**Report on the Development of a Data Processing Application for Digital Audio Broadcasting (DAB) Stations**

Advanced Programming: COM00142M

John Philip

August 2023

A logo with a black and green gradient

Description automatically generated

Table of Contents

[Table of Contents 2](#_Toc143466931)

[Introduction: 3](#_Toc143466932)

[Section 1: Theory 3](#_Toc143466933)

[1A. Concurrency 3](#_Toc143466934)

[1B. GUI Interface 4](#_Toc143466935)

[1C. Java vs Python for ‘manipulation of data containers’ 5](#_Toc143466936)

[Section 2: Design Decisions 6](#_Toc143466937)

[2A. JSON Data Format and Reasoning (2nd Requirement) 7](#_Toc143466938)

[2B. Calculating 3rd Requirement: Mean/Mode/Median 7](#_Toc143466939)

[2C. Client Visualisations (4th Requirement) 8](#_Toc143466940)

[2D. Data Correlation (5th Requirement) 9](#_Toc143466941)

[Section 3: Reflection on the ethical, moral, and legal aspects of computing: 10](#_Toc143466942)

[References: 12](#_Toc143466943)

[Appendix 14](#_Toc143466944)

[Section **1**A 14](#_Toc143466945)

[‘Chardet’ 14](#_Toc143466946)

[‘Overview Diagram’ 14](#_Toc143466947)

[‘Threading diagram’ 15](#_Toc143466948)

[Section 1B 16](#_Toc143466949)

[‘Tkinter Styling’ 16](#_Toc143466950)

[‘Gui Layout’ 16](#_Toc143466951)

[‘LOAD CODE’ 16](#_Toc143466952)

[‘Try Except Button’ 17](#_Toc143466953)

[Tkinter back-up data 17](#_Toc143466954)

[Section **1**C 18](#_Toc143466955)

[‘Dynamic Flexibility in Python’ 18](#_Toc143466956)

[‘Pandas, NumPy, SciPy, and Matplotlib’ 18](#_Toc143466957)

[‘Renaming Imports’ 18](#_Toc143466958)

[‘Lambda Function Example’ 18](#_Toc143466959)

[Section 2A 19](#_Toc143466960)

[‘Data Cleaning & Extracting’ 19](#_Toc143466961)

[Create new EID columns 19](#_Toc143466962)

[‘Join JSON GUI tabs’ 19](#_Toc143466963)

[‘JSON convert, save, & backup’ 19](#_Toc143466964)

[‘JSON DATA FORMAT FULL DIAGRAM’ 20](#_Toc143466965)

[Section 2B 21](#_Toc143466966)

[‘Pandas for Mean/Mode/Median’ 21](#_Toc143466967)

[‘Global Variable Checks’ 21](#_Toc143466968)

[Section 2C 23](#_Toc143466969)

[‘Tkinter Code’ 23](#_Toc143466970)

[‘Facetgrid’ 23](#_Toc143466971)

[Section 2D 25](#_Toc143466972)

[‘p, c, dof, expected’ for Chi-square 25](#_Toc143466973)

[‘Detailed Results for Chi-Square test’ 26](#_Toc143466974)

# Introduction:

This report discusses the creation and execution of a Python application designed to handle data manipulation tasks for datasets related to digital audio broadcasting (DAB) signals. Developed within an Anaconda environment in a Jupyter Notebook, the application utilises a graphical user interface (GUI) to manage the processes of loading, transforming, cleaning, and statistically analysing CSV data. The application performs most of these tasks sequentially, with each operation following the previous one.

The report is divided into three sections, corresponding to the different aspects of the software development process and report guidelines:

* Section 1: Theory and application of programming techniques
* Section 2: Design decisions for the selection, storage, and manipulation of data
* Section 3: Evaluation of software developments' legal and ethical impact within real-world contexts.

# Section 1: Theory

## 1A. Concurrency

Python is one of the most widely used and effective tools for building analytical applications and systems [1]. Like all languages, it has several distinct advantages and disadvantages. Python can be categorised as a high-level interpreted language because the CPU does not directly execute code; instead, it is run by an interpreter. Python’s interpreted nature can lead to challenges in building efficient, highly concurrent, multithreaded applications [1].  Concurrency in programming involves executing multiple tasks or processes simultaneously and leveraging the benefits of multicore processors. Python’s interpreted nature, and its global interpreter lock (GIL), usually prevent the interpreter from executing more than one Python instruction at a time [1, p.20]. The distinctions and categorisations between programming languages are not rigid, however, and there are techniques to circumvent the GIL and develop concurrent multithreaded applications in Python [1],[2]. These techniques include Python threading, Python C extensions that use native multithreading, and other APIs such as NumPy and Twisted that use concurrency in their operation [1, p. 20]. Multithreading can increase speed up to n-times (nx) with n cores, resulting in a quad-core machine with all four cores running in a quarter of the original time [2, p.531]. While concurrency can enhance the performance of Python applications, it also introduces complexity and the risk of errors. Each additional threaded process will increase the complexity and communication overhead while decreasing the available RAM [2]. Therefore, it is crucial to carefully consider which application elements might be enhanced by implementing concurrency.

In the DAB data analysis application developed for this paper, a few functions and applications stand out as candidates for concurrency. The strongest would be for the ‘load\_data()’ function. Neither of the provided datasets was encoded using Unicode, and my application required the additional step of scanning the data for encoding using the Chardet API to load them properly using the encodings ISO-8859-1 and ASCII[[[1]](#footnote-1)](#_‘Chardet’). Evaluating the encoding types for many datasets could require a significant amount of time and resources. Depending on the size, these encoding functions could be processed concurrently using Python’s threading modules to help analyse the data more efficiently. The clean\_data() function could also see greater efficiency, mainly if used in conjunction with a multithreaded load\_data() function, especially for a large number of datasets. It is not uncommon for real-world datasets to be fraught with issues that need to be addressed becoming relatively resource-intensive processes. These could include missing values, inconsistencies in data formats, or even erroneous entries [1]. Running these tasks concurrently can reduce the time taken, especially when dealing with many datasets [2]. An overview of my application with the clean\_data() and load\_data() functions running concurrently is illustrated in the ‘Overview Diagram’ & ‘Threaded Diagram’ in Appendix 1A[[[2]](#footnote-2)](#_‘Overview_Diagram’).

