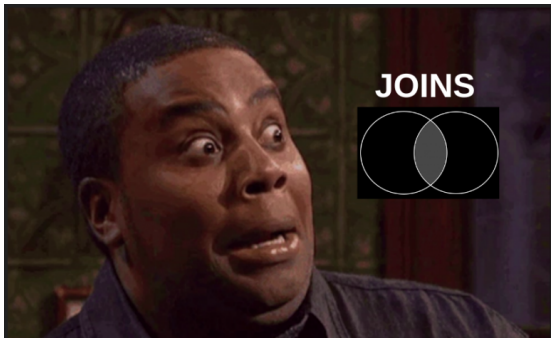
Joins in Pandas: Master the Different Types of Joins in Python

BEGINNER DATA EXPLORATION PROGRAMMING PYTHON STRUCTURED DATA

Introduction to Joins in Pandas

> "I have two different tables in Python but I'm not sure how to join them. What criteria should I consider? What are the different ways I can join these tables?"

Sound familiar? I have come across this question plenty of times on online discussion forums. Working with one table is fairly straightforward but things become challenging when we have data spread across two or more tables.



This is where the concept of Joins comes in. I cannot emphasize the number of times I have used these Joins in Pandas! They've come in especially handy during data science hackathons when I needed to quickly join multiple tables.

We will learn about different types of Joins in Pandas here:

- Inner Join in Pandas
- Full Join in Pandas
- Left Join in Pandas
- Right Join in Pandas

We will also discuss how to handle redundancy or duplicate values using joins in Pandas. Let's begin!

Note: If you're new to the world of Pandas and Python, I recommend taking the below free courses:

- *Pandas for Data Analysis in Python*
- *Python for Data Science*

Looking to learn SQL joins? We have you covered! Head over here to learn all about SQL joins.

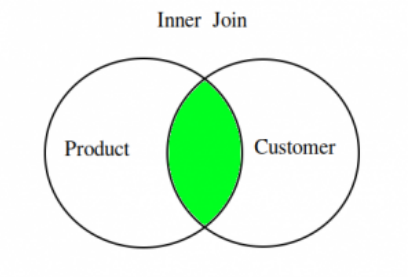
Understanding the Problem Statement

I'm sure you're quite familiar with e-commerce sites like Amazon and Flipkart these days. We are bombarded by their advertisements when we're visiting non-related websites – that's the power of targeted marketing!

We'll take a simple problem from a related marketing brand here. We are given two tables – one which contains data about products and the other that has customer-level information.

We will use these tables to understand how the different types of joins work using Pandas.

Inner Join in Pandas



> Inner join is the most common type of join you'll be working with. It returns a dataframe with only those rows that have common characteristics.

An inner join requires each row in the two joined dataframes to have matching column values. This is similar to the **intersection** of two sets.

Let's start by importing the Pandas library:

```
import pandas as pd
```

For this tutorial, we have two dataframes – product and customer. The product dataframe contains product details like *Product_ID, Product_name, Category, Price, and Seller_City*. The customer dataframe contains details like *id, name, age, Product_ID, Purchased_Product, and City*.

Our task is to use our joining skills and generate meaningful information from the data. I encourage you to follow along with the code we'll cover in this tutorial.

```
product=pd.DataFrame({ 'Product_ID':[101,102,103,104,105,106,107], 'Product_name':
['Watch','Bag','Shoes','Smartphone','Books','Oil','Laptop'], 'Category':
['Fashion','Fashion','Fashion','Electronics','Study','Grocery','Electronics'], 'Price':
[299.0,1350.50,2999.0,14999.0,145.0,110.0,79999.0], 'Seller_City':
['Delhi','Mumbai','Chennai','Kolkata','Delhi','Chennai','Bengalore'] })
```

