Data Ingestion

Contents

- Importing Libraries and Modules
- Establishing the Database Connection
- Reading the dataset and transforming it into a dataframe
- Data profiling

The process conducted in this section involves reading the *candidates.csv* dataset, and transforming it into a Pandas DataFrame for later storage in a database.

Importing Libraries and Modules

- **os and dotenv:** These libraries are used to manage environment variables securely. Loading database credentials from a .env file ensures that sensitive information is not hard-coded into the script, enhancing security and making the codebase more maintainable.

- **sqlalchemy:** This library provides a powerful Object-Relational Mapping (ORM) capability, enabling efficient interaction with the MySQL database. The `create_engine` and `text` modules from SQLAlchemy simplify the process of connecting to and querying the database. The `types as sqltypes` modules provides access to SQLAlchemy's type system, allowing you to explicitly define the data types of columns when creating tables or interacting with data. This is important for data integrity and performance.
- **pandas:** This library is utilized for data manipulation and analysis. Transforming the candidates dataset into a Pandas DataFrame allows for easy manipulation and preparation before writing the data to the MySQL database.

Establishing the Database Connection

To import and reuse the database connection across different notebooks, the connection logic can be encapsulated in Python modules inside a package. This modules will be stored in the `/src/connection` folder of the project.

When get_db_connection function is imported from the module db_utils.py into another script or notebook and called, the connection logic will executed. By following this practice, redundant setup steps can be avoided.

Creating and Using a "connection" Python Package

To organize Python code effectively, directories can be designated as packages. By creating the `connection` package and using the `setup_env.py` and `db_utils.py modules`, the code related to database connection and environment setupcan can be organized and streamlined.This involves the following steps:

1. Create the Directory:

> A directory is created to hold related Python modules. This directory becomes the package. The directory needed to be created in this cases is `connection` and is created inside the `./src` directory.

2. Add an `__init__.py` File:

> An `__init__.py` file is placed inside the directory. This file, even if empty, is *crucial* because its presence signals to Python that the directory should be treated as a package. It can also contain initialization code for the package, such as setting up default configurations or importing commonly used modules within the package.

3. Add Modules to the Package:

> Python modules (`.py` files) containing the actual code are added to the directory. These modules become accessible through the package.
>
> The project directory is then organized as follows:
>
> > ```
> > project/
> > ├── data/
> > ├── docs/
> > ├── notebooks/
> > │   └── example_notebook.ipynb
> > ├── src/
> > │   └── connection/
> > │       ├── __init__.py
> > │       ├── db_utils.py
> > │       └── setup_env.py
> > ├── .gitignore
> > ├── .readthedocs.yaml
> > ├── README.md
> > ├── requirements.txt
> > └── venv/
> > ```

`db_utils.py` module

The db_utils.py module contains utility functions for database operations. These functions include connecting to the database and reading data from the database.

To establish a connection to the MySQL database, environment variables are loaded from the .env file, which securely stores database credentials. The sqlalchemy library's `create_engine` function is used to create a database engine instance, which facilitates the connection to the MySQL database. This approach ensures that the database credentials are not hard-coded into the script, enhancing security.

```
 import os
 from dotenv import load_dotenv
 from sqlalchemy import create_engine

def get_db_connection():
     load_dotenv()
     user = os.getenv('MYSQL_USER')
     password = os.getenv('MYSQL_PASSWORD')
     host = os.getenv('MYSQL_HOST')
     port = os.getenv('MYSQL_PORT')
     dbname = os.getenv('MYSQL_DB')
     db_url = f"mysql+mysqlconnector://{user}:{password}@{host}:{port}/{dbname

 try:
     engine = create_engine(db_url)
     connection = engine.connect()
     print("Connected to the database successfully")
     return connection
 except Exception as e:
     print(f"Error: {e}")
     return None
```