## 1B. GUI Interface

Tkinter was the selected framework for implementing the GUI in the data analysis program. It is a stable and well-developed library with years of development and bug fixes since being added to the standard Python Library in 1994 [3]. Tkinter requires no additional steps to create virtual environments, compile binaries, install pip, or search the web for installation projects [3]. Tkinter's simple and straightforward graphics might be the wrong choice for a game UI or slick commercial application, however, for data-driven applications Tkinter excels. It additionally offers a range of packages such as ‘ttk’ that allow themed widget sets to make the GUI more visually pleasing, accessible, and user-friendly [3]. For my program, I incorporated a more modern theme and additional styling widgets to help set the font, colour and size for the different buttons and better organise their layout for the user[[[3]](#footnote-3)](#_‘Tkinter_Styling’).

The GUI layout in the application includes a root window, a notebook tab manager, buttons, and a text box[[[4]](#footnote-4)](#_‘Gui_Layout’). These were chosen for ease of use and intuitive interactions with the user. A series of clearly labelled buttons allow interaction with the data and execution of application functions. These are directly linked to the specific modular functions requested by the client. For example, the Load button and function meet the client's requirement to load either an initial CSV or the prepared JSON dataset. It also allows handling any number of initial datasets[[[5]](#footnote-5)](#_‘LOAD_CODE’). The buttons also work to handle errors more efficiently using a try-except block[[[6]](#footnote-6)](#_‘Try_Except_Button’).

A text box at the bottom of the GUI provides logging and real-time feedback to the user for all button presses, including errors and executions. Effective feedback is essential in user-interface (UI) design as it communicates status, keeps users oriented, and builds their confidence in the task [4]. Additional feedback includes relevant headers for the tabs in the notebook and the data visualisations themselves[[[7]](#footnote-7)](#_‘Overview_Diagram’). The notebook tab manager helps to effectively organise the outputs and visualisations in a similar format to what people are used to when browsing the internet [3].

To address the client requirement for data backup in a suitable format, two methods were employed. Firstly, the ‘save\_data()’ function can be used at any point to save and convert the data to json. Additionally, the ‘create\_backup’ was integrated with Tkinter’s ‘filedialog’ module for saving the process of the user when closing the program[[[8]](#footnote-8)](#_Tkinter_back-up_data).

## 1C. Java vs Python for ‘manipulation of data containers’

Java and Python are widely used high-level programming languages, offering a myriad of tools for manipulating data containers. One of Java’s advantages includes the superior speed of its compiled nature. Additionally, while Java and Python employ automatic memory management, Java is better performing in memory allocation and garbage collection [5, p.12].

Python, however, stands out in terms of readability, versatile data structures, and tools for data analysis. While at a disadvantage for speed at compiling time, Python’s dynamic typing offers flexibility for managing variables, functions, and data containers [1]. Python lists can hold different data types, such as int, string, and float, within the same data structure [1]. Python lists also benefit from added tools, such as negative index values and slice syntax, allowing users to extract ranges of elements from lists [5]. Examples of Python’s dynamic flexibility within the DAB application include[[[9]](#footnote-9)](#_‘Dynamic_Flexibility_in):

* Flexible load\_data() function where the type of file is based on the extension ‘.json’ or ‘.csv’
* ‘Duck Typing’, where the clean\_data() function assumes the data has certain columns and doesn’t check for types but rather for columns names
* Easily cleaning whitespaces on all data types as a generic operation

This flexibility contrasts with Java, which as a statically typed language, generally confines a data structure to one specific type. Python further excels in data containers through libraries and frameworks tailored for data manipulation such as the Pandas library. The Pandas dataframe, a flexible and efficient 2D table structure, was integral to the developed DAB data analysis program. The application relies on Pandas throughout for loading, cleaning, analysing, and manipulating data. Python-specific tools like NumPy, SciPy, and Matplotlib were used for data structuring and visualisation[[[10]](#footnote-10)](#_‘Pandas,_NumPy,_SciPy,).

Finally, Python is well-known for its user-friendly syntax and can often be considered more efficient in terms of “programmer time” rather than “CPU time” [1, p.20]. Unlike Java, Python allows for library renaming during importation, *‘import pandas as pd’*, reducing coding efforts and preventing namespace conflicts[[[11]](#footnote-11)](#_‘Renaming_Imports’). This ability to use shorthand (e.g., pandas as pd) circumvents the need to use full package/module hierarchy for disambiguation, as required in Java [5, p.45]. Python also offers support for lambda functions, which are a way of writing short functions consisting of a single statement, the result of which is the return value [1]. Used throughout my DAB program, one example of the lambda function is within the cleaning function where it is used to remove trailing and leading whitespaces from dataframe columns[[[12]](#footnote-12)](#_‘Lambda_Function_Example’).

# Section 2: Design Decisions

## 2A. JSON Data Format and Reasoning (2nd Requirement)

A JSON (JavaScript Object Notation) format was selected as it provides greater flexibility and is more adept for hierarchical structures than tabular formats like CSV [1]. The data was first cleaned for whitespace handling, data conversion, data formatting, null value treatment, column name standardisation, and client specified data removal[[[13]](#footnote-13)](#_‘Data_Cleaning’). Users can subsequently extract desired DABs using the 'Extract' button, with the software creating new columns named after the selected DABs and marking them as '1' for a match or '0' otherwise[[[14]](#footnote-14)](#_Create_new_EID). The ‘JOIN JSON’ button in the GUI can then filter the data to satisfy the client's 2nd specification:

* *“join each category, C18A, C18F, C188 to the ‘ NGR’ that signifies the DAB stations location to the following: ‘Site’, ‘Site Height, In-Use Ae Ht, In-Use ERP Total”*

The function generates a three-tier JSON dictionary using the 'EID' and 'NGR' columns, displays the structure in a GUI tab, and offers the option to save it as a JSON file[[[15]](#footnote-15)](#_‘JSON_DATA_FORMAT) [[[16]](#footnote-16)](#_‘Join_JSON_GUI). Separately, the save\_file() function allows users to convert and save the dataframe as a JSON at anytime. Moreover, when exiting the application, the on\_closing() and create\_backup() functions also facilitate saving in the JSON format[[[17]](#footnote-17)](#_‘JSON_convert,_save,).

