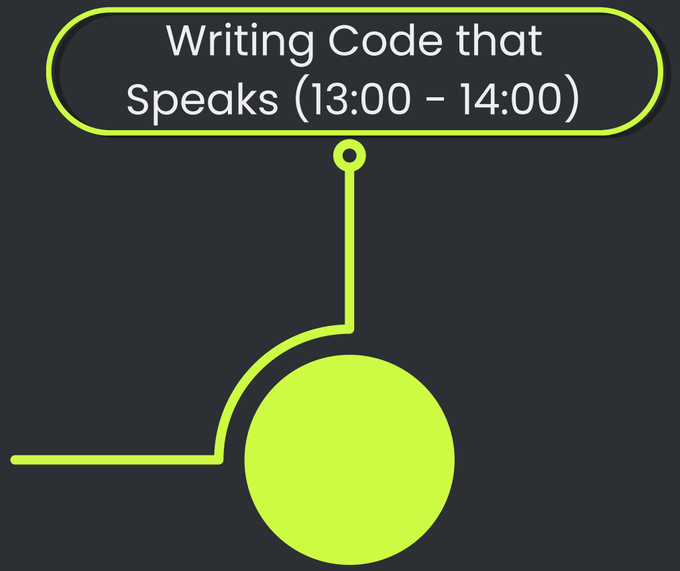
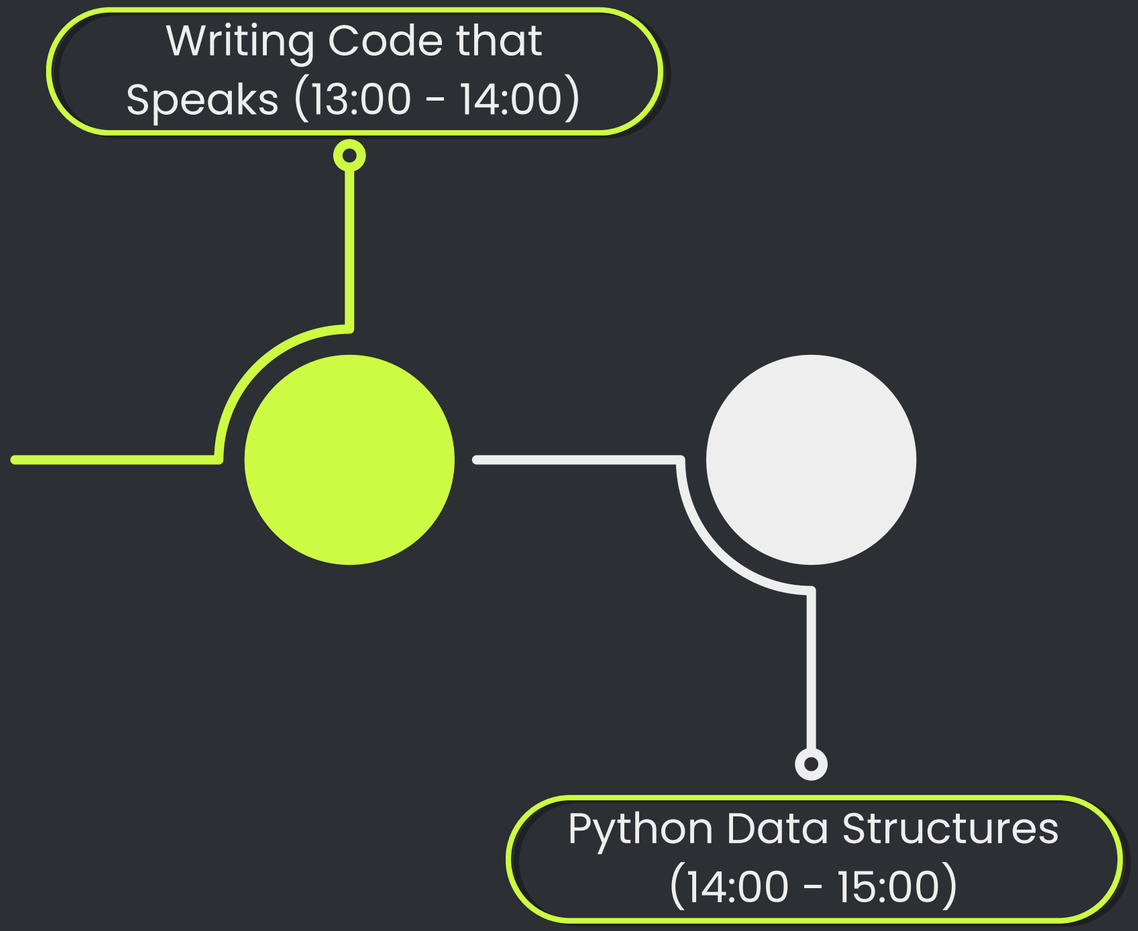
Lunch time

