> df.shape

(126314, 24)

Here, you used the "pts" and "opp\_pts" columns to create a new one called "difference". This new column has the same functions as the old ones:

>>> df["difference"].max()

68

Here, you used an aggregation function .max() to find the largest value of your new column.

You can also rename the columns of your dataset. It seems that "game\_result" and "game\_location" are too verbose, so go ahead and rename them now:

>>> renamed\_df = df.rename(

... columns={"game\_result": "result", "game\_location": "location"}

... )

>>> renamed\_df.info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 126314 entries, 0 to 126313

Data columns (total 24 columns):

gameorder 126314 non-null int64

...

location 126314 non-null object

result 126314 non-null object

forecast 126314 non-null float64

notes 5424 non-null object

difference 126314 non-null int64

dtypes: float64(6), int64(8), object(10)

memory usage: 23.1+ MB

Note that there’s a new object, renamed\_df. Like several other data manipulation methods, .rename() returns a new DataFrame by default. If you want to manipulate the original DataFrame directly, then .rename() also provides an inplace parameter that you can set to True.

Your dataset might contain columns that you don’t need. For example, Elo ratings may be a fascinating concept to some, but you won’t analyze them in this tutorial. You can delete the four columns related to Elo:

>>> df.shape

(126314, 24)

>>> elo\_columns = ["elo\_i", "elo\_n", "opp\_elo\_i", "opp\_elo\_n"]

>>> df.drop(elo\_columns, inplace=True, axis=1)

>>> df.shape

(126314, 20)

Remember, you added the new column "difference" in a previous example, bringing the total number of columns to 24. When you remove the four Elo columns, the total number of columns drops to 20.

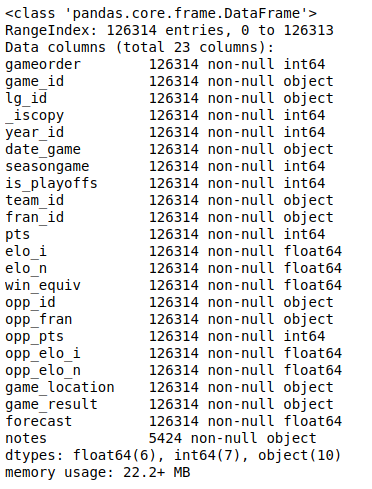
**9. Specifying Column Data Types**

When you create a new DataFrame, either by calling a constructor or reading a CSV file, Pandas assigns a **data type** to each column based on its values. While it does a pretty good job, it’s not perfect. If you choose the right data type for your columns upfront, then you can significantly improve your code’s performance.

Take another look at the columns of the nba dataset:

>>> df.info()

You’ll see the same output as before:



Ten of your columns have the data type object. Most of these object columns contain arbitrary text, but there are also some candidates for data type **conversion**. For example, take a look at the date\_game column:

>>> df["date\_game"] = pd.to\_datetime(df["date\_game"])

Here, you use .to\_datetime() to specify all game dates as datetime objects.

Other columns contain text that are a bit more structured. The game\_location column can have only three different values:

>>> df["game\_location"].nunique()

3

>>> df["game\_location"].value\_counts()

A 63138

H 63138

N 38

Name: game\_location, dtype: int64

Which data type would you use in a relational database for such a column? You would probably not use a varchar type, but rather an enum. Pandas provides the categorical data type for the same purpose:

>>> df["game\_location"] = pd.Categorical(df["game\_location"])

>>> df["game\_location"].dtype

CategoricalDtype(categories=['A', 'H', 'N'], ordered=False)

**categorical data** has a few advantages over unstructured text. When you specify the categorical data type, you make validation easier and save a ton of memory, as Pandas will only use the unique values internally. The higher the ratio of total values to unique values, the more space savings you’ll get.

Run df.info() again. You should see that changing the game\_location data type from object to categorical has decreased the memory usage.

**Note:** The categorical data type also gives you access to additional methods through the .cat accessor. To learn more, check out the official docs.

You’ll often encounter datasets with too many text columns. An essential skill for data scientists to have is the ability to spot which columns they can convert to a more performant data type.

Take a moment to practice this now. Find another column in the nba dataset that has a generic data type and convert it to a more specific one. You can expand the code block below to see one potential solution:

As you work with more massive datasets, memory savings becomes especially crucial. Be sure to keep **performance** in mind as you continue to explore your datasets.

