Revoulut bank project:

1. Overview of the project:
    - Creating a money management tool visualization with Python via using Jupyter Notebook.
    - Data source: created a two-month bank transaction dataset from scratch.
    - Jupyter notebook is the tool to carry out the whole project.
    - Python is the programming language applied.

2. Jupyter notebook steps:
    - Create a new project folder and create a new python file.
    - Upload the CSV/excel file of the revolut or any bank account statement you have to this folder. This will enable the notebook to read the file.
    - In my case, a dataset was manually created.
    - Import the necessary libraries like NumPy, pandas ,gspread , panel, hvplot and holoviews.

3. Jupyter notebook code explanation:

```
#required for manipulating data
import pandas as pd
import numpy as np

#enable Google Drive API
import gspread

#required for building the interactive dashboard
import panel as pn
pn.extension('tabulator')
import hvplot.pandas
import holoviews as hv
hv.extension('bokeh')
```

- panel: A high-level app and dashboarding solution for Python. It allows you to create interactive web applications with minimal code.
- hvplot.pandas: A plotting library for Pandas DataFrames that is built on top of HoloViews. It provides a high-level interface for creating interactive visualizations.

- holoviews: A library for building flexible, high-level visualizations that can be easily composed. It works well with other libraries like Bokeh and Matplotlib.
- hv.extension('bokeh'): Enables the Bokeh plotting backend for HoloViews. Bokeh is a visualization library that creates interactive plots for the web.
- In summary, the code imports Pandas and NumPy for data manipulation, gspread for Google Sheets integration, and Panel, HoloViews, and hvPlot for building interactive dashboards with visualizations. The extensions and backends are configured to work with specific features and plotting libraries.

```
# create dataset manually

data = [
    ['CARD_PAYMENT', 'Current', '2023-01-26 22:02:47', '2023-01-27 10:10:12', 'Tesco Stores 6601', -6.35, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '2023-01-26 16:13:50', '2023-01-27 11:50:32', 'Zettle_*donovan?s Bake', -2.5, 0, 'GB
    ['CARD_PAYMENT', 'Current', '2023-01-26 08:26:35', '2023-01-27 13:42:55', 'apple.com/bill', -3.99, 0, 'GBP', 'CO
    ['CARD_PAYMENT', 'Current', '2023-01-27 13:16:59', '2023-01-28 09:59:20', 'Tesco Stores 6601', -4.85, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '2023-01-27 10:43:22', '2023-01-28 11:48:36', 'Zettle_*the Good Eatin', -3.3, 0, 'GB
    ['CARD_PAYMENT', 'Current', '2023-01-27 19:07:07', '2023-01-28 15:51:42', 'Toogoodtog Mzs3m03xcn9', -5, 0, 'GBP'
    ['CARD_PAYMENT', 'Current', '2023-01-28 01:14:16', '2023-01-29 08:55:25', 'Montys Bar', -21, 0, 'GBP', 'COMPLETE
