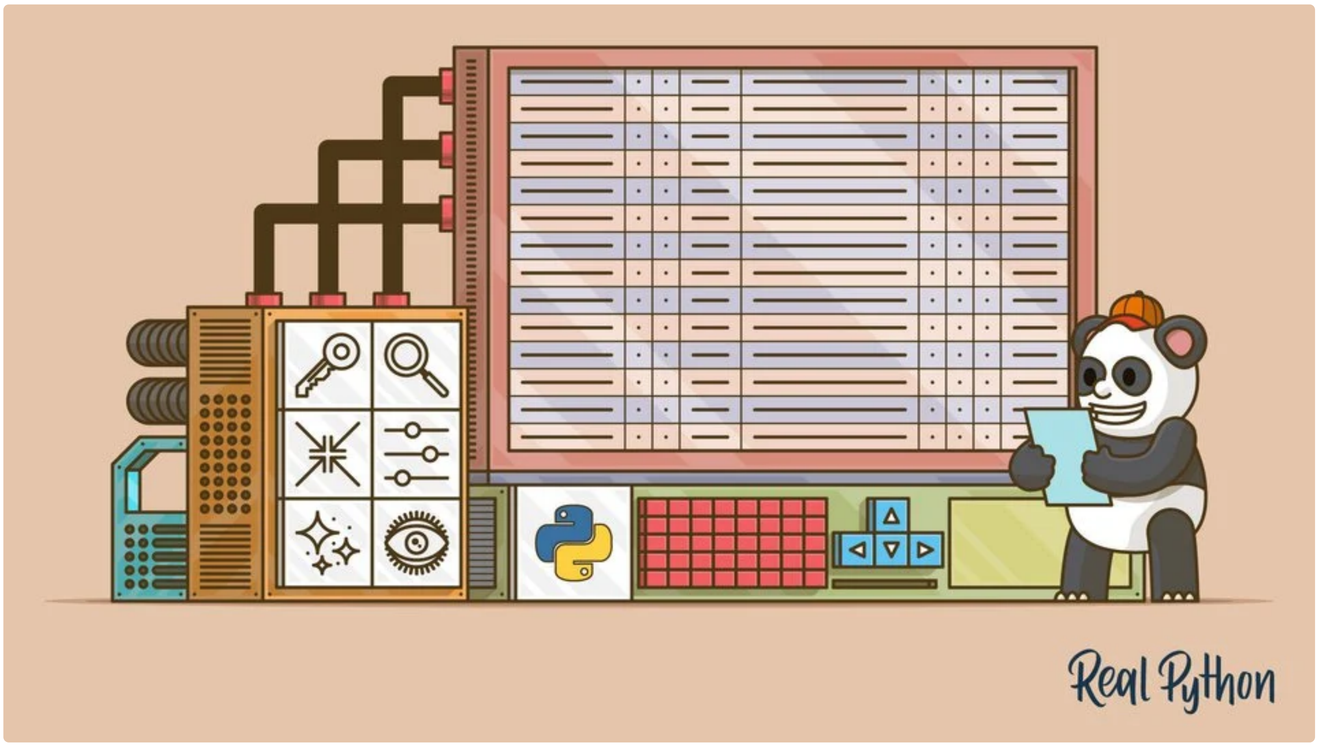


Using Pandas and Python to Explore Your Dataset

by Reka Horvath 14 Comments basics data-science

Mark as Completed

Tweet Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Explore Your Dataset With Pandas**

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. To follow along, you can get all of the example code in this tutorial at the link below:

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

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 Pandas and other Python libraries, which you can install with pip:

Shell

```
$ python3 -m pip install requests pandas matplotlib
```

You can also use the Conda package manager:

Shell

```
$ conda install requests pandas matplotlib
```

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. You can get all the code examples you'll see in this tutorial in a Jupyter notebook 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.

Let's get started!

Remove ads

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:

Python

```python
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:

Python >>>

```python
>>> 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:

Python >>>

```python
>>> len(nba)
126314
>>> nba.shape
(126314, 23)
```

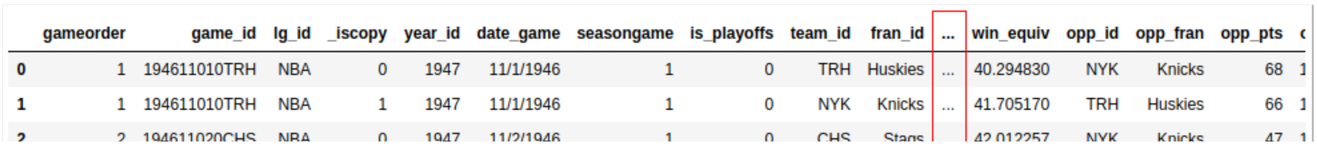
You use the Python built-in function `len()` to determine the number of rows. You also use the `.shape` attribute of the `DataFrame` to see its **dimensionality**. The result is a tuple containing the number of rows and columns.

