International Journal of Management, IT & Engineering
Vol. 15 Issue 3, March 2025
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

Streamlining Data Loading in Python: A Guide for Beginners

Gajula Lokesh Kumar *

Abstract

In today's data-driven world, efficiently loading and managing various data sources is a crucial skill for aspiring data scientists and software developers. This paper provides a complete guide for beginners to streamline data loading processes in Python, utilizing popular libraries. With a focus on handling various data formats, including XLSX, XLS, CSV, TXT files load to the Oracle, SQL or any other databases, the paper targets to explain the difficulties involved in data management tasks. Practical examples and clear scripts are provided to equip readers with hands-on experience and foundational knowledge, enabling them to confidently address common data handling challenges. By using this resource, students and beginners will enhance their data literacy and Python programming skills, laying the foundation for further exploration and mastery in the realm of data science.

Keywords:

Python, Data Loading, CSV, XLSX, SQL, Oracle, Fixed width, Beginners, Data Handling, Scripting, Built-in Libraries, Data Formats

Author correspondence:

Gajula Lokesh Kumar,
Engineer Lead, Elevance Health Inc.
2015 Staples Mill Road, Richmond, VA, USA, 23230
Email: gajulalokeshkumar@gmail.com

1. Introduction

In today's data-centric world, the ability to efficiently load and manage diverse data sources is essential for those entering fields like data science and software development. Python, known for its cleanness and robust ecosystem, provides a multitude of built-in libraries that make this job accessible, even to beginners. However, beginners often face challenges when dealing with various data formats such as XLSX, XLS, CSV, TXT files loading those to the Oracle, SQL or any other databases.

This paper aims to offer a clear and short guide to streamline the process of loading these common data formats like Excel, Csv and Txt using Python's native capabilities. By leveraging Python's libraries, students and beginners can simplify data workflows and automations, allowing them to concentrate more on data analysis and application development.

The guide will walk readers through the fundamentals of each data format, introduce relevant Python libraries, and present practical examples. These examples will provide readers with the foundational understanding and confidence needed to handle more complex data tasks, preparing them for further exploration of Python's powerful data processing capabilities like NumPy, Pyspark and Matplotlib.

Through this resource, we aim to empower budding data enthusiasts and developers with essential data handling skills, paving the way for tackling more sophisticated data management and analytical endeavors.

Furthermore, this section explores the theoretical foundations of data formats and Python's methodologies for handling them. It begins by defining and contrasting the key characteristics of XLSX, XLS, CSV, TXT, and SQL databases, highlighting their unique features and typical use cases. Subsequently, the discussion introduces Python's standard libraries—openpyxl, xlrd, csv, and cx_Oracle—detailing their functionalities and demonstrating their efficient use for data loading and manipulation. This positions Python as a unified tool, efficiently integrating multiple data sources and emphasizing its adaptability.

2. Research Method

This research employs an organized approach to explain and analyze the capabilities of Python's libraries in handling various data formats and loading data into target database like Oracle or Sql. The method is organized into several key points:

* Engineer Lead, Elevance Health Inc. 2015 Staples Mill Road, Richmond, VA, USA, 23230

Data Source Identification

1. **File Formats:** We selected a variety of commonly used file formats XLSX, XLS, CSV, TXT, and fixed-width files to ensure broad applicability across different data handling scenarios. These are the most common type of file formats organization using now a days.
2. **Database Selection:** We chose Oracle as an example of an enterprise-level database; however, the methodology is designed to be adaptable to other databases such as MySQL, PostgreSQL, and SQL Server, to showcase Python's versatility.