`setup_env.py` module

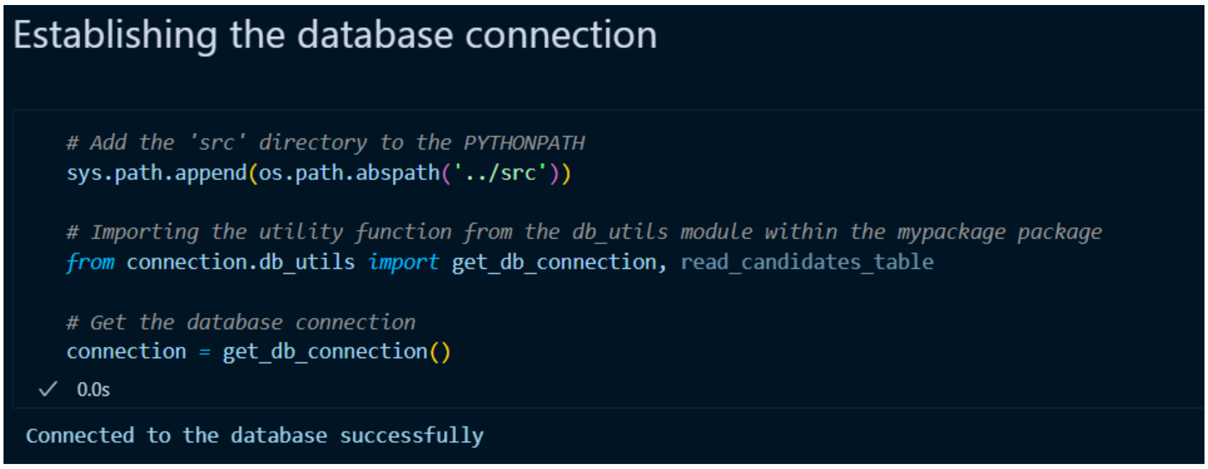
The setup_env.py module handles the environment setup, including adding the `src` directory to the PYTHONPATH. This ensures that the package modules can be imported easily.

```
import sys
import os

def setup_pythonpath():
    # Add the 'src' directory to the PYTHONPATH
    sys.path.append(os.path.abspath('../src'))

def setup_environment():
    setup_pythonpath()
    print("Environment setup complete.")
```

Usage in Notebooks

To use the `connection` package and its modules in the project´s Jupyter notebooks, the following code is used:

> ```
> # Add the 'src' directory to the PYTHONPATH
> sys.path.append(os.path.abspath('../src'))
>
> # Import the setup script
> from src.mypackage.setup_env import setup_environment
>
> # Run the setup script
> setup_environment()
> ```


Establishing the database connection

```
# Add the 'src' directory to the PYTHONPATH
sys.path.append(os.path.abspath('../src'))

# Importing the utility function from the db_utils module within the mypackage package
from connection.db_utils import get_db_connection, read_candidates_table

# Get the database connection
connection = get_db_connection()
```

