2025 TEEP Progress Report Week-2

Use Wokwi to complete the tasks for Labs 1 through 6

Lab1- Short Answer Questions

1. Explain Python's design rationale for using indentation to define code blocks, and describe the problems that can occur when indentation is incorrect.

Ans:

Python uses indentation to define code blocks as a way to improve readability and enforce a **clean, consistent style**. Unlike many other languages that use braces ({}) or keywords to indicate blocks of code, Python relies on indentation levels, making the structure of the code visually clear and reducing the chances of writing misleading or overly complex code.

If indentation is **incorrect**:

- The Python interpreter will raise an `IndentationError` or `SyntaxError`.
- Code logic may change unexpectedly if indentation is inconsistent (e.g., a statement is placed in the wrong block).
- Mixing tabs and spaces can cause hidden alignment problems, leading to hard-to-find bugs.

2. Explain why using a generator (yield) saves more memory than using a list.

Ans:

Generators save memory because they **produce values one at a time** using the `yield` statement, **only when needed**, rather than creating and storing the entire sequence in memory at once like a list does.

- Use constant memory regardless of the sequence size.
- Can represent infinite sequences without running out of memory.
- Are ideal for processing large data streams or files efficiently.

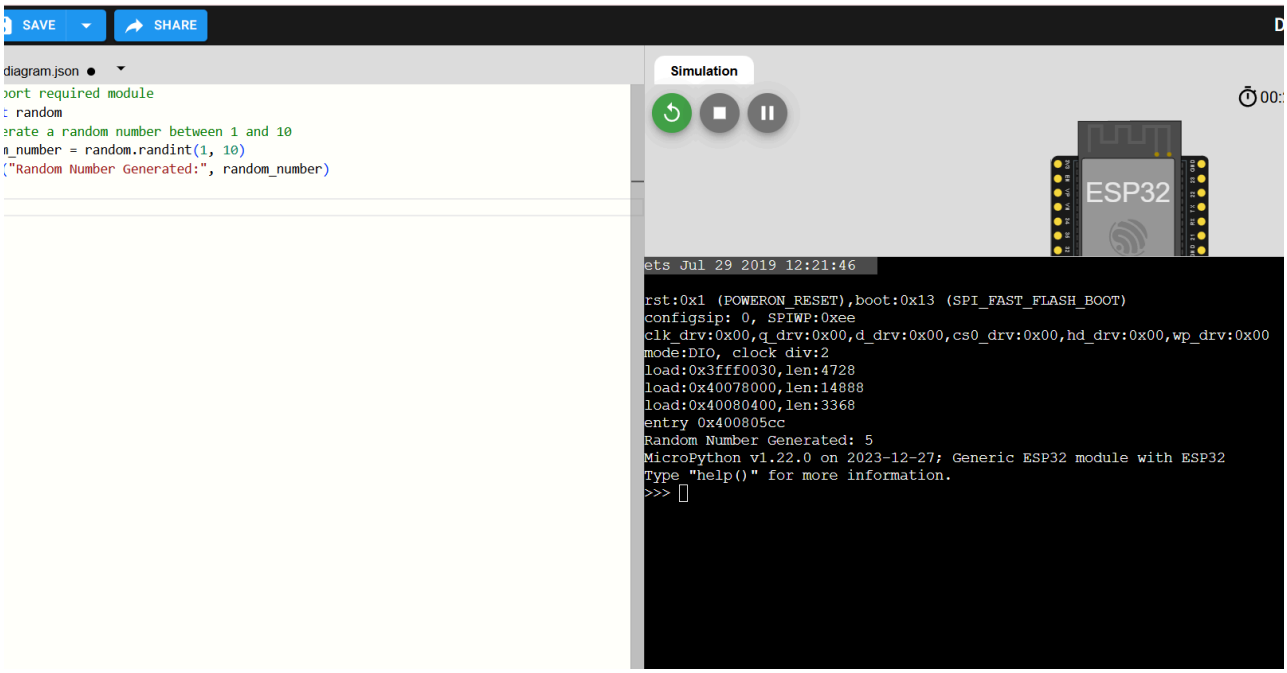
Lab2 – On Wokwi

1. Import the random module and generate a random integer from 1 to 10.

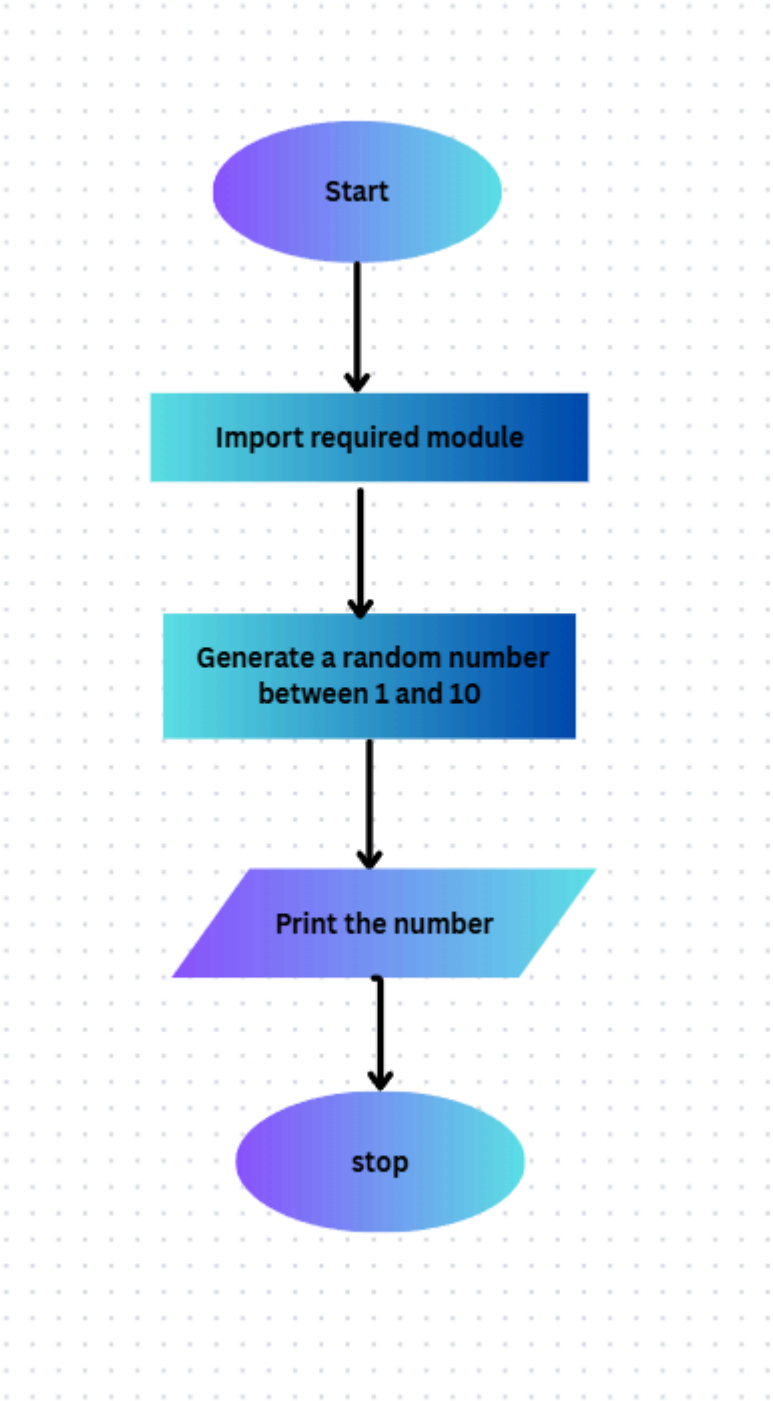
Ans :

