# Join the DataFrames like SQL tables in Python using Pandas

Nilabh Nishchhal — Published On June 28, 2021 and Last Modified On July 22nd, 2022

Beginner Programming Project Python SQL

This article was published as a part of the Data Science Blogathon

In the age of relational databases, Joining and Merging tables is a necessity. This is an integral and unavoidable step of Data Mining and Data Preparation.

But before we go forward, Let me explain what is a relational database, and how do they differ from other forms of dataBase.

## Introduction to Relational Database

A database is simply a set of related information. This information can be stored in many possible ways. One of the most popular ways is storing all the information in one worksheet or table (basically 2-dimensional format, which has dimensions as ROWS and COLUMNS). This format has its advantages like the relevant data is easy to find and it's fast to fetch results. But as the size of the data grows, it becomes more and more difficult to handle this tabular data (because of its size).

Let us understand this by taking an example of a bank keeping details of its customers and their transactions. If the bank starts maintaining data in one table, it shall have customer name, their account numbers, transaction dates, and transactions details, along with many other customer-related details. The customer account details, which do not change frequently, can be called their master data. If the information for transactions is maintained in one sheet, this data will repeat every time whenever the customer transacts, and unnecessarily consume memory. Now when the data is for a large organization like a bank, keeping that all in one table will be simply not possible. What are the options then? There can be many ways, but two simple methods which come to my mind are as below.

1. Every customer's transactions to be maintained in a separate sheet (table)
2. The customers' master data is maintained in one table, and transaction details are maintained in another table.

In the first approach, it might look simplistic at first glance, but is actually very complicated for an organization to handle. Imagine the bankers collating all such sheets to find their daily cumulative transactions, branch-wise and as a company. They will curse you every day if you designed this system for them.

However, in the second system, the master data which doesn't change frequently are kept in separate tables, and the transaction data may be kept in other tables, and these tables can have some common identifier field like customer ID or customer account number. For example, see the image below showing 4 such tables, one for storing customer personal details, second for bank's account details being used by customers (one customer may have many accounts, one saving, one Demat, one deposit account, etc), third for bank's product details and the fourth table for all the transactions.

| cust_id | fname | lname |
|---|---|---|
| 1 | George | Blake |
| 2 | Sue | Smith |

| account_id | product_cd | cust_id | balance |
|---|---|---|---|
| 103 | CHK | 1 | $75.00 |
| 104 | SAV | 1 | $250.00 |
| 105 | CHK | 2 | $783.64 |
| 106 | MM | 2 | $500.00 |
| 107 | LOC | 2 | 0 |

*Product*

| product_cd | name |
|---|---|
| CHK | Checking |
| SAV | Savings |
| MM | Money market |
| LOC | Line of credit |

*Transaction*

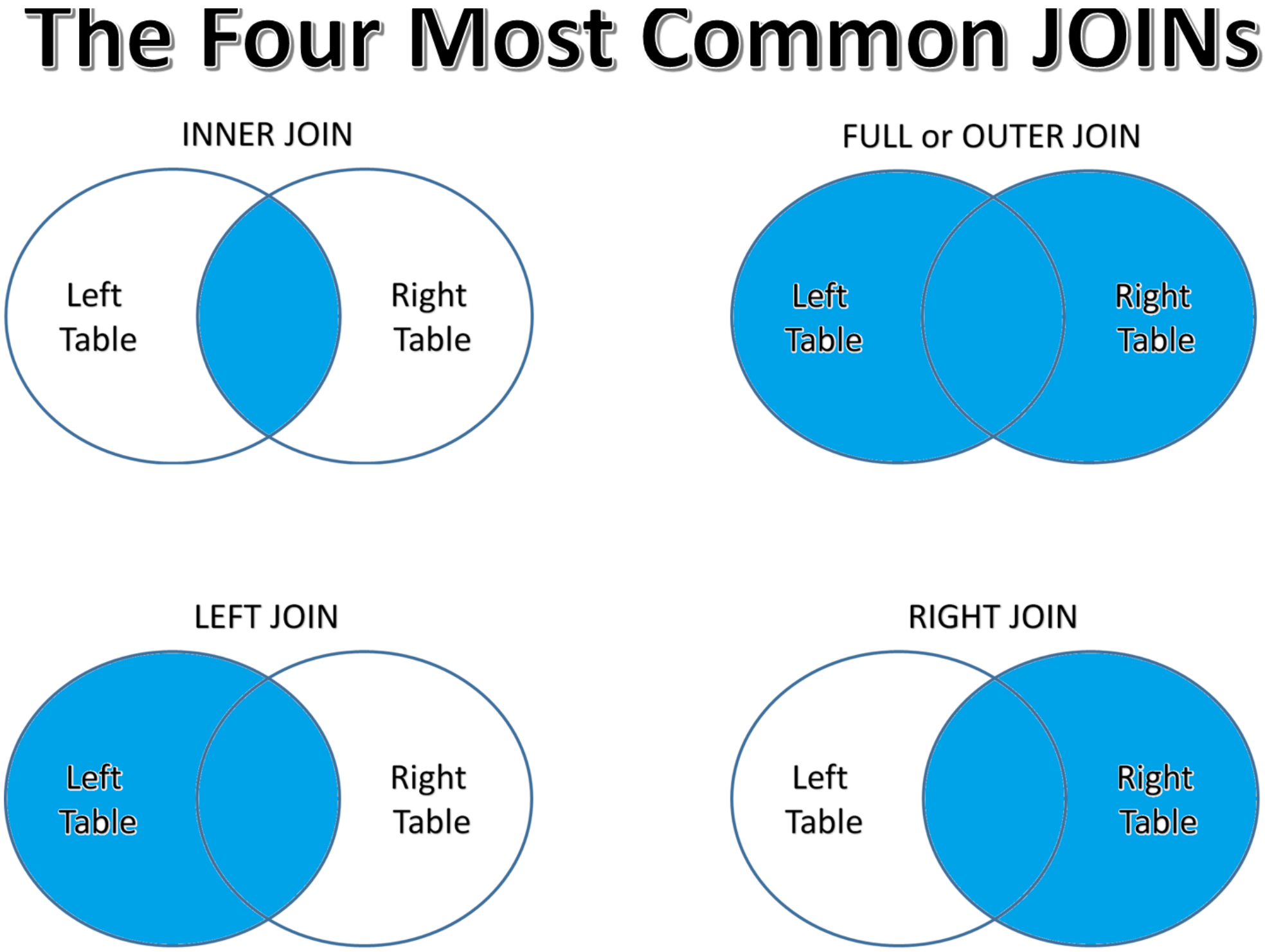
| txn_id | txn_type_cd | account_id | amount | date |
|---|---|---|---|---|
| 978 | DBT | 103 | $100.00 | 2004-01-22 |
| 979 | CDT | 103 | $25.00 | 2004-02-05 |
| 980 | DBT | 104 | $250.00 | 2004-03-09 |
| 981 | DBT | 105 | $1000.00 | 2004-03-25 |
| 982 | CDT | 105 | $138.50 | 2004-04-02 |
| 983 | CDT | 105 | $77.86 | 2004-04-04 |
| 984 | DBT | 106 | $500.00 | 2004-03-27 |

