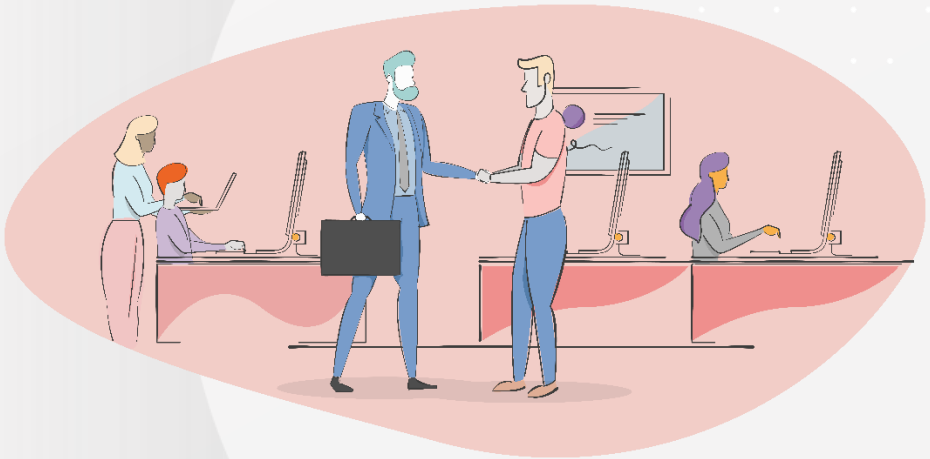
Data manipulation



L5 Data Engineer Higher Apprenticeship
Module 3 / 12 (**“Programming and Scripting Essentials”**)
Topic 3 / 8

rev. 1 (2024)

Ice breaker: Discussion

A bit of fun to start…

1. If you could have dinner with any historical figure, who would it be and why?

2. If you were given a one-time chance to teleport, where would you go and why?

3. Can you share an instance where Python significantly improved your data engineering workflow?

Submit your responses to the chat or turn on your microphone

Real-world data manipulation and Pandas

A finance case study

The scenario:

A banking analyst aims to analyse historical stock data for a particular company to guide investment decisions.

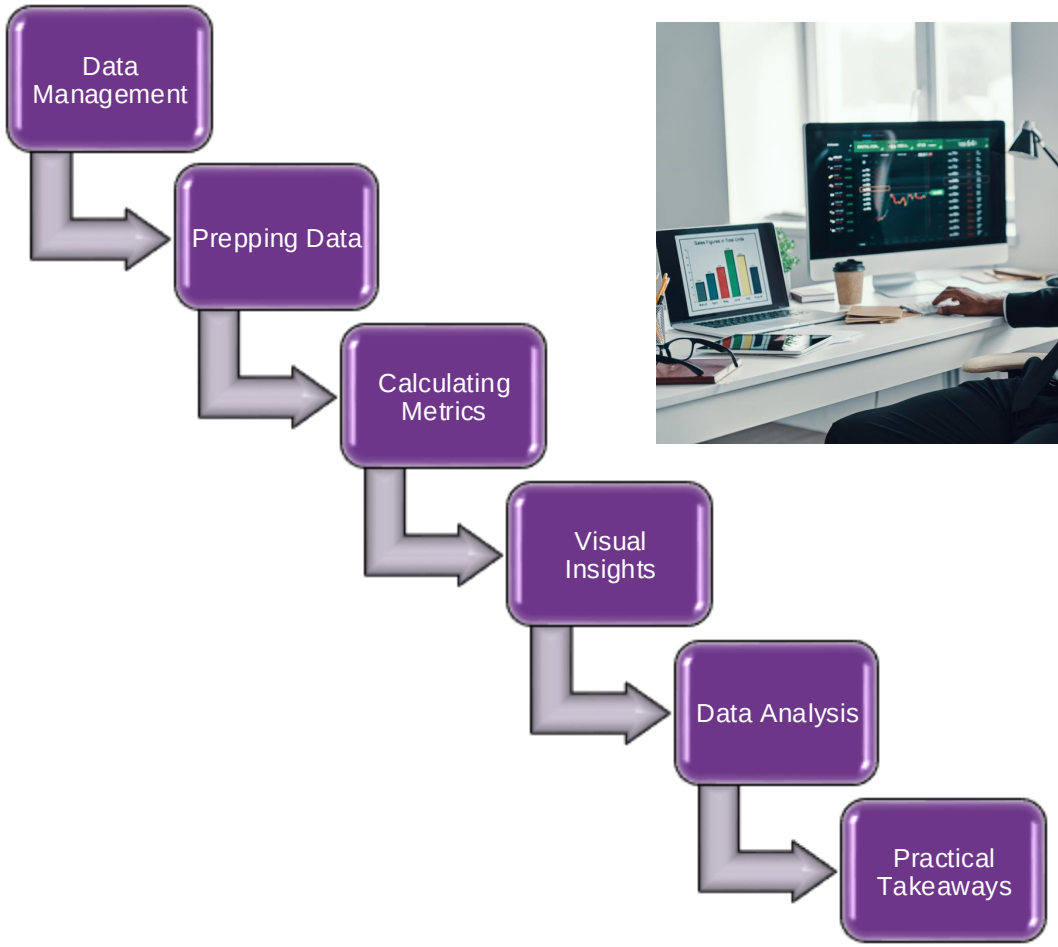
The analyst possesses a dataset containing daily stock prices over recent years.

