03_python_pandas

May 9, 2025

1 Python for Actuaries Part 3

1.1 Agenda

In this notebook, we will cover: - Introduction to Pandas - Series and DataFrames - Data access and inspection - Data manipulation and cleaning - Reading files

2 Introduction to Pandas

Pandas is one of the most popular libraries for data analysis and manipulation in Python. It was specifically designed to work efficiently and easily with structured data, similar to how spreadsheets like Excel operate.

Pandas provides two main structures for handling data: - **Series**: A one-dimensional data structure, similar to a list or an array. - **DataFrame**: A two-dimensional data structure, comparable to a table in a database or an Excel spreadsheet.

2.0.1 Why Pandas?

Pandas is particularly useful when it comes to reading, processing, and analyzing large amounts of data. Typical tasks that Pandas simplifies include: - Reading data from various formats (e.g., CSV, Excel, SQL databases). - Efficient inspection and analysis of data. - Data cleaning and manipulation (e.g., handling missing values, renaming columns). - Transformation and aggregation of data for further analysis or visualizations.

Due to its broad functionality and ease of use, Pandas has become a standard tool for data analysis, especially in the fields of data science, financial analysis, and insurance.

In the following sections, we will look at the key features of Pandas and learn how to efficiently read, analyze, and clean data.

2.0.2 Criticisms of Pandas

Although Pandas is a powerful library, there are also some criticisms to keep in mind:

1. **Memory and computational overhead**: Pandas is memory-intensive as it loads data into memory. With very large datasets, this can lead to performance issues. For large datasets that do not fit into RAM, alternative tools such as **Dask**, **PySpark**, or **Apache Arrow** should be considered, which are optimized for distributed processing.

2. **Complexity with large data pipelines**: For smaller projects and analyses, Pandas is very efficient. However, with larger data pipelines or complex data analyses, the code can become unwieldy and hard to maintain. In particular, dealing with many nested Pandas functions can make the code difficult to understand.

Despite these limitations, Pandas remains a powerful tool for most data analyses, especially with medium-sized datasets.

[ ]:
```
# To work with Pandas, it obviously needs to be imported
import pandas as pd
# Typically, NumPy is also needed
import numpy as np
```

2.1 Series

A **Pandas Series** is a one-dimensional data structure that is comparable to a list or an array. It consists of a series of values, each associated with an index. Pandas Series can contain various data types such as integers, floating-point numbers, or strings.

2.1.1 Properties of a Series:

- **Index**: Each Series has an associated index, which by default starts at 0 and increments, but can also be explicitly set (e.g., dates).
- **Homogeneous Data**: All elements of a Series have the same data type (like in arrays).

Series are useful when working with one-dimensional data, such as a list of damage amounts in an insurance portfolio.

2.2 Example 1: Damage History Over Several Years

In this example, we will store the annual damage history of an insurance company as a Pandas Series.

[ ]:
```
# Claims history over five years
years = ['2018', '2019', '2020', '2021', '2022']
claims_history = pd.Series([10000, 15000, 13000, 12000, 11000], index=years)
print(claims_history)
```

Another example is (synthetic) stock trends:

[ ]:
```
# Parameters for the simulation
np.random.seed(42)
days = pd.date_range(start='2023-01-01', periods=365, freq='D')
starting_price = 100  # Initial stock price
volatility = 0.02  # Stock volatility (2%)
expected_return = 0.001  # Expected daily return (0.1%)

# Generate daily changes based on Brownian motion
changes = np.random.normal(loc=expected_return, scale=volatility,␣
 ↪size=len(days))
```

```
# Calculate stock prices using cumulative sum
prices = starting_price * np.exp(np.cumsum(changes))

# Create the Pandas Series
stock_series = pd.Series(prices, index=days)

# Display the first few days of the stock price series
print(stock_series.head(20))
```

2.3 DataFrames

A **DataFrame** is the core of Pandas and represents a two-dimensional, table-like data structure that contains columns and rows. One can think of a DataFrame as a collection of Pandas Series that share a common structure. Each column is assigned a unique name, and each row can be referenced by an index.