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()`:

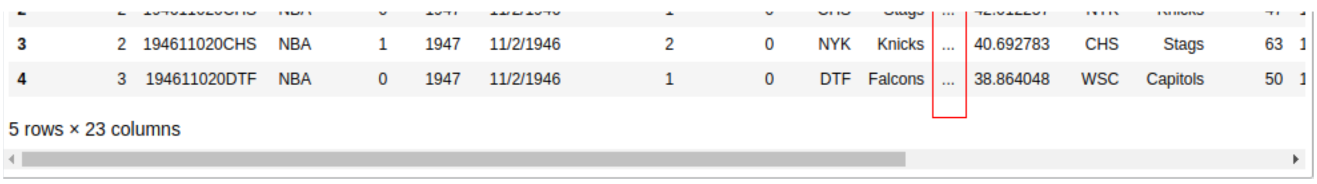
Python >>>

```python
>>> nba.head()
```

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



	gameorder	game_id	lg_id	_iscopy	year_id	date_game	seasongame	is_playoffs	team_id	fran_id	...	win_equiv	opp_id	opp_fran	opp_pts
0	1	194611010TRH	NBA	0	1947	11/1/1946	1	0	TRH	Huskies	...	40.294830	NYK	Knicks	68
1	1	194611010TRH	NBA	1	1947	11/1/1946	1	0	NYK	Knicks	...	41.705170	TRH	Huskies	66
2	2	194611020CHS	NBA	0	1947	11/2/1946	1	0	CHS	Stags		42.012257	NYK	Knicks	47



2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Stags	...	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
3	2	194611020CHS	NBA	1	1947	11/2/1946	2	0	NYK	Knicks	...	40.692783	CHS	Stags	63	1
4	3	194611020DTF	NBA	0	1947	11/2/1946	1	0	DTF	Falcons	...	38.864048	WSC	Capitols	50	1

5 rows × 23 columns

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:

Python >>>

```python
>>> 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:

Python >>>

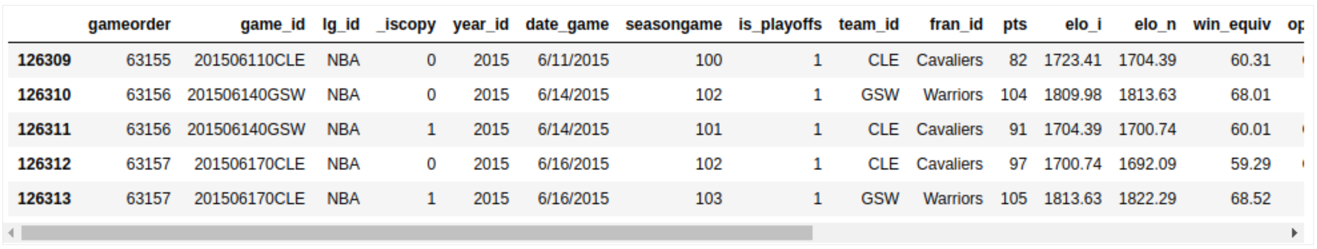
```python
>>> pd.set_option("display.precision", 2)
```

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:

Python >>>