    ['CARD_PAYMENT', 'Current', '2023-01-28 16:26:00', '2023-01-29 09:41:04', 'Amznmktplace', -8.99, 0, 'GBP', 'COMP
    ['CARD_PAYMENT', 'Current', '2023-01-28 19:59:36', '2023-01-29 09:49:29', 'Tesco Stores 6601', -8.65, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '2023-01-27 22:59:58', '2023-01-29 10:08:56', 'Urban 40', -12.6, 0, 'GBP', 'COMPLETE
    ['CARD_PAYMENT', 'Current', '2023-01-27 22:06:03', '2023-01-29 10:08:56', 'Urban 40', -6.1, 0, 'GBP', 'COMPLETED
    ['CARD_PAYMENT', 'Current', '2023-01-29 02:39:46', '2023-01-29 10:55:55', 'Tfl Travel Charge', -4.25, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '2023-01-28 15:06:44', '2023-01-29 11:31:51', 'Sq *lavelle Coffee Limite', -3.2, 0,
    ['CARD_PAYMENT', 'Current', '2023-01-29 15:08:04', '2023-01-31 14:06:07', 'Tubebuddycom*', -2.43, 0.02, 'GBP', '
    ['CARD_PAYMENT', 'Current', '2023-01-29 15:11:03', '2023-01-31 14:45:55', 'Sp Blomma Beauty', -35, 0, 'GBP', 'CO
    ['CARD_PAYMENT', 'Current', '2023-01-29 20:03:55', '2023-01-31 15:11:28', 'Uber* Trip', -13.46, 0, 'GBP', 'COMPL
    ['CARD_PAYMENT', 'Current', '2023-01-29 14:31:09', '2023-01-31 17:29:48', 'Caravan', -27.67, 0, 'GBP', 'COMPLETE
    ['CARD_PAYMENT', 'Current', '2023-01-30 01:57:58', '2023-01-31 17:41:29', 'Tfl Travel Charge', -4.25, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '2023-01-31 02:36:53', '2023-01-31 18:03:14', 'Tfl Travel Charge', -1.65, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '2023-01-29 19:13:59', '2023-01-31 18:46:41', 'Hart Shoreditch Hotel', -3.6, 0, 'GBP
    ['CARD_PAYMENT', 'Current', '2023-01-30 19:03:58', '2023-01-31 19:00:20', 'Tesco Stores 6601', -4.25, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '2023-01-30 13:41:48', '2023-01-31 19:32:36', 'Lidl Gb London', -8.12, 0, 'GBP', 'CO
    ['CARD_PAYMENT', 'Current', '2023-01-30 11:55:40', '2023-01-31 19:55:25', 'E5 Bakehouse Canning T', -12.36, 0, '
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Amznmktplace', -17.07, 0, 'GBP', 'COM
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Starbucks', -3.95, 0, 'GBP', 'COMPLET
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Tfl Travel Charge', -2.6, 0, 'GBP', '
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Millennium Mini Store', -5, 0, 'GBP',
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'cm.com', -253, 0, 'GBP', 'COMPLETED',
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Katsute100', -12.25, 0, 'GBP', 'COMPL
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Ubr* Pending.uber.com', -13.66, 0, 'G
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Ubr* Pending.uber.com', -7.03, 0, 'GB
    ['CARD_PAYMENT', 'Current', '30-02-2023 11:55:40', '02-02-2023 11:55:40', 'Starbucks', -3.9, 0, 'GBP', 'COMPLETE
]

columns = ['Type', 'Product', 'Started Date', 'Completed Date', 'Description', 'Amount', 'Fee', 'Currency', 'State',

df = pd.DataFrame(data=data, columns=columns)
```

