cogs18_final_project

December 10, 2024

1 Budget Expense Tracker

This tracker will help manage your expenses while accounting for how inflation may affect costs over time.

This tracker assumes all expenses are spent in USD and that expenses are adjusted to annual inflation rates from 2000 to 2024.

1.1 Features

1.

- view monthly expenses
- expense categorized by their choosing (i.e. food, utilities, entertainment, rent)

2.

- show monthly or yearly trends in spending across categories

3.

- calculate inflation-adjusted expenses

1.2 Outline

1.2.1 Modules

- `matplotlib` - used to display graphs of spending trends
- `pandas` - to work with data tables

1.2.2 1. Expense Class

stores individual expenses and its details

Attributes:

- `amount` - float - cost of individual expense
- `category` - string - can be food, entertainment, etc
- `month` - integer
- `year` - integer

Methods:

- `adjust_for_inflation(self, inflation_rate)` : adjust expense based on inflation rate
- `get_lst` - store attributes in a list so it can be used in the Tracker class for further analysis

1.2.3 2. Tracker Class

manages and analyzes all expenses

Attributes:

- `expenses` - empty dataframe before all expenses are added

Methods:

- `add_expense(self, expense)`: add new expense to the tracker
- `get_expense_by_category(self, category)`: filter expense by category
- `get_monthly_summary(self, month, year)`: show summary of expenses for a specific month of a year, using a bar graph
- `get_yearly_summary(self, year)`: show summary of expenses for a specific year, using a bar graph
- `calc_inflation_adj_expenses(self)` - adjusts all expenses for inflation, using inflation average

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('us_inflation_rates_2000_to_2024.csv', index_col = 0)
```

[2]:
```
class Expense:

    def __init__(self, amount, category, month, year):
        self.amount = float(amount)
        self.category = str(category)
        self.month = month.capitalize()[:3]
        self.year = int(year)
    def adjust_for_inflation(self):
        inflation_rate = df.loc[self.year, self.month]
        adjusted_expense = self.amount * (1 + (inflation_rate/100))

        return f'your inflation adjusted expense is {adjusted_expense}'

    # stored in list to be used in Tracker for further analysis
    def get_lst(self):
        return [self.amount, self.category, self.month, self.year]
```

[3]:
```
# quick tests for the Expense class
my_expenses = Expense(712.73, 'shopping', 'November', 2021)
my_expenses.adjust_for_inflation()
```

[3]:
```
'your inflation adjusted expense is 761.19564'
```

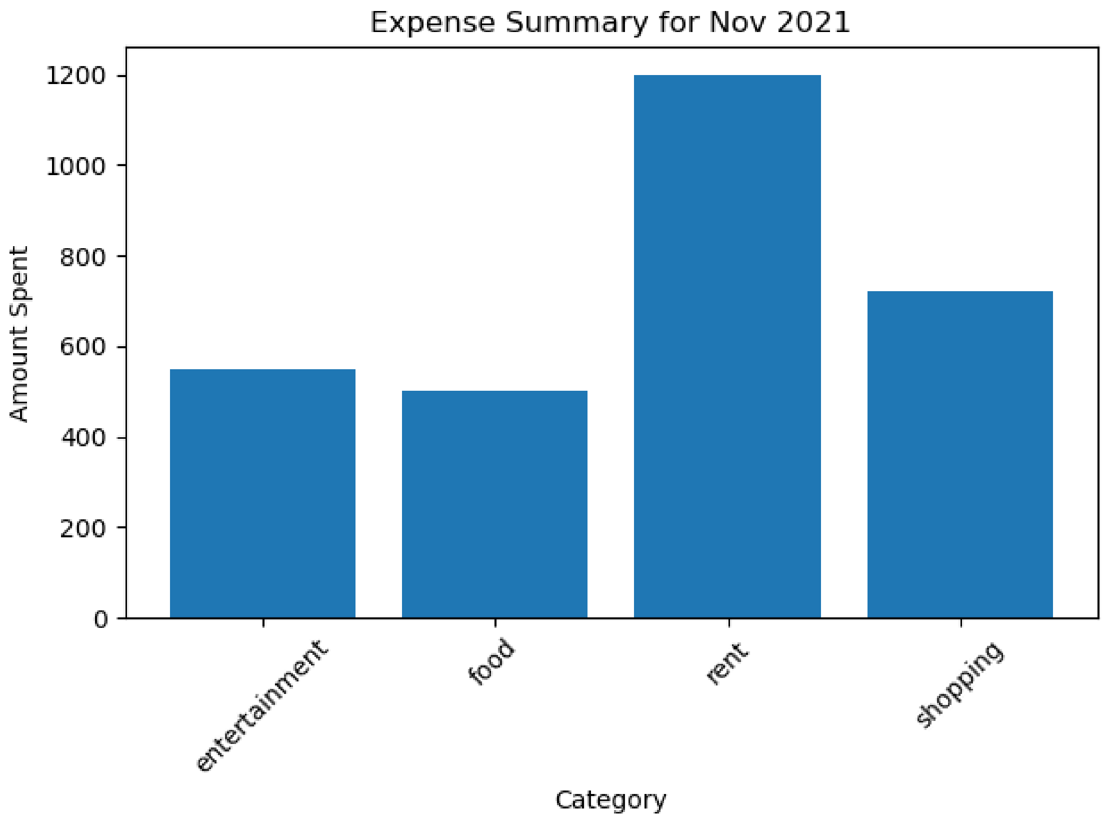
[4]:
```
class Tracker:

    def __init__(self):
        self.expenses = pd.DataFrame({'Amount': pd.Series(dtype='float'),
                                      'Category': pd.Series(dtype='str'),
                                      'Month': pd.Series(dtype='str'),
                                      'Year': pd.Series(dtype='int')})

    def add_expenses(self, expense):
        # create dataframe for new expense to be added to the main dataframe
        new_expense = pd.DataFrame([expense.get_lst()], columns=self.expenses.
 ↪columns)
        self.expenses = pd.concat([self.expenses, new_expense],␣
 ↪ignore_index=True)

    def get_expense_by_category(self, category):
        temp = self.expenses[self.expenses['Category'] == category]
        print(temp)

    def get_monthly_summary(self, month, year):
        # filter by both month and year
        filtered_expenses = self.expenses[
            (self.expenses['Month'] == month.capitalize()[:3]) &
            (self.expenses['Year'] == year)
        ]

        if filtered_expenses.empty == True:
            raise ValueError(f"No data found for {month} {year}")

        # group by category
        category_totals = filtered_expenses.groupby('Category')['Amount'].sum()

        plt.bar(category_totals.index, category_totals.values)
        plt.xlabel('Category')
        plt.ylabel('Amount Spent')
        plt.title(f'Expense Summary for {month.capitalize()} {year}')
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()

    def get_yearly_summary(self, year):
        filtered_expenses = self.expenses[
            (self.expenses['Year'] == year)
        ]

        if filtered_expenses.empty == True:
            raise ValueError(f"No data found for {month} {year}")

        yearly_totals = filtered_expenses.groupby('Month')['Amount'].sum().
↪reindex(
                                                   ['Jan', 'Feb', 'Mar',␣
↪'Apr', 'May',
                                                    'Jun',  'Jul', 'Aug',␣
↪'Sep', 'Oct',
                                                    'Nov', 'Dec'])

        plt.bar(yearly_totals.index, yearly_totals.values)
        plt.xlabel('Category')
        plt.ylabel('Amount Spent')
        plt.title(f'Expense Summary for {year}')
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()

    def calc_inflation_adj_expenses(self, year):
        # create empty dataframe to avoid UnboundLocalError
        calc_expenses = pd.DataFrame(columns=self.expenses.columns)

        # sort dataframe by year
        if year < 2024:
            calc_expenses = self.expenses[(self.expenses['Year'] == year)]
            calc_expenses = calc_expenses.copy()

            if calc_expenses.empty == True:
                raise ValueError(f"No data found for {year}.")

            inflation_avg = df.loc[year, 'Ave'] / 100
            # apply inflation avg of that year to all cells in the 'Amount'␣
↪column
            calc_expenses['Adjusted Amount'] = calc_expenses['Amount'] * (1 +␣
↪inflation_avg)

        else:
            print(f"No inflation average calculated yet for {year}.")