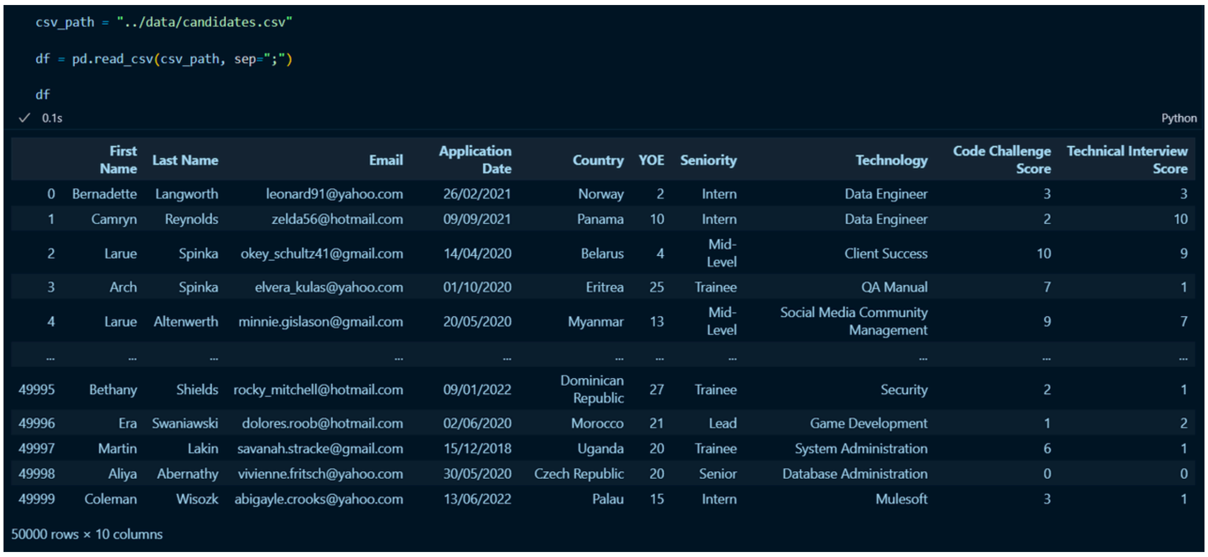
✓ 0.0s

```
Connected to the database successfully
```

Reading the dataset and transforming it into a dataframe

In this section data is loaded from a CSV file into a DataFrame for further data processing and analysis. The variable `csv_path` to the relative file path of the *candidates* CSV file in the proyect. In this case, the file path points to the candidates.csv file located in the data directory, which is one level up from the current working directory.

Then, the `pd.read_csv` function reads the CSV file into a DataFrame, with fields separated by semicolons. The DataFrame `df` holds the data from the CSV file in a structured format suitable for manipulation and analysis using Pandas. It contains **50.000 rows** and 10 **columns**.



```python
csv_path = "../data/candidates.csv"

df = pd.read_csv(csv_path, sep=";")

df
```

✓ 0.1s Python

	First Name	Last Name	Email	Application Date	Country	YOE	Seniority	Technology	Code Challenge Score	Technical Interview Score
0	Bernadette	Langworth	leonard91@yahoo.com	26/02/2021	Norway	2	Intern	Data Engineer	3	3
1	Camryn	Reynolds	zelda56@hotmail.com	09/09/2021	Panama	10	Intern	Data Engineer	2	10
2	Larue	Spinka	okey_schultz41@gmail.com	14/04/2020	Belarus	4	Mid-Level	Client Success	10	9
3	Arch	Spinka	elvera_kulas@yahoo.com	01/10/2020	Eritrea	25	Trainee	QA Manual	7	1
4	Larue	Altenwerth	minnie.gislason@gmail.com	20/05/2020	Myanmar	13	Mid-Level	Social Media Community Management	9	7
...	...	...	...	...	...	...	...	...	...	...
49995	Bethany	Shields	rocky_mitchell@hotmail.com	09/01/2022	Dominican Republic	27	Trainee	Security	2	1
49996	Era	Swaniawski	dolores.roob@hotmail.com	02/06/2020	Morocco	21	Lead	Game Development	1	2
49997	Martin	Lakin	savanah.stracke@gmail.com	15/12/2018	Uganda	20	Trainee	System Administration	6	1
49998	Aliya	Abernathy	vivienne.fritsch@yahoo.com	30/05/2020	Czech Republic	20	Senior	Database Administration	0	0
49999	Coleman	Wisozk	abigayle.crooks@yahoo.com	13/06/2022	Palau	15	Intern	Mulesoft	3	1