- Data set manually created.

```
#clean df

df = df[['Completed Date', 'Description', 'Amount']] #keep only desired columns
df['Description'] = df['Description'].map(str.lower) #lower case of descriptions

df = df.rename(columns={'Completed Date': 'Date'})   #rename columns
df['Category'] = 'unassigned'                        #add category column

df.head()
```

	Date	Description	Amount	Category
0	2023-01-27 10:10:12	tesco stores 6601	-6.35	unassigned
1	2023-01-27 11:50:32	zettle_*donovan?s bake	-2.50	unassigned
2	2023-01-27 13:42:55	apple.com/bill	-3.99	unassigned
3	2023-01-28 09:59:20	tesco stores 6601	-4.85	unassigned
4	2023-01-28 11:48:36	zettle_*the good eatin	-3.30	unassigned

- In summary, this code snippet cleans and restructures the DataFrame df by selecting specific columns, converting text in the

'Description' column to lowercase, renaming columns, adding a new 'Category' column, and displaying the modified DataFrame.

```
#Assign transactions to the correct category

# Self-Care

df['Category'] = np.where(df['Description'].str.contains(
    'cash at tesco old st h exp|boots|royal'),
    'Self-Care', df['Category'] )

# Fines

df['Category'] = np.where(df['Description'].str.contains(
    'car rental'),
    'Fines', df['Category'] )

# Lore So What

df['Category'] = np.where(df['Description'].str.contains(
    'tubebuddy|itunes|dario|calendly|canva|epidemic|upwork|lada'),
    'LoreSoWhat', df['Category'] )

# Coffee

df['Category'] = np.where(df['Description'].str.contains(
    'lavelle|hart|starbucks|barista|new road|mama shelter'),
    'Coffee', df['Category'] )

# Shopping

df['Category'] = np.where(df['Description'].str.contains(
    'islington|at camden town'),
    'Shopping', df['Category'] )

# Restaurants

df['Category'] = np.where(df['Description'].str.contains(
    'bakehouse|zettle|caravan|kod|eating|o ver|mcdonald|manteca|wine ho
    'Restaurants', df['Category'] )
```

- In each block, the np.where function checks if the 'Description' column contains specific keywords or patterns. If true, it assigns the corresponding category to the 'Category' column; otherwise, it leaves the 'Category' column unchanged (df['Category']). This approach allows for conditional categorization of transactions based on the content of the description.

```
# Convert the "Date" column to a datetime format
df['Date'] = pd.to_datetime(df['Date'])

# Extract the month and year information
df['Month'] = df['Date'].dt.month
df['Year'] = df['Date'].dt.year

pd.options.display.max_rows = 999
df.head(200)
```

	Date	Description	Amount	Category	Month	Year
0	2023-01-27 10:10:12	tesco stores 6601	-6.35	Groceries	1	2023
1	2023-01-27 11:50:32	zettle_*donovan?s bake	-2.50	Restaurants	1	2023
2	2023-01-27 13:42:55	apple.com/bill	-3.99	Services	1	2023
3	2023-01-28 09:59:20	tesco stores 6601	-4.85	Groceries	1	2023
4	nd output; double click to hide output	tle_*the good eatin	-3.30	Restaurants	1	2023
5	2023-01-28 15:51:42	toogoodtog mzs3m03xcn9	-5.00	Groceries	1	2023
6	2023-01-29 08:55:25	montys bar	-21.00	Entertainment	1	2023
7	2023-01-29 09:41:04	amznmktplace	-8.99	Excluded	1	2023
8	2023-01-29 09:49:29	tesco stores 6601	-8.65	Groceries	1	2023
9	2023-01-29 10:08:56	urban 40	-12.60	Entertainment	1	2023
10	2023-01-29 10:08:56	urban 40	-6.10	Entertainment	1	2023
11	2023-01-29 10:55:55	tfl travel charge	-4.25	Transport	1	2023
12	2023-01-29 11:31:51	sq *lavelle coffee limite	-3.20	Coffee	1	2023
13	2023-01-31 14:06:07	tubebuddycom*	-2.43	LoreSoWhat	1	2023
14	2023-01-31 14:45:55	sp blomma beauty	-35.00	Gifts	1	2023

- In summary, this code is preparing the DataFrame for analysis by converting the 'Date' column to a datetime format and extracting the month and year information into separate columns. The adjusted display options ensure that up to 999 rows are shown when printing the DataFrame. The head(200) function displays the first 200 rows of the DataFrame.

In [7]:
```
#check unassigned transactions and confirm all transactions are assigned to a category

unassigned = df.loc[df['Category'] == 'unassigned']
unassigned
```

Out[7]:

Date	Description	Amount	Category	Month	Year

- df['Category'] == 'unassigned' creates a boolean mask that is True for rows where the 'Category' column is equal to the string 'unassigned' and False otherwise.
- df.loc[...] is used for label-based indexing. It selects rows based on the condition specified in the boolean mask.
- Therefore, unassigned is a new DataFrame containing only the rows from the original DataFrame (df) where the 'Category' column has the value 'unassigned'.

```
# Get the latest month and year
latest_month = df['Month'].max()
latest_year = df['Year'].max()

# Filter the dataframe to include only transactions from the latest month
last_month_expenses = df[(df['Month'] == latest_month) & (df['Year'] == latest_year)]
```