        return calc_expenses
```

[5]:
```
# example
t = Tracker()
t.add_expenses(Expense(712.73, 'shopping', 'November', 2021))
t.add_expenses(Expense(7.73, 'shopping', 'November', 2021))
t.add_expenses(Expense(550, 'entertainment', 'November', 2021))
t.add_expenses(Expense(1200, 'rent', 'November', 2021))
t.add_expenses(Expense(500, 'food', 'November', 2021))
t.add_expenses(Expense(300, 'food', 'October', 2021))
t.add_expenses(Expense(1000, 'rent', 'December', 2021))
t.add_expenses(Expense(1100, 'rent', 'September', 2020))
t.add_expenses(Expense(1200, 'rent', 'August', 2020))
t.add_expenses(Expense(1200, 'rent', 'July', 2020))
```

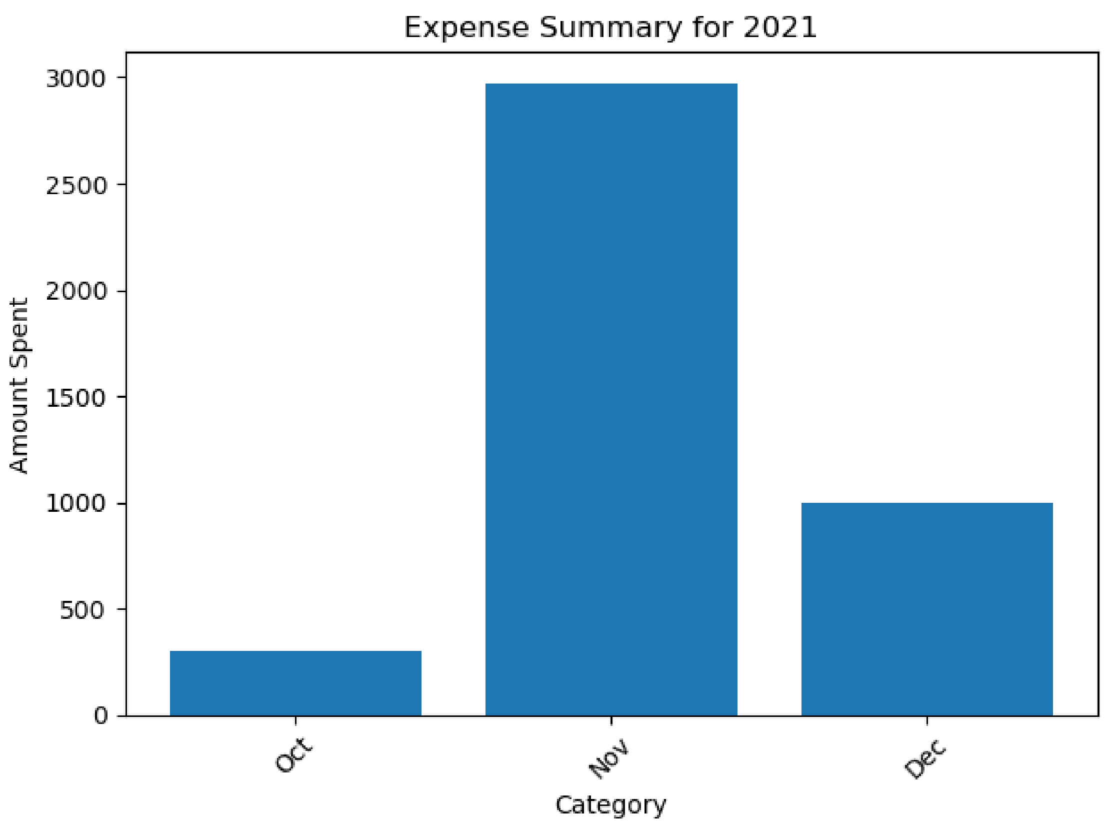
[6]:
```
t.get_expense_by_category('food')
```

```
   Amount Category Month  Year
4   500.0     food   Nov  2021
5   300.0     food   Oct  2021
```

[7]:
```
t.get_monthly_summary('Nov', 2021)
```



[8]:
```
t.get_yearly_summary(2021)
```