```python
>>> nba.tail()
```

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



	gameorder	game_id	lg_id	_iscopy	year_id	date_game	seasongame	is_playoffs	team_id	fran_id	pts	elo_i	elo_n	win_equiv
126309	63155	201506110CLE	NBA	0	2015	6/11/2015	100	1	CLE	Cavaliers	82	1723.41	1704.39	60.31
126310	63156	201506140GSW	NBA	0	2015	6/14/2015	102	1	GSW	Warriors	104	1809.98	1813.63	68.01
126311	63156	201506140GSW	NBA	1	2015	6/14/2015	101	1	CLE	Cavaliers	91	1704.39	1700.74	60.01
126312	63157	201506170CLE	NBA	0	2015	6/16/2015	102	1	CLE	Cavaliers	97	1700.74	1692.09	59.29
126313	63157	201506170CLE	NBA	1	2015	6/16/2015	103	1	GSW	Warriors	105	1813.63	1822.29	68.52

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:

Solution: head & tail Show/Hide

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!

Remove ads

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

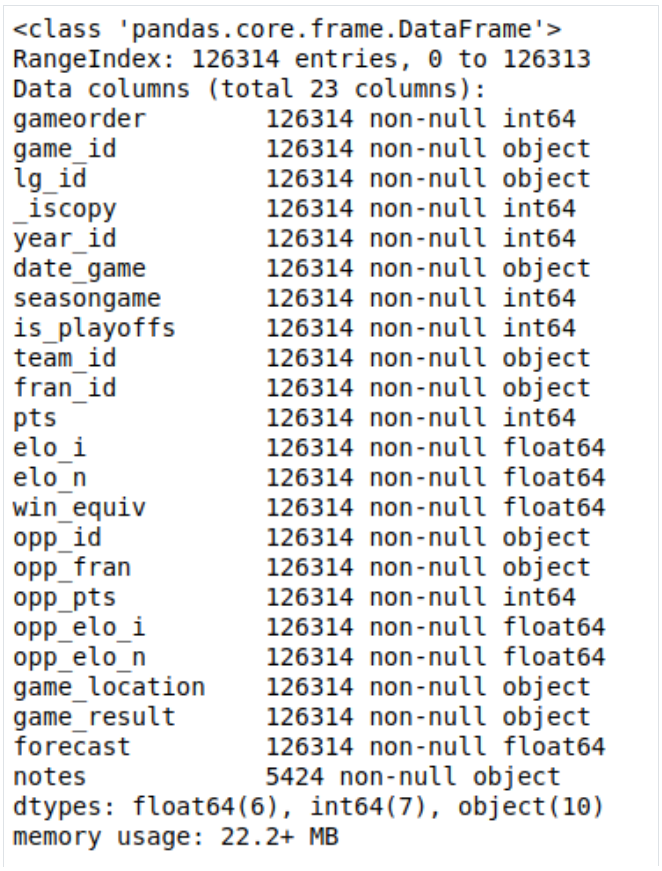
The first step in getting to know your data is to discover the different data types it contains. While you can put anything into a list, the columns of a `DataFrame` contain values of a specific data type. When you compare Pandas and Python data structures, you'll see that this behavior makes Pandas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

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

Python >>>

```
>>> nba.info()
```

This will produce the following output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 23 columns):
gameorder        126314 non-null int64
game_id          126314 non-null object
lg_id            126314 non-null object
_iscopy          126314 non-null int64
year_id          126314 non-null int64
date_game        126314 non-null object
seasongame       126314 non-null int64
is_playoffs      126314 non-null int64
team_id          126314 non-null object
fran_id          126314 non-null object
pts              126314 non-null int64
elo_i            126314 non-null float64
elo_n            126314 non-null float64
win_equiv        126314 non-null float64
opp_id           126314 non-null object
opp_fran         126314 non-null object
opp_pts          126314 non-null int64
opp_elo_i        126314 non-null float64
opp_elo_n        126314 non-null float64
game_location    126314 non-null object
game_result      126314 non-null object
forecast         126314 non-null float64
notes            5424 non-null object
dtypes: float64(6), int64(7), object(10)
memory usage: 22.2+ MB
```

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.

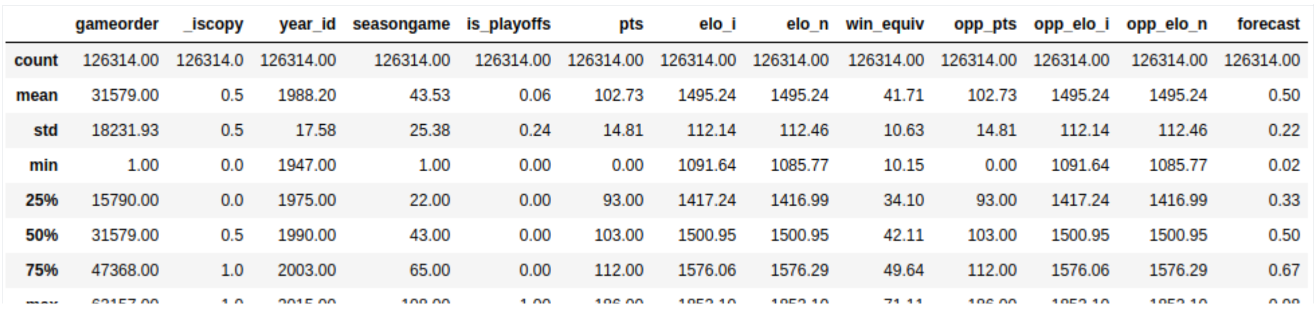
Showing Basics Statistics

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()`:

Python >>>

```
>>> nba.describe()
```

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



	gameorder	_iscopy	year_id	seasongame	is_playoffs	pts	elo_i	elo_n	win_equiv	opp_pts	opp_elo_i	opp_elo_n	forecast
count	126314.00	126314.0	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00
mean	31579.00	0.5	1988.20	43.53	0.06	102.73	1495.24	1495.24	41.71	102.73	1495.24	1495.24	0.50
std	18231.93	0.5	17.58	25.38	0.24	14.81	112.14	112.46	10.63	14.81	112.14	112.46	0.22
min	1.00	0.0	1947.00	1.00	0.00	0.00	1091.64	1085.77	10.15	0.00	1091.64	1085.77	0.02
25%	15790.00	0.0	1975.00	22.00	0.00	93.00	1417.24	1416.99	34.10	93.00	1417.24	1416.99	0.33
50%	31579.00	0.5	1990.00	43.00	0.00	103.00	1500.95	1500.95	42.11	103.00	1500.95	1500.95	0.50
75%	47368.00	1.0	2003.00	65.00	0.00	112.00	1576.06	1576.29	49.64	112.00	1576.06	1576.29	0.67

max	63157.00	1.0	2015.00	108.00	1.00	186.00	1853.10	1853.10	71.11	186.00	1853.10	1853.10	0.98

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

Python >>>

```
>>> import numpy as np
>>> nba.describe(include=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:

	game_id	lg_id	date_game	team_id	fran_id	opp_id	opp_fran	game_location	game_result	notes
count	126314	126314	126314	126314	126314	126314	126314	126314	126314	5424
unique	63157	2	12426	104	53	104	53	3	2	231
top	198801240POR	NBA	4/16/2014	BOS	Lakers	BOS	Lakers	H	W	at New York NY
freq	2	118016	30	5997	6024	5997	6024	63138	63157	440

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.

Remove ads

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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. For that, you'll first define a column that converts the value of `date_game` to the `datetime` data type. Then you can use the `min` and `max` aggregate functions, to find the first and last games of Minneapolis Lakers:

Python >>>

```
>>> nba["date_played"] = pd.to_datetime(nba["date_game"])
>>> nba.loc[nba["team_id"] == "MNL", "date_played"].min()
Timestamp('1948-11-04 00:00:00')
>>> nba.loc[nba['team_id'] == 'MNL', 'date_played'].max()
Timestamp('1960-03-26 00:00:00')
>>> nba.loc[nba["team_id"] == "MNL", "date_played"].agg(("min", "max"))
min   1948-11-04
max   1960-03-26
Name: date_played, dtype: datetime64[ns]
```

It looks like the Minneapolis Lakers played between the years of 1948 and 1960. 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:

Solution: DataFrame intro Show/Hide

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.

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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.

Remove ads

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> city_data.index
Index(['Amsterdam', 'Tokyo', 'Toronto'], dtype='object')
```

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

Python >>>

```
>>> 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**:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Solution: NBA index Show/Hide

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.

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.

Remove ads

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> city_revenues["Toronto"]
8000
>>> city_revenues[1]
8000
```

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

Python >>>

```
>>> 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.

Using `.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:

Python >>>

```
>>> 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**.

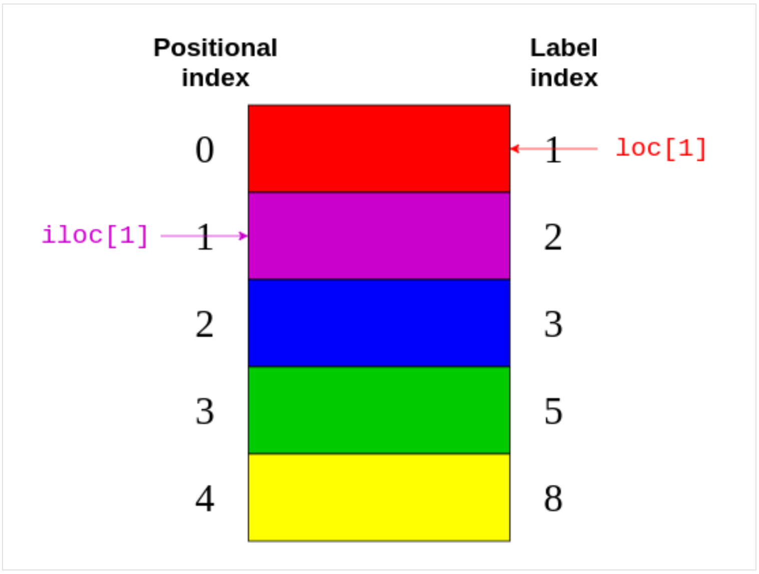
These data access methods are much more readable:

Python >>>

```
>>> 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:

Python >>>

```
>>> # 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:

Python >>>

```
>>> # 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`:

Python >>>

```
>>> 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.

Remove ads

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.

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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).

Using `.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:

Python >>>

```
>>> 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:

Solution: NBA accessing rows Show/Hide

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`:

Python >>>

```
>>> 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:

Solution: NBA accessing a subset Show/Hide

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.

