

Fundamentals of Computer Programming

Lecture 4: Working with Pandas

Zied Bouyahya

Ecole Centrale de Lyon, Bachelor of Science in data science for responsible business



What is Pandas?

- Pandas is a Python library used for working with data sets.
- It provides functions for analyzing, cleaning, exploring, and manipulating data.
- Created by Wes McKinney in 2008, the name "Pandas" refers to both "Panel Data" and "Python Data Analysis."

Why Use Pandas?

- Allows analysis of big data and deriving conclusions based on statistical theories.
- Cleans messy data sets, making them readable and relevant.
- Essential for relevant data in data science.

What Can Pandas Do?

- Pandas provides answers about the data, such as:
 - Correlation between columns.
 - Average value.
 - Max and Min values.
- Pandas can clean data by deleting irrelevant rows or those with wrong values, like empty or NULL values.

Pandas Codebase

The source code for Pandas is available on GitHub at

<https://github.com/pandas-dev/pandas>.

Installation of Pandas

- If you have Python and PIP already installed on your system, installing Pandas is straightforward.
- Use the following command in the terminal:

```
pip install pandas
```

- If the command fails, consider using a Python distribution like Anaconda or Spyder, which already has Pandas installed.

Import Pandas

Once Pandas is installed, import it into your applications by adding the `import` keyword:

```
import pandas
```

Example: Importing Pandas and Creating a DataFrame

```
import pandas

mydataset = {
    'cars': ["BMW", "Volvo", "Ford"],
    'passings': [3, 7, 2]
}

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

Pandas Series

- What is a Series?
 - A Pandas Series similar to a column in a table.
 - It is a one-dimensional array holding data of any type.

- **Example: Creating a Simple Pandas Series**

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)
```

- **Labels**

- If nothing else is specified, the values are labeled with their index number (0, 1, 2, ...).
- This label can be used to access a specified value.

- **Example: Accessing Values by Index**

```
print(myvar[0])
```

Pandas Series (contd.)

- **Create Labels**

- With the `index` argument, you can name your own labels.

- **Example: Creating Series with Custom Labels**

```
a = [1, 7, 2]
myvar = pd.Series(a, index=["x", "y", "z"])
print(myvar)
```

- **Example: Accessing Values by Custom Label**

```
print(myvar["y"])
```

- **Key/Value Objects as Series**

- You can use a key/value object, like a dictionary, when creating a Series.

- **Example: Creating Series from Dictionary**

```
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

Pandas DataFrames

- **What is a DataFrame?**

- A Pandas DataFrame is a 2-dimensional data structure, like a 2-dimensional array, or a table with rows and columns.

- **Example: Creating a Simple Pandas DataFrame**

```
import pandas as pd
data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}
# load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
```

- **Locate Row**

- Pandas uses the `loc` attribute to return one or more specified row(s).

- **Example: Returning Rows**

```
# refer to the row index:
print(df.loc[0])
# use a list of indexes:
print(df.loc[[0, 1]])
```


Pandas DataFrames (contd.)

- **Named Indexes**

- With the `index` argument, you can name your own indexes.

- **Example: Adding Named Indexes**

```
data = {  
    "calories": [420, 380, 390],  
    "duration": [50, 40, 45]  
}  
  
df = pd.DataFrame(data, index=["day1", "day2", "day3"])  
print(df)
```

Locate Named Indexes

Use the named index in the `loc` attribute to return the specified row(s).

Example: Returning Rows by Named Index

```
# refer to the named index:  
print(df.loc["day2"])
```

Load Files Into a DataFrame

- If your data sets are stored in a file, Pandas can load them into a DataFrame.

Example: Loading a CSV File

```
import pandas as pd  
df = pd.read_csv('data.csv')  
print(df)
```

Reading SQL Data into Pandas DataFrames

- Pandas provides powerful tools for data analysis in Python.
- Often, data resides in SQL databases. Pandas facilitates reading SQL data into DataFrames for analysis.
- In this example, we'll focus on reading data from a MySQL database.

Prerequisites

- Ensure you have Pandas and SQLAlchemy installed:

```
pip install pandas sqlalchemy
```

- MySQL Connector may also be needed:

```
pip install mysql-connector-python
```

Creating a Database Connection

- Use the SQLAlchemy library to create a connection to the MySQL database:

```
from sqlalchemy import create_engine
```

```
# Replace 'user', 'password', 'server', and 'database' with your details
```

```
engine = create_engine('mysql://user:password@server/database')
```

Reading SQL Data into Pandas DataFrame

- Use Pandas' `read_sql()` function to execute a SQL query and load the result into a DataFrame:

```
import pandas as pd

# Your SQL query
query = 'SELECT * FROM your_table;'

# Read data into a Pandas DataFrame
df = pd.read_sql(query, engine)

# Display the DataFrame
print(df)
```

Pandas - Cleaning Empty Cells

- **Empty Cells**

- Empty cells can potentially give you a wrong result when you analyze data.

- **Remove Rows**

- One way to deal with empty cells is to remove rows that contain empty cells.
- This is usually OK since data sets can be very big, and removing a few rows will not have a big impact on the result.

- **Example**

- Return a new Data Frame with no empty cells:

```
import pandas as pd

df = pd.read_csv('data.csv')
new_df = df.dropna()
print(new_df.to_string())
```

Pandas - Cleaning Empty Cells (contd.)

- **Remove Rows (contd.)**

- Note: By default, the `dropna()` method returns a new DataFrame and will not change the original.
- If you want to change the original DataFrame, use the `inplace = True` argument.

- **Example**

- Remove all rows with NULL values:

```
import pandas as pd

df = pd.read_csv('data.csv')
df.dropna(inplace=True)
print(df.to_string())
```

- **Replace Empty Values**

- Another way of dealing with empty cells is to insert a new value instead.
- The `fillna()` method allows us to replace empty cells with a value.

Pandas - Cleaning Empty Cells (contd.)

- **Example**

- Replace NULL values with the number 130:

```
import pandas as pd

df = pd.read_csv('data.csv')
df.fillna(130, inplace=True)
```

- **Replace Only For Specified Columns**

- The example above replaces all empty cells in the whole Data Frame.
- To only replace empty values for one column, specify the column name for the DataFrame.

- **Example**

- Replace NULL values in the "Calories" column with the number 130:

```
import pandas as pd

df = pd.read_csv('data.csv')
df["Calories"].fillna(130, inplace=True)
```

Pandas - Cleaning Empty Cells (contd.)

- **Replace Using Mean, Median, or Mode**

- A common way to replace empty cells is to calculate the mean, median, or mode value of the column.
- Pandas uses the `mean()`, `median()`, and `mode()` methods to calculate the respective values for a specified column.

- **Example**

- Calculate the MEAN and replace any empty values with it:

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].mean()
df["Calories"].fillna(x, inplace=True)
```

Pandas - Cleaning Empty Cells (contd.)

Example

- Calculate the MEDIAN and replace any empty values with it:

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].median()
df["Calories"].fillna(x, inplace=True)
```

Example

- Calculate the MODE and replace any empty values with it:

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].mode()[0]
df["Calories"].fillna(x, inplace=True)
```

Pandas - Cleaning Data of Wrong Format

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	NaN
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	'20201226'	100	120	250.0
27	60	'2020/12/27'	92	118	241.0

Pandas - Cleaning Data of Wrong Format

- **Data of Wrong Format**

- Cells with data of the wrong format can make it difficult, or even impossible, to analyze data.

- **To fix it, you have two options:**

1. Remove the rows.
2. Convert all cells in the columns into the same format.

- **Convert Into a Correct Format**

- In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date.

- **Example**

```
import pandas as pd
df = pd.read_csv('data.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

Pandas - Cleaning Data of Wrong Format (contd.)

- Result

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
...					
26	60	'2020/12/26'	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	NaN
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

- Removing Rows

- The result from the converting in the example above gave us a NaN value, which can be handled as a NULL value, and we can remove the row by using the `dropna()` method.

- Example

```
df.dropna(subset=['Date'], inplace=True)
```

Pandas - Fixing Wrong Data

- **Wrong Data**

- "Wrong data" does not have to be "empty cells" or "wrong format"; it can just be wrong, like if someone registered "199" instead of "1.99".
- Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.
- If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows, the duration is between 30 and 60.

- **How can we fix wrong values, like the one for "Duration" in row 7?**

- **Replacing Values**

- One way to fix wrong values is to replace them with something else.
- In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:

- **Example**