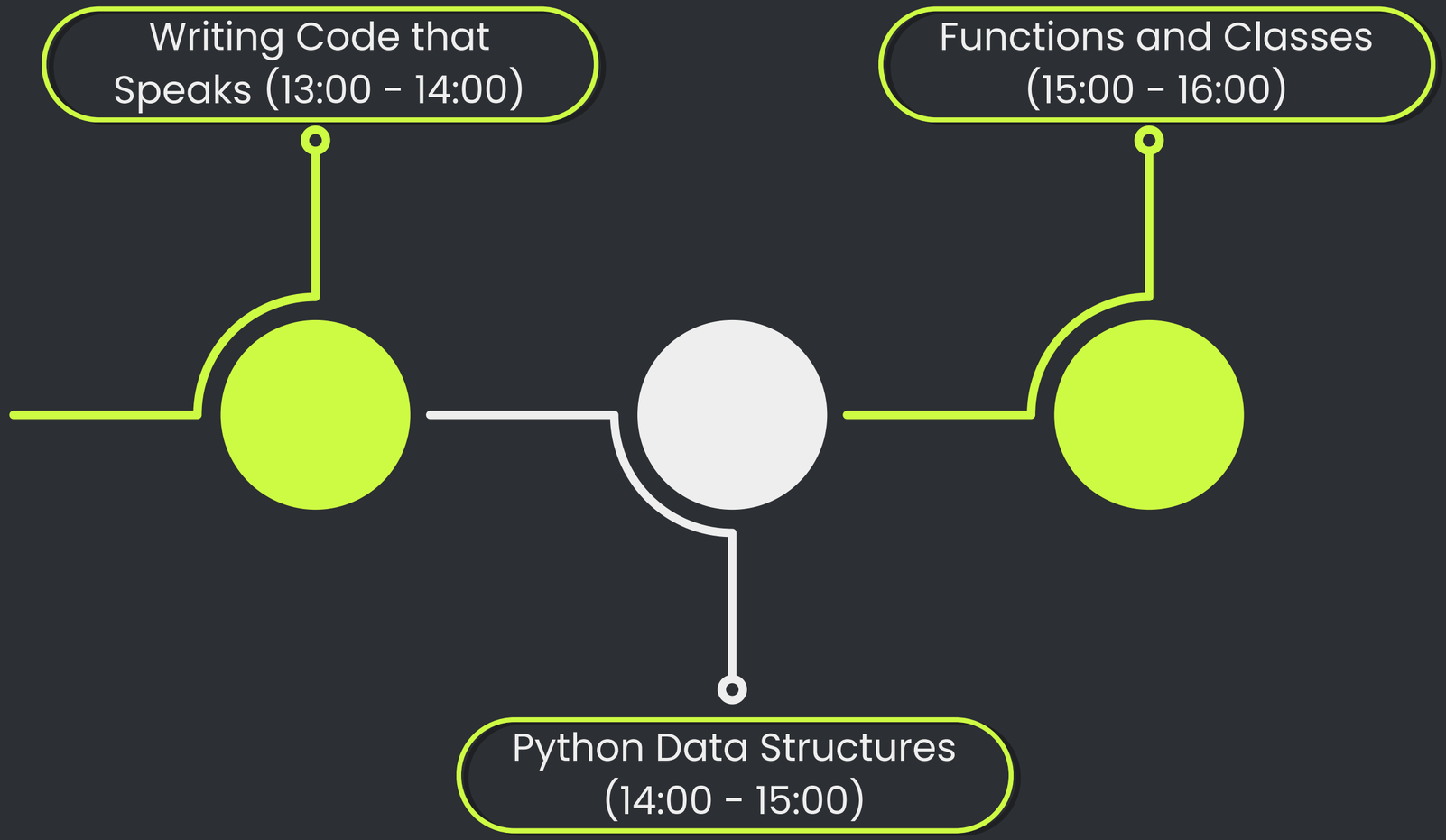
Afternoon Session (13:00 – 17:30)



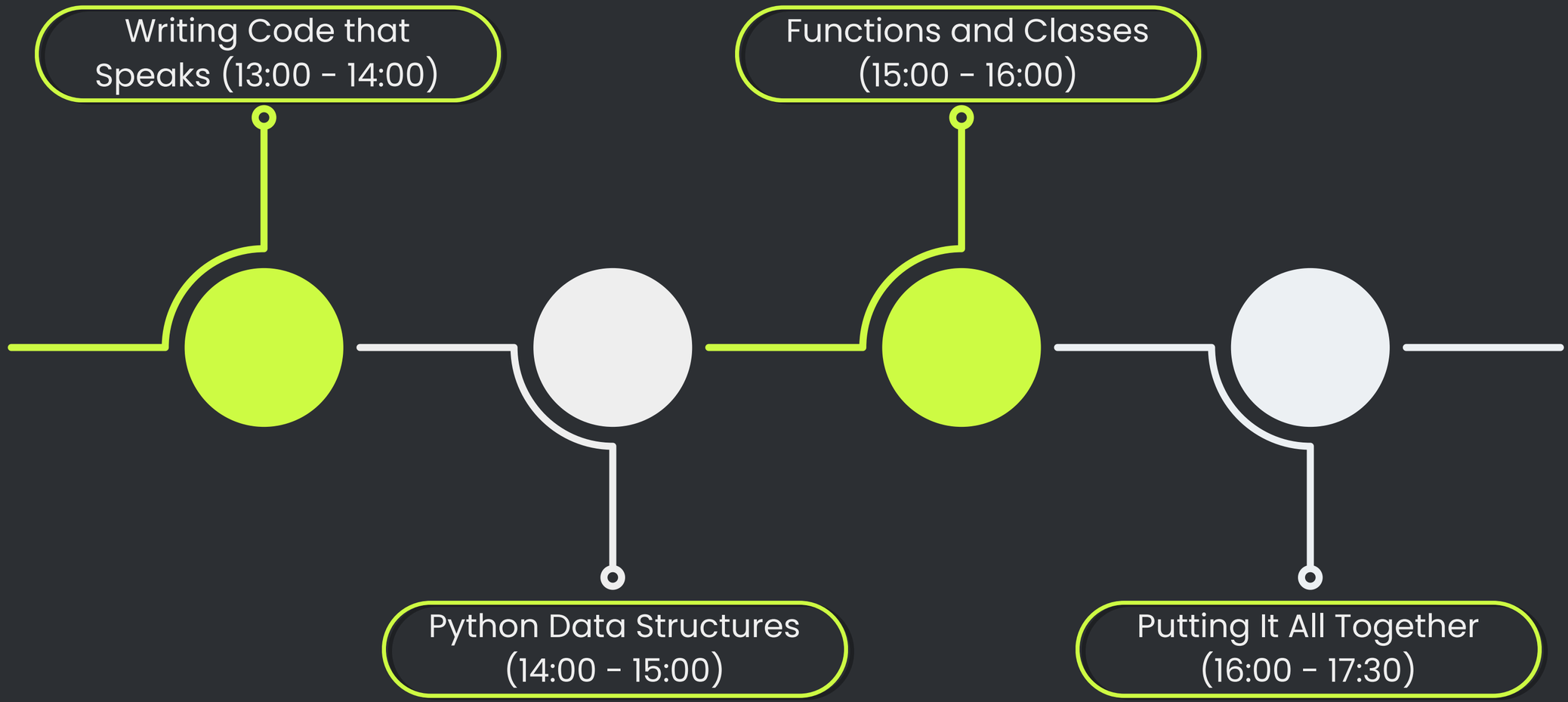
Afternoon Session (13:00 – 17:30)



Afternoon Session (13:00 – 17:30)



Afternoon Session (13:00 – 17:30)



Writing Code that Speaks

Machine Vision
using Python (MVUP01)

Rstats

Writing Code that Speaks

Machine Vision
using Python (MVUP01)

R stats

Writing Code that Speaks

What is a hard to read code

```
p=100;i=60;d=i/p
if d>.8:r="High";elif d>.5:r="Medium";else:r="Low"
print(f"d={d:.2f}",f"r={r}")
```

What does this code do?

