Reading and Writing Data

Note: Parts of this notebook will only run on a Windows machine with Microsoft SQL Server installed. You can install the developer edition for free from here.

Using Pandas

`pandas` can easily read in data from a variety of sources, including CSV files, Excel files, SQL databases, and JSON files. In this section, we will learn how to use `pandas` to extract data from external sources. First of all, we need to import the `pandas` library.

```
import pandas as pd
```

CSV Files

CSV (comma-separated values) files are a common format for storing data. You can read a CSV file using the `read_csv` function. The function accepts both a file path and a URL as input. The documentation of the `read_csv` function is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html.

```
csv_data = pd.read_csv("../../data/AB_NYC_2019_unclean.csv")

csv_data.head()
```

Writing files is as easy as reading them. You can write a CSV file using the `to_csv` function. The documentation of the `to_csv` function is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html.

```
csv_data.to_csv("../../output/AB_NYC_2019_unclean.csv", index=False)
```

JSON Files

JSON (JavaScript Object Notation) is a popular data format that stores data as key-value pairs. You can read a JSON file using the `read_json` function. Like `read_csv`, the function accepts both a file path and a URL as input. The documentation of the `read_json` function is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html.

```
json_data = pd.read_json("../../data/AB_NYC_2019_unclean.json")
```

```
json_data.head()
```

Once again, writing JSON files is a very simple process. You can write a JSON file using the `to_json` function. The documentation of the `to_json` function is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html.

```
json_data.to_json("../../output/AB_NYC_2019_unclean.json")
```

Excel Files

`pandas` can read and write Excel files, but there are some important points to be aware of:

- It can only import data, not images or macros.
- By default, it imports the first sheet of the file.
- It does not import formulae, formatting, merged cells, and so on.
- You may need to install additional packages, depending on the version of Excel you are using. For example, if you are using Excel 2016, you need to install `openpyxl` to read `.xlsx` files.
  - You can install it using `pip install openpyxl`.

`read_excel` supports `xls`, `xlsx`, `xlsm`, `xlsb`, `odf`, `ods` and `odt` file extensions read from a local filesystem or URL. To function also has the ability to read a single sheet or a list of sheets.

See the documentation of the `read_excel` function at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html for further information.

```
excel_data = pd.read_excel("../../data/AB_NYC_2019_unclean.xlsx")

excel_data.head()
```

Writing Excel files is a little more complicated than reading them. You can use the `to_excel` function to write a `DataFrame` to an Excel file as a single sheet. However, you cannot write multiple `DataFrame`s to a single Excel file this way.

```
excel_data.to_excel("../../output/AB_NYC_2019_unclean.xlsx", index=False)
```

If you wish to write multiple `DataFrame`s to a single Excel file, you need to use the `ExcelWriter` class. The documentation of the `ExcelWriter` class is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.ExcelWriter.html. The following example shows how to write multiple `DataFrame`s to a single Excel file. Be aware, this method can take a long time to run if you have a lot of data.

```
from pandas import ExcelWriter

clean_data = pd.read_csv(
    "https://raw.githubusercontent.com/alexeygrigorev/datasets/master/AB_NYC_2019.csv"
)

with ExcelWriter("../../output/AB_NYC_2019.xlsx") as writer:
    excel_data.to_excel(writer, sheet_name="raw_data", index=False)
    clean_data.to_excel(writer, sheet_name="clean_data", index=False)
```

SQL Databases

`pandas` can read and write data from a variety of SQL databases, including MySQL, PostgreSQL, SQLite, and Microsoft SQL Server. We will be focusing on Microsoft SQL Server in this section.

First of all, lets create a database connection to a Microsoft SQL Server database. We will use the `sqlalchemy` and `pyodbc` library to create the connection. You can install `pyodbc` using `pip install pyodbc`. The documentation of the `create_engine` function is available at https://docs.sqlalchemy.org/en/20/core/engines.html. Note: make sure to enable TCP/IP connections in SQL Server Configuration Manager and to allow SQL Server to accept remote connections.

