Integrated Project: Understanding Maji Ndogo's agriculture

In this coding challenge, we will apply all of the skills we learned in Pandas.

⚠️ **Note that this code challenge is graded and will contribute to your overall marks for this module. Submit this notebook for grading. Note that the names of the functions are different in this notebook. Transfer the code in your notebook to this submission notebook**

Instructions

- **Do not add or remove cells in this notebook. Do not edit or remove the `### START FUNCTION` or `### END FUNCTION` comments. Do not add any code outside of the functions you are required to edit. Doing any of this will lead to a mark of 0%!**
- Answer the questions according to the specifications provided.
- Use the given cell in each question to see if your function matches the expected outputs.
- Do not hard-code answers to the questions.
- The use of StackOverflow, Google, and other online tools is permitted. However, copying a fellow student's code is not permissible and is considered a breach of the Honour code. Doing this will result in a mark of 0%.

Introduction

Hey there, I'm glad you're on board for the Maji Ndogo project AGAIN! Let me walk you through what we're up against and how we'll tackle it.

As you know, we're in an ambitious project aimed at automating farming in Maji Ndogo, a place with diverse and challenging agricultural landscapes. Before we dive into the 'how' of farming, we need to figure out the 'where' and 'what'. It's not just about deploying technology; it's about making informed decisions on where to plant specific crops, considering factors like rainfall, soil type, climate, and many others.

Our analysis is the groundwork for this entire automation project. We have an array of variables like soil fertility, climate conditions, and geographical data. By understanding these elements, we can recommend the best locations for different crops. It's a bit like solving a complex puzzle – each piece of data is crucial to seeing the bigger picture.

We'll start by importing our dataset into a DataFrame. It is currently in an SQLite database, and split into tables. Unlike Power BI or SQL, data analysis in Python happens in a single table. So we will have to brush off those dusty SQL skills to get the data imported. Expect a bit of a mess in the data – it's part of the challenge. We need to clean it up and maybe reshape it to make sense of it. It's like sorting out the tools and materials we need and getting rid of what we don't.

Here's where the real fun begins. We'll dive deep into the data, looking for patterns, and correlations. Each clue in the data leads us closer to understanding the best farming practices for Maji Ndogo. I'll be relying on your skills and insights. We'll be working through these steps together, discussing our findings and strategies.

Let's gear up and get ready to make a real difference in Maji Ndogo. Ready to get started? Let's dive into our data and see what stories it has to tell us.

Sanaa.

Data dictionary

1. Geographic features

- **Field_ID:** A unique identifier for each field (BigInt).
- **Elevation:** The elevation of the field above sea level in metres (Float).
- **Latitude:** Geographical latitude of the field in degrees (Float).
- **Longitude:** Geographical longitude of the field in degrees (Float).
- **Location:** Province the field is in (Text).
- **Slope:** The slope of the land in the field (Float).

2. Weather features

- **Field_ID:** Corresponding field identifier (BigInt).
- **Rainfall:** Amount of rainfall in the area in mm (Float).
- **Min_temperature_C:** Average minimum temperature recorded in Celsius (Float).
- **Max_temperature_C:** Average maximum temperature recorded in Celsius (Float).

- **Ave_temps:** Average temperature in Celcius (Float).

3. Soil and crop features

- **Field_ID:** Corresponding field identifier (BigInt).
- **Soil_fertility:** A measure of soil fertility where 0 is infertile soil, and 1 is very fertile soil (Float).
- **Soil_type:** Type of soil present in the field (Text).
- **pH:** pH level of the soil, which is a measure of how acidic/basic the soil is (Float).

4. Farm management features

- **Field_ID:** Corresponding field identifier (BigInt).
- **Pollution_level:** Level of pollution in the area where 0 is unpolluted and 1 is very polluted (Float).
- **Plot_size:** Size of the plot in the field (Ha) (Float).
- **Chosen_crop:** Type of crop chosen for cultivation (Text).
- **Annual_yield:** Annual yield from the field (Float). This is the total output of the field. The field size and type of crop will affect the Annual Yield
- **Standard_yield:** Standardised yield expected from the field, normalised per crop (Float). This is independent of field size, or crop type. Multiplying this number by the field size, and average crop yield will give the Annual_Yield.

Average yield (tons/Ha) per crop type:

- **Coffee:** 1.5
- **Wheat:** 3
- **Rice:** 4.5
- **Maize:** 5.5
- **Tea:** 1.2
- **Potato:** 20
- **Banana:** 30
- **Cassava:** 13

Alright, let's walk through the process of importing our SQL data from multiple tables into a single DataFrame. This is a crucial step as it sets the foundation for all our subsequent analyses.