1. Calculate important financial metrics, like daily returns and moving averages
2. Spot trends and potential investment prospects within the stock data
3. Compare the stock's performance with market indices

The analysts objectives

Real-world Application of Advanced Pandas (tbc)

A finance case study



Steps taken by the analyst

The results:

The banking analyst possesses a dataset containing daily stock prices over recent years.

The analyst effectively utilises Pandas for financial analysis.

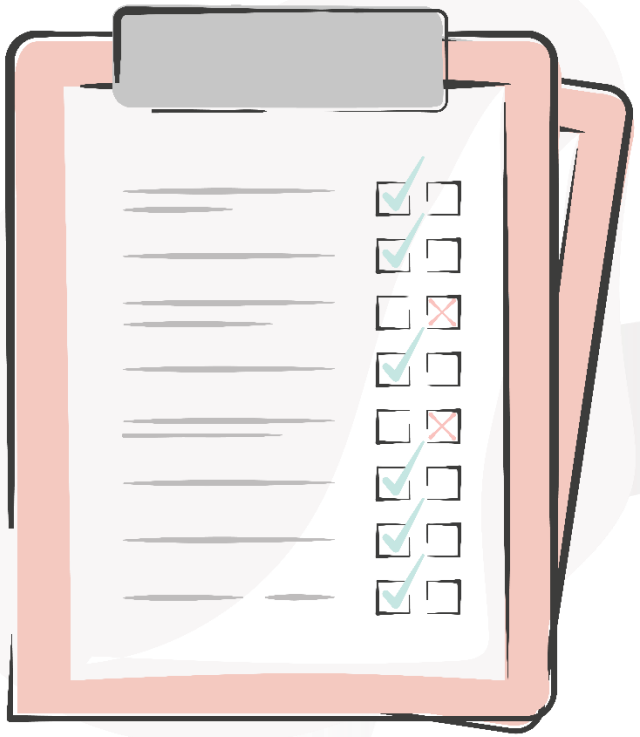
Pandas is a Python library for manipulating data.

Webinar agenda

This webinar will cover the following:

- Exploring data manipulation
- Loading and exploring data using Pandas
- Data wrangling with Pandas
- Understanding Time Series
- RegEx

Webinar length: 3 hours



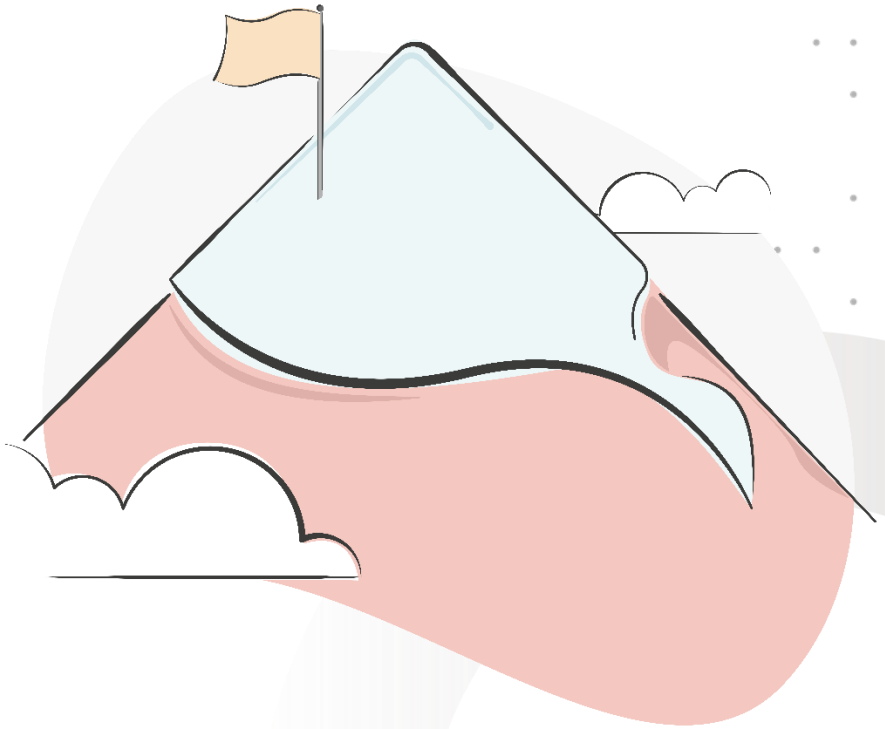
Learning objectives

Building Careers Through Education

Everybody Matters. Stronger Together. Student, Learner and Client Centric. Trust and Respect. Embrace Change.

This webinar supports completion of the following outcomes:

- Explain data manipulation using data engineering tools
- Wrangle and clean more complex datasets using Pandas
- Perform advanced analysis like pivots, aggregates on multi-indexed data
- Apply regular expressions to data



BPP

Explaining data manipulation

Explaining data manipulation

What is it?

- Data manipulation is the method used to modify, structure, format, or sort data so that it becomes useful and more manageable

Chatterjee and Segev (1991) defined Data manipulation as the process of altering or adjusting data for it to be more organized and readable.

Unpacking Pandas

What is Pandas?

Pandas is an open-source Python library that provides high-performance, easy-to-use data structures and data analysis tools.

Pandas enables users to:

- Clean
- Transform
- And explore datasets

Pandas is an open-source Python library

Panda Data Structures

What is a DataFrame?

