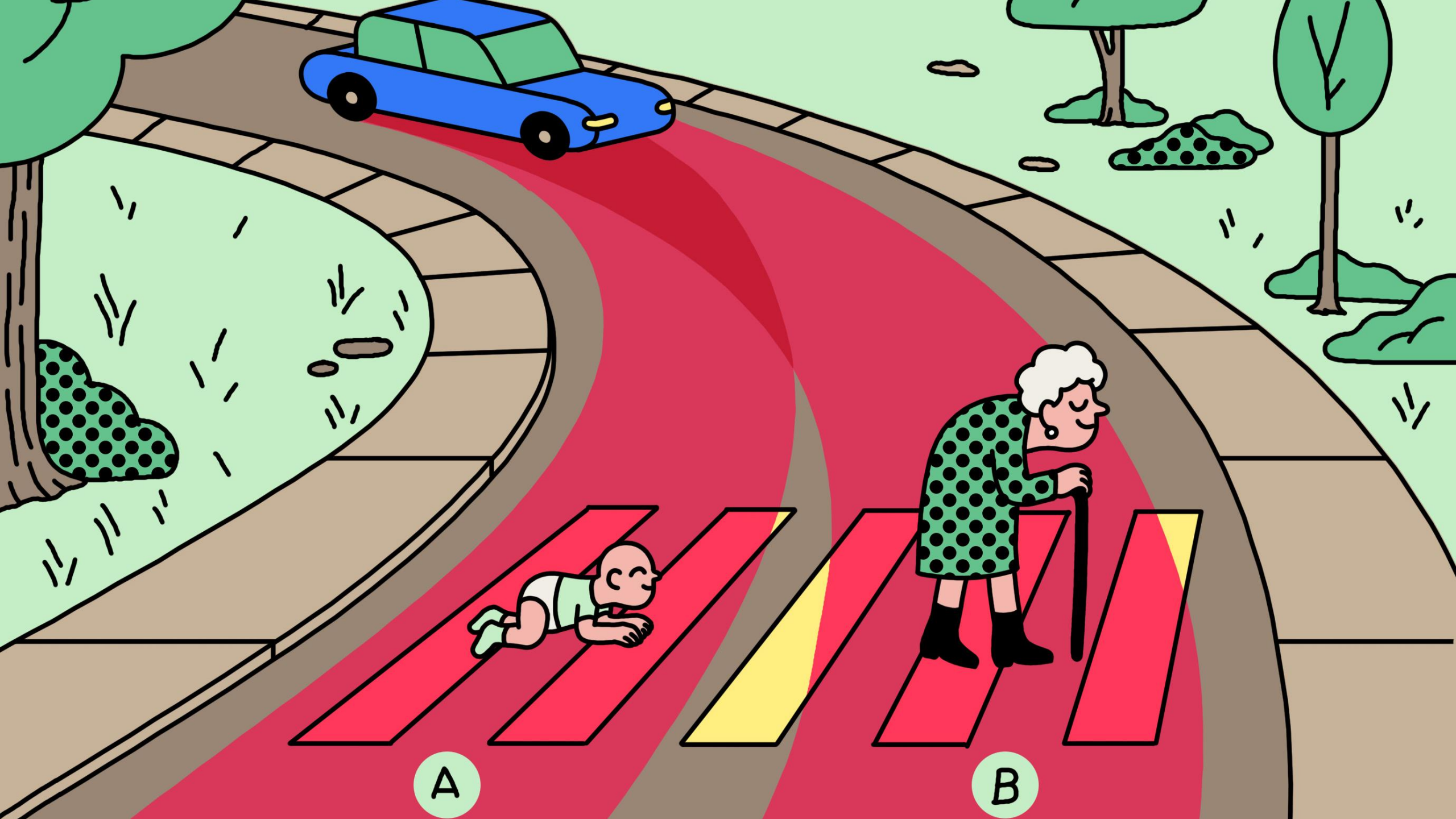
Python for Data Analytics in Manufacturing

A 2-Day Practical Training for Osram Engineers

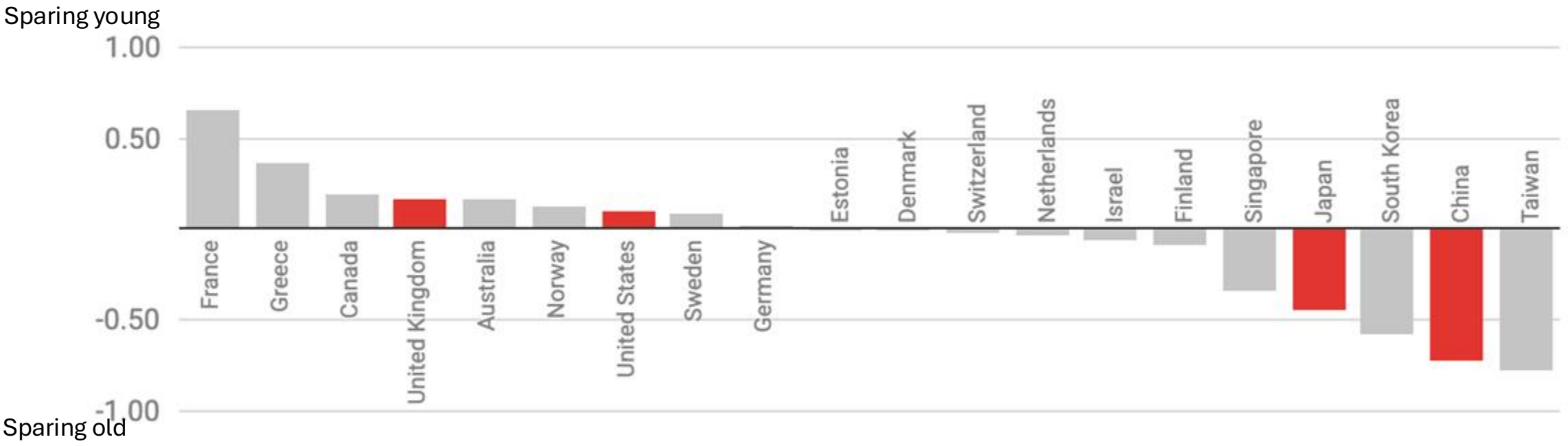
Pre-Assessment Quiz


Ice Breaker


A
B

Countries with more individualistic cultures are more likely to spare the young

GROWTH.PRO
DATA SCIENCE CONSULTING & TRAINING
BigData.Growth.Pro



A comparison of countries piloting self-driving cars: If the bar is closer to 1, respondents placed a greater emphasis on sparing the young; if the bar is closer to -1, respondents placed a greater emphasis on sparing the old; 0 is the global average.

Created with Datawrapper

Objectives & Outcomes

Course Objective

- To gain hands-on experience using Python libraries such as pandas, NumPy, and Matplotlib to clean, explore, visualize, and automate data workflows. The training emphasizes manufacturing-specific use cases, such as sensor data analysis, defect tracking, time series forecasting, and automated reporting.

Learning Outcomes

- Understand key Python libraries used for data handling, transformation, and visualization.
- Clean and process structured manufacturing data from sensors, machines, and production lines.
- Perform exploratory data analysis (EDA) to uncover trends, patterns, and anomalies.
- Create visual dashboards and charts to support operational decision-making.
- Automate reporting processes and analyze time-series data from equipment and environmental sensors.
- Apply Python-based analytics to solve real-world manufacturing problems.

House Rules

This workshop is highly interactive

- Ask lots of questions
- Everyone’s input is equally valued
- Share your experiences
- Discussions and criticisms will focus on interests, not people
- There is no bad idea
- Be open to new concepts and ideas
- Slience your phone ☺

GROWTH.PRO
DATA SCIENCE CONSULTING & TRAINING
BigData.Growth.Pro

Ng Keng Fai

Data Science Lead | Trainer

Intrigued by the ever-evolving world of AI, I'm constantly exploring its potential to create positive change.

All rounder coder | Vision AI | LLM | AI Automations

GROWTH.PRO DATA SCIENCE CONSULTING & TRAINING	Data Science Lead Trainer
UNIRAZAK UNIVERSITI TUN ABDUL RAZAK	Part Time Lecturer
NEW ERA INSTITUTE	Vocational Trainer
MYTUITION	CTO / Consultant
Technip	Instrument & Automation Engineer
FLEXTRONICS	Test Engineer
MONASH University	UAV Human Detection (Search Rescue Mission)

Qualifications

BEng in Mechatronics (Monash)
Prof. Diploma in Digital Marketing

GROWTH.PRO
DATA SCIENCE CONSULTING & TRAINING
BigData.Growth.Pro

Day 1: Laying the Foundation

1. PYTHON ESSENTIALS & GOOGLE COLAB

2. DATA HANDLING WITH PANDAS & NUMPY

3. DATA CLEANING & FEATURE ENGINEERING

4. EDA & VISUALIZATION (CONVENTIONAL STATISTICS)

Beyond Spreadsheets: The Power of Python

Scalability: Analyze large sensor datasets without crashing.

Automation: Automate repetitive reporting and tasks.

Insight: Uncover patterns not visible in dashboards.

Our Training Environment: Google Colab

Zero setup required.

Runs entirely in your web browser.

Perfect for hands-on, collaborative learning.

Core Concepts: Variables & Data Types

```
machine_id = "M01-A" (String)
production_yield = 0.992 (Float)
units_tested = 15000 (Integer)
```

Core Concepts: Lists

```
temp_readings = [24.5, 24.8, 25.1, 25.0, 24.9]
print(temp_readings[2])
# Output: 25.1
```

Core Concepts: Loops

```
for temp in temp_readings:

  if temp > 25.0:

    print(f'High Temp Alert: {temp}°C')
```

Hands-On Activity: First Script

1. Create a list of three machine IDs.
2. Write a for loop to print each ID.
3. Define a function to check if a temperature is in a safe range.

Module 1 Recap

The 'why' behind using Python for data.

Our environment: Google Colab.

Core building blocks: variables, loops, functions.