```
[9]: t.calc_inflation_adj_expenses(2021)
```

```
[9]:     Amount       Category Month  Year  Adjusted Amount
     0   712.73       shopping   Nov  2021        746.22831
     1     7.73       shopping   Nov  2021          8.09331
     2   550.00  entertainment   Nov  2021        575.85000
     3  1200.00           rent   Nov  2021       1256.40000
     4   500.00           food   Nov  2021        523.50000
     5   300.00           food   Oct  2021        314.10000
     6  1000.00           rent   Dec  2021       1047.00000
```

```
[10]: t.calc_inflation_adj_expenses(2024)
```

```
No inflation average calculated yet for 2024.
```

```
[10]: Empty DataFrame
      Columns: [Amount, Category, Month, Year]
      Index: []
```

1.2.4 Creating Unit Tests

```
[11]: expense = Expense(100, 'food', 'March', 2019)
```

```
[12]: class TestExpense:

          def test_init(self):
              # checks if Expense object is initialized correctly
              assert expense.amount == 100
              assert expense.category == 'food'
              assert expense.month == 'Mar'
              assert expense.year == 2019

          def test_adjust_for_inflation(self):
              # create global df as a mock for the df containing the real inflation␣
       ↪rates
              global df

              df = pd.DataFrame(
                  {
                      'Jan': [2],
                      'Feb': [2.5],
                      'Mar': [3],
                      'Apr': [1.8],
                      'May': [2.3],
                      'Jun': [2.7],
                      'Jul': [2.2],
                      'Aug': [1.9],
                      'Sep': [2.6],
                      'Oct': [2.8],
                      'Nov': [3.1],
                      'Dec': [3.0],
                      'Ave': [3]
                  },
                  index=[2019],
              )

              expense = Expense(100, 'food', 'March', 2019)
              result = expense.adjust_for_inflation()

              # checks if output and inflation calculation are correct
              assert 'your inflation adjusted expense is', result == True
              assert result == 'your inflation adjusted expense is 103.0'

          def test_get_lst(self):
              # checks if Expense attributes are correctly represented in a list
              assert type(expense.get_lst()) == list
```

```
[13]: test_expense = TestExpense()
```

```
[14]: test_expense.test_init()
test_expense.test_adjust_for_inflation()
test_expense.test_get_lst()
```

1.2.5 Note on Test Cases

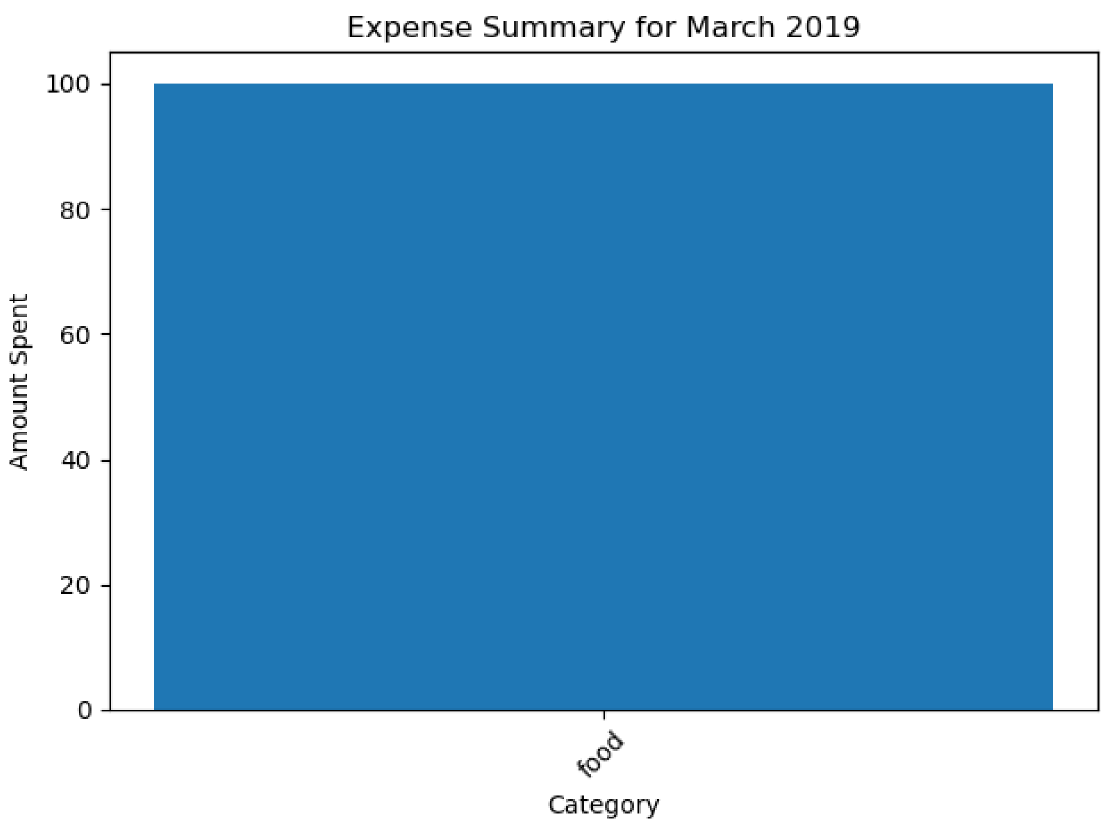
In this test class, I create test cases that test valid and invalid data inputs and see if the error message for invalid data inputs will show up.