Writing Code that Speaks

What is a hard to read code

```
p=100;i=60;d=i/p
if d>.8:r="High";elif d>.5:r="Medium";else:r="Low"
print(f"d={d:.2f}",f"r={r}")
```

```
d=0.60 r=Medium
```

What does this code do?

Writing Code that Speaks

What is a hard to read code

```python
# Define plot measurements
plot_area = 100        # Area in square meters
individuals = 60       # Number of individuals counted

# Calculate density
density = individuals / plot_area

# Assess density level
if density > 0.8:
    density_level = "High"
elif density > 0.5:
    density_level = "Medium"
else:
    density_level = "Low"

# Print results
print(f"Density: {density:.2f} individuals per square meter")
print(f"Density Level: {density_level}")
```

What does this code do?

Writing Code that Speaks

What is a hard to read code

```python
# Define plot measurements
plot_area = 100        # Area in square meters
individuals = 60       # Number of individuals counted

# Calculate density
density = individuals / plot_area

# Assess density level
if density > 0.8:
    density_level = "High"
elif density > 0.5:
    density_level = "Medium"
else:
    density_level = "Low"

# Print results
print(f"Density: {density:.2f} individuals per square meter")
print(f"Density Level: {density_level}")
```

```
Density: 0.60 individuals per square meter
Density Level: Medium
```

What does this code do?

Writing Code that Speaks

Key differences that make the clean code version better:

- Meaningful variable names (plot_area vs p)
- Comments explaining each step
- Clear spacing and structure
- Descriptive output messages
- Easy to modify threshold values
- Logical organisation of calculation and classification

Both codes perform the same density calculation and classification, but the clean version is much easier to understand and maintain.

Writing Code that Speaks

Case conventions:

```
# Snake case
total_spend = 3150.96

# Pascal case
TotalSpend = 3150.96

# Camel case
totalSpend = 3150.96
```

Writing Code that Speaks

Key differences that make the clean code version better:

- Meaningful variable names (plot_area vs p)
- Comments explaining each step
- Clear spacing and structure
- Descriptive output messages
- Easy to modify threshold values
- Logical organisation of calculation and classification

Both codes perform the same density calculation and classification, but the clean version is much easier to understand and maintain.

Writing Code that Speaks

Zen of Python

```
import this
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Writing Code that Speaks

Readability and Beauty:

- Beautiful is better than ugly
- Readability counts
- Simple is better than complex
- Complex is better than complicated
- Flat is better than nested
- Sparse is better than dense

```
# 1. Readability and Beauty
# Clear variable names and simple structure
temperature_celsius = 25.5
humidity_percent = 80
light_intensity = 1500  # lumens
```

Writing Code that Speaks

Explicitness and Clarity:

- Explicit is better than implicit
- There should be one obvious way to do it
- In the face of ambiguity, refuse the temptation to guess
- If the implementation is hard to explain, it's a bad idea
- If the implementation is easy to explain, it may be a good idea

```python
# 2. Explicitness and Clarity
# Clear conditions and explicit thresholds
def is_suitable_growth_condition(temp, humidity, light):
    is_temp_good = 20 <= temp <= 30
    is_humidity_good - 60 <= humidity <= 90
    is_light_good = 1000 <= light <= 2000

    return is_temp_good and is_humidity_good and is_light_good
```

Writing Code that Speaks

Error Handling

- Errors should never pass silently
- Unless explicitly silenced

```
# 3. Error Handling
# Explicit error checking
try:
    if temperature_celsius < 0 or temperature_celsius > 50:
        raise ValueError("Temperature out of realistic range")

    if humidity_percent < 0 or humidity_percent > 100:
        raise ValueError("Humidity must be between 0-100%")

    if light_intensity < 0:
        raise ValueError("Light intensity cannot be negative")

except ValueError as error:
    print(f"Error in measurements: {error}")
    exit()
```

Writing Code that Speaks

Practicality:

- Although practicality beats purity
- Now is better than never
- Although never is often better than right now
- Special cases aren't special enough to break the rules

```
# 4. Practicality
# Simple, practical output
growth_suitable = is_suitable_growth_condition(
    temperature_celsius,
    humidity_percent,
    light_intensity
)
```

Writing Code that Speaks

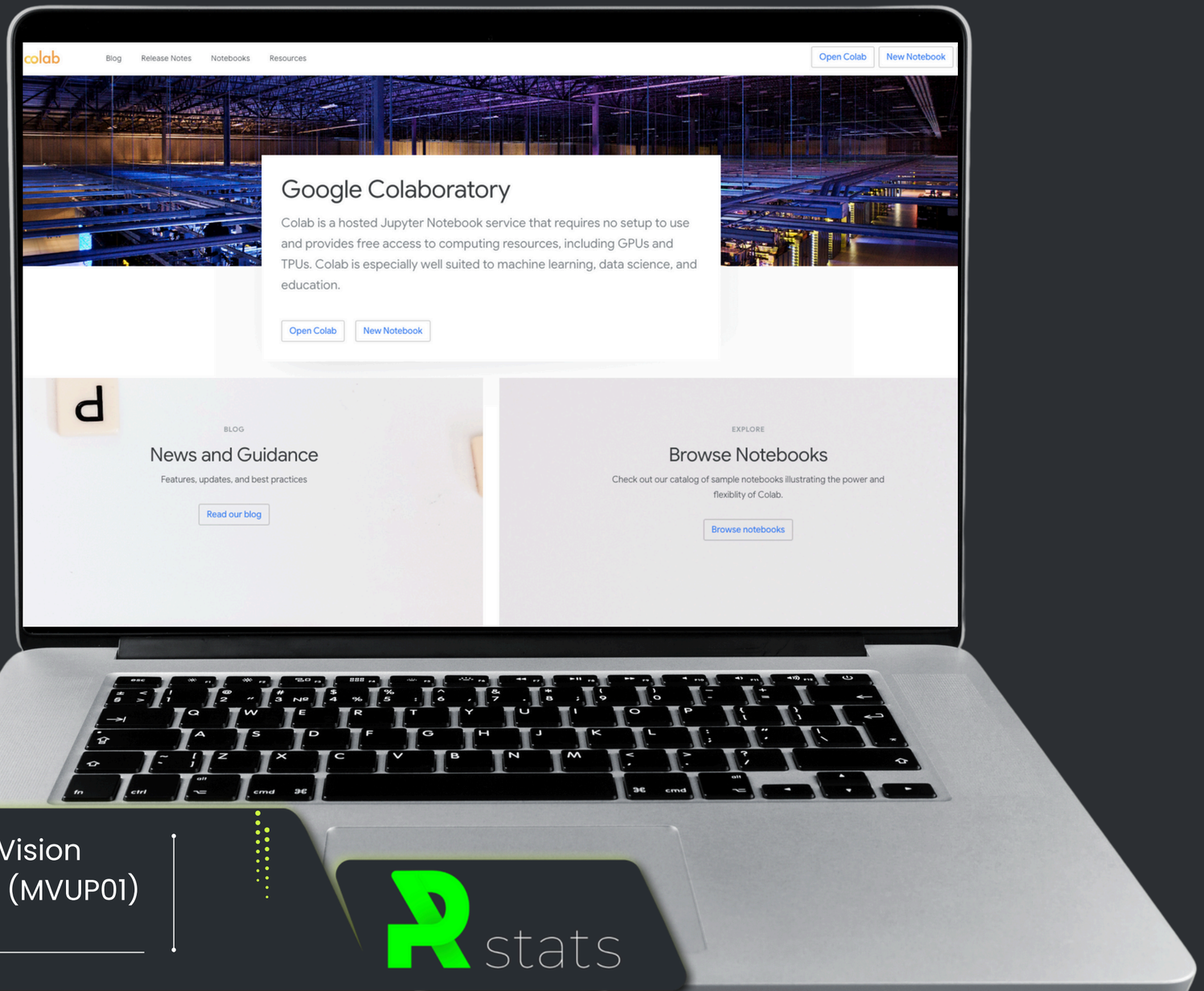
Code Organisation:

- Namespaces are one honking great idea
- Simple and uncluttered syntax
- Meaningful naming conventions