*Source: O'Riley Book – Learning SQL by Alan Beaulieu*

This second approach is a very simplistic example of a Relational Database. Such databases need to join tables for performing many operations and getting out relevant details from them. SQL is the most common language for querying databases, and it has efficient join functions. The same can be replicated in Python and Python also has some additional weapons of its own in its arsenal, which will help you in many join and merge operations. Let us have a look at these functions, starting with types of joins now.

## Popular Joins in SQL

These are the four most common joins in SQL. let us see what these are one by one, and also how to perform them in Python. We will be taking the help of Pandas library for this task.



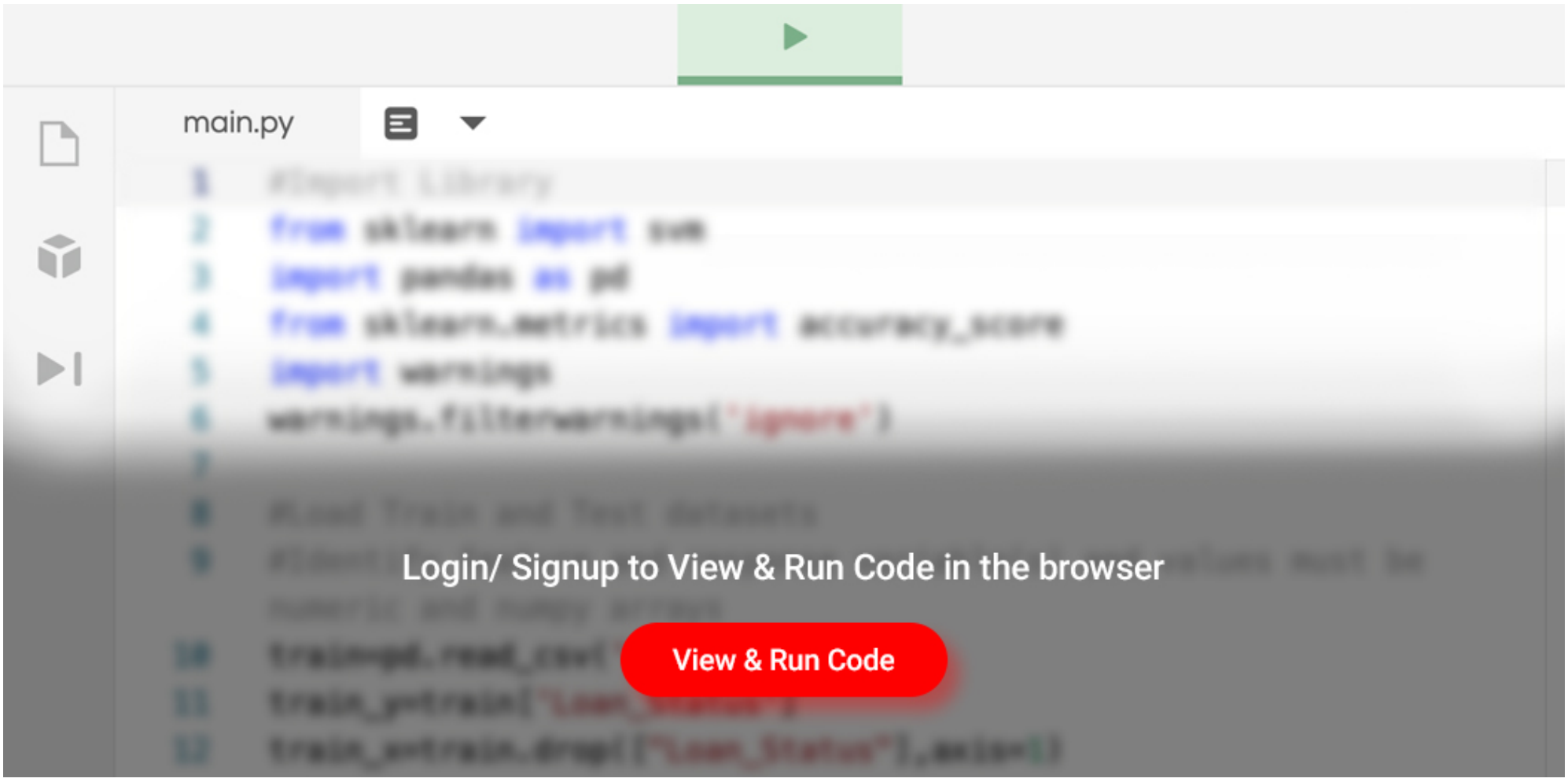

*Source: Created by Author*

We will use two tables to see how the join works. Both the tables will have 10 rows each, out of which 5 are common between them. Let us first create out DataFrames to work upon.