Note: If you are creating a Microsoft SQL Server database from scratch/installing for the first time, make sure to create a user with a password and to enable SQL Server authentication. You can do this by right-clicking on the server in SQL Server Management Studio and selecting Properties. Then, go to the Security tab and select SQL Server and Windows Authentication mode. You can then create a new user by right-clicking on the Security folder and selecting New > Login. Make sure to give the user the db_owner role for the database you are connecting to.

```
from sqlalchemy import create_engine

server = "localhost"
database = "nyc_airbnb"
username = "user"
password = "Pass123!"
driver = "SQL+Server"  # or "ODBC+Driver+18+for+SQL+Server" depending on your configuratio

engine = create_engine(
    f"mssql+pyodbc://{username}:{password}@{server}/{database}?driver={driver}"
```

```
)

# Test the connection
try:
    engine.connect()
except Exception as e:
    print(e)
    raise Exception("Unable to connect to database")
```

First, let's create a table in the database. We will do this using the `execute` method of the `engine` object.

```
from sqlalchemy import text

with engine.connect() as conn:
    conn.execute(text("DROP TABLE data"))
    conn.execute(
        text(
            """
                CREATE TABLE data (
                    id INTEGER PRIMARY KEY,
                    name TEXT,
                    host_id INTEGER,
                    host_name TEXT,
                    neighbourhood_group TEXT,
                    neighbourhood TEXT,
                    latitude REAL,
                    longitude REAL,
                    room_type TEXT,
                    price INTEGER,
                    minimum_nights INTEGER,
                    number_of_reviews INTEGER,
                    last_review TEXT,
                    reviews_per_month REAL,
                    calculated_host_listings_count INTEGER,
                    availability_365 INTEGER
                )
            """
        )
    )
```

Now lets import `pandas` and import the NYC Airbnb data into a `DataFrame`. We

can then commit this data to the database using the `to_sql` function. The documentation of the `to_sql` function is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html. If we were to use `to_sql` without initially creating the table, `pandas` would create the table for us automatically. However, we would have no control over the data types of the columns. By creating the table manually, we can ensure that the data types are correct. In particular, you cannot specify a primary key and non-nullable datatypes using `to_sql`. This can potentially cause problems down the line, especially if you are using the database for production purposes.

```
import pandas as pd

clean_data = pd.read_csv(
    "https://raw.githubusercontent.com/alexeygrigorev/datasets/master/AB_NYC_2019.csv"
)

clean_data.to_sql("data", con=engine, if_exists="append", index=False)
```

Now that the SQL Database is populated with data, we can query it into a `pandas DataFrame` using the `read_sql` function. The documentation of the `read_sql` function is available at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql.html.

```
sql_data = pd.read_sql("SELECT * FROM data", con=engine)

sql_data.head()
```

We can use a SQLAlchemy `Select` statement instead of a string query to read in the database. It is a little more complicated, but it allows us to leverage the full combined power of SQL and Python. First of all, we need to map the tables we're interested in to Python classes. We can do this manually, but it is much easier to use the `automap_base` function. Let's see how it would look if we were to do it manually:

```
from sqlalchemy.orm import declarative_base, sessionmaker
from sqlalchemy import Column, Integer, String, Float

Base = declarative_base()


class DataManual(Base):
    __tablename__ = "data"
    id = Column(Integer, primary_key=True)
    name = Column(String)
    host_id = Column(Integer)
```

```
    host_name = Column(String)
    neighbourhood_group = Column(String)
    neighbourhood = Column(String)
    latitude = Column(Float)
    longitude = Column(Float)
    room_type = Column(String)
    price = Column(Integer)
    minimum_nights = Column(Integer)
    number_of_reviews = Column(Integer)
    last_review = Column(String)
    reviews_per_month = Column(Float)
    calculated_host_listings_count = Column(Integer)
    availability_365 = Column(Integer)


Session = sessionmaker(bind=engine)
session = Session()

query = session.query(DataManual)

sql_data = pd.read_sql(query.statement, con=engine)

sql_data.head()
```

