[Jul 20, 2019·7 min read](https://towardsdatascience.com/sql-in-python-for-beginners-b9a4f9293ecf?source=post_page-----b9a4f9293ecf--------------------------------)

<https://towardsdatascience.com/sql-in-python-for-beginners-b9a4f9293ecf>

**Introduction to SQL in Python for Data Scientists**

**The data scientist’s guide for using SQL in Python environment.**

[Nick Minaie, PhD](https://medium.com/@minaienick?source=post_page-----b9a4f9293ecf--------------------------------)

[Nick Minaie, PhD](https://medium.com/@minaienick?source=post_page-----b9a4f9293ecf--------------------------------)

This article provides an overview of the basic SQL statements for data scientists, and explains how a SQL engine can be instantiated in Python and used for querying data from a database.



A picture containing text

Description automatically generated

As a data scientist using Python, you often need to get your data from a relational database that is hosted either on your local server, or on the cloud (e.g. AWS cloud). There are many ways to approach this. For example, you can query your data in Oracle, save the file as a .csv file, and then import it in Python. However, the most efficient way it to use SQL directly in Python. Coupling SQL and Pandas would give you many options to query, process, and use the data for your project in Python.

**First Things First! What is SQL?**

SQL (aka Structured Query Language) is a programming language used for managing or querying data stored in a relational database management system (RDBMS). SQL has been the dominant language for handling structured data where the entities in the database (e.g. tables, or table entities) are related (that is why these databases are called relational databases). There are other options for handling such data, but SQL has been the most popular, widely used language in the industry.

**How is “SQL” pronounced?**

SQL was developed at IBM in the early 1970s, and it was originally called “[SEQUEL (Structured English Query Language)](https://en.wikipedia.org/wiki/SQL)”. Later on, the name was changed to SQL (Structured Query Language) due to a trademark issue. However, the pronunciation “see-qu-el” ([/ˈsiːkwəl/](https://en.wikipedia.org/wiki/Help:IPA/English) ) stayed with the language, and that is the adopted pronunciation by most practitioners.

[***Pro tip:*** when you go to an interview, make sure you pronounce it “see-qu-el”, if you want the job!]

**What Does a Relational Database Look Like?**

[Amazon Web Services provides the best definition for a relational datab](https://aws.amazon.com/relational-database/)l:

A relational database is a collection of data items with pre-defined relationships between them. These items are organized as a set of **tables** with columns and rows. Tables are used to hold information about the objects to be represented in the database. Each column in a table holds a certain kind of data and a field stores the actual value of an attribute. The rows in the table represent a collection of related values of one object or entity. **Each row in a table** could be marked with a **unique identifier** called a **primary key**, and **rows among multiple** tables can be **made related using foreign keys**. This data can be accessed in many different ways without reorganizing the database tables themselves.

Databases can have very complex designs, with many tables, and each table with many entities (columns) and many rows. It would be extremely difficult, or maybe even impossible, to query data when the relationships between tables is not known. **ERD** (Entity Relationship Diagram) is used to visualize these relationships and also show the entities in each table and their datatypes. Your database administrator should be able to give your the database’s ERD.

Diagram

Description automatically generated

A Sample ERD — <https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/>

**How is SQL used in Python?**

There are many ways to use SQL in Python. Multiple libraries have been developed for this purpose that can be utilized. [SQLite](https://www.geeksforgeeks.org/sql-using-python/) and [MySQL](https://www.w3schools.com/python/python_mysql_getstarted.asp) are examples of these libraries.

In this article, we are going to use Python Pandas in conjunction with [sqlalchemy library.](https://www.sqlalchemy.org/)

**Creating the SQL Engine**

Let’s get started! We need to install and then import the libraries first. We will be using the create\_engine feature from this library.

!pip install sqlalchemyimport pandas as pd  
from sqlalchemy import create\_engine

Once the library is imported, we need to create a SQL engine using this command which creates a new class`.Engine` instance.

engine = create\_engine(\*args, \*\*kwargs)

The first argument is usually a string that indicates database dialect and connection arguments in the form of a URL and can be written as:

dialect[+driver]://user:password@host/dbname[?key=value..]

where dialect is a database name such as mysql, oracle, postgresql, etc., and driver the name of a DBAPI, such as psycopg2, pyodbc, cx\_oracle, etc. More details on this can be found at <https://www.sqlalchemy.org/>.

**Data Queries using SQL Statements**

Now that you are connected to the database, you can submit data queries. to use sqlalchemy you need to wrap your SQL statements in a container, send it to the database, get the response back, and then put the response in a panda dataframe. The two primary clauses that must be present in every query are SELECT, and FROM.

* SELECT allows you to select a subset of columns (or all of them) from a table,
* FROM specifies which table the column(s) are being pulled from.

For example, the following code snippet will return all entities (columns) from table\_1saves the response into a dataframe, and displays the head.

sql = """  
SELECT \*  
FROM table\_1  
"""df = pd.read\_sql\_query(sql, engine)  
df.head()

Instead, you could also pull specific columns from the table, using this code instead:

SELECT entity\_1, entity\_2, entity\_3  
FROM table\_1

If you are dealing with multiple tables (which you surely will in a real world project), you may need to specify which entity from which table because the entities of interest may come from different tables in the database. We will discuss how multiple entities from multiple tables can be queries, but this example is for entities from one table. In this case, you can use namespacing in your SQL statement:

SELECT table\_1.entity\_1, table\_1.entity\_2  
FROM table\_1

You can also assign aliases to each entity name or table name for simplification or readability purposes:

SELECT t.entity\_1 AS name, t.entity\_2 AS id  
FROM table\_1 AS t

If you want to get the **distinct** rows from a column, you can send this SQL statement:

SELECT DISTINCT entity\_1  
FROM table\_1

If you want to order your data by a specific column (or multiple columns) you can use ORDER BY and specify a you want ASC (ascending) or DESC (descending) order. Remember, if you use multiple columns in ORDER BY the order in which SQL orders the data will be from left to right.

SELECT entity\_1, entity\_2, entity\_3  
FROM table\_1  
ORDER BY entity\_1 DESC, entity\_3 ASC

Sometime you are dealing with a very large dataset, but you may only need to retrieve a limited dataset from the database. If this case, you can use LIMIT:

SELECT \*  
FROM table\_1  
LIMIT 10

If you want to include a condition for the query, you can use WHERE. You can either use boolean conditions, or wildcards for string entities. For example:

SELECT \*  
FROM table\_1  
WHERE entity\_1 > 5

or WHERE entity\_1 BETWEEN 5 and 10

or WHERE entity\_1 > 5 AND entity\_1 < 10.

[**Wildcards** (or wild characters) are symbols used to replace or represent one or more characters in a word. The one familiar one is \* that is used for zero or many charactersor ? that is used for one character](https://www.computerhope.com/jargon/w/wildcard.htm). We can use wildcards effectively in SQL when querying string entities using LIKE statement in SQL. The different between % and \* is that % also account for underscore, but \* does not. In Python, you should use %% instead of one %. The statement below returns all rows in which entity\_3 starts with M.

SELECT \*  
FROM table\_1  
WHERE entity\_3 LIKE "M%%"

ILIKE make this query insensitive to the character case, and NOT LIKE returns all rows in which the entity is **NOT** like the wildcard.

To deal with the null values, you can use:

SELECT \*  
FROM table\_1  
WHERE entity\_1 IS NULL

or WHERE entity\_1 IS NOT NULL.

Often you need to aggregate the data, group the data, and apply conditions to aggregated data. These aggregate statements include COUNT, AVG, MIN, MAX, and SUM. For example:

SELECT SUM(entity\_1) AS sum, entity\_2  
FROM table\_1

When using aggregates, you should use HAVING instead of WHERE, like:

SELECT SUM(entity\_1) AS sum, entity\_2  
FROM table\_1  
HAVING entity\_2 BETWEEN 5 and 10

To group your data by specific entity, you can use GROUP BY:

SELECT entity\_1, SUM(entity\_2) AS sum  
FROM table\_1  
GROUP BY entity\_1  
HAVING entity\_3 BETWEEN 5 and 10

**Joining Tables**

When querying data from multiple tables, you need to join these tables. There are multiple ways to join tables in SQL. Figure below illustrates these joins. You will likely work with inner joins more often, but it is important to understand what each type of join does.

Diagram

Description automatically generated

Different Types of Table Joins in SQL — <https://www.dofactory.com/sql/join>

Joining tables can only be done when there is a common entity between the two tables, and you need to define that relationship using ON.

SELECT t\_1.entity\_1, t\_2.entity\_2  
FROM table\_1 AS t\_1   
INNER JOIN table\_2 AS t\_2 ON t\_2.key = t\_1.key

These statements cover the basics of SQL in Python. You can combine these statements based on the database you are dealing with, the type of data you need. There are many more statements that you can use. <https://www.w3schools.com/sql/> provides a more comprehensive overview of SQL statements.