A DataFrame is a two-dimensional tabular data structure, similar to a spreadsheet, where data is organised in rows and columns.

```python
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 22, 28],
    'City': ['New York', 'London', 'Paris', 'Tokyo']
}

df = pd.DataFrame(data)

print(df)
```

A Pandas DataFrame

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   22     Paris
3    David   28     Tokyo
```

A Pandas DataFrame Output

Introduction to Series

What is a Series in Pandas?

Key points:

- A Series is a **one-dimensional** labelled array in Pandas
- It can be thought of as a column in a spreadsheet or a simple array
- Series supports various data types and allows for easy indexing and data alignment
- Each element in a Series has a label (an index) that helps in data alignment and retrieval
- Series are used to represent one-dimensional data sets and are an essential component of DataFrames

```python
import pandas as pd

# Create a Series with labeled data
fruit_series = pd.Series([10, 15, 8, 12], index=['Apple', 'Banana', 'Orange', 'Grapes'])

# Display the Series
print(fruit_series)
```

An example of a one-dimensional labelled array or 'Series'

```
Apple     10
Banana    15
Orange     8
Grapes    12
dtype: int64
```

Output

Knowledge Check Poll

Which of the following statements about Pandas is correct?

A. A DataFrame is a one-dimensional labeled array in Pandas, ideal for handling columns or rows of data

B. Pandas is primarily a data visualisation library, offering a wide range of plotting and graphing functionalities

C. Data Cleaning and Preparation are not supported in Pandas, as it is mainly focused on data analysis

D. A DataFrame is a two-dimensional tabular data structure in Pandas, akin to a spreadsheet, while a Series is a one-dimensional labelled array.

Knowledge Check Poll

Which of the following statements about Pandas is correct?

A. A DataFrame is a one-dimensional labeled array in Pandas, ideal for handling columns or rows of data

B. Pandas is primarily a data visualisation library, offering a wide range of plotting and graphing functionalities

C. Data Cleaning and Preparation are not supported in Pandas, as it is mainly focused on data analysis

D. A DataFrame is a two-dimensional tabular data structure in Pandas, akin to a spreadsheet, while a Series is a one-dimensional labelled array.

Feedback

The correct statement is **D** – A DataFrame is a two-dimensional tabular data structure in Pandas, akin to a spreadsheet, while a Series is a one-dimensional labeled array.

Loading and exploring data using Pandas

Loading and Exploring Data using Pandas

Building Careers Through Education

Everybody Matters. Stronger Together. Student, Learner and Client Centric. Trust and Respect. Embrace Change.

Pandas provides convenient functions to read data from various file formats and explore the structure of the data.

In this section we will cover the following:

- Reading data from different files formats (CSV, Excel, etc)
- Understanding the structure of data

BPP

Reading Data from Different File Formats

Pandas provides several functions to read data from various file formats, making it convenient to load and analyse data from different sources, including:

```python
import pandas as pd

# Reading data from a CSV file
df_csv = pd.read_csv('data.csv')

# Display the DataFrame
print(df_csv.head())
```

Reading Data from CSV Files

```python
import pandas as pd

# Reading data from an Excel file
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# Display the DataFrame
print(df_excel.head())
```

Reading Data from Excel Files

Reading Data from Different File Formats (cont)

Pandas also supports reading data from various file formats, including JSON, SQL databases, HTML, and more.

```python
import pandas as pd

# Reading data from a JSON file
df_json = pd.read_json('data.json')

# Display the DataFrame
print(df_json.head())
```

Reading Data from JSON Files

```python
import pandas as pd
import sqlite3

# Connect to the SQL database
conn = sqlite3.connect('database.db')

# Reading data from an SQL table
query = 'SELECT * FROM table_name'
df_sql = pd.read_sql(query, conn)

# Display the DataFrame
print(df_sql.head())
```

Reading Data from SQL Files

Understanding the Structure of Data

Using head() and tail() Methods

The head() method is used to display the first few rows of the DataFrame, providing a quick overview of the data's structure.

Similarly, the tail() method displays the last few rows of the DataFrame.

```python
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['John', 'Alice', 'Bob', 'Emily', 'David'],
    'Age': [25, 28, 22, 24, 27],
    'City': ['New York', 'San Francisco', 'Chicago', 'Los Angeles', 'Seattle']
}

df = pd.DataFrame(data)

# Display the first few rows using head()
print(df.head())
```

An example of using the head() method

```python
# Display the last few rows using tail()
print(df.tail())
```

An example of using the tail() method

```
    Name  Age           City
0   John   25       New York
1  Alice   28  San Francisco
2    Bob   22        Chicago
3  Emily   24    Los Angeles
4  David   27        Seattle
```

Output

Understanding the Structure of Data

Using the ‘describe()’ method

The describe() method generates statistical summaries of the DataFrame.

This method provides measures like mean, median, standard deviation, and quartiles for numerical columns.

```
# Generate statistical summary using describe()
print(df.describe())
```

An example of the ‘describe()’ method

```
             Age
count   5.000000
mean   25.200000
std     2.949576
min    22.000000
25%    24.000000
50%    25.000000
75%    27.000000
max    28.000000
```

Output

Data Manipulation

Filtering Data

Pandas provides powerful tools to filter data based on specific conditions using Boolean indexing.

Boolean indexing allows filtering rows that meet certain criteria.

Remember: Boolean indexing in pandas is a way to filter data in a DataFrame using True/False values.

```python
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['John', 'Alice', 'Bob', 'Emily', 'David'],
    'Age': [25, 28, 22, 24, 27],
    'City': ['New York', 'San Francisco', 'Chicago', 'Los Angeles', 'Seattle']
}

df = pd.DataFrame(data)

# Filter rows where Age is greater than 24
filtered_data = df[df['Age'] > 24]
print(filtered_data)
```

An example of filtering data using Boolean indexing

```
    Name  Age           City
0   John   25       New York
1  Alice   28  San Francisco
4  David   27        Seattle
```

Output

Data Manipulation

Sorting Data

Pandas allows us to arrange data in a particular order using the sort_values() method.

Sorting can be done based on one or more columns in ascending or descending order.