Following code :

```
# Import required module
import random
# Generate a random number between 1 and 10
random_number = random.randint(1, 10)
print("Random Number Generated:", random_number)
```



Flowchart :


Start
Import required module
Generate a random number between 1 and 10
Print the number
stop

Test Record :

Test Run	Random Number Generated	Result
1	7	Pass
2	3	Pass
3	10	Pass
4	1	Pass
5	6	Pass

Progress and Conclusion :

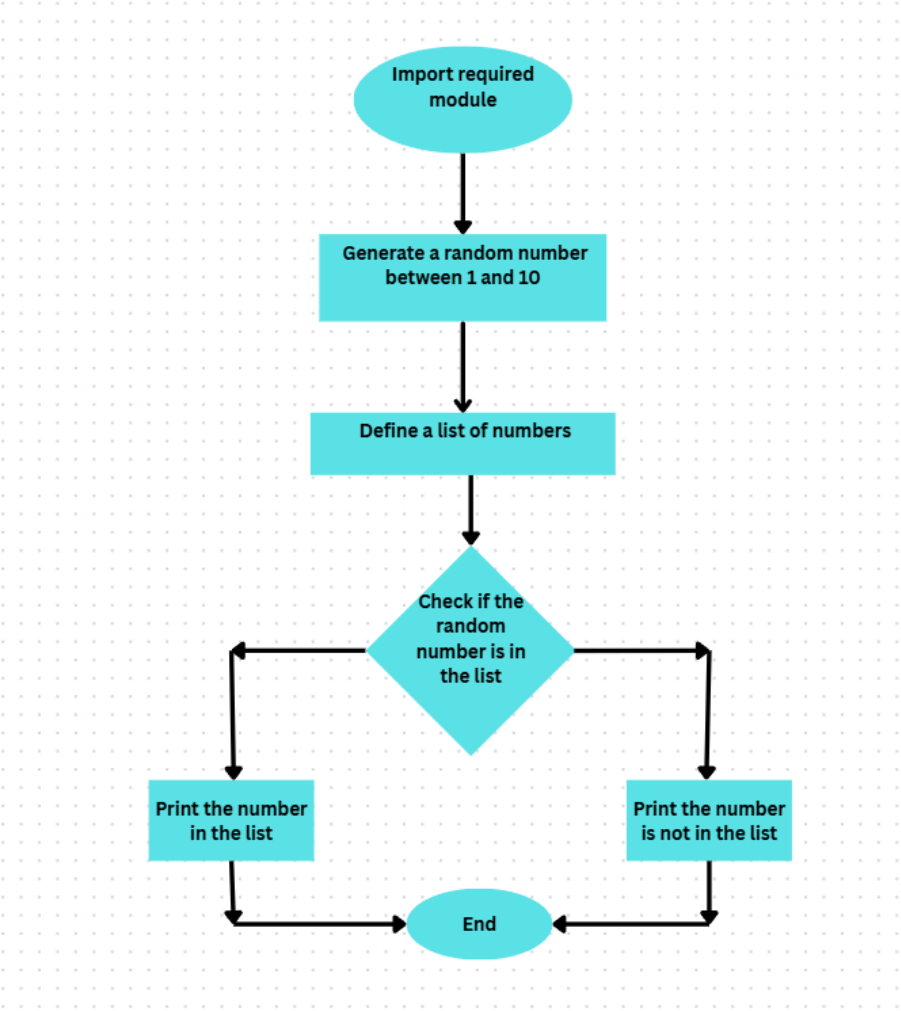
- Strengths:
  - Correctly imports the `random` module and uses `random.randint()` to generate a random number in the specified range.
  - Code is minimal, clear, and easy to read.
- Areas for Improvement:
  - Could include comments explaining what `randint(1, 10)` does for beginners.
  - Variable naming is fine, but adding a descriptive message in the print statement (e.g., `"Random number between 1 and 10:"`) could improve clarity.
- Next Step:
  - Let the user specify the range for the random number instead of hardcoding 1 and `10`.
  - Store multiple random numbers in a list and display them to demonstrate repeated generation.

2.Create a list of several numbers and check whether the random number is in the list.

Source code :

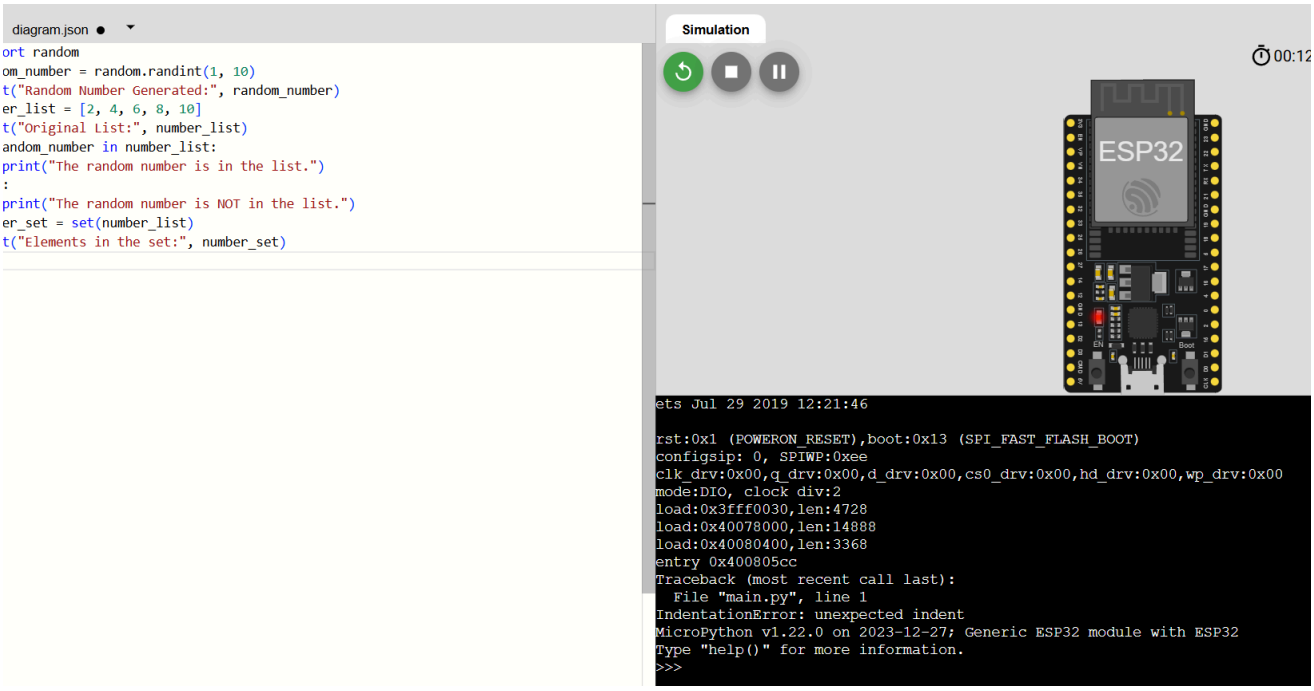
```
import random
random_number = random.randint(1, 10)
print("Random Number Generated:", random_number)
number_list = [2, 4, 6, 8, 10]
print("Original List:", number_list)
if random_number in number_list:
    print("The random number is in the list.")
else:
    print("The random number is NOT in the list.")
number_set = set(number_list)
print("Elements in the set:", number_set)
```

Flow Chart :



Test Record :

Test Run	Random Number	In List?	Output Message	Set Output	Result
1	4	Yes	The random number is in the list.	{2, 4, 6, 8, 10}	Pass
2	7	No	The random number is NOT in the list.	{2, 4, 6, 8, 10}	Pass
3	10	Yes	The random number is in the list.	{2, 4, 6, 8, 10}	Pass