Remove ads

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:

Python >>>

```
>>> current_decade = nba[nba["year_id"] > 2010]
>>> current_decade.shape
(12658, 24)
```

You now have 24 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:

Python >>>

```
>>> games_with_notes = nba[nba["notes"].notnull()]
>>> games_with_notes.shape
(5424, 24)
```

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:

Python >>>

```
>>> ers = nba[nba["fran_id"].str.endswith("ers")]
>>> ers.shape
(27797, 24)
```

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.

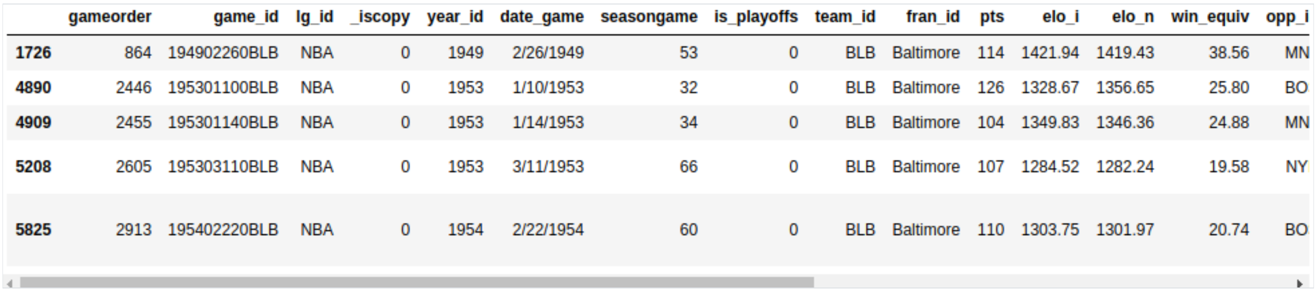
Do a search for Baltimore games where both teams scored over 100 points. In order to see each game only once, you'll need to exclude duplicates:

Python >>>

```
>>> 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:



	gameorder	game_id	lg_id	_iscopy	year_id	date_game	seasongame	is_playoffs	team_id	fran_id	pts	elo_i	elo_n	win_equiv	opp_i
1726	864	194902260BLB	NBA	0	1949	2/26/1949	53	0	BLB	Baltimore	114	1421.94	1419.43	38.56	MN
4890	2446	195301100BLB	NBA	0	1953	1/10/1953	32	0	BLB	Baltimore	126	1328.67	1356.65	25.80	BO
4909	2455	195301140BLB	NBA	0	1953	1/14/1953	34	0	BLB	Baltimore	104	1349.83	1346.36	24.88	MN
5208	2605	195303110BLB	NBA	0	1953	3/11/1953	66	0	BLB	Baltimore	107	1284.52	1282.24	19.58	NY
5825	2913	195402220BLB	NBA	0	1954	2/22/1954	60	0	BLB	Baltimore	110	1303.75	1301.97	20.74	BO

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:

Solution: Queries Show/Hide

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

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:

Python >>>

```
>>> 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`:

Python >>>

```
>>> 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**:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Solution: Aggregation Show/Hide

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.

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:

Python >>>

```
>>> df = nba.copy()
>>> df.shape
(126314, 24)
```

You can define new columns based on the existing ones:

Python >>>

```
>>> df["difference"] = df.pts - df.opp_pts
>>> df.shape
(126314, 25)
```

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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 25 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   gameorder    126314 non-null  int64
 ...
 19  location     126314 non-null  object
 20  result       126314 non-null  object
 21  forecast     126314 non-null  float64
 22  notes        5424 non-null    object
 23  date_played  126314 non-null  datetime64[ns]
 24  difference   126314 non-null  int64
dtypes: datetime64[ns](1), float64(6), int64(8), object(10)
memory usage: 24.1+ 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:

Python >>>