JSON aligns well with Python as it is very nearly valid Python code barring minor differences such as null values and other nuances [1]. Pandas efficiently processes JSON objects, seamlessly translating them into Series or DataFrame structures. My application imported pandas and json libraries to handle the different JSON functionalities[[[18]](#footnote-18)](#_‘JSON_convert,_save,).

XML (eXtensible Markup Language) would have been a practical choice as it has unique advantages such as metadata and namespace support, which are unavailable in JSON [1]. Ultimately, the client's specifications influenced the choice to utilise JSON, which didn't call for metadata.

## 2B. Calculating 3rd Requirement: Mean/Mode/Median

The ‘calculate\_stats\_and\_preview()’ function was used to fulfil the client's third requirement. The function produces two sets of statistical results based on the 'POWER(KW)' using two different filters: ‘Site Height’ and ‘Date’. The function begins by checking the global variables for data loading, cleaning, and extracting[[[19]](#footnote-19)](#_‘Pandas_for_Mean/Mode/Median’). This ensures the data is prepared, and the specified DAB multiplexes have been extracted before the statistics can be calculated. Specific to this task, the ‘Date’ column values were converted during cleaning into a datetime format for filtering[[[20]](#footnote-20)](#_‘Data_Cleaning’). The cleaned, loaded and extracted dataframe is then split into two temporary dataframes with each filtered by ‘Site Height’ and ‘Date’.

The calculations for the mean, median, and mode for the  'POWER(KW)' columns are performed using the Pandas library for statistical computations[[[21]](#footnote-21)](#_‘Pandas_for_Mean/Mode/Median’). Pandas was chosen over other libraries, such as NumPy or SciPy because of its direct integration with the pandas dataframes [1]. When the function is executed using the calculate button, a new tab is generated in the GUI titled 'Mean, Median, and Mode for POWER(kw)'. This tab displays the calculated statistics for each representation and allows users to access and review the results quickly. The function is also encapsulated within a try-except block. This structure enables the user to receive feedback if any exceptions arise during the execution (missing data or data formats), and the user is informed of errors and task completion through the dialog box in the GUI.

## 2C. Client Visualisations (4th Requirement)

In response to the client's visualisation needs, I employed the Seaborn API with Matplotlib's Figure capabilities for plot dimensioning. Additionally, I used FigureCanvasTkAgg to integrate the visualisation into the Tkinter GUI[[[22]](#footnote-22)](#_‘Tkinter_Code’). While several APIs like Bokeh and Plotly offer enhanced interactive visualisations, I opted for Seaborn and Matplotlib primarily because of their seamless integration with pandas [1, p. 472]. My initial standalone script relied on Plotly, but I encountered challenges when attempting to embed it within the GUI. Transitioning to Seaborn, a high-level statistical graphics library known for its adeptness at typical visualisation types [1], resolved these challenges.

The main task was to represent multiple labels ['FREQ', 'SITE', 'BLOCK', 'SERV LABEL1', 'SERV LABEL2', 'SERV LABEL3', 'SERV LABEL4', 'SERV LABEL10'] for distinct EIDs ['C18A', 'C18F', 'C188'] within a singular visualisation. My first strategy was to dedicate a new tab for each label and chart them against each EID using bar charts. This approach effectively utilized colour coding to denote overlapping values across EIDs and incorporated a legend for clear EID identification. However, the client's preference for a single representation, “a suitable graph", led me to explore a consolidated visualisation. I found Seaborn's FacetGrid ideal, as it's tailored for visualising datasets with numerous categorical variables [1, p. 504][[[23]](#footnote-23)](#_‘Facetgrid’). It compactly integrated my prior multi-tab visualisation into one chart. Nonetheless, a trade-off was the complexity of integrating distinct colours or legends into the FacetGrid, which posed a challenge.

## 2D. Data Correlation (5th Requirement)

To investigate potential correlations among the variables a chi-square test of independence was chosen. While 'Freq' is a numerical variable it seems to denote categorical frequency bands. The other columns denote string values which classify distinct radio tower locations and their corresponding radio stations. As these variables seem to represent categorical data, the chi-square test was deemed appropriate due to its proficiency in gauging associations between categorical variables [6, p 61].

The calc\_chi\_square() function begins by checking global variables for error monitoring[[[24]](#footnote-24)](#_‘Global_Variable_Checks’). This ensures that the data has been appropriately loaded, cleaned, and extracted. The function then utilizes the chi2\_contingency method from the Scipy scipy.stats module to execute the Chi-squared test of independence. Within the function, the line ‘*c, p, dof, expected = chi2\_contingency(crosstab)*’ loops through a list of categories, assigning values for: c, p, dof, and expected[[[25]](#footnote-25)](#_‘p,_c,_dof,). Using the widely accepted p-value threshold of 0.05 [6, p.61], the function filters the results of the Chi-squared test to highlight correlations between specific pairs. The pairs demonstrating correlation are listed below:

A screen shot of a computer

Description automatically generated

The strength of these correlations is further reinforced by high chi-square values and expected frequencies above five [6, p. 80] [[[26]](#footnote-26)](#_‘Detailed_Results_for). The relationship between pairs is also evidenced by their consistent one-to-one matches. For instance, all 229.072 'FREQ' values corresponded with the 12D 'BLOCK' value, while the 220.352 'FREQ' values coincided with the '11C' 'BLOCK' value. Pairs involving the 'SITE' variable didn't exhibit correlation, likely due to the distinctiveness of each 'SITE' entry. This is evidenced by their larger ‘degrees of freedom’ values[[[27]](#footnote-27)](#_‘Detailed_Results_for). A list of pair without significance listed below:

A black text on a white background

Description automatically generated

Most intriguing were the correlations observed between the insignificant pairs 'FREQ'/'BLOCK' and 'SERV LABELS 4 & 10'. Their elevated p-values, combined with the observed variable matches in values between pairs, suggest a potential overlap in the framework of DAB broadcasts.