2.3.1 Properties of a DataFrame:

- **Rows and Columns**: A DataFrame consists of rows and columns, with each column potentially having its own data type (e.g., one column with numeric values and another column with strings).
- **Flexible Indexing**: You can access both rows and columns using the index or column names.
- **Data Sources**: DataFrames can be created from various data sources, such as CSV files, Excel files, SQL databases, or even from dictionaries and lists in Python.

2.4 Example 1: Creating a DataFrame for Insurance Data

In this example, we will create a DataFrame that contains some basic information about insurance policies, such as the policy ID, the insured person, the premium amount, and the number of claims.

[ ]:
```
# Creating a DataFrame
# Define the data
data = {
    'Policy_ID': pd.Series([101, 102, 103, 104, 105, 106, 107, 108, 109, 110],␣
 ↪dtype='int64'),
    'Insured_Person': pd.Series(['John Smith', 'Emma Johnson', 'Michael Brown',␣
 ↪'Olivia Davis', 'William Miller',
                                 'Sophia Wilson', 'James Moore', 'Isabella Taylor',␣
 ↪'Lucas Anderson', 'Mia Thomas'], dtype='string'),
    'Age': pd.Series([18, 47, 49, 14, 15, 31, 1, 54, 3, 16], dtype='int64'),
    'Premium_Amount': pd.Series([500, 600, 550, 620, 480, 700, 520, 610, 580,␣
 ↪540], dtype='float64'),
    'Claims': pd.Series([2, 1, 0, 3, 2, 1, 1, 0, 4, 2], dtype='int64'),
    'Policy_Start': pd.to_datetime(['2019-01-15', '2020-05-20', '2021-03-10',␣
 ↪'2018-11-22',
                                    '2019-07-01', '2020-02-17', '2021-09-09',␣
 ↪'2022-01-01',
```

```
                                    '2019-12-25', '2020-06-14'])
}

# Create the DataFrame
insurance_df = pd.DataFrame(data)

# Display the DataFrame
insurance_df
```

Attention, the DataFrame does not have an explicit index. Therefore, the column ranges from 0 to 9 at the front.

[ ]:
```
# Ausgabe der Spaltentypen
insurance_df.dtypes
```

Here's the translation:

Another example of time-dependent (synthetic) data:

[ ]:
```
# Parameters for the simulation
np.random.seed(42)
days = pd.date_range(start='2023-01-01', periods=365, freq='D')

# Stock parameters for different companies
companies = {
    'Apple': {'start_price': 150, 'volatility': 0.02, 'return': 0.001},
    'Nvidia': {'start_price': 220, 'volatility': 0.025, 'return': 0.0012},
    'Google': {'start_price': 180, 'volatility': 0.018, 'return': 0.0008},
    'SAP': {'start_price': 120, 'volatility': 0.015, 'return': 0.0009},
    'Microsoft': {'start_price': 250, 'volatility': 0.02, 'return': 0.0011},
    'Tesla': {'start_price': 200, 'volatility': 0.03, 'return': 0.0015},
}

# Create a DataFrame to store all stock trajectories
stock_prices = pd.DataFrame(index=days)

# Simulate stock prices for each company
for company, params in companies.items():
    start_price = params['start_price']
    volatility = params['volatility']
    expected_return = params['return']

    # Generate daily changes based on Brownian motion
    changes = np.random.normal(loc=expected_return, scale=volatility,␣
 ↪size=len(days))

    # Calculate stock prices with cumulative sum
    prices = start_price * np.exp(np.cumsum(changes))
```

```
    # Add the simulation as a series in the DataFrame
    stock_prices[company] = prices

# Output the first 20 days of the simulated stock prices
stock_prices.head(20)
```

Now we have two nice example datasets `insurance_df` and `stock_prices`. Let's see what we can do with them.