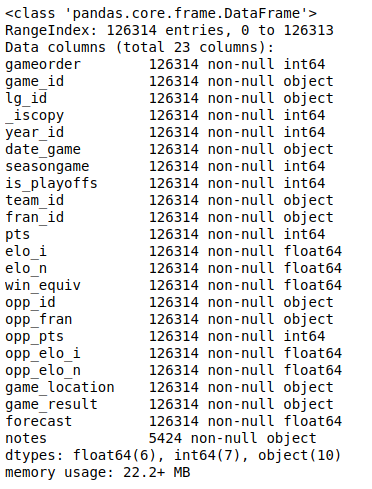
**10. Cleaning Data**

You may be surprised to find this section so late in the tutorial! Usually, you’d take a critical look at your dataset to fix any issues before you move on to a more sophisticated analysis. However, in this tutorial, you’ll rely on the techniques that you’ve learned in the previous sections to clean your dataset.

**Missing Values**

Have you ever wondered why .info() shows how many non-null values a column contains? The reason why is that this is vital information. **Null values** often indicate a problem in the data-gathering process. They can make several analysis techniques, like different types of machine learning, difficult or even impossible.

When you inspect the nba dataset with nba.info(), you’ll see that it’s quite neat. Only the column notes contains null values for the majority of its rows:



This output shows that the notes column has only 5424 non-null values. That means that over 120,000 rows of your dataset have null values in this column.

Sometimes, the easiest way to deal with records containing missing values is to ignore them. You can remove all the rows with missing values using .dropna():

>>> rows\_without\_missing\_data = nba.dropna()

>>> rows\_without\_missing\_data.shape

(5424, 23)

Of course, this kind of data cleanup doesn’t make sense for your nba dataset, because it’s not a problem for a game to lack notes. But if your dataset contains a million valid records and a hundred where relevant data is missing, then dropping the incomplete records can be a reasonable solution.

You can also drop problematic columns if they’re not relevant for your analysis. To do this, use .dropna() again and provide the axis=1 parameter:

>>> data\_without\_missing\_columns = nba.dropna(axis=1)

>>> data\_without\_missing\_columns.shape

(126314, 22)

Now, the resulting DataFrame contains all 126,314 games, but not the sometimes empty notes column.

If there’s a meaningful default value for your use case, then you can also replace the missing values with that:

>>> data\_with\_default\_notes = nba.copy()

>>> data\_with\_default\_notes["notes"].fillna(

... value="no notes at all",

... inplace=True

... )

>>> data\_with\_default\_notes["notes"].describe()

count 126314

unique 232

top no notes at all

freq 120890

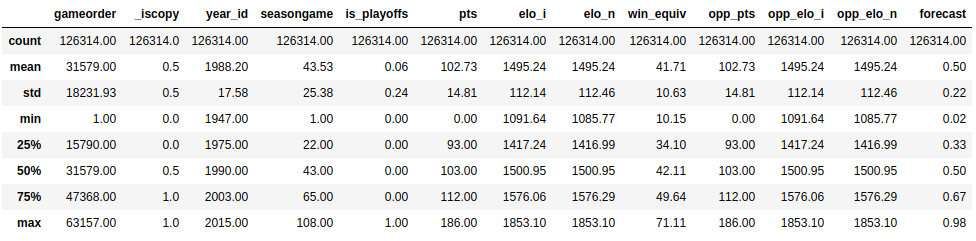
Name: notes, dtype: object

Here, you fill the empty notes rows with the string "no notes at all".

**Invalid Values**

**Invalid values** can be even more dangerous than missing values. Often, you can perform your data analysis as expected, but the results you get are peculiar. This is especially important if your dataset is enormous or used manual entry. Invalid values are often more challenging to detect, but you can implement some sanity checks with queries and aggregations.

One thing you can do is validate the ranges of your data. For this, .describe() is quite handy. Recall that it returns the following output:

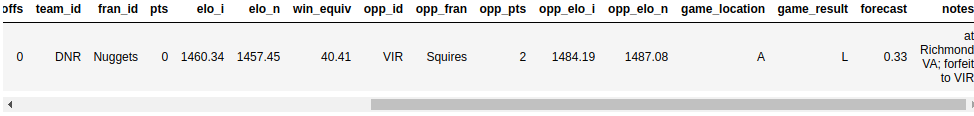


The year\_id varies between 1947 and 2015. That sounds plausible.