# Section 3: Reflection on the ethical, moral, and legal aspects of computing:

*“Software engineers should not be subject to regulation by an ethical framework as this would have a detrimental impact on innovation. It would also add needless bureaucracy around the development of essential security updates and patches thus putting organisations and their data at risk.”*

In America, the saying "Guns don’t kill, people kill" is often used to argue against firearm regulations, despite rising gun violence and mass shootings. A parallel argument can be applied to technology by claiming that it is neutral in its essence. A formal proposition of this idea comes in the value-neutrality thesis (VNT), which states that technology, much like guns, is neutral, lacking inherent values [7].

From this perspective, one could argue that software engineers, like gun manufacturers, shouldn't be tied down by ethics. Such a mindset suggests that ethics only deter innovation and strain resources. A poignant real-world example of how we judge the user can be seen in the outcome of the 'Dieselgate' scandal. 'Bosch', which engineered the deceptive software, received a lighter fine than VW, which implemented it unethically [8][9].

However, a broader societal lens challenges this viewpoint. With over 44,000 gun-related deaths in the U.S. in 2022 alone [10], can we claim guns or software are simply neutral tools? Drawing parallels, pharmaceutical companies, while creating life-saving drugs, are held accountable for potential misuse and side effects [11]. Why shouldn't the same accountability extend to software engineers? Brushing aside ethics in the relentless pursuit of innovation means placing progress above societal health.

Paul Virilio poignantly noted: "When you invent the ship, you also invent the shipwreck"[12], highlighting the unforeseen negative repercussions of unchecked advancements. Dismissing ethical guidelines as mere 'bureaucracy' is both narrow-minded and potentially harmful. Software has profound societal implications: from gamification techniques spurring addiction, rampant data theft compromising personal security, and ingrained algorithmic biases perpetuating societal inequalities [12]. Misinformation, a by-product of unchecked algorithms, shapes public perception and polarises societies while posing enormous challenges to democracies around the globe. One example in the USA in 2010 occurred where a special election was held to fill a vacant US Senate seat. Someone supporting the Republican candidate created nine fake Twitter accounts and used them to send 979 tweets to 573 users in about two hours [13, p. 214]. These tweets ultimately reached 61,732 Twitter users, each with a link to a website with misinformation about the Democratic candidate. During this time period anyone Google searching the name of the Democratic candidate would find a top result showing the URL of the web page with the misinformation.

Therefore, the real question isn't about burdening software engineers with ethics. Instead, it's pondering the potential societal fallout if they aren't. In our intertwined digital-physical reality, the consequences of side-lining ethics are too significant to ignore.

# References:

[1] W. McKinney, “Python for Data Analysis”, 3rd ed. O'Reilly, 2017.

[2] M. Gorelick and I. Ozsvald, "High Performance Python, 2nd Edition," O'Reilly Media, Inc., April 2020. ISBN: 9781492055020.

[3] ​​A. D. Moore, "Python GUI Programming with Tkinter," Packt Publishing, May 2018, ISBN: 9781788835886.

[4] E. N. McKay, "UI is Communication," Morgan Kaufmann, 2013. ISBN: 978-0-12-396980-4

[5] D. Sarda, "Python for the Busy Java Developer: The Language, Syntax, and Ecosystem," 1st ed., Apress, Berkeley, CA, 2017, pp. XI, 72, ISBN: 978-1-4842-3233-0 (Softcover), eBook ISBN: 978-1-4842-3234-7.

[6] Y. Dodge, "The Concise Encyclopedia of Statistics," Springer, New York, NY, 2008. [Online]. Available: <https://doi.org/10.1007/978-0-387-32833-1>

[7] P. Kroes and P.-P. Verbeek, Eds., The Moral Status of Technical Artefacts, Philosophy of Engineering and Technology series. Springer Dordrecht, 1st ed., 2014. [Online]. Available: <https://doi.org/10.1007/978-94-007-7914-3>

[8] "Volkswagen emissions scandal," Wikipedia. [Online]. Available: <https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal>.

[9] D. Levin, "German Parts Maker Bosch Gets Off With Relatively Light $100 Million Fine From VW Dieselgate," Forbes, May 23, 2019. [Online]. Available: <https://www.forbes.com/sites/doronlevin/2019/05/23/german-parts-maker-bosch-gets-off-with-relatively-light-100-million-fine-from-vw-dieselgate/>

[10] Gun Violence Archive Data, Gun Violence Archive, [Online]. Available: <https://www.gunviolencearchive.org/past-tolls>.

[11] European Commission. (n.d.). Legal framework governing medicinal products for human use in the EU. Retrieved from <https://health.ec.europa.eu/medicinal-products/legal-framework-governing-medicinal-products-human-use-eu_en>

[12] C. Bowles. (2018). “Future Ethics”, Nownext. ISBN: 978-1-9996019-1-1 (Paperback), Technical reviewers: L. Nicholas, T. Wendt, and D. Williams. Foreword by A. Cooper.

[13] Michael J. Quinn. (2020). “Ethics for the Information Age”, 8th ed. Pearson.