- calculating the maximum values for the 'Month' and 'Year' columns in the DataFrame df.
- using boolean indexing to filter the DataFrame df. The condition inside the square brackets creates a boolean mask that is True for rows where both the 'Month' column is equal to latest_month and the 'Year' column is equal to latest_year. The & ensures that both conditions are satisfied.
- last_month_expenses will contain only the rows from the original DataFrame (df) where the month matches the latest month and the year matches the latest year.

```
last_month_expenses = last_month_expenses.groupby('Category')['Amount'].sum().reset_index()

last_month_expenses['Amount']=last_month_expenses['Amount'].astype('str')
last_month_expenses['Amount']=last_month_expenses['Amount'].str.replace('-','')
last_month_expenses['Amount']=last_month_expenses['Amount'].astype('float')        #get absolute figures

last_month_expenses = last_month_expenses[last_month_expenses["Category"].str.contains("Excluded|unassigned") == Fal
last_month_expenses = last_month_expenses.sort_values(by='Amount', ascending=False)    #sort values
last_month_expenses['Amount'] = last_month_expenses['Amount'].round().astype(int)      #round values

last_month_expenses
```

	Category	Amount
4	Transport	23
3	Restaurants	12
0	Coffee	8
2	Groceries	5

- last_month_expenses DataFrame by the 'Category' column and calculates the sum of the 'Amount' within each group. The result is a new DataFrame with unique categories and their corresponding total amounts.
- the 'Amount' column is temporarily converted to a string, any '-' characters are removed (assuming the 'Amount' column contains both positive and negative values), and then the column is converted back to float, effectively obtaining the absolute values.
- The 5th line removes rows where the 'Category' column contains the strings "Excluded" or "unassigned".
- In the 6th line, The DataFrame is sorted based on the 'Amount' column in descending order.
- The 7th line, rounds the 'Amount' values to the nearest integer and converts the column to integer type.

```
last_month_expenses_tot = last_month_expenses['Amount'].sum()
last_month_expenses_tot
```

48

- The result of last_month_expenses['Amount'].sum() is the total sum of all the values in the 'Amount' column of the last_month_expenses DataFrame.
- This sum is then assigned to the variable last_month_expenses_tot. After this line of code is executed, last_month_expenses_tot will hold the total sum of expenses for the last month.

```
def calculate_difference(event):
    income = float(income_widget.value)
    recurring_expenses = float(recurring_expenses_widget.value)
    monthly_expenses = float(monthly_expenses_widget.value)
    difference = income - recurring_expenses - monthly_expenses
    difference_widget.value = str(difference)

income_widget = pn.widgets.TextInput(name="Income", value="0")
recurring_expenses_widget = pn.widgets.TextInput(name="Recurring Expenses", value="0")
monthly_expenses_widget = pn.widgets.TextInput(name="Non-Recurring Expenses", value=str(last_month_expenses_tot))
difference_widget = pn.widgets.TextInput(name="Last Month's Savings", value="0")

income_widget.param.watch(calculate_difference, "value")
recurring_expenses_widget.param.watch(calculate_difference, "value")
monthly_expenses_widget.param.watch(calculate_difference, "value")

pn.Row(income_widget, recurring_expenses_widget, monthly_expenses_widget, difference_widget).show()
```

```
Launching server at http://localhost:50987
```

- The function calculate_difference takes an event as an argument. It is designed to be called when there is a change in the value of certain widgets. It retrieves the current values from income_widget, recurring_expenses_widget, and monthly_expenses_widget. It calculates the difference by subtracting recurring expenses and monthly expenses from income. It updates the value of difference_widget with the calculated difference.
- Four TextInput widgets are created for capturing user input related to income, recurring expenses, monthly expenses, and last month's savings. The name parameter sets the label for each widget, and the value parameter sets the default value.
- The third lines use the param.watch method to watch for changes in the 'value' attribute of the three input widgets (income_widget, recurring_expenses_widget, and monthly_expenses_widget). When the value of any of these widgets changes, the calculate_difference function is triggered to update the 'value' of the difference_widget.
- The last line creates a Panel Row layout with the four widgets arranged horizontally. The show() method is called to display the dashboard.
- In summary, this code creates a simple interactive financial dashboard where users can input their income, recurring expenses, and monthly expenses. The difference between income and expenses is dynamically calculated and displayed in real-time as "Last Month's Savings."

```
last_month_expenses_chart = last_month_expenses.hvplot.bar(
    x='Category',
    y='Amount',
    height=250,
    width=850,
    title="Last Month Expenses",
    ylim=(0, 500))

last_month_expenses_chart
```