```
# 5. Code Organization
# Organized output with clear sections
print("\nPlant Growth Condition Analysis:")
print("-" * 30)
print(f"Temperature: {temperature_celsius}°C")
print(f"Humidity: {humidity_percent}%")
print(f"Light Intensity: {light_intensity} lumens")
print("-" * 30)
print(f"Conditions Suitable: {growth_suitable}")
```

Making Decisions in Code

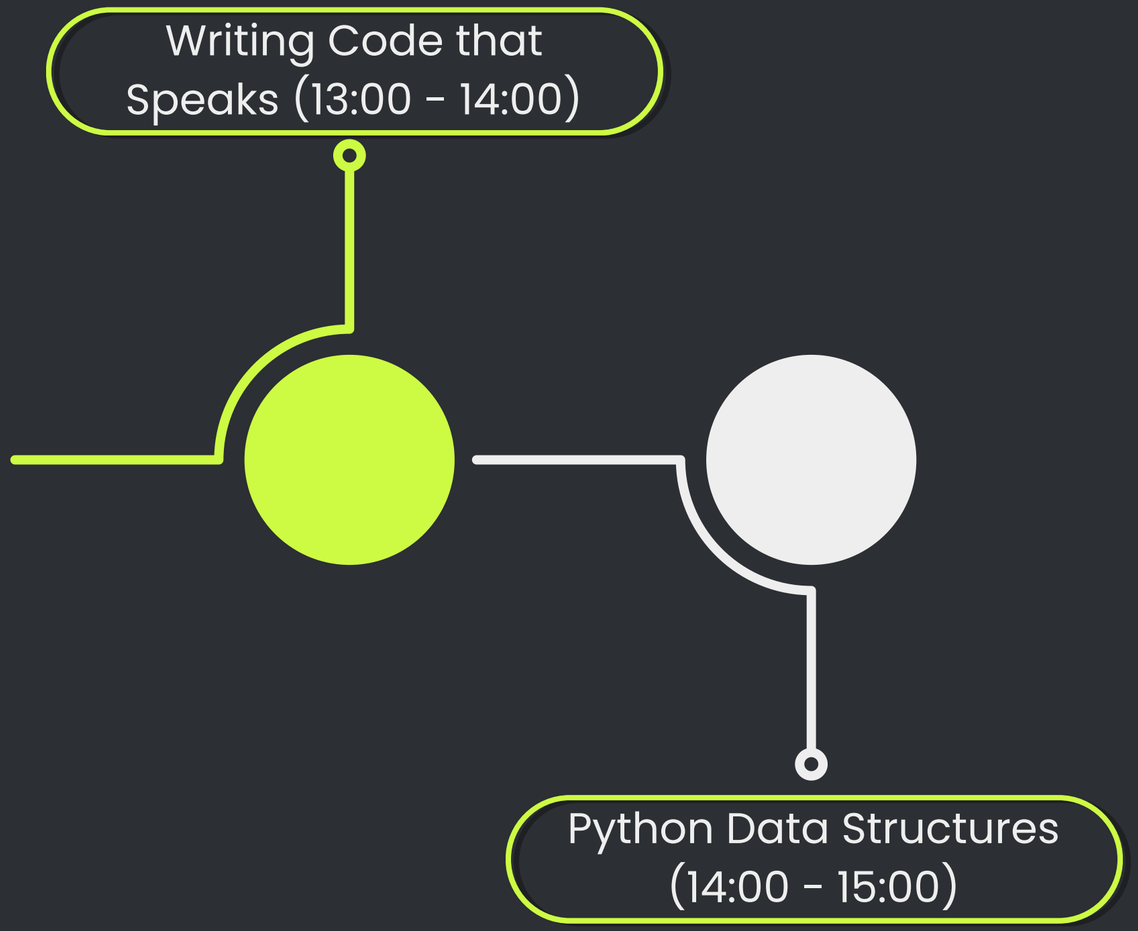
Hands-on activity:



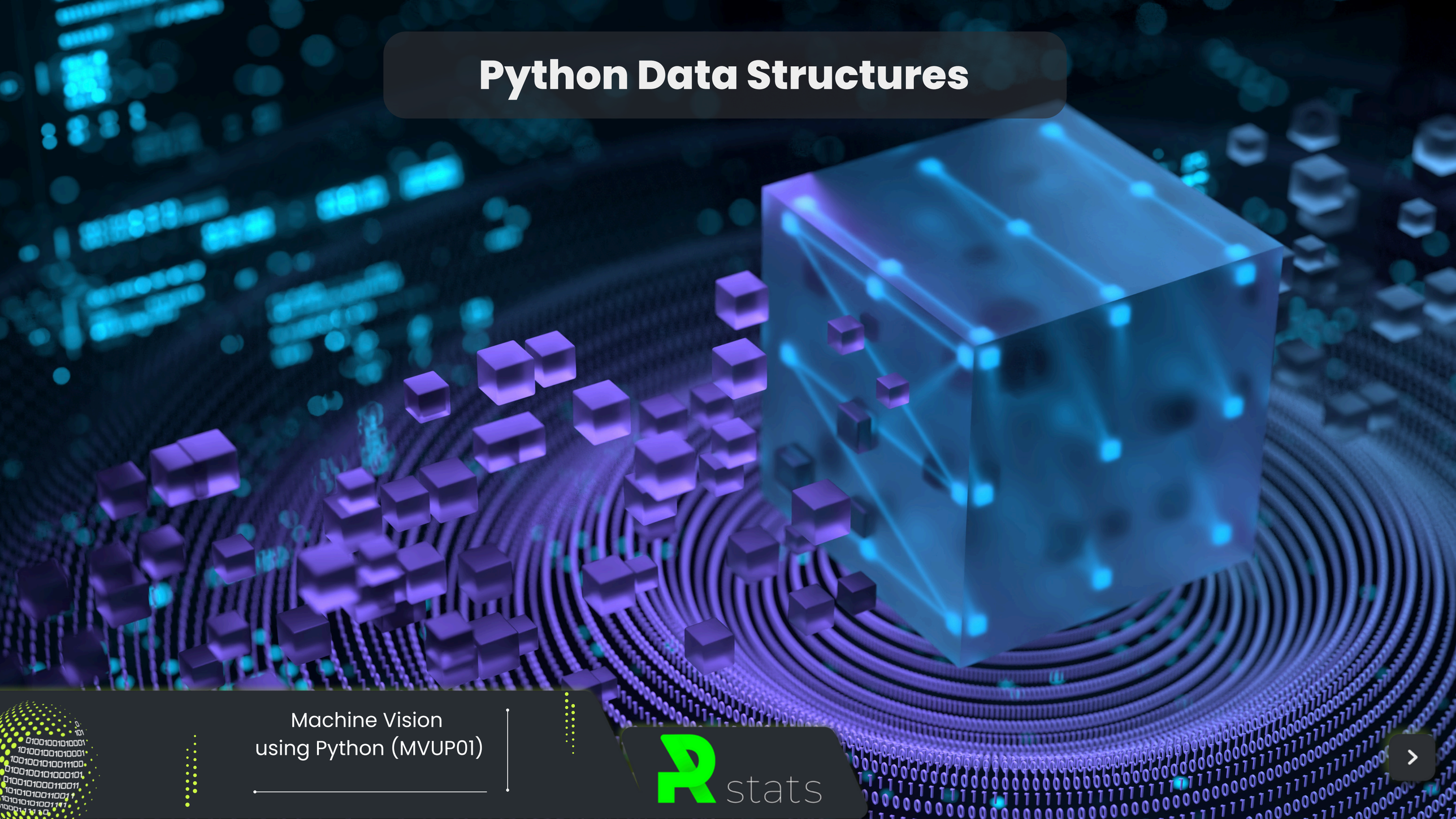
Machine Vision
using Python (MVUP01)

stats

Afternoon Session (13:00 – 17:30)



Python Data Structures



Machine Vision
using Python (MVUP01)

R stats

Python Data Structures

Machine Vision
using Python (MVUP01)

stats

Python Data Structures

List operations and methods