# Appendix

## Section **1**A

### [‘Chardet’](#_1A._Concurrency)

# Chardet used to detect encoding and load properly. Code below from load\_data() function

with open(filename, 'rb') as f:

result = chardet.detect(f.read())

encoding = result['encoding']

A white background with black text

Description automatically generated

### [‘Overview Diagram’](#_1A._Concurrency)

A diagram of a data processing process

Description automatically generated

### [‘Threading diagram’](#_1A._Concurrency_1)

A diagram of a data processing process

Description automatically generated

## Section 1B

### [‘Tkinter Styling’](#_1B._GUI_Interface)

# Ttk modern theme

style = ttk.Style()

style.theme\_use("clam")  # 'clam' is a modern-looking theme

# Setting font for all buttons

button\_font = ("Arial", 24, "bold")

# Setting the colour for each button. The colour help guide the user and distinguish function types (data vs. file)

colors = "#B200ED", "#B200ED", "#B200ED", "#0B6623", "#50C878", "#50C878", "#50C878", "#50C878", "#50C878"]

### [‘Gui Layout’](#_1B._GUI_Interface)

A screenshot of a computer

Description automatically generated

### [‘LOAD CODE’](#_1B._GUI_Interface)

# Allows loading the initial csv or the converted jsons

filenames = filedialog.askopenfilenames(title="Select file(s)", filetypes=(("csv files", "\*.csv"), ("json files", "\*.json")))

# Loading multiple datasets into a temporary list

temp\_dfs = [] # local list for storing individual dataframes

…

# Merges dataframes on 'id' column and combining them to a single dataframe one by one using Python’s reduce function

global dataframes # modifying the global variable

dataframes.clear() # clear existing dataframes

merged\_df = reduce(lambda left, right: pd.merge(left, right, on='id'), temp\_dfs)

dataframes.append(merged\_df)

### [‘Try Except Button’](#_1B._GUI_Interface)

# Try-except blocks used throughout program for error correction

def load\_file\_button\_command():

try:

load\_data()

text\_box.insert(tk.END, "Data loaded successfully.\n")

except Exception as e:

text\_box.insert(tk.END, f"Error loading data: {e}\n")

### [Tkinter back-up data](#_1B._GUI_Interface)

# Tkinter dialog along with pandas used to backup data

if messagebox.askokcancel("Quit", "Do you want to quit?"):

create\_backup() # Backup current state

root.destroy()

…

# Ask the user where to save the JSON file

json\_file\_path = filedialog.asksaveasfilename(defaultextension=".json", filetypes=(("JSON files", "\*.json"), ("All files", "\*.\*"))

A screenshot of a phone

Description automatically generated A screenshot of a computer

Description automatically generated

## Section **1**C

### [‘Dynamic Flexibility in Python’](#_1C._Java_vs)

#1 Inferring type based on extension name in load\_data() function.

if filename.endswith('.csv'):

df = pd.read\_csv(filename, encoding=encoding)

elif filename.endswith('.json'):

df = pd.read\_json(filename, lines=True, encoding=encoding)

#2 Duck typing in clean\_data() function that assumes columns and checks for column names rather than working with types.

if 'Date' in df.columns:

df['Date'] = pd.to\_datetime(df['Date'], dayfirst=True)

#3 Generic programming to remove whitespaces from all columns regardless of type in clean\_data() function.

df = df.apply(lambda x: x.str.strip() if x.dtype == "object" else x)

### [‘Pandas, NumPy, SciPy, and Matplotlib’](#_1C._Java_vs)

# Pandas dataframes were an essential part of the application throughout. Below it is used to crosstab for my correlation() function

crosstab = pd.crosstab(df[labels[i]], df[labels[j]])

# Scipy being used for the chi-square test of independence for correlation()

c, p, dof, expected = chi2\_contingency(crosstab)

# Matplotlib being used for creating a FacetGrid and setting parameters for visualize() function

g = sns.FacetGrid(df\_melted, col='EID', row='variable', height=4, aspect=1)

g.map\_dataframe(sns.countplot, x='value')

# Numpy being used to replace missing values in the clean\_data() function

df.fillna(np.nan, inplace=True)

text\_box.insert(tk.END, "Replaced all missing values with numpy NaN.\n")

### [‘Renaming Imports’](#_1C._Java_vs)

# Library imports can be given an alias in Python as used in my code below:

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import scipy.stats as ss

import seaborn as sns

import tkinter as tk

### [‘Lambda Function Example’](#_1C._Java_vs)

‘’ Below, the Pandas function df.apply applies the lambda function to each column of the dataframe. Lambda function syntax follows ‘lambda arguments: expression’ [1]. Here, the arguments are x representing a series in the dataframe, the expression is to strip() whitespaces if x is type object/string or it just returns x. ‘’’

df = df.apply(lambda x: x.str.strip() if x.dtype == "object" else x)

text\_box.insert(tk.END, "Removed trailing and leading whitespaces from all columns.\n")

## Section 2A

### [‘Data Cleaning & Extracting’](#_2A._JSON_Data)

A screenshot of a computer program

Description automatically generated#1 whitespace handling

df.columns = df.columns.str.strip()

df = df.apply(lambda x: x.str.strip() if x.dtype == "object" else x)

#2 Datetime conversion

df['Date'] = pd.to\_datetime(df['Date'], dayfirst=True)

#3 Data Formatting

df['In-Use ERP Total'] = df['In-Use ERP Total'].str.replace(',', '').astype(float)

#4 Null Value Treatment with numpy

df.fillna(np.nan, inplace=True)

#5 Columns name standardization

df.rename(columns={'In-Use ERP Total': 'Power(kW)'}, inplace=True)

df.rename(columns={'In-Use Ae Ht': 'Aerial height(m)'}, inplace=True)

df.rename(columns={'Freq.': 'Freq'}, inplace=True)

#6 Removing client specified rows

df = df[~df['NGR'].isin(['NZ02553847', 'SE213515', 'NT05399374', 'NT252675908'])]

### [Create new EID columns](#_2A._JSON_Data)

# Add new columns for each selected multiplex in df

for multiplex in multiplexes:

dataframes[0][multiplex] = dataframes[0]['EID'].apply(lambda x: 1 if x == multiplex else 0)

### [‘Join JSON GUI tabs’](#_2A._JSON_Data)

A screenshot of a computer

Description automatically generated

### [‘JSON convert, save, & backup’](#_2A._JSON_Data)

#1 Saving DataFrame as JSON in save\_file() function

json\_str = df.to\_json(orient="records")

json\_file\_path = filedialog.asksaveasfilename(defaultextension=".json", filetypes=(("JSON files", "\*.json"), ("All files", "\*.\*")))