Mapping every table manually in a database would be in extremely tedious process. Fortunately, we can use the `automap_base` function to automatically map all the tables in a database. Note: Tables without a primary key will not be mapped. The documentation of the `automap_base` function is available at https://docs.sqlalchemy.org/en/20/orm/extensions/automap.html.

```
from sqlalchemy.ext.automap import automap_base

# reflect the tables
Base = automap_base()
Base.prepare(autoload_with=engine)

# mapped classes are now created with names by default
# matching that of the table name.
Data = Base.classes.data

Session = sessionmaker(bind=engine)
session = Session()
```

```
query = session.query(Data)

sql_data = pd.read_sql(query.statement, con=engine)

sql_data.head()
```

Alternatively, we could use `pyodbc` directly to connect to the database instead of using `sqlalchemy`. However, this method is not recommended as `pandas` does not natively support `pyodbc` and it is not as flexible as using `sqlalchemy`. You can find the documentation for `pyodbc` at https://github.com/mkleehammer/pyodbc/wiki.

```
import pyodbc

server = "localhost"
database = "nyc_airbnb"
username = "user"
password = "Pass123!"
driver = "{SQL Server}"
conn_str = f"Driver={driver};Server={server};Database={database};Uid={username};Pwd={passw


try:
    pyodbc.connect(conn_str)
except Exception as e:
    print(e)
    raise Exception("Unable to connect to database")

with pyodbc.connect(conn_str) as cnxn:
    # select from SQL table to insert in dataframe.
    query = "SELECT * FROM data"
    df = pd.read_sql(query, cnxn)

df
```

We have only covered a small number of `pandas` IO tools in this section. For more information, see the documentation of the `pandas` library at https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html.

Rest APIs

REST (Representational State Transfer) API is a web-based service that uses HTTP methods to get, send, delete, or update data. Many online platforms provide REST APIs to interact

with their data, and in most cases, this data is returned in JSON format.

To work with REST APIs, you will need a package for sending HTTP requests. Python's requests package is a great choice for this task. You can install it using pip:

```
pip install requests
```

Let's assume we have a REST API that returns a JSON response. In our example, we will use a public API that provides information about users: https://jsonplaceholder.typicode.com/users.

Here is how you can read this data into a pandas DataFrame:

```python
# Import required libraries
import pandas as pd
import requests

# Define the API endpoint
url = "https://jsonplaceholder.typicode.com/users"

# Send a GET request to the REST API
response = requests.get(url)

# Check that the GET request was successful
if response.status_code == 200:
    # Parse the JSON response to a Python dictionary
    data = response.json()

    # Convert the dictionary to a pandas DataFrame
    df = pd.DataFrame(data)

else:
    print(f"Request failed with status code {response.status_code}")

df.head()
```

This script will output a DataFrame where each row corresponds to a user, and columns correspond to user attributes (id, name, username, etc.). If you want to read only a specific attribute from the API response, you can do it by selecting this attribute when creating the DataFrame. Let's say you are interested in the names and email addresses of the users only:

```python
df = pd.DataFrame(data, columns=["name", "email"])

df.head()
```

Please be aware that not all APIs are public and some require authentication. The procedure to authenticate will depend on the specific API, so always refer to the API's documentation for instructions. Also, always be aware of the API's rate limits. If you send too many requests in a short period of time, the server might block your IP address. To avoid this, you can add pauses between your requests using the `time.sleep()` function. Finally, keep in mind that API responses can be large, and downloading and processing them can take a lot of time. Consider filtering the data on the server side (if the API supports this) or processing the data in chunks.

Web Scraping

Web scraping is the process of extracting data from websites. Python, with its rich ecosystem of libraries, is a popular choice for web scraping tasks. In this tutorial, we'll cover the basics of web scraping using Python and two popular libraries: Requests and Beautiful Soup.

