Pandas

`pandas` is a popular open-source data analysis and manipulation library for the Python programming language. It is built on top of the NumPy library and provides easy-to-use data structures and data analysis tools for efficient data manipulation and analysis.

In this tutorial, we will cover the basic functionality of the pandas package including data structures, data selection and manipulation, and data analysis tools.

Installation

First, you need to install the `pandas` package in your Python environment. You can install `pandas` using the following command in your terminal or command prompt:

```
pip install pandas
```

Importing the pandas Package

To import the `pandas` package in your current Python session, use the following command:

```python
import pandas as pd
```

Here, we have imported the `pandas` package and assigned it the alias `pd`. This is a common practice in the Python community. You can use any alias you want, but it is recommended to use the alias `pd` for the `pandas` package. This is the standard convention and will be used in remainder of this tutorial.

Data Structures

`pandas` provides two primary data structures: `Series` and `DataFrame`. A `Series` is a one-dimensional labeled array that can hold any data type, while a `DataFrame` is a two-dimensional labeled data structure with columns of potentially different data types.

Series

A `Series` can be created using the `pd.Series()` function. The following example creates a `Series` object from a list:

```python
import pandas as pd

data = [1, 2, 3, 4, 5]
s = pd.Series(data)

print(s)
```

In the above code, we first imported pandas and created a list `data` containing some numbers. Then, we created a Series `s` using the `pd.Series()` function and passed the data list to it. Finally, we printed the `Series` to the console.

DataFrame

A DataFrame can be created using the `pd.DataFrame()` function. The following example creates a DataFrame object from a dictionary:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)

print(df)
```

In the above code, we created a dictionary `data` containing three keys `"name"`, `"age"`, and `"country"` and their respective values. Then, we created a `DataFrame` df using the `pd.DataFrame()` function and passed the `data` dictionary to it. Finally, we printed the `DataFrame` to the console.

Data Selection and Manipulation

Pandas provides powerful tools for data selection a manipulation. In this section, we will cover some of the most commonly used tools. For a complete list of tools, refer to the official pandas documentation.

Selecting Columns

You can select one or more columns from a DataFrame by using square selection brackets [] with the column name(s). For example:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)
```

```python
# Selecting a single column
print(df["name"])
```

```python
# Selecting multiple columns
print(df[["name", "age"]])
```

Note the usage of double square brackets `[[...]]` to select multiple columns. This is because we are inputting a list of column names inside the selection brackets `[]`. Therefore, if you use a single square bracket to pass multiple column names `[...]`, you will get an error. Notice that the output of `df[...]` is a `Series` object, while the output of `df[[...]]` is a `DataFrame` object.

You can alternatively use a more object oriented method to select columns using dot (.) notation. For example:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)

# Selecting a single column
```

```python
print(df.name)
```

However, this method has caveats. For example, it can cause problems when column names contain spaces or special characters. For example, if you have a column named `"first name"`, you cannot use `.` notation to select it. You will have to use selection brackets `[]` instead. Also note that `.` notation is not designed for selecting multiple columns. Selecting columns through `.` notation is arguably less readable and powerful than using selection brackets `[]`. Therefore, it is recommended for most use cases to use selection brackets `[]` for selecting columns.

Filtering Rows

You can filter rows using conditional statements/boolean arrays within selection brackets `[]`. For example:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)
```

```python
# Creating a mask to filter rows
print(df["age"] > 19)
```

```python
# Filtering rows based on a condition
print(df[df["age"] > 19])
```

Selecting Rows and Columns

The `iloc[]` and `loc[]` operators can be used to select rows and columns simultaneously, making them very powerful tools for data selection and manipulation. The `iloc[]` operator is used to select rows and columns by their integer positions, while the `loc[]` operator is used to select rows and columns by their labels, or by a boolean array/conditional expression. The following example demonstrate the use of `iloc[]`:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)
```

