# **Pandas for data analysis**

Let's start with creating a DataFrame from a dictionary. A DataFrame is a tabular data structure, similar to an Excel spreadsheet. We can think of a dictionary as a set of key-value pairs, where each key has an associated value. In pandas, we can convert a dictionary to a DataFrame using the "DataFrame" function, which takes the dictionary as an argument.

In the following example, a dictionary is created with three keys: "name", "age", and "sex". Each key has a list as its value, and each list contains values ​​corresponding to each person in our data. We then convert this dictionary to a DataFrame using the pandas "DataFrame" function. Finally, we print the DataFrame to display it in the console.

| import pandas as pd dictionary = {'name': ['Anna', 'Pedro', 'Maria', 'Juan'],  'age': [25, 34, 18, 42],  'sex': ['F', 'M', 'F', 'M']} df = pd.DataFrame(dictionary) print(df) |
| --- |
| name age sex 0 Well25 F 1 Pedro 34 M 2 Maria18 F 3 Juan 42 M |

## Useful methods of a Pandas DataFrame

Now, let's move on to the useful methods of a Pandas DataFrame. Pandas provides a wide variety of methods for manipulating and analyzing data. Some common methods are:.

### .head(), .tail() y .sample()

They are Pandas methods that are used to view some rows. For example, if we have a DataFrame called*df*, and we want to see the first 2 rows

| df.head(2) |
| --- |
| name age sex 0 Well 25 F 1 Pedro 34 M |

If we want to see the last 2 rows

| df.tail(2) |
| --- |
| name Age sex 2 Maria 18 F 3 Juan 42 M |

And if we want to see 2 rows or random records

| df.sample(2) |
| --- |
| name age sex 3 Juan 42 M 1 Pedro 34 M |

### .info()

It's a Pandas method that gives us information about our DataFrame, such as the number of rows and columns, the data types of the columns, and whether there are any missing values. For example, if we want to get the information about a DataFrame df, we can use

| df.info() |
| --- |
| <class 'pandas.core.frame.DataFrame'> RangeIndex: 4 entries, 0 to 3 Data columns (total 3 columns):  # Column Non-Null Count Dtype  --- ------ -------------- -----   0 name4 non-null object  1 age4 non-null int64   2 sex4 non-null object dtypes: int64(1), object(2) memory usage: 224.0+ bytes |

### .describe()

It's a Pandas method that gives us descriptive statistics for the numeric columns of our DataFrame, such as the mean, median, standard deviation, and min and max values. For example, if we want to get descriptive statistics for the columns of a DataFrame df with numeric columns, we can use

| df.describe() |
| --- |
| count 4.000000 mean 29.750000 std 10.468206 min 18.000000 25% 23.250000 50% 29.500000 75% 36.000000 max 42.000000 |

### .value\_counts()

It's a Pandas method that allows us to count the frequency of unique values ​​in a given column. For example, if we have a DataFrame df with a "gender" column, we can use df['gender'].value\_counts() to get the frequency of unique values ​​in the "gender" column.

| df['sex'].value\_counts() |
| --- |
| F 2 M 2 Name: sexo, dtype: int64 |

### .pivot\_table()

It is a Pandas method that allows us to create a pivot table from our DataFrame, where we can group the data by one or more columns and calculate the statistics based on one or more numerical columns. For example, if we have a DataFrame df with columns "gender", "age", and "salary", we can use

| df.pivot\_table(index='sex', values=['age'], aggfunc='mean') |
| --- |
| age sex   F 21.5  M 38.0 |

to create a pivot table that displays the mean age and salary for each unique value in the "gender" column.

### Indexing and re-indexing

Another fundamental operation in the manipulation of DataFrames is the selection of data through indexing and re-indexing. Indexing refers to the selection of one or several columns of a DataFrame and re-indexing refers to the selection of rows or columns by label or position.