```
[15]: class TestTracker:
    def setup_method(self):
        """dummy dataframe"""
        self.mock_data = pd.DataFrame({
            'Amount': [100.0, 200.0, 300.0],
            'Category': ['food', 'travel', 'food'],
            'Month': ['Mar', 'Apr', 'May'],
            'Year': [2019, 2019, 2020]
        })
        self.tracker = Tracker()
        self.tracker.expenses = self.mock_data.copy()

    def test_tracker_init(self):
        # checks if DataFrame to store expenses is initialized correctly
        column_names = ['Amount', 'Category', 'Month', 'Year']

        assert type(self.tracker.expenses) == pd.DataFrame
        assert list(self.tracker.expenses.columns) == column_names
        # pandas 64 bit float thingy
        assert self.tracker.expenses['Amount'].dtype == 'float64'
        # pandas uses object for strings
        assert self.tracker.expenses['Category'].dtype == object
        assert self.tracker.expenses['Month'].dtype == object
        assert self.tracker.expenses['Year'].dtype == int

    def test_add_expenses(self):
        # reset dummy dataframe to empty before checking its empty
        self.tracker.expenses = pd.DataFrame(columns=['Amount', 'Category',␣
↪'Month', 'Year'])

        # check if tracker instance is empty before expense is added
        assert self.tracker.expenses.empty == True

        # expense object
        expense = Expense(100.0, 'food', 'Mar', 2019)
        self.tracker.add_expenses(expense)
        # checks if the expense appended is appended to the main dataframe
        assert len(self.tracker.expenses) == 1

    def test_get_expense_by_category(self):
        # defining category, filtering df
        category = 'food'
        filtered_expenses = self.tracker.expenses[self.tracker.
↪expenses['Category'] == category]

        # checking that rows match category
        assert all(filtered_expenses['Category'] == category)

    def test_get_monthly_summary(self):
        # test 1: normal case - valid month and year
        try:
            self.tracker.get_monthly_summary('March', 2019)
        except ValueError:
            assert False, "ValueError raised unexpectedly for valid input."

        # test 2: invalid case - no data for specified month and year
        try:
            self.tracker.get_monthly_summary('June', 2019)
            assert False, "ValueError was not raised for missing data."
        except ValueError as e:
            assert str(e) == "No data found for June 2019.", "Unexpected error␣
↪message."

    def test_get_yearly_summary(self):
        # test 1: normal case - valid year
        try:
            self.tracker.get_yearly_summary(2019)
        except ValueError:
            assert False, "ValueError raised unexpectedly for valid input."

        # test 2: invalid case - no data for specified year
        try:
            self.tracker.get_yearly_summary(2027)
            assert False, "ValueError was not raised for missing data."
        except ValueError as e:
            assert str(e) == "No data found for 2027.", "Unexpected error␣
↪message."

    def test_calc_inflation_adj_expenses(self):
        # test 1: normal case - valid year
        try:
            print(self.tracker.calc_inflation_adj_expenses(2019))
        except ValueError:
            assert False, "ValueError raised unexpectedly for valid input."

