**LOG FILE For DICOM IMAGES V-1.0**

Index Table:

| **Serial No.** | **Contents** | Page No. |
| --- | --- | --- |
| 1 | Introduction | 1 |
| 2 | Objective | 1 |
| 3 | Input | 2 |
| 4 | Output | 2 |
| 5 | Dependencies | 2 |
| 6 | Code | 2-3 |
| 7 | Basic Documentation | 3 |
| 8 | Proper Description of the functions,commands and variables | 4-6 |
| 9 | Algorithm | 6-7 |
| 10 | Testing: Functional/Unit/Integration/Latency Time etc | 7-10 |
| 11 | TRy and Exception - scenario | 10-11 |
| 12 | Summary | 12 |
| 13 | Code Replicating With Bard | 13 |
| 14 | Documentation | 16 |
| 13 | Commenting Swarsat code using BARD | 31 |
| 14 | Upload on Github and other code base libraries |  |

**Introduction**

Python packages are a way to organise and distribute reusable Python code. They provide a structured and modular approach to building, sharing, and reusing functionality across different projects. A package is a directory that contains one or more Python modules along with additional files or subdirectories.

Packages enable code organisation by grouping related modules together, making it easier to manage and maintain large-scale projects. They promote code reuse, as packages can be shared and imported into different projects or distributed to other developers.Here we intend to demonstrate a code with the following parameters for analysing Dicom images

For reading more about packages [click here](https://docs.google.com/document/d/1aO8JdKzBX5REg6sk-N0X5xZ_y3nwYxgCm9JqyWda7aQ/edit?usp=sharing)

**Objective**

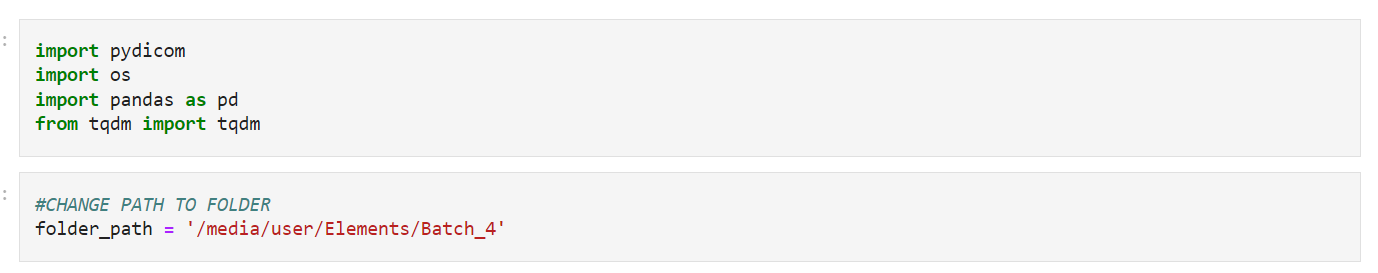
A.To read and parse multiple DICOM files and give out report in an CSV format

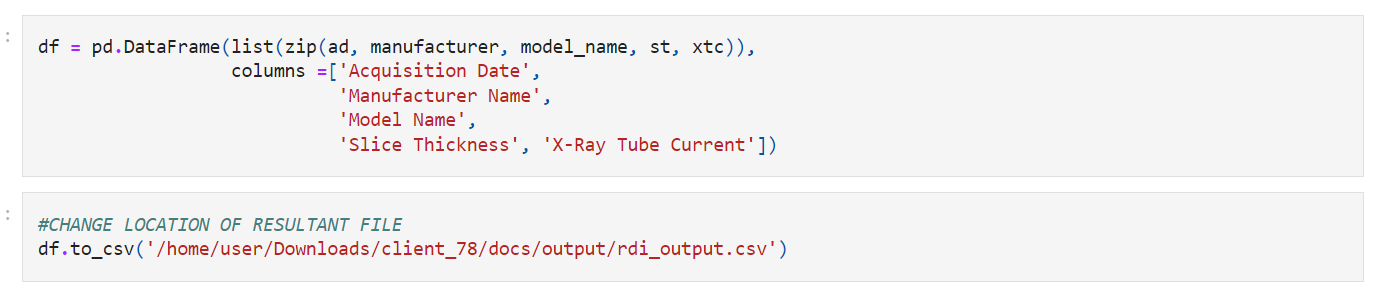
B. AI based experimentation

**Input files**: [DICOM IMAGES](https://drive.google.com/file/d/1ZzTFsM9s6k8MBhoDbBjuhRA7flYw0Don/view?usp=drive_link) (for viewing you can use Dicom viewer DMW)

**Output format or files**: Acquisition Date,Manufacturer Name, Model Name Slice Thickness, X-Ray Tube Current in [CSV format](https://drive.google.com/file/d/1RELcsxNIIAAKdIhayhSG35CE-9NZjZHJ/view?usp=drive_link)

**Dependencies**: Python,Dicom files,panda,pydicom,OS module,tqdm,[clear mention of the paths](https://drive.google.com/drive/folders/13VLVFnphXhthxznugiDe3T5uzNffG98K?usp=drive_link).

**Code**



The full code can be accessed by

<https://github.com/swarsatbioinfo/extract_dicom_info/blob/main/read_DICOM_info.ipynb?short_path=72390f8>

**Documentation**

This code snippet reads DICOM files from a specified folder path, extracts relevant information such as acquisition date, manufacturer, model name, slice thickness, and X-ray tube current, and stores the data in lists. It then creates a DataFrame using the collected data and saves it as a CSV file.

Make sure to modify the `folder\_path` variable to point to the desired folder containing DICOM files, and update the `to\_csv` function with the appropriate path where you want to save the resulting CSV file.

Additionally, please note that this code assumes that the `pydicom` library is installed and imported properly**.**

**Proper Description of the functions,commands and variables**

· **Functions:**

1. `os.walk(folder\_path)`: A function from the `os` module that recursively traverses through the specified folder and its subdirectories, returning a generator that yields tuples containing the path to the directory, a list of subdirectories, and a list of filenames in that directory.

2. `pydicom.dcmread(f)`: A function from the `pydicom` library used to read a DICOM file. It takes the file path as an argument and returns a `pydicom.Dataset` object representing the DICOM data.

3. `pd.DataFrame(list(zip(ad, manufacturer, model\_name, st, xtc)), columns=['Acquisition Date', 'Manufacturer Name', 'Model Name', 'Slice Thickness', 'X-Ray Tube Current'])`: A function from the `pandas` library that creates a DataFrame from the provided data. It takes a list of tuples where each tuple contains the values for each column and a list of column names.

4. `df.to\_csv('/home/user/Downloads/client\_78/docs/output/rdi\_output.csv')`: A method of the DataFrame (`df`) that saves the DataFrame as a CSV file. It takes the file path as an argument.

· **Variables:**

1. `folder\_path`: A string variable that holds the path to the folder containing DICOM files.

2. `ad`(array): A list variable used to store the acquisition dates extracted from DICOM files.

3. `manufacturer`(array): A list variable used to store the manufacturer names extracted from DICOM files.

4. `model\_name`(array): A list variable used to store the model names extracted from DICOM files.

5. `st`(array): A list variable used to store the slice thickness values extracted from DICOM files.

6. `xtc`(array): A list variable used to store the X-ray tube current values extracted from DICOM files.

· **Commands:**

1. `f = os.path.join(root, f)`: Joins the root directory path with the filename (`f`) to get the full path of the DICOM file.

2. `f.endswith('.dcm')`: Checks if the filename (`f`) has a `.dcm` extension, indicating it is a DICOM file.

3. `pydicom.dcmread(f).AcquisitionDate`: Extracts the acquisition date from the DICOM file.