```
# Basic List Creation and Access
species = ["Wolf", "Bear", "Eagle"]
print(f"First species: {species[0]}")
print(f"Last species: {species[-1]}")
```

Python Data Structures

List operations and methods

```
# Basic List Creation and Access
species = ["Wolf", "Bear", "Eagle"]
print(f"First species: {species[0]}")
print(f"Last species: {species[-1]}")
```

```
First species: Wolf
Last species: Eagle
```

Python Data Structures

List operations and methods

```
# Adding Species (append and insert)
species.append("Deer")
species.insert(0, "Fox")
print(f"Updated list: {species}")
```

Python Data Structures

List operations and methods

```
# Adding Species (append and insert)
species.append("Deer")
species.insert(0, "Fox")
print(f"Updated list: {species}")
```

```
Updated list: ['Fox', 'Wolf', 'Bear', 'Eagle', 'Deer']
```

Python Data Structures

List operations and methods

```
# Removing Species (remove and pop)
species.remove("Bear")
last_species = species.pop()
print(f"After removal: {species}")
```

Python Data Structures

List operations and methods

```
# Removing Species (remove and pop)
species.remove("Bear")
last_species = species.pop()
print(f"After removal: {species}")
```

```
After removal: ['Fox', 'Wolf', 'Eagle']
```

Python Data Structures

List operations and methods

```
# List Slicing
first_two = species[:2]
print(f"First two species: {first_two}")
```

Python Data Structures

List operations and methods

```
# List Slicing
first_two = species[:2]
print(f"First two species: {first_two}")
```

```
First two species: ['Fox', 'Wolf']
```

Python Data Structures

List operations and methods

```
# List Methods
species = ["Wolf", "Bear", "Wolf", "Eagle"]
unique_species = set(species)
species_count = species.count("Wolf")
print(f"Unique species: {unique_species}")
print(f"Wolf count: {species_count}")
```

Python Data Structures

List operations and methods

```
# List Methods
species = ["Wolf", "Bear", "Wolf", "Eagle"]
unique_species = set(species)
species_count = species.count("Wolf")
print(f"Unique species: {unique_species}")
print(f"Wolf count: {species_count}")
```

```
Unique species: {'Bear', 'Eagle', 'Wolf'}
Wolf count: 2
```

Python Data Structures

List operations and methods

```
# Sorting
species.sort()
print(f"Sorted species: {species}")
```

Python Data Structures

List operations and methods

```
# Sorting
species.sort()
print(f"Sorted species: {species}")
```

```
Sorted species: ['Bear', 'Eagle', 'Wolf', 'Wolf']
```

Python Data Structures

Dictionary key-value relationships

```
# Example 1: Species Conservation Status
conservation_status = {
    "Amur Leopard": "Critically Endangered",
    "Black Rhino": "Endangered",
    "Giant Panda": "Vulnerable"
}

