**Assignment 1.**

1. Perform the following operations using Python on the Facebook metrics data sets

a. Create data subsets

b. Merge Data

c. Sort Data

d. Transposing Data

e. Shape and reshape Data

**a. Create data subsets**

**How to Subset a DataFrame in Python?**

If you are importing data into Python then you must be aware of Data Frames. A DataFrame is a **two-dimensional data structure**, i.e., data is aligned in a tabular fashion in rows and columns.Subsetting a data frame is the process of **selecting a set of desired rows and columns from the data frame.**

You can select:

* all rows and limited columns
* all columns and limited rows
* limited rows and limited columns.

Subsetting a data frame is important as it allows you to access only a certain part of the data frame. This comes in handy when you want to reduce the number of parameters in your data frame.

**Importing the Data to Build the Dataframe**

Let’s start with importing the data into a data frame using [pandas](https://www.askpython.com/python-modules/pandas/python-pandas-module-tutorial).

Import pandas as pd

*df = pd.read\_csv("D:/dataset\_Facebook.csv",sep = ';')*

**Select a Subset of a Dataframe using the Indexing Operator**

Indexing Operator is just a fancy name for **square brackets.** You can select columns, rows, and a combination of rows and columns using just the square brackets. Let’s see this in action.

**1. Selecting Only Columns**

To select a column using indexing operator use the following line of code.

***print(df['Type'])***

This line of code selects the column with label as ‘Type’ and displays all row values corresponding to that.

**You can also select multiple columns using indexing operator.**

**print(df[['like','share']])**

|  |
| --- |
|  |

To subset a dataframe and store it, use the following line of code :

**df\_subset = df[['like','share']]**

This creates a separate data frame as a subset of the original one.

**2. Selecting Rows**

You can use the indexing operator to select specific rows based on certain conditions.

For example to select rows having likes greater than 100 you can use the following line of code.

df\_subset = df[df['like']>100]

You can also further subset a data frame.

**Subset a Dataframe using Python .loc()**

We can select specific ranges of our data in both the row and column directions using either label or integer-based indexing.

* loc is primarily *label* based indexing. *Integers* may be used but they are interpreted as a *label*.
* iloc is primarily *integer* based indexing

**.**[**loc** indexer](https://www.askpython.com/python-modules/pandas/python-loc-function) is an effective way to select rows and columns from the data frame. It can also be used to select rows and columns simultaneously.

An important thing to remember is that **.loc() works on the labels of rows and columns.** After this, we will look at .iloc() that is based on an index of rows and columns.

**1. Selecting Rows with loc()**

To select a single row using .loc() use the following line of code.

print(df.loc[1])

To select multiple rows use :

print(df.loc[[1,5,7]])

You can also slice the rows between a starting index and ending index.

|  |
| --- |
| df.loc[1:7] |

**2. Selecting rows and columns**

To select specific rows and specific columns out of the data frame, use the following line of code :

|  |
| --- |
| df.loc[1:7,['Type', 'likes']] |

This line of code selects rows from 1 to 7 and columns corresponding to the labels ‘Type’ and ‘likes’.

**Subset a Dataframe using Python iloc()**

[**iloc() function**](https://www.askpython.com/python/built-in-methods/python-iloc-function) is short for **integer location**. It works entirely on integer indexing for both rows and columns.

To select a subset of rows and columns using iloc() use the following line of code:

|  |
| --- |
| df.iloc[[2,3,6], [3, 5]] |

This line of code selects row number **2, 3 and 6** along with column number **3 and 5.**

Using iloc saves you from writing the complete labels of rows and columns.

You can also use iloc() to select rows or columns individually just like loc() after replacing the labels with integers.

**REMEMBER**

* When selecting subsets of data, square brackets [] are used.
* Inside these brackets, you can use a single column/row label, a list of column/row labels, a slice of labels, a conditional expression or a colon.
* Select specific rows and/or columns using loc when using the row and column names
* Select specific rows and/or columns using iloc when using the positions in the table
* You can assign new values to a selection based on loc/iloc.

**Python Syntax**

You can use the syntax below when querying data by criteria from a DataFrame. Experiment with selecting various subsets of the “rainfall” data.

* Equals: ==
* Not equals: !=
* Greater than, less than: > or <
* Greater than or equal to >=
* Less than or equal to <=

**Missing Data Values - NaN**

By now you probably wondered about the NaNs in the data, e.g. in the data column. NaN stands for **N**ot **aN**umber. NaN values are undefined values that cannot be represented mathematically. Pandas, for example, will read an empty cell in a CSV or Excel sheet as a NaN. NaNs have some desirable properties: if we were to average the data column without replacing our NaNs, Python would know to skip over those cells.

rainfall\_df['data'].mean()

0.37230130486357743

We can replace all NaN values with zeroes using the .fillna() method (after making a copy of the data so we don’t lose our work):

df1['data'] = df1['data'].fillna(0)

df1.tail()

However NaN and 0 yield different analysis results. The mean value when NaN values are replaced with 0 is different from when NaN values are simply thrown out or ignored.