#2 Generating the nested JSON as requested by client in create\_json()

df\_json = dataframes[0].copy()

df\_json['DATE'] = df\_json['DATE'].apply(lambda x: x.isoformat() if not pd.isnull(x) else '')

nested\_dict = df\_json.groupby('EID').apply(lambda x: x.groupby('NGR')[['SITE', 'SITE HEIGHT', 'AERIAL HEIGHT(M)', 'POWER(KW)', 'DATE']].apply(lambda y: y.to\_dict('records')).to\_dict()).to\_dict()

json\_str = json.dumps(nested\_dict, indent=2)

#3 Meeting requirement to save current state as JSON in create\_backup() function

json\_file\_path = filedialog.asksaveasfilename(defaultextension=".json", filetypes=(("JSON files", "\*.json"), ("All files", "\*.\*")))

if json\_file\_path:

# Create a copy of the global dataframe for JSON serialization

df\_json = dataframes[0].copy()

…

# Save as JSON

df\_json.to\_json(json\_file, orient='records', lines=True)

### [‘JSON DATA FORMAT FULL DIAGRAM’](#_2A._JSON_Data)

|  |  |  |
| --- | --- | --- |
| {  "C188": {  "NU224152": [  {  "SITE": "Peppermoor Farm",  "SITE HEIGHT": 107,  "AERIAL HEIGHT(M)": 27,  "POWER(KW)": 1.199999,  "DATE": "2016-08-25T00:00:00"  }  ],  "NZ03586532": [  {  "SITE": "NEWTON",  "SITE HEIGHT": 165,  "AERIAL HEIGHT(M)": 45,  "POWER(KW)": 0.6,  "DATE": "2016-05-26T00:00:00"  }  ],  "NZ06555323": [  {  "SITE": "SHOTLEYFIELD",  "SITE HEIGHT": 199,  "AERIAL HEIGHT(M)": 53,  "POWER(KW)": 0.3,  "DATE": "2016-04-05T00:00:00"  }  ],  "NZ18424749": [  {  "SITE": "Burnhope",  "SITE HEIGHT": 240,  "AERIAL HEIGHT(M)": 227,  "POWER(KW)": 4.379999,  "DATE": "2000-11-21T00:00:00"  }  ],  "NZ21486744": [  {  "SITE": "Kenton TE",  "SITE HEIGHT": 120,  "AERIAL HEIGHT(M)": 21,  "POWER(KW)": 0.1,  "DATE": "2016-09-19T00:00:00"  }  ],  "NZ21676488": [  {  "SITE": "Fenham",  "SITE HEIGHT": 120,  "AERIAL HEIGHT(M)": 35,  "POWER(KW)": 1.499999,  "DATE": "2000-11-15T00:00:00"  }  ],  "NZ21818645": [  {  "SITE": "MORPETH",  "SITE HEIGHT": 60,  "AERIAL HEIGHT(M)": 50,  "POWER(KW)": 1.100001,  "DATE": "2016-04-26T00:00:00"  }  ],  "NZ25156398": [  {  "SITE": "Cale Cross House",  "SITE HEIGHT": 15,  "AERIAL HEIGHT(M)": 62,  "POWER(KW)": 0.3,  "DATE": "2017-11-23T00:00:00"  }  ],  "NZ26434233": [  {  "SITE": "DURHAM",  "SITE HEIGHT": 100,  "AERIAL HEIGHT(M)": 27,  "POWER(KW)": 0.3,  "DATE": "2016-07-27T00:00:00"  }  ],  "NZ3501071334": [  {  "SITE": "Whitley Bay",  "SITE HEIGHT": 40,  "AERIAL HEIGHT(M)": 26,  "POWER(KW)": 0.1,  "DATE": "2016-06-30T00:00:00"  }  ],  "NZ40955565": [  {  "SITE": "Hendon",  "SITE HEIGHT": 10,  "AERIAL HEIGHT(M)": 42,  "POWER(KW)": 0.8,  "DATE": "2016-07-27T00:00:00"  }  ]  },  "C18A": {  "NS72138846": [  {  "SITE": "Earls Hill - DAB",  "SITE HEIGHT": 410,  "AERIAL HEIGHT(M)": 49,  "POWER(KW)": 0.74,  "DATE": "2016-05-12T00:00:00"  }  ], | "NS82836457": [  {  "SITE": "Black Hill",  "SITE HEIGHT": 275,  "AERIAL HEIGHT(M)": 155,  "POWER(KW)": 2.399999,  "DATE": "2000-10-10T00:00:00"  }  ],  "NT23338724": [  {  "SITE": "CRAIGKELLY",  "SITE HEIGHT": 181,  "AERIAL HEIGHT(M)": 78,  "POWER(KW)": 5.999997,  "DATE": "2000-10-11T00:00:00"  }  ],  "NT25056945": [  {  "SITE": "Braid Hills",  "SITE HEIGHT": 202,  "AERIAL HEIGHT(M)": 43,  "POWER(KW)": 0.977237,  "DATE": "2007-05-02T00:00:00"  }  ],  "NT25117350": [  {  "SITE": "EDINBURGH CASTLE",  "SITE HEIGHT": 121,  "AERIAL HEIGHT(M)": 18,  "POWER(KW)": 0.0272,  "DATE": "2018-03-28T00:00:00"  }  ],  "NT25265908": [  {  "SITE": "PENICUIK",  "SITE HEIGHT": 248,  "AERIAL HEIGHT(M)": 43,  "POWER(KW)": 2.5,  "DATE": "2000-10-12T00:00:00"  }  ],  "NT51617605": [  {  "SITE": "Athelstaneford",  "SITE HEIGHT": 175,  "AERIAL HEIGHT(M)": 18,  "POWER(KW)": 1.0,  "DATE": "2016-05-25T00:00:00"  }  ]  },  "C18F": {  "SE22291288": [  {  "SITE": "Emley Moor",  "SITE HEIGHT": 256,  "AERIAL HEIGHT(M)": 271,  "POWER(KW)": 5.0,  "DATE": "2013-05-13T00:00:00"  }  ],  "SE23723509": [  {  "SITE": "Beecroft Hill",  "SITE HEIGHT": 149,  "AERIAL HEIGHT(M)": 44,  "POWER(KW)": 0.977237,  "DATE": "2001-05-15T00:00:00"  }  ],  "SE283268": [  {  "SITE": "Morley",  "SITE HEIGHT": 120,  "AERIAL HEIGHT(M)": 59,  "POWER(KW)": 1.0,  "DATE": "2001-11-14T00:00:00"  }  ]  }  } |  |