# Accessing and modifying
print(f"Panda status: {conservation_status['Giant Panda']}")
conservation_status['Black Rhino'] = "Critically Endangered"
print(f"Updated status: {conservation_status}")
```

Python Data Structures

Dictionary key-value relationships

```
# Example 1: Species Conservation Status
conservation_status = {
    "Amur Leopard": "Critically Endangered",
    "Black Rhino": "Endangered",
    "Giant Panda": "Vulnerable"
}

# Accessing and modifying
print(f"Panda status: {conservation_status['Giant Panda']}")
conservation_status['Black Rhino'] = "Critically Endangered"
print(f"Updated status: {conservation_status}")
```

```
Panda status: Vulnerable
Updated status: {'Amur Leopard': 'Critically Endangered', 'Black Rhino': 'Critically Endangered',
'Giant Panda': 'Vulnerable'}
```

Python Data Structures

Dictionary key-value relationships

```
# Example 2: Habitat Temperature Ranges
temperature_ranges = {
    "Desert": {"min": 20, "max": 45},
    "Rainforest": {"min": 15, "max": 35},
    "Tundra": {"min": -40, "max": 10}
}

# Accessing nested data
for habitat, temps in temperature_ranges.items():
    print(f"{habitat}: {temps['min']}°C to {temps['max']}°C")
```

Python Data Structures

Dictionary key-value relationships

```
# Example 2: Habitat Temperature Ranges
temperature_ranges = {
    "Desert": {"min": 20, "max": 45},
    "Rainforest": {"min": 15, "max": 35},
    "Tundra": {"min": -40, "max": 10}
}

# Accessing nested data
for habitat, temps in temperature_ranges.items():
    print(f"{habitat}: {temps['min']}°C to {temps['max']}°C")
```

```
Desert: 20°C to 45°C
Rainforest: 15°C to 35°C
Tundra: -40°C to 10°C
```

Python Data Structures

Nested data structures

```
# Example 1: Species Data with Location and Population
# Structure: (species_name, (latitude, longitude), population_count)
species_data = (
    "Tiger",
    (-2.5, 120.5),  # Geographic coordinates
    350
)

# Accessing nested tuple data
species_name, (lat, long), population = species_data
print(f"Species: {species_name}")
print(f"Location: {lat}°N, {long}°E")
print(f"Population: {population}")
```

Python Data Structures

Nested data structures

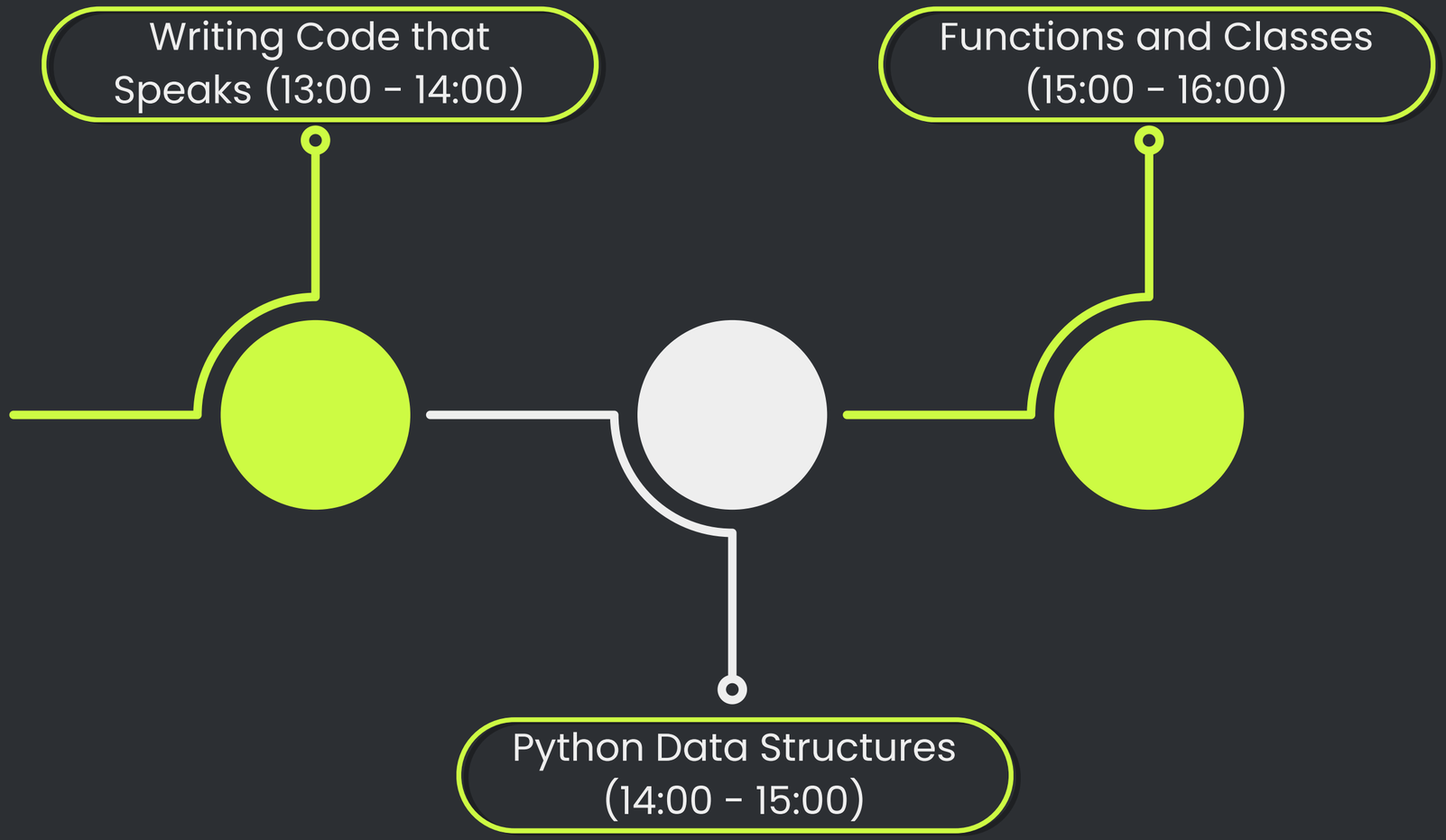
```
# Example 1: Species Data with Location and Population
# Structure: (species_name, (latitude, longitude), population_count)
species_data = (
    "Tiger",
    (-2.5, 120.5),  # Geographic coordinates
    350
)