Tool and Library Selection

1. **Python Libraries:**
    - For XLSX and XLS handling, we are using engine openpyxl and xlrd respectively to facilitate reading and writing operations on spreadsheet data to target database with help of pandas read_excel function.
    - The pandas read_csv module was used for loading CSV files, a standard library providing native support for structured text data.
    - Basic file I/O operations were utilized for TXT and fixed-width files, leveraging Python's built-in capabilities for text processing.
    - For the fixed width file processing we need provide widths of each field as an input parameter to process the fixed width file.
    - To handle those widths, we are taking the input file as parameter and reading the widths from the input file to the fwf_widths for further processing.
    - Please find the below code example for these cases.
    - We can leverage json module to parse and manipulate JSON data, demonstrating handling of hierarchical data structures.

2. **Database Library:**
    - The cx_Oracle module was used to connect and load data into an Oracle database, illustrating Python's integration with relational database systems.in addition to that we can connect to any data base using the python for example to connct to the my sql use the “mysql.connector”

Implementation

1. **Script Development:** We developed Python single script for all data formats using the simple if else block for the file type, emphasizing the use of native commands where possible to ensure accessibility for beginners. Key operations included reading data, data transformation, and loading data into a target database.

2. **Execution and Testing:**
    - Script was executed with datasets of varying sizes to evaluate performance, efficiency, and accuracy.
    - We imported the essential libraries needed to execute the code; only the core logic is provided in the sample code.
    - For database operations, various SQL queries were designed and executed, including data insertion, retrieval, and basic transformation processes.
    - For processing the fixedwidth file we are using the flag to determine whether we are processing fixedwidth or not.
    - If that flag is Y, then we need the fixed widths parm file needs to provide as input parm for futther processing.
    - Based on the file we can determine the file type using the simple commands.
    - Logging is the best way to find out if we have any issues with the code for easy readability, we loged every important message in the code.
    - Once we read the data from source file to the data frame and then converting that to the csv file using the field saparator as “~”. Then the stage file will be loading to the target database using the “subprocess.check” call with help of sqlldr for more details find the below flow diagram and code snippet.
    - Using this process we able to know the if any issues in the log files.

- When we are handling the large files, we made sure we have the correct processor with required memory.
- If you're planning to use this, try with small data sets first then use the bulk files.
- In this process we assumed target table as always kill and fill that means we will delete the existing data and inserting the new data. if your request needs to append the data, we can trun off that setting too.
- We have a system equipped with 8 CPU cores, 62 GB of RAM, and 17 GB of swap space.
- Mostly 1 MB file will take less then 1 min. but it will vary based on the system/cluster size.
- At the end of process, will make sure record count from the source and target are matching or not if matching then only we will set the Job to successful.
- Finally, will make sure close all open databse connections onces job completed.