We can fill NaN values with any value that we chose. The code below fills all NaN values with a mean for all rainfall values.

df1['data'] = df['data'].fillna(likes\_df['data'].mean())

df1.tail()

We could also chose to create a subset of our data, only keeping rows that do not contain NaN values. .dropna() removes all rows with NaNs.

df\_na = df.dropna()

**B. Merge Data**

**Combining Datasets: Merge and Join**

One essential feature offered by Pandas is its high-performance, in-memory join and merge

**Relational Algebra**

The behavior implemented in pd.merge() is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases. Pandas implements several fundamental building-blocks in the pd.merge() function and the related join() method of Series and Dataframes.

**Categories of Joins**

The pd.merge() function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the pd.merge() interface; the type of join performed depends on the form of the input data.

**One-to-one joins**

Perhaps the simplest type of merge expresion is the one-to-one join, which is in many ways very similar to the column-wise concatenation seen in [Combining Datasets: Concat & Append](https://jakevdp.github.io/PythonDataScienceHandbook/03.06-concat-and-append.html). As a concrete example, consider the following two DataFrames which contain information on several employees in a company:

df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],

'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})

df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],

'hire\_date': [2004, 2008, 2012, 2014]})

To combine this information into a single DataFrame, we can use the pd.merge() function:

df3 = pd.merge(df1, df2)

df3

|  | **employee** | **group** | **hire\_date** |
| --- | --- | --- | --- |
| **0** | Bob | Accounting | 2008 |
| **1** | Jake | Engineering | 2012 |
| **2** | Lisa | Engineering | 2004 |
| **3** | Sue | HR | 2014 |

The pd.merge() function recognizes that each DataFrame has an "employee" column, and automatically joins using this column as a key. The result of the merge is a new DataFrame that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the "employee" column differs between df1 and df2, and the pd.merge() function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index

**Many-to-one joins**

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],

'supervisor': ['Carly', 'Guido', 'Steve']})

Print('df3', 'df4', 'pd.merge(df3, df4)')

pd.merge(df3, df4)

|  | **employee** | **group** | **hire\_date** | **supervisor** |
| --- | --- | --- | --- | --- |
| **0** | Bob | Accounting | 2008 | Carly |
| **1** | Jake | Engineering | 2012 | Guido |
| **2** | Lisa | Engineering | 2004 | Guido |
| **3** | Sue | HR | 2014 | Steve |

The resulting DataFrame has an aditional column with the "supervisor" information, where the information is repeated in one or more locations as required by the inputs.

**Many-to-many joins**

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',

'Engineering', 'Engineering', 'HR', 'HR'],

'skills': ['math', 'spreadsheets', 'coding', 'linux',

'spreadsheets', 'organization']})

print('df1', 'df5', "pd.merge(df1, df5)")

pd.merge(df1, df5)

|  | **employee** | **group** | **skills** |
| --- | --- | --- | --- |
| **0** | Bob | Accounting | math |
| **1** | Bob | Accounting | spreadsheets |
| **2** | Jake | Engineering | coding |
| **3** | Jake | Engineering | linux |
| **4** | Lisa | Engineering | coding |
| **5** | Lisa | Engineering | linux |
| **6** | Sue | HR | spreadsheets |
| **7** | Sue | HR | organization |

These three types of joins can be used with other Pandas tools to implement a wide array of functionality.

**Specification of the Merge Key**

We've already seen the default behavior of pd.merge(): it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and pd.merge() provides a variety of options for handling this.

**The on keyword**

Most simply, you can explicitly specify the name of the key column using the on keyword, which takes a column name or a list of column names:

print('df1', 'df2', "pd.merge(df1, df2, on='employee')")

pd.merge(df1, df2, on='employee')

|  | **employee** | **group** | **hire\_date** |
| --- | --- | --- | --- |
| **0** | Bob | Accounting | 2008 |
| **1** | Jake | Engineering | 2012 |
| **2** | Lisa | Engineering | 2004 |
| **3** | Sue | HR | 2014 |

This option works only if both the left and right DataFrames have the specified column name.

**The left\_on and right\_on keywords**

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the left\_on and right\_on keywords to specify the two column names:

df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],

'salary': [70000, 80000, 120000, 90000]})

display('df1', 'df3', 'pd.merge(df1, df3, left\_on="employee", right\_on="name")')

pd.merge(df1, df3, left\_on="employee", right\_on="name")

|  | **employee** | **group** | **name** | **salary** |
| --- | --- | --- | --- | --- |
| **0** | Bob | Accounting | Bob | 70000 |
| **1** | Jake | Engineering | Jake | 80000 |
| **2** | Lisa | Engineering | Lisa | 120000 |
| **3** | Sue | HR | Sue | 90000 |

The result has a redundant column that we can drop if desired–for example, by using the drop() method of DataFrames:

pd.merge(df1, df3, left\_on="employee", right\_on="name").drop('name', axis=1)

|  | **employee** | **group** | **salary** |
| --- | --- | --- | --- |
| **0** | Bob | Accounting | 70000 |
| **1** | Jake | Engineering | 80000 |
| **2** | Lisa | Engineering | 120000 |
| **3** | Sue | HR | 90000 |