3 Pandas Standard Functions

Pandas offers a range of helpful standard functions to quickly and efficiently inspect data. Here are some of the most important ones: `head()` and `tail()`

With `head()` and `tail()`, you can display the first or last entries of a DataFrame or a Series. This is useful for getting a quick overview of the structure of the data. We have already seen examples (from `head()`) above.

More interesting for getting a first impression of a dataset are the functions `describe()` and `info()`.

[ ]:
```
insurance_df.info()
```

[ ]:
```
stock_prices.describe()
```

Other important information includes, for example, `columns`, `index`, or `shape`. (Note that these are not functions but attributes of the DataFrame.)

[ ]:
```
stock_prices.columns
```

[ ]:
```
insurance_df.shape
```

[ ]:
```
stock_prices.index
```

3.1 Data Access in Pandas

In Pandas, there are various methods to access data in a DataFrame. This includes accessing columns, rows, or specific elements using `loc` and `iloc`.

3.1.1 Accessing Columns

A column of a DataFrame can be easily accessed using the column name. There are two ways to do this:

1. Access using dot notation

[ ]:
```
insurance_df.Insured_Person
```

2. With column names in square brackets (more common)

[ ]:
```
insurance_df['Insured_Person']
```

3.2 Accessing Rows

`loc[]` – Access by Labels (Index-based)

With 'loc[]', you can access rows and columns based on the labels. This method uses explicit index names or column names.

[ ]:
```
stock_prices.loc['01-01-2023'] #Watch out for the date format
```

`iloc[]` - Integer-based Access

With `iloc[]`, you access rows and columns based on their position in the DataFrame. This works similarly to a list in Python, where the indices are 0-based.

[ ]:
```
stock_prices.iloc[0]
```

3.3 Accessing Columns and Rows + Slicing

You can also access columns and rows simultaneously, plus select only parts of the data (*slicing*).

[ ]:
```
# Only the first 5 days of the stock prices for Apple and Nvidia
stock_prices.loc['2023-01-01':'2023-01-05', ['Apple', 'Nvidia']]
```

[ ]:
```
# Accessing a specific value in the DataFrame
insurance_df.iloc[2,1]
```

[ ]:
```
stock_prices.loc['2023-12-31', 'Microsoft']
```

4 Conditional Access

In addition to accessing data via column names or indices, you can also access data based on conditions.

[ ]:
```
# all contracts starting in May
insurance_df[insurance_df['Policy_Start'].dt.month == 5]
```

[ ]:
```
# only contracts with a premium amount greater than 600
insurance_df[insurance_df['Premium_Amount'] > 600]
```

[ ]:
```
# access is given by a boolean mask
insurance_df['Premium_Amount'] > 600
```

The conditional access is very powerful; however, it can also become very complex very quickly:

[ ]:
```
stock_prices[
    (stock_prices.index.month == 5) &  # Filter for May
    ((stock_prices['Nvidia'] > 230) |   # Nvidia > 230 or
    (stock_prices['Apple'] > 140)) &    # Apple > 155
    (stock_prices['Google'] > 185) &   # Google > 185
    (stock_prices['Tesla'] < 260)      # Tesla < 220
```

```
]
```

4.1 Enriching Data

In data analysis, it is often necessary to expand existing datasets by adding new columns. This can be done, for example, through calculations, importing additional data, or applying functions. In Pandas, there is a straightforward way to create new columns and fill them with data.

Typical use cases include:

- Calculating new values based on existing columns (e.g., risk assessments, premium calculations)
- Adding external data sources (e.g., inflation, exchange rates)
- Creating categories or groupings (e.g., age divisions, damage classes)

New columns can be created in various ways, such as through simple assignment, using calculations, or assigning a constant or calculated series of values.

In the following examples, we will look at how to add new columns to an existing DataFrame.