4. `pydicom.dcmread(f).Manufacturer`: Extracts the manufacturer name from the DICOM file.

5. `pydicom.dcmread(f).ManufacturerModelName`: Extracts the model name from the DICOM file.

6. `pydicom.dcmread(f).SliceThickness`: Extracts the slice thickness value from the DICOM file.

7. `pydicom.dcmread(f).XRayTubeCurrent`: Extracts the X-ray tube current value from the DICOM file.

8. `print(len(manufacturer))`: Prints the length of the `manufacturer` list, providing an indication of progress.

Note: It's important to have the `pydicom` library installed and imported at the beginning of the code for the functions `pydicom.dcmread()` to work properly.

**Algorithm for the code**

1. Import the required libraries: `os`, `pandas`, `tqdm`, and `pydicom`.

2. Define the folder path where the DICOM files are located.

3. Create empty lists (`ad`, `manufacturer`, `model\_name`, `st`, `xtc`) to store the extracted information from the DICOM files.

4. Traverse through the folder and its subdirectories using `os.walk()`.

- For each file in the current directory:

- Check if the file has a `.dcm` extension and the `manufacturer` list is not full.

- Get the full file path using `os.path.join(root, f)`.

- Use `pydicom.dcmread()` to read the DICOM file and extract the desired information.

- Append the extracted information to the respective lists (`ad`, `manufacturer`, `model\_name`, `st`, `xtc`).

- Print the length of the `manufacturer` list to indicate progress.

5. Create a DataFrame (`df`) using the collected data:

- Use `list(zip(ad, manufacturer, model\_name, st, xtc))` to combine the lists into a list of tuples.

- Specify the column names as `['Acquisition Date', 'Manufacturer Name', 'Model Name', 'Slice Thickness', 'X-Ray Tube Current']`.

6. Save the DataFrame as a CSV file:

- Use the `to\_csv()` method of the DataFrame (`df`) to save it as a CSV file.

- Specify the desired output file path.

Note: Make sure to modify the `folder\_path` variable to point to the actual folder containing the DICOM files and update the `to\_csv()` function with the appropriate output file path.

**Testing and Running**

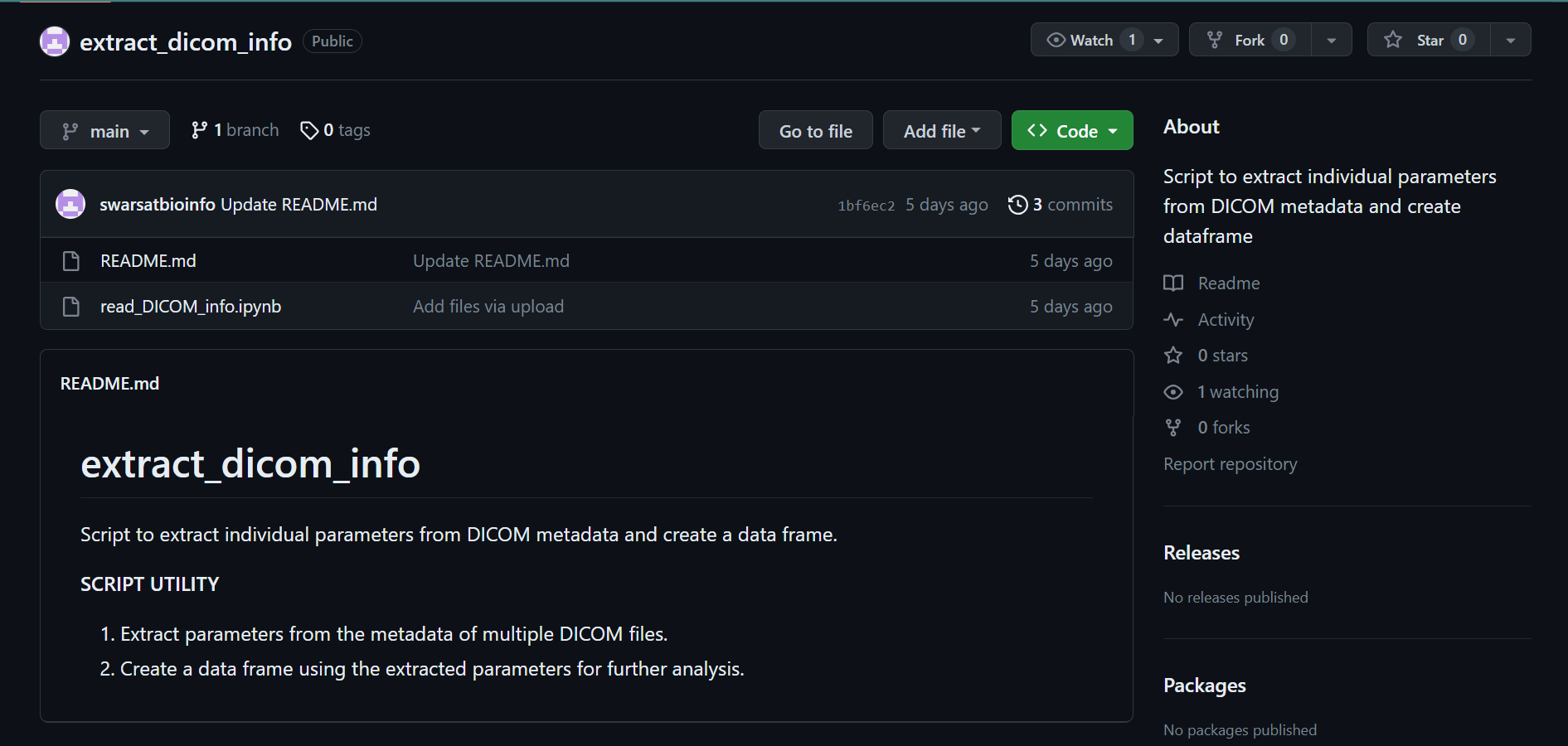
**(This is done in** [**Jupyter Notebook platform**](http://localhost:8888/notebooks/Downloads/extract_dicom_info-main/extract_dicom_info-main/read_DICOM_info.ipynb)**)**

To test the provided code, you can follow these steps:

1. Ensure that you have the required libraries (`os`, `pandas`, `tqdm`, `pydicom`) installed in your [Python environment](https://jupyter.org/install). You can install them using pip or any package manager of your choice.

2. Prepare a folder containing DICOM files. You can create a test folder with a few DICOM files or use an existing folder with DICOM files for testing purposes. Make sure to update the `folder\_path` variable in the code to point to t

