Unit Testing Report

Please provide your GitHub repository link.

GitHub Repository URL: https://github.com/NickBland/2810ICT-milestone-2.git

1. **Test Summary**

Tested Functions	Test Functions
`initDatabase(filePath)`	`test_initDatabase_with_valid_csv()`
	`test_initDatabase_with_empty_csv()`
	`test_initDatabase_with_invalid_file_path()`
	`test_initDatabase_with_empty_file_path()`
	`test_initDatabase_with_nonexistent_file()`
`searchDatabase(filters, db)`	`test_searchDatabase_with_keyword()`
	`test_searchDatabase_with_nutrient_range()`
	`test_searchDatabase_with_nutrient_level()`
	`test_searchDatabase_with_invalid_minmax()`
	`test_searchDatabase_with_all_filters()`
	`test_searchDatabase_with_low_protein()`
	`test_searchDatabase_with_mid_protein()`
	`test_searchDatabase_with_high_protein()`
	`test_searchDatabase_with_no_filters()`
`addToComparison(selected_food, comparison_list)`	`test_addComparisonNone()`
	`test_addComparisonAddOne()`
	`test_addComparisonAddTwo()`
	`test_addComparisonAddSame()`
`displayResults(results, grid)`	`test_displayResults()`

2. **Test Case Details**

Test Case 1: testing valid CSV

- **Test Function/Module**
  - `test_initDatabase_with_valid_csv()`
- **Tested Function/Module**
  - `initDatabase(filePath)`
- **Description**
  - This function loads a CSV file into the database. The purpose of the test is to check whether the database is correctly loaded when given a valid CSV file.
- **1 - Valid Input and Expected Output**

1 - Valid Input	Expected Output
`filePath = "test_valid.csv`	`returns result as Database, error as None`

- **Code for the Test Function**

```python
def test_initDatabase_with_valid_csv(setup_files):
    valid_csv_path, _, _ = setup_files
    result, error = initDatabase(valid_csv_path)
    assert result is not None
    assert error is None
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filePath = "test_invalid.csv"`	`Returns None and raises 'ValueError("File path is not a CSV file")'`

- **2 - Code for the Test Function**

```python
def test_initDatabase_with_invalid_file_path(setup_files):
    _, _, invalid_file_path = setup_files
    result, error = initDatabase(invalid_file_path)
    assert result is None
    assert isinstance(error, ValueError)
    assert str(error) == "File path is not a CSV file"
```

Test Case 2: Testing Empty CSV

- **Test Function/Module**
  - `test_initDatabase_with_empty_csv()`
- **Tested Function/Module**
  - `initDatabase(filePath)`
- **Description**

  - This function tests the `initDatabase` function to ensure that it handles an empty CSV file correctly by returning an error.

- **1 - Valid Input and Expected Output**

Invalid Input	Expected Output
`filePath = "test_invalid.csv`	`Returns None and raises 'ValueError("File path is not a CSV file")'`

- **1 - Code for the Test Function**

```python
def test_initDatabase_with_empty_csv(setup_files):
    _, empty_csv_path, _ = setup_files
    result, error = initDatabase(empty_csv_path)
    assert result is None
    assert isinstance(error, ValueError)
    assert str(error) == "File is empty"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filePath = "test_empty.csv"`	`Returns None and raises 'ValueError("File is empty")'`

- **2 - Code for the Test Function**

```python
def test_initDatabase_with_empty_csv(setup_files):
    _, empty_csv_path, _ = setup_files
    result, error = initDatabase(empty_csv_path)
    assert result is None
    assert isinstance(error, ValueError)
    assert str(error) == "File is empty"
```

Test Case 3: testing initDatabase with an invalid file path

- **Test Function/Module**
  - `test_initDatabase_with_invalid_file_path()`
- **Tested Function/Module**
  - `initDatabase(filePath)`
- **Description**
  - This function attempts to initialize the database from an invalid file path. The purpose of the test is to ensure that the function handles invalid file paths correctly by returning an error.

- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filePath = "test_invalid.csvERROR"`	Returns `None` and raises `ValueError("File path is not a CSV file")`

- **1 - Code for the Test Function**

```
def test_initDatabase_with_invalid_file_path(setup_files):
    _, _, invalid_file_path = setup_files
    result, error = initDatabase(invalid_file_path)
    assert result is None
    assert isinstance(error, ValueError)
    assert str(error) == "File path is not a CSV file"
```

Test Case 4: testing `initDatabase` with an empty file path

- **Test Function/Module**
  - `test_initDatabase_with_empty_file_path()`
- **Tested Function/Module**
  - `initDatabase(filePath)`
- **Description**
  - This function attempts to initialize the database from an empty file path. The purpose of the test is to ensure that the function handles the empty file path correctly by raising a `FileNotFoundError`.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filePath = ""`	Returns `None` and raises `FileNotFoundError("File path is empty")`

- **1 - Code for the Test Function**

```
def test_initDatabase_with_empty_file_path():
    result, error = initDatabase("")
    assert result is None
    assert isinstance(error, FileNotFoundError)
    assert str(error) == "File path is empty"
```

Invalid Input	Expected Output

Invalid Input	Expected Output
`filePath = ""`	Raises `FileNotFoundError("File path is empty")`
`filePath = "/invalid/path/to/file.csv"`	Raises `FileNotFoundError("No such file or directory")`

```
def test_initDatabase_with_empty_file_path():
    result, error = initDatabase("")
    assert result is None
    assert isinstance(error, FileNotFoundError)
    assert str(error) == "File path is empty"
```

Test Case 5: testing `initDatabase` with a nonexistent file

- **Test Function/Module**
  - `test_initDatabase_with_nonexistent_file()`
- **Tested Function/Module**
  - `initDatabase(filePath)`
- **Description**
  - This function attempts to initialize the database with a file path that does not exist. The purpose of the test is to ensure that the function correctly handles the case where the file does not exist by raising a `FileNotFoundError`.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filePath = "nonexistent.csv"`	Returns `None` and raises `FileNotFoundError("No such file or directory")`

- **1 - Code for the Test Function**

```
def test_initDatabase_with_nonexistent_file():
    result, error = initDatabase("nonexistent.csv")
    assert result is None
    assert isinstance(error, FileNotFoundError)
    assert str(error) == "No such file or directory"
```

Invalid Input	Expected Output
`filePath = "/wrong/path/to/file.csv"`	Raises `FileNotFoundError("No such file or directory")`

```
def test_initDatabase_with_wrong_file_path():
    result, error = initDatabase("/wrong/path/to/file.csv")
    assert result is None
    assert isinstance(error, FileNotFoundError)
    assert str(error) == "No such file or directory"
```

Test Case 6: Test Case for Search Function with Keyword

- **Test Function/Module**
  - `test_searchDatabase_with_keyword()`
- **Tested Function/Module**
  - `searchDatabase(filters, db)`
- **Description**
  - This function searches the database for items based on the keyword filter. This test verifies that the correct entries are returned when filtering by a keyword.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = {"keyword": "banana"}`	`Returns DataFrame containing "banana" entries`

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_keyword(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {"keyword": "banana"}
    result = searchDatabase(filters, df)
    assert len(result) == 6
    assert result.iloc[0]["food"] == "banana cream pie"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filters = {"keyword": ""}`	Returns an empty DataFrame (no search results)
`filters = {"keyword": None}`	Returns an empty DataFrame (no search results)
`filters = {"keyword": 123}`	Returns an empty DataFrame (invalid search term)
`filters = {"keyword": "nonexistentfooditem"}`	Returns an empty DataFrame (no matches found)

- **2 - Code for the Test Function**

```
def test_searchDatabase_with_invalid_keyword(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
```