[ ]:
```
# add a column with the contract duration in years
insurance_df['Duration'] = (pd.to_datetime('today') -␣
 ↪insurance_df['Policy_Start']).dt.days / 365
insurance_df
```

[ ]:
```
# Calculate the absolute performance relative to the starting value for Apple␣
 ↪stock
stock_price_apple = pd.DataFrame()
stock_price_apple['Price'] = stock_prices['Apple']
start_value_apple = stock_price_apple['Price'].iloc[0]  # Starting value of the␣
 ↪stock
print(f'Simulated starting value of Apple stock: {start_value_apple}')
stock_price_apple['AbsPerformance'] = stock_price_apple['Price'] -␣
 ↪start_value_apple

# Display the last rows including the new column
print(stock_price_apple.tail())
```

In addition to simple calculations, there are also Pandas standard functions for many use cases.

[ ]:
```
# Calculate the relative performance of Apple stock
stock_price_apple['RelPerformance'] = stock_price_apple['Price'].pct_change()
stock_price_apple.tail()
```

4.2 Delete Data

If you want to delete a column or row, you can do so by:

- **Deleting Rows**: You can remove specific rows using `drop(index)`. Here, you specify the index of the rows that should be deleted.
- **Deleting Columns**: You can remove specific columns using `drop(column_name, axis=1)`.

Inplace Option: If you want to apply the changes directly to the existing DataFrame, you can use `inplace=True` to avoid creating a copy of the DataFrame.

Warning: `drop()` does not allow reverting to the original data once the rows or columns have been deleted, unless a copy was created beforehand.

[ ]:
```
# Delete a column
stock_price_apple.drop(['RelPerformance'], axis=1)
```

Drop returns a new dataframe without the deleted data. Without `inplace=True`, the original dataframe does not change.

[ ]:
```
stock_price_apple
```

[ ]:
```
# Delete the third an fourth row
stock_price_apple.drop(['2023-01-03', '2023-01-04'], inplace=True) # Watch the␣
 ↪index is a date
stock_price_apple.head()
```

5 Task: Working with Damage Data

In this task, you will perform various operations using a sample dataset of damage data. We will create the data using the following code:

[ ]:
```
# Creating example data
data = {
    'Claim_ID': range(1, 16),
    'Policy_Number': [f'PN{str(i).zfill(5)}' for i in np.random.randint(0,␣
 ↪10000, 15)],  # Random policy numbers
    'Claim_Amount': np.random.randint(1000, 5000, size=15),  # Random claim␣
 ↪amounts between 1000 and 5000
    'Claim_Date': pd.date_range(start='2023-01-01', periods=15, freq='D'),
    'Policy_Start': pd.date_range(start='2021-01-01', periods=15, freq='D'),
    'Region': [
        'North', 'East', 'South', 'West', 'North',
        'East', 'South', 'West', 'North', 'East',
        'South', 'West', 'North', 'East', 'South'
    ]
}
```

5.1 Tasks

1. **Create DataFrame**: Create a Pandas DataFrame from the data provided above.
2. **View Data**: Use the `head()` and `describe()` methods to get an overview of the dataset. What do you notice?
3. **Filter Data**: Filter the claims that have a claim amount of more than 2000. Which policyholders are affected?

4. **Determine Contract Duration**: Add a new column `Contract Duration` that indicates the duration of the contract in days until the date of the claim.
5. **Calculate Average Claim Amount**: Calculate the average claim amount of the claims.
6. **Group Claim Amounts**: Group the claim amounts by region.

5.2 Note

You can use Pandas functions to solve the tasks mentioned above. Please note that we have not yet discussed the approach for task 6. Take a look at the documentation for `mean()` and `groupby()`. (Documentation)

You can obtain the number of different values in the *Region* column using the `value_counts()` function.

[ ]:
```
# Add your code here
```

6 Grouping Data and Aggregation Functions in Pandas

In data analysis, it is often necessary to group data in order to calculate aggregated statistics. Pandas provides powerful functions to group data and perform various aggregation operations.

6.1 Basic Concepts

- **Grouping**: Data is grouped using the `groupby()` method. This method splits the data into groups based on the values of one or more columns.
- **Aggregation**: After grouping, various aggregation functions can be applied to calculate statistical measures. Commonly used aggregation functions include:
  - `mean()`: Calculates the average.
  - `sum()`: Calculates the sum.
  - `count()`: Counts the number of elements.
  - `max()`: Determines the maximum value.
  - `min()`: Determines the minimum value.