What about pts? How can the minimum be 0? Let’s have a look at those games:

>>> nba[nba["pts"] == 0]

This query returns a single row:



It seems the game was forfeited. Depending on your analysis, you may want to remove it from the dataset.

**Inconsistent Values**

Sometimes a value would be entirely realistic in and of itself, but it doesn’t fit with the values in the other columns. You can define some query criteria that are mutually exclusive and verify that these don’t occur together.

In the NBA dataset, the values of the fields pts, opp\_pts and game\_result should be consistent with each other. You can check this using the .empty attribute:

>>> nba[(nba["pts"] > nba["opp\_pts"]) & (nba["game\_result"] != 'W')].empty

True

>>> nba[(nba["pts"] < nba["opp\_pts"]) & (nba["game\_result"] != 'L')].empty

True

Fortunately, both of these queries return an empty DataFrame.

Be prepared for surprises whenever you’re working with raw datasets, especially if they were gathered from different sources or through a complex pipeline. You might see rows where a team scored more points than their opponent, but still didn’t win—at least, according to your dataset! To avoid situations like this, make sure you add further data cleaning techniques to your Pandas and Python arsenal.

**11. Combining Multiple Datasets**

In the previous section, you’ve learned how to clean a messy dataset. Another aspect of real-world data is that it often comes in multiple pieces. In this section, you’ll learn how to grab those pieces and **combine** them into one dataset that’s ready for analysis.

Earlier, you combined two Series objects into a DataFrame based on their indices. Now, you’ll take this one step further and use .concat() to combine city\_data with another DataFrame. Say you’ve managed to gather some data on two more cities:

>>> further\_city\_data = pd.DataFrame(

... {"revenue": [7000, 3400], "employee\_count":[2, 2]},

... index=["New York", "Barcelona"]

... )

This second DataFrame contains info on the cities "New York" and "Barcelona".

You can add these cities to city\_data using .concat():

>>> all\_city\_data = pd.concat([city\_data, further\_city\_data], sort=False)

>>> all\_city\_data

Amsterdam 4200 5.0

Tokyo 6500 8.0

Toronto 8000 NaN

New York 7000 2.0

Barcelona 3400 2.0

Now, the new variable all\_city\_data contains the values from both DataFrame objects.

**Note:** As of Pandas version 0.25.0, the sort parameter’s default value is True, but this will change to False soon. It’s good practice to provide an explicit value for this parameter to ensure that your code works consistently in different Pandas and Python versions. For more info, consult the Pandas User Guide.

By default, concat() combines along axis=0. In other words, it appends rows. You can also use it to append columns by supplying the parameter axis=1:

>>> city\_countries = pd.DataFrame({

... "country": ["Holland", "Japan", "Holland", "Canada", "Spain"],

... "capital": [1, 1, 0, 0, 0]},

... index=["Amsterdam", "Tokyo", "Rotterdam", "Toronto", "Barcelona"]

... )

>>> cities = pd.concat([all\_city\_data, city\_countries], axis=1, sort=False)

>>> cities

revenue employee\_count country capital

Amsterdam 4200.0 5.0 Holland 1.0

Tokyo 6500.0 8.0 Japan 1.0

Toronto 8000.0 NaN Canada 0.0

New York 7000.0 2.0 NaN NaN

Barcelona 3400.0 2.0 Spain 0.0

Rotterdam NaN NaN Holland 0.0

Note how Pandas added NaN for the missing values. If you want to combine only the cities that appear in both DataFrame objects, then you can set the join parameter to inner:

>>> pd.concat([all\_city\_data, city\_countries], axis=1, join="inner")

revenue employee\_count country capital

Amsterdam 4200 5.0 Holland 1

Tokyo 6500 8.0 Japan 1

Toronto 8000 NaN Canada 0

Barcelona 3400 2.0 Spain 0

While it’s most straightforward to combine data based on the index, it’s not the only possibility. You can use .merge() to implement a join operation similar to the one from SQL:

>>> countries = pd.DataFrame({

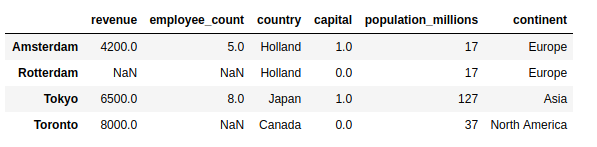
... "population\_millions": [17, 127, 37],

... "continent": ["Europe", "Asia", "North America"]

... }, index= ["Holland", "Japan", "Canada"])

>>> pd.merge(cities, countries, left\_on="country", right\_index=True)

Here, you pass the parameter left\_on="country" to .merge() to indicate what column you want to join on. The result is a bigger DataFrame that contains not only city data, but also the population and continent of the respective countries:



Note that the result contains only the cities where the country is known and appears in the joined DataFrame.