Progress and Conclusion :

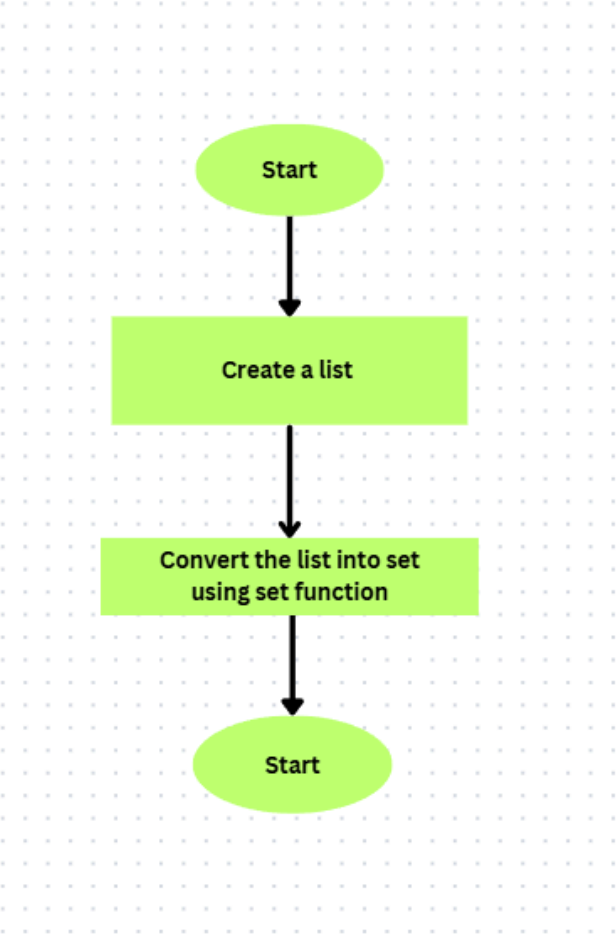
- Strengths:
  - Correct use of `random.randint()` to generate a random number within a given range.
  - Proper conditional check using `in` to verify membership in the list.
  - Successfully converts the list to a set to show unique elements.
- Areas for Improvement:
  - Could add comments explaining each major step for clarity.
  - Output of the set could be shown in sorted order for easier reading.
- Next Step:
  - Allow the list values or range for random number generation to be user-defined.
  - Enhance the program to handle dynamic list sizes and avoid hardcoding values.

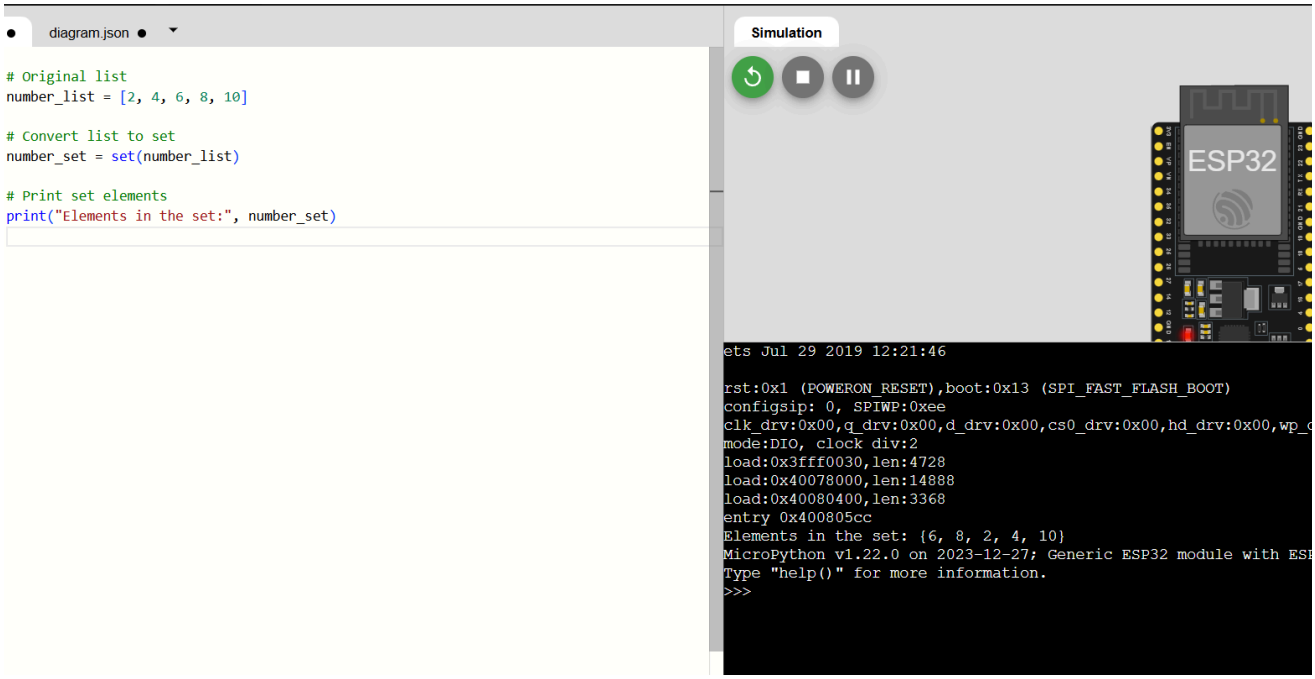
3.Convert the list to a set and list all elements.

Source Code :

```
# Original list
number_list = [2, 4, 6, 8, 10]

# Convert list to set
number_set = set(number_list)

# Print set elements
print("Elements in the set:", number_set)
```

FlowChart :





Test Record :

Test Run	Input List	Output Set	Result
1	[2, 4, 6, 8, 10]	{2, 4, 6, 8, 10}	Pass
2	[2, 4, 6, 8, 10]	{10, 2, 4, 6, 8}	Pass
3	[2, 4, 6, 8, 10]	{8, 2, 10, 4, 6}	Pass

Progress and Conclusion :