# Accessing nested tuple data
species_name, (lat, long), population = species_data
print(f"Species: {species_name}")
print(f"Location: {lat}°N, {long}°E")
print(f"Population: {population}")
```

```
Species: Tiger
Location: -2.5°N, 120.5°E
Population: 350
```

Python Data Structures

Data Structure	Syntax	Immutable	Allow Duplicate Values	Ordered	Subset with []
List	[1, 2, 3]	No	Yes	Yes	Yes - index
Dictionary	{key:value}	No	Yes	Yes	Yes - key
Set	{1, 2, 3}	No	No	No	No
Tuple	(1, 2, 3)	Yes	Yes	Yes	Yes - index

Afternoon Session (13:00 – 17:30)



Functions and Classes

Machine Vision
using Python (MVUP01)

R stats

Functions and Classes

Creating functions:

```
def analyze_species_observation(species_name, count, temperature):
    """
    Analyze single species observation data.
    """
    status = "Common" if count > 10 else "Rare"

    if temperature < 20:
        condition  "Cold"
    elif temperature > 30:
        condition = "Hot"
    else:
        condition = "Optimal"

    return f"Species {species_name}: {status} ({count} individuals) in {condition} conditions"

# Using the function
print("Function Example:")
result = analyze_species_observation("Monarch Butterfly", 5, 25)
print(result)
```

Functions and Classes

Creating functions:

```
def analyze_species_observation(species_name, count, temperature):
    """
    Analyze single species observation data.
    """
    status = "Common" if count > 10 else "Rare"

    if temperature < 20:
        condition  "Cold"
    elif temperature > 30:
        condition = "Hot"
    else:
        condition = "Optimal"

    return f"Species {species_name}: {status} ({count} individuals) in {condition} conditions"

# Using the function
print("Function Example:")
result = analyze_species_observation("Monarch Butterfly", 5, 25)
print(result)
```

```
Function Example:
Species Monarch Butterfly: Rare (5 individuals) in Optimal conditions
```

Functions and Classes

Creating functions:

```
def analyze_species_observation(species_name, count, temperature):
    """
    Analyze single species observation data.
    """
    status = "Common" if count > 10 else "Rare"

    if temperature < 20:
        condition  "Cold"
    elif temperature > 30:
        condition = "Hot"
    else:
        condition = "Optimal"

    return f"Species {species_name}: {status} ({count} individuals) in {condition} conditions"

# Using the function
print("Function Example:")
result = analyze_species_observation("Monarch Butterfly", 5, 25)
print(result)
```

Functions and Classes

Creating functions:

```
def analyze_species_observation(species_name, count, temperature):
    """
    Analyze single species observation data.
    """
    status = "Common" if count > 10 else "Rare"

    if temperature < 20:
        condition  "Cold"
    elif temperature > 30:
        condition = "Hot"
    else:
        condition = "Optimal"

    return f"Species {species_name}: {status} ({count} individuals) in {condition} conditions"

# Using the function
print("Function Example:")
result = analyze_species_observation("Monarch Butterfly", 5, 25)
print(result)
```

```
Function Example:
Species Monarch Butterfly: Rare (5 individuals) in Optimal conditions
```

Functions and Classes

Machine Vision
using Python (MVUP01)

stats

Functions and Classes

Creating classes:

```
# Then: Class-based approach
class SpeciesObservation:
    """
    Class to handle species observation data and analysis.
    """
    def __init__(self, species_name, count, temperature):
        self.species_name = species_name
        self.count = count
        self.temperature = temperature

    def get_status(self):
        return "Common" if self.count > 10 else "Rare"

    def get_condition(self):
        if self.temperature < 20:
            return "Cold"
        elif self.temperature > 30:
            return "Hot"
        return "Optimal"

    def analyze(self):
        status = self.get_status()
        condition   self.get_condition()
        return f"Species {self.species_name}: {status} ({self.count} individuals) in {condition}
conditions"
```

Functions and Classes

Creating classes:

```
# Using the class
print("\nClass Example:")
observation = SpeciesObservation("Monarch Butterfly", 5, 25)
result = observation.analyze()
print(result)
```

```
Class Example:
Species Monarch Butterfly: Rare (5 individuals) in Optimal conditions
```

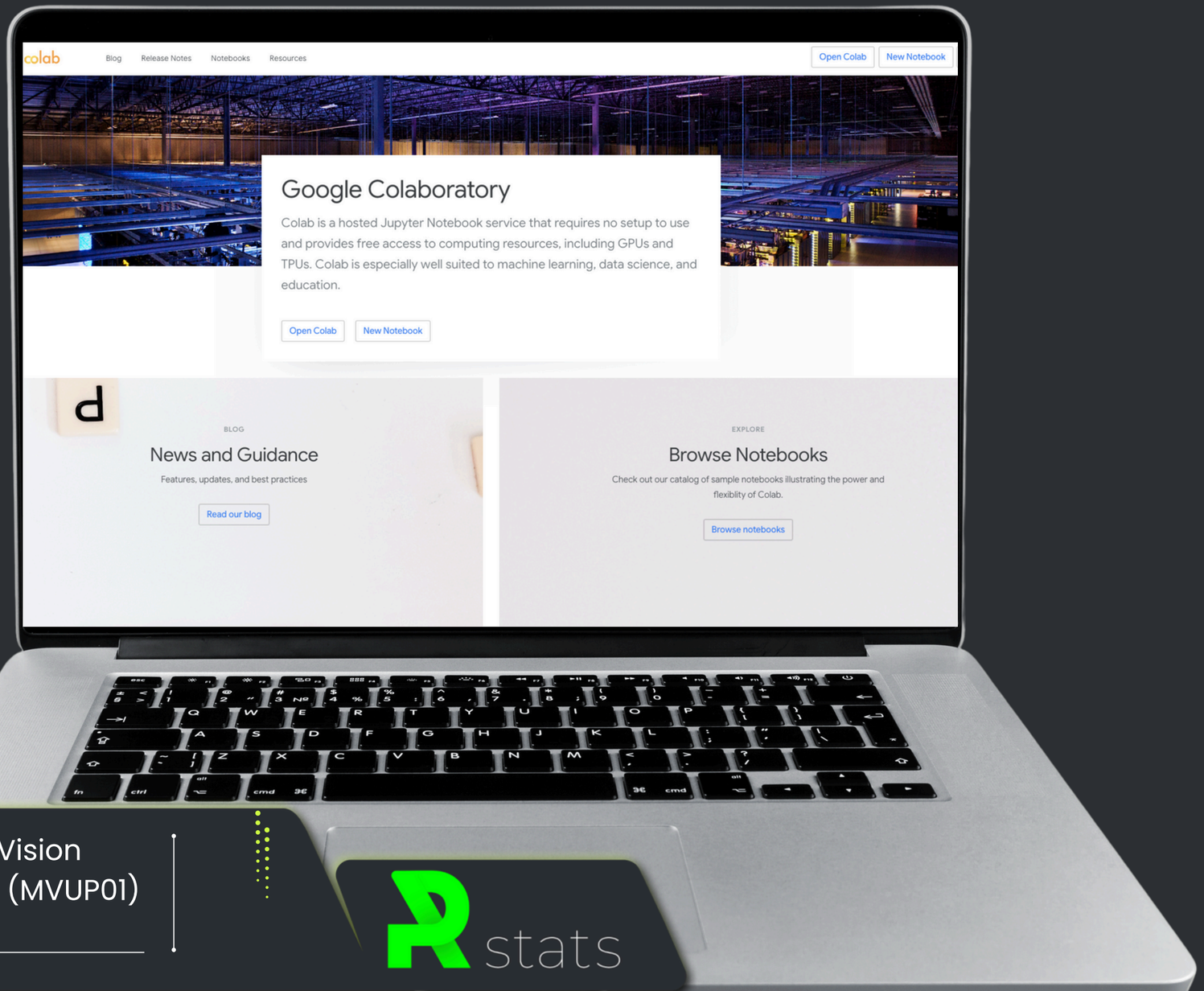
Functions and Classes

Creating a function description