```
df.loc[7, 'Duration'] = 45
```

Option 2: Removing rows

- Another way of handling wrong data is to remove the rows that contain wrong data.
- This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.

Example

```
for x in df.index:  
    if df.loc[x, "Duration"] > 120:  
        df.drop(x, inplace = True)
```


Introduction to data analysis

Exploratory data analysis and visualisation

Lecture 5: Advanced Data Manipulation with Pandas

Zied Bouyahya

Ecole Centrale de Lyon, Bachelor of Science in data science for responsible business



- Working with DataFrames
- DataFrames - Working with Columns
- Conditional Selection in DataFrames
- Grouping and Aggregation in Pandas
- Merging and Joining DataFrames
- Pivot Tables in Pandas
- Time Series Analysis with Pandas

DataFrames

Working with Columns

Key Concept

A DataFrame's columns are instances of the Series object.

To interact with columns in a DataFrame, you can perform various operations, such as retrieving the column index and labels.

- `df.columns` returns the index object representing the column labels.
- `df.columns[0]` retrieves the label of the first column.
- `df.columns.tolist()` converts all column labels into a Python list.

DataFrames

Basic Operations on Columns

Remember A DataFrame column is a Series object.

1. Selecting Columns:

- Retrieve a single column: `df['column_name']` or `df.column_name`
- Retrieve multiple columns: `df[['col1', 'col2']]`

2. Creating New Columns:

- Perform element-wise operations.

```
df['new_column'] = df['col1'] + df['col2']
```

3. Renaming Columns:

- Rename one or more columns.

```
df.rename(columns={'old_name': 'new_name'}, inplace=True)
```

DataFrames

Additional Operations on Columns

4. Dropping Columns:

- Remove columns from the DataFrame.

```
df.drop(columns=['col1', 'col2'], inplace=True)
```

5. Sorting Columns:

- Sort columns alphabetically or based on custom criteria.

```
df = df.reindex(sorted(df.columns), axis=1)
```

6. Handling Missing Values:

- Fill or drop NaN values in specific columns.

```
df['col1'].fillna(value, inplace=True)
```

DataFrames

Changing Data Types

7. Changing Data Types:

- Convert the data type of a column.

```
df['numeric_col'] = pd.to_numeric(df['numeric_col'], errors='coerce')
```

- `pd.to_numeric()` is used to convert a column to a numeric type.
- `errors='coerce'` replaces non-convertible values with NaN.

8. Conditional Operations:

- Perform operations based on conditions.

```
df['high_value'] = np.where(df['value'] > threshold, 'High', 'Low')
```

- Use `np.where()` to assign values based on a condition.
- In this example, a new column 'high_value' is created with 'High' if the condition is met, 'Low' otherwise.

DataFrames

Conditional Selection

In Pandas, conditional selection allows you to filter and select specific rows or columns based on certain conditions.

Conditional Selection Syntax

Example: Selecting rows where 'column_name' meets a condition

```
df[df['column_name'] > threshold]
```

- The syntax involves creating a boolean mask by applying a condition to a specific column.
- The resulting boolean mask is then used to select the rows that satisfy the condition.

Example

Selecting rows where 'Sales' is greater than 1000

```
expensive_bookings = booking_df[booking_df['price'] > 500]
```

Working with DataFrames

Grouping and Aggregation in Pandas

- In data analysis with Pandas, grouping and aggregation are essential operations.
- Grouping allows us to split the data into subsets based on a criterion.
- Aggregation involves applying functions like mean, sum, etc., to obtain summary statistics for each group.

Working with DataFrames

Grouping Data

Grouping Data Syntax

```
grouped_data = df.groupby('category')
```

- Use the `groupby()` function to group data based on a specific column or criterion.
- The result is a `DataFrameGroupBy` object that can be used for further operations.

Example

```
booking_passenger = booking_df.groupby('passenger_id')
```

Working with DataFrames

Aggregating Data

Aggregating Data Syntax