```
>>> df.shape
(126314, 25)
>>> elo_columns = ["elo_i", "elo_n", "opp_elo_i", "opp_elo_n"]
>>> df.drop(elo_columns, inplace=True, axis=1)
>>> df.shape
(126314, 21)
```

Remember, you added the new column `"difference"` in a previous example, bringing the total number of columns to 25. When you remove the four Elo columns, the total number of columns drops to 21.

Remove ads

Specifying Data Types

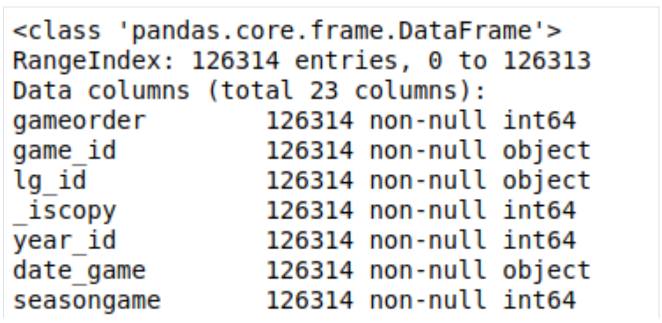
When you create a new `DataFrame`, either by calling a constructor or reading a CSV file, Pandas assigns a **data type** to each column based on its values. While it does a pretty good job, it's not perfect. If you choose the right data type for your columns up front, then you can significantly improve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

Take another look at the columns of the `nba` dataset:

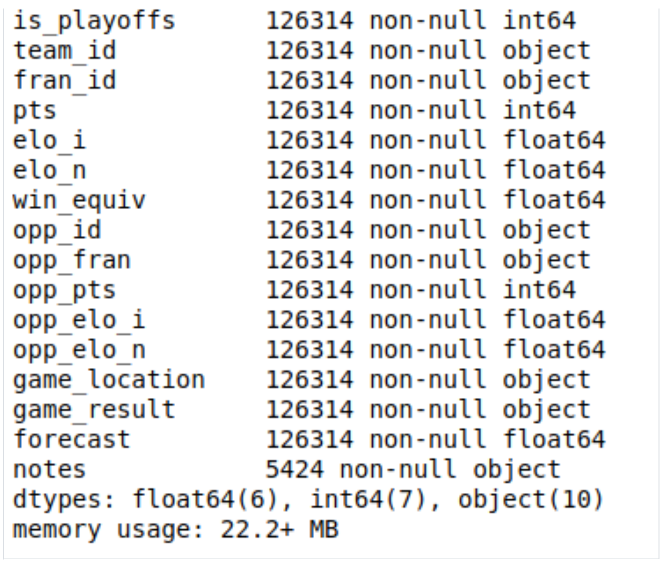
Python >>>

```
>>> df.info()
```

You'll see the same output as before:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 23 columns):
gameorder        126314 non-null int64
game_id          126314 non-null object
lg_id            126314 non-null object
_iscopy          126314 non-null int64
year_id          126314 non-null int64
date_game        126314 non-null object
seasongame       126314 non-null int64
```



```
is_playoffs      126314 non-null int64
team_id          126314 non-null object
fran_id          126314 non-null object
pts              126314 non-null int64
elo_i            126314 non-null float64
elo_n            126314 non-null float64
win_equiv        126314 non-null float64
opp_id           126314 non-null object
opp_fran         126314 non-null object
opp_pts          126314 non-null int64
opp_elo_i        126314 non-null float64
opp_elo_n        126314 non-null float64
game_location    126314 non-null object
game_result      126314 non-null object
forecast         126314 non-null float64
notes            5424 non-null object
dtypes: float64(6), int64(7), object(10)
memory usage: 22.2+ MB
```

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

Solution: Specifying Data Types Show/Hide

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.