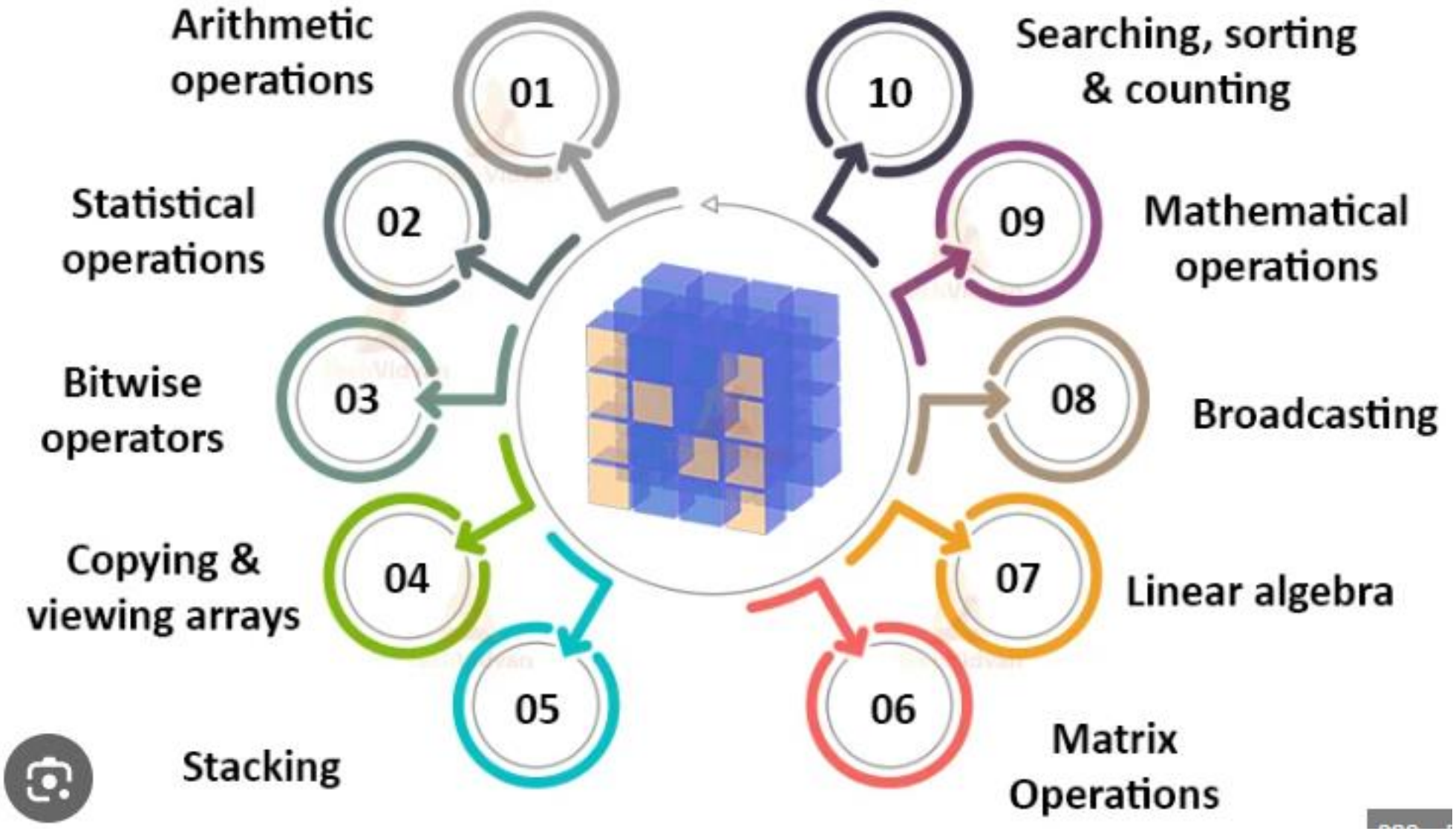
The Need for Specialized Libraries

- Why Not Just Use `Lists`?
- `Lists` are slow for large datasets.
- No built-in tools for statistical analysis.
- **Inefficien**t for multi-dimensional data.

Introduction to NumPy

- The Foundation: NumPy Arrays
- High-performance, multi-dimensional arrays.
- Faster than Python lists for numerical operations.
- The backbone of pandas and other scientific libraries.

Uses of NumPy



Introduction to pandas

- The Data Analyst's Best Friend
- DataFrame: A 2D table-like structure (rows, columns).
- Series: A 1D column of data.
- Think of it as a super-powered spreadsheet.

Loading Our Dataset

```
import pandas as pd

from google.colab import files

uploaded = files.upload()

df = pd.read_csv('Osram_temp_data.csv')
```

Hands-On: Loading the Data

Task: Upload the provided Osram_temp_data.csv to Colab.

Task: Load it into a pandas DataFrame and name it df.

Initial Data Inspection: `head()`

- `df.head()`

Initial Data Inspection: `info()`

- `df.info()`

Initial Data Inspection: `describe()`

- `df.describe()`

Hands-On: Inspecting the Data

- Task: Use df.info() to check for missing values or incorrect data types.
-
- Task: Use df.describe() to see the mean and range of our temperature readings.

Accessing Data: Selecting Columns

```
temp_1 = df['Temp_1']
all_temps = df[['Temp_1', 'Temp_2',
'Temp_3']]
```

Accessing Data: `loc` & `iloc`

- `df.loc[...]`: Selection by label/name (e.g., column names, index values).
- `df.iloc[...]`: Selection by position/integer index (e.g., row 0, column 1).

Hands-On: Selecting Data

1. Select only the position_index and Temp_1 columns.
2. Use iloc to select the first 100 rows and all temperature columns.

Filtering Data with Boolean Masks

- `hot_points = df[df['Temp_1'] > 28.0]`

Hands-On: Filtering the Osram Data

1. Filter the DataFrame to find all rows where Temp_Delta_1-2 > 1.5.
2. Count how many data points meet this condition.

Common DataFrame Operations

- `df.sort_values(by='column_name'):` Sort rows by column values.
- `df.drop('column_name', axis=1):` Remove columns.
- `df.reset_index():` Convert index back to a column.

Module 2 Recap

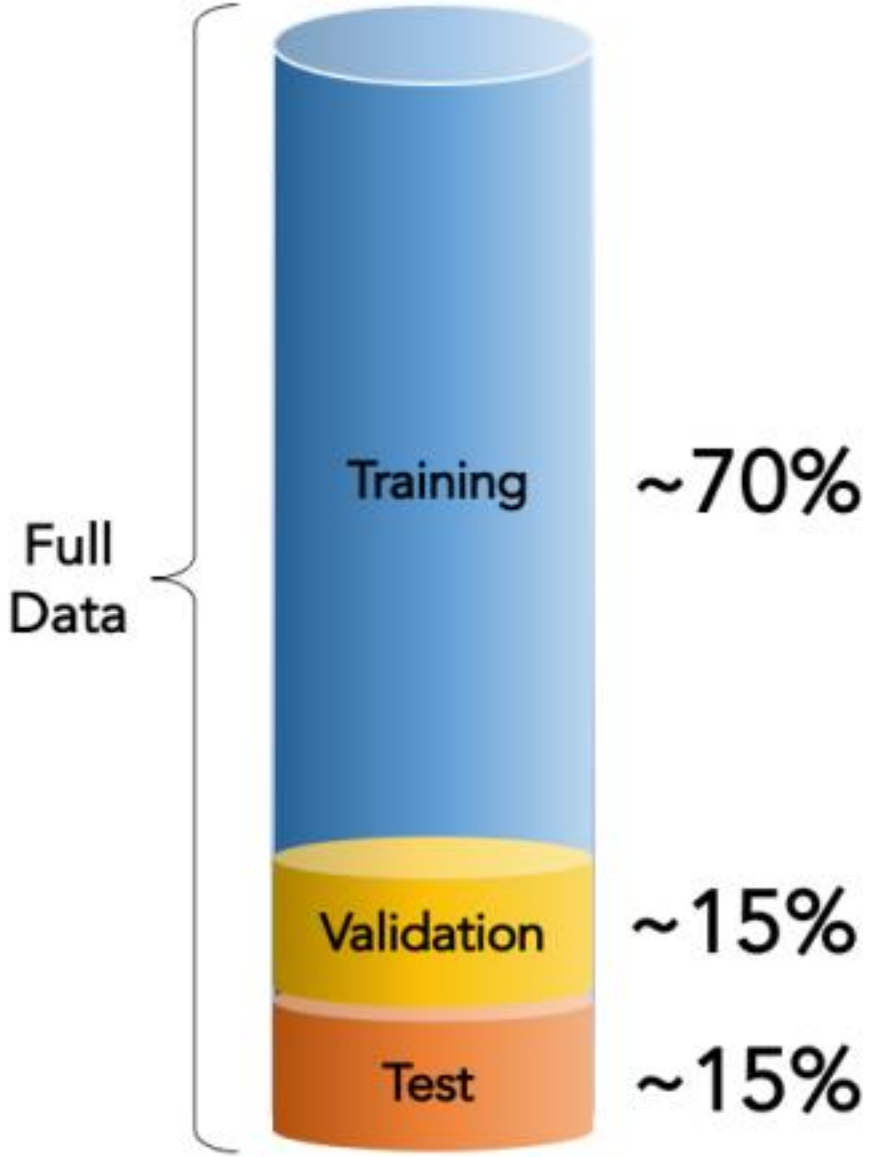
- We learned to load, inspect, select, and filter data using pandas.
- The DataFrame is our primary tool for data manipulation.
- Key functions: head(), info(), describe(), boolean masks.

The Reality of Raw Data

- The 80/20 Rule: 80% of data analysis time is spent on cleaning.
- Common Issues:
- Missing values (e.g., a sensor didn't report).
- Incorrect data types (e.g., numbers as text).
- Duplicates or inconsistencies.

Preparing for Training: Data Splitting

- **Why Split?** To evaluate model performance on unseen data – simulating real-world application.
- **Train Set:** Used to teach the model patterns.
- **Validation Set:** Used during development to fine-tune model settings and choose the best model.
- **Test Set:** A completely separate, untouched dataset for final, unbiased evaluation of the chosen model.
- **Typical Ratios:** 70/15/15 or 80/10/10.
- **Stratified Sampling:** Important for imbalanced data (e.g., ensuring both 'default' and 'non-default' cases are proportionately represented in each split).



Spotting Data Issues

- **Scenario:** Imagine a new loan application system is launched. Below are examples of data you might collect.
- **Task (Group Discussion):** For each example, identify potential data quality issues and suggest *why* it's a problem

Timestamp	Machine_id	Status	Temperature	Pressure
2025-03-01 11:35:00	25A	ERROR	500	100
2025-03-02 11:35:00	30A	Running	750	50
2025-03-03 11:35:00	Missing	Running	1200	-
2025-03-04 11:35:00	150A	Halted	0	0
04-05-2025 11:35:00	40A	S	800	40

Step 1: Handling Missing Values

Why Data is Missing: Data entry errors, non-applicable fields, data collection issues.

Impact: Can bias models, reduce dataset size, lead to errors.

Conceptual Strategies:

Deletion: Remove rows/columns (but risk losing valuable data).

Imputation (Filling in):

- **Mean/Median/Mode:** Replace with average/middle value/most common.
- **More Sophisticated Methods:** Using other data to predict missing values.

Managerial Implication: Data pipelines and data governance are essential to minimize missing data upfront.

Step 2: Outlier Detection and Treatment

What are Outliers? Data points significantly different from others (e.g., someone claiming 150 years old or incredibly high income).

Causes: Measurement errors, data entry errors, genuine extreme values.

Impact: Can skew distributions, distort model parameters, affect performance.

Conceptual Detection Methods:

- Visual inspection (e.g., charts).
- Statistical rules (e.g., values far from the average).

Conceptual Treatment:

- Investigating and correcting errors.
- Capping/transforming extreme values to a reasonable range.
- Deletion (only for clear errors and with caution).