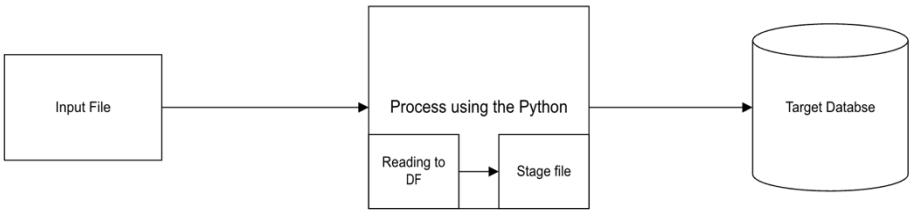

Figure 1. ETL *Flow*

Code Example: Data File Processing and Loading into Oracle Database

To process the data from different file formats (e.g., XLSX, XLS, CSV, TXT, and fixed width) and prepare it for loading into an Oracle database, we used below Python code snipet. This script demonstrates how each file type is read and preprocessed before being added to the database. It uses libraries such as pandas for data manipulation and cx_Oracle for database operations.

```
try:
    for files in glob.glob(FileName):
        file_nm = os.path.basename(files)
        file_load_flg = 1
        lg.info("Processing the file : {} to load into {}
table\n".format(file_nm, table_nm))
        Filetype = file_nm.split('.')[-1]
        if fwf_flg == 'Y':
            fwf_widths_pars()
        record_cnt = 0

        if (Filetype == 'xlsx' or Filetype == 'xlsm') and fwf_flg == 'N':
            lg.info("Source file type is : Excel with extenstion xlsx\n")
            read_input = pd.read_excel(files, dtype=str, index_col=None,
sheet_name=0, engine='openpyxl')
            csv_FileName = 'XX_' + file_nm.split('.')[0] + '.csv'
        elif Filetype == 'xls' and fwf_flg == 'N':
            lg.info("Source file type is : Excel with extenstion xls \n")
            read_input = pd.read_excel(files, dtype=str, index_col=None,
sheet_name=0, engine='xlrd')
            csv_FileName = 'YY_' + file_nm.split('.')[0] + '.csv'
        elif Filetype == 'csv' and fwf_flg == 'N':
            lg.info("Source file type is : CSV \n")
            read_input = pd.read_csv(files, dtype=str, index_col=None)
            csv_FileName = file_nm
        elif Filetype == 'txt' and fwf_flg == 'N':
            lg.info("Source file type is : txt \n")
            csv_FileName = file_nm
        elif fwf_flg == 'Y':
            lg.info("Source file type is : Fixed width \n")
```

```
            read_input = pd.read_fwf(files, colspecs=None, widths=widths,
dtype=str, header=None)
            csv_FileName = 'Fixed_' + file_nm.split('.')[0] + '.csv'
        else:
            lg.error("File format not supporting or not enough arguments provided
\n")

        if Filetype == 'txt':
            record_cnt = sum(1 for i in open(files, 'rb')) - 1
        else:
            read_input['file_nm'] = file_nm
            read_input.to_csv(FilePath + '/' + csv_FileName, sep="~",
index=False, encoding='iso-8859-1')
            record_cnt = read_input.shape[0]

        try:
            Cur.execute("Truncating table {} ".format(table_nm))
        except cx_Oracle.DatabaseError as e:
            if e.args[0].code == 942:
                lg.error("table: {} does not exists \n".format(table_nm))
            else:
                lg.error("Error Occured is: " + str(e) + "\n")
        lg.info("Table : {} is truncated \n".format(table_nm))

        Cntpath = ParmPath + '/' + ctl_nm
        Lgfpath = "{}//{}_{}.log".format(LgPath, table_nm,
datetime.datetime.now().strftime("%Y%m%d%H%M%S"))
        Badfpath = "{}//{}_{}.bad".format(LgPath, table_nm,
datetime.datetime.now().strftime("%Y%m%d%H%M%S"))

        Userfmt = "/@" + TargetDatabase
        lg.info("Started to load the data into the target table\n")

        try:
            outputfunc = subprocess.check_call(
                "sqlldr control='{}', log='{}', bad='{}', data='{}', userid='{}',
errors=0".format(Cntpath, Lgfpath,

Badfpath,

FilePath + '/' + csv_FileName,

Userfmt), shell=True)
        except subprocess.CalledProcessError as e:
            outputfunc = str(e.output)
        lg.info("The output is: {}\n".format(outputfunc))

        if outputfunc == 0:
            lg.info("Data loaded to the table Successfully\n")
            shutil.move(FilePath + "//" + file_nm, ArchivePath + "//" + file_nm)
            if os.path.exists(FilePath + '/' + csv_FileName):
                shutil.move(FilePath + "//" + csv_FileName, ArchivePath + "//" +
csv_FileName)
        else:
            lg.error("The Data load was UnSuccessful")
            lg.info("Data load was UnSuccessful: {} \n".format(outputfunc))

        query1 = "select count(*) As table_count from " + table_nm;
        Count_df = pd.read_sql(query1, conn, index_col=None)
        rowcount = int(Count_df['table_count'].values)

        if int(record_cnt) == int(rowcount):
            print("The no of records in the file and the no of records loaded to
the table are matching\n")
            lg.info("The no of records in the file and the no of records loaded
to the table are matching\n")
        else:
            lg.info("The no of records in the file and the no of records loaded
```

```
to the table are not matching\n")

        if file_load_flg == 0:
            lg.error("The Data load was UnSuccessful")
            lg.info("All the files missing to load. So exiting...\n")
        lg.info("Table Load completed Successfully ----------*******----")

except Exception as e:
    lg.error("Error Occured is: " + str(e) + "\n")

finally:
    Cur.close()
    conn.close()
    lg.shutdown()
```