        # test 2: invalid case - no data for specified year
        try:
            self.tracker.calc_inflation_adj_expenses(2027)
            assert False, "ValueError was not raised for missing data."
        except ValueError as e:
            assert str(e) == "No data found for 2027.", "Unexpected error␣
↪message."
```

[16]:
```
test_t = TestTracker()
test_t.setup_method()  # Initialize mock data for each test
test_t.test_tracker_init()
test_t.setup_method()
test_t.test_add_expenses()
test_t.setup_method()
test_t.test_get_expense_by_category()
```

```
/tmp/ipykernel_492/86803614.py:12: FutureWarning: The behavior of DataFrame
concatenation with empty or all-NA entries is deprecated. In a future version,
this will no longer exclude empty or all-NA columns when determining the result
dtypes. To retain the old behavior, exclude the relevant entries before the
concat operation.
  self.expenses = pd.concat([self.expenses, new_expense], ignore_index=True)
```

[17]:
```
test_t.setup_method()
test_t.test_get_monthly_summary()
```



```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[15], line 57, in TestTracker.test_get_monthly_summary(self)
     56 try:
---> 57     self.tracker.get_monthly_summary('June', 2019)
     58     assert False, "ValueError was not raised for missing data."

Cell In[4], line 26, in Tracker.get_monthly_summary(self, month, year)
     25 if filtered_expenses.empty == True:
---> 26     raise ValueError(f"No data found for {month} {year}")
     28 # group by category

ValueError: No data found for June 2019

During handling of the above exception, another exception occurred:

AssertionError                            Traceback (most recent call last)
Cell In[17], line 2
      1 test_t.setup_method()
----> 2 test_t.test_get_monthly_summary()
```

```
Cell In[15], line 60, in TestTracker.test_get_monthly_summary(self)
     58     assert False, "ValueError was not raised for missing data."
     59 except ValueError as e:
---> 60     assert str(e) == "No data found for June 2019.", "Unexpected error␣
  ↪message."

AssertionError: Unexpected error message.
```

[18]:
```
test_t.setup_method()
test_t.test_calc_inflation_adj_expenses()
```

```
   Amount Category Month  Year  Adjusted Amount
0   100.0     food   Mar  2019            103.0
1   200.0   travel   Apr  2019            206.0
No inflation average calculated yet for 2027.
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[18], line 2
      1 test_t.setup_method()
----> 2 test_t.test_calc_inflation_adj_expenses()

Cell In[15], line 86, in TestTracker.test_calc_inflation_adj_expenses(self)
     84 try:
     85     self.tracker.calc_inflation_adj_expenses(2027)
---> 86     assert False, "ValueError was not raised for missing data."
     87 except ValueError as e:
     88     assert str(e) == "No data found for 2027.", "Unexpected error␣
  ↪message."