Apart from Joins, many other popular SQL functions are easily implementable in Python. Read "15 Pandas functions to replicate basic SQL Queries in Python" for learning how to do that.

In [1]:

```
# Country and its capitals
capitals = pd.DataFrame(
    {'Country':
['Afghanistan','Argentina','Australia','Canada','China','France','India','Nepal','Russia','Spain'],
     'ISO' : ['AF','AR','AU','CA','CN','FR','IN','NP','RU','ES'],
     'Capital' :
['Kabul','Buenos_Aires','Canberra','Ottawa','Beijing','Paris','New_Delhi','Katmandu','Moscow','Madrid'
 },
    columns=['Country', 'ISO', 'Capital'])

# Country and its currencies
currency = pd.DataFrame(
    {'Country':
['France','India','Nepal','Russia','Spain','Sri_Lanka','United_Kingdom','USA','Uzbekistan','Zimbabwe']

     'Currency' :
['Euro','Indian_Rupee','Nepalese_Rupee','Rouble','Euro','Rupee','Pound','US_Dollar','Sum_Coupons','Zim

     'Digraph' : ['FR','IN','NP','RU','ES','LK','GB','US','UZ','ZW'] },
    columns=['Country', 'Currency', 'Digraph'])
```




In [2]:

```
capitals
```

Out[2]:

| | Country | ISO | Capital |
|---|---|---|---|
| 0 | Afghanistan | AF | Kabul |
| 1 | Argentina | AR | Buenos_Aires |
| 2 | Australia | AU | Canberra |
| 3 | Canada | CA | Ottawa |
| 4 | China | CN | Beijing |
| 5 | France | FR | Paris |

| | Country | ISO | Capital |
|---|---|---|---|
| 7 | Nepal | NP | Kathmandu |
| 8 | Russia | RU | Moscow |
| 9 | Spain | ES | Madrid |

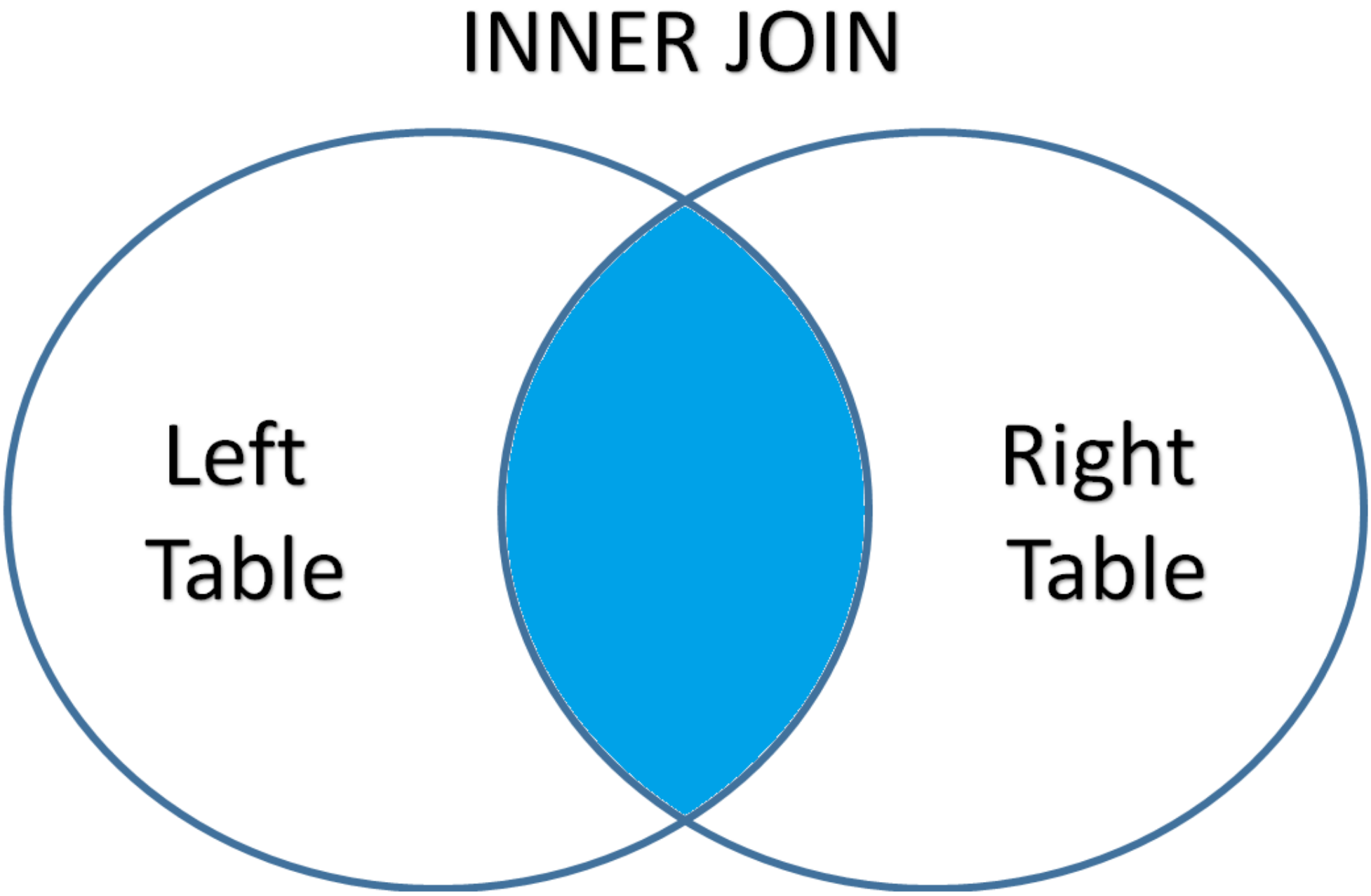
In [3]:

```
currency
```

Out[3]:

| | Country | Currency | Digraph |
|---|---|---|---|
| 0 | France | Euro | FR |
| 1 | India | Indian_Rupee | IN |
| 2 | Nepal | Nepalese_Rupee | NP |
| 3 | Russia | Rouble | RU |
| 4 | Spain | Euro | ES |
| 5 | Sri_Lanka | Rupee | LK |
| 6 | United_Kingdom | Pound | GB |
| 7 | USA | US_Dollar | US |
| 8 | Uzbekistan | Sum_Coupons | UZ |
| 9 | Zimbabwe | Zimbabwe_Dollar | ZW |

Inner Join




*Source: Created By Author*

"common" rows appear in the final joined table.