```python
# Selecting a single row
print(df.iloc[0])
```

```python
# Selecting a single element
print(df.iloc[0, 1])
```

```python
# Selecting multiple rows
print(df.iloc[1:3])
```

```python
# Selecting multiple columns
print(df.iloc[:, 1:3])
```

Note that the `iloc[]` operator follows the same rules as indexing and slicing a Python object like a list or a NumPy array. However, the `loc[]` operator is arguably more powerful, as it allows you to access data through labels and booleans. This approach is considerably more intuitive and readable than the `iloc[]` operator. Allowed inputs can be a number, a string, a list, a slice object, or a boolean array/conditional expression. Therefore for most use cases, the `loc[]` operator is the preferred way of selecting rows and columns. The following example demonstrate the use of `loc[]`:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)
```

```
# Selecting a single row by an integer label
print(df.loc[0])
```

```
# Selecting a single element by an integer label and a string label
print(df.loc[0, "age"])
```

```
# Selecting multiple rows by a slice object
print(df.loc[1:2])
```

```
# Selecting multiple columns by a list of string labels
print(df.loc[:, ["age", "country"]])
```

```
# Selecting multiple rows by a conditional expression and multiple columns by a list of st
print(df.loc[df["age"] > 19, ["name", "age"]])
```

Modifying Data

The operations covered so far are not only useful for selecting data, but also for modifying data. For example:

```
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)

print(df)
```

```
# Modifying a single element using the loc operator
df.loc[0, "name"] = "Sam"

print(df)
```

```
# Modifying a single column using selection brackets [].
df["age"] = [25, 26, 24, 23]

print(df)
```

```
# Modifying a selection of rows in a single column using the loc operator and a conditiona
df.loc[df["age"] > 24, "country"] = "Germany"

print(df)
```

Adding Columns and Rows

You can add a new column to a DataFrame by assigning a value to a new column name. This behavior is similar to adding a new key to a dictionary. There are multitude ways to add a new column to a DataFrame. The following example demonstrates three different ways to add a new column to a DataFrame:

```
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)

print(df)
```

```
# Adding a new column
df["height"] = [180, 165, 175, 160]

print(df)
```

```
# Adding a new column using the loc operator
df.loc[:, "weight"] = [80, 55, 65, 50]

print(df)
```

```python
# Adding a new column through a calculation of existing columns
df["bmi"] = df["weight"] / (df["height"] / 100) ** 2

print(df)
```

Adding a new row to a DataFrame is a bit more complicated. Again, there are multiple ways to add a new row to a DataFrame. The following example demonstrates two different ways to add a new row to a DataFrame:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)

print(df)
```

```python
# Adding a new row using the loc operator
df.loc[4] = ["Bob", 22, "Germany"]

print(df)
```

```python
# Adding a new row using the concat method
df = pd.concat([df, pd.DataFrame([["Sam", 25, "Canada"]], columns=df.columns)])

print(df)
```

Deleting Data

You can delete data from a DataFrame using the `drop()` function. The `drop()` function takes two arguments: `labels` and `axis`. The `labels` argument specifies the row label(s) or column name(s) to be dropped, while the `axis` argument specifies the axis along which the labels are to be dropped. The `axis` argument can take two values: `0` for rows and `1` for columns. The following example demonstrates the use of the `drop()` function:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)

print(df)
```

```python
# Dropping a column
df = df.drop("age", axis=1)

print(df)
```

```python
# Dropping a row
df = df.drop(0, axis=0)

print(df)
```