6.2 Example: Aggregating Damage Data

Suppose we have a DataFrame `damage_data` with damage information, which contains a column for the region and one for the amount of damage. We can calculate the average damage per region as follows:

```
average_damage = damage_data.groupby('Region')['Damage Amount'].mean()
print(average_damage)
```

7 Pivoting with Pandas

Pivoting in Pandas allows you to transform data into a clear table, similar to pivot tables in Excel. With the `pivot()` function, you can restructure data based on key columns (such as categories, regions, or time indicators). One column is used for the values you want to aggregate or display, while other columns are used as the index or column headers.

7.1 Example of Pivoting:

Suppose we want to create an overview of the average damage amount in different regions.

[ ]:
```
# Create a pivot table showing the mean claim amounts by region
claim_pivot = claim_data.pivot_table(values='Claim_Amount', index='Region',␣
 ↪aggfunc='mean')
claim_pivot
```

8 Reading and Writing Data in Pandas DataFrames

Pandas offers powerful functions for reading data from various file formats and writing DataFrames to files. The most common formats are CSV (Comma-Separated Values) and Excel, but it also supports JSON, HTML, LaTeX, Parquet, SAS, and more.

8.1 Reading Data

To read data from a CSV file into a Pandas DataFrame, use the `read_csv()` function. Here is an example:

```
import pandas as pd

# Reading a CSV file
df_csv = pd.read_csv('path/to/file.csv')

# Output the first five rows of the DataFrame
print(df_csv.head())
```

Reading an Excel file is just as simple:

```
# Reading an Excel file
df_excel = pd.read_excel('path/to/file.xlsx', sheet_name='Sheet1')

# Output the first five rows of the DataFrame
print(df_excel.head())
```

8.2 Writing Data

To write a DataFrame to a CSV file, use the `to_csv()` function. Here is an example:

```
# Writing the DataFrame to a CSV file
df_csv.to_csv('path/to/new_file.csv', index=False)
```

Or:

```
# Writing the DataFrame to an Excel file
df_excel.to_excel('path/to/new_file.xlsx', sheet_name='Sheet1', index=False)
```

9 Reading Files from Non-Local Sources

You can easily read data sources with `pandas` that are not local but rather "somewhere" on the internet or on a server. You just need to specify the URL to the location of the file.

For example, we can read files from a GitHub repository like this:

[ ]:
```
# Watch for the "raw" part of the URL
data = pd.read_csv('https://raw.githubusercontent.com/DeutscheAktuarvereinigung/
  ↪Python_fuer_Aktuare/refs/heads/main/data/weather_data.csv')
data.head()
```

9.1 Handling Missing Data

In real datasets, missing data is a common problem. Pandas offers a variety of methods to identify, handle, or clean missing values.

9.1.1 Detecting Missing Data

With the function `isnull()` or `isna()`, you can identify which entries are missing in a DataFrame. `isnull()` returns `True` if the value is missing (NaN) and `False` if a valid value is present.

Let's take a look at this using our insurance dataset from above.

[ ]:
```
# So far the dataset has no missing values, so we'll create some
nan_df = insurance_df.copy()

# Randomly insert NaN values (approx. 25% of the data)
for col in ['Age', 'Policy_Start', 'Duration']:
    nan_df.loc[nan_df.sample(frac=0.25).index, col] = np.nan

nan_df
```

[ ]:
```
nan_df.isna()
```

[ ]:
```
nan_df.isnull().sum()
```

9.2 Remove Missing Data

With `dropna()`, you can remove rows or columns with missing values. You can control whether to consider all missing values (how='all') or only individual missing values (how='any').