To do joins, we are going to use Pandas `pandas.merge()` function. We are going to use the two DataFrames (Tables), `capitals` and `currency` to showcase the joins in Python using Pandas.

In [4]:

```
# Inner Join
pd.merge(left = capitals, right = currency, how = 'inner')
```

Out[4]:

| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| **0** | France | FR | Paris | Euro | FR |
| **1** | India | IN | New_Delhi | Indian_Rupee | IN |
| **2** | Nepal | NP | Katmandu | Nepalese_Rupee | NP |
| **3** | Russia | RU | Moscow | Rouble | RU |
| **4** | Spain | ES | Madrid | Euro | ES |

See how simple it can be. The pandas the function automatically identified the common column `Country` and joined based on that. We did not explicitly say which columns to join on. But if you want to, it can be mentioned explicitly.

This was the case when the columns having the same content was also having the same heading. But you may notice, that's not always the case.

- ***What if the names of the columns are different in the left and right columns?***
- ***Also, What if there is more than one common column, and you want to specify which one you want the join operation to be performed "ON"?***

`merge()` the function gives you answer to both the questions above.

Let's start with specifying the "Country" column specifically in the above code.

In [5]:

```
pd.merge(left = capitals, right = currency, how = 'inner', on = 'Country' )
```

Out[5]:

| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| **0** | France | FR | Paris | Euro | FR |
| **1** | India | IN | New_Delhi | Indian_Rupee | IN |
| **2** | Nepal | NP | Katmandu | Nepalese_Rupee | NP |
| **3** | Russia | RU | Moscow | Rouble | RU |
| **4** | Spain | ES | Madrid | Euro | ES |

The results of the above code are the same as the previous one, and that was expected. Isn't it?

Now notice that there is another column, in both the left and right tables, which has the same content. We can join on that table as well, but in such case, which table name shall be mentioned in `on=`?

In such cases, we use `left_on` and `right_on` keywords in parameter.

In [6]:

```
         right = currency,
        how= 'inner',
        left_on='ISO',
        right_on='Digraph',
        suffixes=('_x', '_y'))
```

Out[6]:

| | Country_x | ISO | Capital | Country_y | Currency | Digraph |
|---|---|---|---|---|---|---|
| **0** | France | FR | Paris | France | Euro | FR |
| **1** | India | IN | New_Delhi | India | Indian_Rupee | IN |
| **2** | Nepal | NP | Katmandu | Nepal | Nepalese_Rupee | NP |
| **3** | Russia | RU | Moscow | Russia | Rouble | RU |
| **4** | Spain | ES | Madrid | Spain | Euro | ES |

## Removing the Duplicate Columns after JOIN

One apparent issue which crept into the result is duplication of the “Country” column, and you may notice that the column names have now suffix that is provided as default. There is no way in the function parameters to avoid this duplication, but a clean and smart workaround may be used in the same line of code. We can smartly use the `suffixes=` for this purpose.

I am going to add “_drop” suffix to the duplicate column.

In [7]:

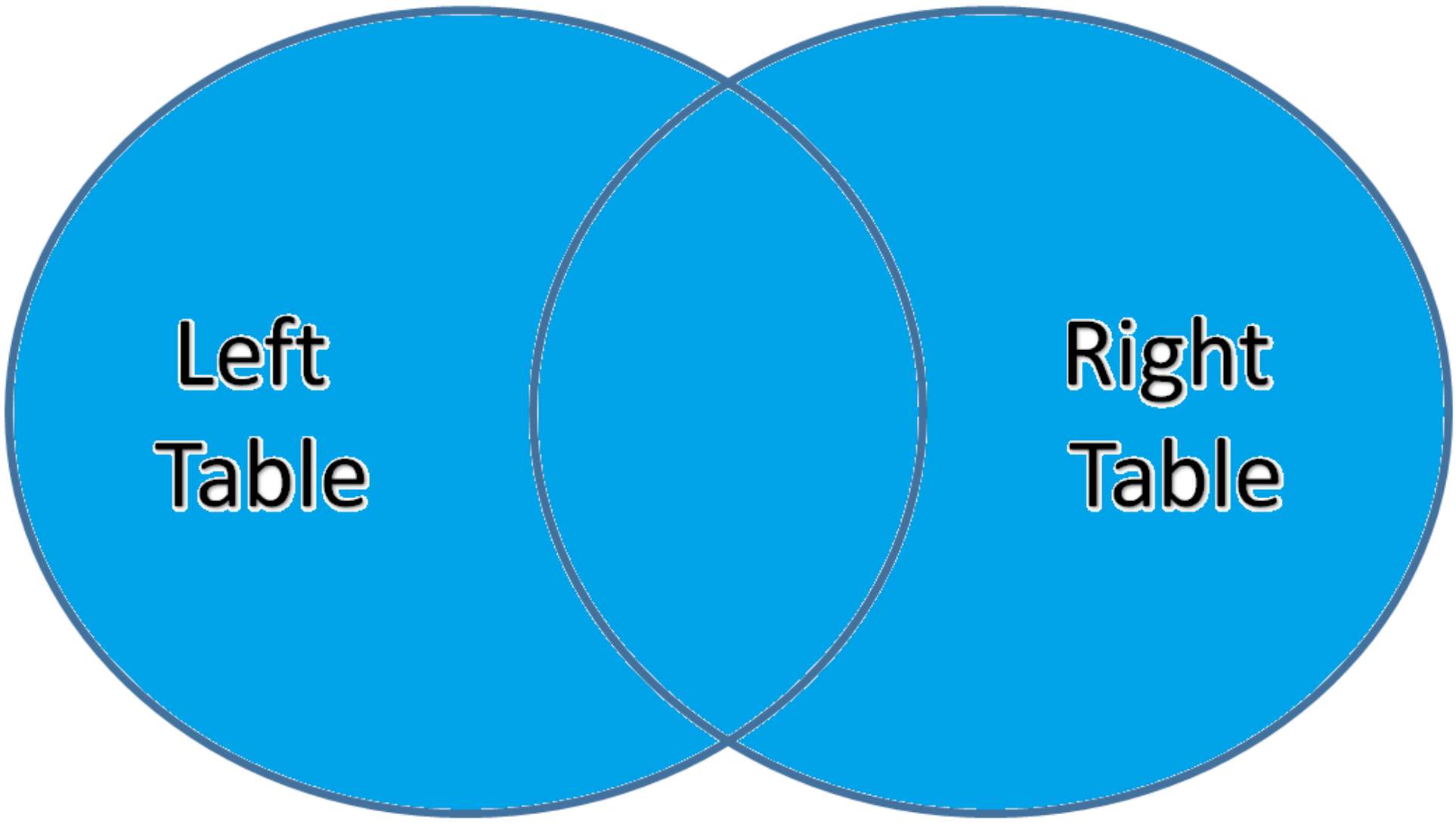
```
# filtering the column by using regular expressions

pd.merge(left = capitals,
         right = currency,
        how= 'inner',
        left_on='ISO',
        right_on='Digraph',
        suffixes=('', '_drop')).filter(regex='^(?!.*_drop)')
```

Out[7]:

| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| **0** | France | FR | Paris | Euro | FR |
| **1** | India | IN | New_Delhi | Indian_Rupee | IN |
| **2** | Nepal | NP | Katmandu | Nepalese_Rupee | NP |
| **3** | Russia | RU | Moscow | Rouble | RU |
| **4** | Spain | ES | Madrid | Euro | ES |

Here onwards, we will include this filter function in all the JOINs that generate duplicate columns.

## Outer Join or Full Join




*Source: Created by Author*

The Outer or Full Join, as the name suggests is joining the two tables, one on right and the other on left, in such a manner that all rows from both the tables appear in the final joined table.

In [8]:

```
# Outer Join

pd.merge(left = capitals, right = currency, how = 'outer')
```

Out[8]:

| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| 0 | Afghanistan | AF | Kabul | NaN | NaN |
| 1 | Argentina | AR | Buenos_Aires | NaN | NaN |
| 2 | Australia | AU | Canberra | NaN | NaN |
| 3 | Canada | CA | Ottawa | NaN | NaN |
| 4 | China | CN | Beijing | NaN | NaN |
| 5 | France | FR | Paris | Euro | FR |
| 6 | India | IN | New_Delhi | Indian_Rupee | IN |
| 7 | Nepal | NP | Katmandu | Nepalese_Rupee | NP |
| 8 | Russia | RU | Moscow | Rouble | RU |
| 9 | Spain | ES | Madrid | Euro | ES |
| 10 | Sri_Lanka | NaN | NaN | Rupee | LK |
| 11 | United_Kingdom | NaN | NaN | Pound | GB |

| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| 13 | Uzbekistan | NaN | NaN | Sum_Coupons | UZ |
| 14 | Zimbabwe | NaN | NaN | Zimbabwe_Dollar | ZW |

Notice that there is a total of 15 rows in the output above, whereas both the tables have 10 rows each. What happened here is, 5 rows are common(which came as an output of inner join) that appeared once, and the rest 5 rows of each table also got included in the final table. The values, which were not there in the tables, are filled with `NaN`, which is python's way of writing `Null` values.

Here also, we can do the join on different column names from the left and right table, and remove the duplicates, as we saw for the inner join.

In [9]:

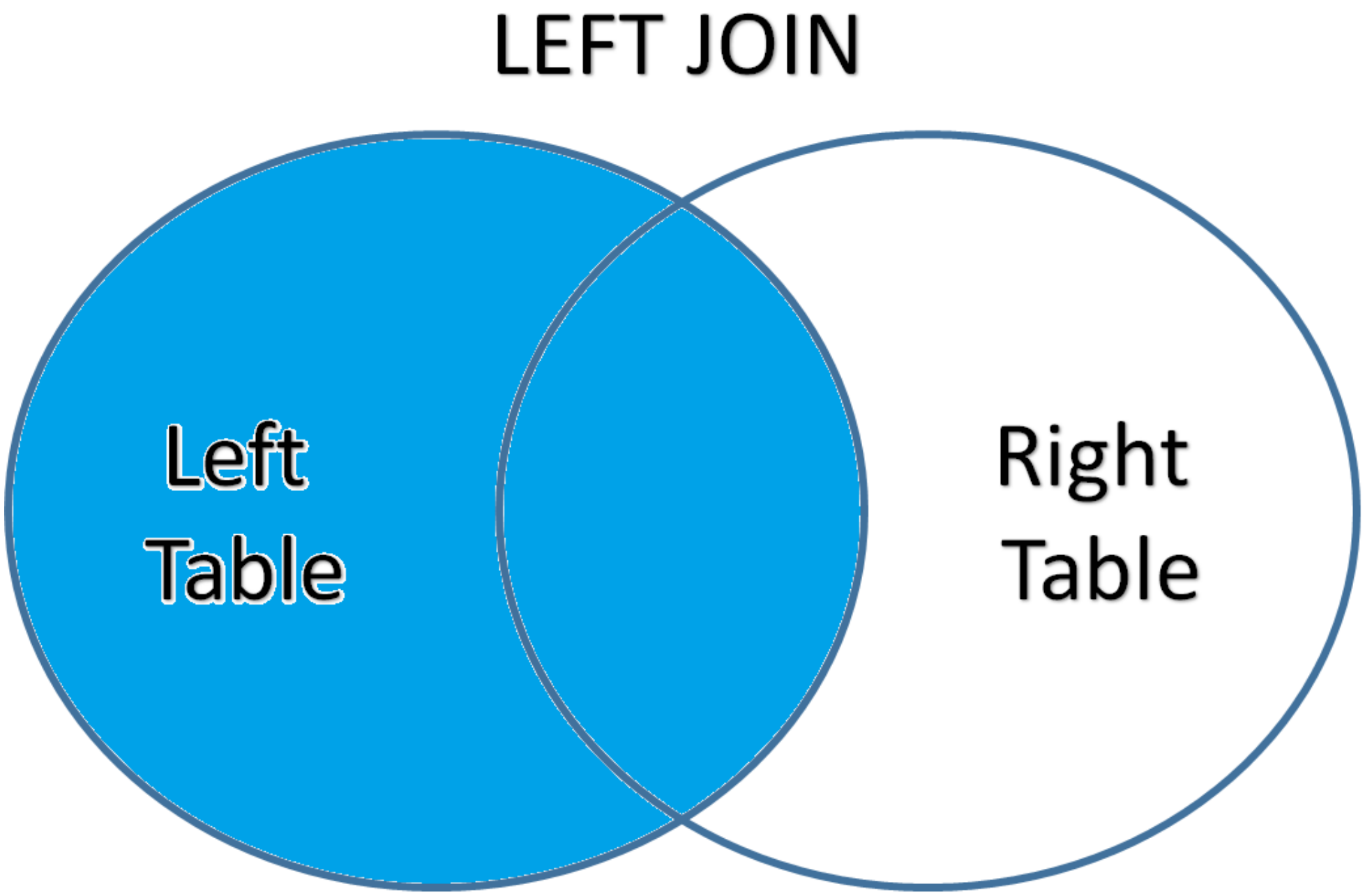
```
# filtering the column by using regular expressions

pd.merge(left = capitals,
         right = currency,
        how= 'outer',
        left_on='ISO',
        right_on='Digraph',
        suffixes=('', '_drop')).filter(regex='^(?!.*_drop)')
```

Out[9]:

| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| 0 | Afghanistan | AF | Kabul | NaN | NaN |
| 1 | Argentina | AR | Buenos_Aires | NaN | NaN |
| 2 | Australia | AU | Canberra | NaN | NaN |
| 3 | Canada | CA | Ottawa | NaN | NaN |
| 4 | China | CN | Beijing | NaN | NaN |
| 5 | France | FR | Paris | Euro | FR |
| 6 | India | IN | New_Delhi | Indian_Rupee | IN |
| 7 | Nepal | NP | Katmandu | Nepalese_Rupee | NP |
| 8 | Russia | RU | Moscow | Rouble | RU |
| 9 | Spain | ES | Madrid | Euro | ES |
| 10 | NaN | NaN | NaN | Rupee | LK |
| 11 | NaN | NaN | NaN | Pound | GB |
| 12 | NaN | NaN | NaN | US_Dollar | US |
| 13 | NaN | NaN | NaN | Sum_Coupons | UZ |
| 14 | NaN | NaN | NaN | Zimbabwe_Dollar | ZW |

But here the "Country" column has `Nan` values for the rows not there in the left table. Hence, for outer join, it's better to change the column name of the common value columns in such a way that both tables have the same column names and then do the outer join as seen above.

# Left Join




*Source: Created by Author*

The Left Join, as the name suggests is joining the two tables, one on right and the other on left, in such a manner that all rows from ONLY the LEFT table and its subsequent common values in the right table appear in the final joined table.

In [10]:

```
# Left Join

pd.merge(left = capitals, right = currency, how = 'left')
```

Out[10]:

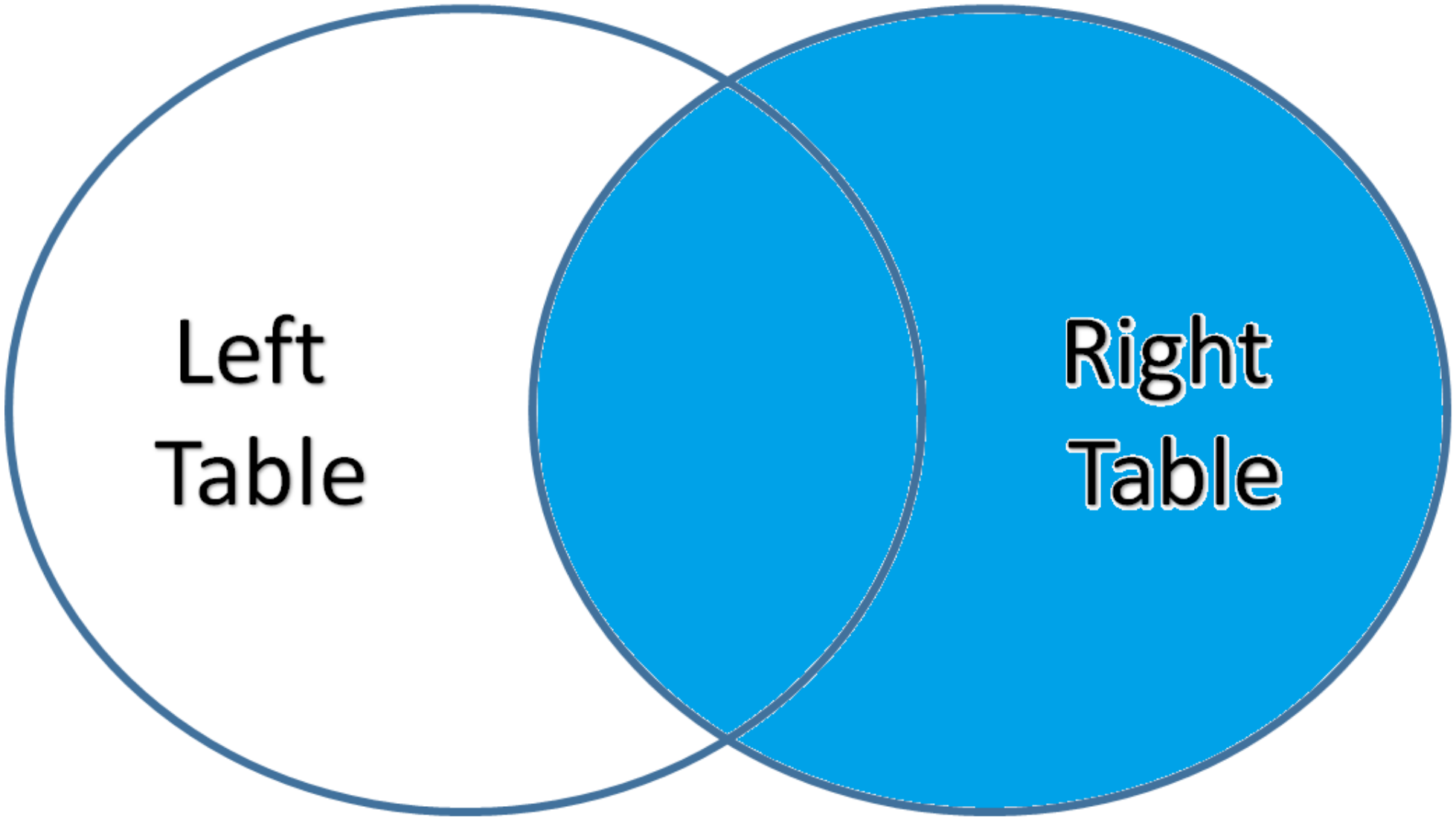
| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| **0** | Afghanistan | AF | Kabul | NaN | NaN |
| **1** | Argentina | AR | Buenos_Aires | NaN | NaN |
| **2** | Australia | AU | Canberra | NaN | NaN |
| **3** | Canada | CA | Ottawa | NaN | NaN |
| **4** | China | CN | Beijing | NaN | NaN |
| **5** | France | FR | Paris | Euro | FR |
| **6** | India | IN | New_Delhi | Indian_Rupee | IN |
| **7** | Nepal | NP | Katmandu | Nepalese_Rupee | NP |
| **8** | Russia | RU | Moscow | Rouble | RU |
| **9** | Spain | ES | Madrid | Euro | ES |

only the 10 rows of the LEFT table got included in the final table. The values, which were not there in the LEFT table(Currency and Digraph), are filled with `NaN`, which is python's way of writing `Null` values.

Now as you already saw twice, how to do a join of different column names, for the LEFT and Right Joins, I am not repeating that. Use the previous codes and see for yourself how easily you can achieve that as well.

> Note: If you have ever used Microsoft Excel or any other similar spreadsheet program, you would have done this kind of joining of data in the excel workbook. The Left Join is similar to "VLOOKUP".

## Right Join




*Source: Created by Author*

The Right Join, as the name suggests is joining the two tables, one on right and the other on left, in such a manner that all rows from ONLY the RIGHT table and its subsequent common values in the left table appear in the final joined table.

In [11]:

```
# Right Join

pd.merge(left = capitals, right = currency, how = 'right')
```

Out[11]:

| | Country | ISO | Capital | Currency | Digraph |
|---|---|---|---|---|---|
| **0** | France | FR | Paris | Euro | FR |
| **1** | India | IN | New_Delhi | Indian_Rupee | IN |
| **2** | Nepal | NP | Katmandu | Nepalese_Rupee | NP |

| | Country | ISO | Capital | Currency | DS_graph |
|---|---|---|---|---|---|
| 4 | Spain | ES | Madrid | Euro | ES |
| 5 | Sri_Lanka | NaN | NaN | Rupee | LK |
| 6 | United_Kingdom | NaN | NaN | Pound | GB |
| 7 | USA | NaN | NaN | US_Dollar | US |
| 8 | Uzbekistan | NaN | NaN | Sum_Coupons | UZ |
| 9 | Zimbabwe | NaN | NaN | Zimbabwe_Dollar | ZW |

Notice that there is a total of 10 rows in the output above, whereas both the tables have 10 rows each. What happened here is, only the 10 rows of the RIGHT table got included in the final table. The values, which were not there in the RIGHT table(ISO and Capital), are filled with `NaN`, which is python's way of writing `Null` values.

# Conclusion:

These 4 are the most utilized types of JOINs in SQL(as well as in Data Science):

**(INNER) JOIN:** Returns only those records that have matching values in both DataFrames

**LEFT (OUTER) JOIN:** Returns all the records from the left DataFrame and the matched records from the right DataFrame

**RIGHT (OUTER) JOIN:** Returns all the records from the right DataFrame, and the matched records from the left DataFrame

**FULL (OUTER) JOIN:** Returns all the records when there is a match in either left or right DataFrame

These JOINs are essential for Data Preparation, as the data kept by all the organizations (Almost all) is in relational databases. Hence, before any meaningful analysis, many tables (or DataFrames) need to be joined together to get all the relevant variables (or columns) in the data.

---

Apart from Joins, many other popular SQL functions are easily implementable in Python. Read "15 Pandas functions to replicate basic SQL Queries in Python" for learning how to do that.