For fixed width file processing we need to align columns accordingly and pass to our script for better results:

```
#Function to parse fixed width file
def fwf_widths_pars():
    print("fwf_flg mode set to Y")
    #Reading the fwf_widths file for formating the src file.
    global widths
    widths = []
    fwf_widths = ParmPath+"//"+ctl_nm.replace('.ctl','_fwf_widths.ctl')
    try:
        with open(fwf_widths, "r") as fwf_widths_file:
            for line in fwf_widths_file:
                cleaned_line = line.strip()
                if cleaned_line:
                    widths.append(cleaned_line)
    except FileNotFoundError:
        print(f"The file {fwf_widths} was not found.")
    widths = ''.join(widths)
    try:
        widths = eval(widths)
    except SyntaxError:
        print("Syntax error in the data.")
    except Exception as e:
        print(f"An error occurred: {e}")
    #print (widths)
    return widths
```

Evaluation Criteria

1. **Performance Metrics:** Execution time and resource usage were recorded to assess efficiency across different data formats and database operations.
2. **Usability Assessment:** Ease of implementation and clarity of code for educational purposes were evaluated, focusing on the script's readability and simplicity for beginners.
3. **Challenges Identification:** We documented challenges such as handling large files or complex data transformations to propose improvements or alternatives, assessing the limitations of basic libraries.

Data Integration Strategies

We demonstrated potential integration scenarios by relating data from multiple formats into unified structures, exploring the initial steps toward using advanced tools like NumPy, Pyspark and Matplotlib for data manipulation and analysis. This phase assessed how well Python's basic tools prepared users for complex data workflows.

Future Encourage Exploration

Finally, the research approach encourages future evaluation of automation and scalability and highlights importances of exploring deeper knowledge of python's capabilities with other technologies, such as cloud-based solutions and big data frameworks.

This includes leveraging cloud-based solutions for greater flexibility and performance, as well as implementing in big data frameworks to handle and process large-scale data more efficiently.

3. Results and Analysis

This section outlines the outcomes of implementing Python scripts to manage various data formats and highlights the practical implications for data handling, focusing on CSV, XLSX, and TXT files to load into databases, as well as specific formats fixed-width files.

Data Loading and Processing

- **XLSX and XLS Handling:** Utilizing openpyxl and xlrd, we efficiently read spreadsheet data across multiple sheets, demonstrating Python's ability to handle tabular data typical in business and academic contexts. This versatility makes Python an excellent tool for beginners exploring structured data processing.
- **CSV File Flexibility:** Our implementation using Python's csv module revealed robust performance in reading and writing operations, effectively handling complex file structures with various delimiter settings. This module aids beginners in understanding important data parsing concepts.
- **TXT and Fixed-Width File Processing:** Python's basic I/O functions (open, readline) and string manipulation effectively processed plain text and fixed-width files. This flexibility illustrates Python's capacity to manage less-structured text data, encouraging the development of problem-solving skills essential for different file types.
- **JSON Data Handling:** The json module's support for loading nested JSON structures highlights its strength in managing hierarchical data, commonly used in web development and API interactions, indicating both memory efficiency and ease of integration with other tools.

Database Interfacing

In executing data loads to an Oracle database using the "cx_Oracle" module, we demonstrated seamless interfacing with enterprise-level systems. This involved executing SQL queries and inserting data from various source files (XLSX, CSV, TXT), underscoring Python's adaptability to different database environments. Python's database API further simplifies integrating with other SQL-based systems such as MySQL, PostgreSQL, and SQL Server, confirming its utility across multiple enterprise scenarios.