- last_month_expenses.hvplot.bar: This line uses the hvplot method on the last_month_expenses DataFrame to create a bar chart.
- x='Category': This specifies that the 'Category' column should be used as the x-axis values.
- y='Amount': This specifies that the 'Amount' column should be used as the y-axis values.
- height=250: This sets the height of the plot to 250 pixels.
- width=850: This sets the width of the plot to 850 pixels.
- title="Last Month Expenses": This sets the title of the plot to "Last Month Expenses".
- ylim=(0, 500): This sets the y-axis limits to be between 0 and 500.

```
df['Date'] = pd.to_datetime(df['Date'])            # convert the 'Date' column to a datetime object
df['Month-Year'] = df['Date'].dt.to_period('M')    # extract the month and year from the 'Date' column and create a
monthly_expenses_trend_by_cat = df.groupby(['Month-Year', 'Category'])['Amount'].sum().reset_index()

monthly_expenses_trend_by_cat['Amount']=monthly_expenses_trend_by_cat['Amount'].astype('str')
monthly_expenses_trend_by_cat['Amount']=monthly_expenses_trend_by_cat['Amount'].str.replace('-','')
monthly_expenses_trend_by_cat['Amount']=monthly_expenses_trend_by_cat['Amount'].astype('float')
monthly_expenses_trend_by_cat = monthly_expenses_trend_by_cat[monthly_expenses_trend_by_cat["Category"].str.contains

monthly_expenses_trend_by_cat = monthly_expenses_trend_by_cat.sort_values(by='Amount', ascending=False)
monthly_expenses_trend_by_cat['Amount'] = monthly_expenses_trend_by_cat['Amount'].round().astype(int)
monthly_expenses_trend_by_cat['Month-Year'] = monthly_expenses_trend_by_cat['Month-Year'].astype(str)
monthly_expenses_trend_by_cat = monthly_expenses_trend_by_cat.rename(columns={'Amount': 'Amount '})

monthly_expenses_trend_by_cat
```

- The first line converts the 'Date' column in the DataFrame df to a DateTime object, making it easier to work with date-related operations.
- The second line extracts the month and year from the 'Date' column and creates a new column 'Month-Year' representing the combined month and year as a Period.
- The third line groups the DataFrame by 'Month-Year' and 'Category' and calculates the sum of the 'Amount' within each group. The result is a new DataFrame (monthly_expenses_trend_by_cat) with unique combinations of month, category, and the corresponding total expenses.
- The 2nd block of code lines converts the 'Amount' column to string, removes any '-' characters, and then converts it back to a float, effectively getting absolute values.
- The 3rd block of code lines performs additional data cleaning and transformation: sorting the DataFrame by 'Amount' in descending order, rounding 'Amount' values, converting 'Month-Year' to string, and renaming the 'Amount' column to 'Amount '.

- In summary, the code performs data cleaning and transformation on financial data to create a DataFrame showing monthly expenses aggregated by category. The resulting DataFrame is then set up for display.

```
#Define Panel widget

select_category1 = pn.widgets.Select(name='Select Category', options=[
    'All',
    'Self-Care',
    'Fines',
    'LoreSoWhat',
    'Coffee',
    'Groceries',
    'Shopping',
    'Restaurants',
    'Transport',
    'Travel',
    'Entertainment',
    'Gifts',
    'Services',
    #'Excluded'
])

select_category1
```