Test Case 7: Test case for search with Nutrient Range

- **Test Function/Module**
  - `test_searchDatabase_with_nutrient_range()`
- **Tested Function/Module**
  - `searchDatabase(filters, db)`
- **Description**
  - This function filters food items based on a specified nutrient range (e.g., protein). The test ensures that the function handles invalid range inputs, such as non-numeric values or nonsensical ranges, correctly.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = {"nutrient": "Protein", "min": 0.5, "max": 1.0}`	`Returns DataFrame containing foods within the protein range 0.5 to 1.0`

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_nutrient_range(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {"nutrient": "Protein", "min": "0.5", "max": "1.0"}
    result = searchDatabase(filters, df)
    assert len(result) == 206
    assert result.iloc[0]["food"] == "cream cheese"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filters = {"nutrient": "Protein", "min": "abc", "max": 1.0}`	Returns an empty DataFrame (invalid min value)
`filters = {"nutrient": "Protein", "min": 0.5, "max": "xyz"}`	Returns an empty DataFrame (invalid max value)
`filters = {"nutrient": "Protein", "min": 2.0, "max": 1.0}`	Returns an empty DataFrame (`min` > `max`)
`filters = {"nutrient": "NonexistentNutrient", "min": 0, "max": 10}`	Returns an empty DataFrame (invalid nutrient)

- **2 - Code for the Test Function**

```
def test_searchDatabase_with_invalid_nutrient_range(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
```

Test Case 8: testing searchDatabase with nutrient level

- **Test Function/Module**
  - `test_searchDatabase_with_nutrient_level()`
- **Tested Function/Module**
  - `searchDatabase(filters, db)`
- **Description**
  - This function filters food items by a specific nutrient level (low, mid, high). The purpose of the test is to ensure the function correctly filters items based on valid nutrient levels and handles invalid levels appropriately.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = {"level-protein": 3}`	Returns DataFrame filtered by high-protein items

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_nutrient_level(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {"level-protein": 3}
    result = searchDatabase(filters, df)
    assert len(result) == 5
    assert result.iloc[0]["food"] == "pork top loin roasts raw"
```

Test Case 9: testing searchDatabase with invalid `min` and `max` values

- **Test Function/Module**
  - `test_searchDatabase_with_invalid_minmax()`
- **Tested Function/Module**
  - `searchDatabase(filters, db)`

- **Description**
  - This function filters food items by a specified nutrient range using `min` and `max` values. The purpose of the test is to ensure the function handles invalid or non-numeric `min` and `max` values correctly.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = {"nutrient": "Protein", "min": 0.5, "max": 1.0}`	Returns DataFrame with items that fall in this protein range

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_nutrient_range(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {"nutrient": "Protein", "min": 0.5, "max": 1.0}
    result = searchDatabase(filters, df)
    assert len(result) == 206
    assert result.iloc[0]["food"] == "cream cheese"
```

Test Case 10: testing searchDatabase with all filters applied

- **Test Function/Module**
  - `test_searchDatabase_with_all_filters()`
- **Tested Function/Module**
  - `searchDatabase(filters, db)`
- **Description**
  - This function applies multiple filters (keyword, nutrient ranges, nutrient levels) to search food items from the database. The purpose of the test is to check if the function handles valid and invalid input combinations for multiple filters correctly.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = { "keyword": "", "nutrient": "", "min": "", "max": "", "level-protein": 1, "level-sugar": 1, "level-carb": 2, "level-fat": 1, "level-nutri": 1}`	Returns a DataFrame filtered by all these filters

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_all_filters(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {
        "keyword": "",
        "nutrient": "",
        "min": "",
        "max": "",
        "level-protein": 1,
        "level-sugar": 1,
        "level-carb": 2,
        "level-fat": 1,
        "level-nutri": 1
    }
    result = searchDatabase(filters, df)
    assert len(result) == 3
    assert result.iloc[0]["food"] == "tapioca pearls"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	**Expected Output**
`filters = {"keyword": "invalid", "level-protein": 0}`	Returns an empty DataFrame (invalid keyword and level)
`filters = {"nutrient": "Nonexistent", "level-protein": 4}`	Returns an empty DataFrame (invalid nutrient and level)
`filters = {"keyword": 123, "nutrient": "Protein", "min": "abc", "max": 1.0}`	Returns full DataFrame (invalid min value)