.merge() performs an inner join by default. If you want to include all cities in the result, then you need to provide the how parameter:

>>> pd.merge(

... cities,

... countries,

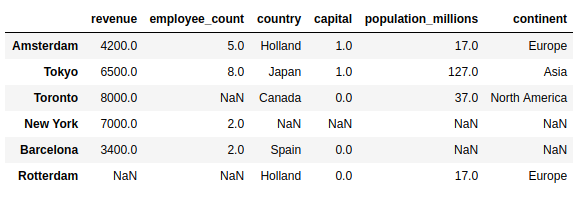
... left\_on="country",

... right\_index=True,

... how="left"

... )

With this left join, you’ll see all the cities, including those without country data:



Welcome back, New York & Barcelona!

**12. Visualizing Your Pandas DataFrame**

**Data visualization** is one of the things that works much better in a Jupyter notebook than in a terminal, so go ahead and fire one up. If you need help getting started, then check out Jupyter Notebook: An Introduction. You can also access the Jupyter notebook that contains the examples from this tutorial by clicking the link below:

**Get Jupyter Notebook:** Click here to get the Jupyter Notebook you’ll use to explore data with Pandas in this tutorial.

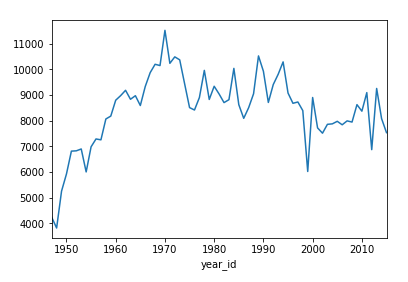
Include this line to show plots directly in the notebook:

>>> %matplotlib inline

Both Series and DataFrame objects have a .plot() method, which is a wrapper around matplotlib.pyplot.plot(). By default, it creates a **line plot**. Visualize how many points the Knicks scored throughout the seasons:

>>> nba[nba["fran\_id"] == "Knicks"].groupby("year\_id")["pts"].sum().plot()

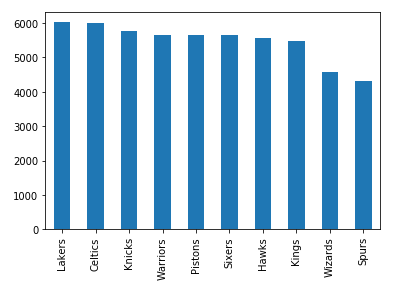
This shows a line plot with several peaks and two notable valleys around the years 2000 and 2010:



You can also create other types of plots, like a **bar plot**:

>>> nba["fran\_id"].value\_counts().head(10).plot(kind="bar")

This will show the franchises with the most games played:



The Lakers are leading the Celtics by a minimal edge, and there are six further teams with a game count above 5000.

Now try a more complicated exercise. In 2013, the Miami Heat won the championship. Create a pie plot showing the count of their wins and losses during that season. Then, expand the code block to see a solution:

Sometimes, the numbers speak for themselves, but often a chart helps a lot with communicating your insights. To learn more about visualizing your data, check out Interactive Data Visualization in Python With Bokeh.

**Conclusion**

In this tutorial, you’ve learned how to start **exploring a dataset** with the Pandas Python library. You saw how you could access specific rows and columns to tame even the largest of datasets. Speaking of taming, you’ve also seen multiple techniques to prepare and clean your data, by specifying the data type of columns, dealing with missing values, and more. You’ve even created queries, aggregations, and plots based on those.

**Now you can:**

* Work with Series and DataFrame objects
* Subset your data with .loc, .iloc, and the indexing operator
* Answer questions with queries, grouping, and aggregation
* Handle missing, invalid, and inconsistent data
* Visualize your dataset in a Jupyter notebook

This journey using the NBA stats only scratches the surface of what you can do with the Pandas Python library. You can power up your project with Pandas tricks, learn techniques to speed up Pandas in Python, and even dive deep to see how Pandas works behind the scenes. There are many more features for you to discover, so get out there and tackle those datasets!

You can get all the code examples you saw in this tutorial by clicking the link below:

Top of Form

Bottom of Form