`pandas` provides an alternative way of deleting data using the `drop()` function. You can use the `inplace` argument to delete data in-place. The `inplace` argument takes a boolean value. If `inplace=True`, the data will be deleted in-place, otherwise, a copy of the DataFrame will be returned which will have to be assigned to a variable. The following example demonstrates the use of the `inplace` argument:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice"],
    "age": [20, 21, 19, 18],
    "country": ["USA", "UK", "Canada", "France"],
}

df = pd.DataFrame(data)

print(df)
```

```python
# Dropping a column
df.drop("age", axis=1, inplace=True)

print(df)
```

```python
# Dropping a row
df.drop(0, axis=0, inplace=True)

print(df)
```

If you wish to delete multiple rows or columns, you can pass a list of labels to the `labels` argument.

Sorting Data

You can sort data in a DataFrame using the `sort_values()` function. The `sort_values()` function takes two arguments: `by` and `ascending`. The `by` argument specifies the column name(s) by which the data will be sorted, while the `ascending` argument specifies whether the data will be sorted in ascending or descending order. The `ascending` argument can take two values: `True` for ascending order and `False` for descending order. The following example demonstrates the use of the `sort_values()` function:

```python
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice", "Bob", "Sam"],
    "age": [20, 21, 19, 18, 22, 25],
    "country": ["USA", "UK", "Canada", "France", "Germany", "Canada"],
}

df = pd.DataFrame(data)

print(df)
```

```python
# Sorting by a single column
df = df.sort_values("age")

print(df)
```

```
# Sorting by multiple columns
df = df.sort_values(["country", "age"])

print(df)
```

```
# Sorting by multiple columns in descending order
df = df.sort_values(["country", "age"], ascending=False)

print(df)
```

```
# Sorting by multiple columns
df = df.sort_values(["country", "age"])

print(df)
```

```
# Sorting by multiple columns in descending order
df = df.sort_values(["country", "age"], ascending=False)

print(df)
```

```
# Sorting by a single column in-place
print(df.sort_values("age", inplace=True))
```

Grouping Data

You can group a DataFrame by one or more columns and perform aggregation operations on the groups using the `groupby()` function. The `groupby()` function takes one argument: `by`. The `by` argument specifies the column name(s) by which the data will be grouped. The following example demonstrates the use of the `groupby()` function:

```
import pandas as pd

data = {
    "name": ["John", "Charlotte", "David", "Alice", "Bob", "Sam"],
    "age": [20, 21, 19, 18, 22, 25],
    "country": ["USA", "UK", "Canada", "UK", "USA", "Canada"],
}

df = pd.DataFrame(data)
```

```python
# Grouping by country and calculating the number of rows in each group
print(df.groupby("country").size())
```

```python
# Grouping by country and calculating the mean age of each group
print(df.groupby("country")["age"].mean())
```

```python
# Grouping by country and counting the number of unique names in each group
print(df.groupby("country")["name"].nunique())
```

This feature is extremely powerful and can be used to perform a wide variety of data analysis tasks. The number of ways in which you can use the `groupby()` function is limited only by your imagination. For a complete list of aggregation functions, refer to the official pandas documentation.

Merging Data

The last feature we will cover in this tutorial is merging `DataFrames`. You can merge two `DataFrames` using the `merge()` function. The `merge()` function takes three arguments: `left`, `right`, and `on`. The `left` and `right` arguments specify the left and right DataFrames to be merged, while the `on` argument specifies the column name(s) on which the DataFrames will be merged. You can also use use the `left_on` and `right_on` arguments to specify the column names on which the DataFrames will be merged, if the column names are different in the two DataFrames. The following examples demonstrate the use of the `merge()` function:

```python
import pandas as pd

data1 = {
    "name": ["Alice", "Bob", "Charlie", "David"],
    "age": [25, 30, 35, 40],
    "country": ["USA", "Canada", "France", "UK"],
}

data2 = {
    "name": ["Alice", "Bob", "Charlie", "David"],
    "salary": [50000, 60000, 70000, 80000],
}

data3 = {
    "first_name": ["Alice", "Bob", "Charlie", "David"],
    "salary": [50000, 60000, 70000, 80000],
}

df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)
df3 = pd.DataFrame(data3)
```