- **2 - Code for the Test Function**

```
def test_searchDatabase_with_invalid_all_filters(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
```

Test Case 11: testing searchDatabase with low protein level

- **Test Function/Module**
    - `test_searchDatabase_with_low_protein()`
- **Tested Function/Module**
    - `searchDatabase(filters, db)`
- **Description**

    - This function filters food items with low protein levels. The purpose of the test is to ensure the function correctly identifies and returns food items with low protein when `level-protein` is set to 1.

- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = {"level-protein": 1}`	Returns DataFrame filtered by low-protein items

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_low_protein(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {"level-protein": 1}
    result = searchDatabase(filters, df)
    assert len(result) == 2383
    assert result.iloc[0]["food"] == "cream cheese"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filters = {"level-protein": 0}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": -1}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": 4}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": "abc"}`	Returns an empty DataFrame (non-numeric protein level)

```
def test_searchDatabase_with_invalid_low_protein(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
```

Test Case 12: testing `searchDatabase` with mid protein level

- **Test Function/Module**
    - `test_searchDatabase_with_mid_protein()`
- **Tested Function/Module**
    - `searchDatabase(filters, db)`
- **Description**
    - This function filters food items with mid protein levels. The purpose of the test is to ensure the function correctly identifies and returns food items with mid protein when `level-protein` is set

to 2.

- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = {"level-protein": 2}`	Returns DataFrame filtered by mid-protein items

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_mid_protein(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {"level-protein": 2}
    result = searchDatabase(filters, df)
    assert len(result) == 7
    assert result.iloc[0]["food"] == "pork backribs raw"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filters = {"level-protein": 0}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": -1}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": 4}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": "xyz"}`	Returns an empty DataFrame (non-numeric protein level)

```
def test_searchDatabase_with_invalid_mid_protein(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
```

Test Case 13: testing searchDatabase with high protein level

- **Test Function/Module**
  - `test_searchDatabase_with_high_protein()`
- **Tested Function/Module**
  - `searchDatabase(filters, db)`
- **Description**
  - This function filters food items with high protein levels. The purpose of the test is to ensure the function correctly identifies and returns food items with high protein when `level-protein` is set to 3.

- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`filters = {"level-protein": 3}`	Returns DataFrame filtered by high-protein items

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_high_protein(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {"level-protein": 3}
    result = searchDatabase(filters, df)
    assert len(result) == 5
    assert result.iloc[0]["food"] == "pork top loin roasts raw"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filters = {"level-protein": 0}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": -1}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": 4}`	Returns an empty DataFrame (invalid protein level)
`filters = {"level-protein": "abc"}`	Returns an empty DataFrame (non-numeric protein level)

```
def test_searchDatabase_with_invalid_high_protein(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
```

Test Case 14: testing searchDatabase with no filters applied

- **Test Function/Module**
  - `test_searchDatabase_with_no_filters()`
- **Tested Function/Module**
  - `searchDatabase(filters, db)`
- **Description**
  - This function searches the database with no filters applied. The purpose of the test is to ensure that when no filters are provided, the function returns the entire dataset.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output

Valid Input	Expected Output
`filters = {}`	Returns the entire DataFrame with all food items

- **1 - Code for the Test Function**

```
def test_searchDatabase_with_no_filters(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    filters = {}
    result = searchDatabase(filters, df)
    assert len(result) == 2396
    assert result.iloc[0]["food"] == "cream cheese"
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`filters = {"keyword": None}`	Returns the entire DataFrame (no filters applied)
`filters = {"nutrient": None, "level-protein": None}`	Returns the entire DataFrame (no filters applied)