```
# Sort the DataFrame by Age in descending order
sorted_data = df.sort_values(by='Age', ascending=False)
print(sorted_data)
```

An example of the sort_values(method)

```
    Name  Age           City
1  Alice   28  San Francisco
4  David   27        Seattle
0   John   25       New York
3  Emily   24    Los Angeles
2    Bob   22        Chicago
```

Output

Data Manipulation

Basic data transformations

Pandas supports various data transformation operations, such as adding or removing columns, renaming columns, and converting data types.

```python
# Adding a new column 'Gender'
df['Gender'] = ['Male', 'Female', 'Male', 'Female', 'Male']

# Removing the 'City' column
df.drop(columns='City', inplace=True)

# Renaming the 'Name' column to 'Full Name'
df.rename(columns={'Name': 'Full Name'}, inplace=True)

# Converting 'Age' column data type to float
df['Age'] = df['Age'].astype(float)

print(df)
```

An example of basic data transformation

```
  Full Name   Age  Gender
0      John  25.0    Male
1     Alice  28.0  Female
2       Bob  22.0    Male
3     Emily  24.0  Female
4     David  27.0    Male
```

Output

Knowledge Check Poll

Which of the following best describes a Series in Pandas?

A. A two-dimensional tabular data structure

B. A Python function for statistical analysis

C. A one-dimensional labelled array

D. A database management system

Submit your responses to the chat!

Knowledge Check Poll

Which of the following best describes a Series in Pandas?

A. A two-dimensional tabular data structure

B. A Python function for statistical analysis

C. A one-dimensional labelled array

D. A database management system

Submit your responses to the chat!

Feedback

The correct statement is **C -** A one-dimensional labelled array.

Data wrangling with Pandas

Section introduction

What is Data wrangling?

Data wrangling is a crucial aspect of data analysis that involves cleaning, transforming, and preparing data to make it suitable for analysis.

Examples of data wrangling include:

Data cleaning	Removing duplicates, handling missing values, and resolving errors
Data filtering and sub-setting	Filtering and extracting relevant subsets of data
Data transformation	Transforming and reshaping data to match analysis or visualisation tasks

Data Cleaning and Handling Missing Values

How is this done in Pandas?

Remember…

Pandas makes data cleaning simple with functions like dropna() to remove missing values and fillna() to impute values.

Also useful:

- astype()

```python
import pandas as pd

# Sample data with missing values
data = {'Name': ['John', 'Mary', np.nan, 'Lee'],
        'Age': [25, 30, np.nan, 28]}

df = pd.DataFrame(data)

# Removing rows with NaN values
df.dropna()

# Filling NaN values with a placeholder
df.fillna('MISSING')

# Casting Age to integer dtype
df['Age'] = df['Age'].astype(int)
```

Examples of Pandas cleaning functions

Data Cleaning and Handling Missing Values

.duplicated()

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

# Sample data with missing values and duplicates
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', np.nan],
    'Age': [25, 30, np.nan, 28, 35],
    'Test_Score': [85, np.nan, 92, np.nan, 78],
    'City': ['New York', 'London', 'Paris', 'Tokyo', 'Sydney']
}

# Creating the DataFrame
df = pd.DataFrame(data)

# Check for missing values in each column
print("Missing Values:")
print(df.isnull().sum())

# Check for duplicate rows
print("\nDuplicate Rows:")
print(df.duplicated())
```

Code

```
Missing Values:
Name          1
Age           1
Test_Score    2
City          0
dtype: int64

Duplicate Rows:
0    False
1    False
2    False
3    False
4    False
dtype: bool
```

Output

Data Cleaning and Handling Missing Values

.drop_duplicates()

In this step, we can use the drop_duplicates() function to remove duplicate rows from the DataFrame.

This function identifies duplicate rows based on all column values and keeps only the first occurrence of each duplicate row.

```
# Remove duplicate rows
df_cleaned = df_cleaned.drop_duplicates()
```

Code

```
DataFrame after removing duplicates:
      Name   Age  Test_Score      City
0    Alice  25.0        85.0  New York
1      Bob  30.0        88.75    London
2  Charlie  28.0        92.0     Paris
3    David  28.0        88.75     Tokyo
4      NaN  35.0        78.0    Sydney
```

Output

Filtering and Sorting Data

How is this done in Pandas?

Filtering data: Pandas provides powerful tools to filter data based on specific conditions.

Sorting data: Arrange data in a particular order using sort_values() method.

```
# Filtering data based on a condition
filtered_data = df[df['column_name'] > 50]
```

Figure 1: *An example of filtering data in Pandas*

```
# Sorting data based on a column
sorted_data = df.sort_values(by='column_name', ascending=False)
```

Figure 2: *An example of sorting data in Pandas*

Grouping Data

How does Pandas allow users to group data?

Pandas allows users to group data based on specific columns using the **groupby() method**.

Grouping is useful for segmenting data and performing aggregate operations on each group.

```python
import pandas as pd

# Sample dataset
data = {
    'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Value': [10, 15, 20, 25, 30, 35]
}

df = pd.DataFrame(data)

# Grouping data by the 'Category' column
grouped_data = df.groupby('Category')

# Printing the groups
for group_name, group_data in grouped_data:
    print(f"Group: {group_name}")
    print(group_data)
    print()
```

Example: Grouping data by a column

Aggregating Grouped Data

How does Pandas allow users to aggregate grouped data?