We're dealing with an SQLite database, `Maji_Ndogo_farm_survey.db` , which contains multiple tables. We'll need to join these tables on a common key to create a comprehensive dataset for our analysis. The common key in our case is `Field_ID` .

Here's how we can do it:

In [1]:
```
import pandas as pd # importing the Pandas package with an alias, pd
from sqlalchemy import create_engine, text # Importing the SQL interface. If this fails, run !pip install sqlc

# Create an engine for the database
engine = create_engine('sqlite:///Maji_Ndogo_farm_survey_small.db') #Make sure to have the .db file in the sam
```

Next up, we test if the connection works by printing out all of the table names in the database.

In [2]:
```
with engine.connect() as connection:
    result = connection.execute(text("SELECT name FROM sqlite_master WHERE type='table';"))
    for row in result:
        print(row)
```

```
('geographic_features',)
('weather_features',)
('soil_and_crop_features',)
('farm_management_features',)
```

Expected output:

`('geographic_features',)`

`('weather_features',)`

`('soil_and_crop_features',)`

`('farm_management_features',)`

At this point, we have two choices:

1. Either we import each table into a DataFrame, for example, `df_geographic` , then merge them together.
2. Use one SQL query and read it into a single DataFrame.

While both are equally viable, let's try to use a single SQL query to keep things simple.

Next, we'll write an SQL query to join our tables. Combine all of the tables into a single query, using `Field_ID` .

In [3]:
```
sql_query = """
SELECT *
FROM geographic_features
JOIN weather_features ON geographic_features.Field_ID = weather_features.Field_ID
JOIN soil_and_crop_features ON geographic_features.Field_ID = soil_and_crop_features.Field_ID
JOIN farm_management_features ON geographic_features.Field_ID = farm_management_features.Field_ID;

"""
```

With our engine and query ready, we'll use `Pandas` to execute the query. The `pd.read_sql_query` function fetches the data and loads it into a DataFrame – essentially transferring our data from the database into a familiar `Pandas` structure. If you use one query, you will import it all into `MD_agric_df` .

In [4]:
```
# Create a connection object
with engine.connect() as connection:

    # Use Pandas to execute the query and store the result in a DataFrame
    MD_agric_df = pd.read_sql_query(text(sql_query), connection)
```

Check the DataFrame to see if it loaded correctly.

In [5]:
```
MD_agric_df
```

Out[5]:

n_temperature_C	Max_temperature_C	...	Field_ID	Soil_fertility	Soil_type	pH	Field_ID	Pollution_level	Plot_size	Crop_type	Annu
-3.1	33.1	...	40734	0.62	Sandy	6.169393	40734	8.526684e-02	1.3	0.751354	
-3.9	30.6	...	30629	0.64	Volcanic	5.676648	30629	3.996838e-01	2.2	1.069865	
-1.8	28.4	...	39924	0.69	Volcanic	5.331993	39924	3.580286e-01	3.4	2.208801	
-5.8	32.2	...	5754	0.54	Loamy	5.328150	5754	2.866871e-01	2.4	1.277635	
-2.5	31.0	...	14146	0.72	Sandy	5.721234	14146	4.319027e-02	1.5	0.832614	
...	...	...	...	...	...	...	...	...	...	...	
-4.3	33.4	...	11472	0.61	Sandy	5.741063	11472	3.286828e-01	1.1	0.609930	
-4.8	32.1	...	19660	0.54	Sandy	5.445833	19660	1.602583e-01	8.7	3.812289	
-3.8	33.4	...	41296	0.64	Volcanic	5.385873	41296	8.221326e-09	2.1	1.681629	
-6.2	34.6	...	33090	0.63	Silt	5.562508	33090	6.917245e-10	1.3	0.659874	
-3.8	29.6	...	8375	0.64	Sandy	5.087792	8375	2.612715e-01	0.5	0.226532	

Note that there are a couple of `Field_ID` columns in our DataFrame that we need to remove since we're not interested in particular farms for now.

In [6]:
```
# Now, drop all columns named 'Field_ID'.
MD_agric_df.drop(columns = 'Field_ID', inplace = True)
```

Data cleanup

I noticed some errors in the data. Here's what I picked up:

1. There are some swapped column names. Please ensure to use the correct name.
2. Some of the crop types contain spelling errors.
3. The `Elevation` column contains some negative values, which are not plausible, so change these to positive values.