***Nick Minaie, PhD*** *(*[*LinkedIn Profile*](https://www.linkedin.com/in/nickminaie/)*) is a senior consultant and a visionary data scientist, and represents a unique combination of leadership skills, world-class data-science expertise, business acumen, and the ability to lead organizational change. His mission is to advance the practice of Artificial Intelligence (AI) and Machine Learning in the industry.*

# Data Grouping in Python

## Examine the “difficult” tasks and try to give alternative solutions

[Easily Simplify Data Processing](https://businessprogramming.medium.com/?source=post_page-----d64f1203f8d3--------------------------------)

[Easily Simplify Data Processing](https://businessprogramming.medium.com/?source=post_page-----d64f1203f8d3--------------------------------)

[Nov 13, 2020·16 min read](https://towardsdatascience.com/data-grouping-in-python-d64f1203f8d3?source=post_page-----d64f1203f8d3--------------------------------)

<https://towardsdatascience.com/data-grouping-in-python-d64f1203f8d3>

Grouping records by column(s) is a common need for data analyses. Such scenarios include counting employees in each department of a company, calculating the average salary of male and female employees respectively in each department, and calculating the average salary of employees of different ages. Pandas has groupby function to be able to handle most of the grouping tasks conveniently. But there are certain tasks that the function finds it hard to manage. Here let’s examine these “difficult” tasks and try to give alternative solutions.

groupby is one of the most important Pandas functions. It is used to group and summarize records according to the split-apply-combine strategy. The following diagram shows the workflow:

Graphical user interface

Description automatically generated with medium confidence

Image by Author

# I Grouping & aggregation by a single field

You group records by a certain field and then perform aggregate over each group.

This is the simplest use of the above strategy. To count employees in each department based on employee information, for instance:

Problem analysis: Use department as the key, group records by it and count the records in each group.

Below is part of the employee information:

Table

Description automatically generated

Python script:

import pandas as pd#Import data  
employee = pd.read\_csv("Employees.csv")#Grouping and perform count over each group  
dept\_emp\_num = employee.groupby('DEPT')['DEPT'].count()print(dept\_emp\_num)

Explanation: groupby(‘DEPT’)groups records by department, and count() calculates the number of employees in each group.

# II Grouping & aggregation by multiple fields

You group records by multiple fields and then perform aggregate over each group.

We handle it in a similar way. To calculate the average salary for both male and female employees in each department based on the same employee information in the previous instance.

Problem analysis: There are two grouping keys, department and gender. We treat thea composite key as a whole to perform grouping and aggregate.

Python script:

import pandas as pdemployee = pd.read\_csv("Employees.csv")#Group by two keys and then summarize each group  
dept\_gender\_salary = employee.groupby(['DEPT','GENDER'],as\_index=False).SALARY.mean()print(dept\_gender\_salary)

Explanation: The expression groupby([‘DEPT’,‘GENDER’])takes the two grouping fields as parameters in the form of a list. The expression as\_index specifies whether to use the grouping fields as the index using True or False (Here False means not using them as the index). The mean() function calculates the average salary.

# III Grouping & aggregation by a computed column

The grouping key is not explicit data and needs to be calculated according to the existing data. Such a key is called computed column. To calculate the average salary for employees of different years, for instance:

Problem analysis: There isn’t a years column in the employee information. We need to calculate it according to the employees’birthdays, group records by the calculated column, and calculate the average salary.

Python script:

import pandas as pdimport numpy as npemployee = pd.read\_csv("Employees.csv")#Convert the BIRTHDAY column into date format  
employee['BIRTHDAY']=pd.to\_datetime(employee['BIRTHDAY'])#Calculate an array of calculated column values, group records by them, and calculate the average salary  
years\_salary = employee.groupby(np.floor((employee['BIRTHDAY'].dt.year-1900)/10)).SALARY.mean()print(years\_salary)

Explanation: Since the years values don’t exist in the original data, Python uses np.floor((employee[‘BIRTHDAY’].dt.year-1900)/10) to calculate the years column, groups the records by the new column and calculate the average salary.

# IV Multiple aggregates

You perform one type of aggregate operation over each of multiple columns or several types of aggregates over one or more columns.

1. One aggregate on each of multiple columns

You perform one type of aggregate on each of multiple columns. To count the employees and calculate the average salary in every department, for example:

Problem analysis: The count aggregate is on EID column, and the average aggregate is over the salary column. Each column has its own one aggregate.

Python script:

import pandas as pdemployee = pd.read\_csv("Employees.csv")#Group records by DEPT, perform count on EID and average on SALARY  
dept\_agg = employee.groupby('DEPT',as\_index=False).agg({'EID':'count','SALARY':'mean'})#Rename the columns  
print(dept\_agg.rename(columns={'EID':'NUM','SALARY':'AVG\_SALARY'}))

Explanation: Pandas agg() function can be used to handle this type of computing tasks. Relevant columns and the involved aggregate operations are passed into the function in the form of dictionary, where the columns are keys and the aggregates are values, to get the aggregation done.

2. Multiple aggregates on one column

You perform more than one type of aggregate on a single column. For the previous task, we can also sum the salary and then calculate the average. This way we perform two aggregates, count and average, on the salary column.

Python script:

import pandas as pdemployee = pd.read\_csv("Employees.csv")#Perform count and then average on SALARY column  
dept\_agg = employee.groupby('DEPT').SALARY.agg(['count','mean']).reset\_index()#Rename columns  
print(dept\_agg.rename(columns={'count':'NUM','mean':'AVG\_SALARY'}))

Explanation: We can combine the aggregate operations as a list and take it as the parameter to pass to the agg() function.

3. Multiple aggregates over multiple columns

You summarize multiple columns during which there are multiple aggregates on a single column. The aggregate operation can be user-defined.

To get the number of employees, the average salary and the largest age in each department, for instance:

Problem analysis: Counting the number of employees and calculating the average salary are operations on the SALARY column (multiple aggregates on one column). Finding the largest age needs a user-defined operation on BIRTHDAY column.

Python script:

import pandas as pdimport datetime#The user-defined function for getting the largest age  
def max\_age(s):  
 #Year  
 today = datetime. datetime.today().year  
 #Get ages  
 age = today-s.dt.year return age.max()employee = pd.read\_csv("Employees.csv")employee['BIRTHDAY']=pd.to\_datetime(employee\['BIRTHDAY'\])#Group records by DEPT, perform count and average on SALARY, and use the user-defined max\_age function to get the largest age  
dept\_agg = employee.groupby('DEPT').agg({'SALARY':['count','mean'],'BIRTHDAY':max\_age})#Rename columns  
dept\_agg.columns = ['NUM','AVG\_SALARY','MAX\_AGE']print(dept\_agg.reset\_index())

Explanation: Columns to be summarized and the aggregate operations are passed through parameters to the function in the form of dictionary. For a column requiring multiple aggregate operations, we need to combine the operations as a list to be used as the dictionary value.

# V Copying the grouping & aggregate results

You extend each of the aggregated results to the length of the corresponding group. This is equivalent to copying an aggregate result to all rows in its group. To add a new column containing the average salary of each department to the employee information, for instance:

Problem analysis: Group records by department, calculate the average salary in each department, and populate each average value to the corresponding group while maintaining the original order.

Python script:

import pandas as pdemployee = pd.read\_csv("Employees.csv")#Group records by DEPT and calculate average on SLARYemployee['AVG\_SALARY'] = employee.groupby('DEPT').SALARY.transform('mean')print(employee)

Explanation: Group records by department and calculate average salary in each group. transform() function calculates aggregate on each group, returns the result and populates it to all rows in the order of the original index. That makes sure that the records maintain the original order.

# VI Handling grouping subsets

You perform one or more non-aggregate operations in each group. To sort each group, for example, we are concerned with the order of the records instead of an aggregate. To sort records in each department by hire date in ascending order, for example:

Problem analysis: Group records by department, and loop through each group to order records by hire date.

Python script:

import pandas as pdemployee = pd.read\_csv("Employees.csv")#Modify hire date format  
employee['HIREDATE']=pd.to\_datetime(employee['HIREDATE'])#Group records by DEPT, sort each group by HIREDATE, and reset the index  
employee\_new = employee.groupby('DEPT',as\_index=False).apply(lambda x:x.sort\_values('HIREDATE')).reset\_index(drop=True)print(employee\_new)

Explanation: To sort records in each group, we can use the combination of apply()function and lambda. The lambda expression loops through groups to sort records in each group using sort\_values() function, and returns the sorting result.

There are more complicated computing goals. To find the difference between salary of the eldest employee and that of the youngest employee in each department, for instance:

Problem analysis: Group records by department, locate the eldest employee record and the youngest employee record, and calculate their salary difference.

Python script:

import pandas as pd#salary\_diff(g)function calculates the salary difference over each group  
def salary\_diff(g):  
   
 #The index of the eldest employee record  
 max\_age = g['BIRTHDAY'].idxmin() #The index of the youngest employee record  
 min\_age = g['BIRTHDAY'].idxmax() #Calculate the salary difference  
 diff = g.loc[max\_age]['SALARY']-g.loc[min\_age]['SALARY'] return diffemployee = pd.read\_csv("Employees.csv")employee['BIRTHDAY']=pd.to\_datetime(employee['BIRTHDAY'])#Group by DEPT and use a user-defined function to get the salary difference  
salary\_diff = employee.groupby('DEPT').apply(salary\_diff)print(salary\_diff)

Explanation: The script uses apply()and a user-defined function to get the target. apply() passes the grouping result to the user-defined function as a parameter. Parameter g in the user-defined function salary\_diff()is essentially a data frame of Pandas DataFrame format, which is the grouping result here. The script gets the index of the eldest employee record and that of the youngest employee record over the parameter and then calculate the difference on salary field.