50000 rows × 10 columns

Data profiling

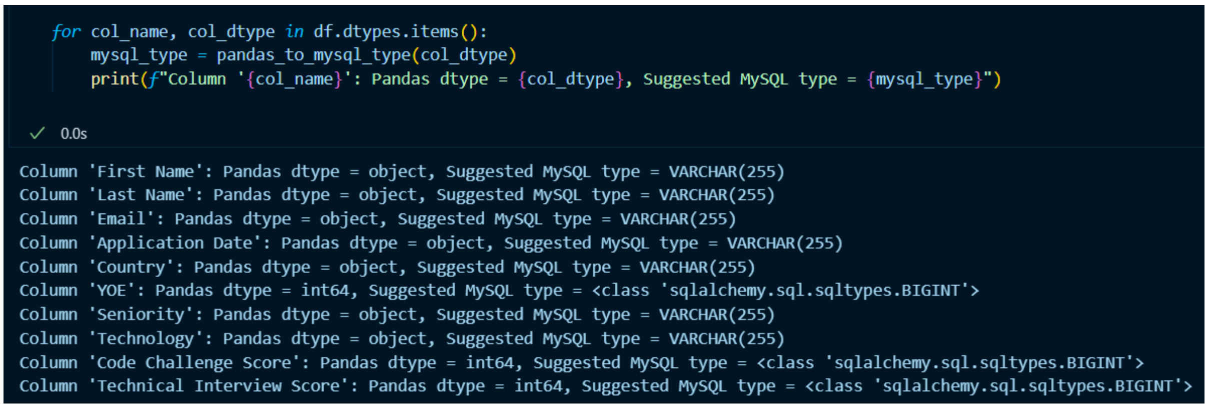
Data profiling is an invaluable step in the data preparation process. While it doesn't fully automate data type mapping, it provides crucial insights needed to make informed decisions and create a well-designed database schema.

Profiling data types

The code used in this step defines a function `pandas_to_mysql_type` that suggests appropriate MySQL data types based on the data types of Pandas Series (columns). It iterates through the columns of a Pandas DataFrame (df), determines the Pandas dtype, and uses a series of conditional checks to map these to corresponding SQLAlchemy types (which can be used to define MySQL columns).

> The function handles integer, floating-point, datetime, boolean, categorical, and string types, providing a suggested MySQL type for each.

The results are suggestions and should be reviewed and adjusted based on the specific data and application requirements.



```
    for col_name, col_dtype in df.dtypes.items():
        mysql_type = pandas_to_mysql_type(col_dtype)
        print(f"Column '{col_name}': Pandas dtype = {col_dtype}, Suggested MySQL type = {mysql_type}")
```