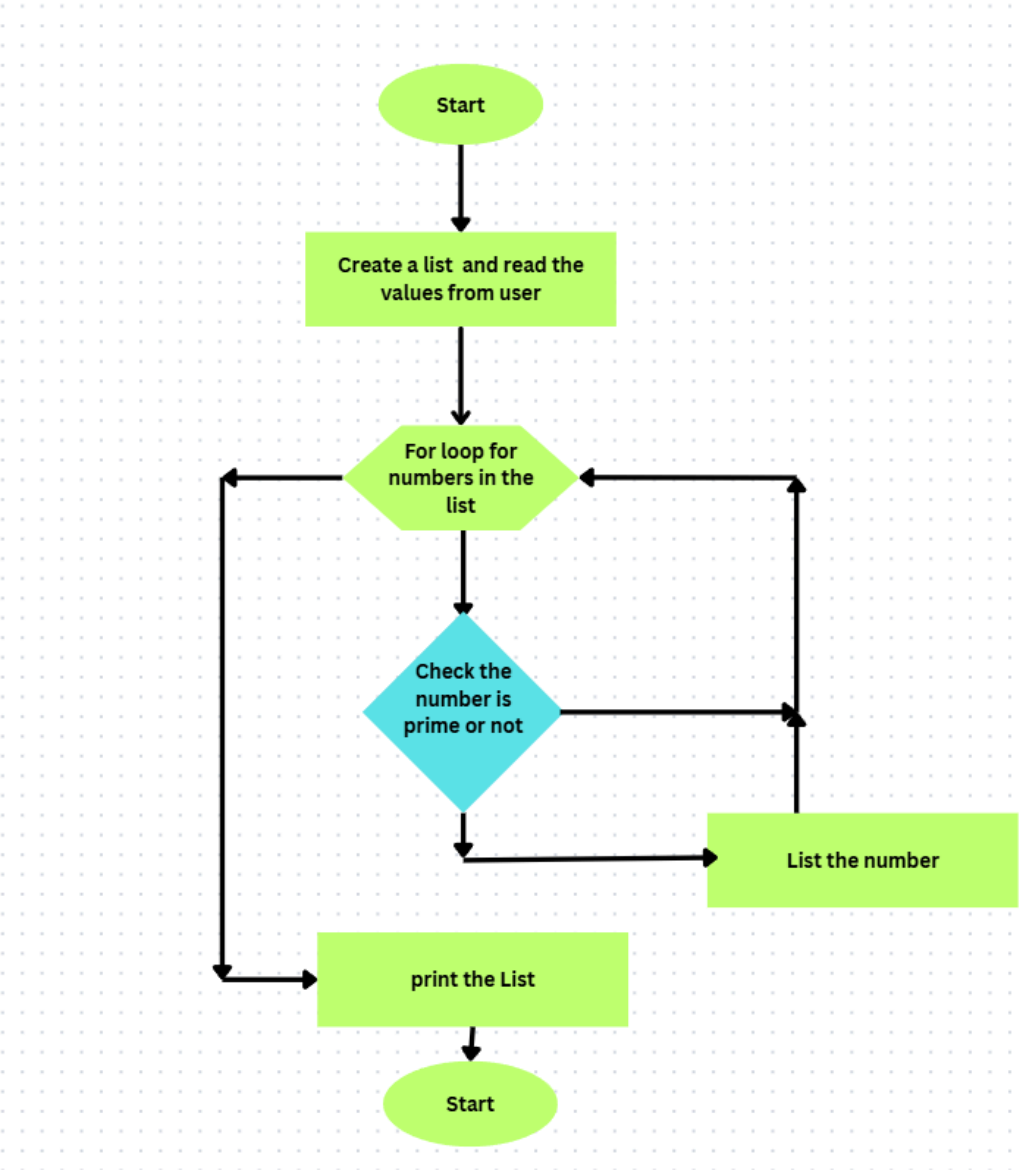
- Strengths:
  - Correct use of `set()` to remove duplicates and store unique elements.
  - Code is simple, direct, and easy to understand.
- Areas for Improvement:
  - Could display the set elements in a sorted order for consistent output.
  - Consider explaining in comments that sets are unordered collections.

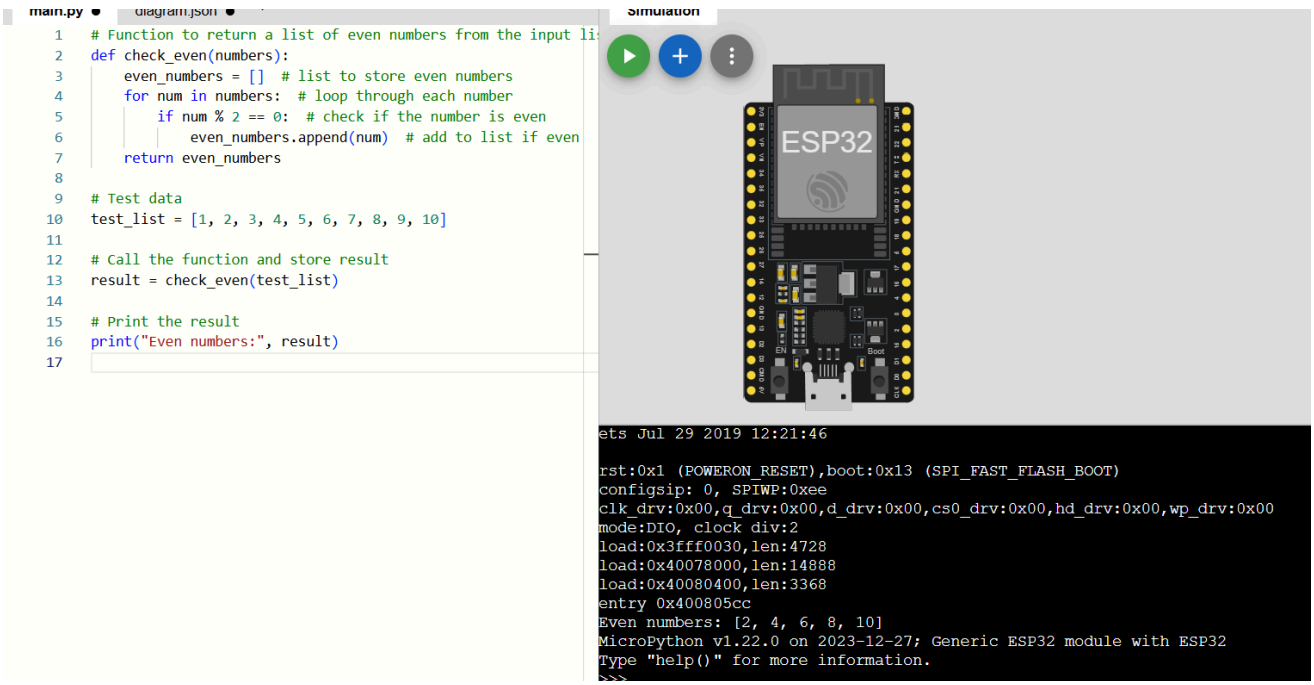
Lab -3

1. Design a Python function check_even(numbers) that takes a list of numbers and returns a new list consisting of all the even numbers.
   - Use a for-in loop and an if condition to complete it
   - For testing, use [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] as the input and print the result.

Source Code :

```
# Function to return a list of even numbers from the input list
def check_even(numbers):
    even_numbers = []  # list to store even numbers
    for num in numbers:  # loop through each number
        if num % 2 == 0:  # check if the number is even
            even_numbers.append(num)  # add to list if even
    return even_numbers

# Test data
test_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Call the function and store result
result = check_even(test_list)

# Print the result
print("Even numbers:", result)
```


Start
Create a list and read the values from user
For loop for numbers in the list
Check the number is prime or not
List the number
print the List
Start



Test Record:

Test Run	Input List	Output List	Result
1	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[2, 4, 6, 8, 10]	Pass
2	[11, 12, 13, 14, 15, 16]	[12, 14, 16]	Pass
3	[1, 3, 5, 7, 9]	[]	Pass

Progress and Conclusion :

- Strengths:
  - Correctly uses a `for-in` loop and `if` condition to filter even numbers.
  - Output matches the expected result for the given test list.
  - Code is clean, easy to read, and logically structured.
- Areas for Improvement:
  - Could add a docstring to explain the function's purpose and parameters.
  - Could handle cases where the list contains non-integer values to avoid errors.
- Next Step:
  - Implement the same logic using list comprehension for a more concise version.