```
# Merging the two DataFrames on the 'name' column
merged = pd.merge(df1, df2, on="name")

print(merged)
```

```
# Alternative syntax for merging the two DataFrames on the 'name' column - notice that the
merged = df1.merge(df2, on="name")

print(merged)
```

```
# Merging the two DataFrames where the column names are different.
merged = pd.merge(df1, df3, left_on="name", right_on="first_name")

print(merged)
```

An optional `how` argument can be used to specify the type of merge. The `how` argument can take four values: `"left"`, `"right"`, `"outer"`, and `"inner"`. The `"left"` value specifies a left merge, the `"right"` value specifies a right merge, the `"outer"` value specifies an outer merge, and the `"inner"` value specifies an inner merge.

When you first acquire a new dataset, it is important to explore the data to get a better understanding of the data. This process is often referred to as data exploration. Data exploration is an iterative process. You will often have to go back and forth between data exploration and data cleaning. This tutorial will cover some of the very basics, with more advanced techniques covered in future tutorials.

Exercises

This tutorial covered but a mere fraction of the features of the `pandas` library. The `pandas` library is extremely powerful and can be used to perform a wide variety of data analysis tasks. The following exercises will sometimes cover topics that were not covered in this tutorial, giving you a chance to learn new features of the `pandas` library. If you get stuck, you can refer to the official pandas documentation, or you can ask for help. Create your first DataFrame by running the following code:

```python
import pandas as pd
import numpy as np

# Generating random data for the first DataFrame
np.random.seed(123)
df1 = pd.DataFrame({
    'ID': np.arange(1000),
    'Age': np.random.randint(18, 65, 1000),
    'Country': np.random.choice(
        ['USA', 'Canada', 'France', 'UK'], 1000
    )
})
```

1. Display the first 10 rows of the DataFrame.
2. Display the number of rows and columns in the DataFrame.
3. Select the `Age` column from the DataFrame and display its values.
4. Select the `ID` and `Country` columns from the DataFrame and display their values.
5. Filter the DataFrame to only include rows where the age of the employee is less than or equal to 60.
6. Create another DataFrame using the code below and merge it to the original DataFrame. We will be using the merged DataFrame for the rest of the exercises.

   ```python
   np.random.seed(456)
   df2 = pd.DataFrame({
       'ID': np.arange(1000),
       'Salary': np.random.randint(50000, 100000, 1000),
       'Department': np.random.choice(
           ['Sales', 'Marketing', 'Engineering', 'Finance'], 1000
       )
   })
   ```

7. Create a new column called `Bonus` that is 10% of the `Salary` column for each employee.
8. Sort the DataFrame by department and salary in descending order.
9. Calculate the mean, median, and standard deviation of the salary for each department.
10. Give a 10% raise to all employees in the Sales department.
11. Set the salaries of all Canadian engineers to 100,000.

12. Create a new column in the DataFrame called `salary_range` that contains the salary range for each employee. The salary range should be one of the following: `low`, `medium`, or `high`. The salary range should be determined by the following rules:

    - If the salary is less than 60,000, the salary range is `low`.
    - If the salary is greater than or equal to 60,000 and less than 80,000, the salary range is `medium`.
    - If the salary is greater than or equal to 80,000, the salary range is `high`.

    Hint: You can use the `apply()` function to apply a function to each row or column of a DataFrame.

13. Create a pivot table with Department as the and Age as the columns, and the mean Salary as the values.

14. Explore the `pandas` documentation to see what other techniques you can use to manipulate the data. For example:

    - How can you create bins for the `Age` column?
    - How can we find the correlation between the `Age` and `Salary` columns?
    - How can we save the DataFrame to a CSV file?

    `pandas` is a very powerful library and can be used to perform a wide variety of data analysis tasks. The number of ways in which you can use the `pandas` library is limited only by your imagination. The more you explore the `pandas` library, the more you will learn about its capabilities.

Further Reading

Check out the following resources for more information on the `pandas` library:

- Official pandas documentation
- pandas Cheat Sheet
- pandas Cookbook

Tutorials:

- W3Schools
- DataCamp
- Real Python
- Python Data Science Handbook
- kaggle