**Summary:**

Mastering Pandas groupby methods are particularly helpful in dealing with data analysis tasks.

Let’s take a further look at the use of Pandas groupby though real-world problems pulled from Stack Overflow.

# VII Position-based grouping

You group records by their positions, that is, using positions as the key, instead of by a certain field. Such a scenario includes putting every three rows to same group, and placing rows at odd positions to a group and those at even positions to the other group. Below is an example:

source: https://stackoverflow.com/questions/59110612/pandas-groupby-mode-every-n-rows

Below is part of the source data:

time a b  
0 0.5 -2.0  
1 0.5 -2.0  
2 0.1 -1.0  
3 0.1 -1.0  
4 0.1 -1.0  
5 0.5 -1.0  
6 0.5 -1.0  
7 0.5 -3.0  
8 0.5 -1.0

We want to group and combine data every three rows, and keep the mode in each column in each group. The expected result is as follows:

time a b  
2 0.5 -2.0  
5 0.1 -1.0  
8 0.5 -1.0

Problem analysis: This grouping task has nothing to do with column values but involve positions. We perform integer multiplications by position to get a calculated column and use it as the grouping condition.

Python script:

import pandas as pdimport numpy as npdata = pd.read\_csv("group3.txt",sep='\\t')#Group records by the calculated column, calculate modes through the cooperation of agg function and lambda, and get the last mode of each column to be used as the final value in each groupres = data.groupby(np.arange(len(data))//3).agg(lambda x: x.mode().iloc[-1])print(res)

Explanation: The expression np.arange(len(data)) // 3generates a calculated column, whose values are [0 0 0 1 1 1 2 2 2]. The script uses it as the key to group data every three rows. The expression agg(lambda x: x.mode())gets the mode from each column in every group. In the first group the modes in time column is [0,1,2], and the modes in a and b columns are [0.5]and [-2.0]respectively. The script then uses iloc[-1] to get their last modes to use as the final column values.

# VIII Grouping by changed value

You group ordered data according to whether a value in a certain field is changed. That is, a new group will be created each time a new value appears. Here’s an example:

Source: https://stackoverflow.com/questions/41620920/groupby-conditional-sum-of-adjacent-rows-pandas

Below is part of the original data:

duration location user  
0 10 house A  
1 5 house A  
2 5 gym A  
3 4 gym B  
4 10 shop B  
5 4 gym B  
6 6 gym B

After data is grouped by user, sum duration values whose location values are continuously the same, and perform the next sum on duration when location value changes. Below is the expected result:

duration location user  
15 house A  
5 gym A  
4 gym B  
10 shop B  
10 gym B

Problem analysis: Order is import for location column. Records with continuously same location values are put into same group, and a record is put into another group once the value is changed. When user is B, location values in row 4 (whose index is 3) are [gym,shop,gym,gym]. Here we shouldn’t just put threesame gyms into one group but should put the first gym in a separate group, becausethe location value after the first gym is shop, which is a different value. Shop should be put another separategroup. And then the other two gyms should be in same group because they are continuously same. So the grouping result for user B should be [[gym],[shop],[gym,gym]]. That’s why we can’t use df.groupby([‘user’,‘location’]).duration.sum()to get the result. Instead we need a calculated column to be used as the grouping condition.

They Python script:

import pandas as pd#Generate data for computation  
df = pd.DataFrame({'user' : ['A', 'A', 'A', 'B', 'B', 'B','B'], 'location' : ['house','house','gym','gym','shop','gym','gym'], 'duration':[10,5,5,4,10,4,6]})#Create a calculated column  
derive = (df.location != df.location.shift()).cumsum()#Group records by user, location and the calculated column, and then sum duration values  
res = df.groupby(['user', 'location', derive], as\_index=False, sort=False)['duration'].sum()print(res)

Explanation: The calculated column derive gets its values by accumulating location values before each time they are changed. The cumulated values are [1 1 2 2 3 4 4]. Then group the original data by user, location and the calculated array, and perform sum on duration.

# IX Grouping by a condition

You create a new group whenever the value of a certain field meets the specified condition when grouping ordered data. Below is an example:

Source: https://stackoverflow.com/questions/62461647/choose-random-rows-in-pandas-datafram

Below is part of the original data:

ID code  
333\_c\_132 x  
333\_c\_132 n06  
333\_c\_132 n36  
333\_c\_132 n60  
333\_c\_132 n72  
333\_c\_132 n84  
333\_c\_132 n96  
333\_c\_132 n108  
333\_c\_132 n120  
999\_c\_133 x  
999\_c\_133 n06  
999\_c\_133 n12  
999\_c\_133 n24  
998\_c\_134 x  
998\_c\_134 n06  
998\_c\_134 n12  
998\_c\_134 n18  
998\_c\_134 n36  
997\_c\_135 x  
997\_c\_135 n06  
997\_c\_135 n12  
997\_c\_135 n24  
997\_c\_135 n36  
996\_c\_136 x  
996\_c\_136 n06  
996\_c\_136 n12  
996\_c\_136 n18  
996\_c\_136 n24  
996\_c\_136 n36  
995\_c\_137 x

We want to get a random row between every two x values in code column.

The expected result is as follows:

333\_c\_132 n06  
999\_c\_133 n12  
998\_c\_134 n18  
997\_c\_135 n36  
996\_c\_136 n18

Problem analysis: To get a row from two x values randomly, we can group the rows according to whether the code value is x or not (that is, create a new group whenever the code value is changed into x), and get a random row from the current group. So we still need a calculated column to be used as the grouping key.

The Python script:

import pandas as pddf = pd.read\_csv("data.txt")#Generate a calculated column  
derive = df.code.eq('x').cumsum()#Group records by the calculated column and get a random record from each groupthrough the cooperation of apply function and lambda  
res=df[df.code.ne('x')].groupby(derive).apply(lambda x : x.sample(1))#Reset the index  
res=res.reset\_index(level=0, drop=True)print(res)

Explanation: code.eq(x) returns True when code is x and False when code isn’t x. cumsum()accumulates the number of true values and false values to generate a calculated column [1 1 1 1 1 1 1 1 1 2 2…]. Then the script finds the records where code is x, group records by those x values, and get a random record from each group.

**Summary:**

In all the above examples, the original data set is divided into a number of subsets according to a specified condition, and has the following two features:

1）No subset is empty;

2）Each member in the original data set belongs to and only belongs to one subset.

We call this type of grouping the full division. There is also partial division.

Here are examples.

# X Alignment grouping

Alignment grouping has a base set. It compares an attribute (a field or an expression) of members of the to-be-grouped set with members of the base set and puts members matching a member of the base set into same subset. The number of subsets is the same as the number of members in the base set. The alignment grouping has three features:

1）There may be empty subsets (one or more members of the base set don’t exist in the to-be-grouped set, for instance);

2）There may be members of the to-be-grouped set that are not put into any group (they are not so important as to be included in the base set, for instance);

3）Each member in the to-be-grouped set belongs to one subset at most.

1. Empty subsets

A company wants to know the precise number of employees in each department. If a department doesn’t have male employees or female employees, it records their number as 0.

Problem analysis: If we group data directly by department and gender, which is groupby([‘DEPT’,’GENDER’]), employees in a department that doesn’t have female employees or male employees will all be put into one group and the information of absent gender will be missing. It’s easy to think of an alternative. That solution groups records by department, generates a [male, female] base set to left join with each group, groups each joining result by gender and then count the numbers of male and female employees. This will make sure that each subgroup includes both female employees and male employees.

Python script:

import pandas as pd#Alignment grouping function  
def align\_group(g,l,by):  
   
 #Generate the base dataframe set and use merge function to perform the alignment grouping  
 d = pd.DataFrame(l,columns=[by]) m = pd.merge(d,g,on=by,how='left')return m.groupby(by,sort=False)employee = pd.read\_csv("Employees.csv")#Define a sequence  
l = ['M','F']#Group records by DEPT, perform alignment grouping on each group, and perform count on EID in each subgroupres = employee.groupby('DEPT').apply(lambda x:align\_group(x,l,'GENDER').apply(lambda s:s.EID.count()))print(res)

Explanation:

The user-defined function align\_groupuses merge()function to generate the base set and perform left join over it and the to-be-grouped set, and then group each joining result set by the merged column. After records are grouped by department, the cooperation of apply() function and the lambda expression performs alignment grouping on each group through a user-defined function, and then count on EID column. (Note: You shouldn’t perform count on GENDER because all GENDER members are retained during the merge operation. When there is an empty subset, the result of count on GENDER will be 1 and the rest of columns will be recorded as null when being left-joined. That will result in a zero result for a count on EID).

2. Members of the to-be-grouped set that are not put into any group

The task is to group records by the specified departments [‘Administration’, ‘HR’, ‘Marketing’, ‘Sales’], count their employees and return result in the specified department order.

