Python Practice by: Ranjitsingh Mugavekar

MACHINE LEARNING USING PYTHON || DATA SCIENCE USING PYTHON || MindWave MUGAON LAB 2

1. Write a program using a for loop to print numbers from 1 to 10.
2. Use a for loop to print all even numbers between 1 and 20.
3. Ask the user to enter a word, then use a for loop to print each character on a new line.
4. Write a program using a while loop to print numbers from 1 to 5.
5. Keep asking the user to enter numbers. Stop when the user enters 0. Then print the total

sum. 6. Use a while loop to print numbers from 10 down to 1. 7. Write a function named say_hello() that prints "Hello, welcome to Python!" 8. Write a function add(a, b) that returns the sum of a and b. 9. Write a function square(n) that returns the square of a number. 10. Write a function that takes a name as input and prints "Hello, [name]!" 11. Write a function is_palindrome(num) that checks if a number is a palindrome. A number is a palindrome if it reads the same backward as forward (like 121 or 1331). Use a while loop to reverse the number inside the function. 12. Write a function print_star_pattern(n) that prints a right-angled triangle pattern of stars using a for loop. The number of lines n should be passed to the function.

Q.) 1.Program to print numbers from 1 to 10

Method 1: Using for loop with range

In [ ]:
```
# Method 1: for loop with range
for number in range(1, 11):
    print(number)
```

In [ ]:
```

```

Method 2: Using a while loop

In [ ]:
```
# A `while` loop runs as long as a condition is true. This method uses manual incre
# Method 2: while loop
number = 1
while number <= 10:
    print(number)
    number += 1
```

In [ ]:
```

```

Method 3: Using list and for loop

In [ ]:
```
###  <span style="color:green">Method 1: Using for loop with range</span>
```

In [ ]:
```
# we first create a list of numbers from 1 to 10, then loop through the list.
# Method 3: Using list
numbers = list(range(1, 11))
for number in numbers:
    print(number)
```

In [ ]:
```

```

Method 4: Using recursion (advanced)

In [ ]:
```
#A function that calls itself to print numbers from 1 to 10. This is more of a conc

# Method 4: Recursion
def print_numbers(n):
    if n > 10:
        return
    print(n)
    print_numbers(n + 1)

print_numbers(1)
```

In [ ]:
```

```

5: Using list comprehension (compact)

In [ ]:
```
#A Pythonic one-liner using list comprehension for side-effect (printing).
# Method 5: List comprehension
[print(number) for number in range(1, 11)]
```

In [ ]:
```

```

Q.) 2.Print All Even Numbers Between 1 to 20

Method 1: For Loop with Step in range()

In [ ]:
```
# Method 1: Using `range(start, stop, step)` in a `for` loop.
# Method 1: For loop with step of 2
for number in range(2, 21, 2):
    print(number)
```

In [ ]:
```

```

Method 2: Using for loop with if condition

In [ ]:
```
# Method 2: Loop through all numbers and check even using `if` condition.
# Method 2: Filtering even numbers
for number in range(1, 21):
    if number % 2 == 0:
        print(number)
```

In [ ]:
```

```

Method 3: Looping through a list of even numbers

In [ ]:
```
###  <span style="color:green">Method 1: Using for loop with range</span>
```

In [ ]:
```
# Create a list of even numbers and iterate through it.
# Method 3: Using predefined list
even_numbers = list(range(2, 21, 2))
for number in even_numbers:
    print(number)
```

In [ ]:
```

```

Method 4: Using List Comprehension inside a for loop

In [ ]:
```
# Use list comprehension to generate even numbers, then loop through them.
# Method 4: List comprehension
even_numbers = [num for num in range(1, 21) if num % 2 == 0]
for number in even_numbers:
    print(number)
```

In [ ]:
```

```

5: Using for loop with function and range()

In [ ]:
```
# Use a function with `for` loop to print even numbers.
# Method 5: For loop in a function
def print_even_numbers():
    for number in range(1, 21):
        if number % 2 == 0:
            print(number)

print_even_numbers()
```

In [ ]:
```

```

Q.) 3.Ask the user to enter a word, then use a for loop to print each character on a new line.