Installing the Libraries

Before starting, you need to install Requests and Beautiful Soup libraries. To do this, run the following command:

```
pip install requests beautifulsoup4
```

Fetching a Web Page

First, we need to fetch the web page's content. We'll use the Requests library for this purpose. In this tutorial, we'll use the Wikipedia page for "List of state and union territory capitals in India" as an example. The following code fetches the content of the web page and stores it in the `page_content` variable:

```
import requests

url = "https://en.wikipedia.org/wiki/List_of_state_and_union_territory_capitals_in_India"
response = requests.get(url)

if response.status_code == 200:
    print("Page successfully fetched!", end="\n\n")
    page_content = response.text
    print(f"Object type: {type(page_content)}", end="\n\n")
    print("Page content:", end="\n\n")
    print(page_content)
```

```
else:
    print(f"Error {response.status_code}: unable to fetch the page.")
```

Parsing the Web Page

Now that we have the HTML content, we need to parse it to navigate and extract the data. Beautiful Soup will help us with this task:

```
from bs4 import BeautifulSoup

soup = BeautifulSoup(page_content, "html.parser")

print(f"Object type: {type(soup)}", end="\n\n")
print("Page content:", end="\n\n")
print(soup.prettify())
```

Extracting the Data

After parsing the HTML, we can now locate and extract the desired information using Beautiful Soup. Let's say we want to extract all the headings (h1, h2, and h3 tags) from the page. We can do this using the `find_all` method:

```
headings = soup.find_all(["h1", "h2", "h3"])

for heading in headings:
    print(heading.get_text())
```

We can extract a specific heading by using the `find` method and passing the tag id as a keyword argument:

```
specific_heading = soup.find(id="List")

print(specific_heading.get_text())
```

Extracting Tables to Pandas DataFrames

The page contains tables with of states and union territories. We can extract the tables using the `find_all` method of Beautiful Soup and convert them to a list of Pandas DataFrames using the `read_html` method of Pandas.

Note you may need to install the `lxml` and `html5lib` libraries to use the `read_html` method. You can install them using the following command:

```
pip install lxml html5lib
```

```
import pandas as pd

tables = soup.find_all("table")

df_list = pd.read_html(str(tables))

for df in df_list:
    display(df)
```

Project 01: Extracting Books Data Into A Database

As a starting point you can use the library management system you built in the previous tutorial and ingest books data from external sources to populate your database. If you haven't built the library management system yet, you can use the solution code from the previous tutorial (check the GitHub repository) as a starting point.

For this project however, you can use any external data source you want and create an appropriate database schema to store the data. The data sets below are merely suggestions. You can use any data source you want as long as it is in a format that can be ingested into a database. You can use CSV files, REST APIs, or web scraping to get the data. You can use any database you want, but we recommend using Microsoft SQL Server or SQLite for this project.

For CSV files, here are some datasets you can use:

- Books Dataset
- Goodreads-books
- goodbooks-10k
- Book Recommender: Collaborative Filtering, Shiny
- Amazon Top 50 Bestselling Books 2009 - 2019

If you are using a REST API, here are some APIs you can use:

- Open Library Books API
- Google Books APIs

If you are using web scraping. Here are some websites you can use:

- Books To Scrape

- Project Gutenberg
- Open Library

If you are web scraping, to get you started, here's a code snippet that fetches and prints the titles of all the books on the first page of the Books To Scrape website:

```
import requests
from bs4 import BeautifulSoup

url = "http://books.toscrape.com/"
response = requests.get(url)

if response.status_code == 200:
    print("Page successfully fetched!")
    page_content = response.text
else:
    print(f"Error {response.status_code}: unable to fetch the page.")

soup = BeautifulSoup(page_content, "html.parser")

books = soup.find_all("article", class_="product_pod")

for book in books:
    title = book.find("h3").find("a")["title"]

    print(f"Title: {title}")
    print("-----------")
```

You may need to inspect the HTML of the website to figure out the tags and attributes to use for extracting the data. You can use the "Inspect" option in your browser's right-click menu to open the developer tools. You can also use the "View Page Source" option to view the HTML source code of the page.