Once data is grouped, aggregate functions like sum(), mean(), count(), etc., can be applied to each group to compute meaningful statistics.

```python
import pandas as pd

# Sample dataset
data = {
    'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Value': [10, 15, 20, 25, 30, 35]
}

df = pd.DataFrame(data)

# Grouping data by the 'Category' column
grouped_data = df.groupby('Category')

# Applying aggregate functions to each group
print(grouped_data.sum())    # Calculate the sum of 'Value' for each group
print(grouped_data.mean())   # Calculate the mean of 'Value' for each group
print(grouped_data.count())  # Calculate the count of items in each group
```

Example: Aggregating grouped data

Aggregating Grouped Data

How does Pandas allow users to aggregate grouped data?

Building Careers Through Education

Everybody Matters. Stronger Together. Student, Learner and Client Centric. Trust and Respect. Embrace Change.

```
groupby('Sales Rep')
```

Order Id	Company Name	Val	Sale	Sales Rep
B0REXA478NU6HVR7	Pee-Wee Pigeon	3166	0	William Taylor
62O0815JUA2Q97T8	Off-Beat Anthea	8816	0	William Taylor
FEO9H0OZUXT7N3ER	Eerie Uselessness	8448	1	William Taylor
EYVCF5P7AMBC92BA	Wrong Crow	1200	0	William Taylor
UI7AN81HH6WM78IK	Extra-Thick 129%	7825	0	Willie Rau
WBTPPKDIK74QMFI7	Extra-Thick 129%	4471	0	Willie Rau
U0EOZ04DKLWEOPU5	Masterful Rose	7717	0	Willie Rau
Y5K9THTETEJE7N4I	Scary Experience	6673	0	Sam Rhodes
YPHVCGRPV49I068D	Later Pi	7546	0	Alvin Jenson
YQI7ELDM3GYJUO1W	Later Pi	4845	0	Alvin Jenson
9PMYCHJYRPLACQLO	Identical Mules	9092	1	Alvin Jenson
DCPULQV7JZJGA271	Brutal Machines	1957	0	Alvin Jenson
LP00YGZA9DEJTT3F	Brutal Machines	4100	0	Alvin Jenson
1KKHA7YOO5C54II9	Rectilinear Toll	8081	0	Helen Payne

Order Id	Company Name	Val	Sale	Sales Rep
B0REXA478NU6HVR7	Pee-Wee Pigeon	3166	0	William Taylor
62O0815JUA2Q97T8	Off-Beat Anthea	8816	0	William Taylor
FEO9H0OZUXT7N3ER	Eerie Uselessness	8448	1	William Taylor
EYVCF5P7AMBC92BA	Wrong Crow	1200	0	William Taylor

Order Id	Company Name	Val	Sale	Sales Rep
UI7AN81HH6WM78IK	Extra-Thick 129%	7825	0	Willie Rau
WBTPPKDIK74QMFI7	Extra-Thick 129%	4471	0	Willie Rau
U0EOZ04DKLWEOPU5	Masterful Rose	7717	0	Willie Rau

Order Id	Company Name	Val	Sale	Sales Rep
Y5K9THTETEJE7N4I	Scary Experience	6673	0	Sam Rhodes

Order Id	Company Name	Val	Sale	Sales Rep
YPHVCGRPV49I068D	Later Pi	7546	0	Alvin Jenson
YQI7ELDM3GYJUO1W	Later Pi	4845	0	Alvin Jenson
9PMYCHJYRPLACQLO	Identical Mules	9092	1	Alvin Jenson
DCPULQV7JZJGA271	Brutal Machines	1957	0	Alvin Jenson
LP00YGZA9DEJTT3F	Brutal Machines	4100	0	Alvin Jenson

Order Id	Company Name	Val	Sale	Sales Rep
1KKHA7YOO5C54II9	Rectilinear Toll	8081	0	Helen Payne

```
agg({
  'Order Id': 'size',
  'Val': ['sum','mean'],
  'Sale': ['sum','mean']
})
```

		Val		Sale	
Sales Rep	Count	Sum	Mean	Sum	Mean
William Taylor	4	21630	5408	1	25%
Willie Rau	3	20013	6671	0	0%
Sam Rhodes	1	6673	6673	0	0%
Alvin Jenson	5	27540	5508	1	20%
Helen Payne	1	8081	8081	0	0%

BPP

Aggregating Grouped Data

How does Pandas allow users to aggregate grouped data?

Pivot

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

```
df.pivot(index='foo',
         columns='bar',
         values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

Melt

df3

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

```
df3.melt(id_vars=['first', 'last'])
```

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

Stack

df2

		A	B
first	second		
bar	one	1	2
	two	3	4
baz	one	5	6
	two	7	8

MultiIndex

```
stacked = df2.stack()
```

first	second		
bar	one	A	1
		B	2
	two	A	3
		B	4
baz	one	A	5
		B	6
	two	A	7
		B	8

MultiIndex

Unstack(1)

stacked

first	second		
bar	one	A	1
		B	2
	two	A	3
		B	4
baz	one	A	5
		B	6
	two	A	7
		B	8

MultiIndex