Method 1: Basic for loop

In [ ]:
```
# This is the most direct and common way to iterate over each character.
# Method 1
word = input("Enter a word: ")
for char in word:
    print(char)
```

In [ ]:

Method 2: Using index with range(len())

In [ ]:
```
# We use `range(len(word))` to loop through the indices.
# Method 2
word = input("Enter a word: ")
for i in range(len(word)):
    print(word[i])
```

In [ ]:

Method 3: Using enumerate()

In [ ]:
```
###  <span style="color:green">Method 1: Using for loop with range</span>
```

In [ ]:
```
# `enumerate()` gives both index and character, which can be useful.
# Method 3
word = input("Enter a word: ")
for index, char in enumerate(word):
    print(char)
```

In [ ]:

Method 4: Using List Comprehension (side effect).

In [ ]:
```
# List comprehensions can execute print functions in a compact style.
# Method 4
word = input("Enter a word: ")
[print(char) for char in word]
```

In [ ]:

Method 5: Using a function and loop.

In [ ]:
```
# This method separates input and printing logic for clarity and reusability.
# Method 5
def print_characters(word):
    for char in word:
        print(char)

word = input("Enter a word: ")
print_characters(word)
```

In [ ]:

Q.) 4.Write a program using a while loop to print numbers from 1 to 5.

Method 1: Basic while loop (incrementing).

In [ ]:
```
#This is the most straightforward and beginner-friendly version.
# Method 1: Basic while loop
number = 1
while number <= 5:
    print(number)
    number += 1
```

In [ ]:

Method 2: Using break to exit the loop.

In [ ]:
```
# Here, we run an infinite loop and use `break` to exit when the condition is met.
# Method 2: Using break
number = 1
while True:
    print(number)
    number += 1
    if number > 5:
        break
```

In [ ]:

Method 3: Printing in reverse (5 to 1), then reversing logic.

In [ ]:
```
# Demonstrates that logic can be flipped and still controlled with `while`.
# Method 3: Reverse while loop (5 to 1)
number = 5
while number >= 1:
    print(6 - number)
    number -= 1
```

In [ ]:

Method 4: Using a function to wrap the while loop.

In [ ]:
```
#Encapsulates the logic in a reusable function.
```

In [ ]:
```
# Method 4: While loop inside a function
def print_numbers():
    number = 1
    while number <= 5:
```

```
        print(number)
        number += 1

print_numbers()
```

In [ ]:

Method 5: Using a list and index with while loop.

In [ ]:
```
#This shows how to loop through elements with manual indexing.
# Method 5: Using list and index
numbers = [1, 2, 3, 4, 5]
index = 0
while index < len(numbers):
    print(numbers[index])
    index += 1
```

In [ ]:

Q.) 5. Keep asking the user to enter numbers. Stop when the user enters 0. Then print the total sum..

Method 1: Basic while True with break.

In [ ]:
```
# Method 1: while True with break
total = 0
while True:
    num = int(input("Enter a number (0 to stop): "))
    if num == 0:
        break
    total += num

print("Total sum:", total)
```

In [ ]:

Method 2: Use a condition-based while loop (num != 0).

In [ ]:
```
# Initialize outside loop and use condition `while num != 0`.
# Method 2: while num != 0
num = int(input("Enter a number (0 to stop): "))
total = 0

while num != 0:
    total += num
    num = int(input("Enter a number (0 to stop): "))

print("Total sum:", total)
```

In [ ]:

Method 3: Use a function for modular structure.

In [ ]:
```python
#Encapsulate logic in a function for reusability.
# Method 3: Using function
def sum_until_zero():
    total = 0
    while True:
        num = int(input("Enter a number (0 to stop): "))
        if num == 0:
            break
        total += num
    return total

print("Total sum:", sum_until_zero())
```

In [ ]:

Method 4: Use a list to store numbers and then sum.

In [ ]:
```python
# Append inputs to a list, stop at 0, then sum the list.
# Method 4: Using list to store inputs
numbers = []
while True:
    num = int(input("Enter a number (0 to stop): "))
    if num == 0:
        break
    numbers.append(num)

print("Total sum:", sum(numbers))
```

In [ ]:

Method 5: Using recursion (advanced).

In [ ]:
```python
# Recursive approach (for learning purposes).
# Method 5: Using recursion
def recursive_sum():
    num = int(input("Enter a number (0 to stop): "))
    if num == 0:
        return 0
    return num + recursive_sum()

print("Total sum:", recursive_sum())
```

In [ ]:

Q.) 6. Use a while loop to print numbers from 10 down to 1..