**The left\_index and right\_index keywords**

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

df1a = df1.set\_index('employee')

df2a = df2.set\_index('employee')

display('df1a', 'df2a')

df1a

|  | **group** |
| --- | --- |
| **employee** |  |
| **Bob** | Accounting |
| **Jake** | Engineering |
| **Lisa** | Engineering |
| **Sue** | HR |

df2a

|  | **hire\_date** |
| --- | --- |
| **employee** |  |
| **Lisa** | 2004 |
| **Bob** | 2008 |
| **Jake** | 2012 |
| **Sue** | 2014 |

You can use the index as the key for merging by specifying the left\_index and/or right\_index flags in pd.merge():

display('df1a', 'df2a',

"pd.merge(df1a, df2a, left\_index=True, right\_index=True)")

df1a

|  | **group** |
| --- | --- |
| **employee** |  |
| **Bob** | Accounting |
| **Jake** | Engineering |
| **Lisa** | Engineering |
| **Sue** | HR |

df2a

|  | **hire\_date** |
| --- | --- |
| **employee** |  |
| **Lisa** | 2004 |
| **Bob** | 2008 |
| **Jake** | 2012 |
| **Sue** | 2014 |

pd.merge(df1a, df2a, left\_index=True, right\_index=True)

|  | **group** | **hire\_date** |
| --- | --- | --- |
| **employee** |  |  |
| **Lisa** | Engineering | 2004 |
| **Bob** | Accounting | 2008 |
| **Jake** | Engineering | 2012 |
| **Sue** | HR | 2014 |

For convenience, DataFrames implement the join() method, which performs a merge that defaults to joining on indices:

display('df1a', 'df2a', 'df1a.join(df2a)')

df1a

|  | **group** |
| --- | --- |
| **employee** |  |
| **Bob** | Accounting |
| **Jake** | Engineering |
| **Lisa** | Engineering |
| **Sue** | HR |

df2a

|  | **hire\_date** |
| --- | --- |
| **employee** |  |
| **Lisa** | 2004 |
| **Bob** | 2008 |
| **Jake** | 2012 |
| **Sue** | 2014 |

df1a.join(df2a)

|  | **group** | **hire\_date** |
| --- | --- | --- |
| **employee** |  |  |
| **Bob** | Accounting | 2008 |
| **Jake** | Engineering | 2012 |
| **Lisa** | Engineering | 2004 |
| **Sue** | HR | 2014 |

If you'd like to mix indices and columns, you can combine left\_index with right\_on or left\_on with right\_index to get the desired behavior:

display('df1a', 'df3', "pd.merge(df1a, df3, left\_index=True, right\_on='name')")

df1a

|  | **group** |
| --- | --- |
| **employee** |  |
| **Bob** | Accounting |
| **Jake** | Engineering |
| **Lisa** | Engineering |
| **Sue** | HR |

df3

|  | **name** | **salary** |
| --- | --- | --- |
| **0** | Bob | 70000 |
| **1** | Jake | 80000 |
| **2** | Lisa | 120000 |
| **3** | Sue | 90000 |

pd.merge(df1a, df3, left\_index=True, right\_on='name')

|  | **group** | **name** | **salary** |
| --- | --- | --- | --- |
| **0** | Accounting | Bob | 70000 |
| **1** | Engineering | Jake | 80000 |
| **2** | Engineering | Lisa | 120000 |
| **3** | HR | Sue | 90000 |

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the ["Merge, Join, and Concatenate" section](http://pandas.pydata.org/pandas-docs/stable/merging.html) of the Pandas documentation.

Concatenating objects

The [**concat()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html#pandas.concat) function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say “if any” because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of concat and what it can do, here is a simple example:

***In [1]:*** *df1 = pd.DataFrame(*

***...:***  *{*

***...:*** *"A": ["A0", "A1", "A2", "A3"],*

***...:*** *"B": ["B0", "B1", "B2", "B3"],*

***...:*** *"C": ["C0", "C1", "C2", "C3"],*

***...:*** *"D": ["D0", "D1", "D2", "D3"],*

***...:***  *},*

***...:***  *index=[0, 1, 2, 3],*

***...:*** *)*

***...:***

***In [2]:*** *df2 = pd.DataFrame(*

***...:***  *{*

***...:*** *"A": ["A4", "A5", "A6", "A7"],*

***...:*** *"B": ["B4", "B5", "B6", "B7"],*

***...:*** *"C": ["C4", "C5", "C6", "C7"],*

***...:*** *"D": ["D4", "D5", "D6", "D7"],*

***...:***  *},*

***...:***  *index=[4, 5, 6, 7],*

***...:*** *)*

***...:***

***In [3]:*** *df3 = pd.DataFrame(*

***...:***  *{*

***...:*** *"A": ["A8", "A9", "A10", "A11"],*

***...:*** *"B": ["B8", "B9", "B10", "B11"],*

***...:*** *"C": ["C8", "C9", "C10", "C11"],*

***...:*** *"D": ["D8", "D9", "D10", "D11"],*

***...:***  *},*

***...:***  *index=[8, 9, 10, 11],*

***...:*** *)*

***...:***

***In [4]:*** *frames = [df1, df2, df3]*

***In [5]:*** *result = pd.concat(frames)*

**C. Sort Data**

**Python Pandas - Sorting**

There are two kinds of sorting available in Pandas. They are −

* By label
* By Actual Value

**By Label**

Using the **sort\_index()** method, by passing the axis arguments and the order of sorting,

**Order of Sorting**

By passing the Boolean value to ascending parameter, the order of the sorting can be controlled.

**Sort the Columns**

By passing the axis argument with a value 0 or 1, the sorting can be done on the column labels. By default, axis=0, sort by row. Let us consider the following example to understand the same.

**By Value**

Like index sorting, **sort\_values()** is the method for sorting by values. It accepts a 'by' argument which will use the column name of the DataFrame with which the values are to be sorted.

As a quick reminder, a **DataFrame** is a data structure with labeled axes for both rows and columns. You can sort a DataFrame by row or column value as well as by row or column index.Both rows and columns have **indices**, which are numerical representations of where the data is in your DataFrame. You can retrieve data from specific rows or columns using the DataFrame’s index locations. By default, index numbers start from zero. You can also manually assign your own index.

**Getting Familiar With .sort\_values()**

You use [.sort\_values()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html) to sort values in a DataFrame along either axis (columns or rows). Typically, you want to sort the rows in a DataFrame by the values of one or more columns:

**Getting Familiar With .sort\_index()**

You use .sort\_index() to sort a DataFrame by its row index or column labels. The difference from using .sort\_values() is that you’re sorting the DataFrame based on its row index or column names, not by the values in these rows or columns:

**Sorting Your DataFrame on a Single Column**

To sort the DataFrame based on the values in a single column, you’ll use .sort\_values(). By default, this will return a new DataFrame sorted in ascending order. It does not modify the original DataFrame.

**Sorting by a Column in Ascending Order**

To use .sort\_values(), you pass a single argument to the method containing the name of the column you want to sort by.

>>>df.sort\_values(by = "like")

By default, .sort\_values() sorts your data in **ascending order**.

**Changing the Sort Order**

Another parameter of .sort\_values() is ascending. By default .sort\_values() has ascending set to True. If you want the DataFrame sorted in **descending order**, then you can pass False to this parameter:

>>>df.sort\_values(by = "like",ascending= False)

**Choosing a Sorting Algorithm**

It’s good to note that pandas allows you to choose different **sorting algorithms** to use with both .sort\_values() and .sort\_index(). The available algorithms are quicksort, mergesort, and heapsort.

The algorithm used by default when sorting on a single column is [quicksort](https://en.wikipedia.org/wiki/Quicksort). To change this to a [stable](https://en.wikipedia.org/wiki/Category:Stable_sorts) sorting algorithm, use [mergesort](https://en.wikipedia.org/wiki/Merge_sort). You can do that with the kind parameter in .sort\_values() or .sort\_index(), like this:

>>>df.sort\_values(by = "like",ascending= False, kind="mergesort")

Using kind, you set the sorting algorithm to mergesort..

**Note:** In pandas, kind is ignored when you sort on more than one column or label.

When you’re sorting multiple records that have the same key, a **stable sorting algorithm(**[mergesort](https://en.wikipedia.org/wiki/Merge_sort)) will maintain the original order of those records after sorting. For that reason, using a stable sorting algorithm is necessary if you plan to perform multiple sorts.

**Sorting Your DataFrame on Multiple Columns**

In data analysis, it’s common to want to sort your data based on the values of multiple columns. Imagine you have a dataset with people’s first and last names. It would make sense to sort by last name and then first name, so that people with the same last name are arranged alphabetically according to their first names.

>>>df.sort\_values(by=["like", "share])

you sort the DataFrame on two columns using .sort\_values

**Changing the Column Sort Order**

Since you’re sorting using multiple columns, you can specify the order by which your columns get sorted. If you want to change the logical sort order from the previous example, then you can change the order of the column names in the list you pass to the by parameter:

>>>df.sort\_values(by=["share”,"like"])

Your DataFrame is now sorted by the model column in ascending order, then sorted by make if there are two or more of the same model. You can see that changing the order of columns also changes the order in which the values get sorted.

**Sorting by Multiple Columns in Descending Order**

Up to this point, you’ve sorted only in ascending order on multiple columns. In the next example, you’ll sort in descending order based on the make and model columns. To sort in descending order, set ascending to False:

>>>df.sort\_values(by=["share”,"like"],ascending=False)

The values in the share column are in reverse alphabetical order, and the values in the like column are in descending order. With textual data, the sort is **case sensitive**, meaning capitalized text will appear first in ascending order and last in descending order.

**Sorting by Multiple Columns With Different Sort Orders**

You might be wondering if it’s possible to sort using multiple columns and to have those columns use different ascending arguments. With pandas, you can do this with a single method call. If you want to sort some columns in ascending order and some columns in descending order, then you can pass a list of [Booleans](https://realpython.com/python-boolean/) to ascending.

In this example, you sort your DataFrame by the make, model, and city08 columns, with the first two columns sorted in ascending order and city08 sorted in descending order. To do so, you pass a list of column names to by and a list of Booleans to ascending:

>>>df.sort\_values(by=["share”,"like"],ascending=[True,False])

Now your DataFrame is sorted by share in ascending order, but with the like column in descending order.

**Sorting Your DataFrame on Its Index**

Before sorting on the index, it’s a good idea to know what an index represents. A DataFrame has an **.index** property, which by default is a numerical representation of its rows’ locations. You can think of the index as the row numbers. It helps in quick row lookup and identification.

**Sorting by Index in Ascending Order**

You can sort a DataFrame based on its row index with .sort\_index(). Sorting by column values like you did in the previous examples reorders the rows in your DataFrame, so the index becomes disorganized. This can also happen when you filter a DataFrame or when you drop or add rows.

To illustrate the use of .sort\_index(), start by creating a new sorted DataFrame using .sort\_values():

>>> sorted\_df = df.sort\_values(by=["like", "share"])

>>>print(sorted\_df)

You’ve created a DataFrame that’s sorted using multiple values. Notice how the row index is in no particular order. To get your new DataFrame back to the original order, you can use .sort\_index():

>>> sorted\_df.sort\_index()

Now the index is in ascending order. Just like .sort\_values(), the default argument for ascending in .sort\_index() is True, and you can change to descending order by passing False. Sorting on the index has no impact on the data itself as the values are unchanged.

This is particularly useful when you’ve assigned a custom index with **.set\_index()**. If you want to set a custom index using the like and share columns, then you can pass a list to .set\_index():

>>> assigned\_index\_df = df.set\_index(["like", "share"])

>>> assigned\_index\_df

Using this method, you replace the default integer-based row index with two axis labels. This is considered a [MultiIndex](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.MultiIndex.html#pandas.MultiIndex) or a **hierarchical index**. Your DataFrame is now indexed by more than one key, which you can sort on with .sort\_index():

>>> assigned\_index\_df.sort\_index()

**Sorting by Index in Descending Order**

For the next example, you’ll sort your DataFrame by its index in descending order. Remember from sorting your DataFrame with .sort\_values() that you can reverse the sort order by setting ascending to False. This parameter also works with .sort\_index(), so you can sort your DataFrame in reverse order like this:

>>> assigned\_index\_df.sort\_index(ascending=False)

**Sorting the Columns of Your DataFrame**

You can also use the column labels of your DataFrame to sort row values. Using .sort\_index() with the optional parameter axis set to 1 will sort the DataFrame by the column labels. The sorting algorithm is applied to the **axis labels** instead of to the actual data. This can be helpful for visual inspection of the DataFrame.

**Working With the DataFrame axis**

When you use .sort\_index() without passing any explicit arguments, it uses axis=0 as a default argument. The **axis** of a DataFrame refers to either the index (axis=0) or the columns (axis=1). You can use both axes for [indexing and selecting data](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html) in a DataFrame as well as for sorting the data.

**Using Column Labels to Sort**

You can also use the column labels of a DataFrame as the sorting key for .sort\_index(). Setting axis to 1 sorts the columns of your DataFrame based on the column labels:

>>> df.sort\_index(axis=1)

The columns of your DataFrame are sorted from left to right in ascending alphabetical order. If you want to sort the columns in descending order, then you can use ascending=False:

>>> df.sort\_index(axis=1, ascending=False)

**D. Transposing Data**

**Transpose DataFrame (swap rows and columns)**

Use the T attribute or the transpose() method to swap (= transpose) the rows and columns of pandas.DataFrame.

Neither method changes the original object but returns a new object with the rows and columns swapped (= transposed object).

Note that depending on the data type dtype of each column, a view is created instead of a copy, and changing the value of one of the original and transposed objects will change the other.

## Syntax

dataframe.transpose(args, copy)

**Parameters**

The parameters are [keyword arguments](https://www.w3schools.com/python/gloss_python_function_keyword_arguments.asp).

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value** | **Description** |
| args | Tuple | Optional. arguments that can be used in NumPy functions |
| copy | True False | Optional, default False. Specifies whether to copy the data or not |

## Return Value

A [DataFrame](https://www.w3schools.com/python/pandas/pandas_dataframes.asp) where the columns have been rows and vice versa.

This method does not change the original DataFrame.

Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

Examples

**Square DataFrame with homogeneous dtype**

d1 = {'col1': [1, 2], 'col2': [3, 4]}

df1 = pd.DataFrame(data=d1)

df1

col1 col2

0 1 3

1 2 4

df1\_transposed = df1.T # or df1.transpose()

df1\_transposed

0 1

col1 1 2

col2 3 4

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

df1.dtypes

col1 int64

col2 int64

dtype: object

df1\_transposed.dtypes

0 int64

1 int64

dtype: object

**Non-square DataFrame with mixed dtypes**

d2 = {'name': ['Alice', 'Bob'],

'score': [9.5, 8],

'employed': [False, True],

'kids': [0, 0]}

df2 = pd.DataFrame(data=d2)

df2

name score employed kids

0 Alice 9.5 False 0

1 Bob 8.0 True 0

df2\_transposed = df2.T # or df2.transpose()

df2\_transposed

0 1

name Alice Bob

score 9.5 8.0

employed False True

kids 0 0

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

df2.dtypes

name object

score float64

employed bool

kids int64

dtype: object

df2\_transposed.dtypes

0 object

1 object

dtype: object

Example : result = df.transpose()

print(result)

**E. Shape and reshape Data**

The shape property returns a tuple containing the shape of the DataFrame.

The shape is the number of rows and columns of the DataFrame

dataframe.shape

Return Value

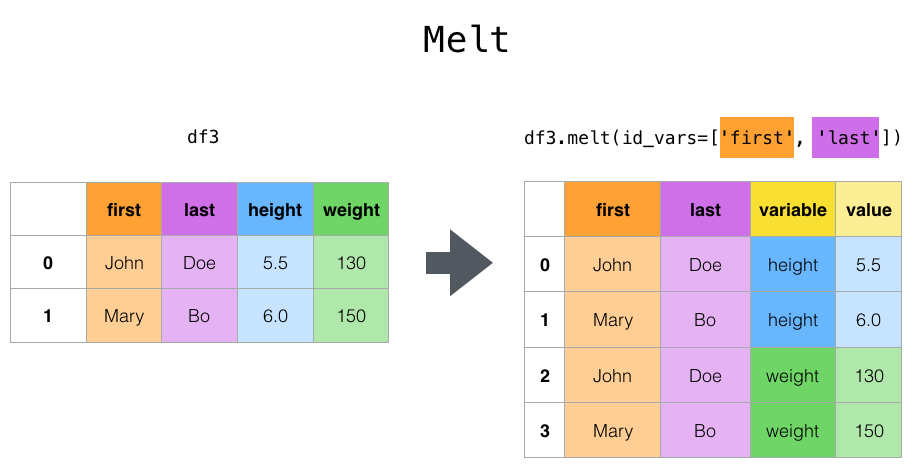
a Python Tuple showing the number of rows and columns.

reshaping means the transformation of the structure of a table or vector (i.e. DataFrame or Series) to make it suitable for further analysis..

Reshaping by melt

To make analysis of data in table easier, we can reshape the data into a more computer-friendly form using Pandas in Python. Pandas.melt() is one of the function to do so..  
Pandas.melt() unpivots a DataFrame from wide format to long format.  
**melt()** function is useful to message a DataFrame into a format where one or more columns are identifier variables, while all other columns, considered measured variables, are unpivoted to the row axis, leaving just two non-identifier columns, variable and value.  
Syntax :

*pandas.melt(frame, id\_vars=None, value\_vars=None,var\_name=None, value\_name='value', col\_level=None)*



The top-level [**melt()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html#pandas.melt) function and the corresponding [**DataFrame.melt()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.melt.html#pandas.DataFrame.melt) are useful to massage a [**DataFrame**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html#pandas.DataFrame) into a format where one or more columns are *identifier variables*, while all other columns, considered *measured variables*, are “unpivoted” to the row axis, leaving just two non-identifier columns, “variable” and “value”. The names of those columns can be customized by supplying the var\_name and value\_name parameters.

For instance,

When transforming a DataFrame using [**melt()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html#pandas.melt), the index will be ignored. The original index values can be kept around by setting the ignore\_index parameter to False (default is True). This will however duplicate them.

***Exa.***

***In [50]:*** *index = pd.MultiIndex.from\_tuples([("person", "A"), ("person", "B")])*

***In [51]:*** *cheese = pd.DataFrame(*

***....:***  *{*

***....:*** *"first": ["John", "Mary"],*

***....:*** *"last": ["Doe", "Bo"],*

***....:*** *"height": [5.5, 6.0],*

***....:*** *"weight": [130, 150],*

***....:***  *},*

***....:***  *index=index,*

***....:*** *)*

***....:***

***In [52]:*** *cheese*

***Out[52]:***

*first last height weight*

*person A John Doe 5.5 130*

*B Mary Bo 6.0 150*

***In [53]:*** *cheese.melt(id\_vars=["first", "last"])*

***Out[53]:***

*first last variable value*

*0 John Doe height 5.5*

*1 Mary Bo height 6.0*

*2 John Doe weight 130.0*

*3 Mary Bo weight 150.0*

***In [54]:*** *cheese.melt(id\_vars=["first", "last"], ignore\_index=****False****)*

***Out[54]:***

*first last variable value*

*person A John Doe height 5.5*

*B Mary Bo height 6.0*

*A John Doe weight 130.0*

*B Mary Bo weight 150.0*

Another way to transform is to use the [**wide\_to\_long()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.wide_to_long.html#pandas.wide_to_long) panel data convenience function. It is less flexible than [**melt()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html#pandas.melt), but more user-friendly.

***In [55]:*** *dft = pd.DataFrame(*

***....:***  *{*

***....:*** *"A1970": {0: "a", 1: "b", 2: "c"},*

***....:*** *"A1980": {0: "d", 1: "e", 2: "f"},*

***....:*** *"B1970": {0: 2.5, 1: 1.2, 2: 0.7},*

***....:*** *"B1980": {0: 3.2, 1: 1.3, 2: 0.1},*

***....:*** *"X": dict(zip(range(3), np.random.randn(3))),*

***....:***  *}*

***....:*** *)*

***....:***

***In [56]:*** *dft["id"] = dft.index*

***In [57]:*** *dft*

***Out[57]:***

*A1970 A1980 B1970 B1980 X id*

*0 a d 2.5 3.2 -0.121306 0*

*1 b e 1.2 1.3 -0.097883 1*

*2 c f 0.7 0.1 0.695775 2*

***In [58]:*** *pd.wide\_to\_long(dft, ["A", "B"], i="id", j="year")*

***Out[58]:***

*X A B*

*id year*

*0 1970 -0.121306 a 2.5*

*1 1970 -0.097883 b 1.2*

*2 1970 0.695775 c 0.7*

*0 1980 -0.121306 d 3.2*

*1 1980 -0.097883 e 1.3*

*2 1980 0.695775 f 0.1*

**Pivot**

The pivot function is used to create a new derived table out of a given one. Pivot takes 3 arguements with the following names: index, columns, and values. As a value for each of these parameters you need to specify a column name in the original table. Then the pivot function will create a new table, whose row and column indices are the unique values of the respective parameters. The cell values of the new table are taken from column given as the values parameter.

Assume that we are given the following small table:

In [1]:

**fromcollectionsimport** OrderedDict

**frompandasimport** DataFrame

**importpandasaspd**

**importnumpyasnp**

table = OrderedDict((

("Item", ['Item0', 'Item0', 'Item1', 'Item1']),

('CType',['Gold', 'Bronze', 'Gold', 'Silver']),

('USD', ['1$', '2$', '3$', '4$']),

('EU', ['1€', '2€', '3€', '4€'])

))

d = DataFrame(table)

d

Out[1]:

|  | **Item** | **CType** | **USD** | **EU** |
| --- | --- | --- | --- | --- |
| **0** | Item0 | Gold | 1$ | 1€ |
| **1** | Item0 | Bronze | 2$ | 2€ |
| **2** | Item1 | Gold | 3$ | 3€ |
| **3** | Item1 | Silver | 4$ | 4€ |

In [2]:

p = d.pivot(index='Item', columns='CType', values='USD')

p

Out[2]:

| **CType** | **Bronze** | **Gold** | **Silver** |
| --- | --- | --- | --- |
| **Item** |  |  |  |
| **Item0** | 2$ | 1$ | None |
| **Item1** | None | 3$ | 4$ |

As can be seen above, the value of USD for every row in the original table has been transferred to the new table, where its row and column match the Item and CType of its original row. Cells in the new table which do not have a matching entry in the original one are set with NaN.

**Pivoting By Multiple Columns**

Now what if we want to extend the previous example to have the EU cost for each item on its row as well? This is actually easy - we just have to omit the values parameter as follows:

In [4]:

p = d.pivot(index='Item', columns='CType')

p

Out[4]:

|  | **USD** | | | **EU** | | |
| --- | --- | --- | --- | --- | --- | --- |
| **CType** | **Bronze** | **Gold** | **Silver** | **Bronze** | **Gold** | **Silver** |
| **Item** |  |  |  |  |  |  |
| **Item0** | 2$ | 1$ | None | 2€ | 1€ | None |
| **Item1** | None | 3$ | 4$ | None | 3€ | 4€ |

As shown above, Pandas will create a hierarchical column index (MultiIndex) for the new table. You can think of a hierarchical index as a set of trees of indices. Each indexed column/row is identified by a unique sequence of values defining the “path” from the topmost index to the bottom index. The first level of the column index defines all columns that we have not specified in the pivot invocation - in this case USD and EU. The second level of the index defines the unique value of the corresponding column.

We can use this hierarchical column index to filter the values of a single column from the original table. For example p.USD returns a pivoted DataFrame with the USD values only and it is equivalent to the pivoted DataFrame from the previous section.

In [5]:

p.USD

Out[5]:

| **CType** | **Bronze** | **Gold** | **Silver** |
| --- | --- | --- | --- |
| **Item** |  |  |  |
| **Item0** | 2$ | 1$ | None |
| **Item1** | None | 3$ | 4$ |

In [6]:

p.USD.Bronze

Out[6]:

Item

Item0 2$

Item1 None

Name: Bronze, dtype: object

In [7]:

*# Original DataFrame: Access the USD cost of Item0 for Gold customers*

print(d[(d.Item=='Item0') & (d.CType=='Gold')].USD.values)

*# Pivoted DataFrame: p.USD gives a "sub-DataFrame" with the USD values only*

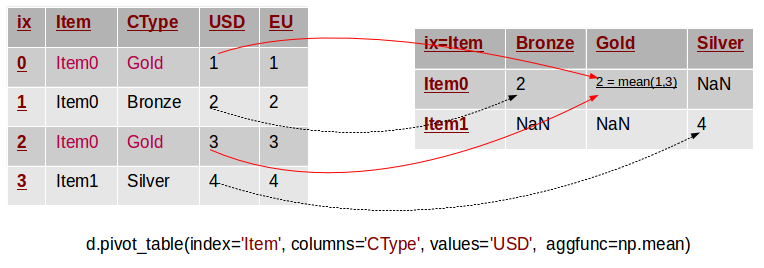
print(p.USD[p.USD.index=='Item0'].Gold.values)

['1$']

['1$']

**Pivot Table**

The pivot\_table method comes to solve this problem. It works like pivot, but it aggregates the values from rows with duplicate entries for the specified columns. In other words, in the previous example we could have used the mean, the median or another aggregation function to compute a single value from the conflicting entries. This is depicted in the example below.



In [8]:

table = OrderedDict((

("Item", ['Item0', 'Item0', 'Item0', 'Item1']),

('CType',['Gold', 'Bronze', 'Gold', 'Silver']),

('USD', [1, 2, 3, 4]),

('EU', [1.1, 2.2, 3.3, 4.4])

))

d = DataFrame(table)

p = d.pivot\_table(index='Item', columns='CType', values='USD', aggfunc=np.sum)

p.fillna(value='--',inplace=**True**)

p

Out[8]:

| **CType** | **Bronze** | **Gold** | **Silver** |
| --- | --- | --- | --- |
| **Item** |  |  |  |
| **Item0** | 2 | 4 | -- |
| **Item1** | -- | -- | 4 |

In essence pivot\_table is a generalisation of pivot, which allows you to aggregate multiple values with the same destination in the pivoted table.

While [**pivot()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pivot.html#pandas.DataFrame.pivot) provides general purpose pivoting with various data types (strings, numerics, etc.), pandas also provides [**pivot\_table()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.pivot_table.html#pandas.pivot_table) for pivoting with aggregation of numeric data.

The function [**pivot\_table()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.pivot_table.html#pandas.pivot_table) can be used to create spreadsheet-style pivot tables. It takes a number of arguments:

* data: a DataFrame object.
* values: a column or a list of columns to aggregate.
* index: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
* columns: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
* aggfunc: function to use for aggregation, defaulting to numpy.mean.

Consider a data set like this:

**In [64]: importdatetime**

**In [65]:** df = pd.DataFrame(

**....:**  {

**....:** "A": ["one", "one", "two", "three"] \*6,

**....:** "B": ["A", "B", "C"] \*8,

**....:** "C": ["foo", "foo", "foo", "bar", "bar", "bar"] \*4,

**....:** "D": np.random.randn(24),

**....:** "E": np.random.randn(24),

**....:** "F": [datetime.datetime(2013, i, 1) **for** i **in**range(1, 13)]

**....:** + [datetime.datetime(2013, i, 15) **for** i **in**range(1, 13)],

**....:**  }

**....:** )

**....:**

**In [66]:** df

**Out[66]:**

A B C D E F

0 one A foo 0.341734 -0.317441 2013-01-01

1 one B foo 0.959726 -1.236269 2013-02-01

2 two C foo -1.110336 0.896171 2013-03-01

3 three A bar -0.619976 -0.487602 2013-04-01

4 one B bar 0.149748 -0.082240 2013-05-01

.. ... .. ... ... ... ...

19 three B foo 0.690579 -2.213588 2013-08-15

20 one C foo 0.995761 1.063327 2013-09-15

21 one A bar 2.396780 1.266143 2013-10-15

22 two B bar 0.014871 0.299368 2013-11-15

23 three C bar 3.357427 -0.863838 2013-12-15

[24 rows x 6 columns]

We can produce pivot tables from this data very easily:

**In [67]:** pd.pivot\_table(df, values="D", index=["A", "B"], columns=["C"])

**Out[67]:**

C bar foo

A B

one A 1.120915 -0.514058

B -0.338421 0.002759

C -0.538846 0.699535

three A -1.181568 NaN

B NaN 0.433512

C 0.588783 NaN

two A NaN 1.000985

B 0.158248 NaN

C NaN 0.176180

**Note that**[**pivot\_table()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pivot_table.html#pandas.DataFrame.pivot_table)**is also available as an instance method on DataFrame,**

i.e. [**DataFrame.pivot\_table()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pivot_table.html#pandas.DataFrame.pivot_table).

Adding margins

If you pass margins=True to [**pivot\_table()**](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.pivot_table.html#pandas.DataFrame.pivot_table), special All columns and rows will be added with partial group aggregates across the categories on the rows and columns:

***In [74]:*** *table = df.pivot\_table(index=["A", "B"], columns="C", margins=****True****, aggfunc=np.std)*

***In [75]:*** *table*

***Out[75]:***

*D E*

*C bar foo All bar foo All*

*A B*

*one A 1.804346 1.210272 1.569879 0.179483 0.418374 0.858005*

*B 0.690376 1.353355 0.898998 1.083825 0.968138 1.101401*

*C 0.273641 0.418926 0.771139 1.689271 0.446140 1.422136*

*three A 0.794212 NaN 0.794212 2.049040 NaN 2.049040*

*B NaN 0.363548 0.363548 NaN 1.625237 1.625237*

*C 3.915454 NaN 3.915454 1.035215 NaN 1.035215*

*two A NaN 0.442998 0.442998 NaN 0.447104 0.447104*

*B 0.202765 NaN 0.202765 0.560757 NaN 0.560757*

*C NaN 1.819408 1.819408 NaN 0.650439 0.650439*

*All 1.556686 0.952552 1.246608 1.250924 0.899904 1.059389*

**Conclusion**

You have now learned the three most important techniques for combining data in Pandas:

1. creating data subset
2. **merge()** for combining data on common columns or indices
3. **concat()** for combining DataFrames across rows or columns.
4. Sort a **pandas DataFrame** by the values of one or more columns
5. Use the ascending parameter to change the **sort order**
6. Sort a DataFrame by its index using **.sort\_index()**
7. **transposing data**
8. Reshaping data