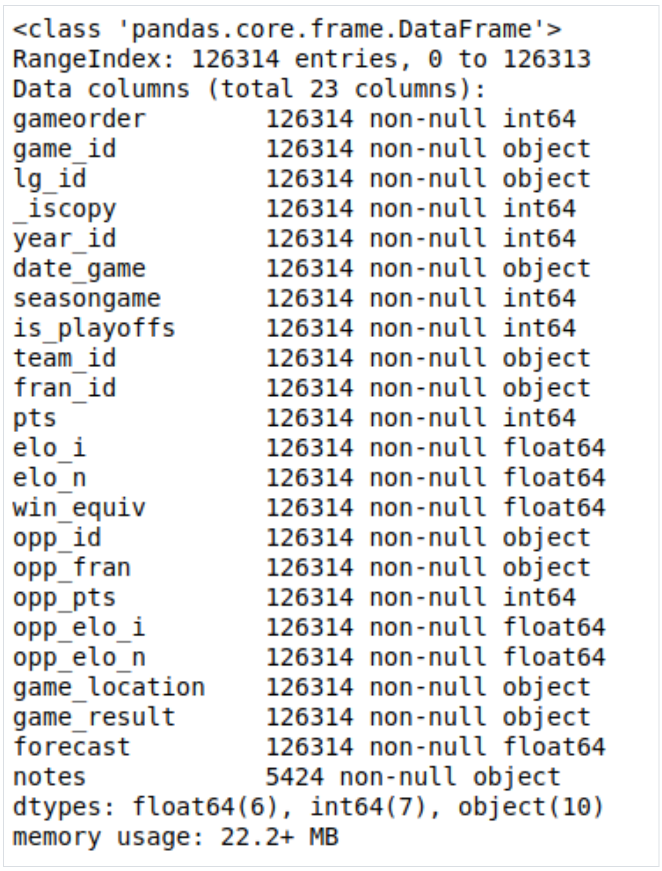
Cleaning Data

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:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 126314 entries, 0 to 126313
Data columns (total 23 columns):
gameorder        126314 non-null int64
game_id          126314 non-null object
lg_id            126314 non-null object
_iscopy          126314 non-null int64
year_id          126314 non-null int64
date_game        126314 non-null object
seasongame       126314 non-null int64
is_playoffs      126314 non-null int64
team_id          126314 non-null object
fran_id          126314 non-null object
pts              126314 non-null int64
elo_i            126314 non-null float64
elo_n            126314 non-null float64
win_equiv        126314 non-null float64
opp_id           126314 non-null object
opp_fran         126314 non-null object
opp_pts          126314 non-null int64
opp_elo_i        126314 non-null float64
opp_elo_n        126314 non-null float64
game_location    126314 non-null object
game_result      126314 non-null object
forecast         126314 non-null float64
notes            5424 non-null object
dtypes: float64(6), int64(7), object(10)
memory usage: 22.2+ MB
```

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()`:

Python >>>

```python
>>> rows_without_missing_data = nba.dropna()
>>> rows_without_missing_data.shape
(5424, 24)
```

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:

Python >>>

```
>>> data_without_missing_columns = nba.dropna(axis=1)
>>> data_without_missing_columns.shape
(126314, 23)
```

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:

Python >>>

```
>>> 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:

	gameorder	_iscopy	year_id	seasongame	is_playoffs	pts	elo_i	elo_n	win_equiv	opp_pts	opp_elo_i	opp_elo_n	forecast
count	126314.00	126314.0	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00	126314.00
mean	31579.00	0.5	1988.20	43.53	0.06	102.73	1495.24	1495.24	41.71	102.73	1495.24	1495.24	0.50
std	18231.93	0.5	17.58	25.38	0.24	14.81	112.14	112.46	10.63	14.81	112.14	112.46	0.22
min	1.00	0.0	1947.00	1.00	0.00	0.00	1091.64	1085.77	10.15	0.00	1091.64	1085.77	0.02
25%	15790.00	0.0	1975.00	22.00	0.00	93.00	1417.24	1416.99	34.10	93.00	1417.24	1416.99	0.33
50%	31579.00	0.5	1990.00	43.00	0.00	103.00	1500.95	1500.95	42.11	103.00	1500.95	1500.95	0.50
75%	47368.00	1.0	2003.00	65.00	0.00	112.00	1576.06	1576.29	49.64	112.00	1576.06	1576.29	0.67
max	63157.00	1.0	2015.00	108.00	1.00	186.00	1853.10	1853.10	71.11	186.00	1853.10	1853.10	0.98

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:

Python >>>

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

This query returns a single row:

offs	team_id	fran_id	pts	elo_i	elo_n	win_equiv	opp_id	opp_fran	opp_pts	opp_elo_i	opp_elo_n	game_location	game_result	forecast	notes
0	DNR	Nuggets	0	1460.34	1457.45	40.41	VIR	Squires	2	1484.19	1487.08	A	L	0.33	at Richmond VA; forfeit to VIR

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:

Python >>>

```
>>> 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.

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:

Python >>>

```
>>> 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()`:

Python >>>

```
>>> 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`:

Python >>>

```
>>> 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`:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

	revenue	employee_count	country	capital	population_millions	continent
Amsterdam	4200.0	5.0	Holland	1.0	17	Europe
Rotterdam	NaN	NaN	Holland	0.0	17	Europe
Tokyo	6500.0	8.0	Japan	1.0	127	Asia
Toronto	8000.0	NaN	Canada	0.0	37	North America

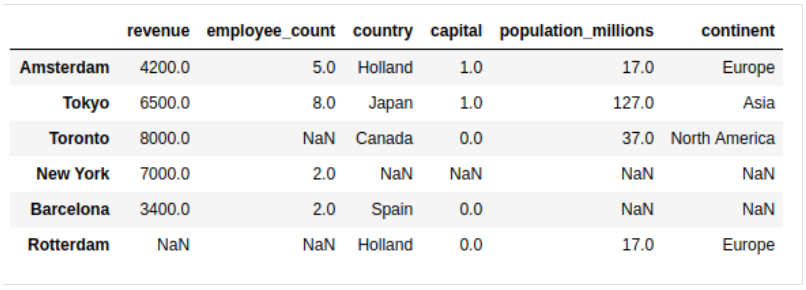
Note that the result contains only the cities where the country is known and appears in the joined `DataFrame`.