	Product_ID	Product_name	Category	Price	Seller_City
0	101	Watch	Fashion	299.0	Delhi
1	102	Bag	Fashion	1350.5	Mumbai
2	103	Shoes	Fashion	2999.0	Chennai
3	104	Smartphone	Electronics	14999.0	Kolkata
4	105	Books	Study	145.0	Delhi
5	106	Oil	Grocery	110.0	Chennai
6	107	Laptop	Electronics	79999.0	Bengalore

```
customer=pd.DataFrame({ 'id':[1,2,3,4,5,6,7,8,9], 'name':
['Olivia','Aditya','Cory','Isabell','Dominic','Tyler','Samuel','Daniel','Jeremy'], 'age':
[20,25,15,10,30,65,35,18,23], 'Product_ID':[101,0,106,0,103,104,0,0,107], 'Purchased_Product':
['Watch','NA','Oil','NA','Shoes','Smartphone','NA','NA','Laptop'], 'City':
['Mumbai','Delhi','Bangalore','Chennai','Chennai','Delhi','Kolkata','Delhi','Mumbai'] })
```

	id	name	age	Product_ID	Purchased_Product	City
0	1	Olivia	20	101	Watch	Mumbai
1	2	Aditya	25	0	NA	Delhi
2	3	Cory	15	106	Oil	Bangalore
3	4	Isabell	10	0	NA	Chennai
4	5	Dominic	30	103	Shoes	Chennai
5	6	Tyler	65	104	Smartphone	Delhi
6	7	Samuel	35	0	NA	Kolkata
7	8	Daniel	18	0	NA	Delhi
8	9	Jeremy	23	107	Laptop	Mumbai

Let’s say we want to know about all the products sold online and who purchased them. We can get this easily using an inner join.

The **merge()** function in Pandas is our friend here. By default, the merge function performs an inner join. It takes both the dataframes as arguments and the name of the column on which the join has to be performed:

```
pd.merge(product,customer,on='Product_ID')
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1	Olivia	20	Watch	Mumbai
1	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai
2	104	Smartphone	Electronics	14999.0	Kolkata	6	Tyler	65	Smartphone	Delhi
3	106	Oil	Grocery	110.0	Chennai	3	Cory	15	Oil	Bangalore
4	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	Laptop	Mumbai

Here, I have performed inner join on the product and customer dataframes on the ‘Product_ID’ column.

But, what if the column names are different in the two dataframes? Then, we have to explicitly mention both the column names.

'left_on' and 'right_on' are two arguments through which we can achieve this. 'left_on' is the name of the key in the left dataframe and 'right_on' in the right dataframe:

```
pd.merge(product,customer,left_on='Product_name',right_on='Purchased_Product')
```

	Product_ID_x	Product_name	Category	Price	Seller_City	id	name	age	Product_ID_y	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1	Olivia	20	101	Watch	Mumbai
1	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	103	Shoes	Chennai
2	104	Smartphone	Electronics	14999.0	Kolkata	6	Tyler	65	104	Smartphone	Delhi
3	106	Oil	Grocery	110.0	Chennai	3	Cory	15	106	Oil	Bangalore
4	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	107	Laptop	Mumbai

Let's take things up a notch. The leadership team now wants more details about the products sold. *They want to know about all the products sold by the seller to the same city i.e., seller and customer both belong to the same city.*

In this case, we have to perform an inner join on both *Product_ID* and *Seller_City* of product and *Product_ID* and *City* columns of the customer dataframe.

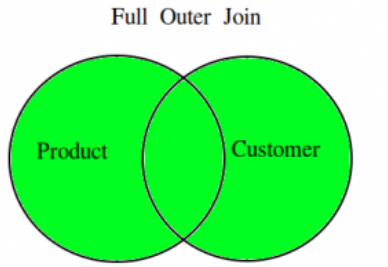
So, how we can do this?

Don't scratch your head! Just pass an array of column names to the **left_on** and **right_on** arguments:

```
pd.merge(product,customer,how='inner',left_on=['Product_ID','Seller_City'],right_on=['Product_ID','City'])
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai

Full Join in Pandas



Here's another interesting task for you. We have to combine both dataframes so that we can find all the products that are not sold and all the customers who didn't purchase anything from us.

We can use Full Join for this purpose.