```
stacked.unstack(1)
or
stacked.unstack('second')
```

	second	one	two
first			
bar	A	1	3
	B	2	4
baz	A	5	7
	B	6	8

MultiIndex

Knowledge Check Poll

Data wrangling in Pandas includes which of the following tasks?

A. Creating visual representations of the data using data visualization libraries like Matplotlib and Seaborn

B. Aggregating data to compute group-wise statistics or perform time-based aggregations

C. Removing duplicates, handling missing values, and resolving inconsistencies in the data

D. Using the astype() function to convert data types in a DataFrame.

Knowledge Check Poll

Data wrangling in Pandas includes which of the following tasks?

A. Creating visual representations of the data using data visualization libraries like Matplotlib and Seaborn

B. Aggregating data to compute group-wise statistics or perform time-based aggregations

C. Removing duplicates, handling missing values, and resolving inconsistencies in the data

D. Using the astype() function to convert data types in a DataFrame.

Feedback

The correct statement is **C -** Removing duplicates, handling missing values, and resolving inconsistencies in the data.

Pivot Tables

How does Pandas allow users to create a pivot table?

Remember…

Pandas provides the **pivot_table()** function, allowing users to create a spreadsheet-style pivot table.

This summarises data from a DataFrame, enabling better data analysis and understanding.

```python
import pandas as pd

# Sample dataset
data = {
    'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Value1': [10, 15, 20, 25, 30, 35],
    'Value2': [50, 55, 60, 65, 70, 75]
}

df = pd.DataFrame(data)

# Creating a pivot table that shows the mean of 'Value1' and 'Value2' for each 'Category'
pivot_table = df.pivot_table(index='Category', values=['Value1', 'Value2'], aggfunc='mean')

print(pivot_table)
```

Example: Creating a Pivot table

Merging, Joining, and Concatenating

Bringing DataFrames together

The merging, joining and concatenating functions in Pandas allow you to combine or bring together DataFrames in various ways:

Here is an example…

```
df1 = pd.DataFrame({'A': ['A0', 'A1'], 'B': ['B0', 'B1']})
df2 = pd.DataFrame({'A': ['A2', 'A3'], 'C': ['C2', 'C3']})

pd.concat([df1, df2])

df1.merge(df2, on='A')
```

An example of quick/concat merging in Pandas

Pandas recap summary

Here are the key points to remember from this Pandas recap:

Key points:

- **DataFrames** - 2D labeled data structures for easy manipulation
- **Series** - 1D labeled arrays that can store any data type
- **Data Cleaning** - Handling missing data, duplicates, formatting issues
- **GroupBy** - Split-apply-combine by category for aggregation
- **Merge/Join** - Combine DataFrames together by rows or columns
- **Pivot Tables** - Reshape data into spreadsheet-style summaries

Knowledge Check Poll

Which Pandas function allows you to join two DataFrames by matching the indexes row-wise?

A. concat()

B. append()

C. concat()

D. join()

Knowledge Check Poll

Which Pandas function allows you to join two DataFrames by matching the indexes row-wise?

A. concat()

B. append()

C. concat()

D. join()

Feedback

The correct statement is **D** – The pandas .join() method joins DataFrames by matching the index labels row-wise, similar to a SQL left join.

Understanding time series

Time Series Definition

Time series data is a sequence of data points ordered by time.

The data is indexed by time (for example - days, months, years).

Date	Temperature
1/1/2023	32
1/2/2023	35
1/3/2023	31
...	
12/30/2023	29
12/31/2023	25

A simple time series could be daily temperature data for a year…

Data/Time Data Types

Time Series Data with Pandas

- Pandas has datetime data types to represent dates, times, timestamps
- The pd.to_datetime() converts other data types to datetime
- pd.date_range() can generate a DatetimeIndex for a date range

```
from datetime import datetime

dates = pd.to_datetime([datetime(2023, 1, 1),
                        '1/5/2023',
                        1671424000000, # epoch time
                        '2023-01-07'])

index = pd.date_range('1/1/2023', periods=5, freq='D')