3. You can access the file by clicking on the file given above and download the file with the correct folder path.



4. Copy the code into a Python script file (e.g., `dicom\_extraction.py`) or open the code in a Jupyter Notebook.

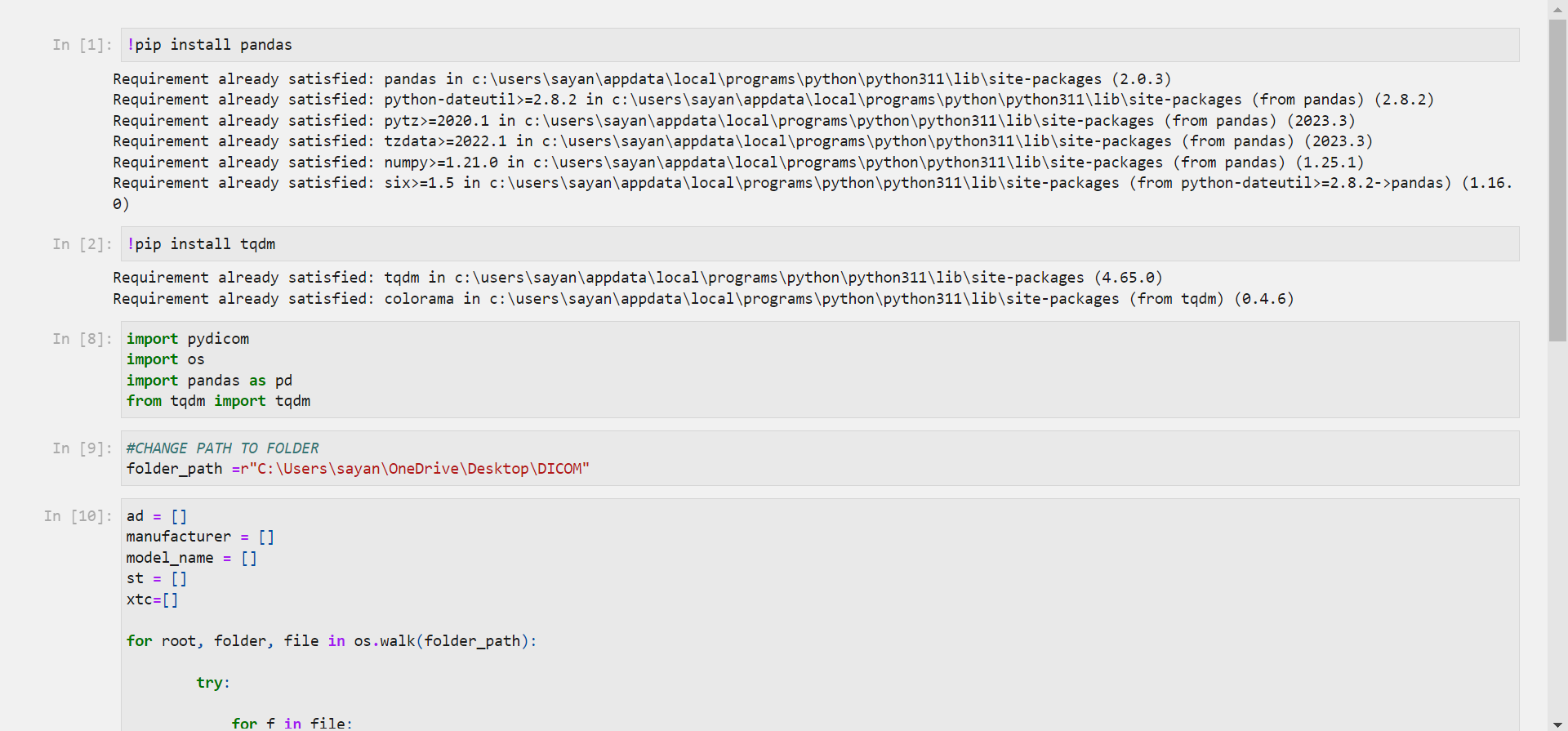
5. Run the code in your Python environment. You can execute the script file or run the cells in the Jupyter Notebook**.\***Replace the paths with the destination of your files and give a sample Dicom file

6. Monitor the progress by observing the printed length of the `manufacturer` list, which indicates the number of DICOM files processed.

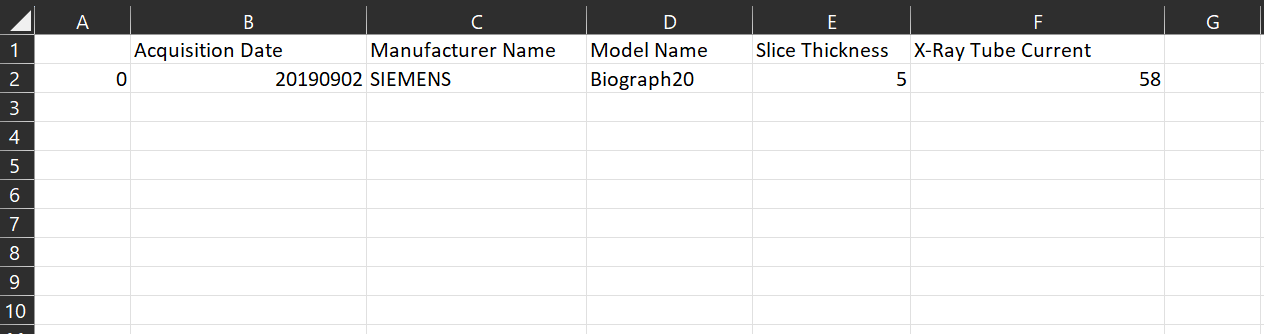
7. After the code finishes executing, check the specified output file path to find the generated CSV file (`rdi\_output.csv`). Verify that the file contains the extracted information from the DICOM files.

7. Analyze the generated CSV file using any appropriate tools or further process the data as needed.

Remember to ensure that the `pydicom` library is correctly installed and imported for the code to work properly. If any errors occur during testing, check for missing dependencies or syntax issues and make the necessary adjustments.

Note: It's always recommended to test the code on a small subset of data before running it on a large dataset to ensure everything is working as expected.

Note\*When testing the code, it is important to ensure that all the dependencies, including the required modules, are present. Failing to have the necessary dependencies can lead to unexpected errors and issues. Therefore, it is crucial to have the following considerations:

Output while testing

There are several types of code testing methodologies used in software development to ensure the quality and reliability of the code. Some of the common types of code testing are:

* **1. Unit Testing**
* **2. Integration Testing**
* **3. Functional Testing**
* **4. System Testing**
* **5. Acceptance Testing**
* **6. Regression Testing**
* **7. Performance Testing:**
* **8. Security Testing**
* **9. Exploratory Testing**

[For more click](https://docs.google.com/document/d/1AFZD9oyXST_XXd6i0flyS1kWYFrSMUOGlelaS29Aatg/edit?usp=sharing)

**Try and exception scenario**

In the provided code, a try-except block is already implemented to handle exceptions that may occur during DICOM parsing. However, it is generally recommended to specify the specific exception(s) you want to catch and handle. Here's an example of how you can modify the code to handle specific exceptions:

In this code, two specific exceptions are caught and handled:

- `pydicom.errors.InvalidDicomError`: This exception is raised when a DICOM file is invalid or cannot be properly parsed. In case of such an exception, an error message is printed, and the loop continues to the next file.

- `FileNotFoundError`: This exception is raised when a file is not found at the specified path. If this exception occurs, an error message is printed, and the loop continues to the next file.

By handling these exceptions explicitly, you can provide more informative error messages and have more control over exception handling in your code. Feel free to adjust the exception types or add more specific exception handling as needed for your specific use case

**Summary**

In simple words, the provided code allows us to extract information from DICOM files in a specified folder and store the extracted data in a CSV file. It utilises the `pydicom` library to read the DICOM files and extract relevant attributes such as acquisition date, manufacturer name, model name, slice thickness, and X-ray tube current.

By running this code with a folder containing DICOM files, you can automate the extraction of important metadata from medical imaging data. The resulting CSV file can be further analyzed, integrated into other applications, or used for research and analysis purposes.