`.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:

Python >>>

```
>>> 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:



	revenue	employee_count	country	capital	population_millions	continent
Amsterdam	4200.0	5.0	Holland	1.0	17.0	Europe
Tokyo	6500.0	8.0	Japan	1.0	127.0	Asia
Toronto	8000.0	NaN	Canada	0.0	37.0	North America
New York	7000.0	2.0	NaN	NaN	NaN	NaN
Barcelona	3400.0	2.0	Spain	0.0	NaN	NaN
Rotterdam	NaN	NaN	Holland	0.0	17.0	Europe

Welcome back, New York & Barcelona!

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:

Python >>>

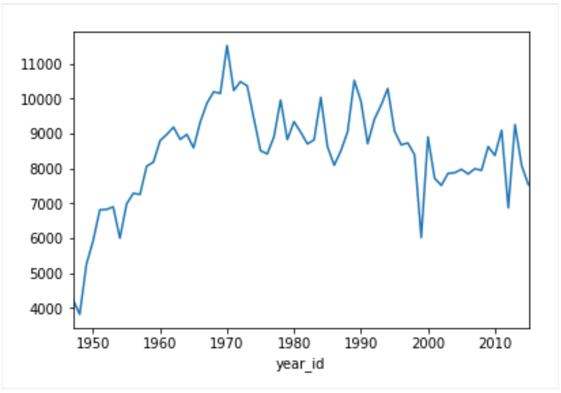
```
>>> %matplotlib inline
```

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:

Python >>>

```
>>> 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:

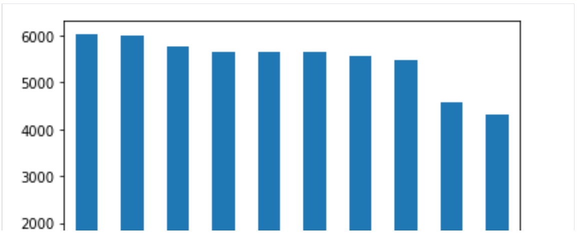


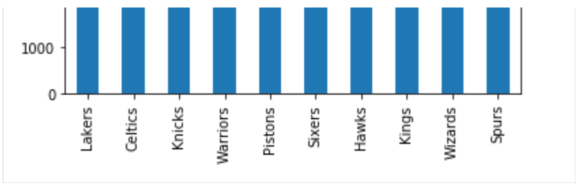
You can also create other types of plots, like a **bar plot**:

Python >>>

```
>>> 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:

Solution: Plot Show/Hide

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:

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

Mark as Completed

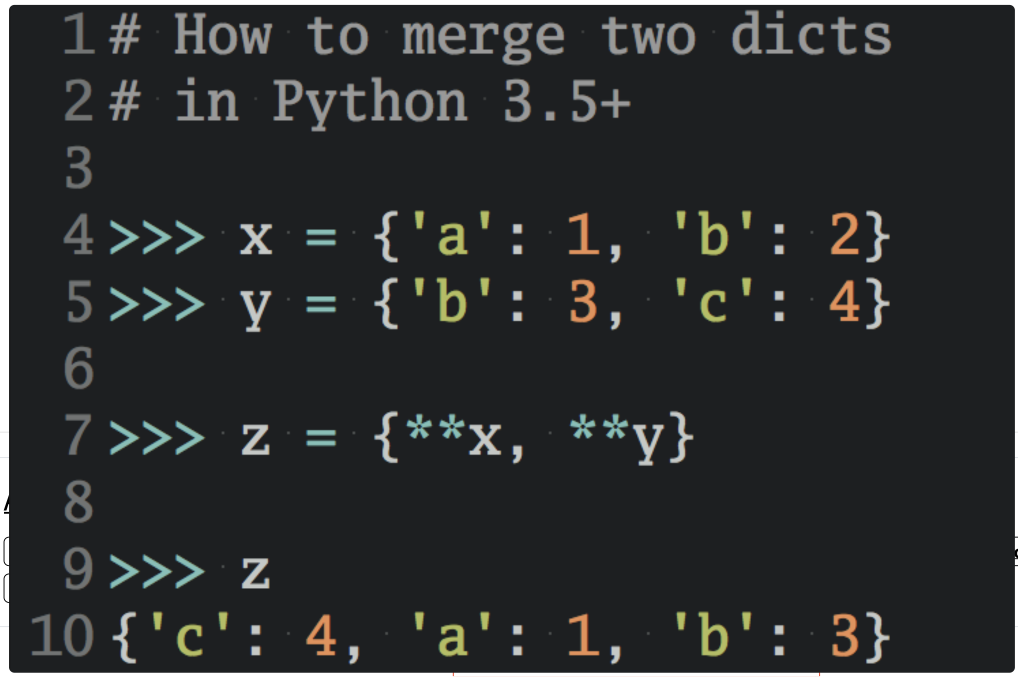
Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Explore Your Dataset With Pandas**

About **Reka Horvath**

Reka is an avid Pythonista and writes for Real Python.

» More about Reka

ho



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

ont-end gamedev gui

Table of Contents

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Mark as Completed

Tweet Share Email

Keep Learning

Recommended Video Course

Explore Your Dataset With Pandas

Recommended Video Course: Explore Your Dataset With Pandas

Remove ads