```
mean_values = grouped_data.mean()  
total_values = grouped_data.sum()  
mean_values = grouped_data.mean()
```

- After grouping, use aggregation functions like `mean()`, `sum()`, etc.
- These functions provide summary statistics for each group in the grouped data.

Example

```
avg_per_customer = booking_df.groupby('passenger_id')['price'].mean().reset_index()
```

Working with DataFrames

Multiple Aggregations

Multiple Aggregations Syntax

```
result = grouped_data.agg({'column1': 'mean', 'column2': 'sum'})
```

- Perform multiple aggregations simultaneously using the `agg()` function.
- Specify the columns and corresponding aggregation functions in a dictionary.

Example

```
price_based_data = booking_df.groupby('passenger_id')
agg_price = price_based_data.agg({'price': ['sum', 'mean'],
                                  'booking_id': 'count'},).reset_index()
```

Working with DataFrames

Custom Aggregation Functions

Custom Aggregation Function Syntax

```
# Example: Applying a custom aggregation function
def custom_function(x):
    return x.max() - x.min()

result = grouped_data.agg(custom_function)
```

- Define custom aggregation functions and apply them to groups using `agg()`.
- Custom functions should take a Series as input and return a scalar.

Pandas: reset_index()

Understanding the reset_index() Function

- In Pandas, the `reset_index()` function is used to reset the index of a DataFrame.
- By default, when you perform operations like grouping, the group-by columns become the new index.
- The `reset_index()` function moves the index back to regular columns and assigns default integer indices.

Example

Assuming 'df' is the DataFrame

```
total_amount_per_customer = df.groupby('passenger_id')['price'].sum().reset_index()
```

Merging and Joining DataFrames

- Merging and joining are operations used to combine data from multiple DataFrames.
- These operations are crucial when dealing with datasets distributed across different tables or files.
- In Pandas, the functions `merge()` and `join()` facilitate these operations.

Merging and Joining DataFrames

Combining Data

Combining Data Syntax (merge)

Example: Merging based on a common column

```
merged_data = pd.merge(df1, df2, on='common_column')
```

- Use the `merge()` function to combine DataFrames based on a common column.
- The result is a new DataFrame containing columns from both input DataFrames.

Merging and Joining DataFrames

Types of Joins

- Different types of joins determine how rows are combined when merging DataFrames.
- Common types include inner, outer, left, and right joins.
- The choice of join type depends on the desired outcome and the structure of the data.

Merging and Joining DataFrames

Inner Join

Inner Join Syntax

Example: Performing an inner join

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

- Inner join returns only the rows where there is a match in both DataFrames.
- Use the `how='inner'` parameter in `merge()` to perform an inner join.

Merging and Joining DataFrames

Outer Join

Outer Join Syntax

Example: Performing an outer join

```
outer_joined = pd.merge(df1, df2, on='common_column', how='outer')
```

- Outer join returns all rows from both DataFrames, filling in missing values with NaN where there is no match.
- Use the `how='outer'` parameter in `merge()` to perform an outer join.

Pivot Tables in Pandas

- Pivot tables are powerful tools in Pandas for reshaping and summarizing data.
- They allow you to aggregate and analyze data based on one or more columns.
- In Pandas, the `pivot_table()` function is used to create pivot tables.

Pivot Tables in Pandas

Creating Pivot Tables

Creating Pivot Table Syntax

Example: Creating a pivot table

```
pivot_table = df.pivot_table(values='Value',  
                               index='Category',  
                               columns='Month',  
                               aggfunc='sum')
```

- Use the `pivot_table()` function to create a pivot table in Pandas.
- Specify the values, index, columns, and aggregation function (if needed).

Pivot Tables in Pandas

Customizing Pivot Tables

- Pivot tables can be customized to meet specific analysis requirements.
- Customize the structure, aggregation functions, and appearance of the pivot table.
- Use the parameters in the `pivot_table()` function to tailor the results.

Example

```
data = {'Category': ['A', 'B', 'A', 'B', 'A', 'B'],  
        'Value': [10, 15, 20, 25, 30, 35]}  
df = pd.DataFrame(data)
```

```
# Create a pivot table
```

```
pivot_table = df.pivot_table(index='Category', values='Value', aggfunc='sum')  
print(pivot_table)
```