Indexing in Pandas is a way to select a subset of rows and columns from a DataFrame. Indexes are used to identify and access data in a DataFrame. In a DataFrame, the index can be numeric or alphanumeric and is used to label the rows of the table. By default, Pandas DataFrames have a numeric index starting at 0 and incrementing by 1 for each row. Indexing is done using the bracket notation []. To select a single column, you can refer to it by name, for example:

| df['name'] |
| --- |
| 0 Well 1 Pedro 2 Maria 3 Juan Name : name , dtype : object |

To select multiple columns, the column names must be passed as a list, for example:

| df[['name','age']] |
| --- |
| name age 0 Well 25 1 Pedro 34 2 Maria 18 3 Juan 42 |

Re-indexing, on the other hand, is done with the .loc[] method and is used to select rows or columns by position. For example:

| df.loc[1] |
| --- |
| name Peter age34 M sex Name: 1, dtype: object |

To select multiple rows or columns, the positions can be passed as a list, for example:

| df.loc[[2,0]] |
| --- |
| name age sex 2 Maria 18 F 0 Well 25 F |

Re-indexing can also be done using the .reindex() function. This function allows us to create a new DataFrame with a different index and fill in the missing values ​​with a specific value. For example:

| import pandas as pd  # Create a DataFrame data = {'name': ['Juan', 'Pedro', 'Luis', 'Maria'],  'age': [25, 30, 20, 27],  'city': ['Madrid', 'Barcelona', 'Valencia', 'Sevilla']} df = pd.DataFrame(data)  # We reindex the DataFrame new\_index = [3, 2, 1, 0] df\_reindexed = df.reindex(new\_index) |
| --- |

In this example, we have created a new index and reordered the DataFrame based on this index using the .reindex() function. The new DataFrame df\_reindexed will have the same values ​​as the original DataFrame df, but with the row order changed.

The .reindex() function also allows us to specify how we want to handle missing values ​​using the fill\_value parameter. For example:

| import pandas as pd  # Create a DataFrame data = {'name': ['Juan', 'Pedro', 'Luis', 'Maria'],  'age': [25, 30, 20, 27],  'city': ['Madrid', 'Barcelona', 'Valencia', 'Sevilla']} df = pd.DataFrame(data) # We reindex the DataFrame and fill the missing values ​​with zeros new\_index = [0, 1, 2, 3, 4] df\_reindexed = df.reindex(new\_index, fill\_value=0) |
| --- |

In this example, we have created a new index that includes an extra row (4) that does not exist in the original DataFrame df. By specifying fill\_value=0, we have filled the missing value with zeros.

Sequence containers are data types that can store multiple values ​​in a single variable, such as lists, tuples, and strings. Each value in a sequence container has an index, which is a numeric position starting at zero.

Sequence container indexing allows us to access a specific value at a certain position in the sequence. For example, to access the third element of a list called my\_list, we can use the following syntax:

| my\_list = [10, 20, 30, 40, 50]  print(my\_list[2]) |
| --- |
| > 30 |

In this case, my\_list[2] accesses the third element of the list my\_list (index 2), which is the value 30.

Indexing can also be negative, which means we start counting from the end of the sequence. For example, to access the last element of the list my\_list, we can use the following syntax:

| print(my\_list[-1]) |
| --- |
| > 50 |

Here, my\_list[-1] accesses the last element of the list my\_list, which is the value 50.

In Pandas, DataFrames are two-dimensional objects that allow us to work with structured data in the form of rows and columns. Reindexing of DataFrames allows us to change the row index and/or the name of the columns.

For example, consider the following DataFrame df:

| import pandas as pd  data = {'name': ['Juan', 'Maria', 'Pedro', 'Sofia'],  'age': [28, 25, 19, 32]} df = pd.DataFrame(data) print(df) |
| --- |
| name Age 0 Juan 28 1 Maria25 2 Pedro 19 3 Sofia32 |

Here, the row indices are the default integers from 0 to 3. If we want to change the row index to people's names, we can use Pandas' set\_index() method:

| df = df.set\_index('name') print(df) |
| --- |
| age name Juan 28 Maria25 Pedro 19 Sofia32 |

Here, we have reindexed the DataFrame df with the row index as the names of the people. We can also reindex the DataFrame with a new set of column names using the rename() method:

| df = df.rename(columns={'age': 'years'}) print(df) |
| --- |
| years name Juan 28 Maria25 Pedro 19 Sofia32 |

Here, we have renamed the 'age' column to 'years'. Reindexing is useful when we need to change the way data is accessed in a DataFrame

In addition to range indices, dataframes can also be indexed using tags. This is done using the .loc[] method, which allows you to select rows and columns based on row and column labels.