Discussion

These findings affirm Python's versatility in handling diverse data formats, making it a preferred choice in domains like education, business analytics, and research:

- **Ease of Use:** Python's in-built syntax reduces the learning curve, empowering beginners to transition from simple to complex data tasks with confidence, preparing them for advanced library usage.
- **Integration and Future Work:** Combining data sources and transitioning to libraries like pandas for sophisticated analyses signifies the potential for extended applications. Future research could explore cloud-based integrations or automation scripts, further expansion of Python's applicability in data-intensive projects.
- **Educational and Practical Implications:** The practice highlights initial skills in data processing, crucial for connecting introductory learning with real-world applications in software development and data science. Challenges identified include managing large datasets or complex file formats, encouraging exploration of additional tools like Pandas, Dask, or SQLAlchemy.
- **Performance Considerations and Challenges:** While effective for basic tasks, handling very large or non-standard data may require more specialized tools. Streamlining this process for educational purposes underlines Python's value in teaching data literacy and software engineering.

In conclusion, Python's robust feature set for database interfacing and file handling positions it as a comprehensive solution for diverse data management requirements, empowering users to effectively engage with dynamic data-driven environments.

4. Conclusion

The exploration into Python's capabilities for handling various file formats—XLSX, XLS, CSV, TXT, and fixed-width files—demonstrates the language's versatility and effectiveness in streamlining data management processes for beginners. By utilizing Python's built-in and standard libraries, this research provides a solid foundation for understanding and executing essential data tasks, proving particularly beneficial for educational settings where students and newcomers are introduced to programming and data handling.

The successful implementation of data extraction and manipulation techniques across these formats underscores Python's adaptability to different data types. Each format presents unique challenges, yet Python's extensive ecosystem offers suitable tools to navigate these complexities efficiently. The ability to process spreadsheet files using openpyxl and xlrd, handle CSV files through the csv module, and manage text data with basic I/O operations showcases a comprehensive approach for addressing diverse data requirements.

Moreover, the study affirms the importance of building foundational data handling skills, which can significantly enhance a learner's capability to transition into using more advanced data analysis libraries such as pandas, thereby expanding analytical possibilities. By equipping learners with the skills to effectively load and manipulate data, they are better prepared for tackling complex real-world data challenges, from small-scale educational projects to larger, industry-oriented applications.

This paper opens avenues for further research into optimizing data handling processes in Python, including performance improvements for larger datasets and integration with modern data technologies like cloud computing and big data platforms. Future work should also consider exploring comparative analyses with other programming languages and libraries to provide a broader perspective on the efficiency and scalability of data management tools.

In summary, the study demonstrates that Python's tools are not only accessible and efficient but also crucial for fostering an understanding of essential data operations among beginners. By laying the groundwork for future exploration and skill development, this research contributes to the growing body of knowledge aimed at enhancing data literacy and promoting innovation in data-driven fields.

References

[1] **Python Software Foundation. (n.d.).** "csv — CSV File Reading and Writing." Python 3.x Documentation. Available at: https://docs.python.org/3/library/csv.html.

[2] **Python Software Foundation. (n.d.).** "openpyxl — Read/Write Excel 2010 xlsx/xlsm files." Python Package Index (PyPI). Available at: https://pypi.org/project/openpyxl/.

[3] **Python Software Foundation. (n.d.).** "xlrd — Python library for reading data from Excel files (xls)." Available at: https://pypi.org/project/xlrd/.

[4] **Python Software Foundation. (n.d.).** "pandas — Powerful data structures for data analysis, time series, and statistics." Python Package Index (PyPI). Available at: https://pypi.org/project/pandas/.

[5] **Python Software Foundation. (n.d.).** "cx_Oracle — Python interface to Oracle Database." Python Package Index (PyPI). Available at: https://pypi.org/project/cx-Oracle/.