You can further customize your code to navigate through multiple pages and extract more data. You could try to extract the data into a pandas DataFrame to perform further analyses and manipulations before committing to your database. The possibilities are endless!

Project 02: Data Exploration

We have created a sample AdventureWorks SQL database on Microsoft Azure that you can use for this project. Use `sqlalchemy` and `pyodbc` to connect to the database and read the data into a pandas DataFrame. You can use the following code snippet to connect to the database:

```
import pandas as pd
from sqlalchemy import create_engine

server = "rdp-phsa.database.windows.net"
database = "db"
username = "reader"
password = "Temppassword123"
driver = "ODBC+Driver+18+for+SQL+Server"

engine = create_engine(
    f"mssql+pyodbc://{username}:{password}@{server}/{database}?driver={driver}"
)

try:
    engine.connect()
except Exception as e:
    print(e)
    raise Exception("Unable to connect to database")

query = "SELECT * FROM SalesLT.SalesOrderDetail"
df = pd.read_sql(query, con=engine)

df
```

	SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	UnitPrice	UnitPriceDiscount	LineTota
0	71774	110562	1	836	356.898	0.0	356.898
1	71774	110563	1	822	356.898	0.0	356.898
2	71776	110567	1	907	63.900	0.0	63.900
3	71780	110616	4	905	218.454	0.0	873.816
4	71780	110617	2	983	461.694	0.0	923.388
...	...	...	...	...	...	...	...
537	71938	113312	3	715	29.994	0.0	89.982
538	71938	113313	1	881	32.394	0.0	32.394
539	71938	113314	2	875	5.394	0.0	10.788
540	71938	113315	3	800	672.294	0.0	2016.882
541	71946	113406	1	916	31.584	0.0	31.584

```
from sqlalchemy.ext.automap import automap_base
from sqlalchemy.orm import sessionmaker
```

```
# reflect the tables
Base = automap_base()
Base.prepare(autoload_with=engine, schema="SalesLT")

for table in Base.classes.keys():
    print(table)

SalesOrderDetail = Base.classes.SalesOrderDetail

Session = sessionmaker(bind=engine)
session = Session()

query = session.query(SalesOrderDetail)

df = pd.read_sql(query.statement, con=engine)

df
```

```
Address
Customer
CustomerAddress
Product
ProductCategory
ProductModel
ProductDescription
ProductModelProductDescription
SalesOrderDetail
SalesOrderHeader
```

	SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	UnitPrice	UnitPriceDiscount	LineTota
0	71774	110562	1	836	356.898	0.0	356.898
1	71774	110563	1	822	356.898	0.0	356.898
2	71776	110567	1	907	63.900	0.0	63.900
3	71780	110616	4	905	218.454	0.0	873.816
4	71780	110617	2	983	461.694	0.0	923.388
...	...	...	...	...	...	...	...
537	71938	113312	3	715	29.994	0.0	89.982
538	71938	113313	1	881	32.394	0.0	32.394
539	71938	113314	2	875	5.394	0.0	10.788
540	71938	113315	3	800	672.294	0.0	2016.882

	SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	UnitPrice	UnitPriceDiscount	LineTota
541	71946	113406	1	916	31.584	0.0	31.584

Read in the data from each table into separate DataFrames and explore and manipulate the data using pandas. If you wish, you can also use the `sqlalchemy` library to perform SQL queries on the database and read the results into a DataFrame. Try generating some visualizations using the `matplotlib` library. You can also use the `seaborn` library to generate more advanced visualizations. Alternatively, you can output the data to a CSV file and use a tool like Microsoft Excel or Power BI to generate visualizations.

Further Reading

Check out the following resources for more information on importing data into Python:

- pandas: How to Read and Write Files
- pandas Documentation: IO Tools
- REST APIs with Python
- Web Scraping with Python: A Beginner's Guide
- Beautiful Soup: Build a Web Scraper With Python
- Python Web Scraping Tutorial
- Requests Documentation
- Beautiful Soup Documentation