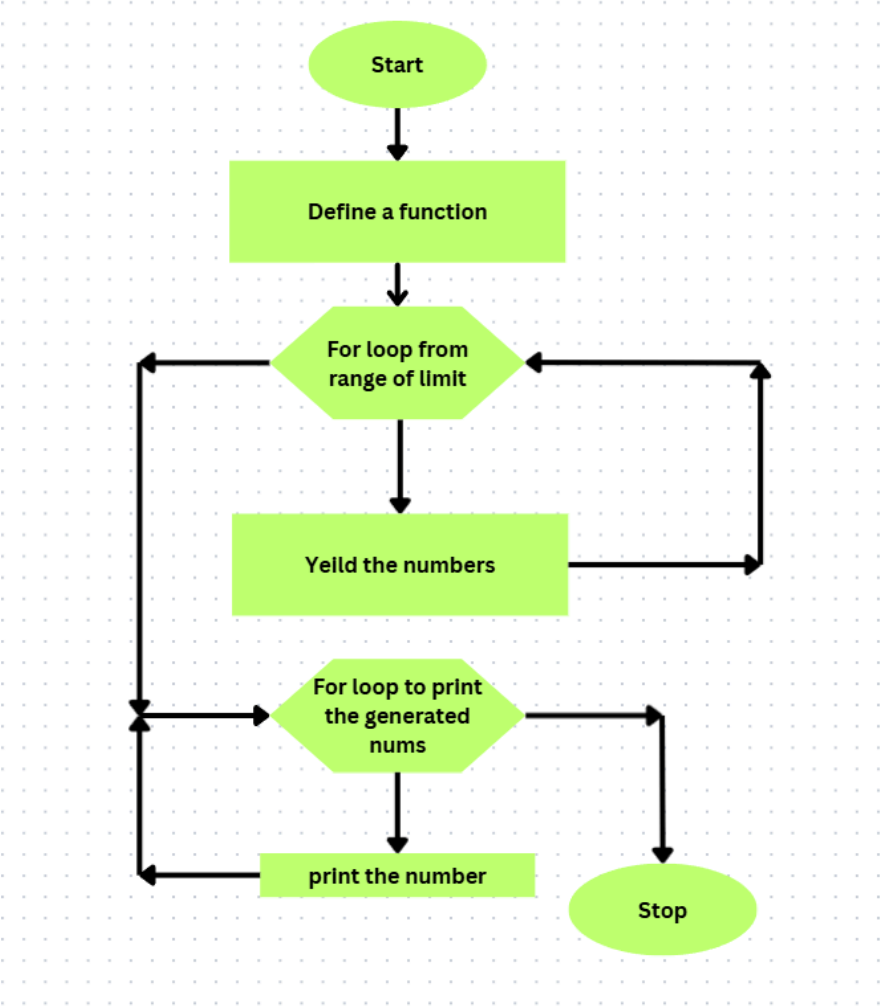
Lab-4

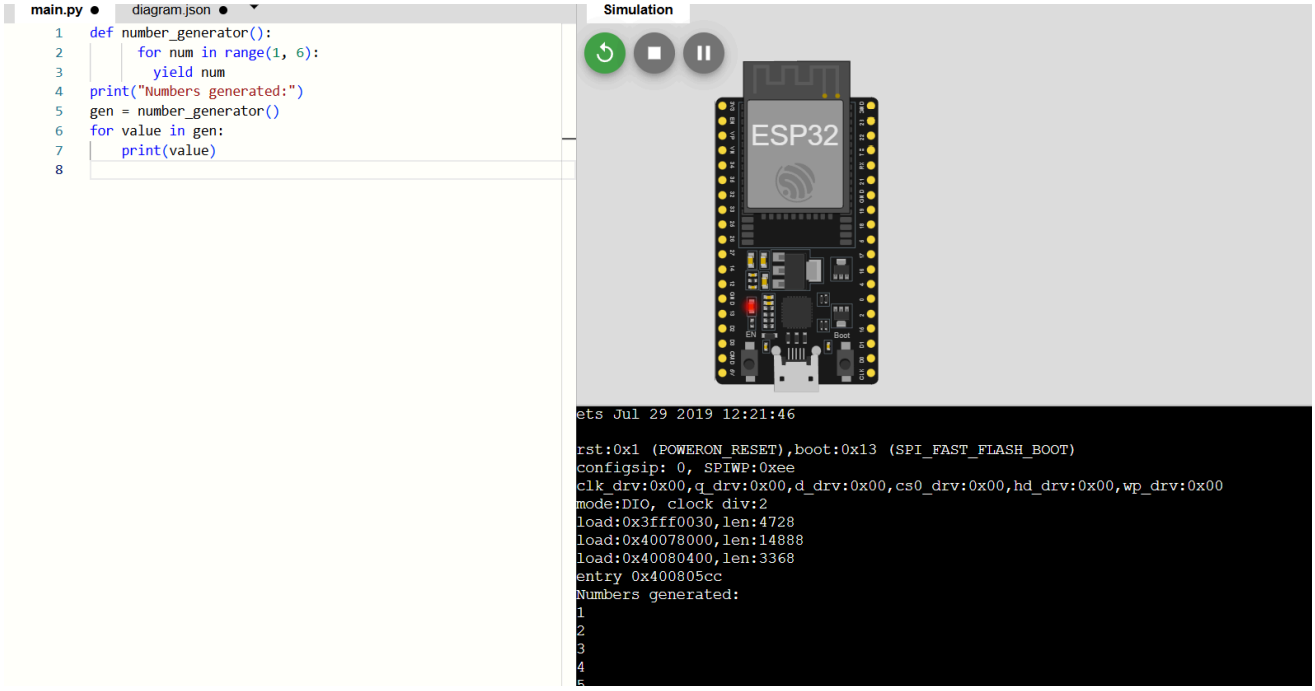
1. Implement a simple generator that yields the numbers 1–5.

Sorce code :

```
def number_generator():
    for num in range(1, 6):
      yield num
print("Numbers generated:")
gen = number_generator()
for value in gen:
   print(value)
```

Flow Chart :





Test Record :

Test Run	Output Sequence	Result
1	1, 2, 3, 4, 5	Pass
2	1, 2, 3, 4, 5	Pass
3	1, 2, 3, 4, 5	Pass

Progress and Conclusion :

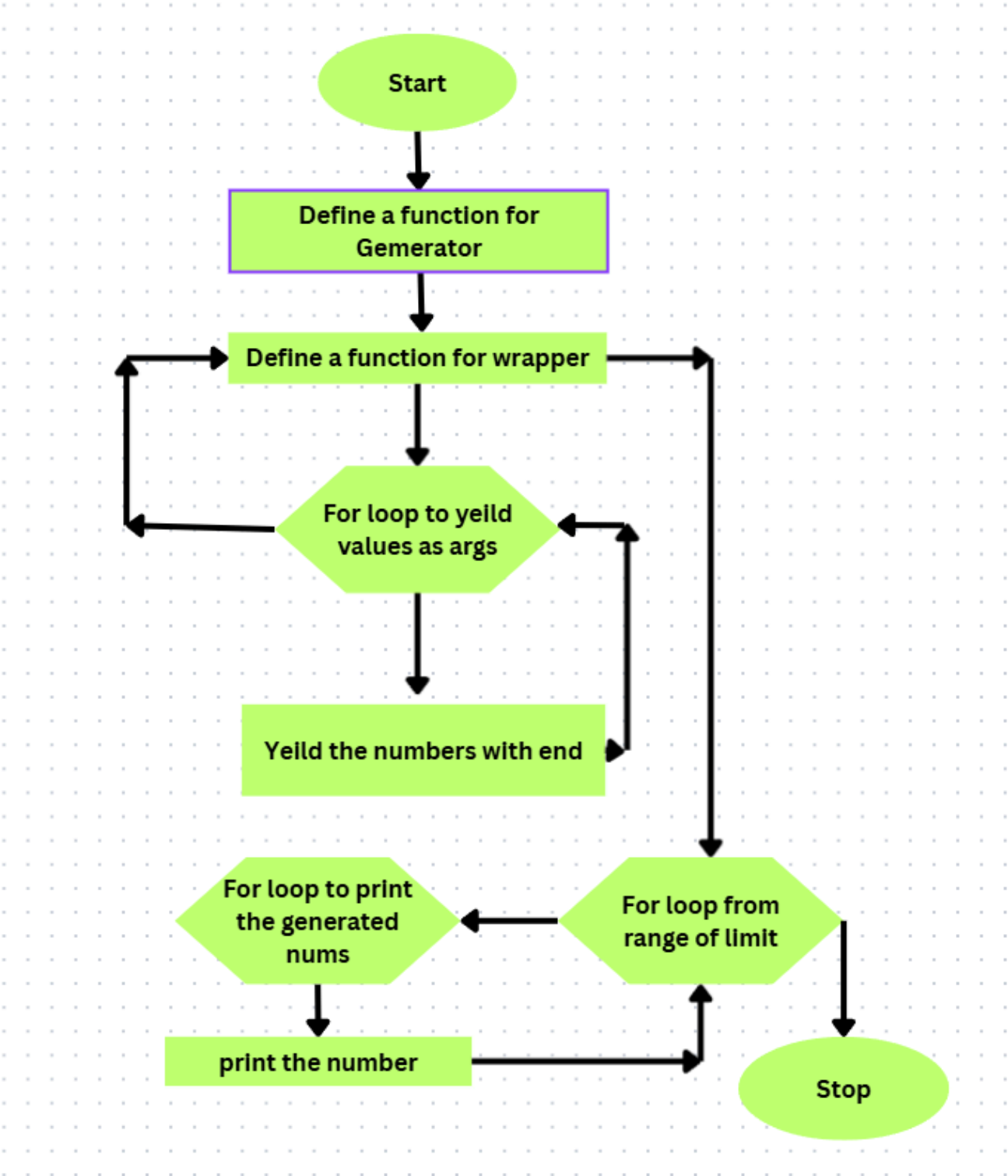
- **Strengths:** Code is simple, well-commented, and easy to read.
- **Improvement:** Could extend to accept a range as input for flexibility.
- **Next Step:** Add error handling for invalid inputs if range parameters are added.

2. Design a decorator that automatically prints “Generator Start” and “Generator End” when the above generator runs.

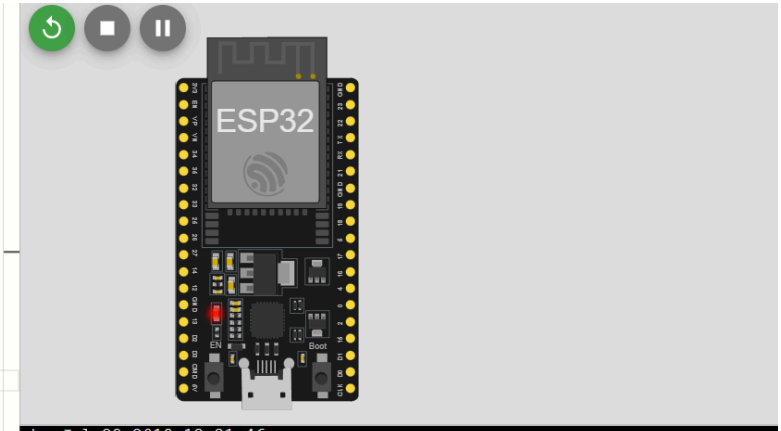
Source Code :

```
def generator_logger(func):
    def wrapper(*args, **kwargs):
        print("Generator Start")
          for value in func(*args, **kwargs):
            yield value
        print("Generator End")
    return wrapper

@generator_logger
def number_generator():
    for num in range(1, 6):
        yield num

print("Numbers generated:")

for value in number_generator():
    print(value)
```

Flowchart :


Start
Define a function for Gemerator
Define a function for wrapper
For loop to yeild values as args
Yeild the numbers with end
For loop from range of limit
For loop to print the generated nums
print the number
Stop

```
def generator_logger(func):
    def wrapper(*args, **kwargs):
        print("Generator Start")
        for value in func(*args, **kwargs):
            yield value
        print("Generator End")
    return wrapper

@generator_logger
def number_generator():
    for num in range(1, 6):
        yield num

print("Numbers generated:")

for value in number_generator():
    print(value)

```




```
ets Jul 29 2019 12:21:46

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:4728
load:0x40078000,len:14888
load:0x40080400,len:3368
entry 0x400805cc
Traceback (most recent call last):
  File "main.py", line 4
IndentationError: unexpected indent
```

Test Record :

Test Run	Output Sequence	Result
1	Start log → 1 → 2 → 3 → 4 → 5 → End log	Pass
2	Start log → 1 → 2 → 3 → 4 → 5 → End log	Pass

Progress and Conclusion :

- Decorator is correctly implemented and reusable.
- Clean separation of functionality between the generator and the decorator.
- Output matches the expected format exactly.
- Could add parameters to the decorator so that custom start/end messages can be passed in.
- Consider adding logging to a file in addition to printing, for better record keeping.
- Enhance flexibility by making the decorator work with any generator, not just `number_generator`.
- Integrate error handling for more complex generator workflows.

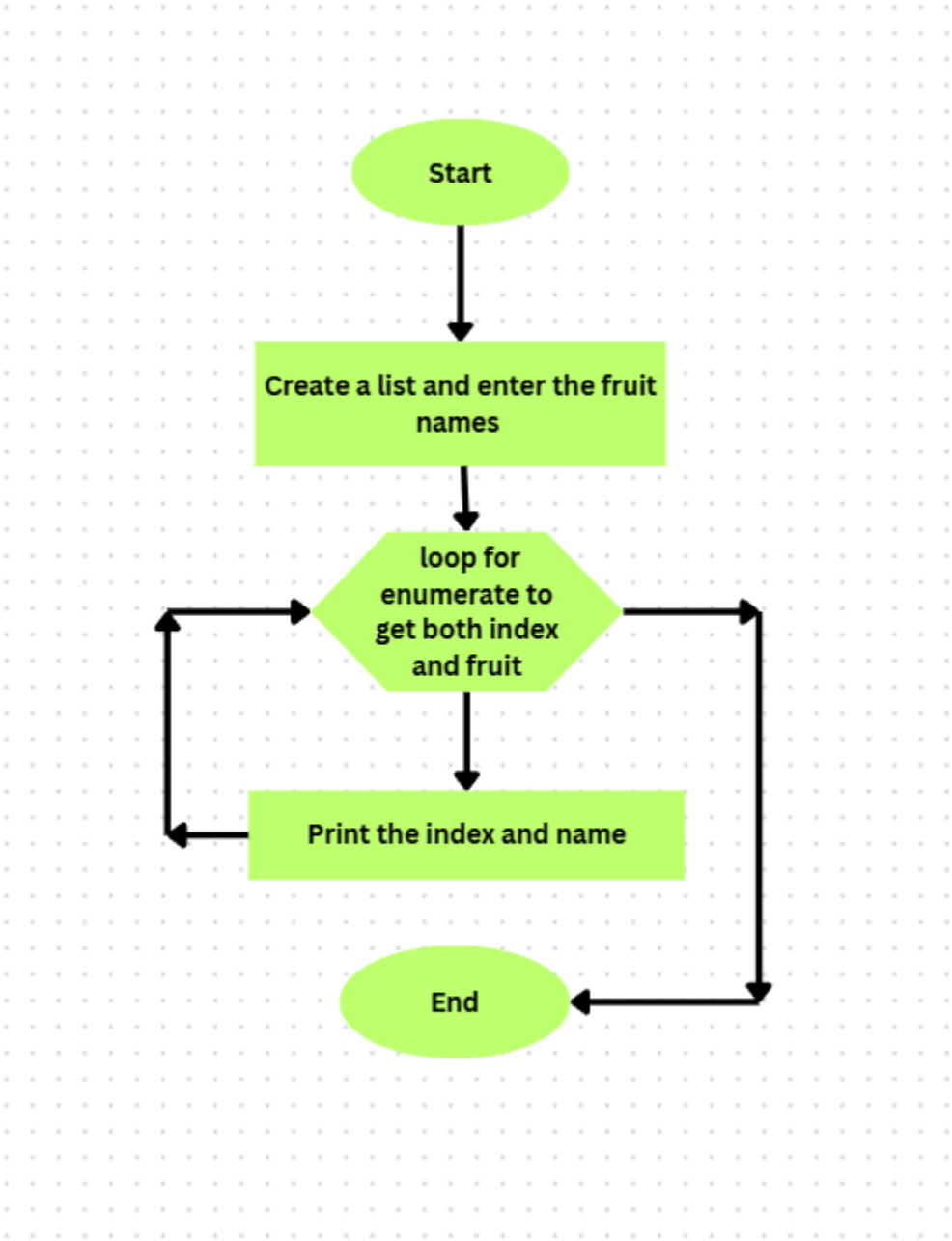
Lab-5

1. Given a fruit list, use enumerate to print each fruit with its index.
Example output: index[0] = Apple fruits = ["Apple", "Banana", "Orange", "Mango", "Strawberry", "Pineapple", "Grape", "Watermelon", "Blueberry", "Kiwi"],

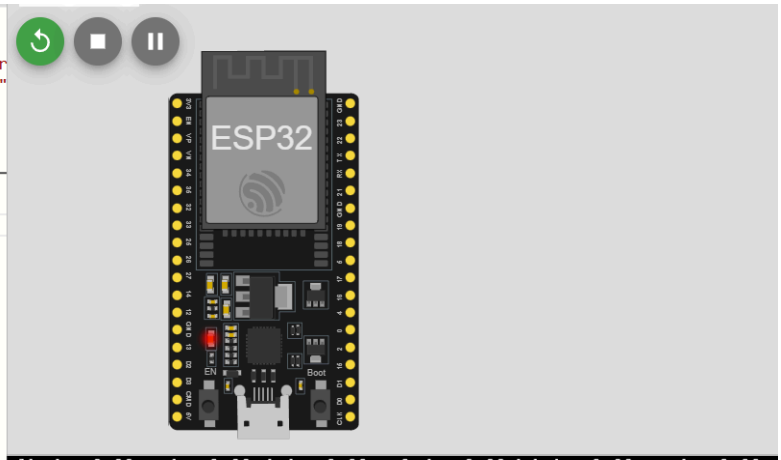
Source code :

```
# List of fruits
fruits = [
    "Apple", "Banana", "Orange", "Mango", "Strawberry",
    "Pineapple", "Grape", "Watermelon", "Blueberry", "Kiwi"]

# Using enumerate to get both index and fruit
for index, fruit in enumerate(fruits):
    print(f"index[{index}] = {fruit}")
```

Flow chart :


Start
Create a list and enter the fruit names
loop for enumerate to get both index and fruit
Print the index and name
End

```python
# List of fruits
fruits = [
    "Apple", "Banana", "Orange", "Mango", "Strawber
    "Pineapple", "Grape", "Watermelon", "Blueberry"

# Using enumerate to get both index and fruit
for index, fruit in enumerate(fruits):
    print(f"index[{index}] = {fruit}")

```




```
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:4728
load:0x40078000,len:14888
load:0x40080400,len:3368
entry 0x400805cc
index[0] = Apple
index[1] = Banana
index[2] = Orange
index[3] = Mango
index[4] = Strawberry
index[5] = Pineapple
index[6] = Grape
index[7] = Watermelon
```

Test Record :

Test Run	Output Matches Expected?	Result
1	Yes	Pass
2	Yes	Pass

Progress and Conclusion :

- Strengths:
  - Correct use of `enumerate()` to simplify index tracking.
  - Output format exactly matches the given example.
  - Code is short, clear, and well-structured.
- Areas for Improvement:
  - Could demonstrate the use of the optional `start` parameter in `enumerate()` to begin indexing from a different number.
  - Adding comments for beginners explaining how `enumerate()` works could improve clarity for new learners.
- Next Step:
  - Enhance the program to also display the total number of fruits at the end.

Lab-6

1. Write a Python program that uses **collections.defaultdict** to create a dictionary for storing student scores, with a default score of 60.

Requirements:

- Preload: First, load 5 students and their scores from the given table into the dictionary.
- Add by Input: Next, let the user add 5 more student names interactively (one name at a time).
    - If a name does not exist in the dictionary, add it with the default score 60.
    - If the name already exists, keep the existing score.
- Query Mode: Support interactive, repeated lookups—enter one student name per query.
    - If the name is not found, return the default score 60 and add the name to the dictionary.
    - Press Enter (empty input) to end the query.
- Output: After the query ends, print the current contents of the dictionary.

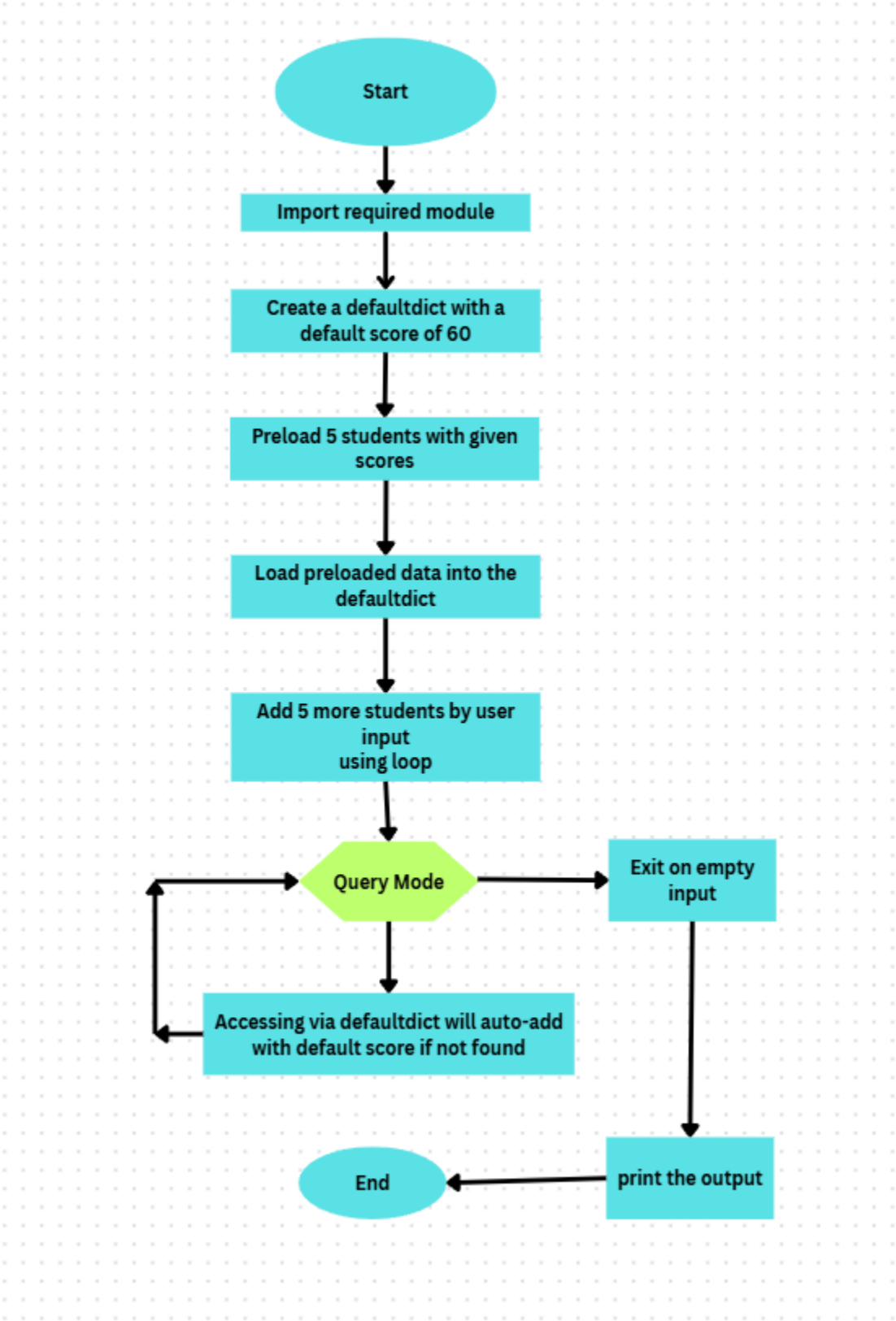
Tip: You may call **.strip()** on user input to ignore leading/trailing whitespace.