The format of the code is in such a manner the snippets can be used in a modular manager for multiple uses in several fields

**TESTING-BARD**  
**The below code replicating the core code of generating csv files is done using BARD and no errors were found but path must be replace while running the code and attributes should be checked**

**import os**

**import csv**

**import pydicom**

**def read\_dicom\_files(directory):**

**"""Reads DICOM files from the specified directory and returns a list of DICOM objects."""**

**dicom\_files = []**

**for file in os.listdir(directory):**

**if file.endswith(".dcm"):**

**dicom\_file = pydicom.dcmread(os.path.join(directory, file))**

**dicom\_files.append(dicom\_file)**

**return dicom\_files**

**def write\_csv(dicom\_files, output\_file):**

**"""Writes the specified DICOM files to a CSV file."""**

**with open(output\_file, "w") as csvfile:**

**csvwriter = csv.writer(csvfile, delimiter=",")**

**csvwriter.writerow(["Manufacturer", "Acquisition Date", "Model Name", "Slice Thickness", "X Ray Tube Current"])**

**for dicom\_file in dicom\_files:**

**csvwriter.writerow([**

**dicom\_file.Manufacturer,**

**dicom\_file.AcquisitionDate,**

**dicom\_file.ModelName,**

**dicom\_file.SliceThickness,**

**dicom\_file.XRayTubeCurrent**

**])**

**def main():**

**directory = "/home/amity/Desktop/ai.py"**

**output\_file = "output.csv"**

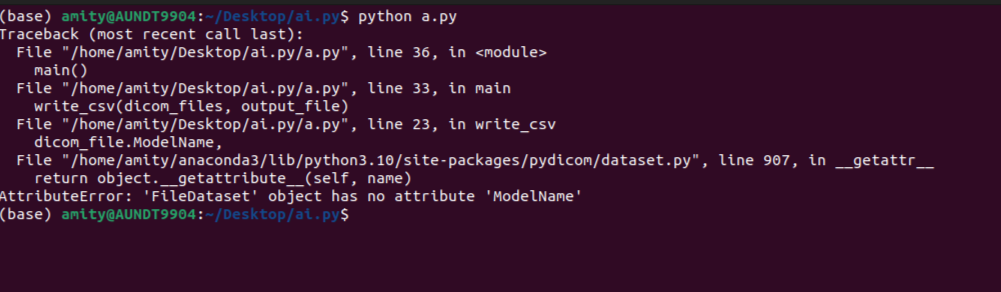
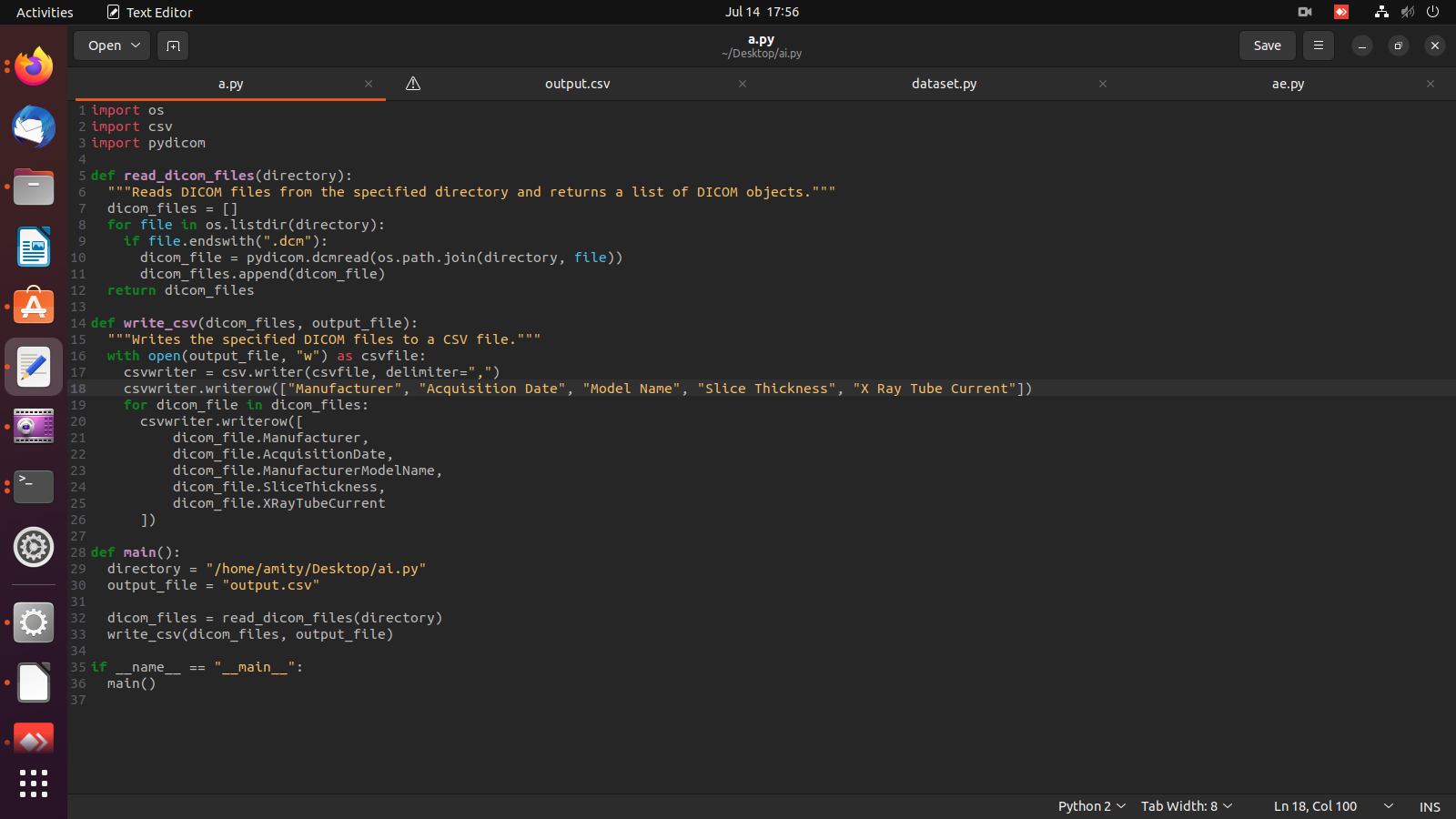
**dicom\_files = read\_dicom\_files(directory)**

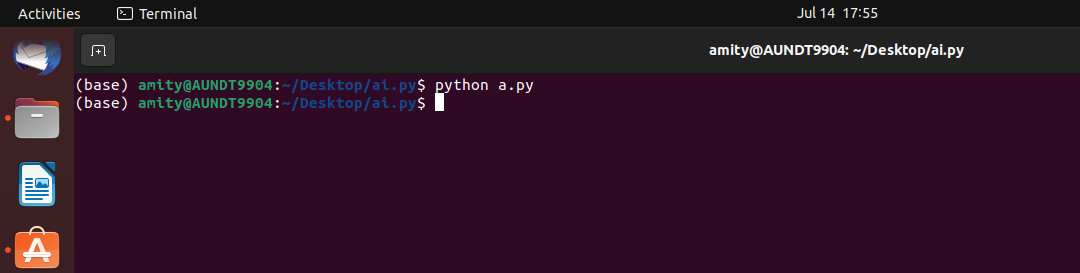
**write\_csv(dicom\_files, output\_file)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**Note\* the above code replicating the core code of generating csv files is done using BARD and errors were found but path must be replace while running the code**



**The debugged code can be accessed from** [**here**](https://drive.google.com/file/d/1CSnZIaw4A9J-CMGC7pLCPV4PyqZJ7aYG/view?usp=drive_link)

**The result can be accessed** [**from**](https://drive.google.com/file/d/1XN145wjb2fnr3j3Gag6jlzU8VQOyZtji/view?usp=drive_link)

**Documentation of the code using BARD**

**1. \*\*Parameters\*\*:**

- `directory` (str): The directory path where the DICOM files are located. It is passed as an argument to the `read\_dicom\_files` function.

- `dicom\_files` (list): A list of pydicom Dataset objects representing the DICOM files. It is passed as an argument to the `write\_csv` function.

- `output\_file` (str): The path to the output CSV file. It is passed as an argument to the `write\_csv` function and used to specify the output file name in the `main` function.