Method 1: Simple countdown with decrement.

In [ ]:
```
# Basic countdown using `while` loop with decrement.
# Method 1: Basic while loop
number = 10
while number >= 1:
    print(number)
    number -= 1
```

In [ ]:
```

```

Method 2: Use while True and break.

In [ ]:
```
# Infinite loop with `break` condition when number < 1.
# Method 2: while True with break
number = 10
while True:
    print(number)
    number -= 1
    if number < 1:
        break
```

In [ ]:
```

```

Method 3: Loop in reverse using a function.

In [ ]:
```
# Encapsulate countdown in a function.
# Method 3: Function-based countdown
def countdown():
    number = 10
    while number >= 1:
        print(number)
        number -= 1

countdown()
```

In [ ]:
```

```

Method 4: Using a list with index.

In [ ]:
```
# Create a list from 10 to 1 and loop through using index.
# Method 4: Loop through list using index
numbers = list(range(10, 0, -1))
i = 0
while i < len(numbers):
```

```
    print(numbers[i])
    i += 1
```

In [ ]:

Method 5: Using recursion (advanced technique).

In [ ]:
```
# Recursive function to print from 10 to 1.
# Method 5: Recursive approach
def print_reverse(n):
    if n == 0:
        return
    print(n)
    print_reverse(n - 1)

print_reverse(10)
```

In [ ]:

Q.) 7. Write a function named say_hello() that prints "Hello, welcome to Python!".

Method 1: Standard Function Definition.

In [ ]:
```
# Method 1
def say_hello():
    print("Hello, welcome to Python!")

say_hello()
```

In [ ]:

Method 2: Function with Optional Parameter (can be reused).

In [ ]:
```
# Function with a parameter (default greeting).
# Method 2
def say_hello(message="Hello, welcome to Python!"):
    print(message)

say_hello()
```

In [ ]:

Method 3: Lambda Function (though unconventional for print).

In [ ]:
```
# Using a lambda (for functional programming learners).
# Method 3
say_hello = lambda: print("Hello, welcome to Python!")
```

```
say_hello()
```

In [ ]:

Method 4: Inside a Class (OOP approach).

In [ ]:
```
# Function as a method inside a class.
# Method 4
class Greeter:
    def say_hello(self):
        print("Hello, welcome to Python!")

g = Greeter()
g.say_hello()
```

In [ ]:

Method 5: Function Returning the Message Instead of Printing.

In [ ]:
```
# Return the greeting instead of printing.
# Method 5
def say_hello():
    return "Hello, welcome to Python!"

# Print the returned message
print(say_hello())
```

In [ ]:

Q.) 8. Write a function add(a, b) that returns the sum of a and b..

Method 1: Basic Function Definition.

In [ ]:
```
# Method 1: Simple function to add two numbers

def add(a, b):
    return a + b

# Pros:
# - Simple and readable
# - Easy to use for any numerical types
# Cons:
# - No input validation

print(add(10, 5))  # Output: 15
```

In [ ]:

Method 2: With Type Hints (Python 3.5+).

In [ ]:
```
# Method 2: Function with type hints

def add(a: int, b: int) -> int:
    return a + b

# Pros:
# - Improves code readability and editor support
# - Good for static type checking with tools like mypy
# Cons:
# - Python does not enforce types at runtime
# - Fails silently if types are wrong unless manually checked

print(add(10, 5))  # Output: 15
```

In [ ]:
```

```

Method 3: Lambda Function (Compact).

In [ ]:
```
# Method 3: Using a lambda (anonymous function)

add = lambda a, b: a + b

# Pros:
# - One-liner, useful for quick expressions
# - Good for functional-style programming
# Cons:
# - Less readable for beginners
# - Cannot include comments or complex logic

print(add(10, 5))  # Output: 15
```

In [ ]:
```

```

Method 4: Function with Input Validation.

In [ ]:
```python
# Method 4: Add validation for better safety

def add(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        return "Both inputs must be numbers."
    return a + b

# Pros:
# - Safer and more robust
# - Prevents common input errors
# Cons:
# - Slightly more code
# - Performance overhead in strict scenarios

print(add(10, "5"))  # Output: Both inputs must be numbers.
```

In [ ]:
```python

```

Method 5: Inside a Class as a Static Method.