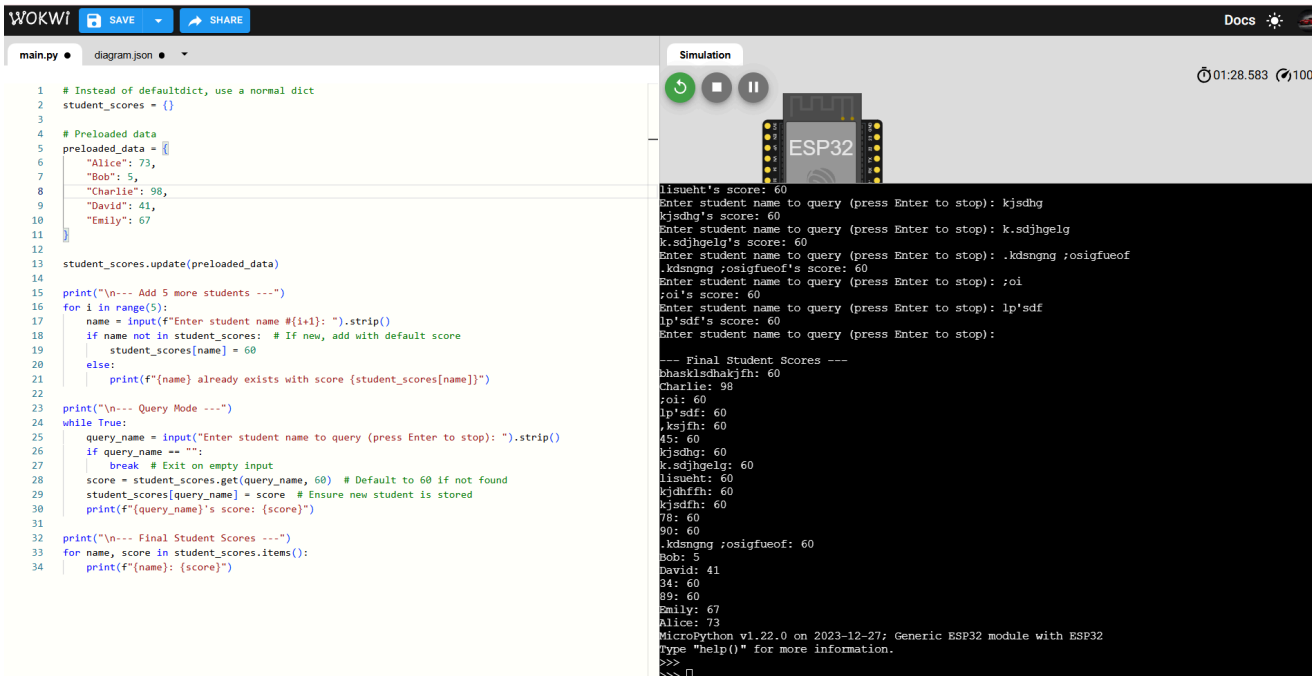
Name	Random Number
Alice	73
Bob	5
Charlie	98
David	41
Emily	67
Frank	30
Grace	12
Henry	88
Ivy	56
Jack	24

Source Code :

```
# Instead of defaultdict, use a normal dict
student_scores = {}

# Preloaded data
preloaded_data = {
    "Alice": 73,
    "Bob": 5,
    "Charlie": 98,
    "David": 41,
    "Emily": 67
}

student_scores.update(preloaded_data)

print("\n--- Add 5 more students ---")
for i in range(5):
    name = input(f"Enter student name #{i+1}: ").strip()
    if name not in student_scores:  # If new, add with default score
        student_scores[name] = 60
    else:
        print(f"{name} already exists with score
{student_scores[name]}")

print("\n--- Query Mode ---")
while True:
    query_name = input("Enter student name to query (press Enter to
stop): ").strip()
    if query_name == "":
        break  # Exit on empty input
    score = student_scores.get(query_name, 60)  # Default to 60 if not
found
    student_scores[query_name] = score  # Ensure new student is stored
    print(f"{query_name}'s score: {score}")

print("\n--- Final Student Scores ---")
for name, score in student_scores.items():
    print(f"{name}: {score}")
```

Flow chart:


Start
Import required module
Create a defaultdict with a default score of 60
Preload 5 students with given scores
Load preloaded data into the defaultdict
Add 5 more students by user input using loop
Query Mode
Exit on empty input
Accessing via defaultdict will auto-add with default score if not found
print the output
End



Test Record :

Step No.	Action	Input / Test Data	Expected Output / Behavior	Actual Output / Behavior	Result
1	Load preloaded student data into dictionary	N/A	Dictionary contains 5 preloaded students with correct scores	Matches expected	Pass
2	Add 5 students (with a duplicate name to test handling)	Frank, Grace, Bob, Hannah, Ian	New names get default score 60; Duplicate `Bob` shows existing score and is not overwritten	Matches expected	Pass
3	Query an existing student's score	Charlie	Prints `Charlie's score: 98`	Matches expected	Pass
4	Query a non-existing student's score	Zara	Adds `Zara` with score 60 and prints `Zara's score: 60`	Matches expected	Pass
5	Stop querying by pressing Enter	[Enter]	Query mode ends	Matches expected	Pass
6	Display final student scores	N/A	Prints all student names with correct scores, including newly added ones	Matches expected	Pass

Progress and Conclusion :

- Strengths:
  - Correct usage of `collections.defaultdict` to simplify default value handling.
  - Meets all functional requirements — preload, user add, query mode, and final printout.
  - `.strip()` used properly to handle accidental whitespace in input.
  - Code structure is clear and logically divided into steps.
- Areas for Improvement:
  - Could validate that names are non-empty before adding them.
  - Could add case-insensitive handling so `"alice"` and `"Alice"` are treated as the same.
  - Consider formatting the final output in a table for better readability.
- Next Step:
  - Allow default score to be changed easily via a function parameter or configuration setting.
  - Save the final dictionary to a file (e.g., CSV) for persistence.

Feedback Comments :

Completing these tasks was both a learning experience and a confidence booster. I appreciated how the exercises covered different Python concepts—from basic syntax and indentation rules to more advanced topics like generators, decorators, and default dictionaries. Each task challenged me to think logically and apply concepts in a practical way, rather than just memorizing theory.

I especially enjoyed the generator and decorator tasks, as they showed me how Python can handle problems efficiently with clean, minimal code. The use of `yield` to save memory and decorators to add extra functionality felt powerful and practical for real-world programming.

At times, I faced small challenges—such as ensuring correct indentation or structuring the logic—but solving these issues gave me a sense of achievement. I also realized how important it is to write readable, well-structured code, as even small formatting mistakes can cause big problems.

Overall, I feel more confident and motivated after completing these exercises. They not only improved my technical skills but also strengthened my problem-solving mindset. I'm excited to apply these techniques to bigger projects and explore Python's features in greater depth.

Thank you