**2. \*\*Function Attributes\*\*:**

- `read\_dicom\_files`:

- `directory` (str): The directory path where the DICOM files are located. This attribute is a parameter of the function and represents the input directory path.

- `write\_csv`:

- `dicom\_files` (list): A list of pydicom Dataset objects representing the DICOM files. This attribute is a parameter of the function and represents the input DICOM files.

- `output\_file` (str): The path to the output CSV file. This attribute is a parameter of the function and represents the output file path.

**3. \*\*Variables\*\*:**

- `dicom\_files` (list): This variable is used to store the DICOM files read from the specified directory. It is assigned the return value of the `read\_dicom\_files` function in the `main` function.

- `file` (str): This variable is used in the `for` loop to iterate over the files in the directory when reading DICOM files.

- `dicom\_file` (pydicom.Dataset): This variable is used to store each DICOM file read in the `read\_dicom\_files` function. It is then appended to the `dicom\_files` list.

- `csvfile` (file object): This variable is used to open the output CSV file in write mode for writing the DICOM file data.

- `csvwriter` (csv.writer): This variable is used to create a CSV writer object for writing the DICOM file data to the CSV file.

The code uses these parameters, function attributes, and variables to read DICOM files from a directory, store them in a list, and write the DICOM file data to a CSV file.

**Algorithm**

**Step 1:** The read\_dicom\_files() function reads all of the DICOM files in the specified directory and returns a list of DICOM objects.

A DICOM file is a file format used to store medical images and associated data. The read\_dicom\_files() function uses the os.listdir() function to get a list of all of the files in the specified directory. Then, it iterates through the list of files and checks if each file ends with the .dcm extension. If it does, the function calls the pydicom.dcmread() function to read the DICOM file and return a DICOM object.

**Step 2:** The write\_csv() function opens the specified CSV file in write mode and creates a csv.writer object.

A CSV file is a file format that stores tabular data in plain text. The write\_csv() function uses the open() function to open the specified CSV file in write mode. Then, it creates a csv.writer object and uses it to write the following information from each DICOM file to the CSV file**:**

* **Manufacturer**
* **Acquisition Date**
* **Model Name**
* **Slice Thickness**
* **X Ray Tube Current**

**Step 3:** The csvwriter object is used to write the following information from each DICOM file to the CSV file:

The csvwriter object is a Python object that can be used to write data to a CSV file. The writerow() method of the csvwriter object is used to write a row of data to the CSV file. The row of data is a list of values, and the values are separated by commas.

**Step 4:** The main() function specifies the directory containing the DICOM files and the name of the CSV file to create.

The main() function is the entry point for the program. It specifies the directory containing the DICOM files and the name of the CSV file to create. Then, it calls the read\_dicom\_files() function to read the DICOM files and the write\_csv() function to write the information to the CSV file.

**Step 5:** The main() function calls the read\_dicom\_files() function to read the DICOM files and the write\_csv() function to write the information to the CSV file.

The read\_dicom\_files() function reads all of the DICOM files in the specified directory and returns a list of DICOM objects. The write\_csv() function opens the specified CSV file in write mode and creates a csv.writer object. The csvwriter object is used to write the following information from each DICOM file to the CSV file**:**

* **Manufacturer**
* **Acquisition Date**
* **Model Name**
* **Slice Thickness**
* **X Ray Tube Current**

**1. Unit Testing:**

Unit testing is the process of testing individual units or components of the code in isolation. It focuses on verifying the functionality of small, independent parts of the code. Unit tests are typically written by developers and executed frequently to catch bugs early and ensure that each component behaves as expected. For more click below

[**https://www.javatpoint.com/python-unit-testing**](https://www.javatpoint.com/python-unit-testing)

**Here's a general process for performing unit testing:**

* **Choose a Testing Framework:**

Select a testing framework suitable for your programming language. For Python, the built-in `unittest` module, `pytest`, or `nose` are commonly used testing frameworks.

* **Organize Test Files:**

te a separate directory or module to store your test files. Typically, test files are named with a prefix like `test\_` followed by the name of the module or class being tested (e.g., `test\_module.py`).

* **Write Test Cases:**

Define test cases as individual functions or methods within your test files. Each test case should focus on testing a specific behavior or functionality of the unit being tested.

* **Set Up and Tear Down:**

Use set up and tear down methods provided by the testing framework to set up any necessary test dependencies and clean up after each test. This ensures a clean and isolated environment for each test case.

* **Write Assertions:**

Within each test case, use assertions to verify that the expected behavior matches the actual behaviour of the unit being tested. Common assertions include checking values, comparing results, or verifying exceptions.

* **Run Tests:**

Execute the tests using the testing framework's command-line interface or by running a test runner script. The framework will run all the test cases and report the results, including which tests passed and which ones failed.

* **Analyse Test Results:**

Review the test results to identify any failed test cases. Inspect the failure details to understand the cause of the failure and determine if there are any issues with the unit being tested.

* **Refine and Iterate:**

Fix any issues identified in the failed tests and re-run the tests to ensure they pass. Repeat the process until all tests pass and the desired behaviour is achieved.

**Here's a basic example using the `unittest` framework in Python:**

**import os**

**import unittest**

**import pandas as pd**

**from your\_module import process\_dicom\_files**

**class TestDicomProcessing(unittest.TestCase):**

**def setUp(self):**

**self.folder\_path = '/media/user/Elements/Batch\_4'**

**def test\_dicom\_processing(self):**

**ad, manufacturer, model\_name, st, xtc = process\_dicom\_files(self.folder\_path)**

**# Assert that the lists are not empty**

**self.assertTrue(ad)**

**self.assertTrue(manufacturer)**

**self.assertTrue(model\_name)**

**self.assertTrue(st)**

**self.assertTrue(xtc)**

**# Assert that the lengths are less than 100000**

**self.assertLessEqual(len(manufacturer), 100000)**

**# Assert that the DataFrame is created successfully**

**df = pd.DataFrame(list(zip(ad, manufacturer, model\_name, st, xtc)),**

**columns=['Acquisition Date', 'Manufacturer Name', 'Model Name', 'Slice Thickness', 'X-Ray Tube Current'])**

**self.assertIsInstance(df, pd.DataFrame)**

**# Assert that the DataFrame has the correct columns**

**expected\_columns = ['Acquisition Date', 'Manufacturer Name', 'Model Name', 'Slice Thickness', 'X-Ray Tube Current']**

**self.assertListEqual(list(df.columns), expected\_columns)**

**# Assert that the DataFrame is not empty**

**self.assertFalse(df.empty)**

**# You can add more specific assertions as per your requirements**

**if \_\_name\_\_ == '\_\_main\_\_':**

**unittest.main()**

**In this example, the `unittest.TestCase` class is inherited to define a test case class. Within the test case class, a test method is defined to test the `add\_numbers` function. The `self.assertEqual()` assertion checks if the result of `add\_numbers(2, 3)` is equal to the expected result of `5`. The tests are executed by running the script.**

**You can add more test methods, test different functionalities, and use various assertions based on your specific requirements. Unit testing helps ensure the correctness and robustness of your code by detecting issues early and providing confidence during software development.**

**2. Integration Testing:**

Integration testing involves testing the interactions and integration between multiple components or modules of the code. It ensures that different parts of the system work together correctly when combined. Integration testing may identify issues like incompatible interfaces, incorrect data flow, or communication problems between components.for more and tutorial click here

[**https://www.softwaretestinghelp.com/what-is-integration-testing/**](https://www.softwaretestinghelp.com/what-is-integration-testing/)

The provided code is an example of unit testing, specifically testing the `process\_dicom\_files` function in the `your\_module` module. Integration testing focuses on testing the interaction and integration of multiple components or modules. To perform integration testing for this scenario, you can follow these steps:

**1. Identify the Components for Integration:**

Determine the key components or modules that interact with each other during the execution of the system. In this case, it could involve testing the interaction between the `process\_dicom\_files` function and other relevant modules or dependencies.