Managerial Implication: Flagging processes for unusual data for human review.

Step 3: Categorical Data Encoding

Why Encode? ML algorithms inherently work with numerical data, not text categories like 'Male'/'Female' or 'Student'/'Employed'.

Types of Categories:

- **Nominal:** No inherent order (e.g., 'Gender', 'Loan Type').
- **Ordinal:** Has a clear order (e.g., 'Education Level: Primary, Secondary, Tertiary').

Conceptual Encoding:

- **'One-Hot' Encoding:** Create separate 'yes/no' columns for each category (e.g., 'Is_Male', 'Is_Female').
- **Label Encoding:** Assign numbers based on order (e.g., Primary=1, Secondary=2).
- **Advanced:** Transform based on relationship with outcome (e.g., 'Weight of Evidence').

Managerial Implication: Data dictionary and clear definitions for categories are crucial.

Step 4: Feature Scaling

Why Scale? Features with very different numerical ranges can disproportionately influence models (e.g., 'Income' vs. 'Number of Dependents').

Impact: Prevents features with larger values from dominating the learning process.

Conceptual Methods:

- **Min-Max Scaling:** Shrink all values to a fixed range (e.g., 0 to 1).
- **Standardization:** Transform values to have an average of 0 and consistent spread.

Essential for many algorithms (e.g., those using distances).

Managerial Implication: Consistency in data representation across systems.

Identifying Missing Values

```
df.isna().sum()
```

Handling Missing Values: `fillna()`

```
# Fill missing values with a specific number (e.g., 0)

df.fillna(0, inplace=True)

# Fill with the mean of the column

df['Temp_1'].fillna(df['Temp_1'].mean(), inplace=True)
```

Hands-On: Missing Data Practice

- Task: Check your Osram DataFrame for any missing values.
- Task: (Instructor will manually remove a value for demonstration). Fill the missing value in Temp_1 with the average of that column.

Identifying Duplicates

```
df.duplicated().sum()
# To see the duplicate rows:
df[df.duplicated()]
```

Correcting Data Types

When Numbers are Treated as Text:

```
df['units_tested'] =
df['units_tested'].astype(int)
```

Ensuring Consistency: Check df.info() for object types that should be numeric.

The Importance of Timestamps

- Beyond a position_index:
- The position_index is sequential, but it needs to be treated as a time.
- This unlocks powerful time-based analysis like trends and seasonality.

Converting `position_index` to Datetime

- `df['position_index'] = pd.to_datetime(df['position_index'], unit='ms')`

Setting the Index

- `df.set_index('position_index', inplace=True)`

Hands-On: Converting to Timestamps

1. Convert your position_index column to a datetime type.
2. Set this new datetime column as the DataFrame's index.
3. Confirm the change with df.info().

What is Feature Engineering?

- Creating New, More Powerful Features:
- Generating new columns from existing data.
- Example: Temp_Delta is a manually engineered feature.
- Provides more context to our analysis and models.

Engineering Our Own Delta

- `df['Calculated_Delta_1-2'] = abs(df['Temp_1'] - df['Temp_2'])`

Categorizing Data

- Binning Numerical Data:
- bins = [0, 25, 30, 100]
- labels = ['Cold', 'Normal', 'Hot']
- df['Temp_Category'] = pd.cut(df['Temp_1'], bins=bins, labels=labels)

Module 3 Recap

- We learned to handle messy data: missing values, duplicates, and incorrect types.
- We also practiced creating new features and converting to time-series format.

What is Exploratory Data Analysis (EDA)?

- The Detective Work of Data:
- Goal: To uncover patterns, trends, and anomalies.
- Tools: Statistical summaries and data visualization.
- Outcome: Insights that drive operational decisions.

Statistical Summaries with `groupby()`

- Aggregating Data:
- `df.groupby('Temp_Category')['Temp_1'].mean()`

Hands-On: Grouping Data

1. Group the DataFrame by your new Temp_Category column.
2. Calculate the average Temp_Delta_1-2 for each category.
3. What insights does this reveal about temperature differences across categories?

Introduction to `Matplotlib`

- The Visual Language of Data:
- The most widely used foundational plotting library in Python.
- Provides flexibility for creating various types of plots.
- Essential for visualizing trends, patterns, and outliers.

Introduction to Seaborn

- Beautiful & Informative Plots:
- A higher-level library built on top of Matplotlib.
- Makes creating complex statistical charts easy and visually appealing.
- Ideal for exploring relationships between variables.

Simple Line Plots

- Visualizing Trends Over Time:

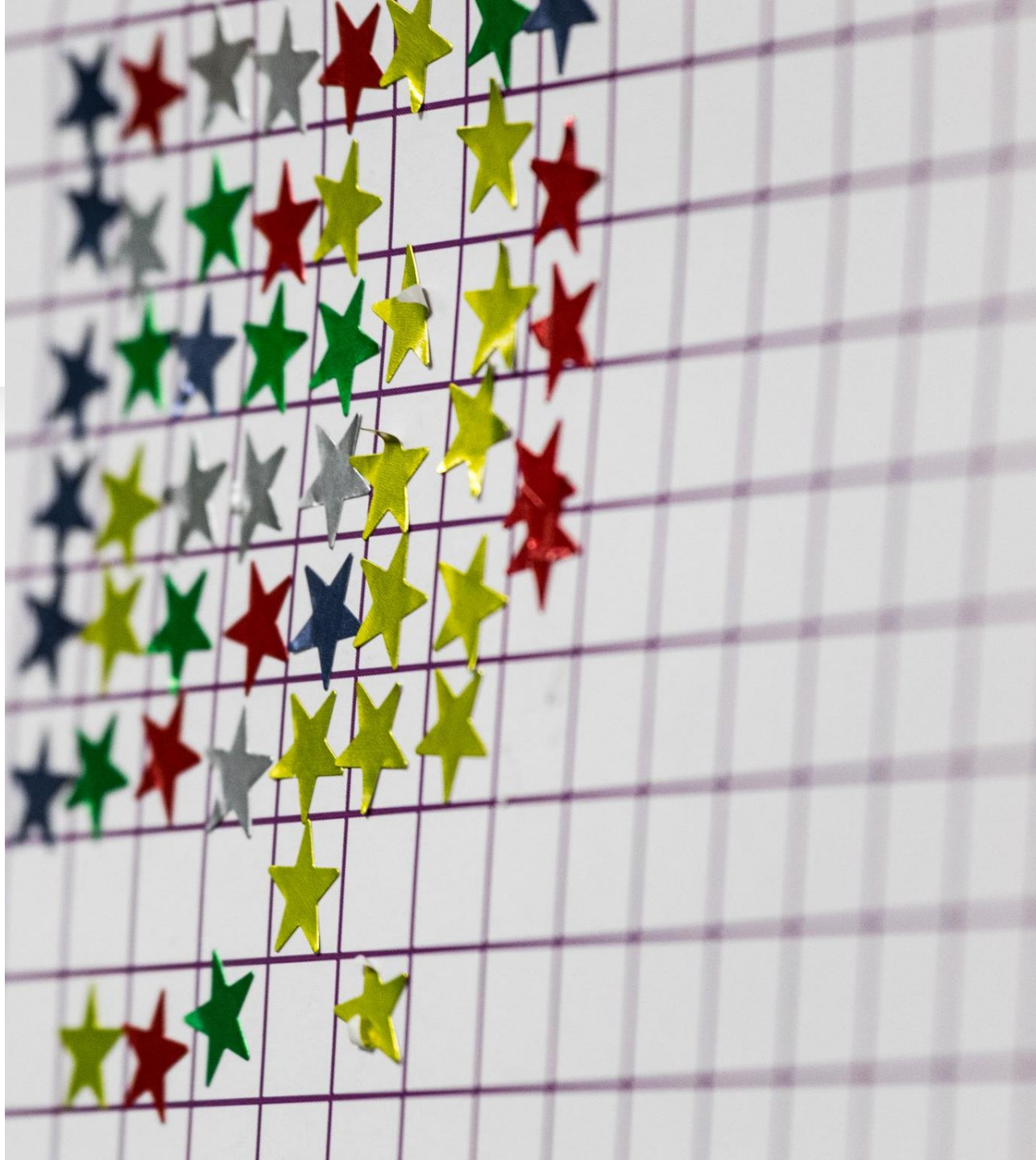
```
import matplotlib.pyplot as plt
df[['Temp_1', 'Temp_2', 'Temp_3']].plot(figsize=(12, 6))
plt.title('Temperature Readings Over Time')
plt.xlabel('Position Index (Time)')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.grid(True)
plt.show()
```

Hands-On: Plotting Our Temperatures

1. Create a line plot showing Temp_1, Temp_2, and Temp_3 over time.
2. Add a title and axis labels to your plot.
3. What do you notice? Are the temperatures stable? Do you see any unusual spikes or dips?

Conventional Outlier Detection

- Finding the 'Red Flags':
- Thresholds: Simple rule-based flags (e.g., if temp > 30).
- Z-score: Measures how many standard deviations a point is from the mean.
- Box Plots: Visualizing data distribution and outliers.



Calculating the Z-score

- The Math Behind the Outlier:

```
from scipy.stats import zscore
df['Temp_1_zscore'] = zscore(df['Temp_1'])
# A common threshold for an outlier is |z-score| > 3
```

Visualizing `z-score` Outliers

Highlighting Anomalies:

We will create a plot that shows the raw data and highlights the points where the absolute z-score is > 3 with a different color.

Hands-On: Z-score Outliers

1. Calculate the z-score for the Temp_Delta_1-2 column.
2. Create a new column to flag any data point with an absolute z-score > 3.0 as an outlier.
3. Plot the raw Temp_Delta_1-2 data and highlight these flagged outliers.

Trend and Drift Analysis

- Spotting Gradual Changes:
- The rolling() function calculates statistics over a sliding window.
- It helps to smooth out noise and reveal long-term trends or gradual shifts.
- Essential for predictive maintenance and process control.