Use your Pandas skills to clean up the data.

In [36]:
```
# Insert your code here
# To swap column names
# MD_agric_df["Annual_yield"], MD_agric_df["Crop_type"] = MD_agric_df["Crop_type"], MD_agric_df["Annual_yield'

# To correct spealing errors step1
# MD_agric_df["Crop_type"] = MD_agric_df["Crop_type"].replace({"cassaval": "cassava", "teaa": "tea", "wheatn":

# To correct spealing errors step2
MD_agric_df["Crop_type"] = MD_agric_df["Crop_type"].replace({"cassava ": "cassava", "tea ": "tea", "wheat ": '

# To change Elevation columns with negative values to positive
# MD_agric_df["Elevation"] = MD_agric_df["Elevation"].abs()  # Change negative values to positive
```

In [37]:
```
print(MD_agric_df['Crop_type'].value_counts())
```

```
Crop_type
wheat      1342
tea         913
potato      823
cassava     672
banana      633
coffee      607
maize       399
rice        265
Name: count, dtype: int64
```

Final data checkup

Compare your answers to the expected output to make sure your data is corrected.

In [38]:
```
len(MD_agric_df['Crop_type'].unique())
```

Out[38]:
```
8
```

Expected output: `8`

In [39]:
```
MD_agric_df['Elevation'].min()
```

Out[39]:
```
35.910797
```

Expected output: `35.910797`

In [40]:
```
MD_agric_df['Annual_yield'].dtype
```

Out[40]:
```
dtype('float64')
```

Expected outcome: `dtype('float64')`

Analysis

Challenge 1: Uncovering crop preferences

Now that we have our data ready, let's delve into understanding where different crops are grown in Maji Ndogo. Our initial step is to focus on tea, a key crop in Maji Ndogo. We need to determine the optimal conditions for its growth. By analyzing data related to elevation, rainfall, and soil type specifically for tea plantations, we'll start to paint a picture of where our farming systems could thrive.

Task: Create a function that includes only tea fields and returns a tuple with the mean `Rainfall` and the mean `Elevation`. The function should input the full DataFrame, a string value for the crop type to filter by, and output a tuple with rainfall and elevation.

In [50]:
```
### START FUNCTION
def explore_crop_distribution(df,crop_filter):
    # Filtering DF by the specified crop type
    filtered_df = df[df['Crop_type'] == crop_filter]
    # Calculating the mean Rainfall and mean Elevation for the filtered crop type
    mean_rainfall = filtered_df['Rainfall'].mean()
    mean_elevation = filtered_df['Elevation'].mean()
    # Return the results as a tuple
    return mean_rainfall, mean_elevation

### END FUNCTION
```

Input:

In [51]:
```
explore_crop_distribution(MD_agric_df, "tea")
```

Out[51]:
```
(1534.5079956188388, 775.208667535597)
```

Expected output: `(1534.5079956188388, 775.208667535597)`

In [52]:
```
explore_crop_distribution(MD_agric_df, "wheat")
```

Out[52]:
```
(1010.2859910581222, 595.8384148002981)
```

Expected output: `(1010.2859910581222, 595.8384148002981)`

Repeat this for a couple of crops to get a feeling for where crops are planted in Majio Ndogo.

Challenge 2: Finding fertile grounds

With insights into tea cultivation, let's broaden our horizons. Fertile soil is the bedrock of successful farming. By grouping our data by location and soil type, we'll pinpoint where the most fertile soils in Maji Ndogo are. These fertile zones could be prime candidates for diverse crop cultivation, maximising our yield.

We'll group our data by soil type to see where the most fertile grounds are. This information will be vital for deciding where to deploy our farming technology.

Task: Create a function that groups the data by `Soil_type` , and returns the `Soil_fertility` .

In [66]:
```
### START FUNCTION
def analyse_soil_fertility(df):
  # Group data by Soil_type
  soil_groups = df.groupby('Soil_type')

  # Return the average Soil_fertility for each group
  return soil_groups['Soil_fertility'].mean()
### END FUNCTION
```

Input:

In [67]:
```
analyse_soil_fertility(MD_agric_df)
```

Out[67]:
```
Soil_type
Loamy       0.585868
Peaty       0.604882
Rocky       0.582368
Sandy       0.595669
Silt        0.652654
Volcanic    0.648894
Name: Soil_fertility, dtype: float64
```

Try digging into the data a bit more by aggregating various data to identify some more patterns.

