# Pandas

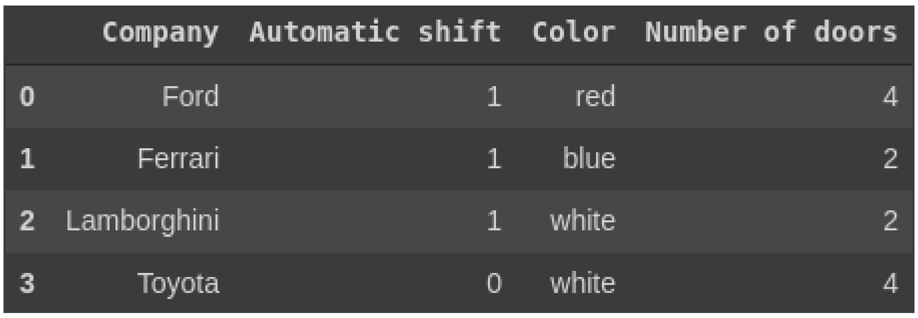
## Learning Objectives:

By the end of this lesson, all learners will be able to:

- create series and dataframes to manipulate and organize the data efficiently
- implement row and column selection techniques with loc and iloc in pandas dataframe
- use Join, Merge and Concat for combining multiple dataframes into single one
- discuss apply and map funtions for various operations in dataframe

## Introduction

The "ndarray" data structure of Numpy provides features for clean, well-organized, and many high-level operations. However, it is inflexible while working with missing data, attaching labels to the data, and attempting operations that don't require broadcasting (e.g., grouping, pivots, etc.). Pandas, built on top of NumPy, provides an efficient solution to these sorts of "data munging" tasks.



| | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| 0 | Ford | 1 | red | 4 |
| 1 | Ferrari | 1 | blue | 2 |
| 2 | Lamborghini | 1 | white | 2 |
| 3 | Toyota | 0 | white | 4 |

In general, "Pandas" is a way to store data efficiently on a computer. In the above image, you can see a table of data storing information about 4 cars. Usually, data required for machine learning contains many more features (the columns that tell us the different attributes of the cars), and observations (the rows that represent different cars). "Pandas" helps us organize and manipulate this data. It is a popular open-source library in Python, which means that you can get it for free.

## Series

If you look at one individual column, "Company," for example, this is a Series. There are four columns, which means that this table is made up of four pandas series put together. If you are wondering about the numbers 0, 1, 2, and 3 on the left, they are just indices, so we don't count them.

Alternatively, you can also look at each row in above dataframe as a Series. The first row(column headings) where it says "Company," "Automatic shift," etc. is the equivalent of indices, we do not count them. They simply describe what the data means as they are heading for each columns. If you look at it this way, there are four rows, so there are four pandas Series.

Let us create a pandas Series.

In [ ]:

```python
import pandas as pd

name_companies = ['Ford', 'Ferrari', 'Lamborghini', 'Toyota']

# Create Series
names = pd.Series(name_companies)

names
```

Out[1]:

```
0           Ford
1        Ferrari
2    Lamborghini
3         Toyota
dtype: object
```

In the above code, we first made a list of the values we want to put in the Series, and then assigned it to the variable `names` . Then we created a Series object and assigned it to `names` .

We can also print the values and index of the Series.

In [ ]:

```python
print(names.values)
print(names.index)
```

```
['Ford' 'Ferrari' 'Lamborghini' 'Toyota']
RangeIndex(start=0, stop=4, step=1)
```

## Series - an enhanced version of NumPy

The series object looks like a one-dimensional NumPy array. However, while the index in a NumPy array is implicitly defined, the series in Pandas have explicitly defined the index. Due to this explicit definition, the series object is not limited only to the integer index, but we can also have strings as an index.