**2. Set Up the Test Environment:**

Configure the test environment by providing any necessary resources, dependencies, or mock objects required for the integration testing. This may involve setting up test data, creating mock objects, or establishing connections to external services if applicable.

**3. Define Integration Test Cases:**

Design test cases that cover various integration scenarios, including different combinations of components and their interactions. For example, you can simulate different folder paths, test the integration with a database, or verify the behavior when specific conditions are met.

**4. Perform Integration Tests:**

Implement the integration test cases by calling the appropriate functions or methods from the relevant modules and verifying the expected behavior. This may involve executing the `process\_dicom\_files` function and checking how it interacts with other modules or external dependencies.

**5. Validate Results and Assertions:**

Evaluate the results of the integration tests by comparing the actual output with the expected output or behaviour. Use assertions or comparison functions to verify that the integration between components is functioning correctly. Ensure that the system behaves as expected and any data exchanges or interactions are accurate.

**6. Handle Dependencies:**

Pay attention to any dependencies or external services involved in the integration testing. If necessary, use techniques like mocking, stubbing, or simulating external dependencies to isolate the integration tests and ensure consistent results.

**7. Monitor and Debug:**

Monitor the execution of the integration tests and investigate any failures or unexpected behaviors. Debug the code, review logs, or use debugging tools to identify the root causes of issues and fix them.

**8. Refine and Iterate:**

Iterate on the integration tests, making adjustments and enhancements based on feedback and any issues encountered. Continue to refine the integration testing process to ensure comprehensive coverage and reliability.

By following these steps, you can perform integration testing to validate the interactions between components and ensure the smooth integration of the `process\_dicom\_files` function with other modules or dependencies.

**3. Functional Testing:**

Functional testing evaluates the behaviour of the code from the end-user's perspective. It verifies that the code meets the specified requirements and performs the intended functions accurately. Functional testing focuses on testing the system as a whole and can include both positive and negative scenarios to ensure proper functionality in different situations. For more and tutorial click here

[**https://www.guru99.com/functional-testing.html**](https://www.guru99.com/functional-testing.html)

To perform functional testing on the provided code, you can focus on verifying the expected functionality and outputs. Here's an example of how you can perform functional testing for the given code:

**1. Prepare the Test Environment:**

- Set up a test environment with a specific folder structure and DICOM files for testing.

- Ensure that the folder path (`folder\_path`) points to the test folder containing the DICOM files.

**2. Define Functional Test Cases:**

- Identify the functional requirements and create test cases based on those requirements.

- For example, test cases can include:

- Verifying that the code correctly traverses the folder and its subdirectories.

- Checking if the code correctly identifies DICOM files with the ".dcm" extension.

- Verifying that the code extracts the desired metadata (Acquisition Date, Manufacturer, Model Name, Slice Thickness, X-Ray Tube Current) correctly from the DICOM files.

- Checking if the generated DataFrame has the expected structure and data.

- Verifying that the DataFrame is saved to the specified CSV file location with the correct column names.

**3. Perform Functional Tests:**

- Implement the functional test cases by executing the code and observing its behavior.

- For each test case, use assertions or comparison functions to validate the actual outputs against the expected outputs.

- For example, you can assert that the length of the `manufacturer` list is less than 100,000 and that the DataFrame is not empty.

- You can also compare specific metadata values from the DataFrame with the expected values.

**4. Validate Results and Assertions:**

- Review the results of the functional tests and verify that the code meets the functional requirements defined in the test cases.

- If any assertions fail, investigate and debug the code to identify and resolve any functional issues.

**5. Handle Dependencies:**

- Ensure that any necessary dependencies, such as the `pydicom` library, are properly installed and configured.

- Handle the dependency by importing the required modules or mocking the behavior of the dependencies during testing if needed.

**6. Monitor and Debug:**

**-** Monitor the execution of the functional tests and investigate any failures or unexpected behaviors.

- Debug the code, review logs, or use debugging tools to identify and fix any functional issues.

**7. Refine and Iterate:**

- Refine the functional tests based on feedback and any issues encountered.

- Make adjustments or enhancements to the code to improve functional correctness and ensure it meets the desired functional requirements.

By following these steps, you can perform functional testing for the provided code, ensuring that it functions correctly according to the specified requirements and produces the expected outputs.

**4. System Testing:**

System testing involves testing the entire system or application as a whole to verify that it meets the specified requirements. It evaluates the system's behaviour in different environments and configurations. System testing may include performance testing, security testing, compatibility testing, and other tests to ensure the system's overall functionality and reliability.For more and tutorial click here [**https://www.guru99.com/system-testing.html**](https://www.guru99.com/system-testing.html)

System testing involves testing the entire system or application as a whole to ensure that it meets the desired functionality, performance, and reliability. In the provided code, system testing can involve verifying the overall behavior and outputs of the code. Here's an example of how you can perform system testing for the given code:

1. **Prepare the Test Environment:**

- Set up a test environment that closely resembles the production environment.

- Ensure that the folder structure and DICOM files in the specified `folder\_path` are representative of real-world scenarios.

2. **Define System Test Cases:**

- Identify the system-level requirements and create test cases based on those requirements.

- System test cases can include:

- Verifying that the code correctly traverses the folder and its subdirectories, including nested subdirectories.

- Checking if the code handles various file types and formats within the folder, not just DICOM files.

- Verifying that the code correctly extracts and handles metadata from the DICOM files, considering different DICOM variations.

- Checking if the generated DataFrame has the expected structure and data, even with a large number of files.

- Verifying that the DataFrame is correctly saved to the specified CSV file location, ensuring the file is created and contains the expected data.

3. **Perform System Tests:**

- Implement the system test cases by executing the code and observing its behavior as a complete system.

- Validate the actual outputs against the expected outputs for each test case.

- Use assertions, comparison functions, or other validation techniques to ensure that the overall behavior of the code aligns with the system requirements.

4. **Validate Results and Assertions:**

- Review the results of the system tests and verify that the code meets the system-level requirements defined in the test cases.

- Compare the actual outputs with the expected outputs, ensuring that the system behavior is consistent and accurate.

5. **Monitor and Debug:**

- Monitor the execution of the system tests and investigate any failures or unexpected behaviors.

- Debug the code, review logs, or use debugging tools to identify and resolve any issues encountered during system testing.

6. **Handle Dependencies:**

- Ensure that any required dependencies, such as the `pydicom` library, are properly installed and configured.

- Handle the dependencies by importing the required modules or mocking the behavior of the dependencies during system testing if necessary.

7. **Refine and Iterate**:

- Refine the system tests based on feedback and any issues encountered.

- Make adjustments or enhancements to the code to improve system behavior and ensure it meets the desired system-level requirements.

By following these steps, you can perform system testing for the provided code, ensuring that the system as a whole functions correctly and meets the specified requirements in various scenarios.

**5. Acceptance Testing:**

Acceptance testing is performed to determine whether the software meets the expectations and requirements of the end-users or stakeholders. It is often conducted by the client or end-users to validate the system's functionality against their specific needs. Acceptance testing can include user acceptance testing (UAT) and alpha/beta testing.For more and tutorial click here

[**https://www.guru99.com/user-acceptance-testing.html**](https://www.guru99.com/user-acceptance-testing.html)

Acceptance testing, also known as user acceptance testing (UAT), is conducted to verify if the system meets the requirements and expectations of the end-users or stakeholders. It involves testing the code to ensure that it meets the user's acceptance criteria and is ready for deployment. Here's an example of how you can perform acceptance testing for the given code**:**

**1. Define Acceptance Criteria:**

- Understand the requirements and expectations of the end-users or stakeholders.