The implied learning in this article was, that you can use Python to do things that you thought were only possible using SQL. There may or may not be straight forward solution to things, but if you are inclined to find it, there are enough resources at your disposal to find a way out.

# About the Author:

I am Nilabh Nishchhal. I like making seemingly difficult topics easy and write about them. Check out more at https://www.authornilabh.com/. My attempt to make Python easy and Accessible to all is "**Python Made Easy**".

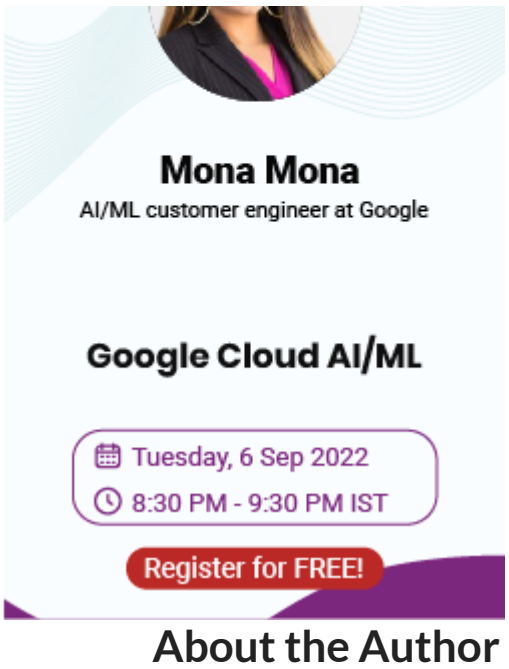
Cover Photo Credit: Photo by Shaojie on Unsplash

***The media shown in this article are not owned by Analytics Vidhya and are used at the Author's discretion.***

---

blogathon join dataframes python SQL

---




## About the Author

Nilabh Nishchhal

## Our Top Authors

view more

## Download

**Analytics Vidhya App for the Latest blog/Article**

**Previous Post**

Tricks for Data visualization using Plotly Library

**Next Post**

Part- 19: Step by Step Guide to Master NLP – Topic Modelling using LDA (Matrix Factorization Approach)

## Leave a Reply

**Your email address will not be published. Required fields are marked ***

Comment

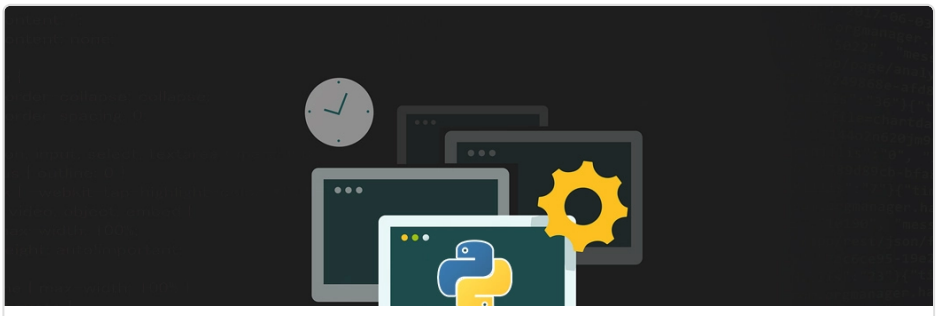
Name*

Email*

Notify me of new posts by email.

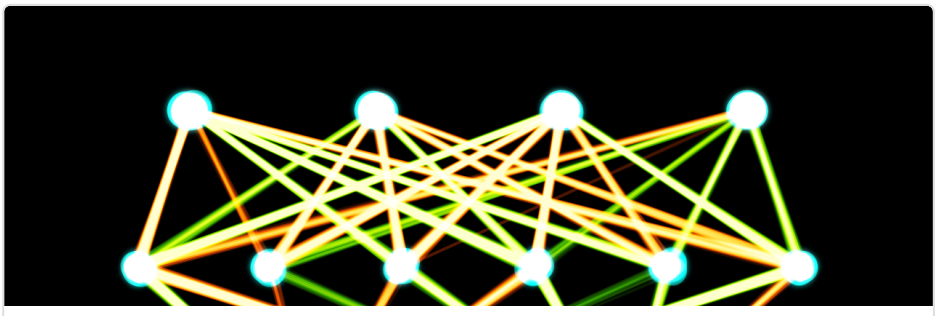
Submit

## Top Resources



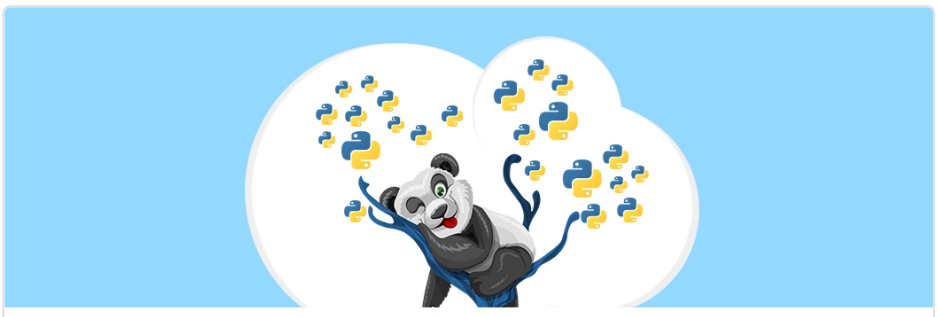
Python Tutorial: Working with CSV file for Data Science

Harika Bonthu - AUG 21, 2021



Boost Model Accuracy of Imbalanced COVID-19 Mortality Prediction Using GAN-based..

Bala Gangadhar Thilak Adiboina - OCT 07, 2020



Joins in Pandas: Master the Different Types of Joins in..

Abhishek Sharma - FEB 27, 2020

Understanding Random Forest

Sruthi E R - JUN 17, 2021