```
def test_searchDatabase_with_no_filters_invalid(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
```

Test Case 15: testing `displayResults(results, grid)`

- **Test Function/Module**
  - `test_displayResults()`
- **Tested Function/Module**
  - `displayResults(results, grid)`
- **Description**
  - This function displays the results in a grid. The purpose of the test is to verify that the function properly updates the grid with the provided data.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`results = DataFrame with food items`, `grid = Mock grid object`	The grid is updated with the contents of the results DataFrame

- **1 - Code for the Test Function**

```python
def test_displayResults(fetch_real_data, mocker):
    data = fetch_real_data
    df = pd.DataFrame(data)

    # Create a mock grid
    # grid = SimpleMock()
    grid = mocker.MagicMock()

    # Call the displayResults function
    displayResults(df, grid)

    # Assertions to check if the grid methods were called correctly
    grid.ClearGrid.assert_called_once()
    grid.SetTable.assert_called_once()
    grid.AutoSizeColumns.assert_called_once()
    grid.HideRowLabels.assert_called_once()
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`results = None`, `grid = Mock grid object`	Grid remains unchanged; no exceptions raised
`results = Empty DataFrame`, `grid = Mock grid object`	Grid is cleared but no rows are populated

```python
def test_displayResults(fetch_real_data, mocker):
    data = fetch_real_data
    df = pd.DataFrame(data)

    # Create a mock grid
    # grid = SimpleMock()
    grid = mocker.MagicMock()

    # Call the displayResults function
    displayResults(df, grid)

    # Assertions to check if the grid methods were called correctly
    grid.ClearGrid.assert_called_once()
    grid.SetTable.assert_called_once()
    grid.AutoSizeColumns.assert_called_once()
    grid.HideRowLabels.assert_called_once()
```

Test Case 16: testing `addToComparison` with no food selected

- **Test Function/Module**

    - `test_addComparisonNone()`
- **Tested Function/Module**
    - `addToComparison(selected_food, comparison_list)`
- **Description**
    - This function adds a selected food to the comparison list. The purpose of the test is to ensure the function behaves correctly when no food is selected (i.e., when `selected_food` is `None`).
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`selected_food = None`, `comparison_list = []`	Comparison list remains unchanged (empty)

- **1 - Code for the Test Function**

```python
def test_addComparisonNone(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    comparison_list = []

    # Empty selected food
    selected_food = None
    currently_selected_food = selected_food
    addToComparison(currently_selected_food, comparison_list)
    assert comparison_list == []  # Ensure the comparison list is unchanged
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`selected_food = None`, `comparison_list = None`	Raises `TypeError` or returns unchanged comparison list
`selected_food = 12345`, `comparison_list = []`	Does not add the invalid food type, list remains unchanged

Test Case 17: testing addToComparison with one food item

- **Test Function/Module**
    - `test_addComparisonAddOne()`
- **Tested Function/Module**
    - `addToComparison(selected_food, comparison_list)`
- **Description**

    - This function adds a selected food to the comparison list. The purpose of the test is to ensure that the function properly adds a single food item to the comparison list.

- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`selected_food = df[df["food"] == "cream cheese"]`, `comparison_list = []`	Comparison list contains "cream cheese"

- **1 - Code for the Test Function**

```python
def test_addComparisonAddOne(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    comparison_list = []

    selected_food = df[df["food"] == "cream cheese"]
    currently_selected_food = selected_food
    addToComparison(currently_selected_food, comparison_list)
    assert "cream cheese" in comparison_list  # Ensure the food is added
```

- **2 - Invalid Input and Expected Output**

Invalid Input	Expected Output
`selected_food = 12345`, `comparison_list = []`	Does not add invalid food type, list remains unchanged
`selected_food = None`, `comparison_list = []`	List remains unchanged

```python
def test_addComparisonAddOne_invalid(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    comparison_list = []
```

Test Case 18: testing addToComparison with two different food items

- **Test Function/Module**
    - `test_addComparisonAddTwo()`
- **Tested Function/Module**
    - `addToComparison(selected_food, comparison_list)`
- **Description**

    - This function adds multiple selected food items to the comparison list. The purpose of the test is to ensure the function correctly adds two different food items.

- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`selected_food = df[df["food"] == "cream cheese"]`, `comparison_list = []`	Comparison list contains "cream cheese" and "gruyere cheese"

- **1 - Code for the Test Function**