> Full Join, also known as Full Outer Join, returns all those records which either have a match in the left or right dataframe.

When rows in both the dataframes do not match, the resulting dataframe will have NaN for every column of the dataframe that lacks a matching row.

We can perform Full join by just passing the **how** argument as **'outer'** to the **merge()** function:

```
pd.merge(product,customer,on='Product_ID',how='outer')
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1.0	Olivia	20.0	Watch	Mumbai
1	102	Bag	Fashion	1350.5	Mumbai	NaN	NaN	NaN	NaN	NaN
2	103	Shoes	Fashion	2999.0	Chennai	5.0	Dominic	30.0	Shoes	Chennai
3	104	Smartphone	Electronics	14999.0	Kolkata	6.0	Tyler	65.0	Smartphone	Delhi
4	105	Books	Study	145.0	Delhi	NaN	NaN	NaN	NaN	NaN
5	106	Oil	Grocery	110.0	Chennai	3.0	Cory	15.0	Oil	Bangalore
6	107	Laptop	Electronics	79999.0	Bengalore	9.0	Jeremy	23.0	Laptop	Mumbai
7	0	NaN	NaN	NaN	NaN	2.0	Aditya	25.0	NA	Delhi
8	0	NaN	NaN	NaN	NaN	4.0	Isabell	10.0	NA	Chennai
9	0	NaN	NaN	NaN	NaN	7.0	Samuel	35.0	NA	Kolkata
10	0	NaN	NaN	NaN	NaN	8.0	Daniel	18.0	NA	Delhi

Did you notice what happened here? All the non-matching rows of both the dataframes have NaN values for the columns of other dataframes. But wait – we still don't know which row belongs to which dataframe.

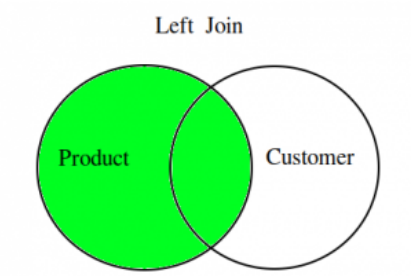
For this, Pandas provides us with a fantastic solution. We just have to mention the **indicator** argument as **True** in the function, and a new column of name **_merge** will be created in the resulting dataframe:

```
pd.merge(product,customer,on='Product_ID',how='outer',indicator=True)
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City	_merge
0	101	Watch	Fashion	299.0	Delhi	1.0	Olivia	20.0	Watch	Mumbai	both
1	102	Bag	Fashion	1350.5	Mumbai	NaN	NaN	NaN	NaN	NaN	left_only
2	103	Shoes	Fashion	2999.0	Chennai	5.0	Dominic	30.0	Shoes	Chennai	both
3	104	Smartphone	Electronics	14999.0	Kolkata	6.0	Tyler	65.0	Smartphone	Delhi	both
4	105	Books	Study	145.0	Delhi	NaN	NaN	NaN	NaN	NaN	left_only
5	106	Oil	Grocery	110.0	Chennai	3.0	Cory	15.0	Oil	Bangalore	both
6	107	Laptop	Electronics	79999.0	Bengalore	9.0	Jeremy	23.0	Laptop	Mumbai	both
7	0	NaN	NaN	NaN	NaN	2.0	Aditya	25.0	NA	Delhi	right_only
8	0	NaN	NaN	NaN	NaN	4.0	Isabell	10.0	NA	Chennai	right_only
9	0	NaN	NaN	NaN	NaN	7.0	Samuel	35.0	NA	Kolkata	right_only
10	0	NaN	NaN	NaN	NaN	8.0	Daniel	18.0	NA	Delhi	right_only

As you can see, the **_merge** column mentions which row belongs to which dataframe.

Left Join in Pandas



Now, let's say the leadership team wants information about only those customers who bought something from us. You guessed it – we can use the concept of Left Join here.

> Left join, also known as Left Outer Join, returns a dataframe containing all the rows of the left dataframe.