- pn.widgets.Select:
  - This is a part of the Panel library and is used to create a dropdown (select) widget.
- name='Select Category':
  - This sets the label or name for the dropdown widget. The label is displayed next to the dropdown.
- options:
  - This parameter specifies the available options in the dropdown. In this case, the options include categories such as 'All', 'Self-Care', 'Fines', 'Coffee', 'Groceries', and so on.
- The resulting select_category1 variable is a Panel widget representing a dropdown menu. Users can select one option from the menu, and the selected value can be used in your application logic. This widget is used for creating interactive dashboards or applications where users can dynamically choose a category to display or analyze specific data.

```
# define plot function
def plot_expenses(category):
    if category == 'All':
        plot_df = monthly_expenses_trend_by_cat.groupby('Month-Year').sum()
    else:
        plot_df = monthly_expenses_trend_by_cat[monthly_expenses_trend_by_cat['Category'] == category].groupby('Mon
    plot = plot_df.hvplot.bar(x='Month-Year', y='Amount ')
    return plot
```

- Function Signature:
  - plot_expenses(category), This function takes a single parameter, category, which represents the category for which the expenses will be plotted.
- Plotting Logic:

    - The function first checks if the selected category is 'All'. If the category is 'All', it groups the monthly_expenses_trend_by_cat DataFrame by 'Month-Year' and calculates the sum of expenses for each month. If the category is not 'All', it filters the DataFrame to include only rows with the specified category and then groups by 'Month-Year' and calculates the sum of expenses for each month.
- Plot Generation:
    - plot_df.hvplot.bar(x='Month-Year', y='Amount '), This line uses the hvplot library to create a bar chart. x='Month-Year': Specifies the 'Month-Year' column as the x-axis values. y='Amount ': Specifies the 'Amount ' column as the y-axis values. The space in 'Amount ' is intentional, and it refers to the renamed column from the monthly_expenses_trend_by_cat DataFrame.
- Return Statement:
    - return plot, The function returns the generated plot.
    - This function can be used in a Panel application or another interactive environment where users can select a category from a dropdown menu (perhaps created using the select_category1 widget) to dynamically update and display the corresponding expenses over time in a bar chart.

```
# define callback function
@pn.depends(select_category1.param.value)
def update_plot(category):
    plot = plot_expenses(category)
    return plot
```

- @pn.depends Decorator:
    - This decorator is part of the Panel library (pn).
    - It is used to declare a dependency relationship between the callback function and the specified parameters.
    - In this case, it specifies that the update_plot function depends on the value of the select_category1 widget (select_category1.param.value).
- Function Signature:
    - update_plot(category): This function takes a single parameter, category, which represents the selected category from the select_category1 widget.
- Function Logic:
    - plot = plot_expenses(category): This line calls the plot_expenses function with the selected category as an argument. It generates a bar chart based on the selected category.
- Return Statement:
    - return plot: The function returns the generated plot.

```
# create layout
monthly_expenses_trend_by_cat_chart = pn.Row(select_category1, update_plot)
monthly_expenses_trend_by_cat_chart[1].width = 600

monthly_expenses_trend_by_cat_chart
```

- pn.Row Function:

    - pn.Row(select_category1, update_plot): This creates a horizontal layout (Row) with two elements.
    - The first element is the select_category1 widget, which is a dropdown for selecting expense categories.
    - The second element is the update_plot function, which generates a plot based on the selected category.
- Adjust Width of the Second Element:
    - monthly_expenses_trend_by_cat_chart[1].width = 600: This line adjusts the width of the second element (the plot generated by the update_plot function) to 600 pixels. It modifies the width attribute of the second element.
- Resulting Layout Variable:
    - monthly_expenses_trend_by_cat_chart: This variable holds the resulting layout, which consists of a row with a dropdown widget and a plot. The width of the plot is set to 600 pixels.
- In summary, this code creates a layout where users can select an expense category from a dropdown (select_category1), and a corresponding plot is displayed next to it. The width of the plot is set to 600 pixels. This layout is used in a Panel application or another interactive environment where users can dynamically explore monthly expenses by category.

```
df = df[['Date', 'Category', 'Description', 'Amount']]
df['Amount']=df['Amount'].astype('str')
df['Amount']=df['Amount'].str.replace('-','')
df['Amount']=df['Amount'].astype('float')        #get absolute figures

df = df[df["Category"].str.contains("Excluded") == False]    #exclude "excluded" category
df['Amount'] = df['Amount'].round().astype(int)      #round values
df
```

