World Wide Technology

أدنوك ADNOC

ADNOC Accelerator Programme

Artificial Intelligence

Cohort 2

Introduction to Python for Data Science

Introduction to Python

LEARNING OBJECTIVES

1. **Understand the fundamentals of programming**
2. **Grasp basic principles of Python**
3. **Leverage conditional statements, loops and functions**

Programming allows you to talk to your computer

You can instruct your computer to execute certain

commands

Simple as addition

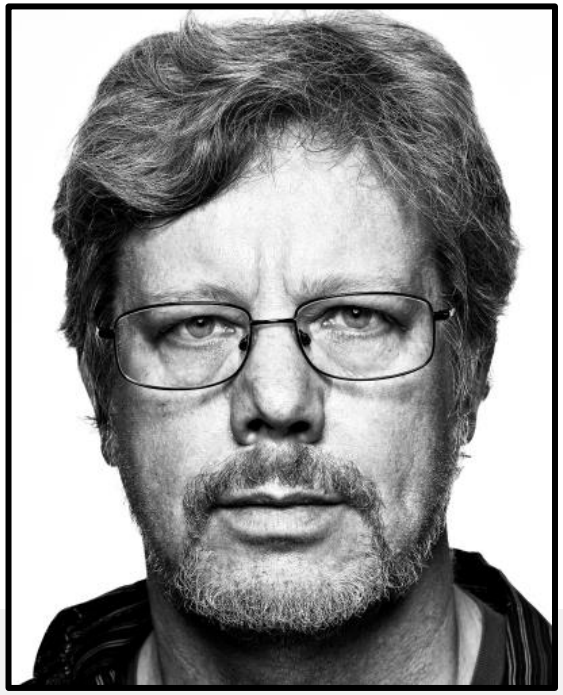
```
x = input("Type a number: ")
y = input("Type another number: ")
sum = int(x) + int(y)
print("The sum is: ", sum)...
```

Complex as website design

```
!DOCTYPE html>
<html lang="en">
<head>
<title>Page Title</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,
initial-scale=1">
<style>
body {
  font-family: Arial, Helvetica, sans-serif;
}
```

Some common programming languages

Python was designed to be both practical and powerful



The name **Python** doesn't come from the snake. It comes from a comedy group called Monty Python's Flying Circus.

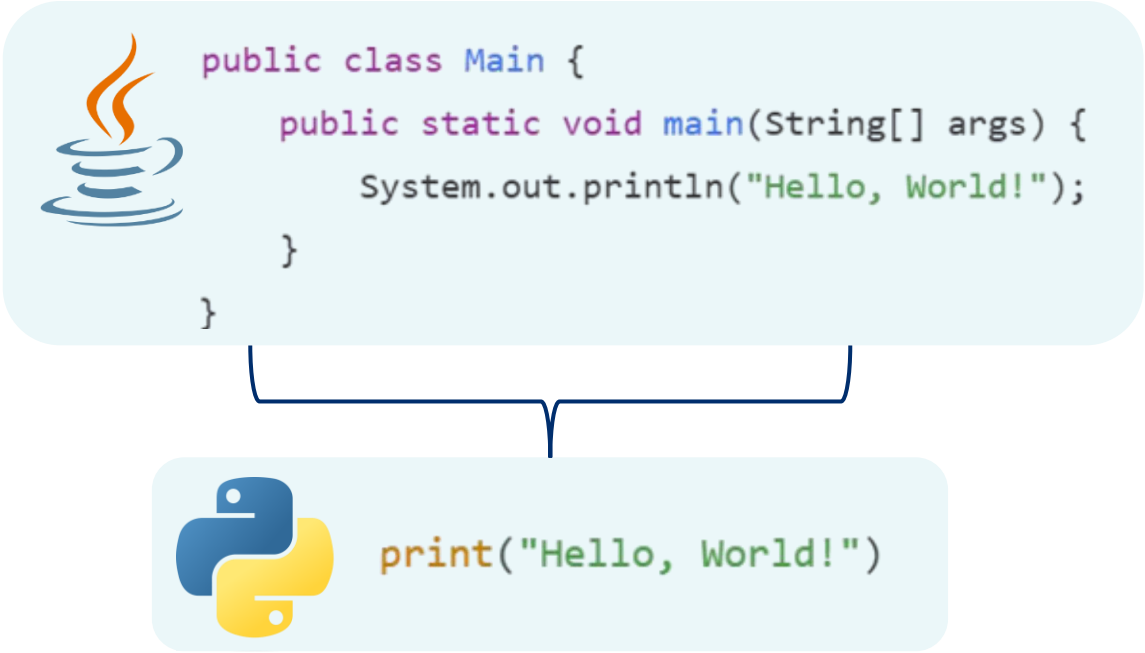
Guido Van Rossum released Python in 1991 with the vision of creating a language that was easy to read, easy to write, and powerful enough to be used in real world application

Python is go-to language across industries like **oil & gas, finance,** and **healthcare** due to its:

- ✅ Ease of use & productivity
- ✅ Powerful data science & AI support
- ✅ Seamless integration
- ✅ Scalability
- ✅ Automation & scripting
- ✅ Cross-platform & open-source

Python is simple, readable, and versatile

Python requires **fewer lines of code** than other languages, meaning faster and more efficient development



Simple Readable Syntax eliminates need for special characters

No need to worry about semi columns or braces

Code Blocks are defined by indentation

```
if True:
    print("This is indented correctly")
```

Indentation shows the hierarchy of the code, that is, this line belongs to the block of code

Dynamic Typing means you don’t have to specify data types explicitly

```
x = 10        # Integer
y = 3.14      # Float
name = "Oil & Gas Industry"  # String
is_running = True  # Boolean
```

Variables store value and track data that can change over time

A variable in Python functions much like those in math – it holds values that can change and be used in calculations

Variables can be of many types

Numbers: Integers, real numbers, and so on

```
[1, 2, 3, 4, 5]
```

Strings: Ordered sequence of characters

"Python is easy"

List: Ordered collection of objects

```
list is - [2, 3, 5, 12, 30, 40, 90]
```

What do variables do for you?

Keep track of the changing data in your programme

Basics of Python: Data Structures

Lists are used for storing ordered sequences of data

Command → Output

```
# List of daily oil production (in barrels)
oil_production = [3500, 4200, 3900, 4100, 3800]

# Displaying Original List
print("Original List:", oil_production)

# Accessing elements to show order is preserved
print("First day's production:", oil_production[0])
# Accessing first element
print("Last day's production:", oil_production[-1])
# Accessing last element
```

```
Original List: [3500, 4200, 3900, 4100, 3800]
First day's production: 3500
Last day's production: 3800
```

Mutability of lists allows you to modify elements even after printing

```
# Mutable: Lists allow modification of elements
oil_production[2] = 4000  # Changing value for Day 3

# Display updated list
print("Updated List:", oil_production)
```

```
Updated List: [3500, 4200, 4000, 4100, 3800]
```


Tuples are essentially fixed lists which cannot be modified

Command → Output

```
# Tuple storing wellhead location (Latitude, Longitude, Depth)
wellhead_location = (24.4667, 54.3667, 3000)  # Abu Dhabi coordinates + depth in meters

# Ordered: Tuples maintain the sequence in which elements are defined
print("Original Tuple:", wellhead_location)

# Accessing elements to show order
print(f"Wellhead Latitude: {wellhead_location[0]}")
```

```
Original Tuple: (24.4667, 54.3667, 3000)
Wellhead Latitude: 24.4667
```

Immutability of tuples does not allow modification

```
try:
    wellhead_location[2] = 3500  # Attempting to change depth
except TypeError as e:
    print("Error:", e)  # This will confirm immutability
```

```
Error: 'tuple' object does not support item assignment
```


Dictionaries store unique key values of data

Command → Output

```
# Dictionary storing oil production with duplicate keys
well_production = {
    "Well A": {"Oil": 4000, "Gas": 25000},
    "Well B": {"Oil": 3200, "Gas": 27000},
    "Well A": {"Oil": 4200, "Gas": 26000},  # Overwrites previous "Well A"
}

# Display dictionary to show unique keys behaviour
print("Dictionary with Unique Keys:", well_production)
```

```
{'Well A': {'Oil': 4200, 'Gas': 26000}, 'Well B': {'Oil': 3200, 'Gas': 27000}}
```

Python dictionaries allow fast lookup just like a real dictionary

```
# Fast Lookup: Retrieving oil production for "Well B" using O(1) complexity
print("Oil Production for Well B:", well_production["Well B"]["Oil"], "barrels")
```

```
Oil Production for Well B: 3200 barrels
```


Sets are unordered mutable collections of data

Command → Output

```
# Defining a set of well names
well_names = {"Well A", "Well B", "Well C", "Well D"}

# Printing the set multiple times to show unordered nature
print("Set of Well Names:", well_names)
```

```
Set of Well Names: {'Well A', 'Well C', 'Well B', 'Well D'}
```

Only **unique** values get stored within a set

```
# Defining a set with duplicate values
oil_fields = {"Field X", "Field Y", "Field Z", "Field X", "Field Y"}

# Printing the set
print("Unique Oil Fields:", oil_fields)
```

```
Unique Oil Fields: {'Field Y', 'Field X', 'Field Z'}
```

Lists are used for storing ordered sequences of data

Command

Output

```
# List of daily oil production (in barrels)
oil_production = [3500, 4200, 3900, 4100, 3800]

# Displaying Original List
print("Original List:", oil_production)

# Accessing elements to show order is preserved
print("First day's production:", oil_production[0])
# Accessing first element
print("Last day's production:", oil_production[-1])
# Accessing last element
```

```
Original List: [3500, 4200, 3900, 4100, 3800]
First day's production: 3500
Last day's production: 3800
```

Mutability of lists allows you to modify elements even after printing

```
# Mutable: Lists allow modification of elements
oil_production[2] = 4000  # Changing value for Day 3

# Display updated list
print("Updated List:", oil_production)
```

```
Updated List: [3500, 4200, 4000, 4100, 3800]
```

Tuples are essentially fixed lists which cannot be modified

Command

Output

```
# Tuple storing wellhead location (Latitude, Longitude, Depth)
wellhead_location = (24.4667, 54.3667, 3000)  # Abu Dhabi coordinates +
depth in meters


# Ordered: Tuples maintain the sequence in which elements are defined
print("Original Tuple:", wellhead_location)


# Accessing elements to show order
print(f"Wellhead Latitude: {wellhead_location[0]}")
```

```
Original Tuple: (24.4667, 54.3667, 3000)
Wellhead Latitude: 24.4667
```

Immutability of tuples does not allow modification

```
try:
    wellhead_location[2] = 3500  # Attempting to change depth
except TypeError as e:
    print("Error:", e)  # This will confirm immutability
```

```
Error: 'tuple' object does not support item assignment
```

Dictionaries store unique key values of data

Command

Output

```
# Dictionary storing oil production with duplicate keys
well_production = {
    "Well A": {"Oil": 4000, "Gas": 25000},
    "Well B": {"Oil": 3200, "Gas": 27000},
    "Well A": {"Oil": 4200, "Gas": 26000},  # Overwrites previous "Well A"
}


# Display dictionary to show unique keys behaviour
print("Dictionary with Unique Keys:", well_production)
```

```
{'Well A': {'Oil': 4200, 'Gas': 26000}, 'Well B': {'Oil': 3200, 'Gas': 27000}}
```

Python dictionaries allow fast lookup just like a real dictionary

```
# Fast Lookup: Retrieving oil production for "Well B" using O(1) complexity
print("Oil Production for Well B:", well_production["Well B"]["Oil"],
"barrels")
```

```
Oil Production for Well B: 3200 barrels
```

Sets are unordered mutable collections of data

Command

Output

```
# Defining a set of well names
well_names = {"Well A", "Well B", "Well C", "Well D"}


# Printing the set multiple times to show unordered nature
print("Set of Well Names:", well_names)
```

```
Set of Well Names: {'Well A', 'Well C', 'Well B', 'Well D'}
```

Only **unique** values get stored within a set

```
# Defining a set with duplicate values
oil_fields = {"Field X", "Field Y", "Field Z", "Field X", "Field Y"}


# Printing the set
print("Unique Oil Fields:", oil_fields)
```

```
Unique Oil Fields: {'Field Y', 'Field X', 'Field Z'}
```

Which of these do not allow modification?

A. Lists

B. Tuples

C. Sets

Which of these do not allow modification?

A. Lists

B. Tuples

C. Sets

Decision making can also be simplified through Python

Let’s say you want to keep track oil production in different oil rigs

How would you do that?

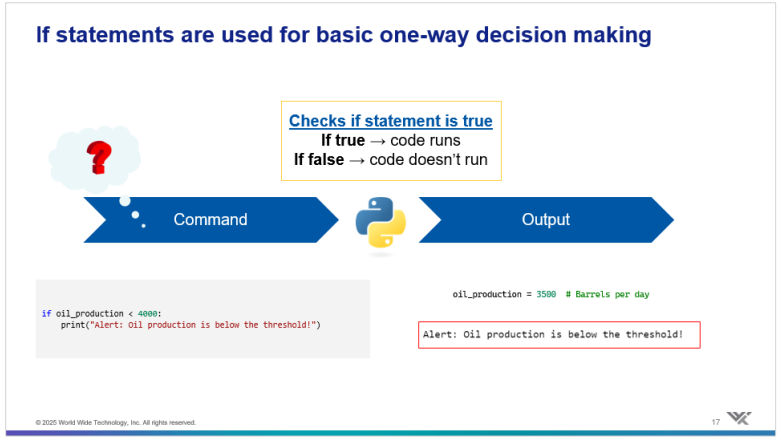
Decision making can also be simplified through Python

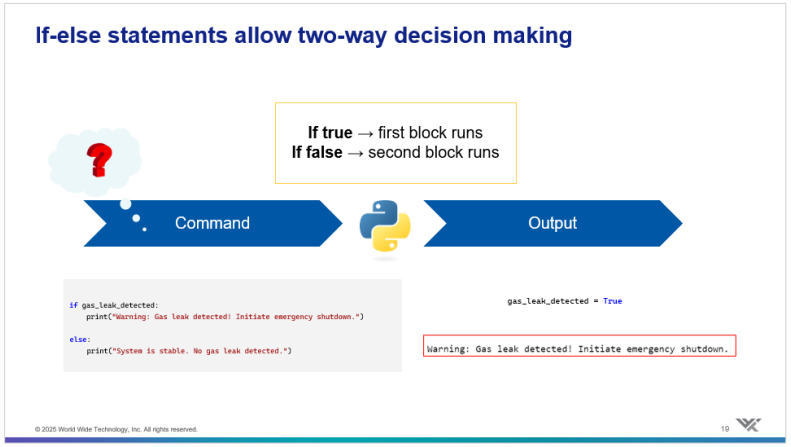
Let’s say you want to keep track oil production in different oil rigs

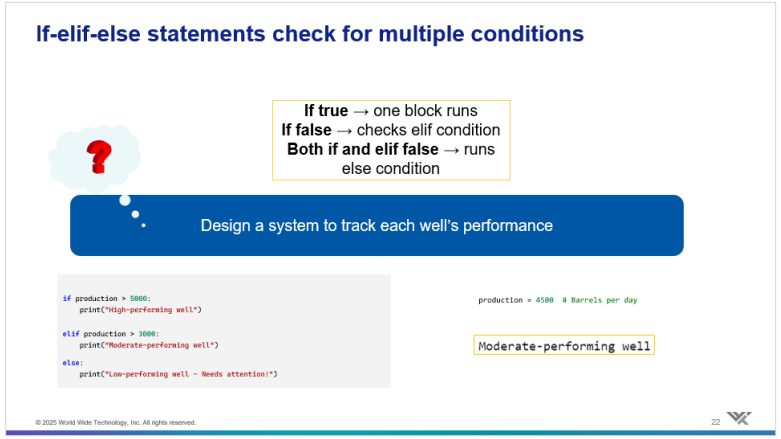
One way to do this is through conditional statements

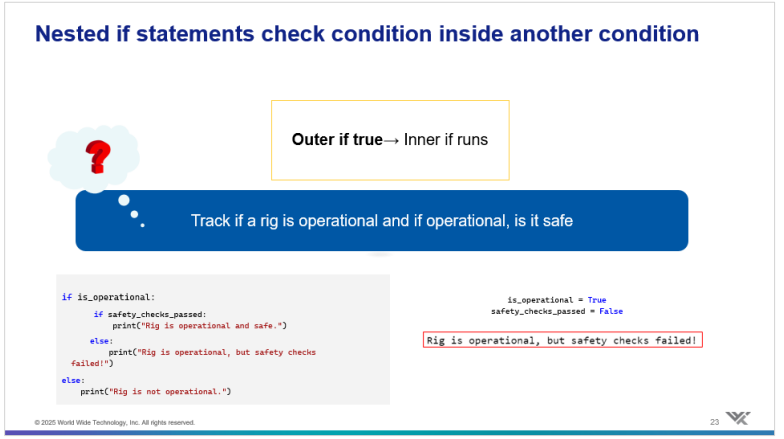
Basics of Python: Conditional Statements

Conditional statements allow decision-making by controlling the order of following commands









If statements are used for basic one-way decision making

Checks if statement is true

If true → code runs

If false → code doesn't run

Command → Output

```
if oil_production < 4000:
    print("Alert: Oil production is below the threshold!")
```

```
oil_production = 3500  # Barrels per day
```

```
Alert: Oil production is below the threshold!
```

If statements are used for basic one-way decision making

Checks if statement is true

If true → code runs

If false → code doesn't run

- Create an alert for when oil production falls below 4000 barrels

```
if oil_production < 4000:
    print("Alert: Oil production is below the threshold!")
```

```
oil_production = 3500  # Barrels per day
```

```
Alert: Oil production is below the threshold!
```

If-else statements allow two-way decision making

If true → first block runs
If false → second block runs

Command → Output

```
if gas_leak_detected:
    print("Warning: Gas leak detected! Initiate emergency shutdown.")

else:
    print("System is stable. No gas leak detected.")
```

```
gas_leak_detected = True
```

```
Warning: Gas leak detected! Initiate emergency shutdown.
```

If-else statements allow two-way decision making

If true → first block runs
If false → second block runs

Send a warning if gas leak is detected

```
if gas_leak_detected:
    print("Warning: Gas leak detected! Initiate emergency shutdown.")

else:
    print("System is stable. No gas leak detected.")
```

```
gas_leak_detected = True
```

```
Warning: Gas leak detected! Initiate emergency shutdown.
```

If-elif-else statements check for multiple conditions

If true → one block runs
If false → checks elif condition
Both if and elif false → runs else condition

Command

Output

```
if production > 5000:
    print("High-performing well")

elif production > 3000:
    print("Moderate-performing well")

else:
    print("Low-performing well - Needs attention!")
```

```
production = 4500  # Barrels per day
```

```
Moderate-performing well
```

If-elif-else statements check for multiple conditions

If true → one block runs
If false → checks elif condition
Both if and elif false → runs else condition

Design a system to track each well's performance

```
if production > 5000:
    print("High-performing well")

elif production > 3000:
    print("Moderate-performing well")

else:
    print("Low-performing well - Needs attention!")
```

```
production = 4500  # Barrels per day
```

```
Moderate-performing well
```

Nested if statements check condition inside another condition

Outer if true→ Inner if runs

Command

Output

```
if is_operational:
       if safety_checks_passed:
           print("Rig is operational and safe.")
       else:
           print("Rig is operational, but safety checks
  failed!")
else:
    print("Rig is not operational.")
```

```
is_operational = True
safety_checks_passed = False
```

```
Rig is operational, but safety checks failed!
```

Nested if statements check condition inside another condition

Outer if true→ Inner if runs

Track if a rig is operational and if operational, is it safe

```
if is_operational:
       if safety_checks_passed:
           print("Rig is operational and safe.")
       else:
           print("Rig is operational, but safety checks
  failed!")
else:
    print("Rig is not operational.")
```

```
is_operational = True
safety_checks_passed = False
```

```
Rig is operational, but safety checks failed!
```

Loops make your code shorter, smarter, and faster

Loops automate repetition so you don’t have to write the same code multiple times

Before

```
# Daily production data for multiple wells
wells = {"A": 1000, "B": 1200, "C": 1100, "D":
950}

# Calculating production after a week (7 days)
well_A_weekly = wells["A"] * 7
well_B_weekly = wells["B"] * 7
well_C_weekly = wells["C"] * 7
well_D_weekly = wells["D"] * 7

print("Weekly Production:")
print("Well A:", well_A_weekly, "barrels")
print("Well B:", well_B_weekly, "barrels")
print("Well C:", well_C_weekly, "barrels")
print("Well D:", well_D_weekly, "barrels")
```

```
Weekly Production:
Well A: 7000 barrels
Well B: 8400 barrels
Well C: 7700 barrels
Well D: 6650 barrels
```

Loops make your code shorter, smarter, and faster

Loops automate repetition so you don’t have to write the same code multiple times

After

```
for well, daily_production in wells.items():
    weekly_production = daily_production * 7
    print(f"Well {well}: {weekly_production} barrels")
```

```
Well A: 7000 barrels
Well B: 8400 barrels
Well C: 7700 barrels
Well D: 6650 barrels
```

Different loops are used based on iterations and actions

For loops repeat a sequence a fixed number of times

While loops keep repeating as long as a condition is true

Break statements are used to exit a loop completely

Continue statements skip current iteration but continue the loop

Functions make it easier to do tasks repeatedly

When you want to do some task repeatedly in Python, you can create a function that can complete the task without being given instructions again and again

Before

```
# Daily production and operational hours for multiple wells
wells = {
    "A": {"production": 1000, "hours": 20},
    "B": {"production": 1200, "hours": 12},
    "C": {"production": 1100, "hours": 40},
}
# Efficiency = Oil Production / Operational Hours
eff_A = wells["A"]["production"] / wells["A"]["hours"]
eff_B = wells["B"]["production"] / wells["B"]["hours"]
eff_C = wells["C"]["production"] / wells["C"]["hours"]

print("Production Efficiency:")
print(f"Well A: {eff_A} barrels/hour")
print(f"Well B: {eff_B} barrels/hour")
print(f"Well C: {eff_C} barrels/hour")
```

```
Production Efficiency:
Well A: 50.0 barrels/hour
Well B: 100.0 barrels/hour
Well C: 27.5 barrels/hour
```

Functions make it easier to do tasks repeatedly

When you want to do some task repeatedly in Python, you can create a function that can complete the task without being given instructions again and again

After

```
# Function to calculate production efficiency
def calculate_efficiency(production, hours):
    return production / hours
for well, data in wells.items():
    efficiency = calculate_efficiency(data["production"], data["hours"])
    print(f"Well {well}: {efficiency:.2f} barrels/hour")
```

```
Well A: 50.00 barrels/hour
Well B: 100.00 barrels/hour
Well C: 27.50 barrels/hour
```

Functions can be defined using keywords

A function must be defined by using the **def** keyword

Functions take inputs known as **parameters** (hours, flow rate)

```
def estimate_production(hours=24, flow_rate=200):
    return hours * flow_rate

print(f"Estimated Production: {estimate_production()} barrels")
print(f"Half-Day Production: {estimate_production(12)} barrels")
```

return keyword is used to get the results

To run a function, **call** it using its name followed by ()

Basics of Python

In this session, we covered:

- **Understanding fundamentals of programming**
- **Learning about variables and data structures in Python**
- **Using conditional statements for decision-making**
- **Using loops and functions to simplify coding**

ADNOC Accelerator Programme

Artificial Intelligence

Cohort 2

Python Libraries

Python Libraries

LEARNING OBJECTIVES

1. **Install Python Libraries**
2. **Utilise Pandas for data analysis**
3. **Perform data visualisation with Matplotlib**

A Python library helps perform tasks without starting from scratch

Imagine you had to build an oil refinery

Would you prefer to manufacture each part yourself (from pipelines to turbines) or use pre-built components wherever possible?

That’s right! Use pre-built components.

That would save time and money.

A Python library helps perform tasks without starting from scratch

Imagine you had to build an oil refinery

Libraries serve as toolboxes or ingredient kits for your task

Would you prefer to manufacture each part yourself (from pipelines to turbines) or use pre-built components wherever possible?

A python library contains pre-built functions that perform complicated tasks for you by utilising existing solutions

Python libraries can be managed through a simple process

Install | Verify | Check | Upgrade | List

pip (Python Package Installer) is the standard tool for installing libraries in Python

```
pip install <library_name>
```

```
pip install pandas
```

```
import pandas as pd
```

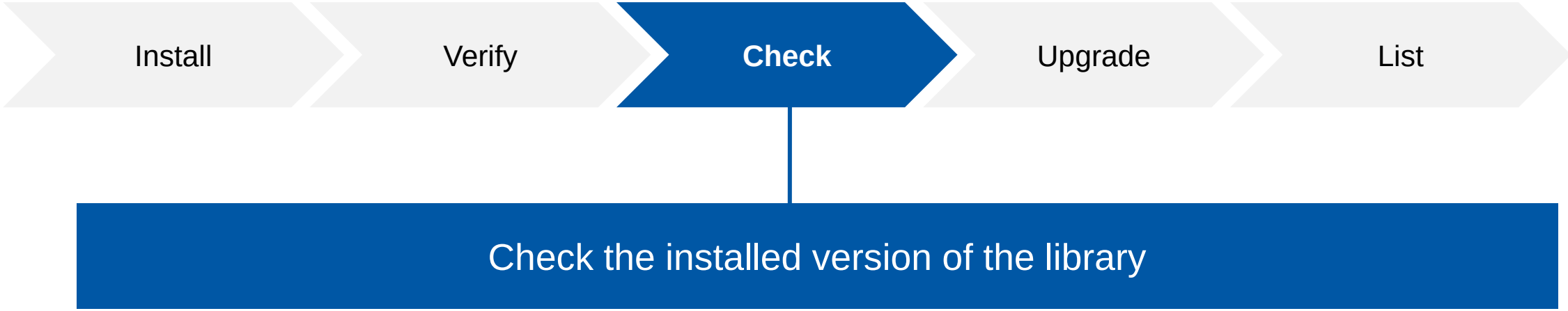
Python libraries can be managed through a simple process

Install | **Verify** | Check | Upgrade | List

After installation, verify if the libraries are installed correctly

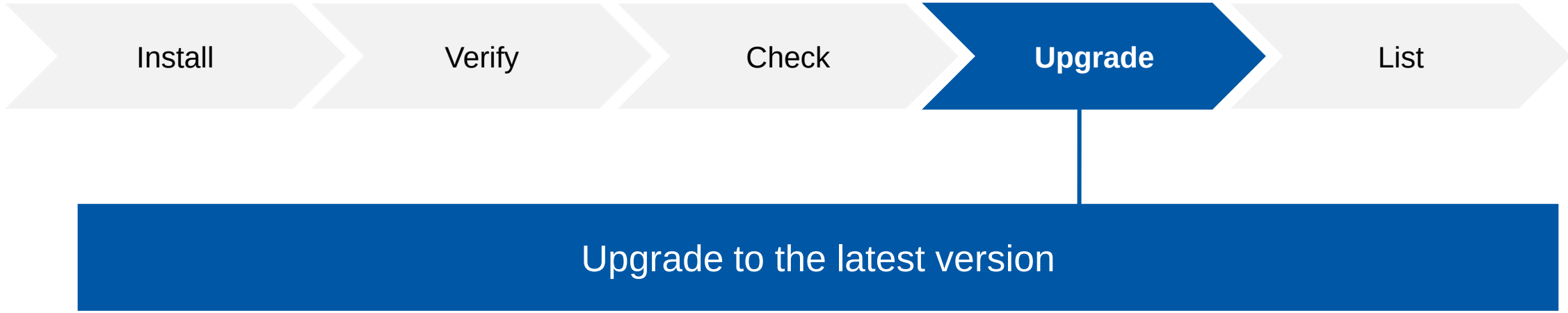
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

print("Libraries installed successfully!")
```

Python libraries can be managed through a simple process



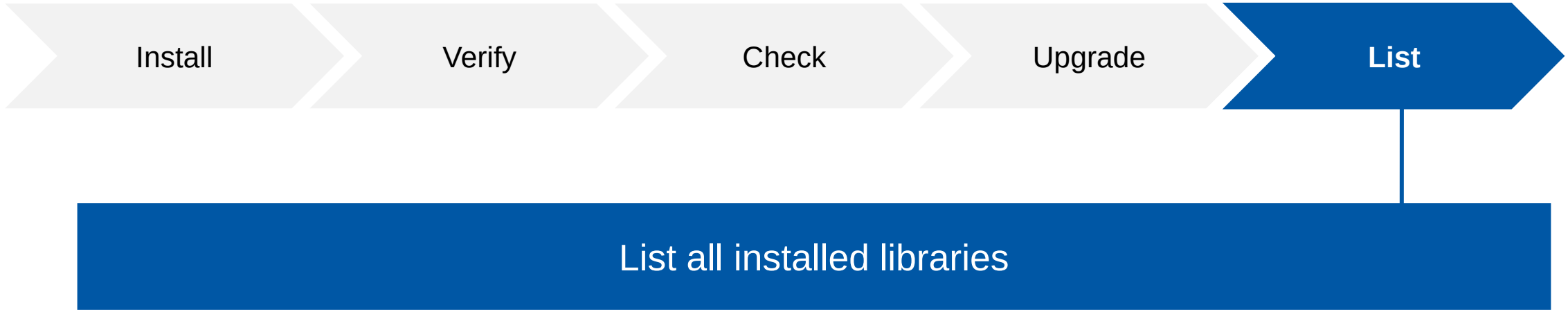

```
pip show numpy
```

Python libraries can be managed through a simple process




```
pip install --upgrade numpy
```

Python libraries can be managed through a simple process




```
pip list
```

Pandas and Matplotlib are some commonly used Python libraries

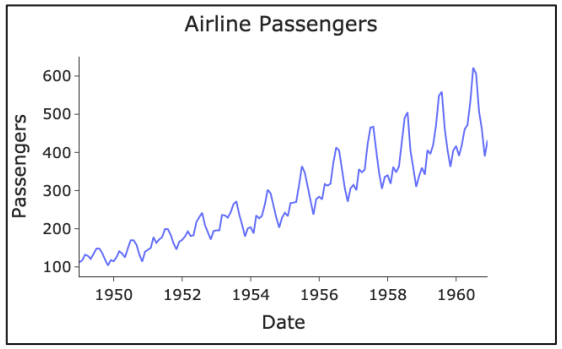
Pandas is a powerful library used to manipulate and analyse data

Pandas

Pandas helps you work with structured data, similarly to excel or google sheets. It provides easy-to-use data structures and functions to work efficiently with structured data types such as tables, time-series, and matrices.

	A	B	D	E	F
1	Incident Number	Incident Title	Date	Actual Severity	Potential Severity
22	2461462	Steam leak.	05.01.2024	1-Notable	1-Notable
23	2461497	Steam leak.	05.01.2024	1-Notable	1-Notable
24	2461503	Steam leak.	05.01.2024	1-Notable	1-Notable
25	2462554	44-LV2126 B/P I/V minor gland leak.	06.01.2024	1-Notable	1-Notable
26	2463477	Gas leak from plug of tube bundle 393 E101 (Gas Cooler)	06.01.2024	1-Notable	1-Notable
27	2463515	ACID LEAKS	06.01.2024	1-Notable	1-Notable

Tables



Time series

$$\begin{pmatrix} 1 & 5 & 3 \\ 4 & 9 & 2 \\ 5 & 6 & 7 \end{pmatrix} \quad \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 4 & 5 \end{bmatrix}$$

Matrices

The power of Pandas lies in handling large datasets extremely fast

Managing multiple excels can get cumbersome and forget handling heavy files at anything faster than a snail's pace

Funstional Location		Inspection	Loss of Primary Contameant	Work Orders	Reck Orders	Potential Ruocss
120345	Separator	Signs of internal	Internal corrosion		Corrosion Injuries	Rust
00389	Centritugal pump	Misalignment	Misaligement			
00251	Storage tank	Dents on shell	Inspect damage			
00144	Heat exchanger	Minor leaks Investigate	Investigate leaks			

Change Request ID	Type	Status
00456	Compressor oher	Downgraded Situation	Pending
00389	Update alarm set	Downgraded Situation	Complete
00251	Replace PSV on	Downgraded Situation	Open
00144	Pipeline rerouting	Operational	Closed

Incident Number	Incident Iroy	Date	Actual Severity	Potential Severity
120345	Production manifold	06/14/2023	High	Closed
118902	Flare line	05/09/2023	Minor	Moderate
116457	Turbine	03/22/2023	Serious	Closed
114036	Crude oll leak	01/30/2023	Warning	Closed

Recommendation Hedaline			
Work Order	Priority	Equip.ID	Type
Callbrate pressure	101567	A-26U	INSPECTION
Inspect PSV for de	100934	H-467	INSPECTION
Install new filter	095343	M-320	MAINTENAINCE
Replace control val	099522	C-1004	INSPECTION

Recommendation Description		Work Order	Priority	Equip. ID	Wor k Ce	Type
Calibrate pressure gauge	LOKE	101567	2	A-26U	PM01	INSPECTION
Inspect PSV for defects	Checks	100934	3	H-467	PM01	INSPECTION
Install a new filter	FUEL	095343	1	M-320	FABS	MAINTENANCE
Replace control valve	CLAIM	089522	1	C-1004	CFD	INSPECTION

Excel for Data Scientists

Pandas solves these problems by processing data quickly, efficiently, and at scale, all the while keeping it error free

```
import pandas as pd

# Reading from a CSV file
df = pd.read_csv("oil_gas_production.csv")

# Display the first few rows
print(df.head())
```

✓ 0.1s

```
         Date  Oil_Production  Gas_Production  Water_Cut   Field
0  01-01-2023            1360           49298   0.573058  Ruwais
1  02-01-2023            4272           24683   0.286655  Buhasa
2  03-01-2023            3592           10504   0.235372  Buhasa
3  04-01-2023             966           43982   0.422000  Buhasa
4  05-01-2023            4926           44299   0.304367    Asab
```

Command to import pandas

```
import pandas as pd
```

Master these Pandas skills to take control of your data

- Reading data with dataframes
- Handling missing values
- Exploring data
- Slicing data
- Renaming and dropping rows / columns
- Grouping and aggregating

Snapshot of the dataset used: oil_gas_production

500 rows!

	A	B	C	D	E
1	Date	Oil_Production	Gas_Production	Water_Cut	Field
2	01-01-2023	1360	49298	0.573	Ruwais
3	02-01-2023	4272	24683	0.287	Buhasa
4	03-01-2023	3592	10504	0.235	Buhasa
5	04-01-2023	966	43982	0.422	Buhasa
6	05-01-2023	4926	44299	0.304	Asab
7	06-01-2023	3944	38016	0.113	Habshan
8	07-01-2023	3671	33960	0.178	Habshan
9	08-01-2023	3419	43591	0.458	Buhasa
10	09-01-2023	630	27312	0.429	Habshan
11	10-01-2023	2185	47797	0.114	Habshan
12	11-01-2023	1269	12105	0.211	Buhasa
13	12-01-2023	2891	46395	0.216	Buhasa
14	13-01-2023	2933	32700	0.436	Ruwais
15	14-01-2023	1684	44620	0.110	Habshan
16	15-01-2023	3885	47678	0.152	Asab
17	16-01-2023	4617	30559	0.500	Buhasa
18	17-01-2023	3404	37509	0.189	Ruwais
19	18-01-2023		37860	0.426	Asab
20	19-01-2023	1582	21003	0.219	Habshan
21	20-01-2023	3058	31732	0.150	Ruwais
22	21-01-2023	2547	35826	0.222	Buhasa
23	22-01-2023	3247	40354	0.461	Asab
24	23-01-2023	1475	23843	0.528	Buhasa
25	24-01-2023	2306	16190	0.515	Habshan
26	25-01-2023	689	27640	0.299	Asab
27	26-01-2023	3234	48413	0.434	Habshan
28	27-01-2023	3505	13330	0.202	Habshan
29	28-01-2023	2399	29087	0.247	Ruwais
30	29-01-2023	1767	34504	0.548	Buhasa
31	30-01-2023	2028	17114	0.107	Asab
32	31-01-2023	3702	23323	0.143	Habshan
33	01-02-2023	4056	44121	0.204	Habshan
34	02-02-2023	4390	20975	0.113	Asab
35	03-02-2023	1146	21023	0.191	Habshan
36	04-02-2023	3388	31447	0.392	Ruwais
37	05-02-2023	2935	34933	0.311	Ruwais
38	06-02-2023	1100	33959	0.546	Ruwais
39	07-02-2023	2863	10667	0.509	Ruwais
40	08-02-2023	2561	39703	0.271	Habshan
41	09-02-2023	741	19337	0.230	Asab
42	10-02-2023	2541	46487	0.290	Asab

5 Columns

Column 1: Date
States production date (from 01-01-2023 to 14-05-2024)

Column 2: Oil_Production
Measures daily oil output (in barrels per day)

Column 3: Gas_Production
Measures daily gas output (in cubic feet per day)

Column 4: Water_Cut
Measures proportion of water mixed in with oil (in%)

Column 5: Field
Oil field where production happened
(Asab, Buhasa, Habshan or Ruwais)

Data is stored in dataframes which are very similar to excel tables

Command

Output

The **df** variable translates the data from your file to a Pandas dataframe

```
import pandas as pd

# Reading from a CSV file
df = pd.read_csv("oil_gas_production.csv")

# Display the first few rows
print(df.head())
```

	Date	Oil_Production	Gas_Production	Water_Cut	Field
0	01-01-2023	1360	49298	0.573058	Ruwais
1	02-01-2023	4272	24683	0.286655	Buhasa
2	03-01-2023	3592	10504	0.235372	Buhasa
3	04-01-2023	966	43982	0.422000	Buhasa
4	05-01-2023	4926	44299	0.304367	Asab

Similarly, you can **read** excel, json files, etc

You can use different commands to explore the data

Command

Output

Display first 5 rows

```
print(df.head())
```

✓ 0.0s

	Date	Oil_Production	Gas_Production	Water_Cut	Field
0	01-01-2023	1360	49298	0.573058	Ruwais
1	02-01-2023	4272	24683	0.286655	Buhasa
2	03-01-2023	3592	10504	0.235372	Buhasa
3	04-01-2023	966	43982	0.422000	Buhasa
4	05-01-2023	4926	44299	0.304367	Asab

Display last 5 rows

```
print(df.tail())
```

✓ 0.0s

	Date	Oil_Production	Gas_Production	Water_Cut	Field
495	10-05-2024	1743	36737	0.466557	Ruwais
496	11-05-2024	4209	21485	0.165784	Buhasa
497	12-05-2024	1581	48565	0.457912	Buhasa
498	13-05-2024	955	35522	0.554516	Habshan
499	14-05-2024	1394	22342	0.189842	Asab

You can use different commands to explore the data

Command

Output

Give a quick numerical summary

```
print(df.describe())
```

✓ 0.0s

	Oil_Production	Gas_Production	Water_Cut
count	500.000000	500.000000	500.000000
mean	2805.660000	29609.624000	0.351864
std	1261.356268	11928.592366	0.148604
min	504.000000	10009.000000	0.103193
25%	1666.750000	19298.750000	0.216319
50%	2930.000000	29126.000000	0.361152
75%	3830.750000	39213.500000	0.480836
max	4999.000000	49964.000000	0.598967

You can use different commands to explore the data

Command

Output

Display unique values in a column

```
print(df["Field"].unique())
```

✓ 0.1s

```
['Ruwais' 'Buhasa' 'Asab' 'Habshan']
```

Count occurrences of each value

```
print(df["Field"].value_counts())
```

✓ 0.0s

```
Field
Habshan    137
Buhasa     130
Ruwais     117
Asab       116
Name: count, dtype: int64
```

Pandas allows you to rename and drop rows or columns

Original Table

	Date	Oil_Production	Gas_Production	Water_Cut	Field
0	01-01-2023	1360	49298	0.573058	Ruwais
1	02-01-2023	4272	24683	0.286655	Buhasa
2	03-01-2023	3592	10504	0.235372	Buhasa
3	04-01-2023	966	43982	0.422000	Buhasa
4	05-01-2023	4926	44299	0.304367	Asab

Command

```
# Drop a column
df.drop(columns=["Water_Cut"], inplace=True)
```

✓ 0.0s

```
# Rename a column
df.rename(columns={"Oil_Production": "Oil_Output"}, inplace=True)
```

✓ 0.0s

```
# Drop rows with specific conditions
df = df[df["Oil_Output"] > 3000]  # Keep only wells producing more than 3000 bpd
```

✓ 0.0s

Pandas allows you to rename and drop rows or columns

Original Table

	Date	Oil_Production	Gas_Production	Water_Cut	Field
0	01-01-2023	1360	49298	0.573058	Ruwais
1	02-01-2023	4272	24683	0.286655	Buhasa
2	03-01-2023	3592	10504	0.235372	Buhasa
3	04-01-2023	966	43982	0.422000	Buhasa
4	05-01-2023	4926	44299	0.304367	Asab

Modified Table

	Date	Oil_Output	Gas_Production	Field
1	02-01-2023	4272	24683	Buhasa
2	03-01-2023	3592	10504	Buhasa
4	05-01-2023	4926	44299	Asab
5	06-01-2023	3944	38016	Habshan
6	07-01-2023	3671	33960	Habshan

Handling missing values becomes much easier in Pandas

Command

Output

Check for missing values

```
print(df.isnull().sum())
```

✓ 0.0s

```
Date              0
Oil_Production    4
Gas_Production    4
Water_Cut         3
Field             0
dtype: int64
```

Other actions you can do with missing values

Action	Command
Drop rows with missing values	`df.dropna ( inplace=True )`
Fill missing values with 0	`df.fillna (0, inplace=True )`
Fill missing values with column mean	`df["Oil_Output"].fillna(df["Oil_Output"].mean(), inplace=True)`

Slicing allows you to extract specific rows, columns, or both

Original Table

	Date	Oil_Production	Gas_Production	Water_Cut	Field
0	01-01-2023	1360	49298	0.573058	Ruwais
1	02-01-2023	4272	24683	0.286655	Buhasa
2	03-01-2023	3592	10504	0.235372	Buhasa
3	04-01-2023	966	43982	0.422000	Buhasa
4	05-01-2023	4926	44299	0.304367	Asab

Slicing

You may not need to work on all the data in your dataframe. By slicing the data, you can access the particular subsets you want to work on.

Slicing allows you to extract specific rows, columns, or both

Select specific columns

Command

Output

```
df_subset = df[["Field", "Oil_Output"]]
print(df_subset)
```

```
     Field  Oil_Output
0   Ruwais      1360.0
1  Buhasa       4272.0
2  Buhasa       3592.0
3  Buhasa        966.0
4    Asab       4926.0
```

Slicing allows you to extract specific rows, columns, or both

Select rows based on conditions

Command

Output

```
high_output_wells = df[df["Oil_Output"] > 3000]
print(high_output_wells)
```

	Date	Oil_Output	Gas_Production	Water_Cut	Field
1	02-01-2023	4272.0	24683.0	0.286655	Buhasa
2	03-01-2023	3592.0	10504.0	0.235372	Buhasa
4	05-01-2023	4926.0	44299.0	0.304367	Asab
5	06-01-2023	3944.0	38016.0	0.112693	Habshan
6	07-01-2023	3671.0	33960.0	0.178076	Habshan

Slicing allows you to extract specific rows, columns, or both

.loc[] (Label-based selection) extracts data using column names or row labels

Command

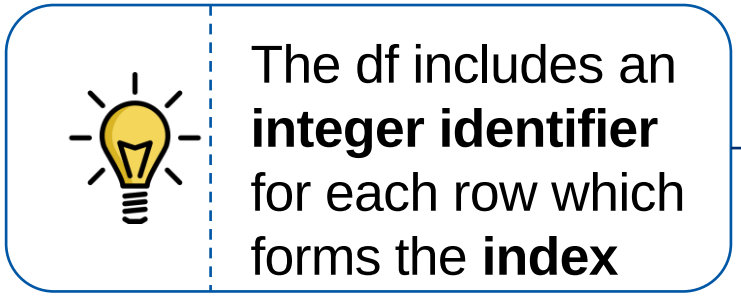
Output

```
subset_loc = df.loc[df["Field"] == "Buhasa", ["Field", "Oil_Output"]]
print(subset_loc)
```

```
      Field  Oil_Output
1    Buhasa      4272.0
2    Buhasa      3592.0
3    Buhasa       966.0
7    Buhasa      3419.0
10   Buhasa      1269.0
```

Slicing allows you to extract specific rows, columns, or both

.iloc[] (Index-based selection) extracts data by row and column positions (integer index)



	Date	Oil_Production	Gas_Production	Water_Cut	Field
0	01-01-2023	1360	49298	0.573058	Ruwais
1	02-01-2023	4272	24683	0.286655	Buhasa
2	03-01-2023	3592	10504	0.235372	Buhasa
3	04-01-2023	966	43982	0.422000	Buhasa
4	05-01-2023	4926	44299	0.304367	Asab

Slicing allows you to extract specific rows, columns, or both

.iloc[] (Index-based selection) extracts data by row and column positions (integer index)

Command

Output

You can select specific rows by index

```
df_head = df.iloc[:5]
print(df_head)
```

	Date	Oil_Output	Gas_Production	Water_Cut	Field
0	01-01-2023	1360.0	49298.0	0.573058	Ruwais
1	02-01-2023	4272.0	24683.0	0.286655	Buhasa
2	03-01-2023	3592.0	10504.0	0.235372	Buhasa
3	04-01-2023	966.0	43982.0	0.422000	Buhasa
4	05-01-2023	4926.0	44299.0	0.304367	Asab

Slicing allows you to extract specific rows, columns, or both

.iloc[] (Index-based selection) extracts data by row and column positions (integer index)

You can specify which columns to extract from the index

```
subset_iloc = df.iloc[2:5, 1:3]  # Rows 2 to 4, Columns 1 to 2
print(subset_iloc)
```

	Oil_Output	Gas_Production
2	3592.0	10504.0
3	966.0	43982.0
4	4926.0	44299.0

You can group and aggregate data to filter and analyse categories

Command

Output

Group by field and calculate
total output per field

```
field_production = df.groupby("Field")["Oil_Output"].sum()
print(field_production)
```

```
Field
Asab        302942.0
Buhasa      355313.0
Habshan     387073.0
Ruwais      325225.0
Name: Oil_Output, dtype: float64
```

You can group and aggregate data to filter and analyse categories

Command

Output

Count the number of days the field was operational

```
well_count = df.groupby("Field")["Date"].count()
print(well_count)
```

```
Field
Asab        113
Buhasa      128
Habshan     135
Ruwais      116
Name: Date, dtype: int64
```

You can group and aggregate data to filter and analyse categories

Command

Output

Aggregate multiple statistics (sum, mean, max) for oil output

```
agg_stats = df.groupby("Field").agg({
    "Oil_Output": ["sum", "mean", "max"],
})
print(agg_stats)
```

	Oil_Output		
	sum	mean	max
Field			
Asab	307935.0	2725.088496	4999.0
Buhasa	355313.0	2775.882812	4996.0
Habshan	389554.0	2885.585185	4988.0
Ruwais	328160.0	2828.965517	4996.0

You can group and aggregate data to filter and analyse categories

Command

Output

Group by field and filter those with total oil production above 50,000

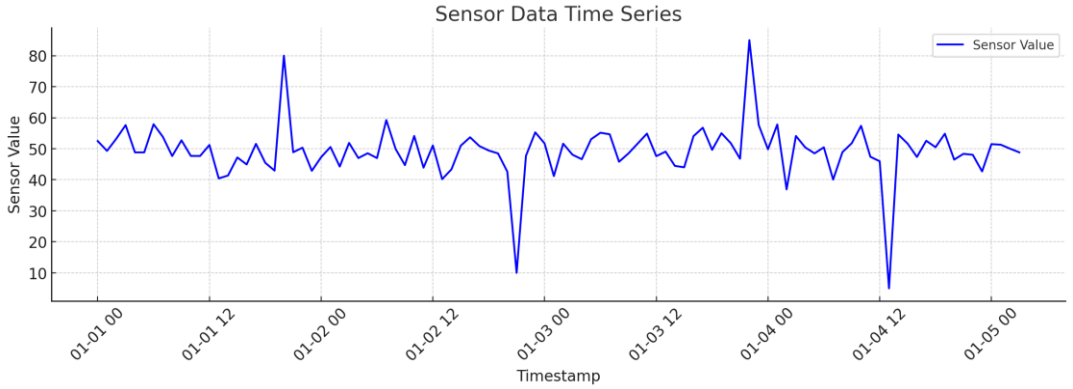
```
high_producing_fields = df.groupby("Field").filter(lambda x: x["Oil_Output"].sum() > 50000)
print(high_producing_fields)
```

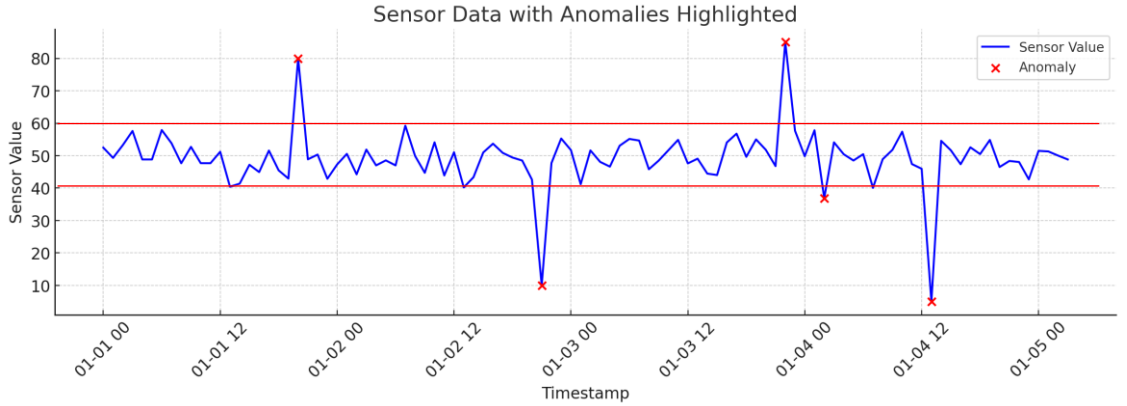
```
           Date  Oil_Output  Gas_Production    Field
0    01-01-2023      1360.0         49298.0   Ruwais
1    02-01-2023      4272.0         24683.0   Buhasa
2    03-01-2023      3592.0         10504.0   Buhasa
3    04-01-2023       966.0         43982.0   Buhasa
4    05-01-2023      4926.0         44299.0     Asab
..          ...         ...             ...      ...
495  10-05-2024      1743.0         36737.0   Ruwais
496  11-05-2024      4209.0         21485.0   Buhasa
497  12-05-2024      1581.0         48565.0   Buhasa
498  13-05-2024       955.0         35522.0  Habshan
499  14-05-2024      1394.0         22342.0     Asab

[492 rows x 4 columns]
```

Real world application: Finding anomalies in sensor data using time series analysis

Command

Output






```
Sensor[“Anomaly”] = sensor[‘value’]
                    .map(lambda x: True if x > 60 or x < 40 else False)
```

Possible use cases

Predicting Energy Demand

Monitoring Operational Performance

Data visualisation makes it easy to explore complex data

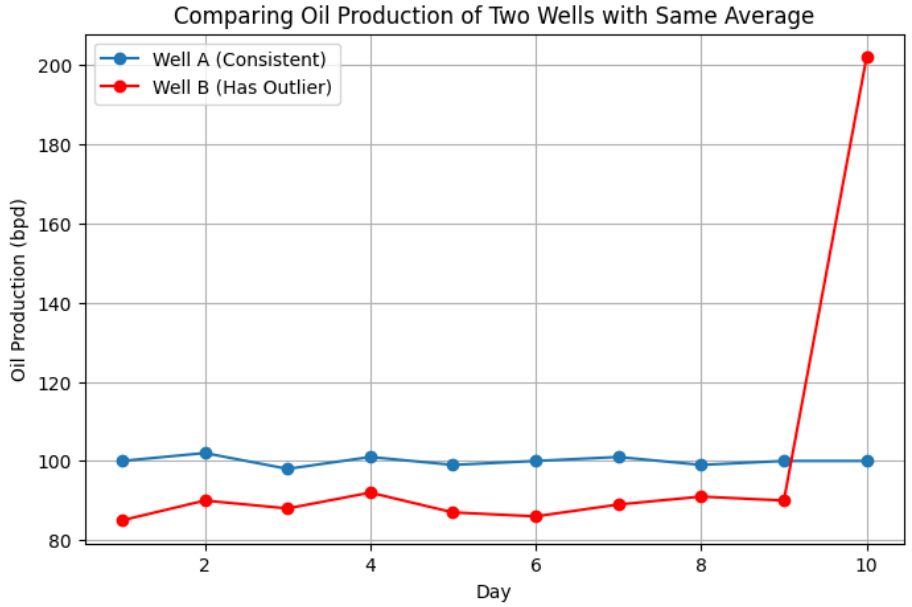
Instead of presenting raw data as a table, you can present data graphically using charts, graphs, and plots to help interpret trends, patterns, and relationships within datasets

But why do you need data visualisation when you can work with raw data?

Data visualisation makes it easy to explore complex data

Consider a case where average production of two oil wells is the same. Does that mean they are equal in every aspect?

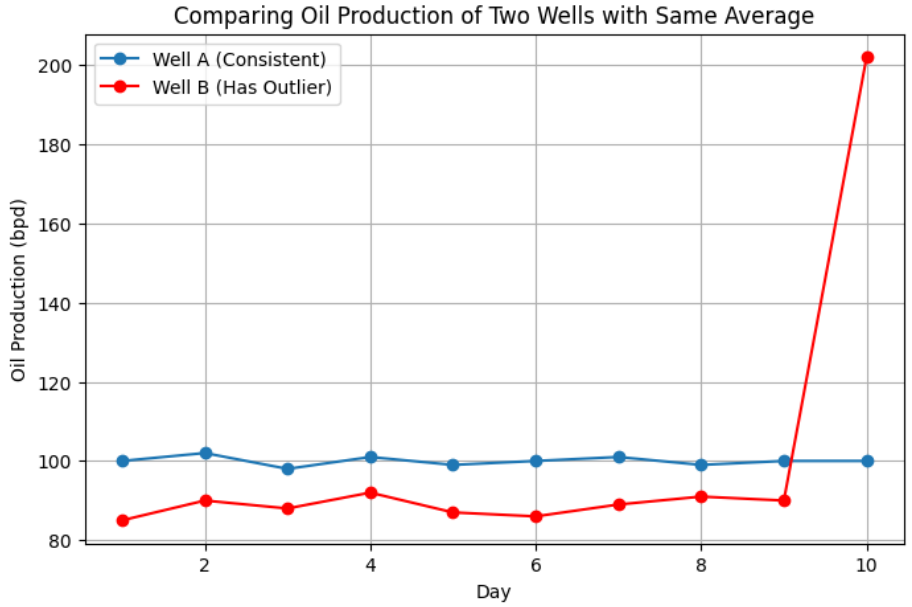
	A	B	C
1	**Day**	**Well A**	**Well B**
2	1	100	85
3	2	102	90
4	3	98	88
5	4	101	92
6	5	99	87
7	6	100	86
8	7	101	89
9	8	99	91
10	9	100	90
11	10	100	202
12	**Average**	**100**	**100**



Data visualisation makes it easy to explore complex data

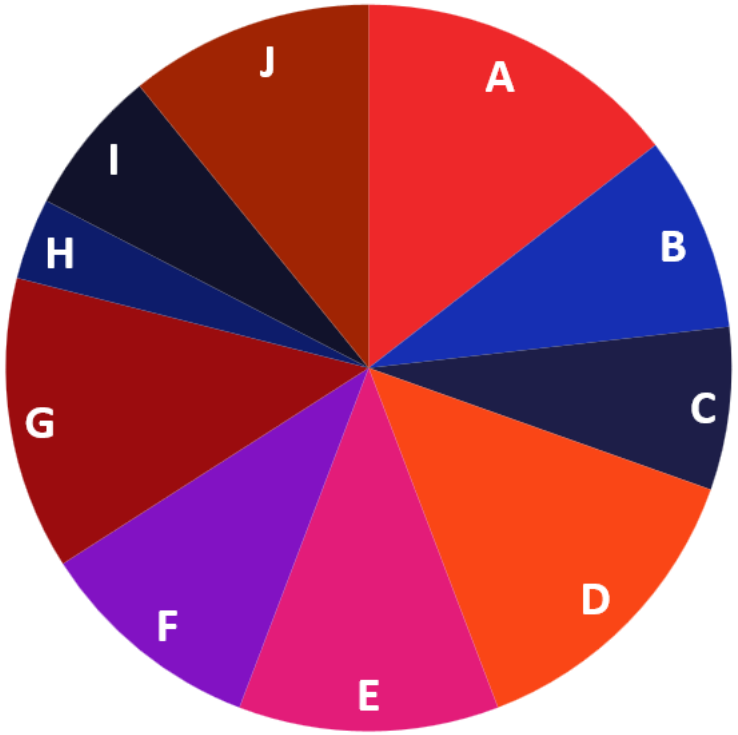
Graphically, we can see that Well A is performing consistently but Well B has an outlier on one day which will prompt us to dig into the issue

	A	B	C
1	**Day**	**Well A**	**Well B**
2	1	100	85
3	2	102	90
4	3	98	88
5	4	101	92
6	5	99	87
7	6	100	86
8	7	101	89
9	8	99	91
10	9	100	90
11	10	100	202
12	**Average**	**100**	**100**



Different types of data benefit from different ways of visualisation

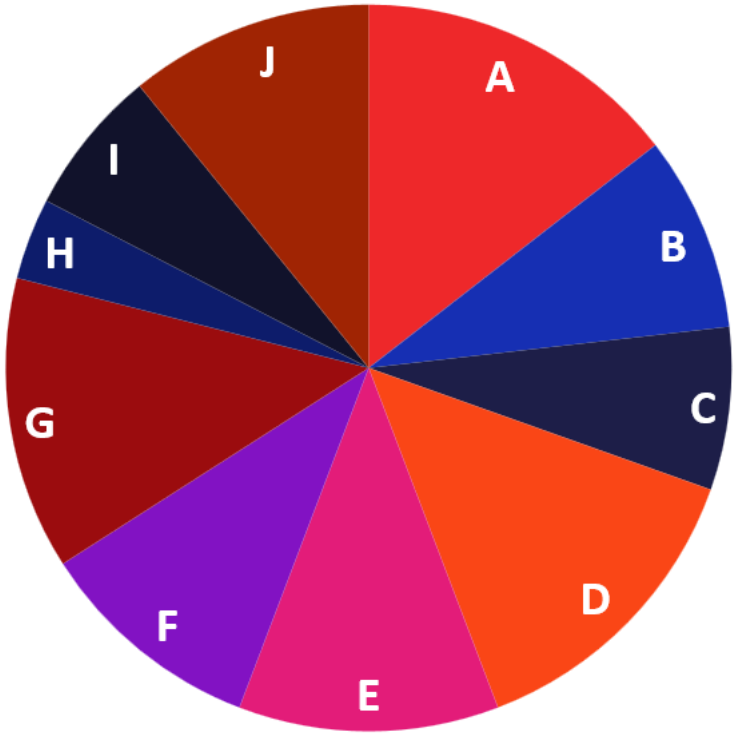
Which is the third largest segment?



vs.

Different types of data benefit from different ways of visualisation

It’s difficult to answer from a pie chart,
while it’s evident from a bar chart arranged in order



vs.

Matplotlib is a Python library used for data visualisation

Matplotlib is a Python library used for creating static, animated, and interactive visualizations. It is widely used for scientific computing, engineering, and business applications.

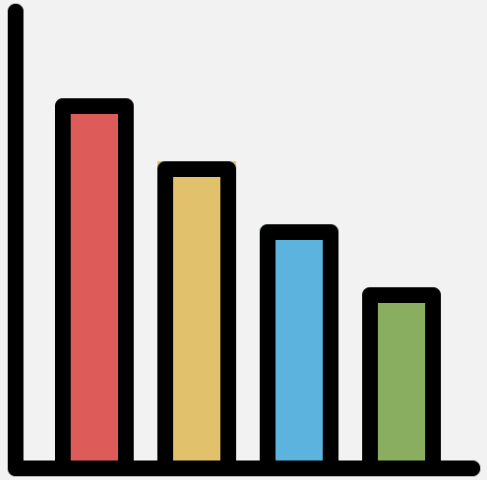
Command to import matplotlib

```
import matplotlib.pyplot as plt
```

Powerful graphs you can create with Matplotlib

Line Plots

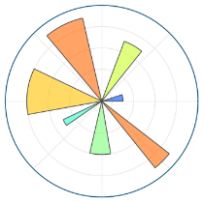
Bar Charts



Scatter Plots

Line Plots are useful to show trends over time

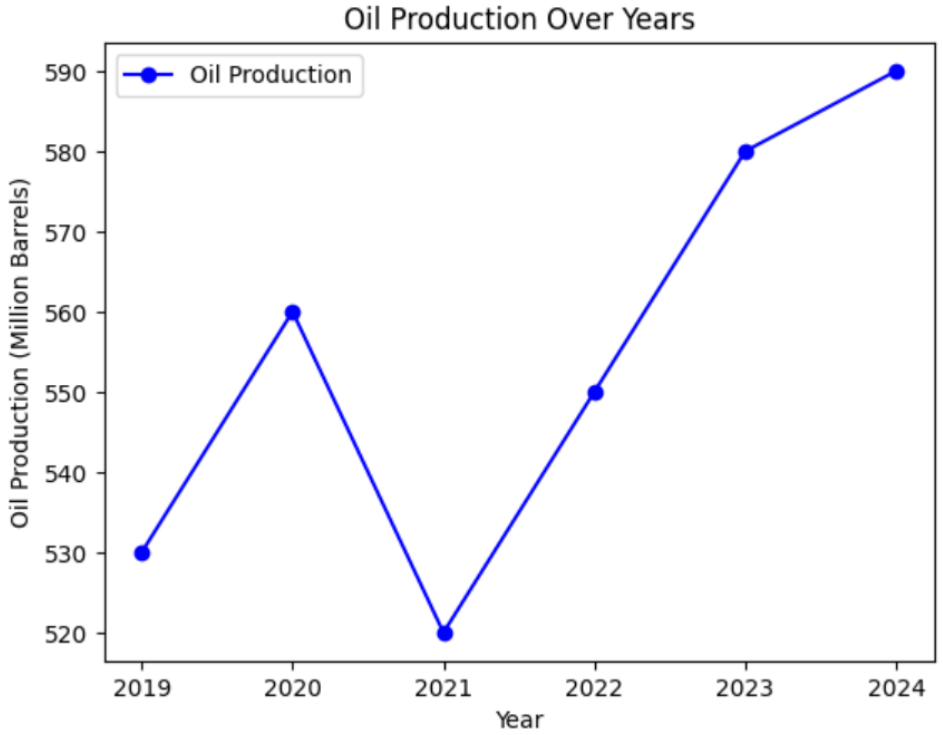
Command



Output

Use **plt.plot**

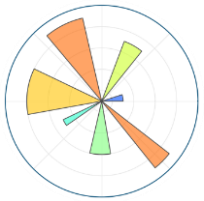
```
import matplotlib.pyplot as plt

# Data
years = [2019, 2020, 2021, 2022, 2023, 2024]
production = [530, 560, 520, 550, 580, 590]  # In million barrels

# Creating the plot
plt.plot(years, production, marker='o', linestyle='-',
colour='b', label="Oil Production")

# Labels and title
plt.xlabel("Year")
plt.ylabel("Oil Production (Million Barrels)")
plt.title("Oil Production Over Years")

# Display Legend
plt.legend()

# Show plot
plt.show()
```



This line plot shows clearly the trend of oil production over time, making it easy to observe patterns, fluctuations, and overall direction

Bar Plots make comparisons fun and easy

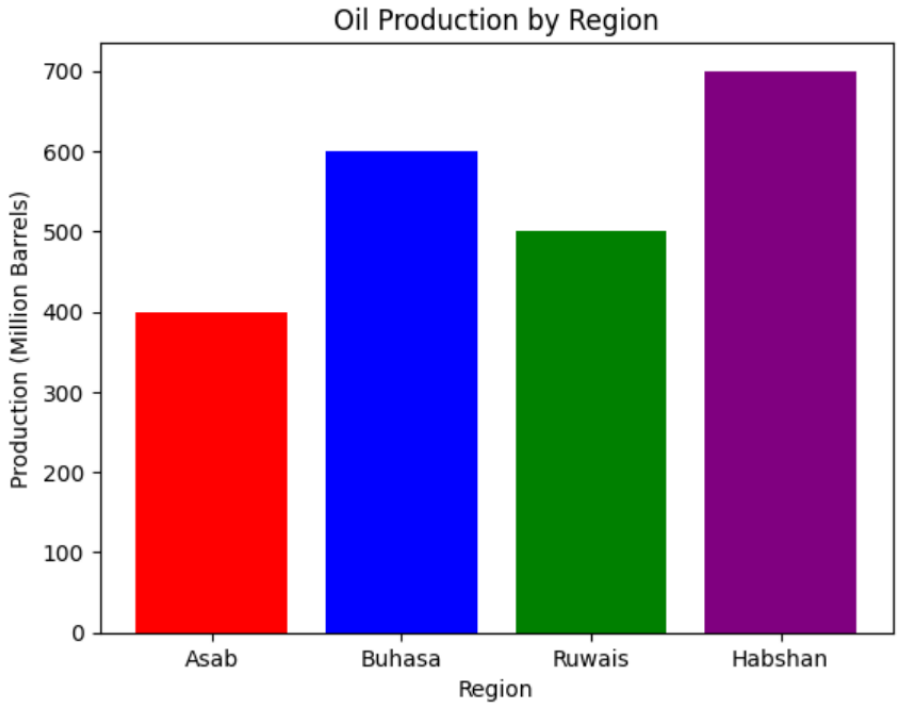
Command



Output

Use **plt.bar**

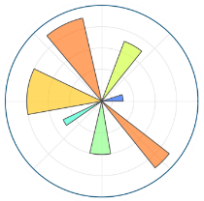
```
import matplotlib.pyplot as plt
# Data
regions = ["Asab", "Buhasa", "Ruwais", "Habshan"]
production = [400, 600, 500, 700]  # Production in million barrels

# Creating the plot
plt.bar(regions, production, colour=['red', 'blue', 'green',
'purple'])

# Labels and title
plt.xlabel("Region")
plt.ylabel("Production (Million Barrels)")
plt.title("Oil Production by Region")

# Show plot
plt.show()
```



This bar plot shows oil production across different regions, highlighting differences in production levels at a glance

Scatter plots highlight relationships between data

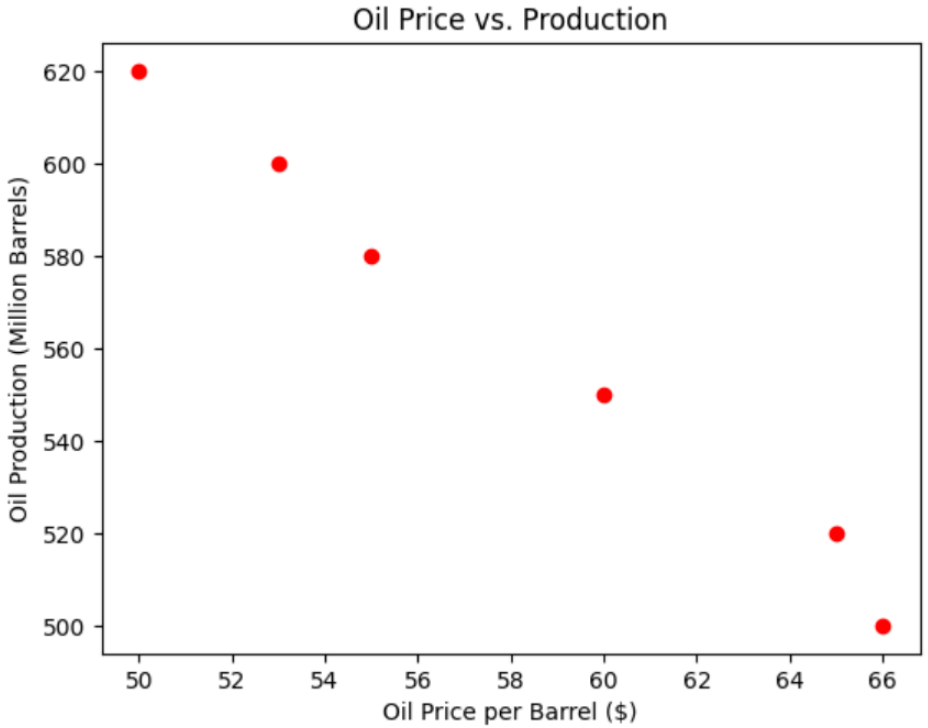
Command



Output

Use **plt.scatter**

```
import matplotlib.pyplot as plt

# Data
prices = [66, 65, 60, 55, 53, 50]  # Price per barrel
production = [500, 520, 550, 580, 600, 620]  # Million barrels


# Creating the plot
plt.scatter(prices, production, colour='red', marker='o')

# Labels and title
plt.xlabel("Oil Price per Barrel ($)")
plt.ylabel("Oil Production (Million Barrels)")
plt.title("Oil Price vs. Production")

# Show plot
plt.show()
```



This scatter plot illustrates the inverse correlation between oil production and prices, showing that as production increases, oil prices tend to decrease

Which would you use to compare expenses per region?

A. Scatter Plots

B. Line Plots

C. Bar Chart

Which would you use to compare expenses per region?

A. Scatter Plots

B. Line Plots

C. Bar Chart

Python Libraries

In this session, we covered:

- **How to install a Python Library**
- **Using Pandas to explore datasets**
- **Using Matplotlib to visualise data**