All the non-matching rows of the left dataframe contain NaN for the columns in the right dataframe. It is simply an inner join plus all the non-matching rows of the left dataframe filled with NaN for columns of the right dataframe.

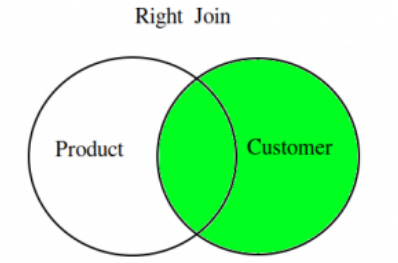
Performing a left join is actually quite similar to a full join. Just change the **how** argument to **'left'**:

```
pd.merge(product,customer,on='Product_ID',how='left')
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1.0	Olivia	20.0	Watch	Mumbai
1	102	Bag	Fashion	1350.5	Mumbai	NaN	NaN	NaN	NaN	NaN
2	103	Shoes	Fashion	2999.0	Chennai	5.0	Dominic	30.0	Shoes	Chennai
3	104	Smartphone	Electronics	14999.0	Kolkata	6.0	Tyler	65.0	Smartphone	Delhi
4	105	Books	Study	145.0	Delhi	NaN	NaN	NaN	NaN	NaN
5	106	Oil	Grocery	110.0	Chennai	3.0	Cory	15.0	Oil	Bangalore
6	107	Laptop	Electronics	79999.0	Bengalore	9.0	Jeremy	23.0	Laptop	Mumbai

Here, you can clearly see that all the unsold products contain NaN for the columns belonging to the customer dataframe.

Right Join in Pandas



Similarly, if we want to create a table of customers including the information about the products they bought, we can use the right join.

> Right join, also known as Right Outer Join, is similar to the Left Outer Join. The only difference is that all the rows of the right dataframe are taken as it is and only those of the left dataframe that are common in both.

Similar to other joins, we can perform a right join by changing the **how** argument to **'right'**:

```
pd.merge(product,customer,on='Product_ID',how='right')
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1	Olivia	20	Watch	Mumbai
1	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai
2	104	Smartphone	Electronics	14999.0	Kolkata	6	Tyler	65	Smartphone	Delhi
3	106	Oil	Grocery	110.0	Chennai	3	Cory	15	Oil	Bangalore
4	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	Laptop	Mumbai
5	0	NaN	NaN	NaN	NaN	2	Aditya	25	NA	Delhi
6	0	NaN	NaN	NaN	NaN	4	Isabell	10	NA	Chennai
7	0	NaN	NaN	NaN	NaN	7	Samuel	35	NA	Kolkata
8	0	NaN	NaN	NaN	NaN	8	Daniel	18	NA	Delhi

Take a look carefully at the above dataframe – we have NaN values for columns of the product dataframe. Pretty straightforward, right?

Handling Redundancy/Duplicates in Joins

Duplicate values can be tricky obstacles. They can cause problems while performing joins. These values won't give an error but will simply create redundancy in our resulting dataframe. I'm sure you can imagine how harmful that can be!

Here, we have a dataframe **product_dup** with duplicate details about products:

```
product_dup=pd.DataFrame({ 'Product_ID':[101,102,103,104,105,106,107,103,107], 'Product_name':
['Watch','Bag','Shoes','Smartphone','Books','Oil','Laptop','Shoes','Laptop', 'Category':
['Fashion','Fashion','Fashion','Electronics','Study','Grocery','Electronics','Fashion','Electronics'],
'Price':[299.0,1350.50,2999.0,14999.0,145.0,110.0,79999.0,2999.0,79999.0], 'Seller_City':
['Delhi','Mumbai','Chennai','Kolkata','Delhi','Chennai','Bengalore','Chennai','Bengalore'] })
```

	Product_ID	Product_name	Category	Price	Seller_City
0	101	Watch	Fashion	299.0	Delhi
1	102	Bag	Fashion	1350.5	Mumbai
2	103	Shoes	Fashion	2999.0	Chennai
3	104	Smartphone	Electronics	14999.0	Kolkata
4	105	Books	Study	145.0	Delhi
5	106	Oil	Grocery	110.0	Chennai
6	107	Laptop	Electronics	79999.0	Bengalore
7	103	Shoes	Fashion	2999.0	Chennai
8	107	Laptop	Electronics	79999.0	Bengalore