## Section 2B

### [‘Pandas for Mean/Mode/Median’](#_2B._Calculating_3rd)

# Filter based on 'Site Height'

df\_filtered\_height = df[df['SITE HEIGHT'] > 75]

# Filter based on 'Date'

df\_filtered\_year = df[df['DATE'].dt.year >= 2001]

# Pandas used for calculations

mean\_power\_height = df\_filtered\_height['POWER(KW)'].mean()

median\_power\_height = df\_filtered\_height['POWER(KW)'].median()

mode\_power\_height = df\_filtered\_height['POWER(KW)'].mode()[0] if not df\_filtered\_height['POWER(KW)'].mode().empty else "No…

A screenshot of a computer

Description automatically generated

### [‘Global Variable Checks’](#_2B._Calculating_3rd)

# The application uses global variables as state flags to manage the sequential processing requirements of the data. These are declared at the beginning of the notebook:

# Global variables declared at beginning of notebook:

dataframes = []

is\_data\_loaded = False

is\_data\_cleaned = False

is\_data\_extracted = False

# These are updated to true when tasks are completed. For example at the end of clean\_data() are the lines:

is\_data\_cleaned = True

text\_box.insert(tk.END, "Dataframe cleaned.\n")

# This prevents potential errors and guides users through the correct sequence of events. For example in calculating the mean/median/mode:

global is\_data\_loaded

global is\_data\_cleaned

global is\_data\_extracted

if not is\_data\_loaded or not is\_data\_cleaned or not is\_data\_extracted:

text\_box.insert(tk.END, "Please load, clean, and extract the data.\n")

return

## Section 2C

### [‘Tkinter Code’](#_2C._Client_Visualisations)

# Tkinter root window to host all widgets and elements

root = tk.Tk()

root.title("DAB Data Management")

# Using ttk for tab management

tab\_parent = ttk.Notebook(root)

# Using FigureCavasTkAgg and Matplotlib to create the plot

fig = plt.Figure(figsize=(5, 4), dpi=100)

plot\_canvas = FigureCanvasTkAgg(fig, master=parent)

### [‘Facetgrid’](#_2C._Client_Visualisations)

# Code shows Seaborn’s Facetgrid integrated into the GUI. The plot\_label() function melts the dataframe with Pandas and plots it using Facetgrid.

labels = ['FREQ', 'SITE', 'BLOCK', 'SERV LABEL1', 'SERV LABEL2', 'SERV LABEL3', 'SERV LABEL4', 'SERV LABEL10']

df\_melted = df.melt(id\_vars=['EID'], value\_vars=labels)

g = sns.FacetGrid(df\_melted, col='EID', row='variable', height=4, aspect=1)

g.map\_dataframe(sns.countplot, x='value')

g.set\_axis\_labels("Values", "Count")

g.set\_titles(col\_template="{col\_name} EID", row\_template="{row\_name}")

for ax in g.axes.flat:

for label in ax.get\_xticklabels():

label.set\_rotation(90)

facet\_tab = ttk.Frame(tab\_parent)

tab\_parent.add(facet\_tab, text="EID FacetGrid")

# Full scrollable Facetgrid available in application

A screenshot of a computer

Description automatically generated

## Section 2D

### [‘p, c, dof, expected’ for Chi-square](#_2D._Data_Correlation)

* P-value(p) – Measures probability of null hypothesis being true, the null hypothesis being that there is no association between the two variables (they are independent). A low p-value suggests that the null hypothesis is false and that there is a correlation between the two. It is widely accepted that a p-value from a chi-square test is deemed significant if it is less than 0.05 [6, p.61].
* Chi-square Test Statistic (c): Gives an idea of the magnitude of the discrepancy between observed and expected frequencies. A larger chi-square value indicates a greater discrepancy.
* Degrees of Freedom (dof): Categorical measure determined by the number of categories in the variables being tested. My application results showed very high values (20) for ‘SITE’ as each ‘SITE’ value was unique.
* Expected frequencies(expected): Provides insight into how the categories relate. When frequencies are below 5 the chi-square test may not be valid [6, p.80].

### [‘Detailed Results for Chi-Square test’](#_2D._Data_Correlation)

# Note that the full screen output for the Chi-sqared test could not fit within the report’s formatting requirements (1 page). Full output can be found using the application. Sample provided below:

|  |  |  |
| --- | --- | --- |
| Significant Pairs:  Between FREQ - BLOCK:  Between FREQ - SERV LABEL1:  Between FREQ - SERV LABEL2:  Between FREQ - SERV LABEL3:  Between BLOCK - SERV LABEL1:  Between BLOCK - SERV LABEL2:  Between BLOCK - SERV LABEL3:  Between SERV LABEL1 - SERV LABEL2:  Between SERV LABEL1 - SERV LABEL3:  Between SERV LABEL1 - SERV LABEL4:  Between SERV LABEL1 - SERV LABEL10:  Between SERV LABEL2 - SERV LABEL3:  Between SERV LABEL2 - SERV LABEL4:  Between SERV LABEL2 - SERV LABEL10:  Between SERV LABEL3 - SERV LABEL4:  Between SERV LABEL3 - SERV LABEL10:  Between SERV LABEL4 - SERV LABEL10:  Not Significant Pairs:  Between SITE - FREQ:  Between SITE - BLOCK:  Between SITE - SERV LABEL1:  Between SITE - SERV LABEL2:  Between SITE - SERV LABEL3:  Between SITE - SERV LABEL4:  Between SITE - SERV LABEL10:  Between FREQ - SERV LABEL4:  Between FREQ - SERV LABEL10:  Between BLOCK - SERV LABEL4:  Between BLOCK - SERV LABEL10:  Detailed Results:  --------------------------------------------------  Significant Results:  Between FREQ and BLOCK:  P-value: 0.0000  Chi-square: 17.1823  Degrees of Freedom: 1  Expected Frequencies:  [[5.76190476 5.23809524]  [5.23809524 4.76190476]]  Between FREQ and SERV LABEL1:  P-value: 0.0000  Chi-square: 21.0000  Degrees of Freedom: 2  Expected Frequencies:  [[3.66666667 1.57142857 5.76190476]  [3.33333333 1.42857143 5.23809524]]  Between FREQ and SERV LABEL2:  P-value: 0.0000  Chi-square: 21.0000  Degrees of Freedom: 2  Expected Frequencies:  [[3.66666667 5.76190476 1.57142857]  [3.33333333 5.23809524 1.42857143]] | Between BLOCK and SERV LABEL3:  P-value: 0.0000  Chi-square: 21.0000  Degrees of Freedom: 2  Expected Frequencies:  [[1.57142857 3.66666667 5.76190476]  [1.42857143 3.33333333 5.23809524]]  Between SERV LABEL1 and SERV LABEL2:  P-value: 0.0000  Chi-square: 42.0000  Degrees of Freedom: 4  Expected Frequencies:  [[2.33333333 3.66666667 1. ]  [1. 1.57142857 0.42857143]  [3.66666667 5.76190476 1.57142857]]  Between SERV LABEL1 and SERV LABEL3:  P-value: 0.0000  Chi-square: 42.0000  Degrees of Freedom: 4  Expected Frequencies:  [[1. 2.33333333 3.66666667]  [0.42857143 1. 1.57142857]  [1.57142857 3.66666667 5.76190476]]  Between SERV LABEL1 and SERV LABEL4:  P-value: 0.0000  Chi-square: 21.0000  Degrees of Freedom: 2  Expected Frequencies:  [[6. 1. ]  [2.57142857 0.42857143]  [9.42857143 1.57142857]]  Between SERV LABEL1 and SERV LABEL10:  P-value: 0.0000  Chi-square: 21.0000  Degrees of Freedom: 2  Expected Frequencies:  [[1. 6. ]  [0.42857143 2.57142857]  [1.57142857 9.42857143]]  Between SERV LABEL2 and SERV LABEL3:  P-value: 0.0000  Chi-square: 42.0000  Degrees of Freedom: 4  Expected Frequencies:  [[1. 2.33333333 3.66666667]  [1.57142857 3.66666667 5.76190476]  [0.42857143 1. 1.57142857]] | Not Significant Results:  Between SITE and FREQ:  P-value: 0.3971  Chi-square: 21.0000  Degrees of Freedom: 20  Expected Frequencies:  [[0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]  [0.52380952 0.47619048]]  Between SITE and BLOCK:  P-value: 0.3971  Chi-square: 21.0000  Degrees of Freedom: 20  Expected Frequencies:  [[0.52380952 0.47619048]  …  Between FREQ and SERV LABEL4:  P-value: 0.1810  Chi-square: 1.7898  Degrees of Freedom: 1  Expected Frequencies:  [[9.42857143 1.57142857]  [8.57142857 1.42857143]]  Between FREQ and SERV LABEL10:  P-value: 0.1810  Chi-square: 1.7898  Degrees of Freedom: 1  Expected Frequencies:  [[1.57142857 9.42857143]  [1.42857143 8.57142857]] |

1. ‘Chardet’ in Appendix 1A [↑](#footnote-ref-1)
2. ‘Overview Diagram’ & ‘Threaded Diagram’ in Appendix 1A [↑](#footnote-ref-2)
3. ‘Tkinter styling’ in Appendix 1B [↑](#footnote-ref-3)
4. ‘GUI Layout’ in Appendix 1B [↑](#footnote-ref-4)
5. ‘Load-Code’ in Appendix 1B [↑](#footnote-ref-5)
6. ‘Try-Except’ in Appendix 1B [↑](#footnote-ref-6)
7. ‘Overview Diagram’ in Appendix 1A [↑](#footnote-ref-7)
8. ‘Tkinter back-up data’ in Appendix 1B [↑](#footnote-ref-8)
9. ‘Dynamic Flexibility in Python’ in Appendix 1C [↑](#footnote-ref-9)
10. ‘Pandas, NumPy, SciPy, and Matplotlib’ in Appendix Section 1C [↑](#footnote-ref-10)
11. ‘Renaming’ in Appendix 1C [↑](#footnote-ref-11)
12. ‘Lambda Functions’ in Appendix 1C [↑](#footnote-ref-12)
13. ‘Data Cleaning’ in Appendix 2A [↑](#footnote-ref-13)
14. ‘Create new EID columns’ in Appendix 2A [↑](#footnote-ref-14)
15. ‘Full Data Format Diagram’ in Appendix 2A [↑](#footnote-ref-15)
16. ‘Join JSON GUI tabs’ in Appendix 2A [↑](#footnote-ref-16)
17. ‘JSON convert, save, & backup’ in Appendix 2A [↑](#footnote-ref-17)
18. ‘JSON convert, save, & backup’ in Appendix 2A [↑](#footnote-ref-18)
19. See ‘Global Variable Checks’ in Appendix 2B [↑](#footnote-ref-19)
20. ‘Data Cleaning’ in Appendix 2A [↑](#footnote-ref-20)
21. ‘Pandas for Mean/Mode/Median’ in Appendix 2B [↑](#footnote-ref-21)
22. ‘Tkinter Code’ in Appendix 2C [↑](#footnote-ref-22)
23. See ‘FacetGrid’ in Appendix 2C [↑](#footnote-ref-23)
24. ‘Global Variable Checks’ in Appendix 2B [↑](#footnote-ref-24)
25. see ‘p, c, dof, expected for Chi-square’ in Appendix 2D for detailed description of the values [↑](#footnote-ref-25)
26. See ‘Detailed Results for Chi-Square test’ [↑](#footnote-ref-26)
27. See ‘Detailed Results for Chi-Square test’ [↑](#footnote-ref-27)