Problem analysis: We can filter away the records not included by the specified set of departments using left join.

The Python script:

import pandas as pd#Alignment grouping function  
def align\_group(g,l,by): d = pd.DataFrame(l,columns=[by]) m = pd.merge(d,g,on=by,how='left') return m.groupby(by,sort=False)employee = pd.read\_csv("Employees.csv")#The specified subset of departments  
sub\_dept = ['Administration', 'HR', 'Marketing', 'Sales']#Use the alignment function to group records and perform count on EID  
res = align\_group(employee,sub\_dept,'DEPT').apply(lambda x:x.EID.count())print(res)

Explanation: Pandas doesn’t directly support the alignment grouping functionality, so it’s roundabout to implement it. Besides, the use of merge function results in low performance.

# XI Enumeration grouping

An enumeration grouping specifies a set of conditions, computes the conditions by passing each member of the to-be-grouped set as the parameter to them, and puts the record(s) that make a condition true into same subset. The subsets in the result set and the specified condition has a one-to-one relationship. One feature of the enumeration grouping is that a member in the to-be-grouped set can be put into more than one subset.

Here’s an example

The task is to group employees by durations of employment, which are [employment duration<5 years, 5 years<= employment duration<10 years, employment duration>=10 years, employment duration>=15 years], and count female and male employees in each group (List all eligible employee records for each enumerated condition even if they also meet other conditions).

Problem analysis: The enumerated conditions employment duration>=10 years and employment duration>=15 years have overlapping periods. Employees who have stayed in the company for at least 15 years also meet the other condition. A calculated column doesn’t support putting one record in multiple groups. We need to loop through all conditions, search for eligible records for each of them, and then perform the count.

import pandas as pdimport datetime#The function for converting strings into expressions  
def eval\_g(dd:dict,ss:str): return eval(ss,dd) emp\_file = 'E:\\txt\\employee.txt'emp\_info = pd.read\_csv(emp\_file,sep='\\t')employed\_list = ['Within five years','Five to ten years','More than ten years','Over fifteen years']#Grouping conditions  
employed\_str\_list = ["(s<5)","(s>=5) & (s<10)","(s>=10)","(s>=15)"]today = datetime.datetime.today().yeararr = pd.to\_datetime(emp\_info['HIREDATE'])#Calculate employment durations  
employed = today-arr.dt.yearemp\_info['EMPLOYED']=employeddd = {'s':emp\_info['EMPLOYED']}group\_cond = []#Loop through grouping conditionsfor n in range(len(employed\_str\_list)): #Group records by conditions  
 emp\_g = emp\_info.groupby(eval\_g(dd,employed\_str\_list[n])) #Grouping indexes  
 emp\_g\_index = [index for index in emp\_g.size().index] #If there are not eligible records Then the number of female or male employees are 0 if True not in emp\_g\_index: female\_emp=0 male\_emp=0  
   
 #If there are records meeting the current condition Then create a group for them And count the female and male employees else: group = emp\_g.get\_group(True) sum\_emp = len(group) female\_emp = len(group[group['GENDER']=='F']) male\_emp = sum\_emp-female\_emp group\_cond.append([employed\_list[n],male\_emp,female\_emp])#Summarize the count results for all conditionsgroup\_df = pd.DataFrame(group\_cond,columns=['EMPLOYED','MALE','FEMALE'])print(group\_df)

Explanation: EMPLOYED is a column of employment durations newly calculated from HIREDATE column. The user-defined function eval\_g()converts enumerated conditions into expressions. The enumerated conditions<5, for instance, is equivalent to the eval\_g(dd,ss) expression emp\_info[‘EMPLOYED’]<5. The new calculated column value will then be used to group the records. The script loops through the conditions to divide records into two groups according to the calculated column. get\_group(True) gets eligible groups. Finally the script uses concat() function to concatenate all eligible groups.

**Summary**

Python can handle most of the grouping tasks elegantly. It needs to generate a calculated column that meets the grouping condition when dealing with order-based grouping tasks, such as grouping by changed value/condition. It is a little complicated. It becomes awkward when confronting the alignment grouping an enumeration grouping tasks because it needs to take an extremely roundabout way, such the use of merge operation and multiple grouping. That’s time and effort consuming. Pandas still has its weaknesses in handling grouping tasks

esProc SPL handles the grouping tasks tactfully. esProc is specialized data computing engine. SPL, the language it is based, provides a wealth of grouping functions to handle grouping computations conveniently with a more consistent code style.

Two esProc grouping functions groups()and group() are used to achieve aggregation by groups and subset handling. They are able to handle the above six simple grouping problems in a concise way:

Table

Description automatically generated

Python is also convenient in handling them but has a different coding style by involving many other functions, including agg, transform, apply, lambda expression and user-defined functions. SPL takes consistent coding styles in the form of groups(x;y) and group(x).(y).

Python scripts are a little complicated in handling the following three problems by involving calculated columns. The ordered set based SPL is able to maintain an elegant coding style by offering options for handling order-based grouping tasks

Table

Description automatically generated

You can choose to use groups or group function to handle a grouping and aggregate task according to whether you need a post-grouping aggregation or you want to further manipulate data in each subset.

Python is really awkward in managing the last two types groups tasks, the alignment grouping and the enumeration grouping, through the use of merge function and multiple grouping operation. SPL has specialized alignment grouping function, align(), and enumeration grouping function, enum(), to maintain its elegant coding style.

Graphical user interface, application, table

Description automatically generated

Python’s fatal weakness is the handling of big data grouping (data can’t fit into the memory). The language requires external storage read/write and hash grouping. It’s almost impossible for a non-professional programmer to get it done in Python. Read [How Python Handles Big Files](https://medium.com/analytics-vidhya/how-python-handles-big-files-fc60ff90b819) to learn more.

That article points out Python problems in computing big data (including big data grouping), and introduces esProc SPL’s cursor mechanism. This mechanism supplies group function and groupx() function to handle big data calculations in an elegant way.

# Joining Datasets with Python’s Pandas

## How to concatenate, append, and merge Datasets with Pandas

[Thiago Carvalho](https://thiago-bernardes-carvalho.medium.com/?source=post_page-----ed832f01450c--------------------------------)

[Thiago Carvalho](https://thiago-bernardes-carvalho.medium.com/?source=post_page-----ed832f01450c--------------------------------)

<https://towardsdatascience.com/joining-datasets-with-pythons-pandas-ed832f01450c>

[Oct 26, 2020·9 min read](https://towardsdatascience.com/joining-datasets-with-pythons-pandas-ed832f01450c?source=post_page-----ed832f01450c--------------------------------)



Picture from PixBay — [Nick115](https://pixabay.com/photos/panda-family-pandas-cute-bamboo-3811734/)

It’s definitely not uncommon to work with more than one dataset when performing your analysis. Therefore, there’s an abundant amount of methods to bring this data together.

SQL call those operations ‘Joins’ or ‘Unions’; in other languages and tools, you may find functions like Merge or LookUp to do the job.

In this article, I’ll go through some of the functions we can use to join datasets with [Pandas](https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html).

I’ll run my code in a [Jupyter notebook](https://jupyterlab.readthedocs.io/en/stable/getting_started/installation.html), and the only thing we need for the examples is [Pandas](https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html).

import pandas as pd

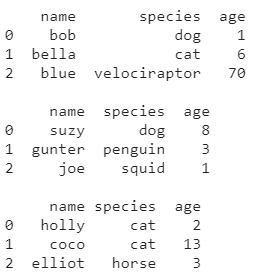
We’ll start by defining some dummy data for the examples, I’ll use lists for simplification, but you’re definitely encouraged to [load a dataset](https://medium.com/python-in-plain-english/reading-data-with-pythons-pandas-2715ff925b1d).

# create 5 lists with the same size  
names = ['bob', 'bella', 'blue', 'suzy', 'gunter', 'joe', 'holly', 'coco', 'elliot']species = ['dog', 'cat', 'velociraptor', 'dog', 'penguin', 'squid', 'cat', 'cat', 'horse']age = [1, 6, 70, 8, 3, 1, 2, 13, 3]  
weight = [10, 5, 15, 7, 4, 1, 3, 2, 380]color = ['brown', 'black', 'blue', 'black', 'black', 'gray', 'white', 'orange', 'white']

After defining the lists, we can create our data frames.

# create 3 data frames with the values from the listsdf1 = pd.DataFrame( {'name': names[:3],  
 'species': species[:3],  
 'age': age[:3]})df2 = pd.DataFrame( {'name': names[3:6],  
 'species': species[3:6],  
 'age': age[3:6]})df3 = pd.DataFrame( {'name': names[6:],  
 'species': species[6:],  
 'age': age[6:]})print(df1, '\n')  
print(df2, '\n')  
print(df3)





The three data frames

Ok, we have three datasets with the same columns and size. Let’s say we want to group those in a single data frame.

# .concat

For that, we can use .concat, which is a function that accepts a list of data frames and concatenates them into one.