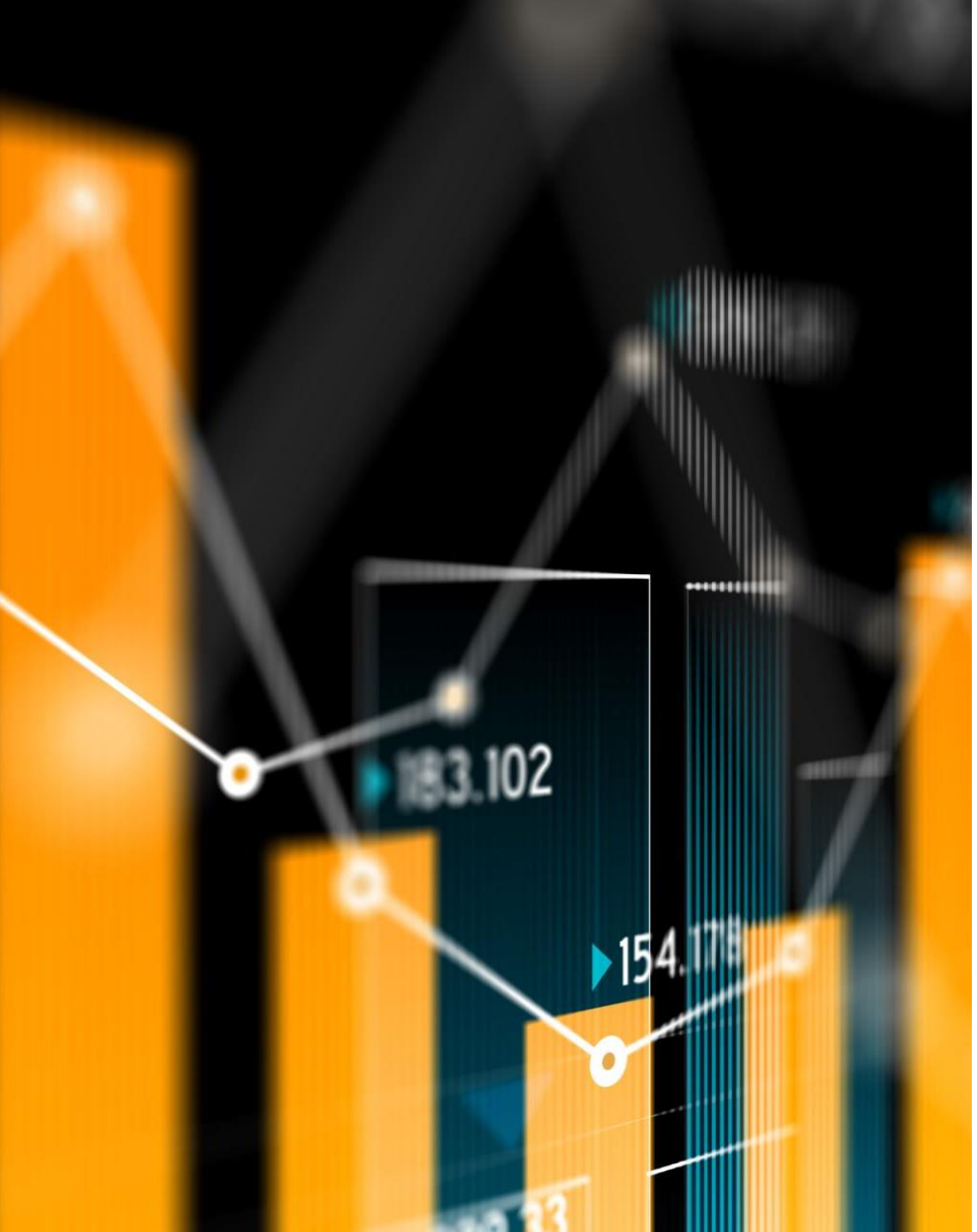
Calculating the Rolling Mean

- Smoothing Out Noise:

```
rolling_avg = df['Temp_1'].rolling(window=100).mean()
plt.figure(figsize=(12, 6))
plt.plot(df['Temp_1'], label='Raw Temp_1', alpha=0.7)
plt.plot(rolling_avg, label='100-Point Rolling Average',
color='red')
plt.legend()
plt.show()
```

Hands-On: Rolling Mean

1. Calculate and plot the rolling mean for Temp_3 with a window of 50.
2. Plot the raw Temp_3 data and its rolling mean on the same chart.
3. What gradual changes or drifts do you observe? How might this impact production?



End of Day 1: Final Q&A

- Today: From raw data to clean, insightful plots.
- We covered Python essentials, pandas for data handling, and performed cleaning, basic EDA, and conventional statistical outlier/trend analysis.
- Tomorrow: We'll take this analysis to the next level with AI/ML.

Day 2 Agenda

1. TIME SERIES ANALYSIS

2. INTRO TO AI/ML ANOMALY DETECTION

3. REPORTING AUTOMATION & FINAL PROJECT

The Power of a Datetime Index

The position_index is now our main reference for time.

Having a datetime index unlocks powerful time-based operations.

We can easily select data based on specific time ranges.

Slicing Data by Time

- Filtering by a Time Window:
- df.loc['2025-07-27 10:00':'2025-07-27 11:00']
- df.loc['2025-07-27']

Hands-On: Time-Based Slicing

1. Using your Osram dataset, select and display all data points from a specific time window (e.g., a short window of 500 data points if your index is sequential, or a specific hour if it's a real timestamp).
2. Plot Temp_1 and Temp_2 for just that time window.

Resampling Data

- Changing the Granularity:
- Goal: To analyze data at a different frequency (e.g., from seconds to minutes).

df.resample('H'): Resample to hourly.

df.resample('D'): Resample to daily.

- Useful for summarizing high-frequency sensor data.

Resampling with Aggregation

- From Raw to Summary:
- `daily_avg_temp1 = df['Temp_1'].resample('D').mean()`
- `five_min_max_delta = df['Temp_Delta_1-2'].resample('5T').max()`

Hands-On: Resampling Our Data

1. Resample the Temp_1 and Temp_2 data to the average value every 500 position index points (or a 1-hour average if using real timestamps).
2. Plot the raw data and the resampled data on the same chart.
3. What does this reveal about the overall trend that was harder to see before?

Advanced Time Series Visualization

1. Visualizing rolling standard deviation to show volatility.
2. Highlighting specific events or anomalies on a time series plot.
3. Using subplots for comparing different metrics over time.

Module 5 Recap

We learned to use the position_index as a time index, slice data by time, and resample our data to a different frequency.

These are crucial for understanding time-dependent manufacturing processes.

Why We Need More Than `z-score`

z-score works well for simple, single-variable outliers.

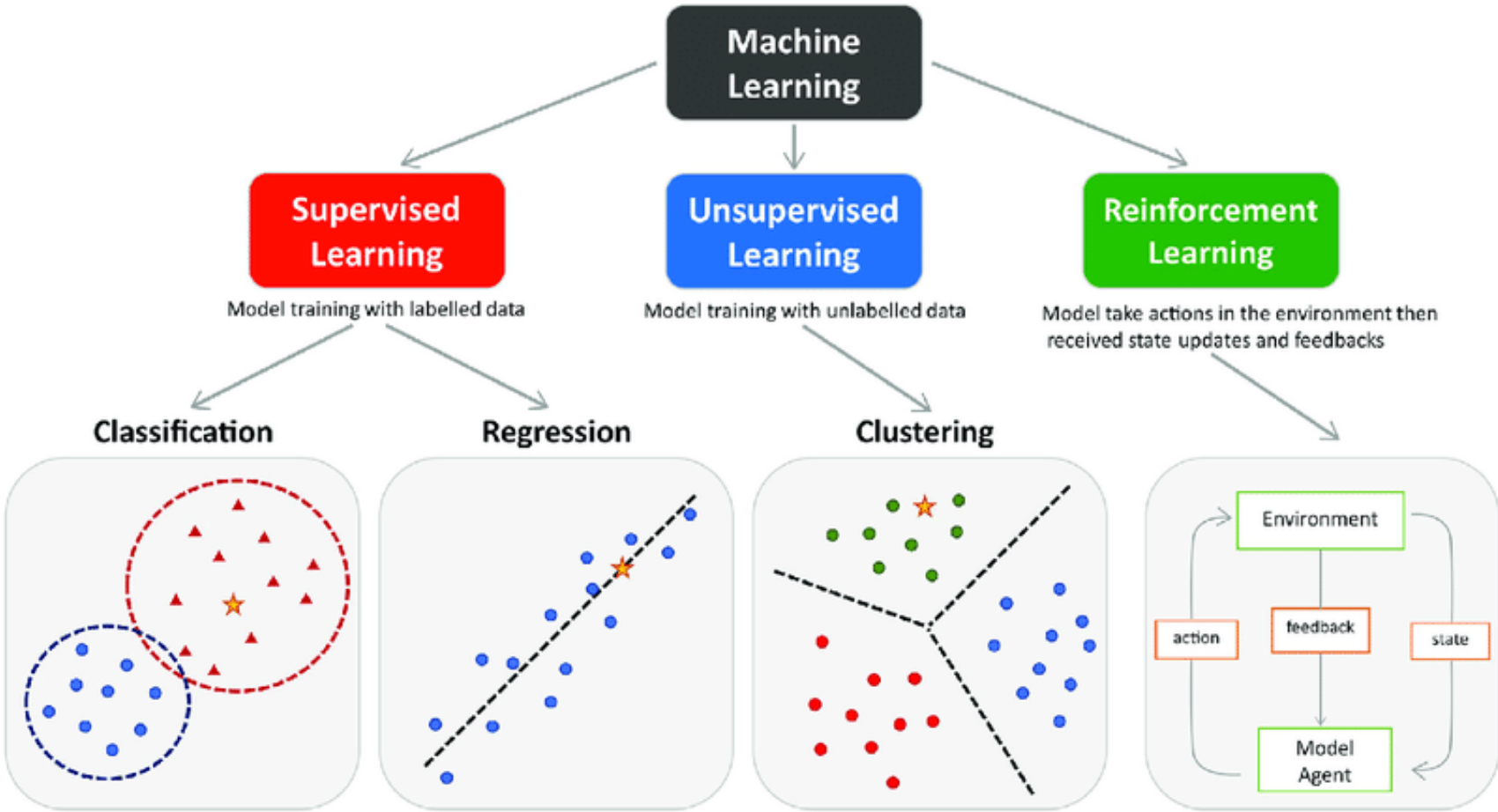
What if the anomaly is a **complex** combination of Temp_1 and Temp_Delta?

Traditional rules can be hard to define for complex systems.

The AI/ML Approach to Anomaly Detection

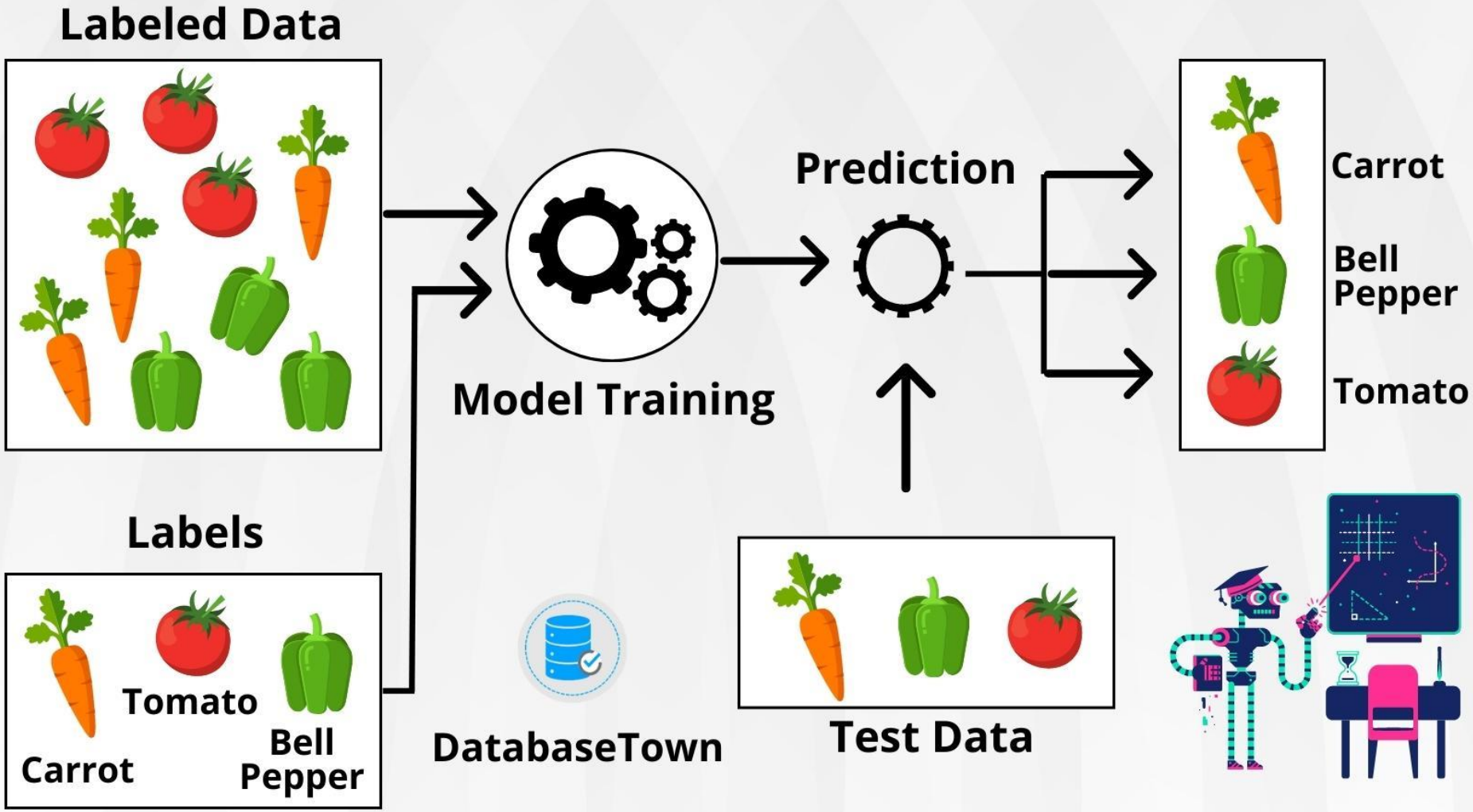
- Learning 'Normal' Behavior:
- Instead of us defining rules, the model learns the patterns from the data itself.
- It can detect subtle, multi-variable anomalies that are hard to spot manually.
- Adapts to changing 'normal' conditions.

Types of Machine Learning



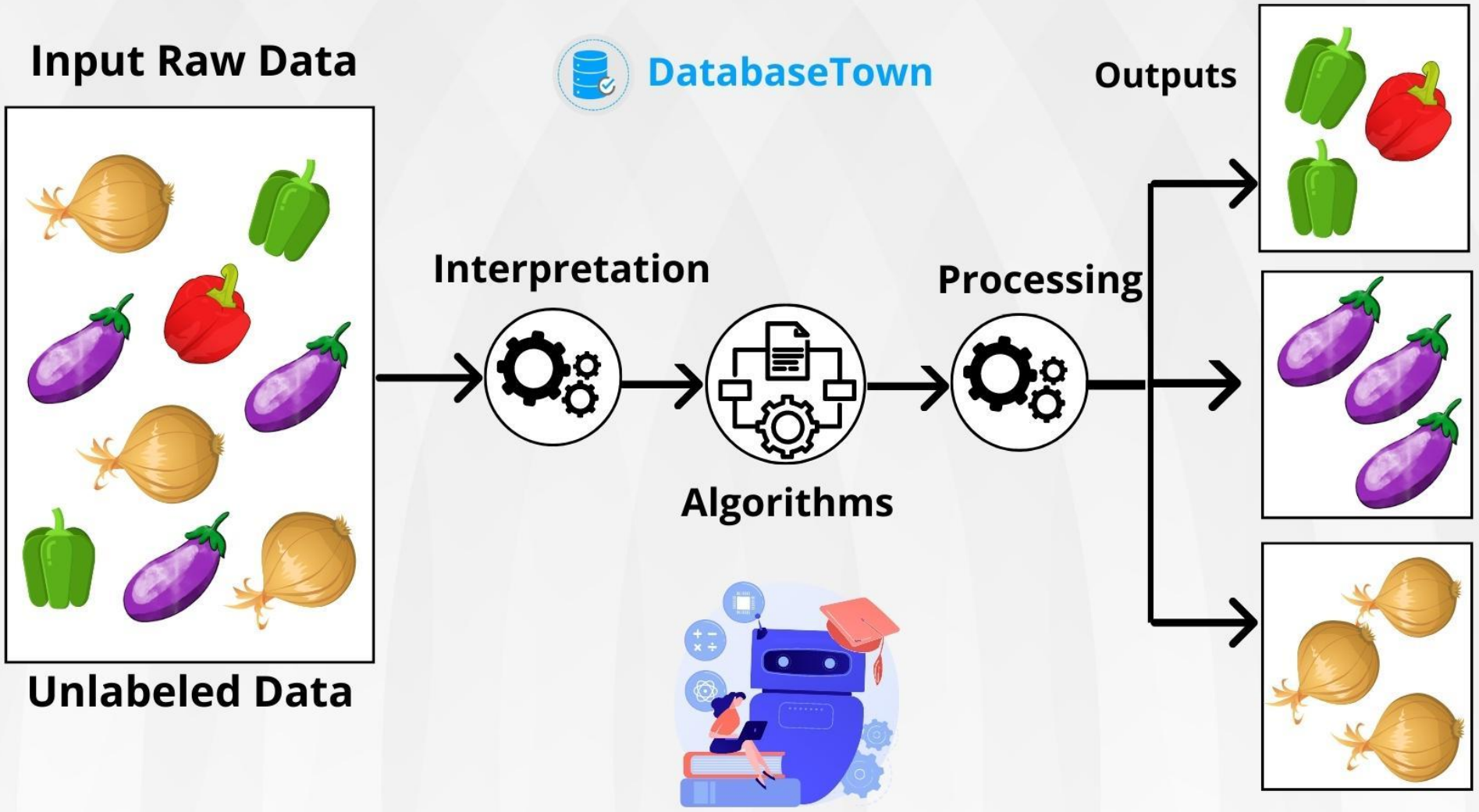
SUPERVISED LEARNING

Supervised machine learning is a branch of artificial intelligence that focuses on training models to make predictions or decisions based on labeled training data.



UNSUPERVISED LEARNING

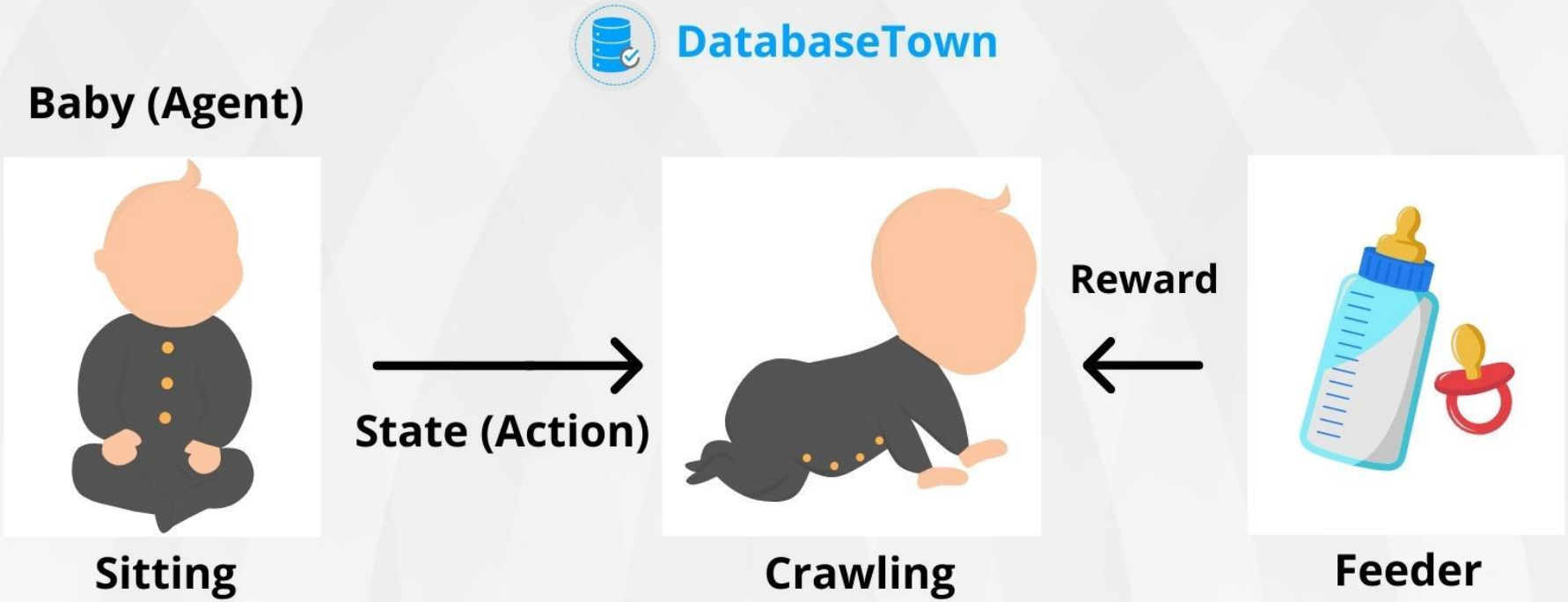
Unsupervised learning is a type of machine learning where the algorithm learns from unlabeled data without any predefined outputs or target variables.



REINFORCEMENT LEARNING

Reinforcement learning is a machine learning paradigm that focuses on how agents learn to interact with an environment to maximize cumulative rewards.

DatabaseTown



Algorithms and Approaches in Reinforcement Learning

- Q-learning
- Deep Q-networks (DQN)
- Policy Gradients Methods
- Proximal Policy Optimization (PPO)

Unsupervised Learning for Anomalies

- Finding Patterns Without Labels:
- We don't need a pre-labeled 'normal' or 'abnormal' dataset.
- The model finds data points that don't fit the dominant pattern or cluster.
- Ideal for scenarios where anomalies are rare and undefined.

Introducing `scikit-learn`

- The Standard for Machine Learning in Python
- A powerful, easy-to-use Python library for machine learning.
- Contains a wide range of models for classification, regression, clustering, and more.
- We'll use it to apply an anomaly detection model.

Introducing `Isolation Forest`

- A Simple, Powerful Anomaly Detector:
- Concept: Anomalies are 'isolated' more easily (in fewer steps) than normal points.
- Benefit: Very fast and effective for high-dimensional data.
- Intuitive: Outliers are 'different' and thus easier to separate.

Preparing Data for `Isolation Forest`

- What to Feed the Model:
- Isolation Forest works best with **numerical data**.
- We need to provide the model with the numerical columns it should analyze.
- For our Osram data, we'll use Temp_1, Temp_2, and Temp_3.

The Code in Action: Applying `Isolation Forest`

```
from sklearn.ensemble import IsolationForest

# Select the features for anomaly detection

features = df[['Temp_1', 'Temp_2', 'Temp_3']]

# Initialize the model (random_state for reproducibility)

model = IsolationForest(random_state=42)

# Fit the model to our data (the model learns 'normal' patterns)

model.fit(features)

# Predict anomalies: -1 for outlier, 1 for normal

df['anomaly_label_IF'] = model.predict(features)

# Get a decision score (lower score = more anomalous)

df['anomaly_score_IF'] = model.decision_function(features)
```

Hands-On: Isolation Forest

1. Fit an IsolationForest model to your Osram temperature data (Temp_1, Temp_2, Temp_3).
2. Add a new column (anomaly_label_IF) to your DataFrame showing the result (-1 or 1).
3. Count how many anomalies were found by the model.
4. What do you observe about the flagged points?

Visualizing the AI/ML Results

- Plotting the AI's Findings:
- We will create a plot (e.g., Temp_1 over time) and highlight the points the model flagged as anomalies with a different color.
- This helps us visually confirm the model's performance.

The Big Comparison: Traditional Stats vs. AI/ML

Two Lenses on the Same Data:

We have now found anomalies using two methods:

1. Z-score: Based on a single variable's distribution.

2. Isolation Forest: Based on the relationships between all three variables.

Which method is better? It depends on the problem!

Comparison: `z-score` vs `Isolation Forest`

	z-score	Isolation Forest
Type	The 'Rule-Based' Analyst	The 'Pattern-Based' Analyst
Pros	Simple, fast, easy to understand and interpret. No training data needed.	Finds subtle, multi-variable anomalies. Learns what 'normal' is automatically. More robust to complex data distributions.
Cons	Misses complex, multi-variable anomalies. Requires a pre-defined threshold. Sensitive to data distribution.	More of a 'black box'. Can be harder to explain the 'why' of a specific anomaly. Requires sufficient 'normal' data to learn from.

Group Discussion: When to Use Which?

Scenario 1: A simple, high/low temperature alarm. Which method is better?

Scenario 2: A machine failure where the Temp_Delta and Temp_1 subtly change simultaneously. Which method is better?

How do these methods complement each other in a real manufacturing environment?

Module 6 Recap

We moved from rule-based to pattern-based anomaly detection.

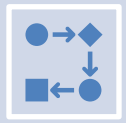
We applied an unsupervised model to find anomalies.

We critically compared the results with conventional statistical methods.

The Final Step: Reporting

Turning Analysis into Action:

Analysis is useless if it stays in a notebook.



How can we automate the process of sharing our insights?

We'll use pandas to export our findings to Excel/CSV.

Exporting Reports to CSV

- The to_csv() Method:
- df.to_csv("anomaly_report.csv", index=True)

Exporting Reports to Excel

- The to_excel() Method:
- df.to_excel("anomaly_report.xlsx", index=True)

Hands-On: Automating Your Report

1. Export your current DataFrame (including all anomaly labels) to an Excel file named my_osram_analysis.xlsx.
2. Open the generated Excel file to verify its contents.
3. Imagine how this could automate your daily/weekly reports.

The Final Mini-Project

- Applying the Full Workflow:
- Get into groups of 2-3.
- You'll be given a slightly different, but similar, dataset.
- Your Goal: Apply everything you've learned to:
- 1. Clean the data.
- 2. Find anomalies using both z-score and Isolation Forest.
- 3. Plot the results.
- 4. Automate an Excel report summarizing your findings.

Format: Group Presentation: Findings

1. What problem did you address?
2. Key insights from EDA.
3. Anomalies found by z-score vs. Isolation Forest.
4. Which method was more effective for your problem and why?
5. How would this automate a report?

Final Q&A: Ask Me Anything

Resources for Continued Learning

pandas Documentation: pandas.pydata.org

scikit-learn Documentation: scikit-learn.org

Python Cheat Sheets: Search 'Python pandas cheat sheet' for quick references.

Post-Assessment Quiz

Feedback Form

CHEAT SHEETS

Data Wrangling

with pandas Cheat Sheet
http://pandas.pydata.org

Pandas API Reference Pandas User Guide

Tidy Data – A foundation for wrangling in pandas

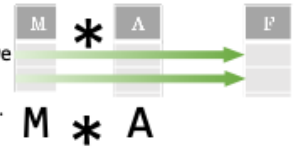
In a tidy data set:

&

Each **variable** is saved in its own **column**

Each **observation** is saved in its own **row**

Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4, 5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```
Specify values for each column.

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```
Specify values for each row.

		a	b	c
N	v			
D	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d', 1), ('d', 2),
         ('e', 2)], names=['n', 'v']))
```
Create DataFrame with a MultiIndex

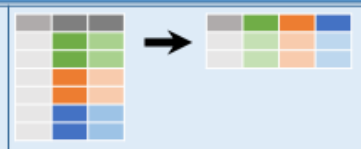
Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

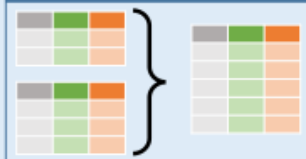
```
df = (pd.melt(df)
        .rename(columns={
                'variable':'var',
                'value':'val'})
        .query('val >= 200')
    )
```

Reshaping Data – Change layout, sorting, reindexing, renaming

pd.melt(df)
Gather columns into rows.



df.pivot(columns='var', values='val')
Spread rows into columns.



pd.concat([df1,df2])
Append rows of DataFrames

pd.concat([df1,df2], axis=1)
Append columns of DataFrames

df.sort_values('mpg')
Order rows by values of a column (low to high).

df.sort_values('mpg', ascending=False)
Order rows by values of a column (high to low).

df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame

df.sort_index()
Sort the index of a DataFrame

df.reset_index()
Reset index of DataFrame to row numbers, moving index to columns.

df.drop(columns=['Length', 'Height'])
Drop columns from DataFrame

Subset Observations - rows

```
df[df.Length > 7]
```
Extract rows that meet logical criteria.

df.drop_duplicates()
Remove duplicate rows (only considers columns).

df.sample(frac=0.5)
Randomly select fraction of rows.

df.sample(n=10) Randomly select n rows.

df.nlargest(n, 'value')
Select and order top n entries.

df.nsmallest(n, 'value')
Select and order bottom n entries.

df.head(n)
Select first n rows.

df.tail(n)
Select last n rows.

Subset Variables - columns

```
df[['width', 'length', 'species']]
```
Select multiple columns with specific names.

df['width'] *or* **df.width**
Select single column with specific name.

df.filter(regex='*regex*')
Select columns whose name matches regular expression *regex*.

Using query

query() allows Boolean expressions for filtering rows.

```
df.query('Length > 7')
df.query('Length > 7 and Width < 8')
df.query('Name.str.startswith("abc")',
         engine="python")
```

Subsets - rows and columns

Use **df.loc[]** and **df.iloc[]** to select only rows, only columns or both.
Use **df.at[]** and **df.iat[]** to access a single value by row and column.
First index selects rows, second index columns.

df.iloc[10:20]
Select rows 10-20.

df.iloc[:, [1, 2, 5]]
Select columns in positions 1, 2 and 5 (first column is 0).

df.loc[:, 'x2':'x4']
Select all columns between x2 and x4 (inclusive).

df.loc[df['a'] > 10, ['a', 'c']]
Select rows meeting logical condition, and only the specific columns .

df.iat[1, 2] Access single value by index

df.at[4, 'A'] Access single value by label

Logic in Python (and pandas)

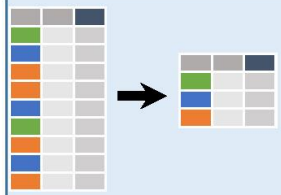
<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

regex (Regular Expressions) Examples

'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species$).*'	Matches strings except the string 'Species'

Cheatsheet for pandas (http://pandas.pydata.org/ originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Group Data



`df.groupby(by="col")`
Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")`
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

`size()`
Size of each group.

`agg(function)`
Aggregate group using function.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)`
Copy with values shifted by 1.

`rank(method='dense')`
Ranks with no gaps.

`rank(method='min')`
Ranks. Ties get min rank.

`rank(pct=True)`
Ranks rescaled to interval [0, 1].

`rank(method='first')`
Ranks. Ties go to first value.

`shift(-1)`
Copy with values lagged by 1.

`cumsum()`
Cumulative sum.

`cummax()`
Cumulative max.

`cummin()`
Cumulative min.

`cumprod()`
Cumulative product.

Summarize Data

`df['w'].value_counts()`
Count number of rows with each unique value of variable

`len(df)`
# of rows in DataFrame.

`df.shape`
Tuple of # of rows, # of columns in DataFrame.

`df['w'].nunique()`
# of distinct values in a column.

`df.describe()`
Basic descriptive and statistics for each column (or GroupBy).

`df.info()`
Prints a concise summary of the DataFrame.

`df.memory_usage()`
Prints the memory usage of each column in the DataFrame.

`df.dtypes()`
Prints a Series with the dtype of each column in the DataFrame.

pandas provides a large set of summary functions that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(function)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

`std()`
Standard deviation of each object.

Handling Missing Data

`df.dropna()`
Drop rows with any column having NA/null data.

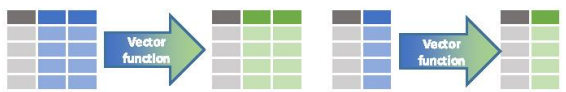
`df.fillna(value)`
Replace all NA/null data with value.

Make New Columns

`df.assign(Area=lambda df: df.Length*df.Height)`
Compute and append one or more new columns.

`df['Volume'] = df.Length*df.Height*df.Depth`
Add single column.

`pd.qcut(df.col, n, labels=False)`
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

`max(axis=1)`
Element-wise max.

`clip(lower=-10,upper=10)`
Trim values at input thresholds

`min(axis=1)`
Element-wise min.

`abs()`
Absolute value.

Windows

`df.expanding()`
Return an Expanding object allowing summary functions to be applied cumulatively.

`df.rolling(n)`
Return a Rolling object allowing summary functions to be applied to windows of length n.

Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

`pd.merge(adf, bdf, how='left', on='x1')`
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

`pd.merge(adf, bdf, how='right', on='x1')`
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

`pd.merge(adf, bdf, how='inner', on='x1')`
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

`pd.merge(adf, bdf, how='outer', on='x1')`
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`adf[adf.x1.isin(bdf.x1)]`
All rows in adf that have a match in bdf.

x1	x2
C	3

`adf[~adf.x1.isin(bdf.x1)]`
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

`pd.merge(ydf, zdf)`
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

`pd.merge(ydf, zdf, how='outer')`
Rows that appear in either or both ydf and zdf (Union).

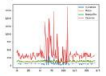
x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(columns=['_merge'])
```
Rows that appear in ydf but not zdf (Setdiff).

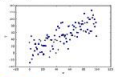
Cheatsheet for pandas (http://pandas.pydata.org/) originally written by Irv Lustig, Princeton Consultants, inspired by Rstudio Data Wrangling Cheatsheet

Plotting

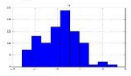
`df.plot()`
Plot a line graph for the DataFrame.



`df.plot.scatter(x='w', y='h')`
Plot a scatter graph of the DataFrame.



`df.plot.hist()`
Plot a histogram of the DataFrame.

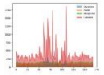


`df.plot.pie()`
Plot a pie chart of the DataFrame.

`df.plot.bar()`
Plot a line graph for the DataFrame.

`df.plot.boxplot()`
Plot a scatter graph of the DataFrame.

`df.plot.area()`
Plot an area graph of the DataFrame.



`df.plot.hexbin()`
Plot a hexbin graph of the DataFrame.

`df.plot(subplots=True)`
Separate into different graphs for each column in the DataFrame.

`df.plot(title="Graph of A against B")`
Sets the title of the graph.

`df.plot(cumulative=True)`
Creates a cumulative plot

`df.plot(bins=30)`
Set the number of bins into which data is grouped (histograms)

`df.plot(stacked=True)`
Stacks the data for the columns on top of each other. (bar, barh and area only)

`df.plot(alpha=0.5)`
Sets the transparency of the plot to 50%.

`df.plot(subplots=True, title=['col1', 'col2', 'col3'])`
Arguments can be combined for more flexibility when graphing, this would plot a separate line graph for of column of a 3-columned DataFrame. The first string in the list of titles applies to the graph of the left-most column.

Changing Type

`pd.to_numeric(data)`
Convert non-numeric types to numeric.

`pd.to_datetime(data)`
Convert non-datetime types to datetime type

`pd.to_timedelta(data)`
Convert non-timedelta types to timedelta

`df.astype(type)`
Convert data to (almost) any given type including categorical

`df.infer_objects()`
Attempts to infer a better type for object type data.

`df.convert_dtypes()`
Convert columns to best possible dtypes

Datetime

With a Series containing data of type datetime, the dt accessor is used to get various components of the datetime values:

`s.dt.year`
Extract the year

`s.dt.month`
Extract the month as an integer.

`s.dt.day`
Extract the day (int) from the date.

`s.dt.quarter`
Find which quarter the date lies in.

`s.dt.hour`
Extract the hour.

`s.dt.minute`
Extract the minute.

`s.dt.second`
Extract the second.

Mapping

Apply a mapping to every element in a DataFrame or Series, useful for recategorizing or transforming data.

`s.map(lambda x: 2*x)`
Returns a copy of the series where every entry is doubled

`df.apply(lambda s: s.max() - s.min(), axis=1)`
Returns a Series with the difference of the maximum and minimum values of each row of the DataFrame

Series String Operations

Similar to python string operations, except these are vectorized to apply to the entire Series efficiently.

`s.str.count(pattern)`
Returns a series with the integer counts in each element.

`s.str.get(index)`
Returns a series with the data at the given index for each element.

`s.str.join(sep)`
Returns a series where each element has been concatenated.

`s.str.title()`
Converts the first character of each word to be a capital.

`s.str.len()`
Returns a series with the lengths of each element.

`s.str.cat()`
Concatenate elements into a single string

`s.str.partition(sep)`
Splits the string on the first instance of the separator

`s.str.slice(start, stop, step)`
Slices each string

`s.str.replace(pat, rep)`
Use regex to replace patterns in each string.

`s.str.isalnum()`
Checks whether each element is alpha-numeric

Input/Output

Common file types for data input include CSV, JSON, HTML which are human-readable, while the common output types are usually more optimized for performance and scalability such as feather, parquet and HDF.

`df = pd.read_csv(filepath)`
Read data from csv file

`df = pd.read_html(filepath)`
Read data from html file

`df = pd.read_excel(filepath)`
Read data from xls (and related) files

`df = pd.read_sql(filepath)`
Read data from sql file

`pd.read_clipboard()`
Read text from clipboard

`df.to_parquet(filepath)`
Write data to parquet file

`df.to_feather(filepath)`
Write data to feather file

`df.to_hdf(filepath)`
Write data to HDF file

`df.to_clipboard()`
Copy object to the system clipboard

Frequently Used Options

Pandas offers some 'options' to globally control how Pandas behaves, display etc. Options can be queried and set via: `pd.options.option_name` (where *option_name* is the name of an option). For example:

`pd.options.display.max_rows = 20`
Set the `display.max_rows` option to 20.

Functions

`get_option(option)`
Fetch the value of the given option.

`set_option(option)`
Set the value of the given option.

`reset_option(options)`
Reset the values of all given options to default settings.

`describe_option(options)`
Print descriptions of given options.

`option_context(options)`
Execute code with temporary option settings that revert to prior settings after execution.

Display options

`display.max_rows`
The maximum number of rows displayed in pretty-print.

`display.max_columns`
The maximum number of columns displayed in pretty-print.

`display.expand_frame_repr`
Controls whether the DataFrame representation stretches across pages.

`display.large_repr`
Controls whether a DataFrame that exceeds maximum rows/columns is truncated or summarized

`display.precision`
The output display precision in decimal places.

`display.max_colwidth`
The maximum width of columns, longer cells will be truncated.

`display.max_info_columns`
The maximum number of columns displayed after calling `info()`.

`display.chop_threshold`
Sets the rounding threshold to zero when displaying a Series/DataFrame.

`display.colheader_justify`
Controls how column headers are justified.

Importing Data

Syntax for	How to use	Explained
IMPORT	`import numpy as np`	Imports NumPy using its standard alias, `np`
LOADTXT	`np.loadtxt('file.txt')`	Create an array from a `.txt` file
GENFROMTXT	`np.genfromtxt('file.csv', delimiter=',')`	Create an array from a `.csv` file
SAVETXT	`np.savetxt('file.txt', arr, delimiter=' ')`	Writes an array to a `.txt` file
	`np.savetxt('file.csv', arr, delimiter=',')`	Writes an array to a `.csv` file

Creating Arrays

Syntax for	How to use	Explained
ARRAY	`arr = np.array([1, 2, 3])`	Create a 1D array
	`arr = np.array([(1, 2, 3), (4, 5, 6)])`	Create a 2D array
ZEROS	`arr = np.zeros(3)`	1D array of length 3; all values set to 0
ONES	`arr = np.ones((3, 4))`	3x4 array with all values set to 1
EYE	`arr = np.eye(5)`	5x5 array of 0 with 1 on diagonal (identity matrix)

Syntax for	How to use	Explained
LINSPACE	`arr = np.linspace(0, 100, 6)`	Array of 6 evenly divided values from 0 to 100 (`[0, 20, 40, 60, 80, 100]`)
ARANGE	`arr = np.arange(0, 10, 3)`	Array of values from 0 to less than 10 with step 3 (`[0, 3, 6, 9]`)
FULL	`arr = np.full((2, 3), 8)`	2x3 array with all values set to 8
RAND	`arr = np.random.rand(4, 5)`	4x5 array of random floats between 0 and 1
	`arr = np.random.rand(6, 7) * 100`	6x7 array of random floats between 0-100
RANDINT	`arr = np.random.randint(5, size=(2, 3))`	2x3 array with random integers between 0 and 4

Inspecting Properties

Syntax for	How to use	Explained
ASTYPE	`arr.astype(dtype)`	Convert `arr` elements to type `dtype`
TOLIST	`arr.tolist()`	Convert `arr` to a Python list
INFO	`np.info(np.eye)`	View documentation for `np.eye`
SIZE	`arr.size`	Returns number of elements in `arr`
SHAPE	`arr.shape`	Returns dimensions of `arr` (rows, columns)
DTYPE	`arr.dtype`	Returns type of elements in `arr`

Copying, Sorting, & Reshaping

Syntax for	How to use	Explained
COPY	`np.copy(arr)`	Copies `arr` to new memory
VIEW	`arr.view(dtype)`	Creates view of `arr` elements with type `dtype`
SORT	`arr.sort()`	Sorts `arr`
SORT	`arr.sort(axis=0)`	Sorts specific axis of `arr`
FLATTEN	`two_d_arr.flatten()`	Flattens 2D array `two_d_arr` to 1D
T	`arr.T`	Transposes `arr` *(rows become columns and vice versa)*
RESHAPE	`arr.reshape(3, 4)`	Reshapes `arr` to 3 rows, 4 columns without changing data
RESIZE	`arr.resize((5, 6))`	Changes `arr` shape to 5x6 and fills new values with 0

Adding & Removing Elements

Syntax for	How to use	Explained
APPEND	`np.append(arr, values)`	Appends values to end of `arr`
INSERT	`np.insert(arr, 2, values)`	Inserts values into `arr` before index 2
DELETE	`np.delete(arr, 3, axis=0)`	Deletes row on index 3 of `arr`
	`np.delete(arr, 4, axis=1)`	Removes the 5th column from `arr`

Combining & Splitting

Syntax for	How to use	Explained
CONCATENATE	`np.concatenate((arr1, arr2), axis=0)`	Adds `arr2` as rows to the end of `arr1`
	`np.concatenate((arr1, arr2), axis=1)`	Adds `arr2` as columns to end of `arr1`
SPLIT	`np.split(arr, 3)`	Splits `arr` into 3 sub-arrays
HSPLIT	`np.hsplit(arr, 5)`	Splits `arr` horizontally on the index 5

Indexing & Slicing

Syntax for	How to use	Explained
INDEXING	`arr[5]`	Returns the element at index 5
	`arr[2, 5]`	Returns the 2D array element on index [2][5]
	`arr[1] = 4`	Assigns array element on index 1 the value 4
	`arr[1, 3] = 10`	Assigns array element on index [1][3] the value 10
SLICING	`arr[0:3]`	Returns the elements at indices 0, 1, 2
	`arr[0:3, 4]`	Returns the elements on rows 0, 1, 2 in column index 4
	`arr[:2]`	Returns the elements at indices 0, 1
	`arr[:, 1]`	Returns column index 1, all rows
CONDITIONAL STATEMENTS	`arr < 5`	Returns an array of boolean values

Indexing & Slicing

Syntax for	How to use	Explained
CONDITIONAL STATEMENTS	`(arr1 < 3) & (arr2 > 5)`	To be `True`, both must be `True`
	`~arr`	Inverts a boolean array
	`arr[arr < 5]`	Returns array elements less than 5
	`(arr1 < 3) | (arr2 > 5)`	To be `True`, at least one must be `True`

Scalar Math

Syntax for	How to use	Explained
ADD	`np.add(arr, 1)`	Add 1 to each array element
SUBTRACT	`np.subtract(arr, 2)`	Subtract 2 from each array element
MULTIPLY	`np.multiply(arr, 3)`	Multiply each array element by 3
DIVIDE	`np.divide(arr, 4)`	Divide each array element by 4 (returns `np.nan` for division by zero)
POWER	`np.power(arr, 5)`	Raise each array element to the power of 5

Vector Math

Syntax for	How to use	Explained
ADD	`np.add(arr1, arr2)`	Elementwise add `arr1` to `arr2`
SUBTRACT	`np.subtract(arr1, arr2)`	Elementwise subtract `arr2` from `arr1`
MULTIPLY	`np.multiply(arr1, arr2)`	Elementwise multiply `arr1` by `arr2`
DIVIDE	`np.divide(arr1, arr2)`	Elementwise divide `arr1` by `arr2`
POWER	`np.power(arr1, arr2)`	Elementwise, raise `arr1` to the power of `arr2`
ARRAY_EQUAL	`np.array_equal(arr1, arr2)`	Returns `True` if the arrays have the same elements and shape
SQRT	`np.sqrt(arr)`	Square root of each element in the array
SIN	`np.sin(arr)`	Sine of each element in the array
LOG	`np.log(arr)`	Natural log of each element in the array
ABS	`np.abs(arr)`	Absolute value of each element in the array
CEIL	`np.ceil(arr)`	Rounds up each element to the nearest integer

Vector Math

Syntax for	How to use	Explained
FLOOR	`np.floor(arr)`	Rounds down each element to the nearest integer
ROUND	`np.round(arr)`	Rounds each element to the nearest integer

Statistics

Syntax for	How to use	Explained
MEAN	`np.mean(arr, axis=0)`	Returns mean of `arr` along specified axis
SUM	`arr.sum()`	Returns the sum of elements in `arr`
MIN	`arr.min()`	Returns minimum value of `arr`
MAX	`arr.max(axis=0)`	Returns maximum value of `arr` along specified axis
VAR	`np.var(arr)`	Returns the variance of `arr`
STD	`np.std(arr, axis=1)`	Returns the standard deviation of `arr` along specified axis
CORRCOEF	`arr.corrcoef()`	Returns correlation coefficient of `arr`

Working with Data

Syntax for	How to use	Explained

CREATING NDARRAYS	`import numpy as np`
	`array_1d = np.array([1, 2, 3, 4, 5])`
	`array_2d = np.array([[1, 2, 3], [4, 5, 6]])`
		Create a 1D or 2D `ndarray`

CONVERTING A LIST OF LISTS	`import csv`
	`f = open("nyc_taxis.csv", "r")`
	`taxi_list = list(csv.reader(f))`
	`taxi = np.array(taxi_list)`
		Convert a list of lists into a 2D `ndarray`

SELECTING ROWS	`second_row = taxi[1]`	Select the second row in `taxi`
	`all_but_first_row = taxi[1:]`	Select all rows from the second row onward in `taxi`
	`fifth_row_second_column = taxi[4, 1]`	Select the element from the fifth row and second column in `taxi`
SELECTING COLUMNS	`second_column = taxi[:, 1]`	Select all values from the second column in `taxi`

	`second_third_columns = taxi[:, 1:3]`
	`cols = [1, 3, 5]`
	`second_fourth_sixth_columns = taxi[:, cols]`
		Select the second and third columns, then the second, fourth, and sixth columns in `taxi`

	`twod_slice = taxi[1:4, :3]`	Select a slice of rows 2 to 4 and columns 1 to 3 in `taxi`

Working with Data

Syntax for	How to use	Explained
VECTOR OPERATIONS	`vector_a + vector_b`	Element-wise addition of two `ndarray` objects
	`vector_a - vector_b`	Element-wise subtraction of two `ndarray` objects
	`vector_a * vector_b`	Element-wise multiplication of two `ndarray` objects
	`vector_a / vector_b`	Element-wise division of two `ndarray` objects
STATISTICS FOR 1D NDARRAYS	`array_1d.min()`	Return the minimum value of `array_1d`
	`array_1d.max()`	Return the maximum value of `array_1d`
	`array_1d.mean()`	Calculate the average of values in `array_1d`
	`array_1d.sum()`	Calculate the sum of the values in `array_1d`
STATISTICS FOR 2D NDARRAYS	`array_2d.max()`	Return the maximum value for the entire `array_2d`
	`array_2d.max(axis=1)  # returns a 1D ndarray`	Return the maximum value in each row in `array_2d`
	`array_2d.max(axis=0)  # returns a 1D ndarray`	Return the maximum value in each column in `array_2d`

Syntax for	How to use	Explained

CREATING AN NDARRAY FROM CSV FILE	`import numpy as np`
	`taxi = np.genfromtxt('nyc_taxis.csv',`
	`delimiter=',', skip_header=1)`
		Load data from the `nyc_taxis.csv` file into an `ndarray`, skipping the header row

WORKING WITH BOOLEAN ARRAYS	`np.array([2, 4, 6, 8]) < 5`	Create a Boolean array for elements less than 5

	`a = np.array([2, 4, 6, 8])`
	`filter = a < 5`
	`a[filter]  # returns [2, 4]`
		Use Boolean filtering to return elements less than 5 from an `ndarray`

	`tip_amount = taxi[:, 12]`
	`tip_bool = tip_amount > 50`
	`top_tips = taxi[tip_bool, 5:14]`
		Use Boolean filtering to return rows with `tip_amount > 50` and columns 6 to 14

ASSIGNING NDARRAY VALUES	`taxi[1066, 5] = 1`
	`taxi[:, 0] = 16`
	`taxi[550:552, 7] = taxi[:, 7].mean()`
		Assign values to specific elements, a column, and a slice in `taxi`

	`taxi[taxi[:, 5] == 2, 15] = 1`	Use Boolean indexing to assign a value of 1 in column index 15 to rows where the 6th column equals 2

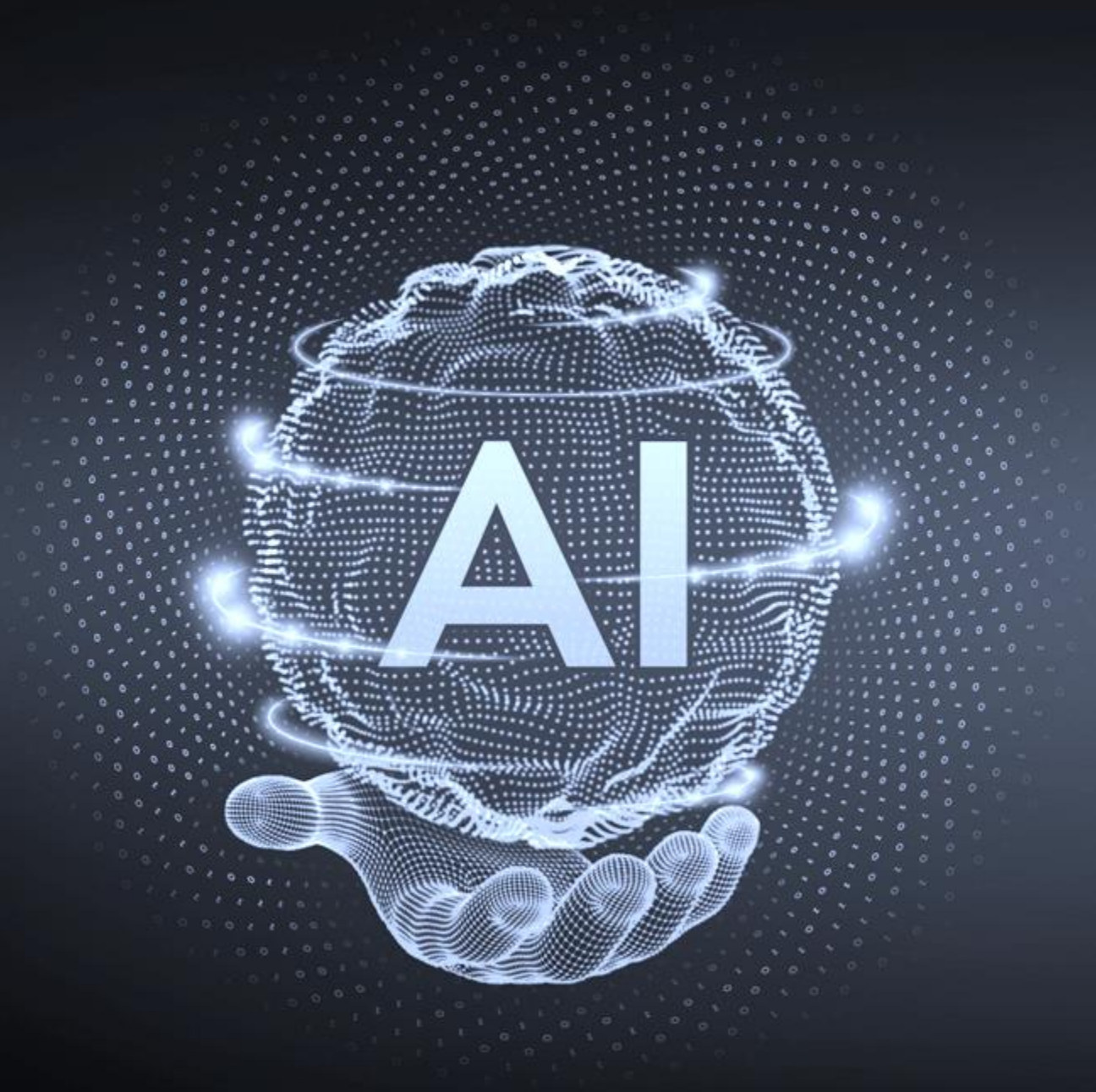
Other Courses

Beginners	Intermediate	Advanced
1. Intro to Generative AI & Prompt Engineering 2. Intro to Python 3. Data Driven Decisions 4. Intro to AI Monetization Intro to Data Analytics and Artificial Intelligence 5. Storytelling with Power BI 6. Fundamentals of AI 7. Data & AI: Awareness and Strategic Implementation 8. Cloud Computing Fundamentals 9. Python Programming 10. SQL for Professionals 11. Data Analysis with Python 12. Statistics & Probability for Data Science	1. Machine Learning in Credit Scoring 2. Power BI Intermediate 3. Excel Intermediate 4. Python for Deep Learning 5. Data Engineering 6. Data Analytics Storytelling 7. AI-Powered Strategies in Consumer Analytics 8. AI-Enhanced Customer Experience: The Future of Personalized Engagement 9. Time Series Forecasting: Business Applications 10. Natural Language Processing: From Text to Insights 11. Computer Vision Fundamentals: Turning Images into Insights 12. Machine Learning 101: Predictions and Insights 13. Python for Data Analytics Revolutionizing Fraud Detection and Prevention with AI	1. Building Generative AI 2. Advanced Excel 3. Data Science Advanced 4. Deep Learning & Computer Vision 5. Responsible AI in Action: Ethics, Safety & Accountability 6. Machine Learning 202: Advanced Models and Techniques 7. MLOps Mastery: From Models to Scalable Production

My Linkedin

Ng Keng Fai

GROWTH.PRO
DATA SCIENCE CONSULTING & TRAINING
BigData.Growth.Pro



Thank You