series = pd.Series(range(5), index=index)
series.dt.day
```

Example usage of data/time data types in Pandas

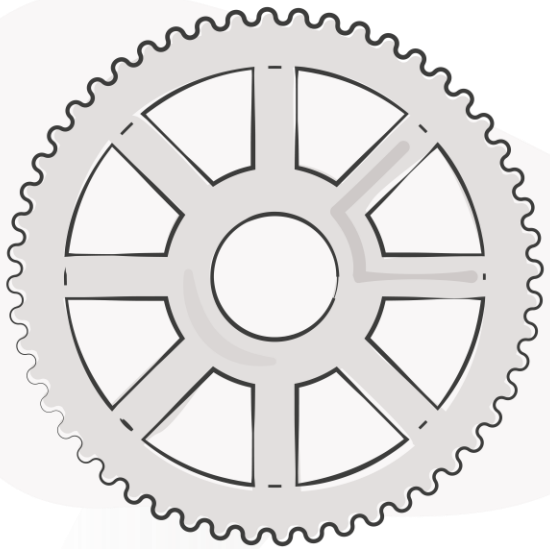
Practical application

Notebook activity

Your instructor will now walk you through the associated Python notebook for this topic.

This file can be found at the following link:

Python notebook link



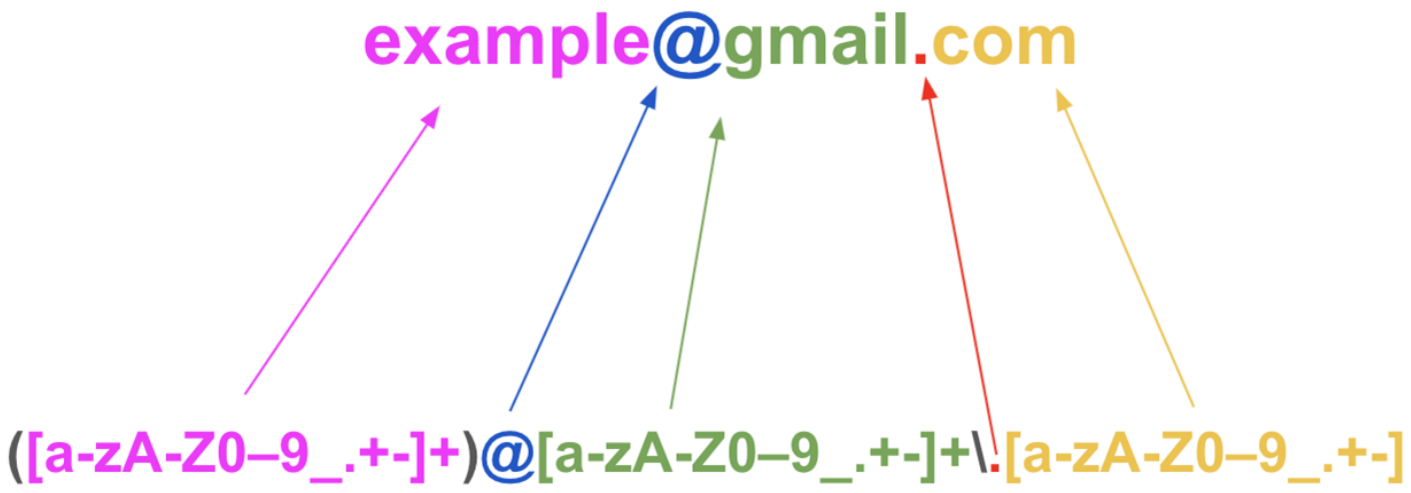
Practical application

Regular Expressions (Regex)

Section introduction

What is Regex?

- RegEX is a specific search pattern that can be used to easily match, locate and manage text
- To validate an input from the user (e.g. email address)



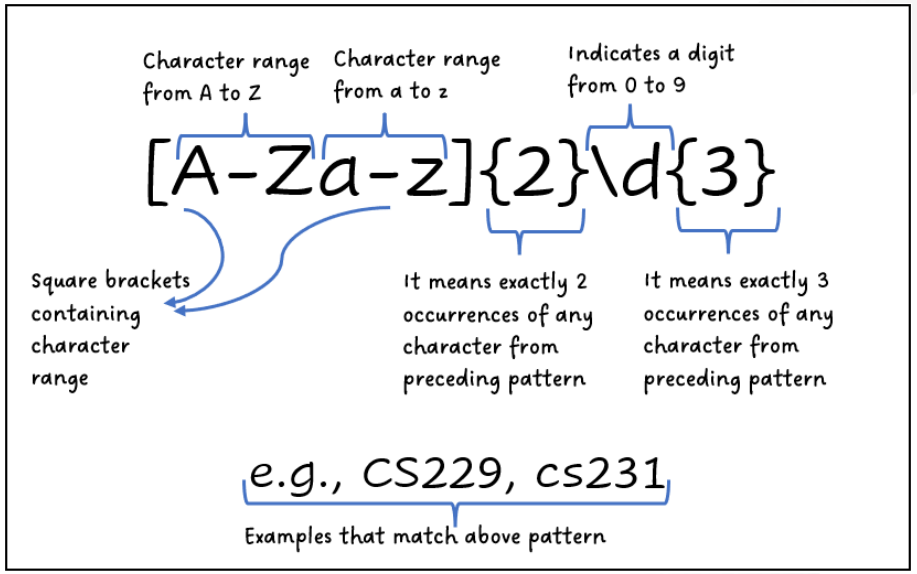
Example of Regex 1

The syntax of RegEx

Understanding the symbols

Symbol	Meaning
^	Start of input
$	End of input
[A-Za-z0-9]	Range of characters
\s	Whitespace
\w	Word characters
\.	. (dot) character
.	Any single character
\D	A single NON-digit
\S	NON-whitespace
\W	NON-word characters

Symbol	Meaning
+	One or more
{5}	Exactly five
{5,}	At least five
?	One or none
{2,5}	Two to five
{,5}	Up to five



Activity

Extracting the provider from email addresses

How to extract the provider from the email addresses below?

- xyz@t-online.de
- abc@gmail.com
- bcd@hotmail.edu
- cde@yahoo.co.uk
- edf@web.de

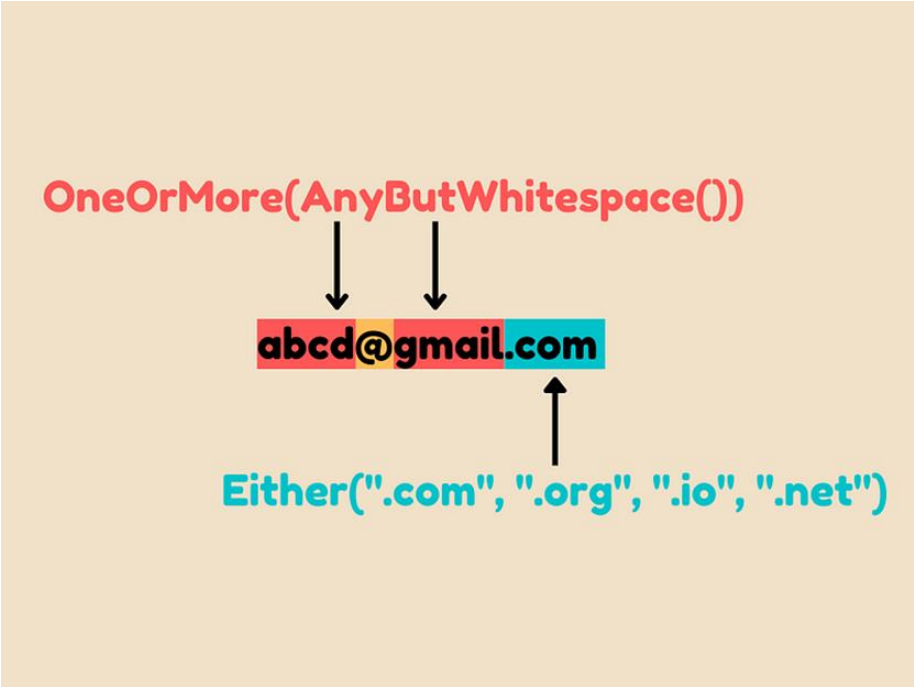
Group activity

One potent method for pattern extraction is the use of Regular Expressions (Regex).

Introduction to PRegEx

What you need to know…

- Introduction to Text Processing and Regular Expressions.
- Challenges with traditional regex: complexity and readability.
- Introducing PRegEx: A more human-friendly way to construct regex patterns.
- Main advantage: Making regex understandable using plain English.
- Installation of PRegEx library: pip install pregex==2.0.1 (requires Python >= 3.9)



An example of PRegEx

Basic usage of PRegEx

What you need to know…

- Import necessary classes and operators from PRegEx.
- Example: Capturing simple URLs from text.
- Understanding PRegEx functions: AnyButWhitespace, OneOrMore, Either.
- Output and equivalent regex pattern demonstration.
- Importance of readable regex for maintainability and collaboration.