- Define the acceptance criteria based on these requirements, which will serve as the basis for acceptance testing.

- For example, the acceptance criteria could include:

- The code should correctly process all DICOM files within the specified folder and subdirectories.

- The extracted metadata, such as acquisition date, manufacturer, model name, slice thickness, and X-ray tube current, should be accurate.

- The generated DataFrame should have the expected structure and contain valid data.

- The CSV output file should be successfully created and saved to the specified location.

**2. Prepare Test Data:**

-Set up a test environment that includes a representative set of DICOM files in the specified `folder\_path`.

- Ensure that the test data covers various scenarios and represents real-world usage.

**3. Perform Acceptance Tests:**

- Execute the code using the provided test data and observe its behaviour.

- Validate the actual outputs against the acceptance criteria defined earlier.

- Verify that the code meets the requirements and expectations of the end-users or stakeholders.

- Use assertions, comparisons, or other validation techniques to ensure that the code satisfies the acceptance criteria.

**4. Validate Results and Assertions:**

- Review the results of the acceptance tests and verify if the code meets the defined acceptance criteria.

- Compare the actual outputs with the expected outputs, ensuring that they align with the expectations of the end-users or stakeholders.

**5. Document Defects and Issues:**

- If any issues or defects are identified during acceptance testing, document them in a defect tracking system or an issue log.

- Include relevant details, such as steps to reproduce the issue, expected behavior, and actual behavior observed.

**6. Collaborate with Stakeholders:**

- Communicate the acceptance test results to the stakeholders or end-users.

- Discuss any issues or concerns identified during testing.

- Collaborate with stakeholders to address any identified issues and determine if the code meets their acceptance criteria.

**7. Refine and Iterate:**

- Refine the code based on the feedback and issues identified during acceptance testing.

- Make adjustments or enhancements to ensure that the code meets the acceptance criteria and satisfies the expectations of the end-users or stakeholders.

By following these steps, you can perform acceptance testing for the provided code, ensuring that it meets the acceptance criteria defined by the end-users or stakeholders and is ready for deployment.

**6. Regression Testing:**

Regression testing ensures that recent changes or updates in the codebase do not introduce new bugs or regressions that affect existing functionality. It involves retesting previously tested functionality to ensure that it still works as expected after modifications or additions. Regression testing helps maintain the stability and quality of the codebase over time. For tutorial and examples click here

<https://www.geeksforgeeks.org/software-engineering-regression-testing/>

<https://www.guru99.com/regression-testing.html>

Regression testing involves retesting the code to ensure that recent changes or additions to the codebase have not introduced new defects or caused any regressions in previously working functionality. It aims to identify any unintended side effects and ensure that the code behaves as expected even after modifications. Here's an example of how you can perform regression testing for the given code:

**1. Identify the Scope:**

- Determine the specific functionality or features affected by recent changes or additions.

- Focus on the areas of the code that are likely to be impacted.

**2. Create a Regression Test Suite:**

- Compile a set of test cases that cover the affected functionality as well as related functionality.

- Include test cases that previously passed to ensure they continue to function correctly after the recent changes.

- Consider both positive and negative test cases, boundary cases, and edge cases.

**3. Execute Regression Test Suite:**

- Run the regression test suite using the modified code.

- Verify that all the previously passing test cases still pass without any issues or regressions.

- Check for any new failures or unexpected behaviors that may have been introduced due to the recent changes.

**4. Investigate and Debug:**

- If any test cases fail or new issues arise, investigate the cause of the failures or regressions.

- Debug the code, review logs, or use debugging tools to identify the root cause of the issues.

**5. Fix and Rerun:**

- Address any identified issues by fixing the code or making necessary adjustments.

- After resolving the issues, rerun the affected test cases to ensure they pass successfully.

**6. Revalidate the System:**

- Execute the entire test suite, including both regression and other functional test cases, to ensure that the system functions as expected as a whole.

- Validate that the recent changes have not introduced any unforeseen issues in unrelated areas of the code.

**7. Automate Regression Tests (Optional):**

- Consider automating the regression test suite to streamline the regression testing process.

- Automating the tests can help identify regressions quickly and provide faster feedback on the stability of the codebase.

**8. Document and Track Defects:**

- Document any identified defects or issues in a defect tracking system or an issue log.

- Include relevant details such as steps to reproduce, expected behavior, and actual behavior observed.

**9. Repeat Regression Testing:**

- Continue to perform regression testing as new changes are made to the codebase.

- Regularly execute the regression test suite to ensure that any future modifications do not introduce regressions.

By following these steps, you can perform regression testing for the provided code, ensuring that recent changes or additions have not introduced any unexpected issues or regressions in the previously functioning code.

**7. Performance Testing:**

Performance testing focuses on evaluating the system's performance and scalability under different workloads and conditions. It measures response times, resource utilisation, throughput, and other performance metrics to identify bottlenecks, optimise performance, and ensure the system can handle expected loads for examples and tutorials,for more and tutorial click here

<https://granulate.io/blog/python-performance-testing-quick-tutorial-and-best-practices/>

<https://www.softwaretestinghelp.com/introduction-to-performance-testing-loadrunner-training-tutorial-part-1/>

Performance testing involves assessing the speed, responsiveness, scalability, and stability of an application or code under different workload conditions. In the given code, performance testing can help evaluate how the code performs when processing a large number of DICOM files. Here's an example of how you can perform performance testing for the provided code:

**1. Identify Performance Metrics:**

- Define the performance metrics that are important for your application, such as response time, throughput, or resource utilization.

- Determine the acceptable thresholds or goals for these metrics.

**2. Generate Performance Test Data:**

- Create a large number of DICOM files in the specified `folder\_path` to simulate a realistic workload.

- Ensure that the generated test data covers various file sizes, metadata variations, and other relevant factors.

3. **Design Performance Test Scenarios:**

- Define different test scenarios to simulate various workloads and stress levels.

- For example, test scenarios can include:

- Processing a moderate number of DICOM files within the specified folder.

- Simulating a high workload by processing a large number of DICOM files.

- Assessing the code's performance when handling a mix of small and large DICOM files.

- Varying the number of concurrent threads or processes executing the code.

**4. Execute Performance Tests:**

- Execute the performance test scenarios and monitor the performance metrics.

- Measure and record the response time, throughput, or other relevant metrics during each test run.

- Use tools such as JMeter, Locust, or custom scripts to simulate concurrent requests and measure performance.

**5. Analyze and Evaluate Results:**

- Analyze the performance test results and compare them against the defined performance metrics and thresholds.

- Identify any performance bottlenecks, such as slow response times or high resource utilization.

- Evaluate if the code meets the expected performance requirements or if optimizations are needed.

**6. Optimize and Retest:**

- If performance issues are identified, optimize the code by addressing the identified bottlenecks.

- Make improvements, such as optimizing algorithms, reducing I/O operations, or optimizing resource utilization.

- Retest the optimized code to validate the impact of the improvements on performance.

**7. Scalability Testing (Optional):**

- If scalability is a concern, consider performing scalability testing.

- Scale up the workload by increasing the number of DICOM files or simulating a higher number of concurrent users.

- Measure the performance metrics under increased workload to assess the code's scalability.

**8. Document and Report:**

- Document the performance test procedures, results, and any optimizations made.

- Report the findings, including any performance issues encountered and the measures taken to address them.

- Communicate the performance test results and recommendations to the relevant stakeholders.