```python
def test_addComparisonAddTwo(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    comparison_list = []

    # add two different foods
    selected_food = df[df["food"] == "cream cheese"]
    currently_selected_food = selected_food
    comparison_list = addToComparison(currently_selected_food, comparison_list)

    selected_food = df[df["food"] == "gruyere cheese"]
    currently_selected_food = selected_food
    addToComparison(currently_selected_food, comparison_list)

    assert "cream cheese" in comparison_list
    assert "gruyere cheese" in comparison_list  # Ensure both foods are added
```

Invalid Input	Expected Output
`selected_food = 12345`, `comparison_list = []`	Does not add invalid food type, list remains unchanged
`selected_food = None`, `comparison_list = []`	List remains unchanged

```python
def test_addComparisonAddTwo_invalid(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    comparison_list = []
```

Test Case 19: testing addToComparison with the same food item added twice

- **Test Function/Module**
    - `test_addComparisonAddSame()`
- **Tested Function/Module**

    - `addToComparison(selected_food, comparison_list)`
- **Description**
    - This function adds the same selected food item multiple times to the comparison list. The purpose of the test is to ensure that the function does not allow duplicate entries.
- **1 - Valid Input and Expected Output**

Valid Input	Expected Output
`selected_food = df[df["food"] == "cream cheese"]`, `comparison_list = []`	Comparison list contains "cream cheese" only once

- **1 - Code for the Test Function**

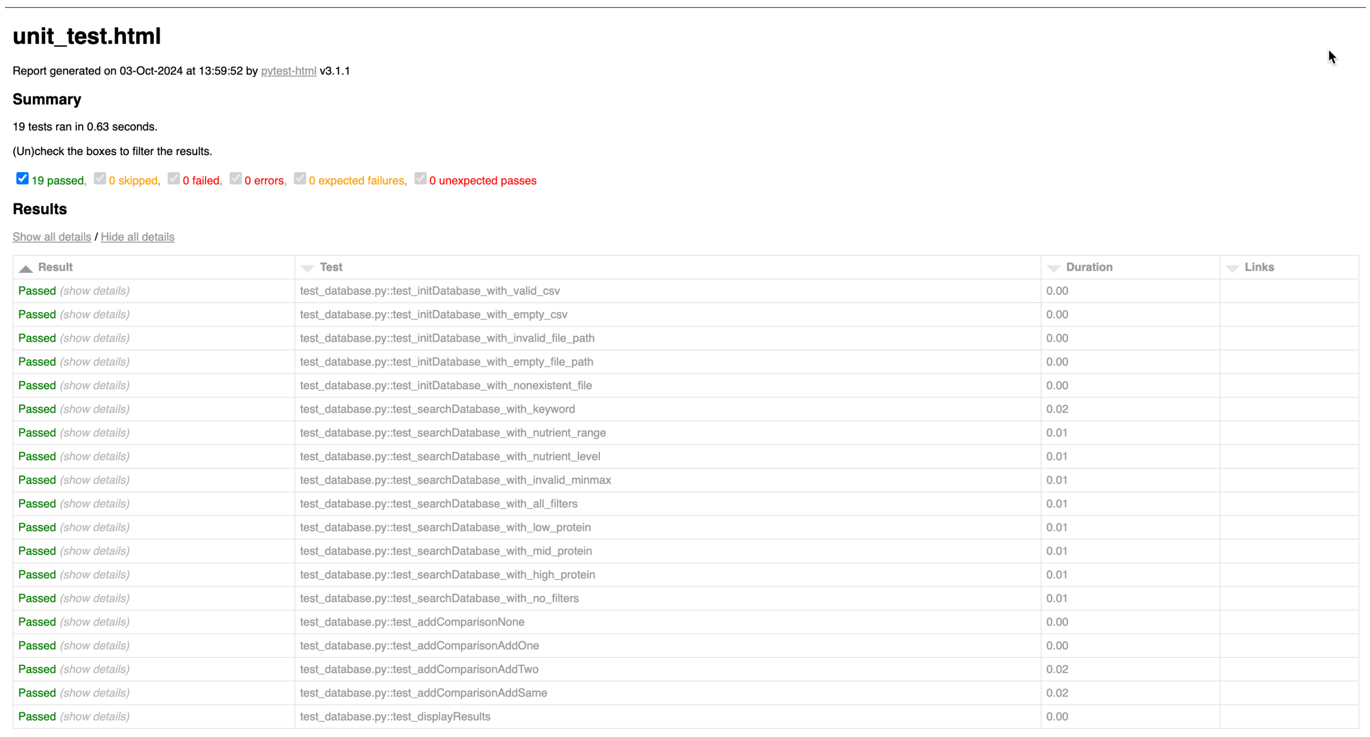
```python
def test_addComparisonAddSame(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    comparison_list = []