Let's see what happens if we perform an inner join on this dataframe:

```
pd.merge(product_dup,customer,how='inner',on='Product_ID')
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1	Olivia	20	Watch	Mumbai
1	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai
2	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai
3	104	Smartphone	Electronics	14999.0	Kolkata	6	Tyler	65	Smartphone	Delhi
4	106	Oil	Grocery	110.0	Chennai	3	Cory	15	Oil	Bangalore
5	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	Laptop	Mumbai
6	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	Laptop	Mumbai

As you can see, we have duplicate rows in the resulting dataset as well. To solve this, there is a **validate** argument in the **merge()** function, which we can set to **'one_to_one', 'one_to_many', 'many_to_one',** and

'many_to_many'.

This ensures that there exists only a particular mapping across both the dataframes. If the mapping condition is not satisfied, then it throws a **MergeError**. To solve this, we can delete duplicates before applying join:

```
pd.merge(product_dup.drop_duplicates(),customer,how='inner',on='Product_ID')
```

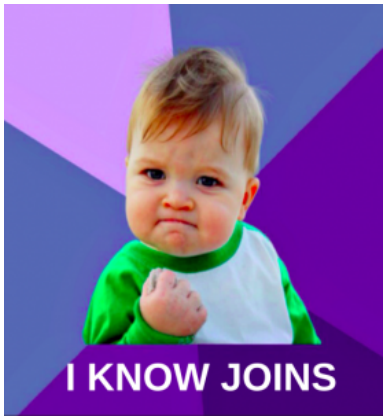
	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1	Olivia	20	Watch	Mumbai
1	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai
2	104	Smartphone	Electronics	14999.0	Kolkata	6	Tyler	65	Smartphone	Delhi
3	106	Oil	Grocery	110.0	Chennai	3	Cory	15	Oil	Bangalore
4	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	Laptop	Mumbai

But, if you want to keep these duplicates, then you can give **validate** values as per your requirements and it will not throw an error:

```
pd.merge(product_dup,customer,how='inner',on='Product_ID',validate='many_to_many')
```

	Product_ID	Product_name	Category	Price	Seller_City	id	name	age	Purchased_Product	City
0	101	Watch	Fashion	299.0	Delhi	1	Olivia	20	Watch	Mumbai
1	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai
2	103	Shoes	Fashion	2999.0	Chennai	5	Dominic	30	Shoes	Chennai
3	104	Smartphone	Electronics	14999.0	Kolkata	6	Tyler	65	Smartphone	Delhi
4	106	Oil	Grocery	110.0	Chennai	3	Cory	15	Oil	Bangalore
5	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	Laptop	Mumbai
6	107	Laptop	Electronics	79999.0	Bengalore	9	Jeremy	23	Laptop	Mumbai

Now, you can say:



What's Next?

There is also a **concat()** function in Pandas that we can use for joining two dataframes. I encourage you to explore that and apply it in your next project alongside what you've learned about joins in this tutorial.

If you have any queries or feedback on this article, feel free to share it in the comments section below. I have listed some insightful and comprehensive articles and courses related to data science and Python below.

Courses:

- Python for Data Science
- Pandas for Data Analysis in Python
- Data Science Hacks, Tips and Tricks
- Introduction to Data Science
- A comprehensive Learning path to become a data scientist in 2020

Tutorials:

- 12 Useful Pandas Techniques in Python for Data Manipulation

---

Article Url - https://www.analyticsvidhya.com/blog/2020/02/joins-in-pandas-master-the-different-types-of-joins-in-python/

Abhishek Sharma

He is a data science aficionado, who loves diving into data and generating insights from it. He is inspired by neural networks, learns and trains himself every day. He is always ready for making machines to learn through code and writing technical blogs.