By following these steps, you can perform performance testing for the provided code and evaluate how it performs under different workload conditions. This helps identify any performance issues and optimize the code for improved speed, responsiveness, and scalability.

**8. Security Testing:**

Security testing is conducted to identify vulnerabilities and weaknesses in the code and system that may lead to security breaches. It includes various techniques such as penetration testing, vulnerability scanning, and code reviews to uncover potential security risks and ensure the system's robustness against attacks,for more and tutorial click here

**.**[**https://www.geeksforgeeks.org/software-testing-security-testing/**](https://www.geeksforgeeks.org/software-testing-security-testing/)

[**https://www.guru99.com/what-is-security-testing.html**](https://www.guru99.com/what-is-security-testing.html)

Security testing aims to identify vulnerabilities, weaknesses, and potential security risks in an application or code. While the provided code does not inherently involve extensive security concerns, there are a few areas you can consider for security testing:

**1. Input Validation:**

- Verify that the code properly validates and sanitizes user input, especially when dealing with file paths and user-controlled data.

- Test for potential security risks such as path traversal, SQL injection, or other injection attacks.

**2. Access Control and Authorization:**

- Assess if the code enforces appropriate access controls to ensure that only authorized users can access and modify the data or perform specific operations.

- Test different user roles and privileges to ensure that access is granted or denied as intended.

**3. Secure File Handling:**

- Check if the code properly handles file operations, including reading, writing, and deleting files.

- Validate that appropriate permissions and access restrictions are in place to prevent unauthorized access or unintended file manipulation.

**4. Confidentiality and Privacy:**

- Review how sensitive data, such as patient information or personally identifiable information (PII), is handled and protected.

- Ensure that the code adheres to relevant privacy regulations and best practices for data protection.

**5. Error Handling and Logging:**

- Evaluate how the code handles errors and exceptions.

- Ensure that error messages do not reveal sensitive information and that appropriate logging mechanisms are in place for security-related events.

**6. Secure Deployment and Configuration:**

- Consider security aspects related to the deployment and configuration of the code.

- Validate that the code is deployed in a secure environment with proper security configurations and safeguards.

**7. Vulnerability Scanning and Code Analysis (Optional):**

- Utilize security scanning tools or code analysis tools to identify potential security vulnerabilities in the codebase.

- Conduct static code analysis or dynamic vulnerability scanning to detect common security issues.

**8. Secure Data Storage and Transmission (if applicable):**

- If the code involves storing or transmitting data, ensure that proper encryption and secure protocols are implemented to protect data in transit and at rest.

**9. Third-Party Dependencies:**

- Assess the security of any third-party libraries or dependencies used in the code.

- Ensure that these dependencies are up to date and do not introduce known security vulnerabilities.

**10. Security Best Practices**:

- Follow security best practices and coding standards to ensure a secure development approach.

- Consider guidelines such as OWASP (Open Web Application Security Project) recommendations and secure coding practices specific to the programming language used.

It's important to note that the security testing requirements may vary depending on the specific context and the sensitivity of the application or data involved. Consider involving a security professional or conducting a comprehensive security review based on your specific needs and industry best practices.

**9. Exploratory Testing:**

Exploratory testing involves ad-hoc and unscripted testing where testers explore the codebase and interact with the system to identify issues, validate functionality, and gain insights into the software. It relies on the tester's experience, intuition, and domain knowledge to uncover hidden defects and provide valuable feedback,for more and tutorial click here

**.**[**https://www.lambdatest.com/learning-hub/exploratory-testing**](https://www.lambdatest.com/learning-hub/exploratory-testing)

[**https://www.guru99.com/exploratory-testing.html**](https://www.guru99.com/exploratory-testing.html)

Exploratory testing is an approach where testers explore and experiment with the software/application without predefined test cases. It focuses on discovering issues, learning the behavior of the software, and gaining insights into its strengths and weaknesses. Here's how you can perform exploratory testing for the provided code:

1**. Understand the Code:**

- Familiarize yourself with the purpose, functionality, and requirements of the code.

- Analyze the code structure, logic, and dependencies.

**2. Identify Test Areas:**

- Identify different aspects of the code that can be tested, such as input handling, data processing, error handling, and output generation.

- Prioritize the areas based on their criticality and potential risks.

**3. Develop Test Ideas:**

- Generate various test ideas based on your understanding of the code and potential scenarios.

- Consider different inputs, edge cases, boundary values, and unexpected situations to test the code's behavior.

**4. Execute Test Ideas:**

- Start exploring the code by executing the test ideas you have identified.

- Perform actions, provide inputs, and observe the code's behavior.

- Take note of any unexpected behavior, errors, or issues encountered during execution.

**5. Document Issues:**

- Document any issues, anomalies, or observations made during the exploration.

- Provide detailed information about the test scenario, inputs used, and the observed behavior.

- Include steps to reproduce the issues if necessary.

**6. Investigate and Analyze**:

- Analyze the issues encountered during exploration.

- Determine the root cause of each issue and its potential impact on the functionality or quality of the code.

- Assess the severity and prioritize the issues based on their impact.

**7. Retest and Validate:**

- Once issues are resolved or changes are made to the code, perform retesting to ensure the issues have been addressed.

- Validate that the code behaves as expected after modifications.

**8. Iterative Exploration:**

- Continue exploring the code by repeating steps 3 to 7 with different test ideas and scenarios.

- Try new inputs, combinations, and edge cases to uncover additional issues and gain more insights into the code.

**9. Collaboration and Feedback:**

- Share your findings, observations, and suggestions with the development team or stakeholders.

- Collaborate with them to discuss issues, improvements, and potential changes to the code.

Exploratory testing helps uncover unexpected issues and provides valuable feedback on the code's behavior. It complements other testing approaches and can be performed iteratively throughout the development process. The goal is to improve the quality and reliability of the code by discovering issues that might have been missed with traditional test cases.

**\*The same Experiment has been done With** [**CHAT-GPT**](https://docs.google.com/document/d/1AFZD9oyXST_XXd6i0flyS1kWYFrSMUOGlelaS29Aatg/edit?usp=sharing)

**Here is the coomenting of our existing code by BARD**

# This code reads DICOM files from the specified directory and creates a CSV file with the following information:

#

# \* Acquisition Date

# \* Manufacturer Name

# \* Model Name

# \* Slice Thickness

# \* X-Ray Tube Current

# Import the necessary libraries.

import pydicom

import os

import pandas as pd

from tqdm import tqdm

# Change the path to the directory containing the DICOM files.

folder\_path = r"C:\Users\sayan\OneDrive\Desktop\DICOM"

# Initialize empty lists to store the data.

ad = []

manufacturer = []

model\_name = []

st = []

xtc = []

# Iterate through the files in the directory.

for root, folder, file in os.walk(folder\_path):

# Only process .dcm files.

for f in file:

if f.endswith('.dcm') and len(manufacturer) < 100000:

# Read the DICOM file.

f = os.path.join(root, f)

dcm = pydicom.dcmread(f)

# Add the data to the lists.

ad.append(dcm.AcquisitionDate)

manufacturer.append(dcm.Manufacturer)

model\_name.append(dcm.ManufacturerModelName)

st.append(dcm.SliceThickness)

xtc.append(dcm.XRayTubeCurrent)

# Print the number of manufacturers.

print(len(manufacturer))

# Create a Pandas DataFrame from the lists.

df = pd.DataFrame(list(zip(ad, manufacturer, model\_name, st, xtc)),

columns=['Acquisition Date', 'Manufacturer Name', 'Model Name', 'Slice Thickness', 'X-Ray Tube Current'])

# Save the DataFrame to a CSV file.

df.to\_csv('C:/Users/sayan/OneDrive/Desktop/ardi\_put.csv')