# .concat to join the dataframes, like a 'union all'  
df\_list = [df1, df2, df3]  
df = pd.concat(df\_list)df

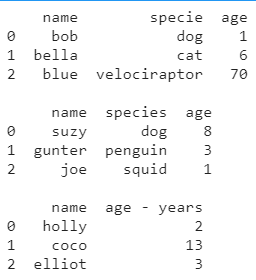


Concatenated data frame

Cool, Pandas matched the columns and returned an almost perfect data frame without much effort.

But what if the column names don’t match? or what if we’re missing a column?

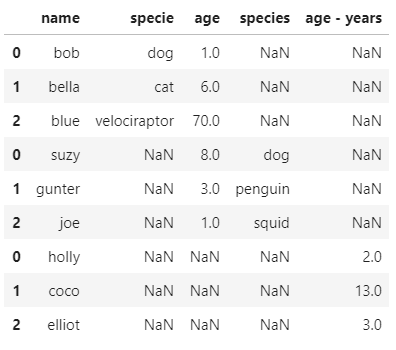
# test with mismatching and missing columnsdf1 = pd.DataFrame( {'name': names[:3],  
 'specie': species[:3],  
 'age': age[:3]})df2 = pd.DataFrame( {'name': names[3:6],  
 'species': species[3:6],  
 'age': age[3:6]})df3 = pd.DataFrame( {'name': names[6:],  
 'age - years': age[6:]})print(df1, '\n')  
print(df2, '\n')  
print(df3)



Mismatching data frames

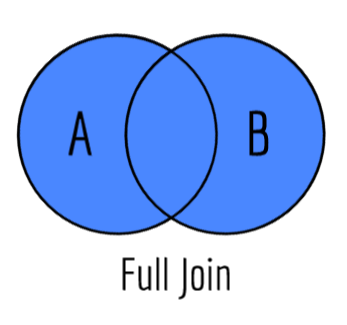
For the ‘species’ column, I changed its name in the first df and removed it from the last; I’ve also renamed ‘age’ to ‘age — years’.

# concat with mismatching and missing columns  
df\_list = [df1, df2, df3]  
df = pd.concat(df\_list)df



Concatenated mismatched data frames

Ok, so Pandas .concat requires the names of the columns to be exact matches. If a column is unique to a dataset, just like a full join, it’ll fill the gaps with null values.



Made by me with Sketch.io — Creative Commons

There’s one more thing we need to pay attention to, the indexes.

Since we didn’t define unique indexes when creating our data frame, Pandas set us with some default values starting from zero, but it doesn't reset them when we use .concat.



Picture from PixBay — [Skeeze](https://pixabay.com/photos/panda-cub-wildlife-zoo-cute-china-649938/)

A unique index is always a good idea. In this case, we can use .reset\_index to create a new column with proper values or use .set\_index to define one of the columns as the index.

But let’s try an even simpler solution that fits our case.

df1 = pd.DataFrame( {'name': names[:3],  
 'species': species[:3],  
 'age': age[:3]})df2 = pd.DataFrame( {'name': names[3:6],  
 'species': species[3:6],  
 'age': age[3:6]})df3 = pd.DataFrame( {'name': names[6:],  
 'species': species[6:],  
 'age': age[6:]})# since we didn't define the indexes when creating the dataframes we can ignore them when concatenatingdf\_list = [df1, df2, df3]  
df = pd.concat(df\_list, ignore\_index=True)df



Concatenated data frames re-indexed

With a single parameter, we ignored the indexes and got new ones in the concatenated result.

Another handy parameter is ‘keys’, which allows us to identify the data source with a new index level.

df1 = pd.DataFrame( {'name': names[:3],  
 'species': species[:3],  
 'age': age[:3]},  
 index = [1,2,3])df2 = pd.DataFrame( {'name': names[3:6],  
 'species': species[3:6],  
 'age': age[3:6]},  
 index = [10,11,12])df3 = pd.DataFrame( {'name': names[6:],  
 'species': species[6:],  
 'age': age[6:]},  
 index = [100,200,300])# we can pass 'keys' which creates another index level to identify the concatenated data frames  
df\_list = [df1, df2, df3, df1]  
df = pd.concat(df\_list, keys=['df1', 'df2', 'df3', 'df4'])df



Concatenated data frames with an added index level

By default, .concat uses the columns as keys and append the values as rows. But what if we want to concatenate columns to our data frame?



Picture from PixBay — [Free-Photos](https://pixabay.com/photos/columns-hallway-architecture-greek-801715/)

First, let’s define some new columns to concatenate.

df4 = pd.DataFrame( {'weight': weight,  
 'color': color  
 })  
df4



Data frame with the extra columns

Similarly, we could add the lists directly to our data frame by assigning them to a column, like so:

df['color'] = color

But a data frame could have lots of fields, and passing them one by one wouldn’t be the best option.

The concatenate function accepts a parameter for ‘axis’, which allows us to do just that — concatenate columns.

df = pd.concat([df, df4], axis=1)  
df

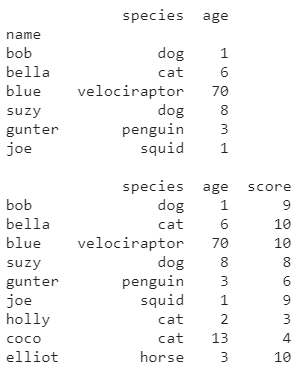


Concatenated data frame

We can also change the behaviour of the join.

Let’s try making the ‘name’ column our index and creating another dataset with an extra column to experiment with.

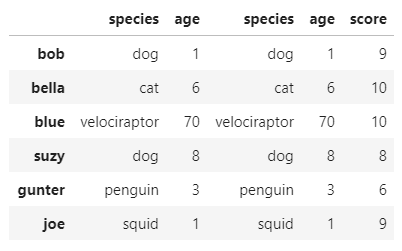
df\_list = [df1, df2]  
df = pd.concat(df\_list)  
df.set\_index('name', inplace=True)df5 = pd.DataFrame( {'species': species,  
 'age': age,  
 'score': [9,10,10,8,6,9,3,4,10]},  
 index = names)print(df, '\n')  
print(df5)



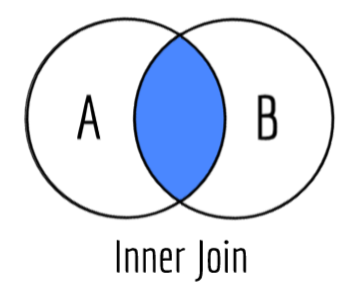
More data frames

Note that our first data frame has fewer values than the second. When we perform an inner join, it should only bring the rows where the indexes match.

# by default concat behaves like an outer join, or a union all  
# we can change that with the 'join' parameter  
df\_list = [df, df5]  
df = pd.concat(df\_list, axis=1, join='inner')  
df



Data frame concatenated with an inner join



Made by me with Sketch.io — Creative Commons

# .append

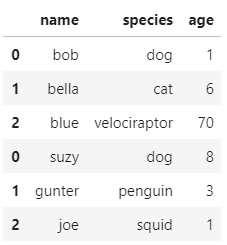
Now let’s have a look at another function called .append.

This function has similar behaviour to .concat. The previous take multiple data frames and concatenates them into a new one. The append method will use an existing data frame to add the data.

Both will return a data frame, but the way you call them is different. You’ll use Pandas.concat() and DataFrame.append().

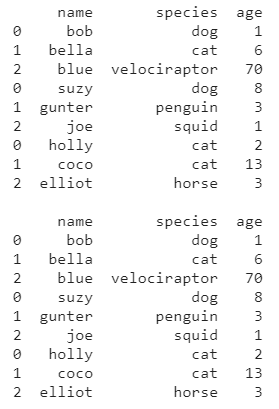
Let’s check some examples.

df1.append(df2)



Data frame 2 appended to data frame 1

df = df1.append(df2)  
df = df.append(df3)  
print(df, '\n')df = df1.append([df2, df3])  
print(df)



Data frame appended one-by-one, and data frame appended with a list

# append a row  
df.append(pd.Series(['oliver', 'monkey', 13], index=['name', 'species', 'age']), ignore\_index=True)



Data frame with an appended row

# .merge

Awesome, with .concat and .append, we can perform most of the joins we might need. Now let’s check a more robust solution named .merge.

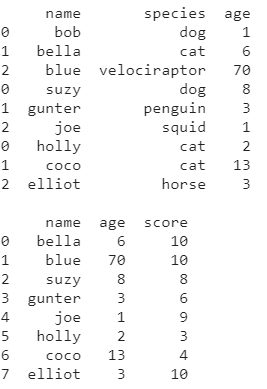


Picture from Dreamstime

Starting with something simple, let’s see how .merge performs a join.

I’ll define another data frame similar to the one we’re already using, but with one more column and one less record.

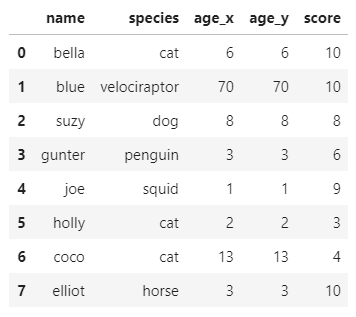
print(df, '\n')df6 = pd.DataFrame( {'name': names[1:],  
 'age': age[1:],  
 'score': [10,10,8,6,9,3,4,10]})  
print(df6)



More data frames for experimenting

Merge allows us to select which column will be the key; in this case, let’s use ‘name’.

merged\_df = pd.merge(df, df6, on='name')  
merged\_df



Data frame merged by the ‘name’ column.