AssertionError: ValueError was not raised for missing data.
```

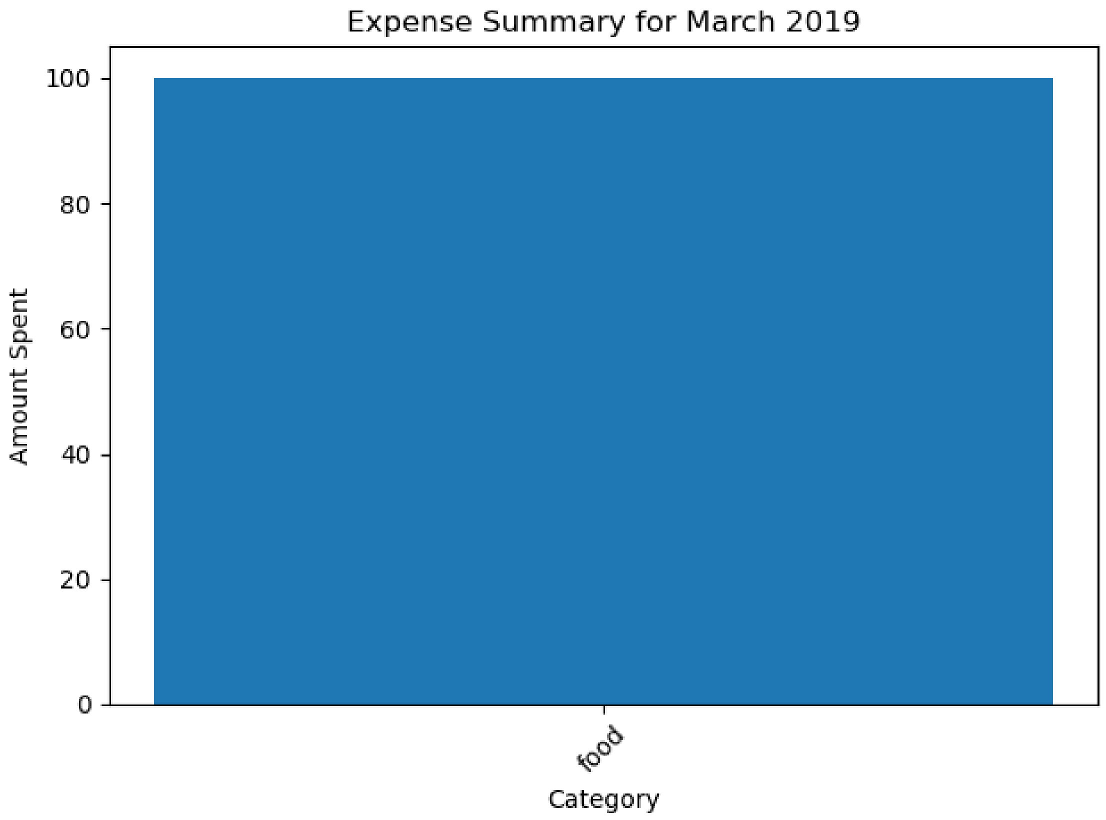
1.2.6 Running Pytest

Note: For test_functions, an error should show up because I used try/except for a test case with invalid data to check that the error message would show up.

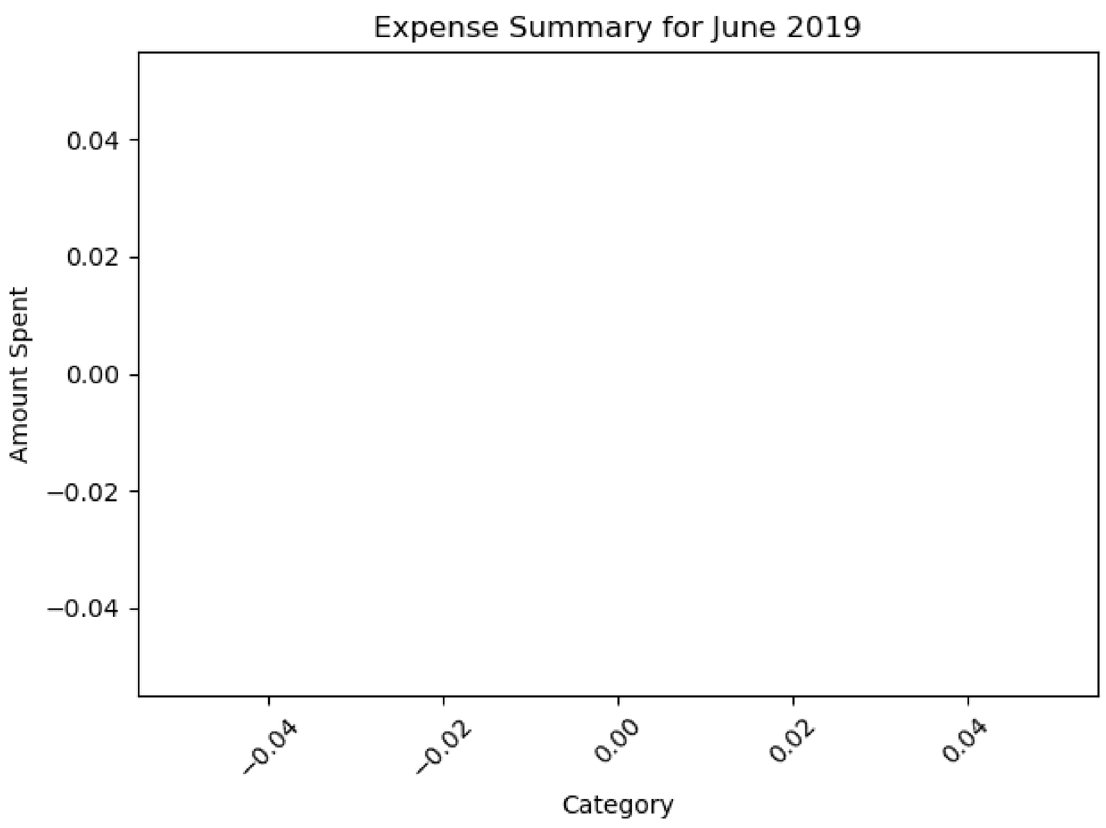
[19]:
```
import pytest
from my_classes import Expense
from my_classes import Tracker
import test_functions
```