```python
from pregex.core.classes import AnyButWhitespace
from pregex.core.quantifiers import OneOrMore
from pregex.core.operators import Either

text = "You can find me through my GitHub https://github.com/abcd1234"

pre = (
    "https://"
    + OneOrMore(AnyButWhitespace())
    + Either(".com", ".org")
    + OneOrMore(AnyButWhitespace())
)
```

A code example of PRegEx

Regular expressions with PRegEx

What you need to know…

- Setting a real-world scenario: Extracting structured information from text.
- Defining patterns for different date parts using PRegEx.
- Code snippet for matching dates in a given text.
- Output and equivalent regex pattern demonstration.
- Comparing traditional regex and PRegEx: Readability and ease of use.

```python
# create function
def find_date_pr(metadata):
    '''
    PRegEx function to extract date
    '''
    # Define the pattern using PRegEx
    day_pattern = OneOrMore(AnyDigit(), 2)
    month_pattern = OneOrMore(AnyLetter())
    year_pattern = Exactly(AnyDigit(), 4)

    pre = (
        day_pattern +
        ' ' +
        month_pattern +
        ' ' +
        year_pattern
    )

    matches = pre.get_matches(metadata)

    if len(matches) == 0:
        print(f"Found no dates in {metadata}")
        return None
    elif len(matches) > 1:
        raise ValueError("Multiple Dates")
    else:
        return matches[0]
```

A code example of regular expressions with PRegEx

Date Format Matching with PRegEx

What you need to know…

- Scenario: Extracting dates in a specific format from text.
- Introducing new PRegEx functions: AnyDigit and Exactly.
- Code snippet for matching dates in DD-MM-YYYY format.
- Output and equivalent regex pattern demonstration.
- Addressing common date format variations

```python
# match date
from pregex.core.classes import AnyDigit
from pregex.core.quantifiers import Exactly

two_digits = Exactly(AnyDigit(), 2)
four_digits = Exactly(AnyDigit(), 4)

pre = (
    two_digits +
    "-" +
    two_digits +
    "-" +
    four_digits
)

pre.get_matches(text)
```

A code example of date format matching with PRegEx

Email format matching with PRegEx

What you need to know…

- Scenario: Validating and extracting email addresses from text.
- Introducing new PRegEx functions: AnyButFrom, AtLeast, MatchAtLineEnd.
- Code snippet for matching email formats.
- Output and equivalent regex pattern demonstration.
- Discussing common challenges in email validation and how PRegEx can assist.

```python
from pregex.core.classes import AnyButFrom
from pregex.core.quantifiers import OneOrMore, AtLeast
from pregex.core.assertions import MatchAtLineEnd

non_at_sign_space = OneOrMore(AnyButFrom("@", ' '))
non_at_sign_space_dot = OneOrMore(AnyButFrom("@", ' ', '.'))
domain = MatchAtLineEnd(AtLeast(AnyButFrom("@", ' ', '.'), 2))

pre = (
    non_at_sign_space +
    "@" +
    non_at_sign_space_dot +
    '.' +
    domain
)

pre.get_matches(text)
```

A code example of email format matching with PRegEx

Knowledge check poll

What is the main advantage of using RegEx over traditional regular expressions?

a) It is faster to process

b) It has more functionality

c) It requires less code

d) It uses plain English

Submit your responses to the chat!

Knowledge check poll

What is the main advantage of using RegEx over traditional regular expressions?

a) It is faster to process

b) It has more functionality

c) It requires less code

d) It uses plain English

Feedback: D – The main advantage of using PRegEx over traditional regular expressions is that it allows writing regex patterns in plain English rather than complex syntax.

This makes PRegEx more readable and understandable compared to traditional regex.

Submit your responses to the chat!

Activity

Writing patterns that match the specification

Go ahead and try writing patterns that match the specification.

Your tutor will walk you through

https://www.regexone.com/

Group practice

Thank you

Do you have any questions, comments, or feedback?