For example, if we have the following dataframe:

| A B C  2010-01-01 1 2 3  2010-01-02 4 5 6  2010-01-03 7 8 9 |
| --- |

We can select a row using its label as follows:

| df.loc['2010-01-02'] |
| --- |
| > A 4  B 5  C 6 Name: 2010-01-02, dtype: int64 |

We can also select a column using its label:

| df.loc[:, 'B'] |
| --- |
| > 2010-01-01 2  2010-01-02 5  2010-01-03 8 Name: B, dtype: int64 |

And finally, we can select a specific value using the row and column tag:

| df.loc['2010-01-02', 'B'] |
| --- |
| > 5 |

It is important to note that the row and column labels must be exactly the same as those that appear in the dataframe's index and columns. If we try to use a label that doesn't exist in the dataframe, we'll get an error.

In summary, dataframe indexing is a fundamental tool in data analysis. Through it we can select and manipulate the data that interests us, either based on its position in the dataframe or on specific labels. Knowledge of these techniques will allow us to carry out a more precise and efficient analysis of our data.

## Filtering data in a Pandas DataFrame

Now, let's talk about filtering data in a Pandas DataFrame. Filtering allows us to select a subset of rows or columns from a DataFrame based on certain conditions. It is one of the most common tasks in data analysis and helps us to find the values ​​that interest us. In Pandas, we can filter data using the bracket [] syntax and the logical operators & (and) and | (or).

For example:

| filter = (DataFrame['age'] > 21) & (DataFrame['sex'] == 'M') results = DataFrame[filter] print(results) |
| --- |
| > name age sex  1 Pedro 34 M  3 Juan 42 M |

This code will filter the data in the DataFrame df to find people over 21 and male, and return a new DataFrame with the results.

## Sorting values ​​in a Pandas DataFrame

Sorting the values ​​in a Pandas DataFrame is an important task in data analysis. Often, you need to sort data in ascending or descending order based on a specific value or column. Pandas offers the "sort\_values()" method to perform this task. This method sorts the values ​​based on the specified column. The "ascending" parameter is used to specify whether the order should be ascending or descending.

| df = pd.DataFrame({'Name':['Pedro', 'Juan', 'Lucy', 'Maria'],  'Age':[25, 31, 28, 22],   'Salary':[50000, 60000, 45000, 70000]})  print("DataFrame original:") print(df) |
| --- |
| > DataFrame original: > Name Age Salary  0 Pedro 25 50000   1 Juan 31 60000  2 Lucy28 45000  3 Maria22 70000 |
| df\_ordenado = df.sort\_values(by='Salary', ascending=False) print("DataFrame sorted by salary in descending order:") print(df\_sorted) |
| > DataFrame sorted by salary in descending order: > Name Age Salary  3 Maria22 70000  1 Juan 31 60000  0 Pedro 25 50000  2 Lucy28 45000 |

In this example, we create a DataFrame with three columns: "Name", "Age", and "Salary". We then use the "sort\_values()" method to sort the values ​​based on the "Salary" column in descending order. The result is a new DataFrame named "df\_ordered" that is displayed on the screen.

## Create columns in a DataFrame

Another important aspect of Pandas is the creation of new columns from existing ones. We can create a new column in our DataFrame df that is twice the age column, for example, using:

| df['age\_double'] = df[Age] \* 2 |
| --- |
| > Name Age Salary age\_double 0 Pedro 25 50000 50 1 Juan 31 60000 62 2 Lucy28 45000 56 3 Maria22 70000 44 |

In Pandas, we can also create new columns from lambda functions. Lambda functions are anonymous functions that are used to perform specific tasks. We can use these functions to create new columns in a Pandas DataFrame. For example:

| df = pd.DataFrame({'Name': ['Pedro', 'Juan', 'Lucy', 'Maria'],   'Age': [25, 31, 28, 22]}) df['Age Square'] = df['Age'].apply(lambda x: x\*\*2) print(df) |
| --- |
| Name Age Square of Age 0 Pedro 25 625 1 Juan 31 961 2 Lucy28 784 3 Maria22 484 |