Challenge 3: Climate and geography analysis

Now, let's delve into how climate and geography influence farming. By understanding the relationship between factors like elevation, temperature, and rainfall with crop yields, we can identify the most suitable areas for different crops. This analysis is key to ensuring our automated systems are deployed in locations that will maximise their effectiveness.

Task: Create a function that takes in a DataFrame and the column name, and groups the data by that column, and aggregates the data by the means of `Elevation` , `Min_temperature_C` , `Max_temperature_C` , and `Rainfall` , and outputs a DataFrame. Please ensure that the order of the columns matches the output.

In [71]:
```
### START FUNCTION
def climate_geography_influence(df,column):
    # Group data by column
    grouped = df.groupby(column)

    #Aggregate mean of Elevation, temperatures, rainfall
    means = grouped[['Elevation','Min_temperature_C','Max_temperature_C','Rainfall']].mean()

    # Reorder columns to match output
    means = means[['Elevation','Min_temperature_C','Max_temperature_C','Rainfall']]

    return means

### END FUNCTION
```

Input:

In [72]:
```
climate_geography_influence(MD_agric_df, 'Crop_type')
```

Out[72]:

Crop_type	Elevation	Min_temperature_C	Max_temperature_C	Rainfall
banana	487.973572	-5.354344	31.988152	1659.905687
cassava	682.903008	-3.992113	30.902381	1210.543006
coffee	647.047734	-4.028007	30.855189	1527.265074
maize	680.596982	-4.497995	30.576692	681.010276
potato	696.313917	-4.375334	30.300608	660.289064
rice	352.858053	-6.610566	32.727170	1632.382642
tea	775.208668	-2.862651	29.950383	1534.507996
wheat	595.838415	-4.968107	30.973845	1010.285991

Expected output:

```
Crop_type   Elevation   Min_temperature_C   Max_temperature_C   Rainfall
banana      487.973572  -5.354344       31.988152       1659.905687
cassava     682.903008  -3.992113       30.902381       1210.543006
coffee      647.047734  -4.028007       30.855189       1527.265074
maize       680.596982  -4.497995       30.576692       681.010276
potato      696.313917  -4.375334       30.300608       660.289064
rice        352.858053  -6.610566       32.727170       1632.382642
tea     775.208668  -2.862651       29.950383       1534.507996
wheat       595.838415  -4.968107       30.973845       1010.285991
```

Challenge 4: Advanced sorting techniques

Quite often it is better to improve the things you're good at than improving the things you're bad at. So the question is, which crop is the top performer in Maji Ndogo, and under what conditions does it perform well?

To answer this, we need to:

1. Filter all the fields with an above-average `Standard_yield` (do you have flashbacks to SQL subqueries right now?).
2. Then group by <?> crop type, using `count()` .
3. Sort the values to get the top crop type on top.
4. Retrieve the name of the top index. See the hint below on how to do this.

Task: Create a function that takes a DataFrame as input, filters, groups and sorts, and outputs a string value of a crop type.

Hint: When you have grouped by a column, we can access the labels of that "index column" using `.index` . For example:

In [73]:
```
grouped_df = MD_agric_df.groupby("Soil_type").mean(numeric_only = True).sort_values(by="Elevation",ascending=F
print(grouped_df.index[0])
grouped_df
```

```
Rocky
```

Out[73]:

Soil_type	Elevation	Latitude	Longitude	Slope	Rainfall	Min_temperature_C	Max_temperature_C	Ave_temps	Soil_fertility
Rocky	892.665740	-4.115932	-4.774825	8.631688	841.874671	-2.425658	29.131579	13.352961	0.582368
Volcanic	750.902092	-7.471007	-1.977809	10.883989	1630.504364	-2.993755	30.089992	13.548119	0.648894
Sandy	743.456509	-4.886504	-4.993451	13.902178	797.665003	-3.821689	30.175496	13.176903	0.595669
Loamy	552.383554	-4.577196	-3.715553	12.559178	724.785612	-5.620966	31.374717	12.876876	0.585868
Peaty	467.291922	-10.635362	-5.611269	3.850355	1344.381176	-5.835294	32.032941	13.098824	0.604882
Silt	424.196238	-11.138730	-6.258565	11.179052	1667.228365	-5.927452	32.306236	13.189392	0.652654

In [74]:
```
### START FUNCTION
def find_ideal_fields(df):
    # Filter for rows with Standard_yield above average
    mean_yield = df['Standard_yield'].mean()
    df = df[df['Standard_yield'] > mean_yield]

    # Group by crop_type and count rows
    grouped = df.groupby('Crop_type')['Standard_yield'].count()

    # Sort values in descending order
    grouped = grouped.sort_values(ascending=False)

    # Return name of top index
    return grouped.index[0]
### END FUNCTION
```

Input:

In [75]:
```
type(find_ideal_fields(MD_agric_df))
```

Out[75]:
```
str
```

Expected output: `str`

Challenge 5: Advanced filtering techniques

Now we know that <?> is our most successful crop, we can look at what makes it successful.