```
def analyze_species_observation(species_name, count, temperature):
    """
    Analyze species observation data and return a formatted status report.

    Parameters
    ----------
    species_name
        The scientific or common name of the observed species
    count
        Number of individuals observed
    temperature
        Ambient temperature in Celsius during observation

    Returns
    -------
    str
        A formatted string containing species status, count, and environmental conditions

    Examples
    --------
    >>> analyze_species_observation("Monarch Butterfly", 5, 25)
    'Species Monarch Butterfly: Rare (5 individuals) in Optimal conditions'

    Notes
    -----
    Status criteria:
        - Common: > 10 individuals
        - Rare: <= 10 individuals

    Temperature conditions:
        - Cold: < 20°C
        - Optimal: 20-30°C
        - Hot: > 30°C
    """
```

Making Decisions in Code

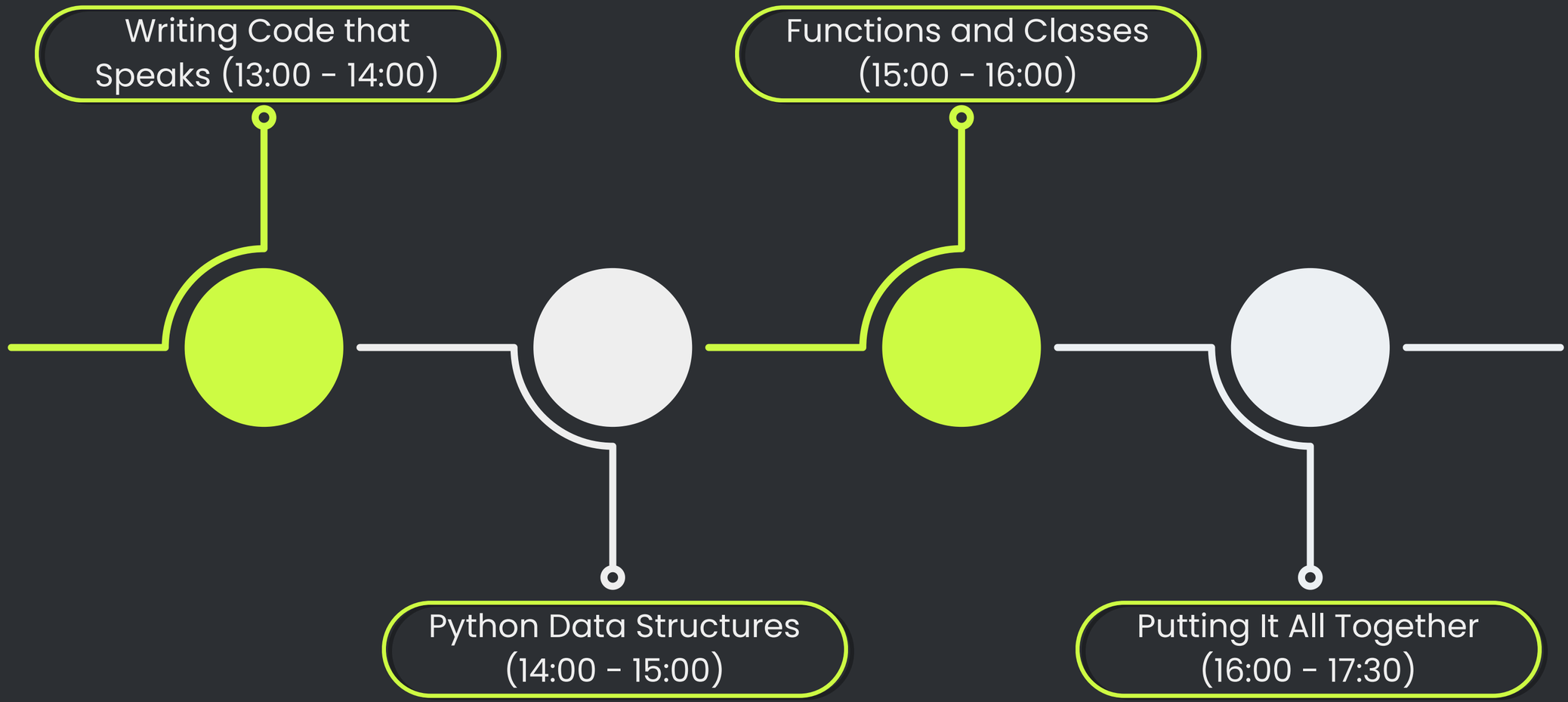
Hands-on activity:



Machine Vision
using Python (MVUP01)

stats

Afternoon Session (13:00 – 17:30)



Putting It All Together

Machine Vision
using Python (MVUP01)

stats