In this example, we create a DataFrame with two columns: "Name" and "Age". We then use the "apply()" method to apply a lambda function to the "Age" column and create a new column called "Age Square". The lambda function takes a value "x" and returns the square of that value. The result is a new DataFrame with three columns

Another way to create columns is from conditional functions. For example, if we want to create a category column that has the values ​​'young' for ages less than 30 and 'old' for ages greater than or equal to 30, we can use:

| df['category'] = df['Age'].apply(lambda x: 'young' if x < 30 else 'old') print(df) |
| --- |
| Name Age Square of Age category 0 Pedro 25 625 young 1 Juan 31 961 old 2 Lucy28 784 young 3 Maria22 484 young |

In some cases, it is necessary to create new columns in a Pandas DataFrame that contain text strings. To do this, we can use bracket notation and assign a text string value to the new column. For example:

| import pandas as pd  df = pd.DataFrame({'Name': ['Pedro', 'Juan', 'Lucy', 'Maria'],  'Age': [25, 31, 28, 22]}) df['City'] = ['Madrid', 'Barcelona', 'Sevilla', 'Valencia'] print(df) |
| --- |
| Name Age City 0 Pedro 25 Madrid 1 Juan 31 Barcelona 2 Lucy28 Sevilla 3 Maria22 Valencia |

In this example, we create a DataFrame with two columns: "Name" and "Age". We then use bracket notation to create a new column called "City" and assign a list of text string values ​​to it. The result is a new DataFrame with three columns: "Name", "Age", and "City".

## Rename and sort columns of a Pandas DataFrame

Sometimes we may need to change the name of a column in a DataFrame. We can do this using the rename() method.

| import pandas as pd  sales = pd.DataFrame({  'Product': ['A', 'B', 'C', 'D'],  'Sales': [100, 200, 300, 400],  'Benefit': [50, 100, 150, 200] })  # Rename the 'Sales' column as 'Units Sold' sales = sales.rename(columns={'Sales': 'Units sold'}) print(sales) |
| --- |
| Product Units sold Profit 0 A 100 50 1 B 200 100 2 C 300 150 3 D 400 200 |

We can also change the order of the columns in a DataFrame using the reindex() method.

| # change the order of the columns sales = sales.reindex(columns=['Product', 'Units sold', 'Benefit']) print(sales) |
| --- |
| Product Units sold Profit 0 A 100 50 1 B 200 100 2 C 300 150 3 D 400 200 |

In this example, we create a DataFrame called sales with three columns: "Product", "Sales", and "Profit". Then, we rename the "Sales" column to "Units Sold" and change the order of the columns so that they are in the order "Product", "Units Sold" and "Profit".

## Using groupby in Pandas

In Pandas, it is also very common to use the groupby function, which allows us to group rows by values ​​of one or more columns and apply functions to those groupings. For example, if we want to group our DataFrame df by the gender column and calculate the mean of the age column for each group, we can use:

| df.groupby('gender')['age'].mean() |
| --- |

## DataFrames concatenation

Finally, we can join two DataFrames using the concat function. If we have two DataFrames df1 and df2 with the same columns and we want to join them by rows, we can use

| df\_concat = pd.concat([df1, df2]) |
| --- |

If we want to join them by columns, we can use:

| df\_concat = pd.concat([df1, df2], axis=1) |
| --- |

In conclusion, in this first topic we have learned the basics of using Pandas for data analysis in Python. We've seen how to create DataFrames from dictionaries, and how to manipulate the data using useful methods like loc, iloc, and at. We've also learned how to filter and sort data, create new columns from lambda and conditional functions, and rename and sort columns. Additionally, we have explored the use of groupby to group and summarize data, and the concatenation of DataFrames.

The use of Pandas is essential for any data analysis project, and these basic concepts that we have covered in this module are essential for any data analyst who wants to master the use of Pandas. In the next module, we'll continue our learning advancing, focusing on cleaning and preparing data for analysis, which is crucial for accurate and reliable data analysis results.