Create a function that takes a DataFrame as input, and the type of crop, and filters the DataFrame using the following conditions:

1. Filter by crop type.
2. Select only rows that have above average `Standard_yield` .
3. Select only rows that have `Ave_temps` >= 12 but =< 15.
4. Have a `Pollution_level` lower than 0.0001.

In [76]:
```python
### START FUNCTION
def find_good_conditions(df, crop_type):
    # Filter by crop type
    filt_df = df[df['Crop_type'] == crop_type]

    # Filter for Standard_yield above average
    mean_yield = df['Standard_yield'].mean()
    filt_df = filt_df[filt_df['Standard_yield'] > mean_yield]

    # Filter for Ave_temps between 12-15
    filt_df = filt_df[(filt_df['Ave_temps'] >= 12) & (filt_df['Ave_temps'] <= 15)]

    # Filter for Pollution_level < 0.0001
    filt_df = filt_df[filt_df['Pollution_level'] < 0.0001]

    return filt_df
### END FUNCTION
```

Input:

In [77]:
```python
find_good_conditions(MD_agric_df, "tea").shape
```

Out[77]: `(14, 17)`

Expected output: `(14, 17)`

Extra Pandas "nuggets"

We have not even scratched the surface of Pandas or our dataset. If you remember back to your days with Chidi, it took a while before we could unlock the secrets the survey data had. So, scratch around a bit.

On the Pandas front, it's the same. Pandas is a very powerful data analysis tool that takes a while to master. Many of the more advanced methods like window functions, dynamically retrieving or changing data, vectorisation, or processing big data with Pandas are all more advanced topics we encounter in the workplace.

But here are two tiny 'nuggets' to upskill in Pandas.

df.query()

Oh, you're going to love this one... `df.query()` was designed to filter data, but using SQL-like syntax. For example:

In [78]:
```python
MD_agric_df.query('Standard_yield > 0.5 and Soil_type == "Loamy"')
```

Out[78]:

	Elevation	Latitude	Longitude	Location	Slope	Rainfall	Min_temperature_C	Max_temperature_C	Ave_temps	Soil_fert
3	574.94617	-2.420131	-6.592215	Rural_Kilimani	7.109855	328.8	-5.8	32.2	13.20	(
19	610.99400	-6.948353	-2.966000	Rural_Hawassa	16.369598	902.8	-4.9	28.4	11.75	(
22	504.42505	-2.379580	-7.601249	Rural_Kilimani	3.906222	335.3	-6.4	32.6	13.10	(
26	525.38104	-2.465188	-5.186775	Rural_Kilimani	6.809244	298.6	-6.3	33.8	13.75	(
44	562.14720	-7.953123	-2.222739	Rural_Sokoto	24.619905	1200.3	-5.3	31.8	13.25	(
...	...	...	...	...	...	...	...	...	...	
5617	636.75790	-7.322326	-5.128215	Rural_Hawassa	17.667543	812.0	-4.8	30.6	12.90	(
5621	509.44156	-6.614932	-1.641644	Rural_Sokoto	19.441830	958.9	-5.8	31.5	12.85	(
5625	614.65955	-6.948353	-3.029065	Rural_Hawassa	2.801264	896.6	-4.9	28.5	11.80	(
5632	618.01624	-6.511301	-2.574099	Rural_Hawassa	19.183754	888.0	-4.9	26.9	11.00	(
5646	589.28860	-4.519784	-7.713864	Rural_Kilimani	33.468243	464.5	-5.5	32.4	13.45	(

808 rows × 17 columns

Isn't that much easier to read and understand than the one below?

In [79]:
```python
MD_agric_df[(MD_agric_df['Standard_yield'] > 0.5) & (MD_agric_df['Soil_type'] == 'Loamy')]
```

Out[79]:

	Elevation	Latitude	Longitude	Location	Slope	Rainfall	Min_temperature_C	Max_temperature_C	Ave_temps	Soil_fert
3	574.94617	-2.420131	-6.592215	Rural_Kilimani	7.109855	328.8	-5.8	32.2	13.20	(
19	610.99400	-6.948353	-2.966000	Rural_Hawassa	16.369598	902.8	-4.9	28.4	11.75	(
22	504.42505	-2.379580	-7.601249	Rural_Kilimani	3.906222	335.3	-6.4	32.6	13.10	(
26	525.38104	-2.465188	-5.186775	Rural_Kilimani	6.809244	298.6	-6.3	33.8	13.75	(
44	562.14720	-7.953123	-2.222739	Rural_Sokoto	24.619905	1200.3	-5.3	31.8	13.25	(
...	...	...	...	...	...	...	...	...	...	
5617	636.75790	-7.322326	-5.128215	Rural_Hawassa	17.667543	812.0	-4.8	30.6	12.90	(
5621	509.44156	-6.614932	-1.641644	Rural_Sokoto	19.441830	958.9	-5.8	31.5	12.85	(
5625	614.65955	-6.948353	-3.029065	Rural_Hawassa	2.801264	896.6	-4.9	28.5	11.80	(
5632	618.01624	-6.511301	-2.574099	Rural_Hawassa	19.183754	888.0	-4.9	26.9	11.00	(
5646	589.28860	-4.519784	-7.713864	Rural_Kilimani	33.468243	464.5	-5.5	32.4	13.45	(

808 rows × 17 columns

The nice thing is, we can use `in []`, `not in []` to filter with, and also pass in variables using `@variable_name`.

In [80]:
```python
soil_types = ['Loamy', 'Sandy', 'Silt']

MD_agric_df.query('Soil_type in @soil_types')
```

Out[80]:

	Elevation	Latitude	Longitude	Location	Slope	Rainfall	Min_temperature_C	Max_temperature_C	Ave_temps	Soil_fe
0	786.05580	-7.389911	-7.556202	Rural_Akatsi	14.795113	1125.2	-3.1	33.1	15.00	
3	574.94617	-2.420131	-6.592215	Rural_Kilimani	7.109855	328.8	-5.8	32.2	13.20	
4	886.35300	-3.055434	-7.952609	Rural_Kilimani	55.007656	785.2	-2.5	31.0	14.25	
5	850.56647	-2.050665	-7.132769	Rural_Kilimani	50.451250	649.4	-3.0	29.5	13.25	
6	331.35538	-13.409517	-6.722849	Rural_Hawassa	5.907423	1586.0	-6.8	31.8	12.50	
...	...	...	...	...	...	...	...	...	...	
5648	748.03925	-2.284961	-3.204745	Rural_Kilimani	3.848560	850.5	-3.7	30.7	13.50	
5649	681.36145	-7.358371	-6.254369	Rural_Akatsi	16.213196	885.7	-4.3	33.4	14.55	
5650	667.02120	-3.154559	-4.475046	Rural_Kilimani	2.397553	501.1	-4.8	32.1	13.65	
5652	429.48840	-14.653089	-6.984116	Rural_Hawassa	13.944720	1272.2	-6.2	34.6	14.20	
5653	763.09030	-4.317028	-6.344461	Rural_Kilimani	35.189430	516.4	-3.8	29.6	12.90	

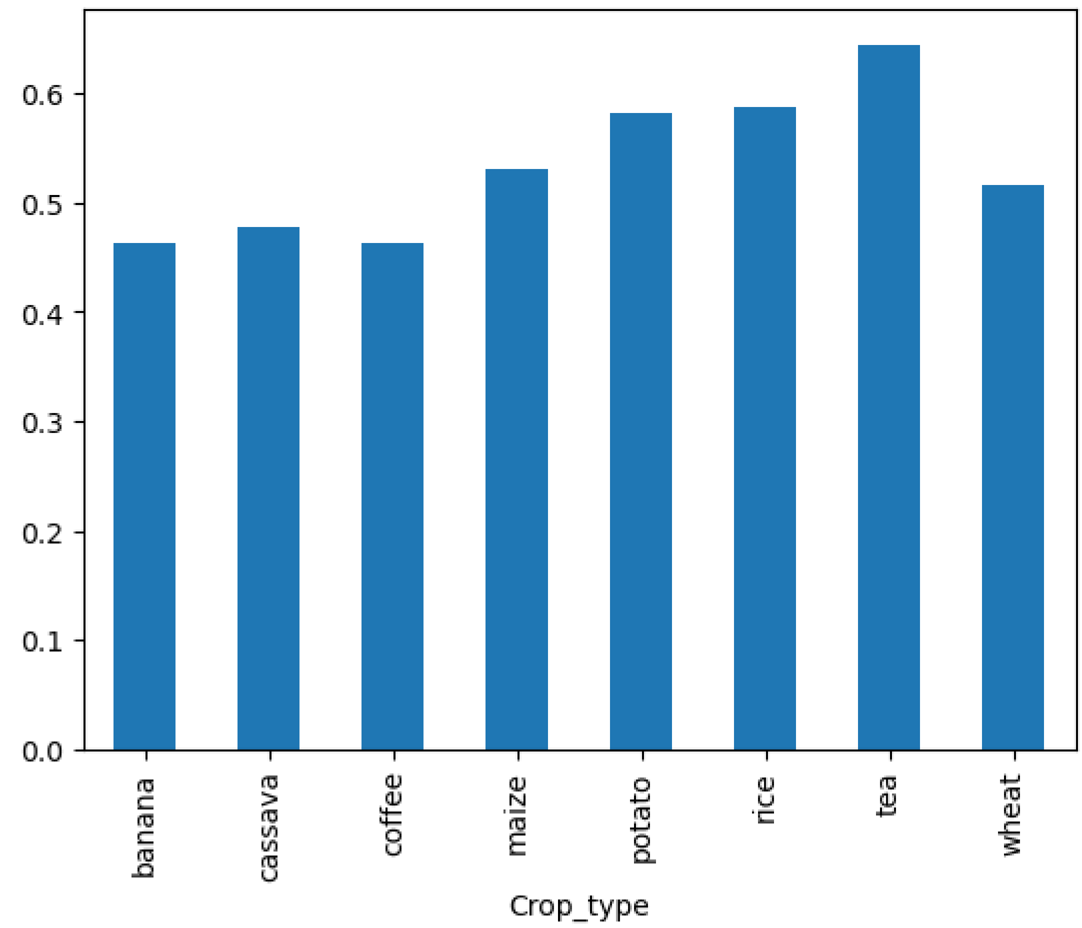
3851 rows × 17 columns

Plotting data with Pandas

Sometimes we quickly want to see a basic visualisation of our data. we can use `df.plot(kind='bar')` to make a bar plot, `df.plot(kind='hist', bins = 10)` to see the distribution of a data column, or `df.plot(kind='scatter', x='Column_on_x', y='Column_on_y')` to understand the relationship between variables.

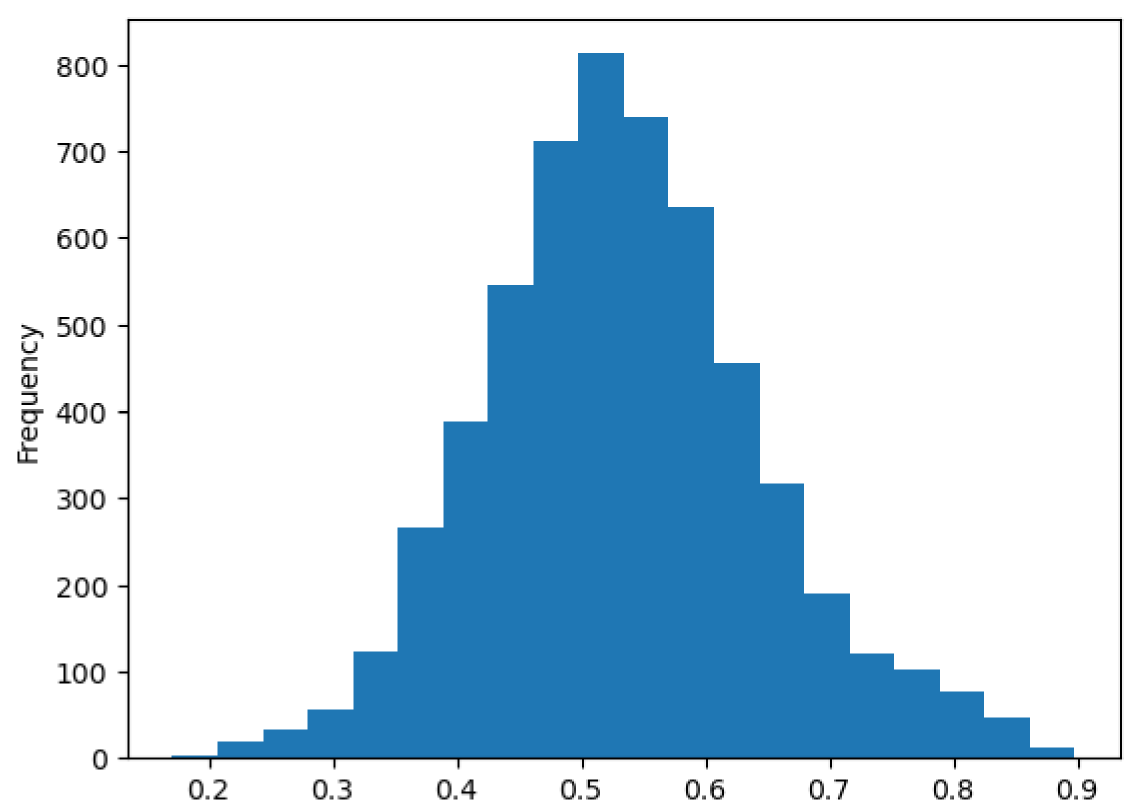
In [81]:
```
MD_agric_df.groupby('Crop_type')['Standard_yield'].mean().plot(kind='bar')
```

Out[81]: <Axes: xlabel='Crop_type'>



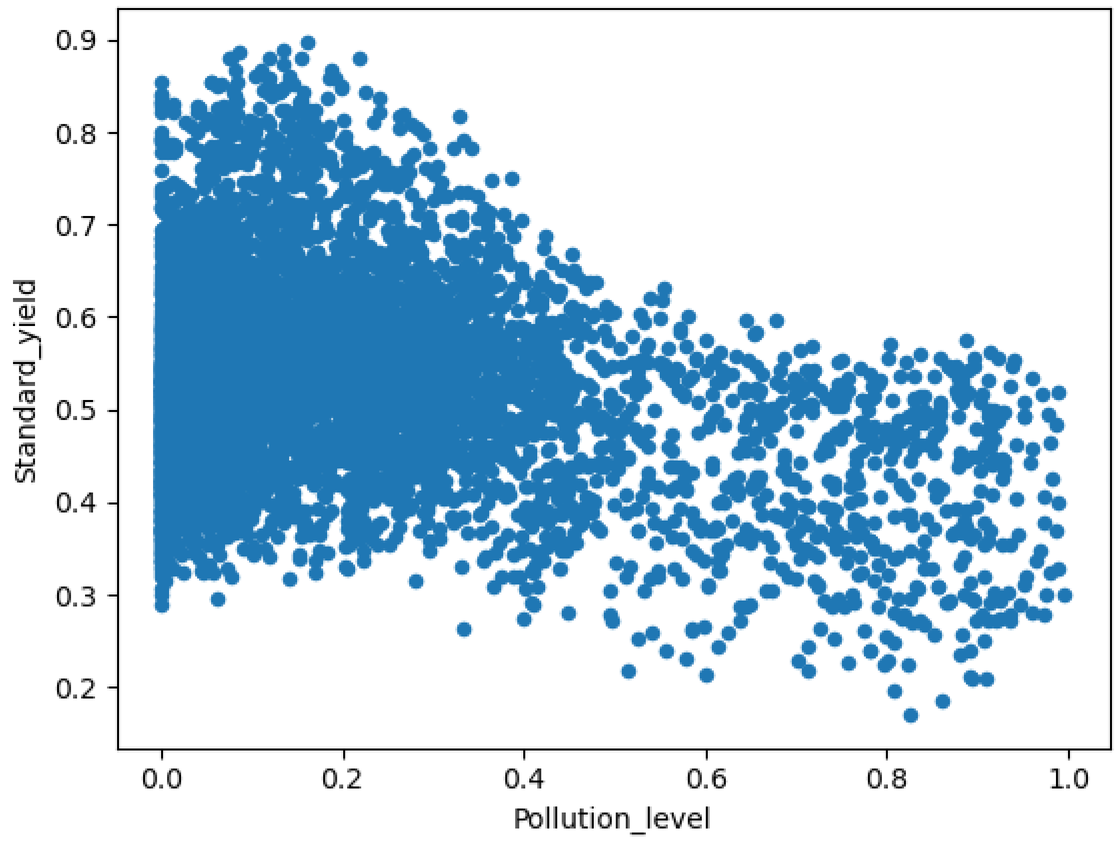
In [82]:
```
MD_agric_df['Standard_yield'].plot(kind='hist', bins =20)
```

Out[82]: <Axes: ylabel='Frequency'>



```
In [83]: MD_agric_df.plot(kind='scatter', x = "Pollution_level", y = "Standard_yield")
```

```
Out[83]: <Axes: xlabel='Pollution_level', ylabel='Standard_yield'>
```



We can use these plots to get a quick feel for the data, but we can't really customise these much. For that we need some better tools.