In [ ]:
```python
# Method 5: Encapsulated in a class for modular design

class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

# Pros:
# - Useful for organizing related math functions
# - Can be easily extended or reused
# Cons:
# - Overhead if you just need a single simple function

print(MathOperations.add(10, 5))  # Output: 15
```

In [ ]:
```python

```

Q.) 9. Write a function square(n) that returns the square of a number.

Method 1: Basic Function Using Multiplication.

In [ ]:
```python
# Method 1: Basic function using multiplication

def square(n):
    return n * n

# Pros:
# - Simple and fast
# - Works for both int and float
# Cons:
```

```
# - No error handling or input validation

print(square(5))  # Output: 25
```

In [ ]:

Method 2: Using Exponentiation (**).

In [ ]:
```
# Method 2: Use exponentiation operator

def square(n):
    return n ** 2

# Pros:
# - More mathematically expressive
# - Can easily extend for cube, powers, etc.
# Cons:
# - Slightly less intuitive for beginners

print(square(5))  # Output: 25
```

In [ ]:

Method 3: With Type Hints.

In [ ]:
```
# Method 3: Use type hints for better readability

def square(n: float) -> float:
    return n * n

# Pros:
# - Helpful for documentation and static checking
# - Good for editors and tools like mypy
# Cons:
# - Python does not enforce type at runtime

print(square(5))  # Output: 25.0
```

In [ ]:

Method 4: Lambda Function.

In [ ]:
```
# Method 4: Lambda function

square = lambda n: n * n

# Pros:
# - Very compact
# - Useful for inline operations
# Cons:
# - Less readable for beginners
# - Can't include validation or complex logic
```

```
print(square(5))  # Output: 25
```

In [ ]:

Method 5: Class with Static Method.

In [ ]:
```
# Method 5: Using a class with a static method

class MathOperations:
    @staticmethod
    def square(n):
        return n * n

# Pros:
# - Good for grouping math-related utilities
# - Scalable for larger codebases
# Cons:
# - Overhead for a simple one-line task

print(MathOperations.square(5))  # Output: 25
```

In [ ]:

Q.) 10. Write a function that takes a name as input and prints "Hello, [name]!" .

Method 1: Simple String Concatenation.

In [ ]:
```
# Method 1: Simple string concatenation

def greet(name):
    print("Hello, " + name + "!")

# Pros:
# - Simple and beginner-friendly
# - Works in all Python versions
# Cons:
# - Harder to manage if many variables are used

greet("Ranjitsingh")  # Output: Hello, Ranjitsingh!
```

In [ ]:

Method 2: Using str.format().

In [ ]:
```
# Method 2: Using str.format()

def greet(name):
    print("Hello, {}!".format(name))
```

```
# Pros:
# - Cleaner for multiple placeholders
# - Compatible with older Python versions (3.0+)
# Cons:
# - Slightly less readable than f-strings

greet("Ranjitsingh")
```

In [ ]:

Method 3: Using f-strings (Python 3.6+)

In [ ]:
```
# Method 3: Using f-strings

def greet(name):
    print(f"Hello, {name}!")

# Pros:
# - Clean, readable, modern
# - Supports expressions inside {}
# Cons:
# - Only available in Python 3.6 and above

greet("Ranjitsingh")
```

In [ ]:

Method 4: Lambda Function

In [ ]:
```
# Method 4: Lambda function

greet = lambda name: print("Hello, " + name + "!")

# Pros:
# - Very concise
# - Good for quick usage
# Cons:
# - Not suitable for multi-line or complex logic
# - Less readable for beginners

greet("Ranjitsingh")
```

In [ ]:

Method 5: Function with Default Parameter.

In [ ]:
```
# Method 5: Function with default value

def greet(name="User"):
    print(f"Hello, {name}!")
```

```
# Pros:
# - Handles missing input gracefully
# - Useful in real-world applications
# Cons:
# - May hide bugs if user forgets to pass a value

greet()              # Output: Hello, User!
greet("Ranjitsingh") # Output: Hello, Ranjitsingh!
```

In [ ]:

Q.) 11. Write a function is_palindrome(num) that checks if a number is a palindrome. A number is a palindrome if it reads the same backward as forward (like 121 or 1331). Use a while loop to reverse the number inside the function.

.

Method 1: Using a while loop to reverse the number (Traditional).