Unlike what we saw earlier, a merge is by default an inner join — That means unless told otherwise, it’ll only return the matching rows from both datasets.

We can also notice that the columns present in both datasets are separated, even though they contain the same values.

Lastly, when we perform an inner join like the above, both data frames must have the key column with the same name.



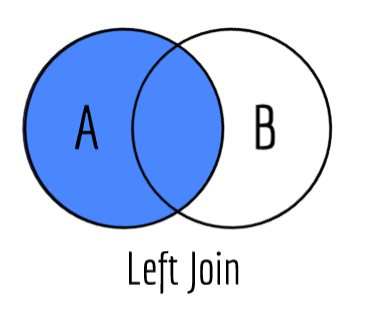
Picture from PixBay — [cocoparisienne](https://pixabay.com/photos/key-close-up-open-door-key-3497145/)

You can select multiple columns as key, like a composite key, and you can also select which kind of join you’ll use.

merged\_df = pd.merge(df, df6, how='left', on=['name', 'age'])  
merged\_df



Data frame merged left on name and age fields.



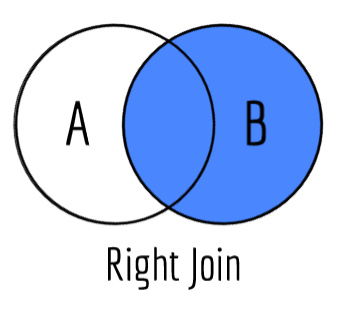
Made by me with Sketch.io — Creative Commons

merged\_df = pd.merge(df6, df, how='right', on=['name', 'age'])  
merged\_df



Data frame merged right on name and age fields.





Made by me with Sketch.io — Creative Commons

Great! Besides all that, the merge function also helps us to validate and understand the data were merging.

# 'one\_to\_one' or ‘1:1’  
merged\_df = pd.merge(df6, df.append(df1), how='right', on=['name', 'age'], validate='one\_to\_one')merged\_df





Error message

For example, the validate parameter will raise an error if the data frames don’t respect the criteria you chose.

Other useful options are:

**‘one\_to\_many’** or **‘1:m’** — Checks if the left keys are unique;

**‘many\_to\_one’** or ‘**m:1’**— Checks if the right keys are unique;

The ‘indicator’ parameter adds a column to the data frame explaining the keys' relationship.

merged\_df = pd.merge(df6, df.append(df1), how='outer', on=['name', 'age'], indicator=True)merged\_df





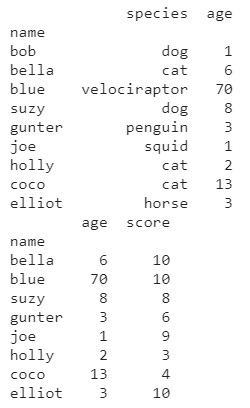
A merged data frame with indicator column

\*If you pass a string to the indicator parameter, it’ll be used as the created column's name.

We don’t need to specify which column contains the key since, by default, Pandas will assume the index is the key.

Let’s try setting the name as our index and merging those data frames again.

df.set\_index('name', inplace=True)  
df6.set\_index('name', inplace=True)print(df)  
print(df6)



Data frames indexed by name

pd.merge(df6, df)



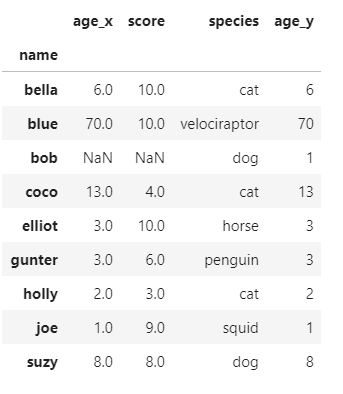


Data frames merged by index

If we don’t want Pandas to reset the index, we have to use the right\_index and left\_index parameters.

pd.merge(df6, df, how='outer', left\_index=True, right\_index=True)





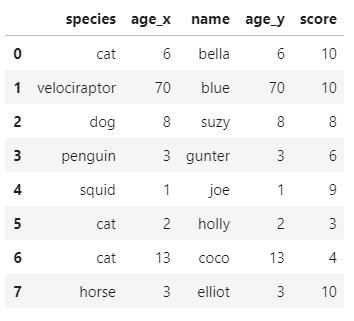
Data frames merged by index.

If we want to merge with an index on one side and with a key on the other, we can specify the right\_on and left\_on parameters.

They accept the column's name holding the key, just like we saw before, but will be applied only to that specific side, using the index column on the other side.

pd.merge(df, df6, how='right', right\_on='name', left\_index=True)





Data frame merged by index and key column.

Great! We got a look at .concat and .append, two convenient functions for joining two data frames. Then we explored .merge, an even better option with lots of flexibility.

A panda bear in a zoo exhibit

Description automatically generated with low confidence

Picture from PixBay — [PredragKezic](https://pixabay.com/photos/panda-cub-wildlife-zoo-cute-china-1203101/)

Pandas have even more methods to help you work with multiple datasets; it’s not unusual to spend time building the logic to solve your problem and then finding out an already implemented solution for that in a library.

So I encourage you to get a look at some of those other functions such as .compare, .combine\_first, and .merge\_asof.

Thanks for reading my article. I hope you enjoyed it!

**References:**[Python for Data Analysis — Wes McKinney](https://www.oreilly.com/library/view/python-for-data/9781449323592/);  
[Pandas — Concat](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html);  
[Pandas — Merge](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.merge.html);  
[Pandas — Append](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.append.html);  
[Pandas — Merge, join, concatenate and compare](https://pandas.pydata.org/docs/user_guide/merging.html);

[[](https://medium.com/analytics-vidhya?source=post_page-----6e11a377751f--------------------------------)](https://medium.com/analytics-vidhya?source=post_page-----6e11a377751f--------------------------------)

* [About Us](https://medium.com/analytics-vidhya/about?source=post_page-----6e11a377751f--------------------------------)
* [Deep Learning](https://medium.com/analytics-vidhya/deep-learning/home?source=post_page-----6e11a377751f--------------------------------)
* [Machine Learning](https://medium.com/analytics-vidhya/machine-learning/home?source=post_page-----6e11a377751f--------------------------------)
* [Hackathon](https://medium.com/analytics-vidhya/tagged/hackathons?source=post_page-----6e11a377751f--------------------------------)
* [Contribute](https://medium.com/analytics-vidhya/contribute/home?source=post_page-----6e11a377751f--------------------------------)
* [Free Courses](https://courses.analyticsvidhya.com/pages/all-free-courses?source=post_page-----6e11a377751f--------------------------------)



<https://medium.com/analytics-vidhya/sql-for-data-scientists-in-6-minutes-or-less-6e11a377751f>

Graphical user interface, text

Description automatically generated

Background image: [Unsplash](https://unsplash.com/photos/Wpnoqo2plFA)

**SQL for Data Scientists, in Under 6 Minutes**

**An Essential Skill for Any Data Science Résumé**

[](https://andre-ye.medium.com/?source=post_page-----6e11a377751f--------------------------------)

[Andre Ye](https://andre-ye.medium.com/?source=post_page-----6e11a377751f--------------------------------)

[Mar 23, 2020](https://medium.com/analytics-vidhya/sql-for-data-scientists-in-6-minutes-or-less-6e11a377751f?source=post_page-----6e11a377751f--------------------------------) · 6 min read

Data scientists often work with DataFrames, be it in R or Python. However, large amounts of data — the vast amounts in today’s data science, ‘Big Data’ — simply can’t be completely loaded into a DataFrame or even into a .csv file. These are stored in massive databases, of which a very common one is a SQL database.

SQL is remarkably simple and easy to learn, and for a data scientist who is familiar with DataFrame operations, is just a matter of learning syntax. Because of its popularity, SQL has integration with numerous applications, from pandas for data analysis to PHP for front-end connections.

Learning SQL is such an easy and useful skill to add to a résumé that it would be almost wasteful not to spend a few minutes learning it.

This article will cover enough of SQL to perform ordinary data science operations. Let’s get into it!

**Contents**

* Selecting tables and columns, selecting distinct values, selecting top values, ordering columns
* Conditional selecting, selecting rows, selecting top values
* Inserting, updating, and deleting values
* Basic Statistical Values: minimum, maximum, count, average, sum
* Joining tables

**Selecting Tables and Columns**

SQL databases are comprised of several tables. A table is equivalent to a DataFrame — it has columns and rows. When calling data within a SQL database, the table needs to be specified.

The below code selects everything (represented by a \*) from the table table\_name.

SELECT \* FROM table\_name;

To select certain columns, the below syntax is used:

SELECT column1, column2 FROM table\_name;

Inside a table, columns may contain duplicate values. In the case that only the unique values are desired, the SELECT DISTINCT command can be used.