    # Add two of the same food
    selected_food = df[df["food"] == "cream cheese"]
    currently_selected_food = selected_food
    comparison_list = addToComparison(currently_selected_food, comparison_list)

    selected_food = df[df["food"] == "cream cheese"]
    currently_selected_food = selected_food
    addToComparison(currently_selected_food, comparison_list)

    assert comparison_list.count("cream cheese") == 1  # Ensure no duplicates
```

Invalid Input	Expected Output
`selected_food = 12345`, `comparison_list = []`	Does not add invalid food type, list remains unchanged
`selected_food = None`, `comparison_list = []`	List remains unchanged

```python
def test_addComparisonAddSame_invalid(fetch_real_data):
    data = fetch_real_data
    df = pd.DataFrame(data)
    comparison_list = []
```

3. **Testing Report Summary**

Here is a screenshot of unit_test.html showing the results of all the above tests.



unit_test.html

Report generated on 03-Oct-2024 at 13:59:52 by pytest-html v3.1.1

Summary

19 tests ran in 0.63 seconds.

(Un)check the boxes to filter the results.

☑ 19 passed, ☑ 0 skipped, ☑ 0 failed, ☑ 0 errors, ☑ 0 expected failures, ☑ 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration	Links
Passed *(show details)*	test_database.py::test_initDatabase_with_valid_csv	0.00	
Passed *(show details)*	test_database.py::test_initDatabase_with_empty_csv	0.00	
Passed *(show details)*	test_database.py::test_initDatabase_with_invalid_file_path	0.00	
Passed *(show details)*	test_database.py::test_initDatabase_with_empty_file_path	0.00	
Passed *(show details)*	test_database.py::test_initDatabase_with_nonexistent_file	0.00	
Passed *(show details)*	test_database.py::test_searchDatabase_with_keyword	0.02	
Passed *(show details)*	test_database.py::test_searchDatabase_with_nutrient_range	0.01	
Passed *(show details)*	test_database.py::test_searchDatabase_with_nutrient_level	0.01	
Passed *(show details)*	test_database.py::test_searchDatabase_with_invalid_minmax	0.01	
Passed *(show details)*	test_database.py::test_searchDatabase_with_all_filters	0.01	
Passed *(show details)*	test_database.py::test_searchDatabase_with_low_protein	0.01	
Passed *(show details)*	test_database.py::test_searchDatabase_with_mid_protein	0.01	
Passed *(show details)*	test_database.py::test_searchDatabase_with_high_protein	0.01	
Passed *(show details)*	test_database.py::test_searchDatabase_with_no_filters	0.01	
Passed *(show details)*	test_database.py::test_addComparisonNone	0.00	
Passed *(show details)*	test_database.py::test_addComparisonAddOne	0.00	
Passed *(show details)*	test_database.py::test_addComparisonAddTwo	0.02	
Passed *(show details)*	test_database.py::test_addComparisonAddSame	0.02	
Passed *(show details)*	test_database.py::test_displayResults	0.00	