```
/home/e2liang/cogs18 final project/my_classes.py:86: FutureWarning: The behavior
of DataFrame concatenation with empty or all-NA entries is deprecated. In a
future version, this will no longer exclude empty or all-NA columns when
determining the result dtypes. To retain the old behavior, exclude the relevant
```

```
entries before the concat operation.
  self.expenses = pd.concat([self.expenses, new_expense], ignore_index=True)
```



```
No data found for June 2019.
```



```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[19], line 4
      2 from my_classes import Expense
      3 from my_classes import Tracker
----> 4 import test_functions

File ~/cogs18 final project/test_functions.py:157
    155 test_t.test_get_expense_by_category()
    156 test_t.setup_method()
--> 157 test_t.test_get_monthly_summary()
    158 test_t.setup_method()
    159 test_t.test_calc_inflation_adj_expenses()

File ~/cogs18 final project/test_functions.py:116, in TestTracker.
 ↪test_get_monthly_summary(self)
    114 try:
    115     self.tracker.get_monthly_summary('June', 2019)
--> 116     assert False, "ValueError was not raised for missing data."
    117 except ValueError as e:
```

```
    118     assert str(e) == "No data found for June 2019.", "Unexpected error␣
↪message."

AssertionError: ValueError was not raised for missing data.
```

[20]:
```
!pytest my_classes.py
```

```
============================= test session starts
==============================
platform linux -- Python 3.11.9, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/e2liang/cogs18 final project
plugins: anyio-4.3.0
collected 0 items


============================ no tests ran in 1.24s
=============================
```

[21]:
```
!pytest test_functions.py
```

```
============================= test session starts
==============================
platform linux -- Python 3.11.9, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/e2liang/cogs18 final project
plugins: anyio-4.3.0
collected 0 items / 1 error


==================================== ERRORS ====================================
______________________ ERROR collecting test_functions.py
______________________
test_functions.py:157: in <module>
    test_t.test_get_monthly_summary()
test_functions.py:116: in test_get_monthly_summary
    assert False,
"ValueError was not raised for missing

data."
E   AssertionError: ValueError was not raised for missing data.
E   assert False
-------------------------------- Captured stdout ---------------------------------
Figure(640x480)
No data found for June 2019.
Figure(640x480)
=============================== warnings summary
===============================
my_classes.py:86
  /home/e2liang/cogs18 final project/my_classes.py:86: FutureWarning: The
behavior of DataFrame concatenation with empty or all-NA entries is deprecated.
In a future version, this will no longer exclude empty or all-NA columns when
determining the result dtypes. To retain the old behavior, exclude the relevant
entries before the concat operation.
    self.expenses = pd.concat([self.expenses, new_expense], ignore_index=True)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
=========================== short test summary info
============================
ERROR test_functions.py - AssertionError: ValueError was not raised for
missing data.
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
========================= 1 warning, 1 error in
1.77s ==========================
```

1.2.7 Extra Credit Explanation

Making this budget expense tracker was definitely the most challenging project I've undertaken this quarter because it forced me to step out of my comfort zone in coding to learn new modules and other educational topics like economics. I learned how to use Matplot and how to create different forms of data visualizations with that particular module. Something we weren't taught in this course is how we can utilize different modules together, and I felt this project was a learning experience as it showed me that Pandas can be useful in holding data beyond the collection types we learned in class, which provided the data for Matplot to access and create all the bar graphs for this project.

Additionally, one of the biggest challenges I faced in this project was creating unit tests for my classes. Although we gained exposure throughout the course on how asserts work and what other code tests can be made, I felt that creating code tests were really open-ended and struggled in deciding how to create each code test within the test class. I didn't really know what to test for the methods within my Tracker class, especially when it came to creating bar graphs of my data. I found that when re-examining my code within my methods, it helped to check the code that used if statements and boolean comparisons and use those as a basis for the methods within my test classes.