SELECT DISTINCT column1, column2 FROM table\_name;

If the values need to be ordered, they can be done so with the ORDER BY keyword. The syntax is

SELECT column1, column2 FROM table\_name ORDER BY column1 ASC;

…where ASC can be substituted with DESC, if the values are to be largest to smallest instead of smallest to largest (default).

For example, SELECT \* FROM table\_name ORDER BY country; will return the entire table, ordered alphabetically by country. ORDER BY also works with numbers. In the case that multiple ORDER BY columns are specified (for example, SELECT \* FROM table\_name ORDER BY country, income; will return the entire table, ordered alphabetically by country, and ordered by income if countries are the same.

Note that because SQL is used in such a diverse array of applications, some consoles may require a semicolon at the end of every row, and some may not. In this tutorial, we’ll be using semicolons just as a precaution, but note that if it throws a syntax error, try removing it.

**Conditional Selection & Row Selection**

The WHERE clause in SQL allows for conditional selection and selection of rows. The syntax is

SELECT column1, column2 FROM table\_name WHERE condition;

For example,

SELECT column1 FROM table\_name where column1 > 5;

This query selects the values column1 from table\_name where the value for column1 is larger than 5.

If you wanted to select the entire row, you could use

SELECT \* FROM table\_name WHERE column1 > 5;

There are no indices in SQL; instead, usually there is a column, perhaps named ID or index. To select, say, the third row, one could query

SELECT \* FROM table\_name WHERE index = 3;

The operators in the WHERE clause include:

* = for equal
* > for greater than
* < for less than
* >= for greater than or equal
* <= for less than or equal
* <> for not equal (this may be written as != in some versions of SQL)
* BETWEEN x AND y for all values between x and y
* LIKE string for a certain pattern (for example, the string s% returns all values of the column that begin with an s)
* IN for inclusion in a list, to specify several possible values

The WHERE clause can also be integrated with SQL AND, OR, and NOT operations.

SELECT \* FROM table\_name WHERE column1 = 5 AND column2 <= 3;

…returns rows where column1 is equal to 5 and where column2 is smaller than or equal to 3.

As another example,

SELECT column1, column2 FROM table\_name WHERE (column1 < 3 OR column2 < 10) AND (NOT column3 BETWEEN 1 AND 10)

This statement selects column1 and column2 of rows where either column1 is smaller than 3 or where column2 is smaller than 10, and where column3 is not between 1 and 10.

To select the top few values, the TOP keyword is used:

SELECT TOP *number or percent* *column\_names*  
FROM *table\_name*WHERE *condition*;

This is the equivalent of the .head(x) function in Python.

**Inserting, Updating, and Deleting Values**

Inserting values follows by the syntax:

INSERT INTO table\_name (column1, column2, column3) VALUES (value1, value2, value3);

If inserting values into every column, there is no need to specify which columns the values are going to — it is assumed that the values or ordered by left-to-right column arrangement.

Updating values follows the syntax:

UPDATE table\_name SET column1 = value1, column2 = value2 WHERE condition;

SQL’s UPDATE is helpful because the syntax for updating several values and for updating only one value is the same. If you wanted to update column1 of the third row, the query would simply be

UPDATE table\_name SET column = value1 WHERE id = 3;

Or, if updating a value for all rows whose country is US:

UPDATE table\_name SET column = value1 WHERE country = 'US';

DELETE’s syntax is

DELETE FROM table\_name WHERE condition;

Be careful! If the WHERE clause is omitted, the entire table content will be deleted.

**Basic Statistical Values**

The MIN() and MAX() functions select the smallest and largest values, respectively.

This code selects the minimum value of column\_name where condition is satisfied.

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

MIN() can be substituted in this example with with

* MAX(column\_name): Maximum instance value
* COUNT(column\_name): Number of instances in the column
* AVG(column\_name): Average of instances in the column
* SUM(column\_name): Sum of all instances in the column

**Joins**

A JOIN clause is used to combine rows from two or more tables, based on a related column in between them.

Say there are two tables, Orders and Customers. We want to join Orders with Customers on the CustomerID.

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

Note that like in pandas, the notation table.column to access a column is used here. In this case, an inner join was used, which only returned values that both Orders (Table A) and Customers (Table B) had.

There are various types of other SQL joins, for several other methods joining. The cheat sheet below shows how the ON keyword should be used on tables A and B for several forms of joins.

Text

Description automatically generated with medium confidence

[Source](https://i.stack.imgur.com/XG4Q6.png). Image free to share and use commercially.

Say we want to select the values of ID in tables record1 and record2 that the are not in both tables. Then, (looking up the scenario in the cheat sheet), the code would be

SELECT ID  
FROM record1  
FULL OUTER JOIN JOIN record2   
ON record1.ID = record2.ID  
WHERE record1.ID IS NULL OR record2.ID IS NULL

That’s it! Pandas has option to load data from SQL, but having knowledge of how to perform basic data manipulations in data before loading a subset of the data into Pandas or R for more complex analysis. What you’ve learned so far should be enough to operate and select Big Data within SQL databases.

If you enjoyed, check out other articles in this series:

* [LaTeX for Data Scientists, in Under 6 Minutes](https://towardsdatascience.com/latex-for-data-scientists-in-under-6-minutes-3815d973c05c)

A picture containing graphical user interface

Description automatically generated

Go ahead and click ‘Add’.

**Sign up for Analytics Vidhya News Bytes**

**By Analytics Vidhya**

Latest news from Analytics Vidhya on our Hackathons and some of our best articles! [Take a look.](https://medium.com/analytics-vidhya/newsletters/analytics-vidhya-news-bytes?source=newsletter_v3_promo--------------------------newsletter_v3_promo-----------)

Emails will be sent to learnermanash@gmail.com.

* [Sql](https://medium.com/analytics-vidhya/tagged/sql)
* [Data Science](https://medium.com/analytics-vidhya/tagged/data-science)
* [Tutorial](https://medium.com/analytics-vidhya/tagged/tutorial)
* [Data](https://medium.com/analytics-vidhya/tagged/data)
* [Data Analysis](https://medium.com/analytics-vidhya/tagged/data-analysis)

[](https://andre-ye.medium.com/?source=follow_footer-------------------------------------)

Written by

[**Andre Ye**](https://andre-ye.medium.com/?source=follow_footer-------------------------------------)

**SQL queries in Python**

[Shubhi Asthana](https://medium.com/@shubhi.asthana?source=post_page-----51ef85b92c1e--------------------------------)

[Shubhi Asthana](https://medium.com/@shubhi.asthana?source=post_page-----51ef85b92c1e--------------------------------)

[Dec 12, 2020·4 min read](https://towardsdatascience.com/sql-queries-in-python-51ef85b92c1e?source=post_page-----51ef85b92c1e--------------------------------)

<https://towardsdatascience.com/sql-queries-in-python-51ef85b92c1e>

A picture containing text, indoor, several

Description automatically generated

Photo courtesy: pexels.com

SQL is typically a go-to tool when you need to get a quick look at some data and draw preliminary conclusions. However the data can be of various shapes and forms, stored across different tables, and not really ‘relational’. Additionally, if one is doing data science in python, they would want to be able to read data and write back their predictions and forecasts to DB quickly.

Until recently I had never come across an opportunity to use SQL in Python. When I started looking around for options, I found many — [Sqlite](https://www.sqlitetutorial.net/sqlite-python/), [MySQL](https://www.w3schools.com/python/python_mysql_getstarted.asp), [SQLAlchemy](https://pypi.org/project/SQLAlchemy/).

In this article, I will give an overview how we can leverage SQLAlchemy and Pandas to perform database queries.

**Step 1: Importing SQLAlchemy and Pandas**

Lets start with importing the sqlalchemy library. We will be using the create\_engine feature from the library.

!pip install sqlalchemy   
from sqlalchemy import create\_engine

We also import pandas, a python library built for data analysis and manipulation

import pandas

**Step 2: Creating a SQL engine**

We create a SQL engine using the command which creates a new class ‘.engine’.

engine = create\_engine(\*args)

The argument is a string which indicates database dialect and connection arguments in the form of a url. It is typically what you would write in the SQL engine to connect to a DB.

**dialect[+driver]://**+ dsn\_uid + **':'** + dsn\_pwd + **'@'**+dsn\_hostname+**':'**+dsn\_port+**'/'** + dsn\_database

Here dialect is the database name such as mysql, oracle, postgresql etc. Each database has a corresponding DBAPI wrapper. All dialects require that an appropriate DBAPI driver is installed.

create\_engine() builds a secure connection with DB so you can read and write to it.

**Step 3 — Running queries using SQL statements**  
Coming to the interesting part in the blog, lets go through the steps one can take to submit data queries using sqlalchemy.   
To submit data queries, the following steps are followed:  
a.Wrap your SQL statements in a container

b. Send it to the database,

c. Receive the response back,

d. Put the response in a pandas dataframe.

Like any SQL query, the two primary clauses that must be present in every query here are SELECT, and FROM.

· SELECT allows you to select a subset of columns (or all of them) from a table,

· FROM specifies which table the column(s) are being pulled from.

For example, to return all columns from a ‘table1’, you can do the following:

sql = "SELECT \* FROM table1 "df = pd.read\_sql\_query(sql, engine)df.head()

If you are dealing with multiple tables, you may need to specify which column from which table because the columns of interest may come from different tables in the database.