[ ]:
```
df_cleaned = nan_df.dropna()
df_cleaned
```

9.2.1 Filling Missing Data

Instead of removing missing values, we can also replace them with meaningful values. This can be, for example, the mean, median, or a specified value. Use the `fillna()` function for this purpose.

```
[ ]: nan_df['Age'].fillna(nan_df['Age'].mean(), inplace=True)
     nan_df
```

10 Task

Now it's your turn. In the *data* folder, you will find two datasets: - car_insurance_claims.csv (Car insurance data) - titanic_train.csv (Titanic passenger data)

Source:

The car insurance data comes from Databrick. Unfortunately, no further source is provided there.

The Titanic data is taken from the Titanic Machine Learning Challenge on Kaggle and contains data from 892 Titanic passengers. The dataset is used in the challenge for training machine learning algorithms.

10.1 Tasks for the Damage Case Dataset

1. Get an Overview

- **Task**: Read in the dataset and get an overview.
  - Use `head()` to view the first 5 rows of the dataset.
  - Use `info()` to obtain information about the data types and the number of entries.
  - Display a summary of the numerical columns with `describe()`.

Note The link to the online storage location of the data is: https://raw.githubusercontent.com/DeutscheAktuarvereinigung/Python_fuer_Aktuare/refs/heads/completed/da

2. Column Access and Simple Calculations

- **Task**: Work with individual columns.
  - Show the average amount of the `total_claim_amount`.
  - Find the maximum and minimum value for the `policy_annual_premium` column.
  - Create a new column `claim_ratio` that calculates the ratio of `total_claim_amount` to `policy_annual_premium`.

3. Filtering Data

- **Task**: Filter the dataset based on specific conditions.
  - Find all cases where the `incident_type` is reported as "Single Vehicle Collision".
  - Display all incidents that occurred in the state "NY" (`incident_state`) and where the claim amount (`total_claim_amount`) is over 10,000.
  - Find all incidents where more than 2 witnesses (`witnesses`) were reported.

4. Grouping and Aggregating

- **Task**: Group data and calculate averages.
  - Group the data by `incident_state` and show the average `total_claim_amount` for each state.
  - Group the data by `insured_occupation` and display the mean amount of `injury_claim`.
  - Find the average `property_claim` by `auto_make`.

5. Handling Missing Data

- **Task**: Deal with missing data.
  - Determine which columns contain missing values (`NaN`).
  - Count the missing values in the `authorities_contacted` column.
  - Fill the missing values in the `authorities_contacted` column with the value "Not Contacted".

6. Time-Based Analyses

- **Task**: Work with time data.
  - Convert the `incident_date` column to date format (`datetime`).
  - Find out in which month the most incidents occurred.
  - Display all incidents reported between February 15, 2015, and February 28, 2015.

10.2 Tasks: Analyses of the Titanic Dataset

Note The link to the online storage location of the data is: https://raw.githubusercontent.com/DeutscheAktuarvereinigung/Python_fuer_Aktuare/refs/heads/completed/da

1. **Who was the youngest passenger?**
   Determine the age of the youngest passenger on board the Titanic. What else can you find out about this passenger?

2. **Survival rate by gender**
   Investigate whether survival on the Titanic was related to the gender of the passengers. Calculate the survival rate for men and women separately. Use the `Survived` and `Sex` columns for this.

3. **Survival chances in different classes**
   Did the class in which a passenger traveled influence their chances of survival? Calculate the survival rate for each of the three classes (`Pclass`) and interpret the results.

4. **Average ticket prices per class**
   Find out how much passengers paid on average for their tickets in each class. Use the `Pclass` and `Fare` columns to calculate the average ticket prices.

5. **Largest families on board**
   Determine how many siblings and parents (columns `SibSp` and `Parch`) the passengers had on board. Who traveled with the largest family?

6. **Where did most passengers embark?**
   Investigate which port (`Embarked`) most passengers boarded from.

7. **Distribution of age groups**
   Divide the passengers into different age groups (e.g., children, teenagers, adults, seniors) and examine how these groups affect the survival rate.

8. **Most valuable ticket on board**
   Find out which was the most expensive ticket (`Fare`) on the Titanic and who purchased it.