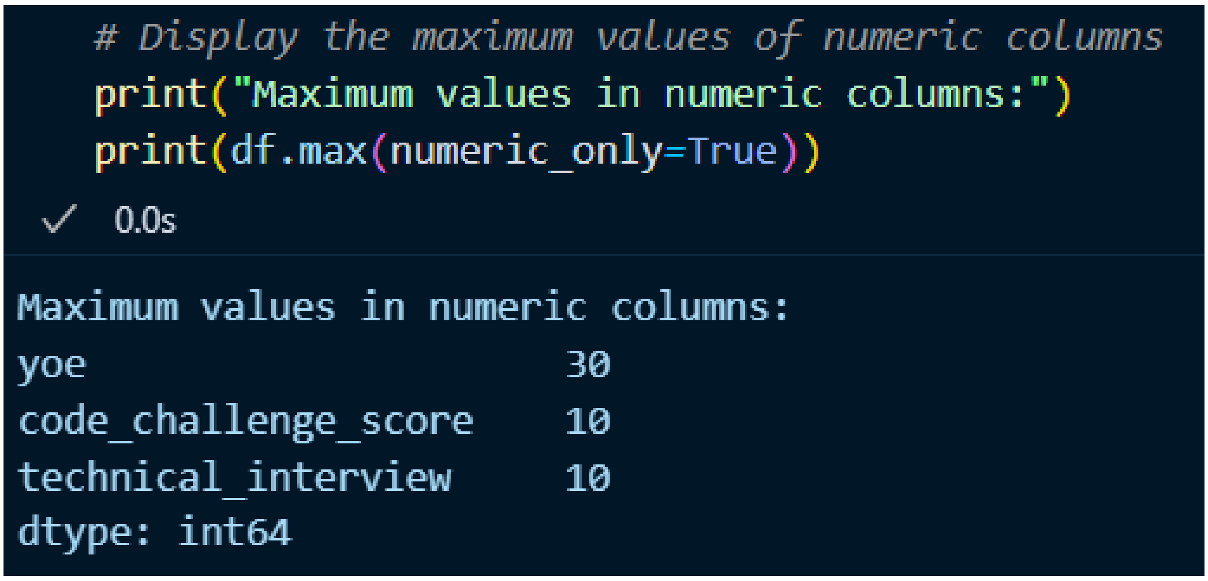
```
✓ 0.0s
```

```
Column 'First Name': Pandas dtype = object, Suggested MySQL type = VARCHAR(255)
Column 'Last Name': Pandas dtype = object, Suggested MySQL type = VARCHAR(255)
Column 'Email': Pandas dtype = object, Suggested MySQL type = VARCHAR(255)
Column 'Application Date': Pandas dtype = object, Suggested MySQL type = VARCHAR(255)
Column 'Country': Pandas dtype = object, Suggested MySQL type = VARCHAR(255)
Column 'YOE': Pandas dtype = int64, Suggested MySQL type = <class 'sqlalchemy.sql.sqltypes.BIGINT'>
Column 'Seniority': Pandas dtype = object, Suggested MySQL type = VARCHAR(255)
Column 'Technology': Pandas dtype = object, Suggested MySQL type = VARCHAR(255)
Column 'Code Challenge Score': Pandas dtype = int64, Suggested MySQL type = <class 'sqlalchemy.sql.sqltypes.BIGINT'>
Column 'Technical Interview Score': Pandas dtype = int64, Suggested MySQL type = <class 'sqlalchemy.sql.sqltypes.BIGINT'>
```

Profiling the lenght of numerical values

The code used in this step uses the Pandas DataFrame df to display the maximum and minimum values for its numeric columns. `df.max(numeric_only=True)` calculates and prints the maximum value for each column that has a numeric data type. The numeric_only=True argument ensures that only numeric columns are considered, preventing errors if the DataFrame contains non-numeric data.

The resulting information is useful for understanding the range of values in the numeric data, which is crucial for choosing appropriate data types for database storage.



```
    # Display the maximum values of numeric columns
    print("Maximum values in numeric columns:")
    print(df.max(numeric_only=True))
```

```
✓ 0.0s
```