	Date	Category	Description	Amount
0	2023-01-27 10:10:12	Groceries	tesco stores 6601	6
1	2023-01-27 11:50:32	Restaurants	zettle_*donovan?s bake	2
2	2023-01-27 13:42:55	Services	apple.com/bill	4
3	2023-01-28 09:59:20	Groceries	tesco stores 6601	5
4	2023-01-28 11:48:36	Restaurants	zettle_*the good eatin	3

- The first line selects and keeps only the columns 'Date', 'Category', 'Description', and 'Amount' in the DataFrame df.
- df['Amount'].astype('str'): Converts the 'Amount' column to a string type.
- df['Amount'].str.replace('-',''): Removes any '-' characters from the 'Amount' column, assuming it represents both positive and negative values.
- df['Amount'].astype('float'): Converts the 'Amount' column back to a float, obtaining the absolute values.
- The 4th line filters out rows where the 'Category' column contains the string "Excluded". Rows with this category are excluded from the DataFrame.
- The 5th line rounds the 'Amount' values to the nearest integer and converts the 'Amount' column to integer type.

- The resulting DataFrame df is a cleaned and processed version of the original financial data. It includes only the selected columns, the 'Amount' values are converted to absolute values, rows with the category "Excluded" are excluded, and the 'Amount' values are rounded to integers.

```
# Define a function to filter the dataframe based on the selected category
def filter_df(category):
    if category == 'All':
        return df
    return df[df['Category'] == category]
```

- Function Signature:
  - filter_df(category): This function takes a single parameter, category, which represents the category used for filtering.
- Filtering Logic:
  - if category == 'All':: This checks if the selected category is 'All'.
  - If the category is 'All', the function returns the entire DataFrame df without any filtering.
- return df[df['Category'] == category]:
  - If the category is not 'All', the function returns a filtered DataFrame that includes only rows where the 'Category' column is equal to the specified category.
- Return Statement:
  - The function returns the filtered DataFrame.
- In summary, the purpose of this function is to provide a way to filter the DataFrame df based on the selected category. If the category is 'All', the entire DataFrame is returned. Otherwise, only rows with the specified category are included in the returned DataFrame. This function can be useful when you want to dynamically explore or analyze data for a specific category or the entire dataset.

```
# Create a DataFrame widget that updates based on the category filter
summary_table = pn.widgets.DataFrame(filter_df('All'), height = 300,width=400)
```

- DataFrame Widget:
  - pn.widgets.DataFrame: This creates a DataFrame widget using the Panel library.
- Initial Data for the DataFrame:
  - filter_df('All'): This immediately applies the filter_df function with the argument 'All' to get the initial data for the DataFrame widget. As a result, the DataFrame widget initially displays the entire DataFrame df because the category is set to 'All'.
- Widget Size Parameters:
  - height=300, width=400: These parameters set the height and width of the DataFrame widget to 300 pixels and 400 pixels, respectively.
- In summary, the summary_table DataFrame widget is created and populated with the initial data based on the category filter ('All'). This widget can be used in a Panel application or interactive environment to dynamically display and explore a summary of the data based on the selected category. If the category filter changes, the widget would update to display the corresponding filtered data.

```
# Define a callback that updates the dataframe widget when the category filter is changed
def update_summary_table(event):
    summary_table.value = filter_df(event.new)
```

- Function Signature:
  - update_summary_table(event): This function takes an event as a parameter. The assumption here is that the event contains information about the change that triggered the callback.
- Update Logic:
  - filter_df(event.new): This calls the filter_df function with the new value of the event. The assumption is that the event has a property named new that contains the updated category.
- summary_table.value = ...: This updates the value attribute of the summary_table DataFrame widget. The value attribute typically represents the data displayed by the widget.
- Callback Triggering:
  - The update_summary_table function is likely intended to be used as a callback associated with an event. For example, it might be connected to the select_category1 widget so that when the user changes the selected category, this function is triggered.
- In summary, this callback function updates the data displayed in the summary_table DataFrame widget based on the new category selected. If the category filter changes (as indicated by the event), the DataFrame widget is updated to display the corresponding filtered data.

```
# Add the callback function to the category widget
select_category1.param.watch(update_summary_table, 'value')

summary_table
```