In [ ]:

```python
data = pd.Series([5, 10, 15, 20],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[3]:

```
a     5
b    10
c    15
d    20
dtype: int64
```

## Access data from series

In [ ]:

```
# numerical indexing/slicing as numpy
print(data[0])
print(data[0:4])
```

```
5
a     5
b    10
c    15
d    20
dtype: int64
```

In [ ]:

```
# accessomg data using the actual index
print(data['a'])
```

```
5
```

## DataFrame

The table that we saw is called a DataFrame in pandas. It is a simple table that stores data. The "pandas" library aims to enhance data manipulation; it is good for us to put the data into a pandas DataFrame object. It allows us to do many operations effortlessly.

Let us create a DataFrame.

In [ ]:

```
import pandas as pd

#create dataframe
df = pd.DataFrame()

print(df)
```

```
Empty DataFrame
Columns: []
Index: []
```

As you can imagine, this DataFrame is empty. Now let us add the first column. We use the Series we described above to create this.

In [ ]:

```
import pandas as pd

name_companies = ['Ford', 'Ferrari', 'Lamborghini', 'Toyota']

# Create Series
names = pd.Series(name_companies)

#create dataframe
df = pd.DataFrame()
df['Company'] = names

df
```

Out[7]:

|  | Company |
|---|---|
| **0** | Ford |
| **1** | Ferrari |
| **2** | Lamborghini |
| **3** | Toyota |

We create a new column inside the DataFrame. We do this when we write `df['Company']` . Since this is an empty DataFrame, there is no column called 'Company', so the DataFrame creates one and puts the contents in it.

This DataFrame has just one column. We need to make other columns as well.

In [ ]:

```
import pandas as pd

name_companies = ['Ford', 'Ferrari', 'Lamborghini', 'Toyota']
shift_list = [1,1,1,0]
colors = ['red', 'blue', 'white', 'white']
door_num = [4,2,2,4]

#create dataframe
df = pd.DataFrame(data={'Company':name_companies,
                        'Automatic shift':shift_list,
                        'Color':colors,
                        'Number of doors':door_num})

#display dataframe
df
```

Out[1]:

|  | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| **0** | Ford | 1 | red | 4 |
| **1** | Ferrari | 1 | blue | 2 |
| **2** | Lamborghini | 1 | white | 2 |
| **3** | Toyota | 0 | white | 4 |

In the code above, we have first created the lists that represent the information in the columns. Then, instead of creating a Series, we put the information directly into a DataFrame. We made a dictionary, with the keys representing the names of the column names, and the values representing the information.

We can also view more information about DataFrame.

In [ ]:

```
df.shape
```

Out[9]:

```
(4, 4)
```

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column            Non-Null Count  Dtype 
---  ------            --------------  ----- 
 0   Company           4 non-null      object
 1   Automatic shift   4 non-null      int64 
 2   Color             4 non-null      object
 3   Number of doors   4 non-null      int64 
dtypes: int64(2), object(2)
memory usage: 256.0+ bytes
```

In [ ]:

```
df.describe()
```

Out[11]:

| | Automatic shift | Number of doors |
|---|---|---|
| **count** | 4.00 | 4.000000 |
| **mean** | 0.75 | 3.000000 |
| **std** | 0.50 | 1.154701 |
| **min** | 0.00 | 2.000000 |
| **25%** | 0.75 | 2.000000 |
| **50%** | 1.00 | 3.000000 |
| **75%** | 1.00 | 4.000000 |
| **max** | 1.00 | 4.000000 |

## Reading CSV file into DataFrame

Usually, in a machine learning project, the data are stored in a file on a computer. This file is usually of a popular format called CSV. It stands for Comma Separated Values. It is a text file that contains the table. A comma separates each element in the table, hence the name.

Reading from this file is made very easy in Pandas.

```
df = pd.read_csv(dataset_path)
```

That's it! All we need to do is pass the path for finding the file to the `read_csv` function.

## Selecting rows and columns

Now that we have learned to build a data frame, we are going to learn how to access information from them. There are multiple ways to select and index rows and columns from Pandas DataFrames. There are three main options to achieve the selection and indexing activities in Pandas.

### Different Approaches for Selecting rows and columns

One dimensional object is obtained as output when we extract a single row or single column, which is called the Pandas Series data structure.

Here, we specify the name of columns and rows for which we want to retrieve the data.

Throughout this notebook, we take a dataframe as a reference. Let's make it then.

**Creating a dataframe**

In [ ]:

```
name_companies = ['Ford', 'Ferrari', 'Lamborghini', 'Toyota']
shift_list = [1,1,1,0]
colors = ['red', 'blue', 'white', 'white']
door_num = [4,2,2,4]

#create dataframe
df = pd.DataFrame(data={'Company':name_companies,
                        'Automatic shift':shift_list,
                        'Color':colors,
                        'Number of doors':door_num})

#display dataframe
df
```

Out[14]:

| | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| **0** | Ford | 1 | red | 4 |
| **1** | Ferrari | 1 | blue | 2 |
| **2** | Lamborghini | 1 | white | 2 |
| **3** | Toyota | 0 | white | 4 |

Hopefully, everyone understands what the above code is doing. It creates four lists and uses them as rows. It names the columns as well.

Now, we will use **`df['column_name']`** approach to extract the data from specified columns.

In [ ]:

```
df['Company']
```

Out[3]:

```
0          Ford
1       Ferrari
2    Lamborghini
3        Toyota
Name: Company, dtype: object
```

We can also extract the specific rows as specified in `df[row_name]`

In [ ]:

```
df[1:2] #only start index is inclusive so only first row is extracted
```

Out[4]:

| | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| **1** | Ferrari | 1 | blue | 2 |

In [ ]:

```
df[1:3]
```

Out[5]:

| | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| **1** | Ferrari | 1 | blue | 2 |
| **2** | Lamborghini | 1 | white | 2 |

## Using `loc` , `iloc`

### Label based Selection: `loc`

`loc` method uses the label of the rows and columns to select the required data. Imagine that you have a table with people and their personal information. This table might contain their name, age, sex, etc. You need to find information about a person, "Peter." To do this, you look at the column marked "name," and then scroll until you find the particular person's name. Then you want to get all the information in his corresponding row.

In the dataframe we have created, let us try to simulate this by supposing that we want the information about the entry, which has a value of "blue" as index.

To do this, first, we tell Pandas about our column of interest. We use the `df.set_index()` command to do it.

Then we use the `.loc` method to find the row that contains the information we want.

In [ ]:

```
df.set_index('Color', inplace= True)
df.loc['blue']  # extracts single row corresponding to the label specified.
```

Out[15]:

```
Company             Ferrari
Automatic shift           1
Number of doors           2
Name: blue, dtype: object
```

The first line in the above code tells Pandas that our data is in the column named "Color". We changed the index for the dataframe from 0, 1, 2, 3, to the items in this column. Then we checked it for where the value was "red" using `df.loc['red']` .

One thing that needs more explanation is the `inplace=True` parameter. When we set the index using `df.set_index()` we modify the dataframe. In our example, if we did not put `inplace=True` , Pandas would first make a copy of the dataframe and then make modifications to the copy. In this case, it would change the index of the copy, and then return this duplicate dataframe with the modifications.

In [ ]:

```
df
```

Out[16]:

| | **Company** | **Automatic shift** | **Number of doors** |
|---|---|---|---|
| **Color** | | | |
| **red** | Ford | 1 | 4 |
| **blue** | Ferrari | 1 | 2 |
| **white** | Lamborghini | 1 | 2 |
| **white** | Toyota | 0 | 4 |

In [ ]:

```
# For multiple rows extraction
df.loc[['red', 'blue']]
```

Out[17]:

| | **Company** | **Automatic shift** | **Number of doors** |
|---|---|---|---|
| **Color** | | | |
| **red** | Ford | 1 | 4 |
| **blue** | Ferrari | 1 | 2 |

In [ ]:

```
# Selecting both rows and columns
df.loc[['red', 'blue'],['Company', 'Number of doors']]
```

Out[18]:

| Color | Company | Number of doors |
|---|---|---|
| **red** | Ford | 4 |
| **blue** | Ferrari | 2 |

### Index based Selection - `iloc`

The `iloc` command for Pandas Dataframe stands for **integer-location(iloc)**. It selects the row and column based on the position.

```
df.iloc[<row selection>, <column selection>]
```

`iloc` in Pandas selects rows and columns by number, in the order that they appear in the data frame. You can imagine that each row has a row number from 0 to the total rows, and `iloc[]` allows selections based on these numbers. The same applies to columns.

For example, in the dataframe above, suppose we want to select the letter 'g'. It is in the second row and third column. However, remember that in programming, we start counting from zero. So, this means that it is on row `1` and column `2`.

We would use the command `df.iloc[1,2]` to access the letter.

In [ ]:

```
# retrieving rows by iloc method
df.iloc[2]
```

Out[19]:

```
Company            Lamborghini
Automatic shift              1
Number of doors              2
Name: white, dtype: object
```

In [ ]:

```
# retrieving a specific element with both row and column
df.iloc[1,2]
```

Out[20]:

```
2
```

In [ ]:

```
# selecting multiple rows by iloc method
df.iloc[[1,2]]
```

Out[21]:

| | Company | Automatic shift | Number of doors |
|---|---|---|---|
| **Color** | | | |
| **blue** | Ferrari | 1 | 2 |
| **white** | Lamborghini | 1 | 2 |

In [ ]:

```
df.iloc[:,1] #retrieve all the rows of second column
```

Out[22]:

```
Color
red      1
blue     1
white    1
white    0
Name: Automatic shift, dtype: int64
```

In [ ]:

```
# retrieving two rows and two columns by iloc method
df.iloc [[0, 1], [1, 2]]
```

Out[23]:

| | Automatic shift | Number of doors |
|---|---|---|
| **Color** | | |
| **red** | 1 | 4 |
| **blue** | 1 | 2 |

In [ ]:

```
#retrieving all rows and some columns by iloc method
df.iloc [:, [1, 2]]
```

Out[24]:

| | Automatic shift | Number of doors |
|---|---|---|
| **Color** | | |
| **red** | 1 | 4 |
| **blue** | 1 | 2 |
| **white** | 1 | 2 |
| **white** | 0 | 4 |

**What is the difference between `iloc` and `loc` ?**

If you want row labeled output then you use `loc[label]` and if you want a row with specific index output, then you use as `iloc[position]` . `iloc` is used to index a dataframe using 0 to (length-1) be it either row or column indices.

Many times, it becomes a necessity to select the data from a specific condition. This selection is quite natural to do with `loc` . We can pass an array to the `loc` method.

## Boolean indices for the selection of the row

Let us suppose you have a database of people and information about them. You might want to find people that are above the age of 50, for example. To do this, we can imagine writing code that looks like something like `"age">50` . But how do we make this work in pandas?

The answer is, by using the `.loc` method. When we use operators to check conditions like less than, greater than, equal to, etc., they return either true or false. If the age is indeed greater, we get True. The `.loc` method then selects these rows that have returned "True" (successfully passed the condition).

Let us do an example. Let us say we want to select all the rows where the column "two" equals "f". In a practical situation, this query is analogous to finding all the people whose addresses equals "China.". In our case, we of finding "f," we use the following command.

In [ ]:

```
name_companies = ['Ford', 'Ferrari', 'Lamborghini', 'Toyota']
shift_list = [1,1,1,0]
colors = ['red', 'blue', 'white', 'white']
door_num = [4,2,2,4]

#create dataframe
df = pd.DataFrame(data={'Company':name_companies,
                        'Automatic shift':shift_list,
                        'Color':colors,
                        'Number of doors':door_num})

#display dataframe
df
```

Out[25]:

| | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| **0** | Ford | 1 | red | 4 |
| **1** | Ferrari | 1 | blue | 2 |
| **2** | Lamborghini | 1 | white | 2 |
| **3** | Toyota | 0 | white | 4 |

In [ ]:

```
df.loc[df['Company']=='Ford']
```

Out[26]:

| | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| **0** | Ford | 1 | red | 4 |

We only got one row here, because we have the letter 'f' in only one row of this column. In a real example, this function returns hundred of rows as the data might contain hundreds of people that live in China. You could also use a command similar to `df['age']>50` to select people above the age of 50.

The code does need more explanation. First, look at the section inside the square brackets `df['Company']` =='Ford'. This code selects the indexes where the condition is true. First, we select the column "two" using `df['Company']`. Then it returns a series from which we check equality with the "Ford". Then we use this information and use `df.loc[]` to find the respective rows where the condition is True. It is a three-step process.

Let us go one step further and find specific information. You might not need all the information from the rows where the condition is true. You might want to display the address column of all the people that are above 50, for example.

Let us suppose you want to find the information in column three for all entries that have "Ford" in column two. Here, the condition is that we need the "Ford" on column two. If we meet this condition, we extract the rows. Finally, from these rows, we only pick out the column "three" because that is the only information we want.

We can do this using the following code:

In [ ]:

```
df.loc[df['Company'] == 'Ford' , ['Color']]
```

Out[27]:

| | Color |
|---|---|
| **0** | red |

Notice how we have selected the third column. In real life, datasets are massive. So commands that select only specific parts of the data are handy.

With our previous example, you might want to find all the people that have their age above 50 and extract their address and phone number. If you wanted to see what they have in two or more columns, you could do so with the following command.

In [ ]:

```
df.loc[df['Company'] == 'Ford' , ['Color','Number of doors']]
```

Out[28]:

| | Color | Number of doors |
|---|---|---|
| **0** | red | 4 |

As you can see, we simply put in the columns we want inside the square brackets.

**Multiple boolean indices for the selection of the row**

In [ ]:

```
df.loc[(df["Company"] =='Ford') & (df["Number of doors"] == 4)]
```

Out[31]:

|  | Company | Automatic shift | Color | Number of doors |
|---|---|---|---|---|
| **0** | Ford | 1 | red | 4 |

# Join, Merge and Apply

When handling data, you might want to merge different datasets in specific ways. You might also want to apply different functions to some particular columns.

## Joining and Merging Dataframes

### Join

Joining is a more convenient method for combining the columns of two potentially differently-indexed DataFrames into a single DataFrame. Joining is similar to merging except join method occurs on the index key instead of any particular column.

There are various join logics available to merge the pandas dataframe based on a common column.

We are creating two datasets.

In [ ]:

```
import pandas as pd
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3']},
                     index=['K0', 'K1', 'K2', 'K3'])
df2 = pd.DataFrame({'C': ['A4', 'A5', 'A6', 'A7', 'A8'],
                    'D': ['B4', 'B5', 'B6', 'B7', 'B8']},
                         index=['K0', 'K1', 'K2', 'K3', 'K4'])
df1
```

Out[1]:

|  | A | B |
|---|---|---|
| **K0** | A0 | B0 |
| **K1** | A1 | B1 |
| **K2** | A2 | B2 |
| **K3** | A3 | B3 |

```
In [ ]:
```

```
df2
```

```
Out[2]:
```

|  | C | D |
|---|---|---|
| **K0** | A4 | B4 |
| **K1** | A5 | B5 |
| **K2** | A6 | B6 |
| **K3** | A7 | B7 |
| **K4** | A8 | B8 |

We can specify the type of join as a parameter to the join function. The join scheme may be Outer join, Inner join, Left join, Right join.

**Inner Join**

```
In [ ]:
```

```
df1.join(df2, how = 'inner') # returns K0, K1, K2, K3
```

```
Out[32]:
```

|  | A | B | C | D |
|---|---|---|---|---|
| **K0** | A0 | B0 | A4 | B4 |
| **K1** | A1 | B1 | A5 | B5 |
| **K2** | A2 | B2 | A6 | B6 |
| **K3** | A3 | B3 | A7 | B7 |

Inner join returns all rows from both tables where the key record of one is equal to the other. It focuses on the commonality of the two tables.

**Outer Join**

```
In [ ]:
```

```
df1.join(df2, how='outer') # returns the record corresponding K0, K1, K2, K3, K4
```

```
Out[33]:
```

|  | A | B | C | D |
|---|---|---|---|---|
| **K0** | A0 | B0 | A4 | B4 |
| **K1** | A1 | B1 | A5 | B5 |
| **K2** | A2 | B2 | A6 | B6 |
| **K3** | A3 | B3 | A7 | B7 |
| **K4** | NaN | NaN | A8 | B8 |

It returns a set of records that include what an inner join would return and also adds other rows for which there is no corresponding match in another table.

**Right Join**

```
In [ ]:
```

```
df1.join(df2, how='right')
```

```
Out[34]:
```

| | A | B | C | D |
|---|---|---|---|---|
| **K0** | A0 | B0 | A4 | B4 |
| **K1** | A1 | B1 | A5 | B5 |
| **K2** | A2 | B2 | A6 | B6 |
| **K3** | A3 | B3 | A7 | B7 |
| **K4** | NaN | NaN | A8 | B8 |

It returns a set of records that include all the data in the right dataframe. In case of missing values of the "on" variable in the left dataframe, it adds empty/ Nan values in the result. This is clear from the above example.

**Left Join**

```
In [ ]:
```

```
df1.join(df2, how='left')
```

```
Out[35]:
```

| | A | B | C | D |
|---|---|---|---|---|
| **K0** | A0 | B0 | A4 | B4 |
| **K1** | A1 | B1 | A5 | B5 |
| **K2** | A2 | B2 | A6 | B6 |
| **K3** | A3 | B3 | A7 | B7 |

It returns a set of records that include all the data in the left dataframe. In case of missing values of the "on" variable in the right dataframe, it adds empty/ Nan values in the result. This is clear from the above example.

**Append Function**

In [ ]:

```
df1.append(df2)
```

Out[36]:

|  | A | B | C | D |
|---|---|---|---|---|
| **K0** | A0 | B0 | NaN | NaN |
| **K1** | A1 | B1 | NaN | NaN |
| **K2** | A2 | B2 | NaN | NaN |
| **K3** | A3 | B3 | NaN | NaN |
| **K0** | NaN | NaN | A4 | B4 |
| **K1** | NaN | NaN | A5 | B5 |
| **K2** | NaN | NaN | A6 | B6 |
| **K3** | NaN | NaN | A7 | B7 |
| **K4** | NaN | NaN | A8 | B8 |

As you can see, `df2` is attached to the end of `df1` .

**Merge**

"Pandas" allows you to merge DataFrames while paying attention to the values as well. For example, you might be working with data about the students of a university. Suppose you have two dataframes, different containing information about the students. However, if both these dataframes has a column called "Name," they will both contain the same names for the same students. We can use it to merge two dataframes by paying attention to the names of the students, and merging wherever the names match.

The syntax for merging DataFrames is

```
pd.merge(left, right, how='inner', on='Key')
```

The left signifies the DataFrame that should appear on the left side; right signifies the DataFrame that should appear on the right side.

Sometimes the data is not complete. You might have information about one student and no information about this student in the other dataframe. If you select `how='inner'` , the join throws out all the entries where it cannot find the names of the students. You can picture this as the central intersection of a Venn diagram. If you choose the outer, it only keeps all the information about the students.

Lastly, on='Key' signifies the key column on which the merge occurs. This key column has to be similar across all the DataFrames before the merge function can occur. In our example, this would be the "Name."

In [ ]:

```
df1 = pd.DataFrame({'key': ['A0', 'A1', 'A2', 'A3'],
                        'A': ['B0', 'B1', 'B2', 'B3']})
df2 = pd.DataFrame({'key': ['A0', 'A1', 'A2', 'A5', 'ZZ'],
                        'B': ['B4', 'B5', 'B6', 'B7', 'OO']})
print(df1)
print('\n')
print(df2)
```

```
  key   A
0  A0  B0
1  A1  B1
2  A2  B2
3  A3  B3


  key   B
0  A0  B4
1  A1  B5
2  A2  B6
3  A5  B7
4  ZZ  OO
```

**Outer Merge**

In [ ]:

```
pd.merge(df1, df2, how = 'inner', on = 'key')
```

Out[38]:

|   | key | A | B |
|---|---|---|---|
| **0** | A0 | B0 | B4 |
| **1** | A1 | B1 | B5 |
| **2** | A2 | B2 | B6 |

Notice how `df2` has a key "ZZ" that `df1` does not have. Since we merged it using `how='inner'` , pandas kept only the keys that we have on the left dataframe `df1` .

Let us see what happens when we put `how='outer'` .

**Inner Merge**

```
In [ ]:
```

```python
pd.merge(df1, df2, how = 'outer', on = 'key')
```

```
Out[39]:

   key    A    B
0   A0   B0   B4
1   A1   B1   B5
2   A2   B2   B6
3   A3   B3  NaN
4   A5  NaN   B7
5   ZZ  NaN   OO
```

Pandas has preserved all the information. Notice how there are NaN values when the value is not available.

## Concatenation

Concatenation glues DataFrames together. When concatenating DataFrames, you have to pay special attention to the dimensions. Concatenate function in pandas concatenates object along a particular axis. It performs all the set logic on other axis.

### Creating three DataFrame

```
In [ ]:
```

```python
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])
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])
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])

print(df1, '\n')
print(df2, '\n')
print(df3, '\n')
```

```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3

    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7

      A    B    C    D
8    A8   B8   C8   D8
9    A9   B9   C9   D9
10  A10  B10  C10  D10
11  A11  B11  C11  D11
```

### Concatinate along the rows

We can concatenate them using pd.concat() method:

```
In [ ]:
```

```
#passing dataframes as list
pd.concat([df1, df2, df3], axis=0)
```

```
Out[41]:
```

| | A | B | C | D |
|---|---|---|---|---|
| **0** | A0 | B0 | C0 | D0 |
| **1** | A1 | B1 | C1 | D1 |
| **2** | A2 | B2 | C2 | D2 |
| **3** | A3 | B3 | C3 | D3 |
| **4** | A4 | B4 | C4 | D4 |
| **5** | A5 | B5 | C5 | D5 |
| **6** | A6 | B6 | C6 | D6 |
| **7** | A7 | B7 | C7 | D7 |
| **8** | A8 | B8 | C8 | D8 |
| **9** | A9 | B9 | C9 | D9 |
| **10** | A10 | B10 | C10 | D10 |
| **11** | A11 | B11 | C11 | D11 |

Since we are concatenating these dataframes along the rows, and all three dataframes have matching columns. Therefore each row aligns with their respective columns. If one dataframe had some columns, let's say "Z" that the others didn't; it would create a new column for it. Since the other dataframes do not have any column named "Z," they perceive NaN as values.

### Concatenate along the columns

We can specify if we want to concatenate along the rows as well. We can do this by setting `axis=1`

In [ ]:

```
pd.concat([df1, df2, df3], axis=1)
```

Out[42]:

| | A | B | C | D | A | B | C | D | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | A0 | B0 | C0 | D0 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **1** | A1 | B1 | C1 | D1 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **2** | A2 | B2 | C2 | D2 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **3** | A3 | B3 | C3 | D3 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **4** | NaN | NaN | NaN | NaN | A4 | B4 | C4 | D4 | NaN | NaN | NaN | NaN |
| **5** | NaN | NaN | NaN | NaN | A5 | B5 | C5 | D5 | NaN | NaN | NaN | NaN |
| **6** | NaN | NaN | NaN | NaN | A6 | B6 | C6 | D6 | NaN | NaN | NaN | NaN |
| **7** | NaN | NaN | NaN | NaN | A7 | B7 | C7 | D7 | NaN | NaN | NaN | NaN |
| **8** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | A8 | B8 | C8 | D8 |
| **9** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | A9 | B9 | C9 | D9 |
| **10** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | A10 | B10 | C10 | D10 |
| **11** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | A11 | B11 | C11 | D11 |

Since we are concatenating along the columns, Pandas look for matching rows. However, if you look at the three datasets, none of the indices match. So Pandas "glues" them together, filling in NaN values wherever it does not have data.

## Apply functions

Let us say you have some data, and you want to apply a function on every item of the dataframe. We can apply it row-wise or column-wise, according to your requirements. It can also be a custom function that you make.

In [ ]:

```
import pandas as pd
dic = {'A':[1,2,3,4], 'B':[5,6,7,8]}
df = pd.DataFrame(dic)
df
```

Out[43]:

| | A | B |
|---|---|---|
| **0** | 1 | 5 |
| **1** | 2 | 6 |
| **2** | 3 | 7 |
| **3** | 4 | 8 |

Consider the dataframe above. Now we define a function that we want to apply to every member of the dataframe. Let us consider the function `sum()` .

In [ ]:

```
df.apply(sum, axis=0)
```

Out[44]:

```
A    10
B    26
dtype: int64
```

Notice that we have passed the function without the parentheses. We have received the sum of all the rows in A and B. This is because we put the parameter `axis=0` . If the axis is 1, we can perform these functions on the columns instead.

In [ ]:

```
df.apply(sum, axis=1)
```

Out[45]:

```
0     6
1     8
2    10
3    12
dtype: int64
```

## Map Functions

Pandas Map function maps values of Series according to input correspondence. `map` accepts `dict` and `Series` data structure

In [ ]:

```
import numpy as np
import pandas as pd
x = pd.Series(['Nepal', 'India', np.nan, 'China'])
```

In [ ]:

```
x.map({'Nepal': 'Kathmandu', 'India': 'New Delhi'})
```

Out[47]:

```
0    Kathmandu
1    New Delhi
2          NaN
3          NaN
dtype: object
```

`map` also accepts the function.

In [ ]:

```
x.map('The country is  {}'.format)
```

Out[48]:

```
0    The country is  Nepal
1    The country is  India
2      The country is  nan
3    The country is  China
dtype: object
```

## Takeaways:

- Series and dataframes are efficient ways to manipulate and organize the data.
- Row and column selection techniques can be implemented with loc and iloc in pandas dataframe.
- Join, Merge and Concat functions are used for combining multiple dataframes into single one.
- Apply and map funtions are used for various operations in the dataframe.