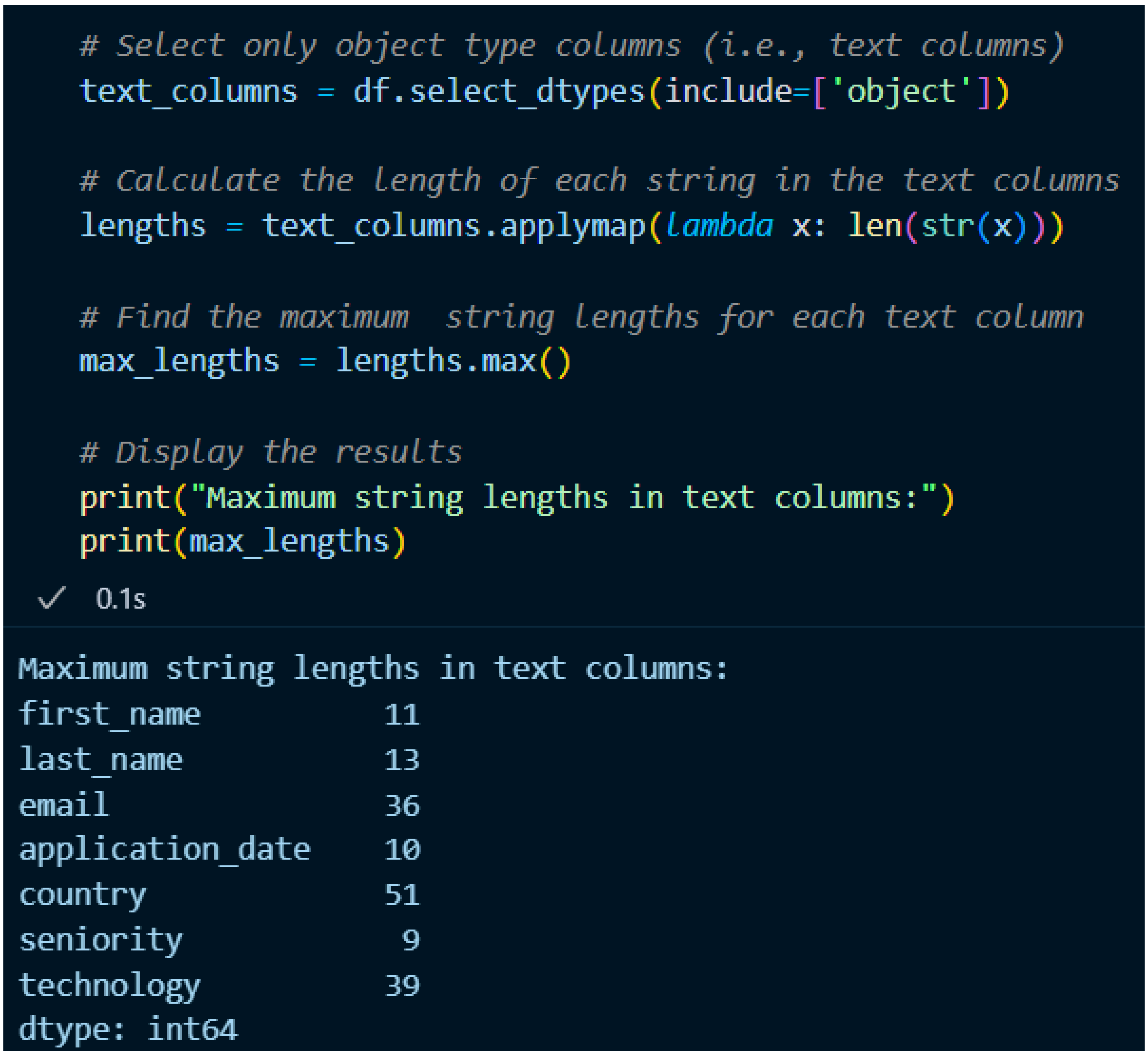
```
Maximum values in numeric columns:
yoe                     30
code_challenge_score    10
technical_interview     10
dtype: int64
```

Profiling the lenght of string values

The code used in this step analyzes the text (object type) columns in a Pandas DataFrame df to determine the maximum and minimum string lengths within each column. It first selects only the columns with a data type of 'object' (typically representing text) using `df.select_dtypes(include=['object'])`.

Then, it calculates the length of each string in these selected columns using `.apply(lambda col: col.map(lambda x: len(str(x))))`. This line applies a function to each column, which itself maps another function `(len(str(x)))` to every element within that column. The inner function converts each element to a string and then calculates its length. This results in a new DataFrame, `lengths`, containing the string lengths. Next, `lengths.max()` calculates the maximum string length for each column, storing the results in a Pandas Series called max_lengths. Finally, the code prints a descriptive label and the max_lengths Series, displaying the maximum string length found in each text column.

The resulting information is valuable for database design (choosing appropriate VARCHAR or TEXT sizes)



```python
# Select only object type columns (i.e., text columns)
text_columns = df.select_dtypes(include=['object'])

# Calculate the length of each string in the text columns
lengths = text_columns.applymap(lambda x: len(str(x)))

# Find the maximum  string lengths for each text column
max_lengths = lengths.max()

# Display the results
print("Maximum string lengths in text columns:")
print(max_lengths)
```

✓ 0.1s

```
Maximum string lengths in text columns:
first_name          11
last_name           13
email               36
application_date    10
country             51
seniority            9
technology          39
dtype: int64
```