- param.watch Method:
  - select_category1.param.watch: This method from the Panel library is used to watch for changes in a specific parameter of a widget.
- 'value': This specifies the parameter to watch. Here, it's the value attribute of the select_category1 widget.
- In summary, this code establishes a connection between the select_category1 widget and the summary_table DataFrame widget. When the user changes the selected category in the dropdown, the update_summary_table callback is triggered, updating the data displayed in the DataFrame widget accordingly.

```
template = pn.template.FastListTemplate(
    title="Personal Finances Summary",
    sidebar=[
        pn.pane.Markdown("## *Money is a tool, not a goal. Use it wisely to craft a life of purpose and abundance.
        pn.pane.PNG('Pound | currency | uk | United Kingdom.png', sizing_mode='scale_both'),
        pn.pane.Markdown(""),
        pn.pane.Markdown(""),
        select_category1
    ],
    main=[
        pn.Row(income_widget, recurring_expenses_widget, monthly_expenses_widget, difference_widget, width=950),
        pn.Row(last_month_expenses_chart, height=240),
        pn.GridBox(
            monthly_expenses_trend_by_cat_chart[1],
            summary_table,
            ncols=2,
            width=500,
            align='start',
            sizing_mode='stretch_width'
        )
    ]
)

template.show()
```

```
Launching server at http://localhost:51298
```

- Template Creation:
  - template = pn.template.FastListTemplate(...): This line creates an instance of the FastListTemplate class, which is a specific template style provided by Panel.
- Template Configuration:
  - title="Personal Finances Summary": Sets the title of the template to "Personal Finances Summary."
- Sidebar Configuration:
  - sidebar=[...]: Configures the sidebar of the template, which contains various components.
- pn.pane.Markdown(...): Displays a Markdown text with information about the purpose of the financial summary.
- pn.pane.PNG(...): Displays an image (presumably a currency-related image) using a PNG file.
- pn.pane.Markdown(""): Inserts an empty Markdown pane for spacing.
- pn.pane.Markdown(""): Another empty Markdown pane for additional spacing.
- select_category1: Displays the select_category1 dropdown widget, allowing the user to select a category for data filtering.
- Main Content Configuration:
  - main=[...]: Configures the main content of the template.
  - pn.Row(...): Displays a row of widgets, including income, recurring expenses, monthly expenses, and a difference widget.
  - pn.Row(...): Displays a row containing the last_month_expenses_chart.
  - pn.GridBox(...): Displays a grid containing two components: the second element of monthly_expenses_trend_by_cat_chart (presumably a plot) and the summary_table DataFrame widget.
- Width and Height Configuration:
  - Various width and height parameters are set to control the dimensions of different components.
- template.show():
  - Finally, the template.show() method is called to display the constructed template.

- In summary, this code sets up a comprehensive Panel template for presenting and interacting with personal finance summary data. The template includes a sidebar with text, an image, and a category selection dropdown. The main content includes rows of financial widgets, charts, and a grid layout containing additional visualizations and a DataFrame widget. The template is then displayed using template.show().

4. Dashboard key points:
    - Recurring expenses are something you must pay every month, week, year etc. like the rent, bills, grocery and so on.
    - Income is the salary/money got from work or any business/asset one may have.
    - Non-recurring expenses are automatically calculated in our bank statement.
    - Last month's savings is the total amount saved each month.
    - Last month's expenses summarise the expenses of the last month, where the x-axis represents the category in which the money is spent, and the amount is the y-axis which shows the total money spent on various categories.
    - the select category filter on the sidebar filters the category you want to search.
    - The second bar chart in the middle shows the month and year on the x-axis and the amount on the y-axis. This bar chart shows the spending done in year and month.
    - The table on the lower right-hand side shows all the details of the selected categories or the details inserted in the income and recurring expenses boxes.