```
>>> Output:
```

Category	Value
A	60
B	75

Pivot Tables Examples

Example 1: Total Prices per Airline

Creating a Pivot Table: Total Prices per Airline

Merging booking_df with flight_df to get the airline information

```
merged_df = pd.merge(booking_df, flight_df, on='flight_id', how='inner')
```

Now merging the result with airline_df to get additional airline details

```
final_merged_df = pd.merge(merged_df, airline_df, on='airline_id', how='inner')
```

Create a pivot table to calculate total prices per airline

```
pivot_table_airline_prices = final_merged_df.pivot_table(  
    values='price',  
    index='airlinename',  
    aggfunc='sum'  
)
```

Display the pivot table

```
print(pivot_table_airline_prices)
```

Pivot Tables Examples

Example 2: Average Prices per Flight

Creating a Pivot Table: Average Prices per Flight

```
# Merging booking_df with flight_df to get flight information
merged_df = pd.merge(booking_df, flight_df, on='flight_id', how='inner')

# Create a pivot table to calculate average prices per flight
pivot_table_average_prices = merged_df.pivot_table(
    values='price',
    index='flight_id',
    aggfunc='mean'
)

# Display the pivot table
print(pivot_table_average_prices)
```

Pivot Tables Examples

Example 3: Number of Passengers per Airline and Flight

Creating a Pivot Table: Number of Passengers per Airline and Flight

```
# Merging booking_df with flight_df to get airline information
```

```
merged_df = pd.merge(booking_df, flight_df, on='flight_id', how='inner')
```

```
# Merging the result with airline_df to get the airline name
```

```
merged_df = pd.merge(merged_df, airline_df, on='airline_id', how='left')
```

```
# Creating a pivot table for the number of passengers per airline and flight
```

```
pivot_table_passengers_per_airline_flight = merged_df.pivot_table(  
    values='passenger_id',  
    index=['airlinename', 'flightno'],  
    aggfunc='count'  
)
```

```
# Display the pivot table
```

```
print(pivot_table_passengers_per_airline_flight)
```


Time Series Analysis with Pandas

Introduction

- Time series data often involves date and time information.
- Pandas provides powerful tools for working with dates and times.
- In this module, we'll explore parsing, formatting, and performing time-related operations on time series data.

Time Series Analysis with Pandas

Parsing and Formatting Dates

Assuming you have a DataFrame named `flights`:

Parsing Dates

```
flight_df['departure'] = pd.to_datetime(flight_df['departure'])
```

Formatting Dates

```
flight_df['departure_formatted'] = \
flight_df['departure'].dt.strftime('%Y-%m-%d %H:%M')
```

Time Series Analysis with Pandas

Time-related Operations

Assuming you have a DataFrame named `flights`:

Time-related Operations

```
flights['hour'] = flights['departure'].dt.hour  
flights['day_of_week'] = flights['departure'].dt.day_name()
```

- Extracting the hour of the day and the day of the week from the departure time.

Time Series Analysis with Pandas

Resampling and Rolling Windows

- Resampling and rolling windows are essential techniques in time series analysis.
- Resampling allows us to change the frequency of the time series data.
- Rolling windows enable the calculation of moving averages and other time-dependent statistics.

Time Series Analysis with Pandas

Resampling Time Series Data

Assuming you have a DataFrame named `flights`:

Resampling Time Series Data

```
daily_flights = flights.resample('D', on='departure').count()
```

- Resampling the flight data to get daily counts.

Time Series Analysis with Pandas

Rolling Windows

Assuming you have a DataFrame named `flights`:

Rolling Windows

```
flights['rolling_mean'] = flights['passenger_count'].rolling(w
```

- Calculating a 7-day rolling mean of passenger counts.

Time Series Analysis with Pandas

More Advanced Time Series Analysis

- Time series analysis in Pandas goes beyond these basics.
- Explore additional functionalities such as seasonality analysis, decomposition, and forecasting.
- Utilize Pandas' capabilities to gain deeper insights into time-dependent data patterns.

Time Series Analysis with Pandas

Wrap up

- Working with dates and times is a fundamental aspect of time series analysis.
- Pandas provides powerful tools for parsing, formatting, and performing time-related operations.
- Resampling and rolling windows enhance the analysis by adjusting the frequency and capturing trends.