SELECT table1.column1, table1.column2  
FROM table1

If you want to get certain rows from a column, you can use this query

SELECT DISTINCT column1  
FROM table1

**Step 4 — Writing to DB**

Writing to DB in python using SQLAlchemy is similar to what you would do in a SQL environment.

Once you create\_engine, and receive data, you can use to\_sql to write to DB. Here the data should be placed inside a dataframe.

from sqlalchemy import create\_engineengine = create\_engine(\*args)

Now create a table with some rows

df = pd.DataFrame({‘name’ : [‘T1’, T2', T3']})print(df)>>> name0 T11 T22 T3

Using to\_sql we can write to the DB

df.to\_sql(tableT, con=engine, if\_exists=’append’)

Check if data was written in the table correctly

engine.execute(“SELECT \* FROM tableT “).fetchall()[(0, ‘T1’), (1, ‘T2’), (2, ‘T3’)]

One can append more rows to the table, or replace the rows with a new dataframe. As follows

df2 = pd.DataFrame({'name' : ['T6', 'T7']})df2.to\_sql(' tableT ', con=engine, if\_exists='replace')

**Step 5— Creating a Table in DB**

To create a table in DB from python, we make use of Metadata. Metadata is a collection of Table objects and their associated schema constructs.

from sqlalchemy import MetaDatameta = MetaData()

Next, we define our table using **the Table construct**, which resembles regular SQL CREATE statement.

SQLAlchemy matches Python data to the best possible generic column data types defined in it. Some of the generic data types are −

* Boolean
* Date
* DateTime
* Float
* Integer
* Numeric
* String
* Text
* Time

Lets take an example and create a students table

from sqlalchemy import create\_engine, MetaData, Table, Column, Integer, Stringengine = create\_engine(\*args)meta = MetaData()students = Table('students', meta, Column('id', Integer, primary\_key = True), Column('name', String), Column('lastname', String))meta.create\_all(engine)

Here create\_all() function uses the engine object to create all the defined table objects and stores the information in metadata.

Questions or feedback? I’d love to hear from you — please feel free to leave out a comment, or connect with me on T[witter](https://twitter.com/shubhi_asthana)/[Linkedin](https://www.linkedin.com/in/shubhi-asthana/).

This is a primer to get started with SQL queries in Python. For more advanced sql queries, please refer to:

<https://www.tutorialspoint.com/sqlalchemy/sqlalchemy_core_using_set_operations.htm>

<https://towardsdatascience.com/sql-in-python-for-beginners-b9a4f9293ecf>

<https://www.kdnuggets.com/2019/05/7-steps-mastering-sql-data-science-2019-edition.html>

**Step 1: Relational Database Basics**

Let's start by revisiting the question posed at the start of this article: **What is Structured Query Language (SQL) and why is it so integral to data management?**

Very simply, SQL is the language for managing and querying data in relational database management systems (RDBMS).

So intertwined are the terms SQL and RDBMS that they are often conflated, sometimes by the uninitiated, but often simply out of convenience, and the term SQL is used adversarially to distinguish relational systems from non-relational database systems, which are often categorized by the umbrella term "NoSQL." Yet, SQL skills aren't wasted on non-RDBMS systems; the top data processing frameworks all have some implementation of SQL that sits atop their architecture.

Let's first watch and listen to Prof. [Andy Pavlo](http://www.cs.cmu.edu/~pavlo/) of Carnegie Mellon University discuss databases, database systems, the relational model, relational algebra, and related concepts. This is a great starting point.

**Step 2: SQL Overview**

Now that we're all familiar with the relational paradigm, we shift focus to SQL.

Let's approach this initial exposure to SQL as follows. First watch the video below, titled "SQL - A Brief Review," by Charles Germany, for some very quick insight into where SQL came from, along with some syntax by way of a few short examples. As with the resources and examples below the video, don't worry about fully understanding the syntax, just get a feel for what SQL is doing, what it can be used for, and what it looks like.

Next, have a look at these 2 short tutorials: [**SQL Basic Queries**](http://lgatto.github.io/sql-ecology/01-sql-basic-queries.html) (via Data Carpentry) does what its title suggests, while [**List of SQL Commands**](https://www.codecademy.com/articles/sql-commands) (via Codecademy) provides a more broad overview of SQL commands and usage. This second resource in particular should make for a handy reference later on. Some of these commands are covered in greater detail in subsequent steps, so, again, don't worry about understanding everything. At the same time, these initial examples are rather intuitive, and so should be helpful for your understanding.

Finally, in preparation of the of next step, get an SQL environment up and running. You may not want to enter every SQL statement you encounter, but having an SQL interpreter up and running just makes sense. I suggest installing SQLite locally; it is a simple, but capable, SQL installation.

* [**Get SQLite**](https://www.sqlite.org/index.html)
* [**Command Line Shell for SQLite**](https://sqlite.org/cli.html)

**Step 3: Selecting, Inserting, Updating**

While SQL can be used to perform a number of varied data management tasks, querying databases with SELECT, inserting records with INSERT, and updating existing records with UPDATE are some of the most heavily-used commands, and are good places to get started in practical SQL.

Read and go through the following preliminary exercises, all courtesy of [SQL Course](http://www.sqlcourse.com).

* [**Selecting Data**](http://www.sqlcourse.com/select.html)
* [**Inserting into a Table**](http://www.sqlcourse.com/insert.html)
* [**Updating Records**](http://www.sqlcourse.com/update.html)

This tutorial, [**MySQL Tutorial 1: Overview, Tables, Queries**](https://arachnoid.com/MySQL/) from [Arachnoid](https://arachnoid.com), covers SQL basics and beginner to intermediate queries, and is a solid review of what we have seen to this point, and then some. It is presented from a MySQL point of view, but nothing in here is exclusive to that platform's SQL implementation.

**Step 4: Creating, Dropping, Deleting**

Our second set of commands include those used to CREATE and DROP tables, as well as to DELETE records. With an understanding of this growing collection of commands, suddenly much of what could be referred to as regular data management and query is attainable (with practice, of course).

Try out the following tutorials again via [SQL Course](http://www.sqlcourse.com).

* [**Creating Tables**](http://www.sqlcourse.com/create.html)
* [**Drop a Table**](http://www.sqlcourse.com/drop.html)
* [**Deleting Records**](http://www.sqlcourse.com/delete.html)

**Step 5: Views and Joins**

On to some slightly more advanced SQL topics.

First, we have a look at views, which can be thoughts of as virtual tables populated by the results of queries, useful for a number of different scenarios including application development, data security, and eased data sharing. Watch this video from Socratica for insight.

Joins come in different flavors, and likely one of the more complex topics you will cover while learning SQL is getting them straight. That's really more of a testament to the ease of SQL than the actual difficulty of learning about joins. Get a better understanding of joins with this Socratica video.

See this [**visual representation of SQL joins**](http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins) from [Code Project](http://www.codeproject.com).



This tutorial, [**MySQL Tutorial 2: Views and Joins**](https://arachnoid.com/MySQL/views_joins.html) from [Arachnoid](https://arachnoid.com), covers vies and joins. It is also presented from a MySQL point of view, but nothing in here is exclusive to that platform's SQL implementation.

**Step 6: Advanced SQL**

Advanced SQL can mean a lot of different things, and so we will narrow our scope here.

First watch this lecture from Prof. [Andy Pavlo](http://www.cs.cmu.edu/~pavlo/) of Carnegie Mellon University, wherein he discusses aggregates, distinct aggregates, group by, having, string operations, date and time operations, nested queries, table expressions, and more.

After getting a sense of these aspects of SQL, turn your attention to the important concept of stored procedures, and check out the [**MySQL Stored Procedure**](http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx) tutorial from [MySQL Tutorial](http://www.mysqltutorial.org).

**Step 7: Query Optimization**

Once you know how to write queries, you are going to want to learn how to optimize them for both results and run time. This is especially true of complex queries on large databases.

Watch Prof. [Andy Pavlo](http://www.cs.cmu.edu/~pavlo/) of Carnegie Mellon University cover this topic in the following lecture video.

Then check out the [**SQL Tuning or SQL Optimization**](https://beginner-sql-tutorial.com/sql-query-tuning.htm) tutorial from [Beginner SQL Tutorial](https://beginner-sql-tutorial.com/sql-query-tuning.htm).

Hopefully you have learned enough SQL at this point to consider yourself a "master" of the subject as relates to data science. But don't stop here; there are plenty more free quality resources online which you can use to build on this new foundation of mastery. You can never know enough SQL, and as with so many other skills, practice in the key to reinforcement.

**Related**:

* [7 Steps to Mastering Basic Machine Learning with Python — 2019 Edition](https://www.kdnuggets.com/2019/01/7-steps-mastering-basic-machine-learning-python.html)
* [SQL Case Study: Helping a Startup CEO Manage His Data](https://www.kdnuggets.com/2018/09/sql-case-study-helping-startup-ceo-manage-data.html)
* [To SQL or not To SQL: that is the question!](https://www.kdnuggets.com/2018/05/sql-not-sql-question.html)