In [ ]:
```
def is_palindrome(num):
    original = num
    reverse = 0

    while num > 0:
        digit = num % 10
        reverse = reverse * 10 + digit
        num //= 10

    # Pros:
    # - No conversion to string
    # - Fast for numeric-only logic
    # Cons:
    # - Slightly more code than other methods

    return original == reverse

# Test
print(is_palindrome(121))  # True
```

In [ ]:

Method 2: Convert to string and use slicing.

In [ ]:
```
def is_palindrome(num):
    str_num = str(num)
```

```
    # Pros:
    # - Very concise and readable
    # - Uses Python string slicing
    # Cons:
    # - Converts number to string (not purely numeric)

    return str_num == str_num[::-1]

# Test
print(is_palindrome(1331))  # True
```

In [ ]:

Method 3: Compare digits manually using indexing

In [ ]:
```
def is_palindrome(num):
    digits = list(str(num))
    left = 0
    right = len(digits) - 1

    while left < right:
        if digits[left] != digits[right]:
            return False
        left += 1
        right -= 1

    # Pros:
    # - Easy to adapt for base conversions
    # - Can be extended to check palindromes in lists
    # Cons:
    # - More code and uses strings internally

    return True

# Test
print(is_palindrome(1221))  # True
```

In [ ]:

Method 4: Use recursion (advanced logic)

In [ ]:
```
def is_palindrome(num):
    def reverse(n, rev=0):
        if n == 0:
            return rev
        return reverse(n // 10, rev * 10 + n % 10)

    # Pros:
    # - Functional style, elegant
    # - No string conversion
    # Cons:
    # - Risk of stack overflow with very large numbers
```

```
    return num == reverse(num)

# Test
print(is_palindrome(1001))  # True
```

In [ ]:

Method 5: Convert to string and use reversed() built-in

In [ ]:
```
def is_palindrome(num):
    str_num = str(num)

    # Pros:
    # - Uses built-in functions
    # - Easy to understand
    # Cons:
    # - Again, not purely numeric comparison

    return str_num == ''.join(reversed(str_num))

# Test
print(is_palindrome(909))  # True
```

In [ ]:

12. Write a function print_star_pattern(n) that prints a right-angled triangle pattern of stars using a for loop. The number of lines n should be passed to the function..

Method 1: Basic for Loop with String Multiplication

In [ ]:
```
def print_star_pattern(n):
    '''
    Method 1: Simple for loop with string multiplication

    Pros:
    - Clean and concise
    - Very efficient for fixed-size patterns
    Cons:
    - Limited flexibility if formatting becomes complex
    '''
    for i in range(1, n + 1):
        print('*' * i)

print_star_pattern(5)
```

In [ ]:

Method 2: Nested Loops (Manual star printing)

In [ ]:
```
def print_star_pattern(n):
    '''
    Method 2: Nested loops for detailed control

    Pros:
    - More control over spacing or pattern details
    - Good for learning loop logic
    Cons:
    - Verbose compared to string multiplication
    '''
    for i in range(1, n + 1):
        for j in range(i):
            print('*', end='')
        print()

print_star_pattern(5)
```

In [ ]:
```

```

Method 3: Using List and join()

In [ ]:
```
def print_star_pattern(n):
    '''
    Method 3: Build line as a list and join it

    Pros:
    - Useful when working with complex row compositions
    - Scalable for advanced pattern manipulation
    Cons:
    - Overhead for simple tasks like stars
    '''
    for i in range(1, n + 1):
        print(''.join(['*' for _ in range(i)]))

print_star_pattern(5)
```

In [ ]:
```

```

Method 4: Using f-strings with Padding (custom spacing)

In [ ]:
```
def print_star_pattern(n):
    '''
    Method 4: Use f-string formatting (if alignment or spacing needed)

    Pros:
    - Great for aligning or padding
    - Easy to modify for other symbols
    Cons:
    - Overkill for simple triangle pattern
    '''
```

```
    for i in range(1, n + 1):
        print(f"{'*' * i}")

print_star_pattern(5)
```

In [ ]:

Method 5: Print from a Pre-generated Pattern List

In [ ]:
```
def print_star_pattern(n):
    '''
    Method 5: Generate all rows first, then print

    Pros:
    - Good for storing, testing, or exporting patterns
    - Allows batch operations or saving to file
    Cons:
    - Slightly more memory usage
    '''
    pattern = ['*' * i for i in range(1, n + 1)]
    for line in pattern:
        print(line)

print_star_pattern(5)
```

In [ ]:

In [ ]: