XI Computer Science 2025

Slides Revision: 1.0

Programming Fundamentals (Python)

S.L.O. # 3

Sub Topics: 9   Total SLO: 39

MCQ: (13) 13 Marks   CRQ: (2) 6 Marks   ERQ: (1) 7 Marks

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903

3.1 Programming Basics

SLO	Students should be able to	Cognitive Level
3.1.1	define program and programming language;	R
3.1.2	differentiate between program syntax and program semantics;	U
3.1.3	explain the following types of programming languages: a. low-level languages: machine language, assembly language, b. high-level languages: procedural language, structured language, object-oriented language;	U
3.1.4	explain the functions of an/a assembler, compiler, and interpreter;	U
3.1.5	explain the Python language and its applications;	U

3.1 Programming Basics

SLO	Students should be able to	Cognitive Level
3.1.6	explain the Python Integrated Development Environment (IDE);	U
3.1.7	describe comments in Python programming;	U
3.1.8	write a Python program using single-line and multiple-line comments;	A

SLO 3.1.1 U

define program and programming language;

SLO 3.1.1 U

Define Program

- A program is a set of instructions or code written in a specific programming language that is designed to perform a particular task or solve a problem when executed by a computer.
- These instructions are processed by the computer's central processing unit (CPU), enabling the system to carry out operations such as calculations, data processing, or interacting with users and other systems.
- Programs can range from simple scripts to complex applications and may involve interacting with various system resources like memory, storage, or network connections.
  - **In summary**: Program is a sequence of instructions that tells the computer what to do.
  - **Purpose**: To automate tasks, solve problems, or create interactive systems.
  - **Written in**: Programming languages such as Python, Java, C++, or JavaScript.

SLO 3.1.1 U

Programming Language

- **Language**: A language is a system of communication consisting of symbols, sounds, words, or gestures that are used to convey meaning. It can be spoken, written, or signed, and it allows humans or systems to share thoughts, emotions, instructions, or information.
- **Programming Language**: A programming language is a formal set of rules and syntax used to communicate instructions to a computer. It provides a way for developers to write code that can be understood and executed by machines to perform specific tasks. Programming languages serve as a bridge between human logic and machine functionality.

SLO 3.1.1 U

Programming Language

Rank	Programming Language
1	Python
2	Java
3	JavaScript
4	C/C++
5	C#
6	R
7	PHP
8	TypeScript
9	Swift
10	Objective-C

Rank	Programming Language
11	Rust
12	Go
13	Kotlin
14	Matlab
15	Ada
16	Dart
17	Ruby
18	PowerShell
19	VBA
20	Lua

SLO 3.1.2 U

differentiate between program syntax and program semantics;

SLO 3.1.2 U

Program Syntax and Program Semantics

- **Program Syntax** (How the code is written)
  - Definition: Syntax refers to the rules of writing Python code correctly like grammar in English. If syntax rules are broken, Python will raise a Syntax Error, and the code won’t run.
  - Example

```
# Missing closing bracket → syntax error
print("Hello, world"

# Wrong indentation → syntax error
def greet():
print("Hi")  # This line should be indented
```

Program Syntax and Program Semantics

SLO 3.1.2 U

Program Semantics (What the code means or does)

- Definition: Semantics is about the meaning or behavior of the code. Code with correct syntax can still be semantically wrong if it doesn't do what you intended.
- Examples of Semantic Errors in Python:

```
# Correct syntax, but wrong logic
def calculate_area(length, width):
    return length + width  # Semantically wrong; should be length * width

# Mixing data types incorrectly
name = "Ali"
age = 10
print(name + age)  # Semantically wrong; cannot add string and integer
```

Program Syntax and Program Semantics

SLO 3.1.2 U

Aspect	Program Syntax	Program Semantics
Definition	The *rules* that define the **structure** and **format** of code	The *meaning* or **behavior** that the code expresses or performs
Focus	How the code is *written*	What the code *does*
Checked by	Compiler or interpreter during the *parsing* phase	Interpreter or runtime system during *execution*
Error Type	Syntax error (e.g., missing punctuation, incorrect structure)	Semantic error (e.g., incorrect logic, wrong output)
Example 1 (Python)	print("Hello" → **Syntax error** (missing closing bracket)	print("Hello" + 5) → **Semantic error** (cannot add string + integer)
Example 2 (Logic)	if x = 10: → **Syntax error** (= is assignment, not comparison)	if x == "10": → syntactically fine, but semantically wrong if x is int
Analogy	Grammar rules in a language	The intended meaning of a correctly formed sentence

SLO 3.1.3 U

explain the following types of programming languages:

a. low-level languages:

I. machine language,

II. assembly language,

b. high-level languages:

I. procedural language,

II. structured language,

III. object-oriented language;

Level of Programming Languages

SLO 3.1.3 U

A. Low-Level Languages

Low-level languages are close to the hardware and provide little abstraction from a computer’s actual machine instructions. They are faster and more memory-efficient but harder to write and understand.

SLO 3.1.3 U

Level of Programming Languages

A. Low-Level Languages

- I. Machine Language
  - Definition: The most basic programming language written in binary code (0s and 1s). It is directly understood by the computer's CPU.
  - Example: 10110000 01100001 (May represent a simple instruction like moving a value into a register.)

```
10111000 00000101 00000000 00000000 00000000
10111011 00000011 00000000 00000000 00000000
00000001 11011000
10001001 11000001
```

  - Features: Fastest execution Very difficult to read or write. CPU-specific (platform dependent)

SLO 3.1.3 U

Level of Programming Languages

A. Low-Level Languages

- II. Assembly Language
  - Definition: A low-level language that uses mnemonics (short words) instead of binary. It still closely corresponds to machine language but is slightly easier for humans to write.
  - Example (x86 Assembly):

```
mov ax, 5       ; Load 5 into register AX
mov bx, 3       ; Load 3 into register BX
add ax, bx      ; Add the value in BX to AX (AX = AX + BX)
mov cx, ax      ; Move the result in AX to CX
```

  - Features: One step above machine language Requires an assembler to convert it into machine code. Used in hardware control, embedded systems, device drivers

Level of Programming Languages

SLO 3.1.3 U

B. High-Level Languages

High-level languages are closer to human language, easy to write, read, and maintain. These languages require compilers or interpreters to translate code into machine language.

SLO 3.1.3 U

Level of Programming Languages

B. High-Level Languages

- I. Procedural Language
  - Definition: These languages follow a step-by-step procedure or function to solve problems. The focus is on writing procedures (functions or routines).
  - Examples: C, Pascal, BASIC

```
import math

a = int(input('a='))
b = int(input('b='))
if a > b:
   print(math.pow(a, b))
else:
   print(math.pow(b, a))
```

  - Features: Uses loops, conditions, and function calls Follows top-down design. Good for small to medium-sized programs

SLO 3.1.3 U

Level of Programming Languages

B. High-Level Languages

- I. Procedural Language
  - Definition: These languages follow a step-by-step procedure or function to solve problems. The focus is on writing procedures (functions or routines).
  - Examples: C, Pascal, BASIC

```
10 PRINT "HELLO! WELCOME TO BASIC PROGRAMMING"
20 LET A = 5
30 LET B = 10
40 LET C = A + B
50 PRINT "THE SUM OF A AND B IS"; C
60 END
```

  - Features: Uses loops, conditions, and function calls Follows top-down design. Good for small to medium-sized programs

Level of Programming Languages

SLO 3.1.3 U

B. High-Level Languages

- II. Structured Language
  - Definition: Structured languages enforce a clear program structure, including blocks, conditionals, loops, and modular design.
  - Examples: C, Ada, ALGOL (C is both structured and procedural)
  - Features: Encourages modular programming. Uses control structures (if, for, while, etc.). Easy to debug and test

```
#include <stdio.h>

int main() {
    int a, b, sum;

    // Ask the user for input
    printf("Enter first number: ");
    scanf("%d", &a);

    printf("Enter second number: ");
    scanf("%d", &b);

    // Add the numbers
    sum = a + b;

    // Display the result
    printf("The sum of %d and %d is %d\n", a, b, sum);

    return 0;
}
```

Level of Programming Languages

SLO 3.1.3 U

B. High-Level Languages

- III. Object-Oriented Language
  - Definition: Object-oriented languages are based on objects and classes. They support concepts like inheritance, encapsulation, and polymorphism.
  - Examples: Python, Java, C++, C#, Ruby
  - Features: Real-world modeling using objects. Promotes code reuse through inheritance. Better for large and complex programs

```
#include <iostream>
using namespace std;

// Define a class
class Student {
public:
   // Attributes (data members)
   string name;
   int age;

   // Method (member function)
   void displayInfo() {
      cout << "Student Name: " << name << endl;
      cout << "Student Age: " << age << endl;
   }
};

int main() {
   // Create an object of Student
   Student s1;

   // Assign values to the object
   s1.name = "Ali";
   s1.age = 16;

   // Call the method
   s1.displayInfo();

   return 0;
}
```

SLO 3.1.4 U

explain the functions of an/a assembler, compiler, and interpreter;

Functions of Assembler, Compiler, and Interpreter

SLO 3.1.4 U

Assembler

- Function: An assembler translates assembly language (low-level mnemonics) into machine language (binary code) that a computer's CPU can execute.
- Example:

```
MOV AL, 1Ah          ; Assembly code
Translated by assembler to:
10110000 00011010    ; Machine code
```

- Use Case: Used in embedded systems, device drivers, or low-level hardware programming.

Functions of Assembler, Compiler, and Interpreter

SLO 3.1.4 U

Compiler

- Function: A compiler translates the entire source code (written in a high-level language like C or C++) into machine code or object code, and saves it as an executable file (.exe) before running.
- Example:

```
int main() {
    printf("Hello");
}
```

Executable (.exe or .out file)

- Use Case: Used in C, C++, Java (partially), Rust. Fast execution, since the program is already compiled.

Functions of Assembler, Compiler, and Interpreter

SLO 3.1.4 U

Interpreter

- An interpreter translates and executes code line-by-line (instead of all at once). It reads one statement, translates it to machine code, and runs it immediately.
- Example:

```
print("Hello")
```

Python interpreter reads and runs this line directly.

- Use Case: Used in Python, JavaScript, Ruby. Easier to debug, but slower than compiled languages.

SLO 3.1.5 U

explain the Python language and its applications;

SLO 3.1.5 U

Python Programming Language

What is Python

- Python is a high-level, interpreted, and general-purpose programming language that is:
  - Easy to read and write
  - Open-source
  - Widely used for many types of applications
  - It was created by Guido van Rossum and first released in 1991.
- Key Features:
  - Simple and clean syntax (looks like English)
  - Interpreted (no need to compile)
  - Supports object-oriented, functional, and procedural programming
  - Large collection of libraries and frameworks
  - Portable (runs on Windows, Mac, Linux, etc.)

Python Programming Language

SLO 3.1.5 U

Applications of Python

Field	How Python is Used
1. Web Development	Building websites and web apps using frameworks like **Django**, **Flask**
2. Data Science	Analyzing data, visualizing graphs, machine learning with **Pandas**, **NumPy**, **Scikit-learn**
3. Artificial Intelligence	Used in AI models, neural networks, deep learning with **TensorFlow**, **Keras**, **PyTorch**
4. Automation (Scripting)	Writing scripts to automate tasks like file handling, sending emails, data scraping
5. Game Development	Making simple games using libraries like **Pygame**
6. Desktop Applications	GUI applications with **Tkinter**, **PyQt**, **Kivy**
7. Cybersecurity	Writing security tools, penetration testing scripts
8. IoT (Internet of Things)	Controlling hardware using **MicroPython** or **Raspberry Pi**
9. Education	Ideal for teaching programming to beginners

SLO 3.1.6 U

explain the Python Integrated Development Environment (IDE);

Python Integrated Development Environment (IDE)

SLO 3.1.6 U

- **IDE**
- An IDE (Integrated Development Environment) is a software application that provides all the tools a programmer needs to write, run, debug, and manage Python code in one place.
- Think of it as a "coding workspace" for Python.

Python Integrated Development Environment (IDE)

SLO 3.1.6 U

Main Features of a Python IDE

Feature	Description
Code Editor	Where you write your Python programs
Syntax Highlighting	Colors and formats code to make it easier to read and spot errors
Code Completion	Suggests code as you type, saving time and reducing errors
Debugger	Helps you find and fix errors in your program by tracking line-by-line execution
Interpreter Integration	Allows you to run Python code directly inside the IDE
File Management	Let's you organize and manage your Python project files

SLO 3.1.6 U

Python Integrated Development Environment (IDE)

- **Popular Python IDEs:**

IDE/Editor	Best For	Key Features
IDLE	Beginners	Simple, built-in with Python installation
PyCharm	Professionals and large projects	Smart code editor, debugger, version control
Thonny	Students and beginners	Very beginner-friendly, easy interface
VS Code	All-purpose, lightweight	Extensions for Python, powerful and customizable
Jupyter Notebook	Data science, AI, ML	Interactive coding with output inline
Spyder	Scientific computing	Built-in for data analysis (used with Anaconda)

SLO 3.1.7 U

describe comments in Python programming;

Comments in Python Programming

SLO 3.1.7 U

- Comments are lines in a Python program that are ignored by the interpreter.
- They are used to explain code, make it easier to understand, or temporarily disable parts of code.

SLO 3.1.7 U

Comments in Python Programming

- Single-line Comments
  - Start with a # symbol.
  - Everything after # on that line is treated as a comment.

```
# This is a single-line comment
print("Hello")  # This prints a message
```

- Multi-line Comments
  - Python doesn't have a built-in multi-line comment symbol.
  - But you can use triple quotes ''' ... ''' or """ ... """ as a workaround for block comments.

```
'''
This is a multi-line comment.
It explains the next block of code.
'''
a = 10
b = 20
print(a + b)
```

SLO 3.1.7 U

Comments in Python Programming

- Why Use Comments

Purpose	Example
Explain complex code logic	# Loop to calculate factorial
Make code easier to read and maintain	# This function calculates area
Temporarily disable code	# print("Debug mode")
Help teamwork	Other programmers can understand your code

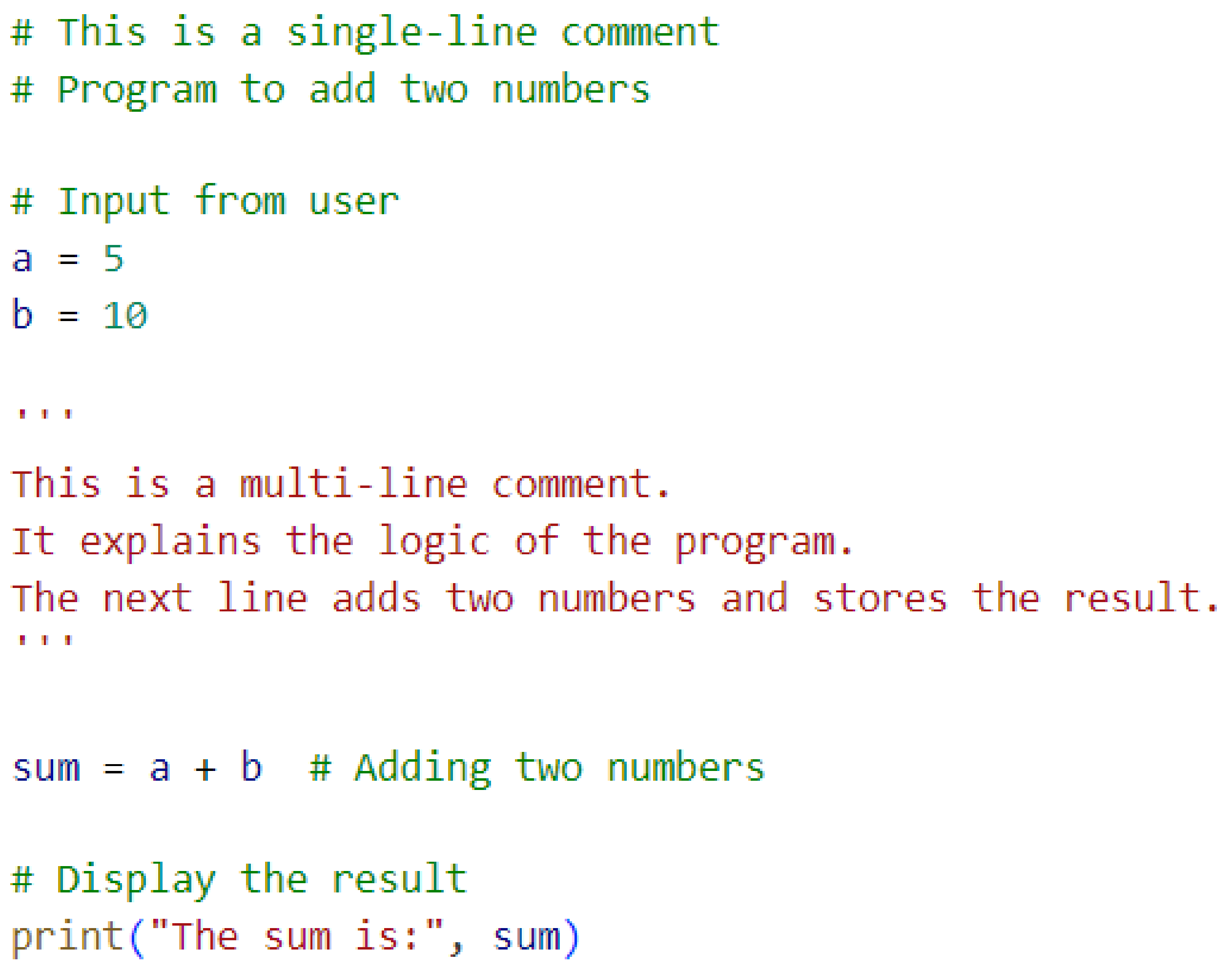
- Notes
  - Comments are not executed by Python.
  - Overusing comments can make code messy — write clear code and use comments only where needed

SLO 3.1.8 A

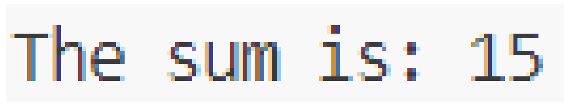
write a Python program using single-line and multiple-line comments;

SLO 3.1.8 A

write a Python program using single-line and multiple-line comments;



```
# This is a single-line comment
# Program to add two numbers

# Input from user
a = 5
b = 10

'''
This is a multi-line comment.
It explains the logic of the program.
The next line adds two numbers and stores the result.
'''


sum = a + b  # Adding two numbers

# Display the result
print("The sum is:", sum)
```



```
The sum is: 15
```

3.2 Turtle Graphics

SLO	Students should be able to	Cognitive Level
3.2.1	define turtle graphics;	R
3.2.2	use turtle library in Python program;	A
3.2.3	draw different shapes using the following functions: a. Movement: get screen( ), right( ) / rt( ), left( )/ lt( ), forward( )/ fd( ), backward( )/ bk( ), b. Screen Functions: clear( ), reset( ), stamp( ), clearstamp( ), bgcolor( ), title( ), c. Changing Turtle Attributes: shapesize( ), pensize( ), fillcolor( ), pencolor( ), color( ), shape( ), speed( ), begin_fill( )….end_fill(), penup( ), pendown( ), d. Preset Shapes: circle( ), dot( );	A

SLO 3.2.1 R

define turtle graphics;

Define Turtle Graphics

SLO 3.2.1 R

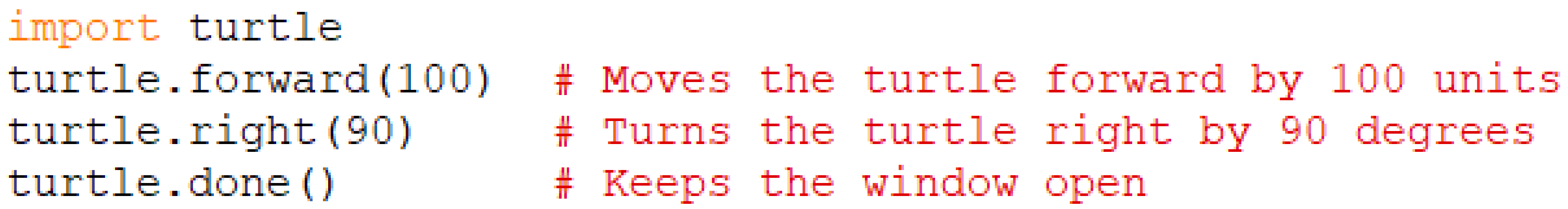
- Turtle graphics is a vector graphics system where a cursor, called the "turtle," moves across a screen to draw lines and shapes.
- The turtle has a position, orientation, and pen attributes (e.g., color, size, up/down state).
- Controlled programmatically, it allows users to create drawings through commands like moving forward, turning, or changing colors.
- Popularized by the Logo programming language, it's implemented in Python via the turtle library, making it an accessible tool for learning programming concepts like loops, conditionals, and functions through visual output.

SLO 3.2.2 A

use turtle library in Python program;

Use Turtle Library in Python program

- In Python, the turtle module is used for creating graphics and drawing using a virtual "turtle" that moves around the screen. There are several ways to import the turtle module or its components.
- Import the Entire Turtle Module
  - **Syntax**: import turtle
  - **User**: You access turtle functions and classes by prefixing them with turtle.
  - **Example**:

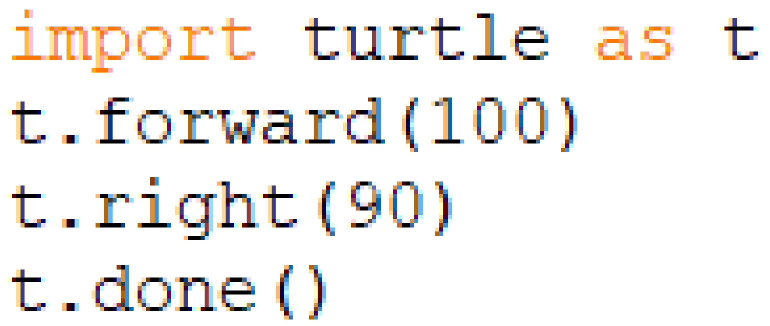


```
import turtle
turtle.forward(100)   # Moves the turtle forward by 100 units
turtle.right(90)      # Turns the turtle right by 90 degrees
turtle.done()         # Keeps the window open
```

SLO 3.2.2 A

Use Turtle Library in Python program

- Import Turtle Module with an Alias
  - **Syntax**: import turtle as t
  - **User**: Use the alias (t in this case) to call turtle functions.
  - **Example**:



```
import turtle as t
t.forward(100)
t.right(90)
t.done()
```

SLO 3.2.2 A

Use Turtle Library in Python program

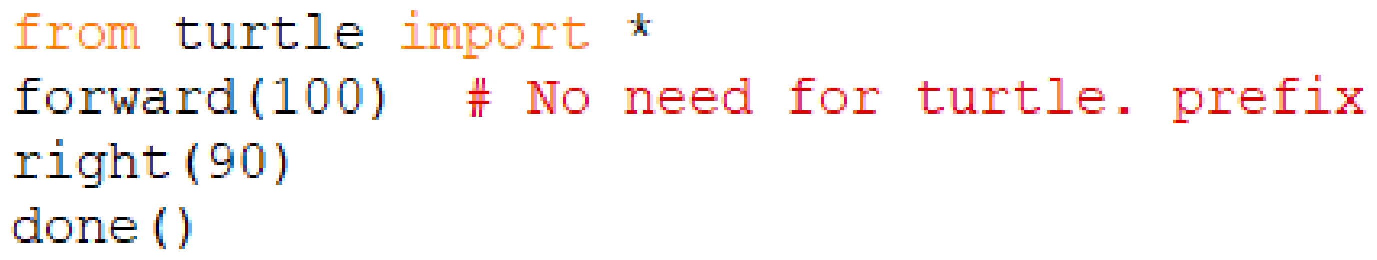
- Import Specific Classes or Functions from Turtle
  - **Syntax**: from turtle import Turtle, Screen
  - **User**: Directly use the imported classes without the turtle. prefix..
  - **Example**:

```
from turtle import Turtle, Screen
t = Turtle()          # Create a Turtle object
s = Screen()          # Create a Screen object
t.forward(100)
t.right(90)
s.mainloop()          # Keeps the window open (similar to turtle.done())
```

SLO 3.2.2 A

Use Turtle Library in Python program

- Import All Functions and Classes from Turtle
  - **Syntax**: from turtle import *
  - **User**: All turtle functions and classes are available without any prefix
  - **Example**:

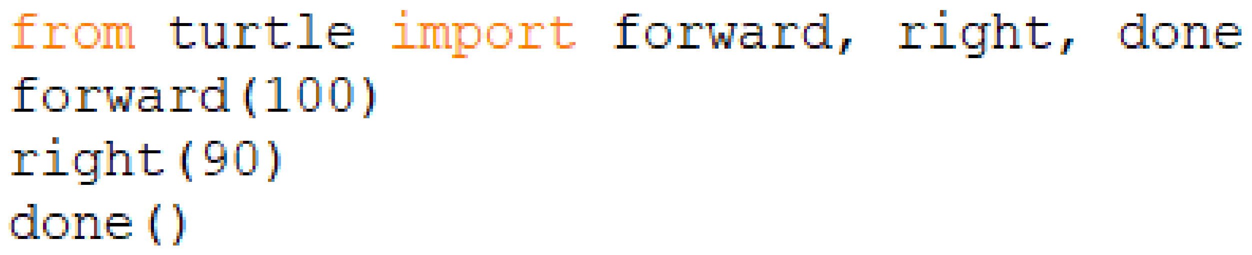


```
from turtle import *
forward(100)  # No need for turtle. prefix
right(90)
done()
```

SLO 3.2.2 A

Use Turtle Library in Python program

- Import Specific Functions from Turtle
  - **Syntax**: from turtle import forward, right, done
  - **User**: Use the imported functions directly.
  - **Example**:

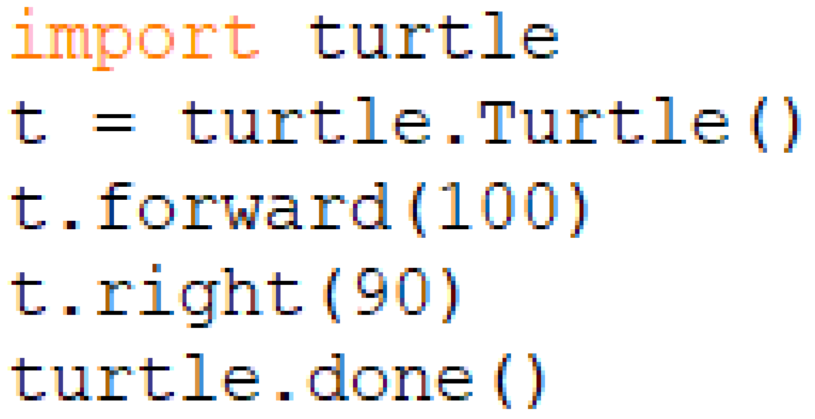


```
from turtle import forward, right, done
forward(100)
right(90)
done()
```

SLO 3.2.2 A

Use Turtle Library in Python program

- Using Turtle in Embedded Environments (e.g., IDLE or Jupyter Notebooks)
  - **Syntax**: import turtle
  - **User**: In some environments like Python's IDLE or Jupyter Notebooks, the turtle module works the same way, but you may need to adjust the done() or mainloop() call depending on the environment. The import methods remain identical.
  - **Example**:



```
import turtle
t = turtle.Turtle()
t.forward(100)
t.right(90)
turtle.done()
```

SLO 3.2.3 A

draw different shapes using the following functions:

a. Movement:
get screen( ), right( ) / rt( ), left( )/ lt( ), forward( )/ fd( ), backward( )/ bk( ),

b. Screen Functions:
clear( ), reset( ), stamp( ), clearstamp( ), bgcolor( ), title( ),

c. Changing Turtle Attributes:
shapesize( ), pensize( ), fillcolor( ), pencolor( ), color( ), shape( ), speed( ), begin_fill( )....end_fill(), penup( ), pendown( ),

d. Preset Shapes:
circle( ), dot( );

a. Movement

SLO 3.2.3 A

- getscreen()
  - Purpose: Returns the TurtleScreen object the turtle is drawing on, allowing access to screen-related settings.
  - Parameters: None.
  - Usage: Used to configure the screen, e.g., setting background color or title.
  - Example:

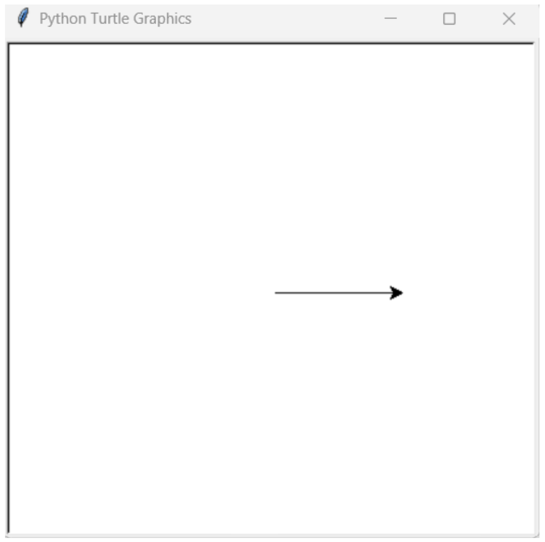
```
import turtle
t = turtle.Turtle()

screen = turtle.getscreen()

screen.bgcolor("red")
```

a. Movement

SLO 3.2.3 A

- forward(distance) / fd(distance)
  - Purpose: Moves the turtle forward by the specified distance in the direction it's facing.
  - Parameters: distance (float) – the distance to move (positive for forward).
  - Usage: Draws a line if the pen is down; moves without drawing if the pen is up.
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(100)

turtle.done()
```

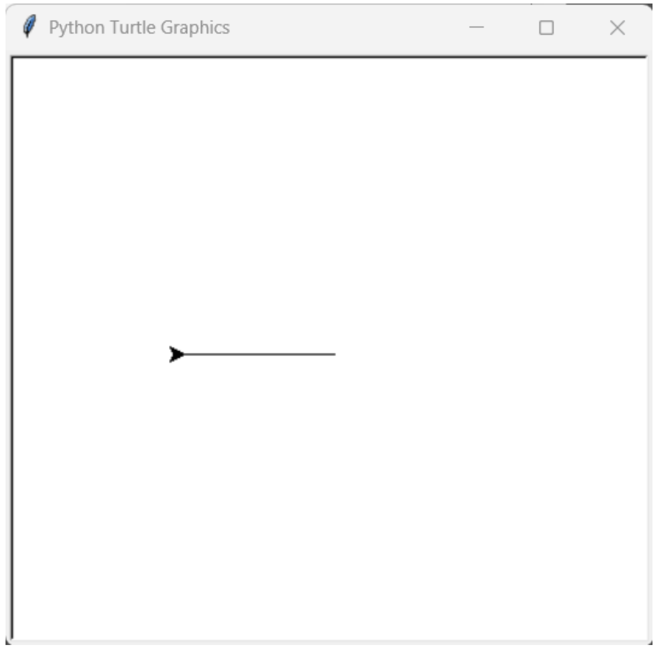


a. Movement

SLO 3.2.3 A

- backward(distance) / bk(distance)
  - Purpose: Moves the turtle backward by the specified distance, opposite to its current direction.
  - Parameters: distance (float) – the distance to move (positive for backward).
  - Usage: Useful for retreating or repositioning while drawing.
  - Example:

```
import turtle
t = turtle.Turtle()

t.backward(100)

turtle.done()
```

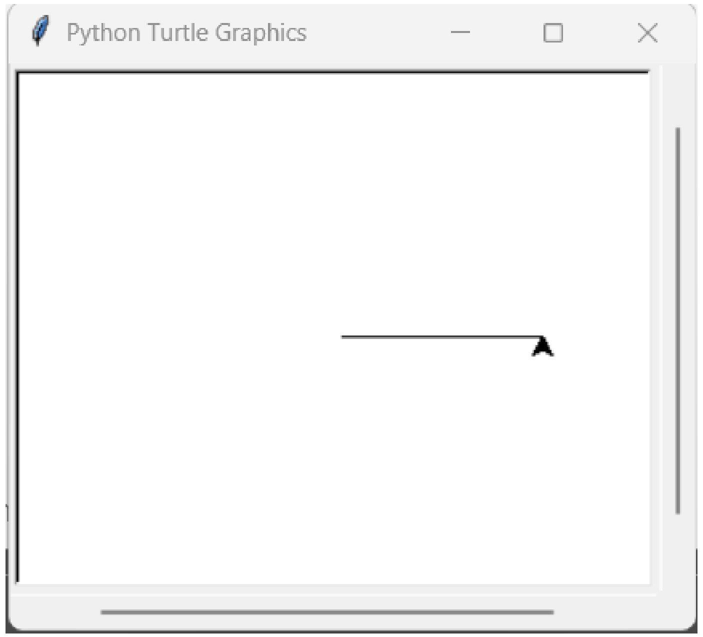


SLO 3.2.3 A

a. Movement

- left(degrees) / lt(degrees)
  - Purpose: Rotates the turtle counterclockwise by the specified angle (in degrees).
  - Parameters: degrees (float) – the angle to turn left.
  - Usage: Adjusts the turtle's orientation for drawing shapes or patterns.
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(100)
t.left(90)

turtle.done()
```

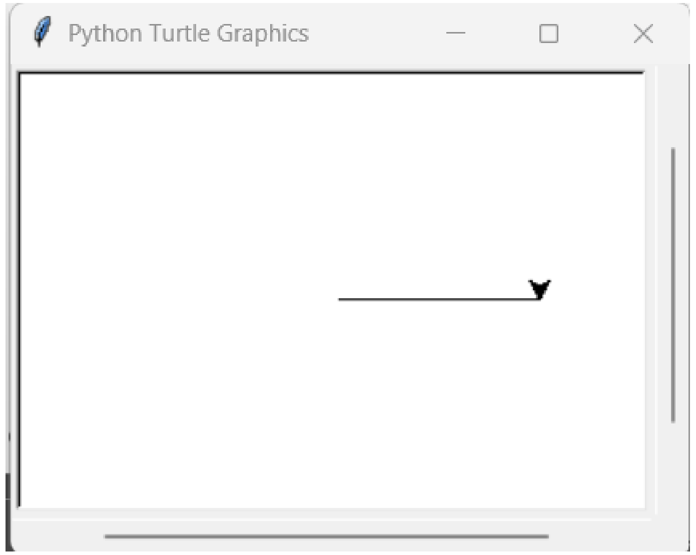


SLO 3.2.3 A

a. Movement

- right(degrees) / rt(degrees)
  - Purpose: Rotates the turtle clockwise by the specified angle (in degrees).
  - Parameters: degrees (float) – the angle to turn right.
  - Usage: Changes the turtle's direction without moving it, affecting subsequent movements.
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(100)
t.right(90)

turtle.done()
```

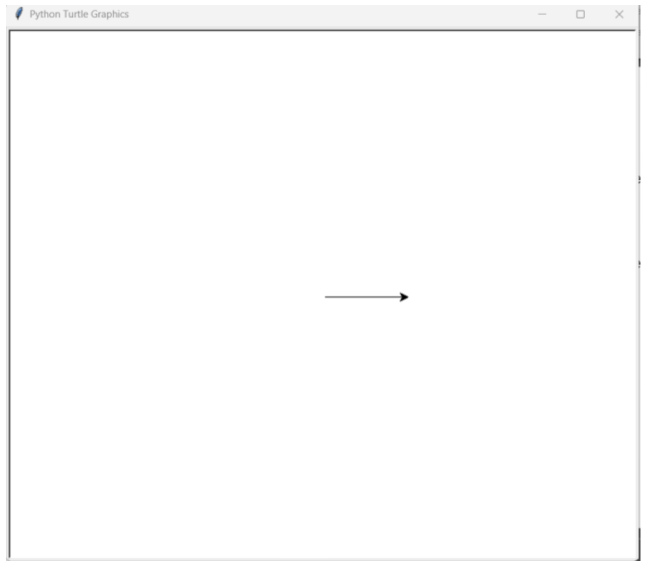


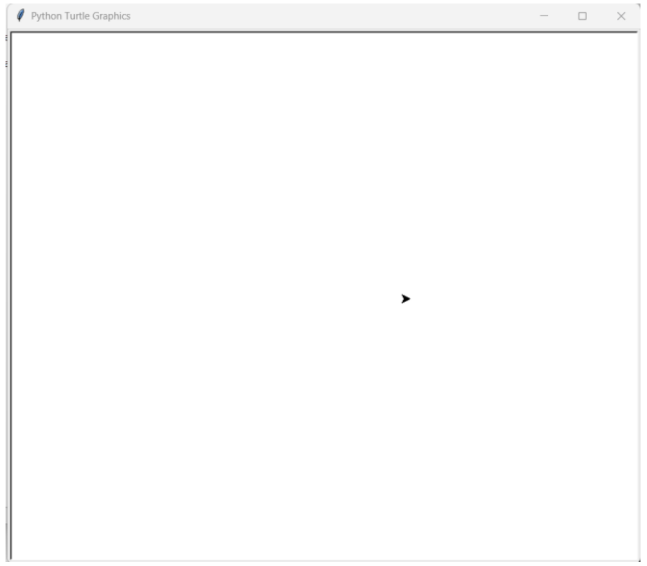
b. Screen Functions

SLO 3.2.3 A

- clear()
  - Purpose: Clears all drawings and turtles from the screen, resetting turtles to their initial positions.
  - Parameters: None.
  - Usage: Clears the canvas but keeps screen settings (e.g., background color).
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(100)

t.clear()

turtle.done()
```



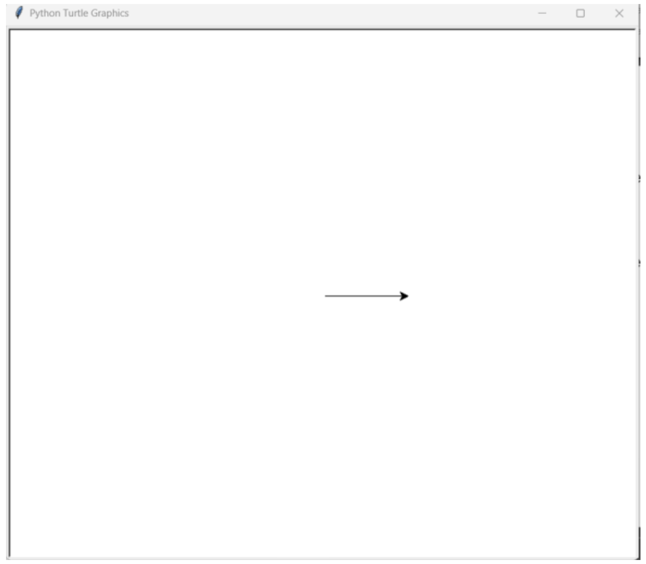


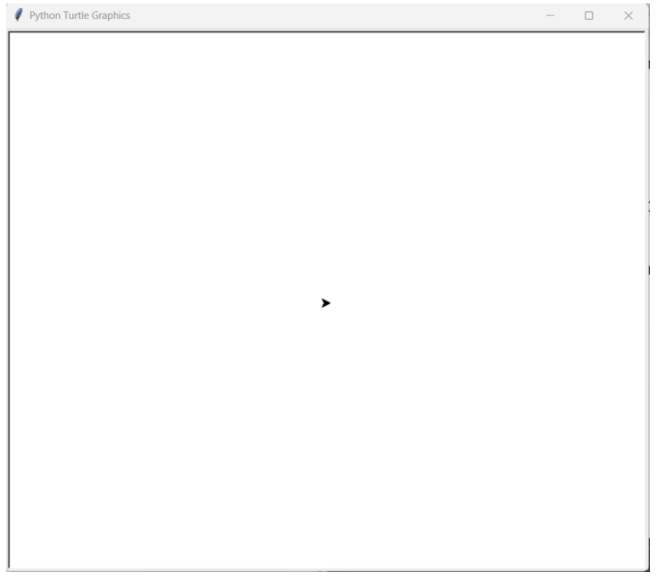
SLO 3.2.3 A

b. Screen Functions

- reset()
  - Purpose: Resets the screen and all turtles to their initial states, clearing drawings and resetting turtle properties (position, direction, etc.).
  - Parameters: None.
  - Usage: Starts a fresh drawing session, resetting everything except screen settings.
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(100)

t.reset()

turtle.done()
```



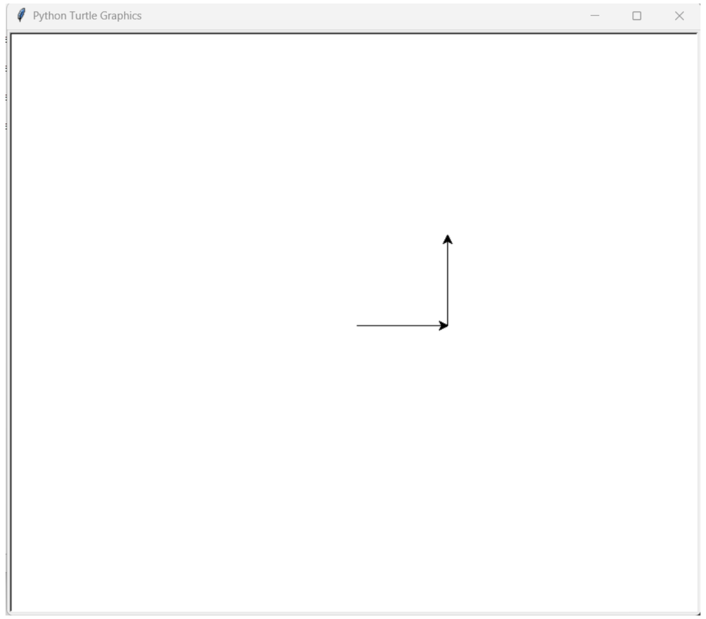


b. Screen Functions

SLO 3.2.3 A

- stamp()
  - Purpose: Leaves an imprint of the turtle's current shape at its current position and returns a stamp ID.
  - Parameters: None.
  - Usage: Used to place a copy of the turtle's shape (e.g., arrow, turtle) on the canvas, which remains even if the turtle moves.
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(100)

stmp = t.stamp()

t.left(90)
t.forward(100)

turtle.done()
```

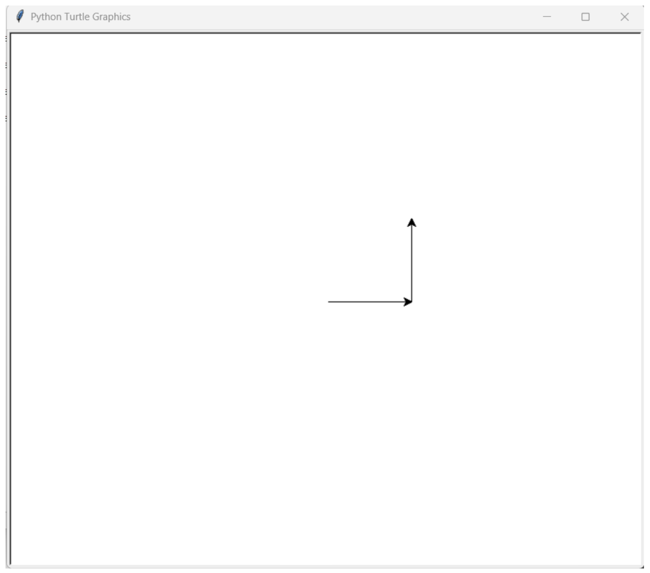


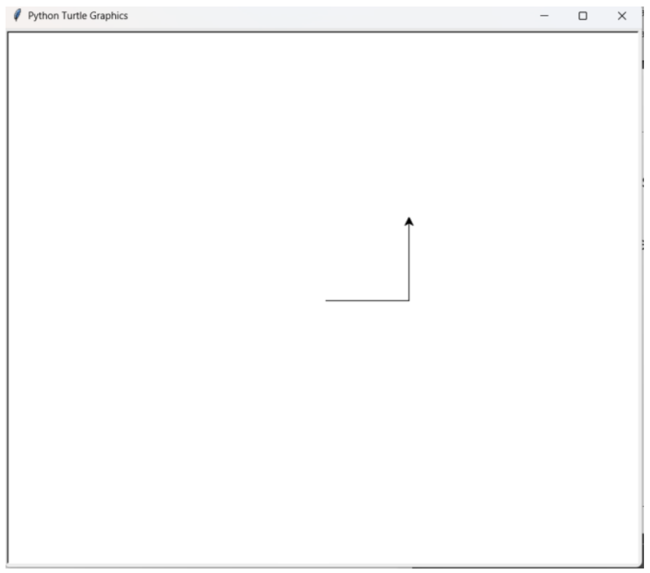
b. Screen Functions

SLO 3.2.3 A

- clearstamp(stampid)
  - Purpose: Removes a specific stamp from the screen using its stamp ID.
  - Parameters: stampid (int) – the ID returned by stamp().
  - Usage: Deletes a specific stamped shape without affecting others.
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(100)

stmp = t.stamp()

t.left(90)
t.forward(100)

t.clearstamp(stmp)

turtle.done()
```



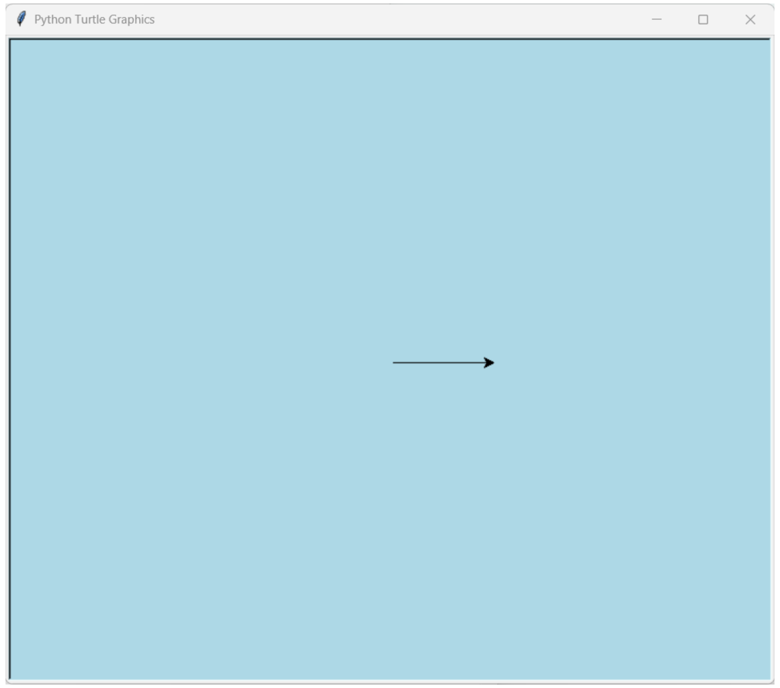


b. Screen Functions

SLO 3.2.3 A

- bgcolor(color)
  - Purpose: Sets or returns the background color of the screen.
  - Parameters: color (string or RGB tuple) – color name (e.g., "lightblue") or RGB values (e.g., (0.2, 0.8, 0.6)).
  - Usage: Changes the canvas background to enhance visual appeal.
  - Example:

```
import turtle
t = turtle.Turtle()

t.screen.bgcolor("lightblue")

t.forward(100)


turtle.done()
```

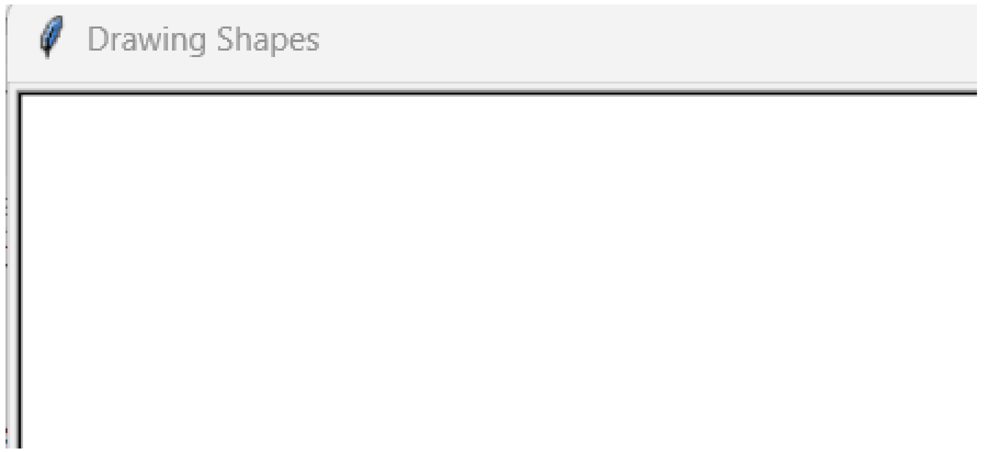


SLO 3.2.3 A

b. Screen Functions

- title(title)
  - Purpose: Sets the title of the turtle graphics window.
  - Parameters: title (string) – the text to display in the window's title bar.
  - Usage: Customizes the window title for clarity or branding.
  - Example:

```
import turtle
t = turtle.Turtle()

t.screen.title("Drawing Shapes")

turtle.done()
```



SLO 3.2.3 A

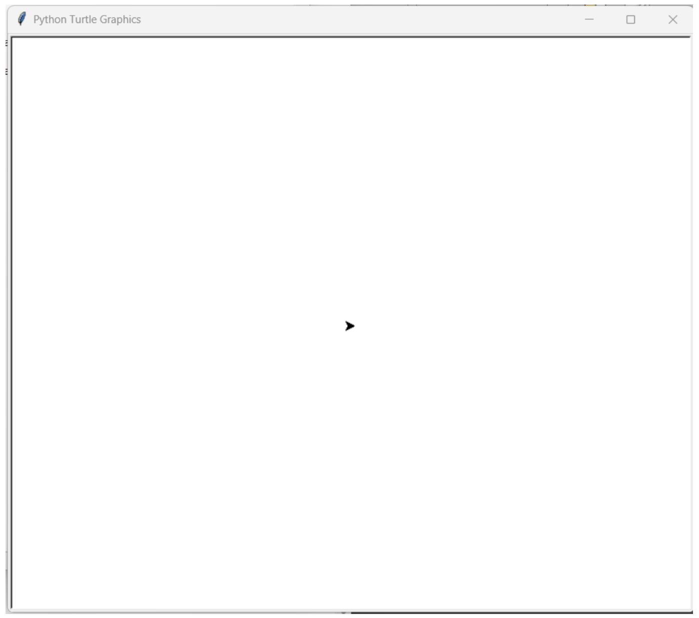
c. Changing Turtle Attributes

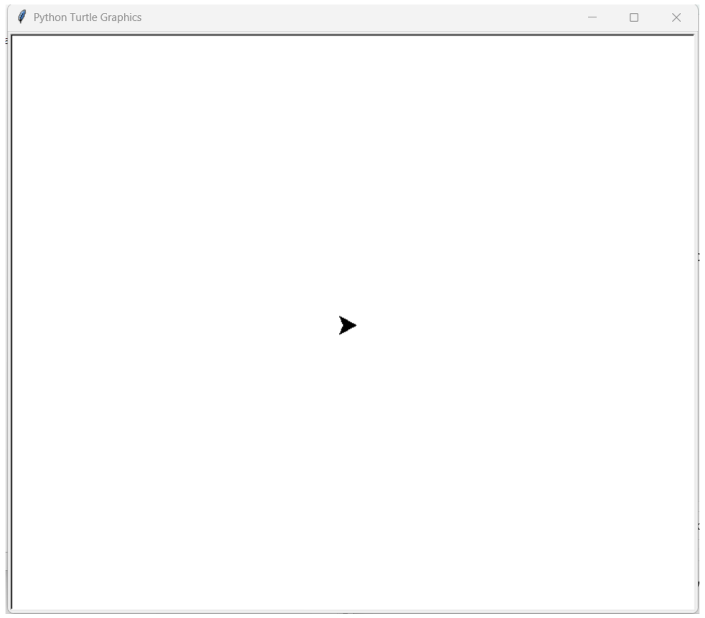
- shapesize(stretch_wid, stretch_len, outline=None)
  - Purpose: Sets the size of the turtle's shape by stretching its width, length, and optionally outline.
  - Parameters:
    - stretch_wid (float) – width scaling factor.
    - stretch_len (float) – length scaling factor.
    - outline (float, optional) – outline thickness scaling factor.
  - Usage: Adjusts the visual size of the turtle's shape (e.g., for stamps).

SLO 3.2.3 A

c. Changing Turtle Attributes

- shapesize(stretch_wid, stretch_len, outline=None)
  - Example:





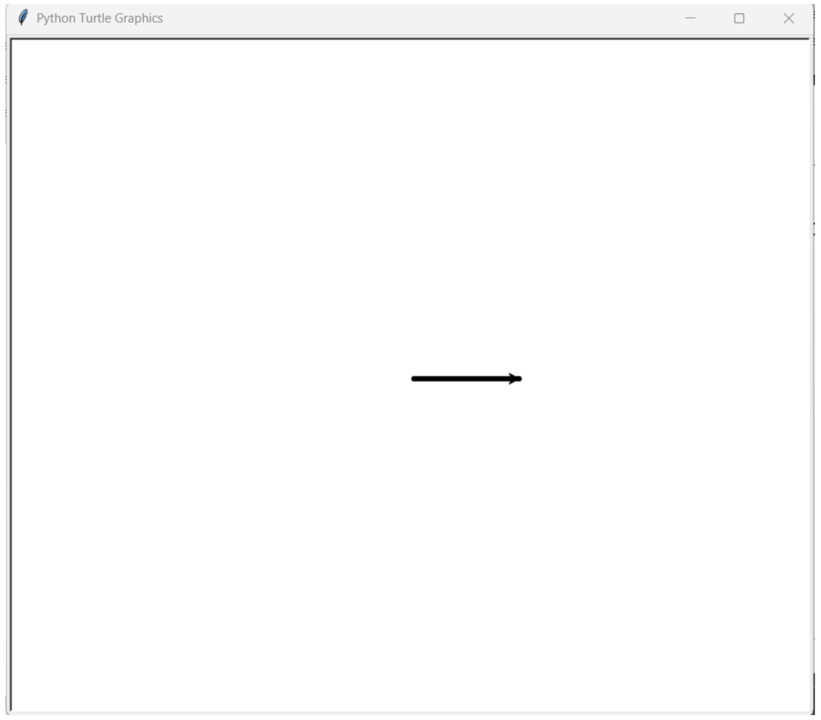
```
import turtle
t = turtle.Turtle()

t.shapesize(2, 2, 1)

turtle.done()
```

c. Changing Turtle Attributes

SLO 3.2.3 A

- pensize(width)
  - Purpose: Sets or returns the thickness of the pen for drawing lines.
  - Parameters: width (float) – the pen thickness in pixels.
  - Usage: Controls line thickness for visual emphasis.
  - Example:

```
import turtle
t = turtle.Turtle()

t.pensize(5)

t.forward(100)

turtle.done()
```

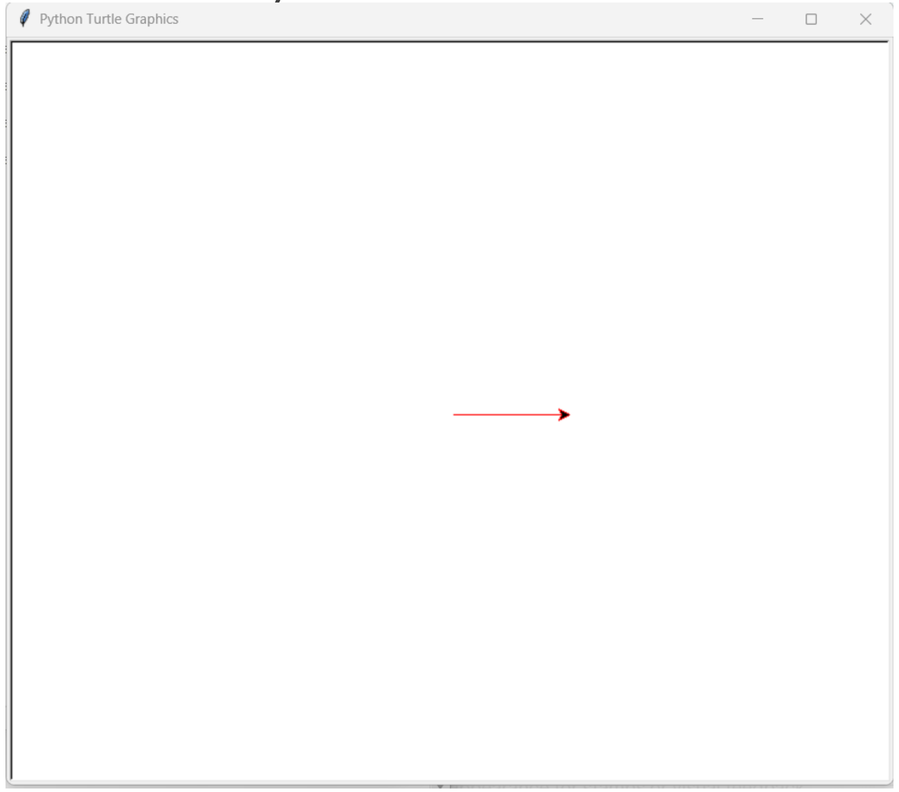


c. Changing Turtle Attributes

SLO 3.2.3 A

- pencolor(color)
  - Purpose: Sets or returns the color of the pen for drawing lines.
  - Parameters: color (string or RGB tuple) – color name or RGB values.
  - Usage: Changes the color of lines drawn by the turtle.
  - Example:

```
import turtle
t = turtle.Turtle()

t.pencolor("red")

t.forward(100)

turtle.done()
```



c. Changing Turtle Attributes

SLO 3.2.3 A

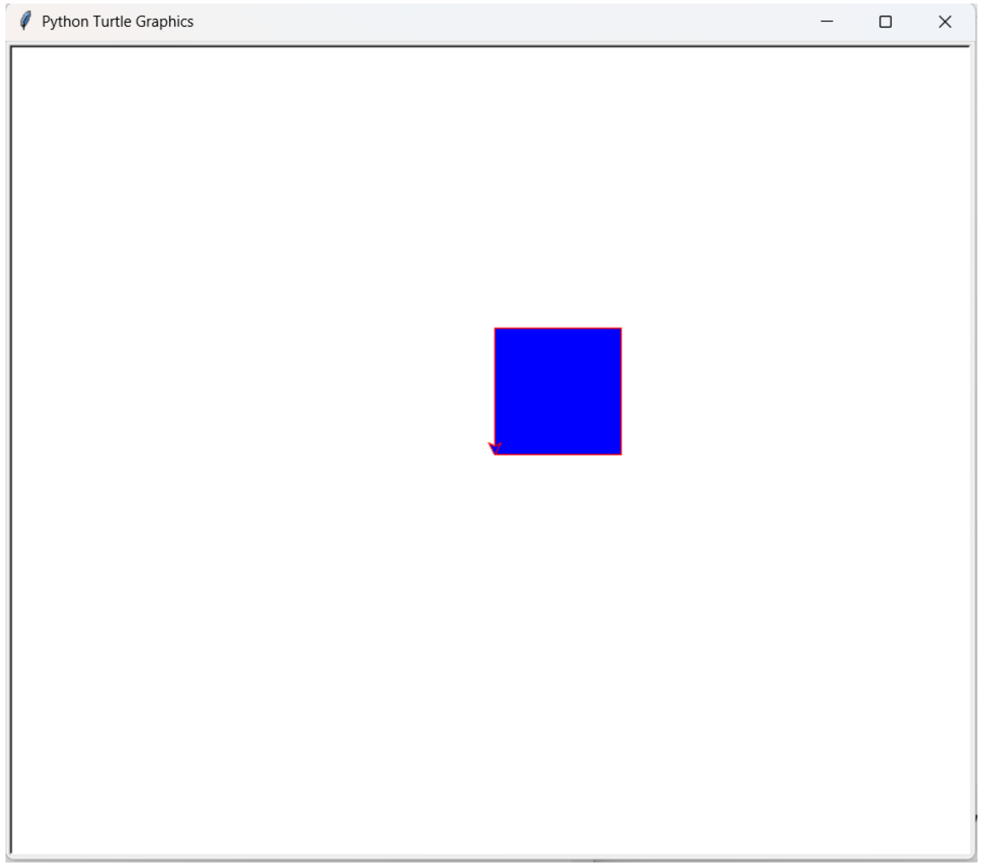
- color(pencolor, fillcolor=None)
  - Purpose: Sets both pen and fill colors at once or returns current colors.
  - Parameters:
    - pencolor (string or RGB tuple) – color for the pen.
    - fillcolor (string or RGB tuple, optional) – color for filling shapes.
  - Usage: Convenient for setting both colors simultaneously.

c. Changing Turtle Attributes

SLO 3.2.3 A

- color(pencolor, fillcolor=None)
  - Example:

```
import turtle
t = turtle.Turtle()

t.color("red", "blue")

t.begin_fill()

t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)

t.end_fill()


turtle.done()
```



c. Changing Turtle Attributes

SLO 3.2.3 A

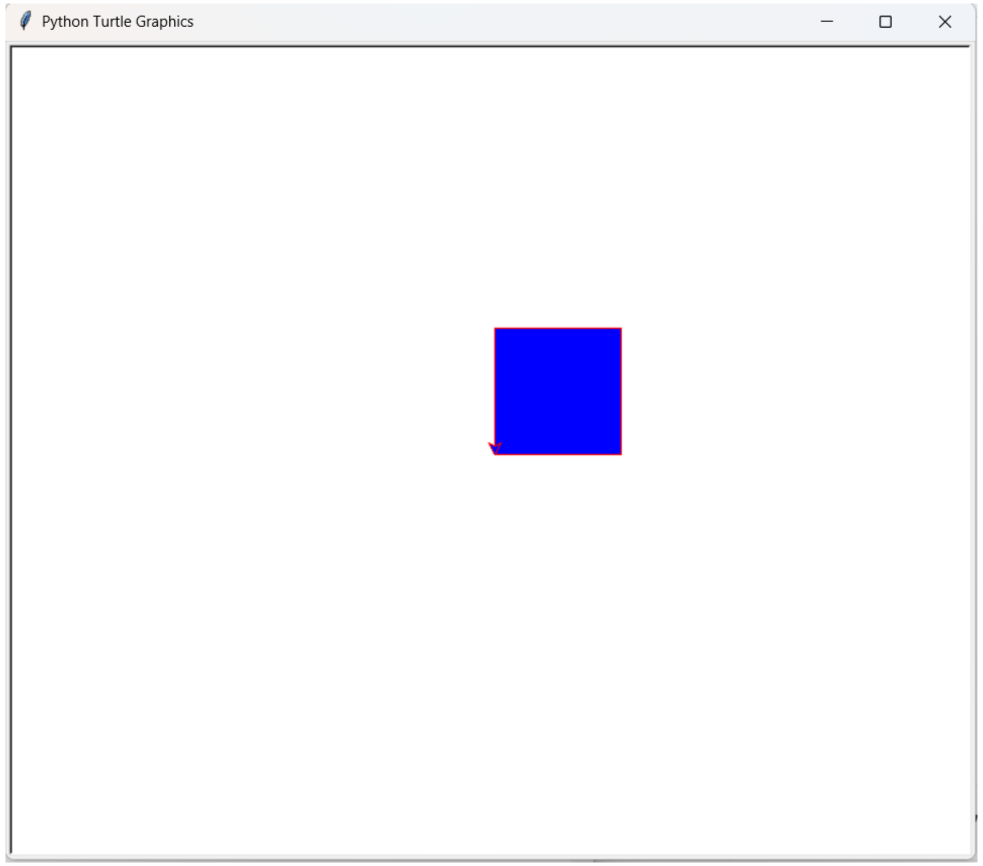
- begin_fill()
  - Purpose: Marks the start of a shape to be filled with the current fill color.
  - Parameters: None.
  - Usage: Used before drawing a closed shape, followed by end_fill().

SLO 3.2.3 A

c. Changing Turtle Attributes

- begin_fill()
  - Example:

```
import turtle
t = turtle.Turtle()

t.color("red", "blue")

t.begin_fill()

t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)

t.end_fill()


turtle.done()
```



c. Changing Turtle Attributes

SLO 3.2.3 A

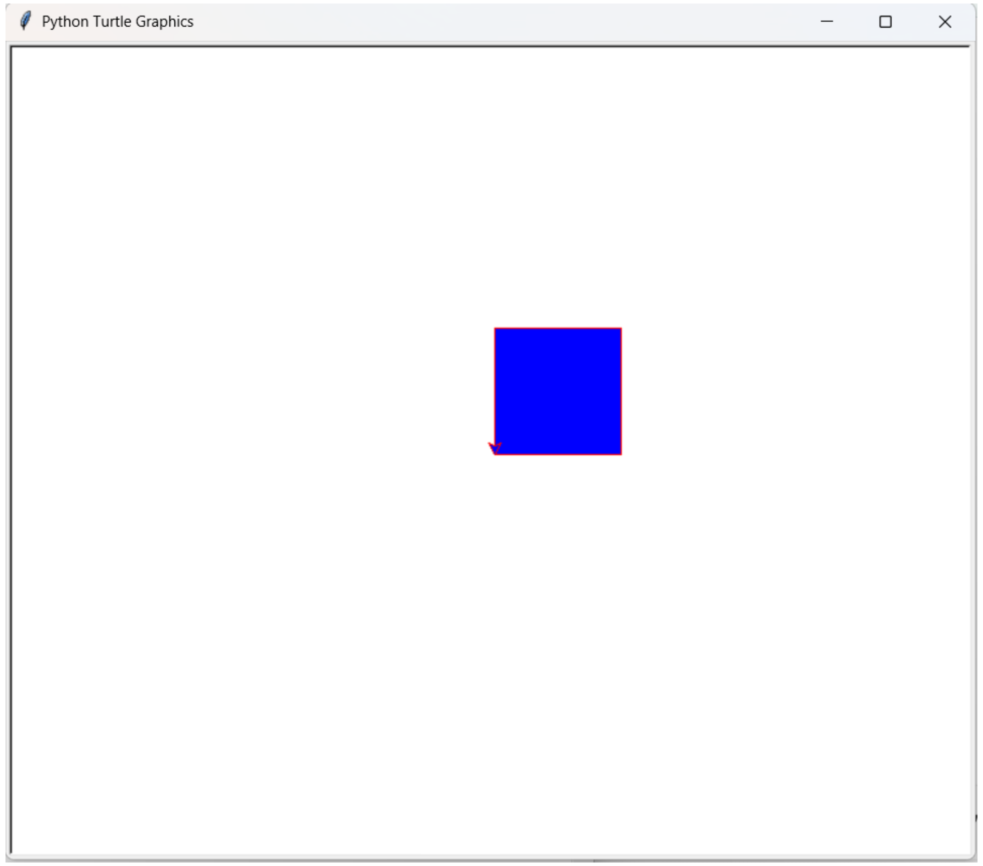
- end_fill()
  - Purpose: Ends the filling process, filling the shape drawn since begin_fill() with the current fill color.
  - Parameters: None.
  - Usage: Completes the fill for a closed shape.

SLO 3.2.3 A

c. Changing Turtle Attributes

- end_fill()
  - Example:

```
import turtle
t = turtle.Turtle()

t.color("red", "blue")

t.begin_fill()

t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)
t.left(90)
t.forward(100)

t.end_fill()


turtle.done()
```



c. Changing Turtle Attributes

SLO 3.2.3 A

- penup()
  - Purpose: Lifts the pen, so the turtle moves without drawing.
  - Parameters: None.
  - Usage: Allows repositioning the turtle without leaving a trail.

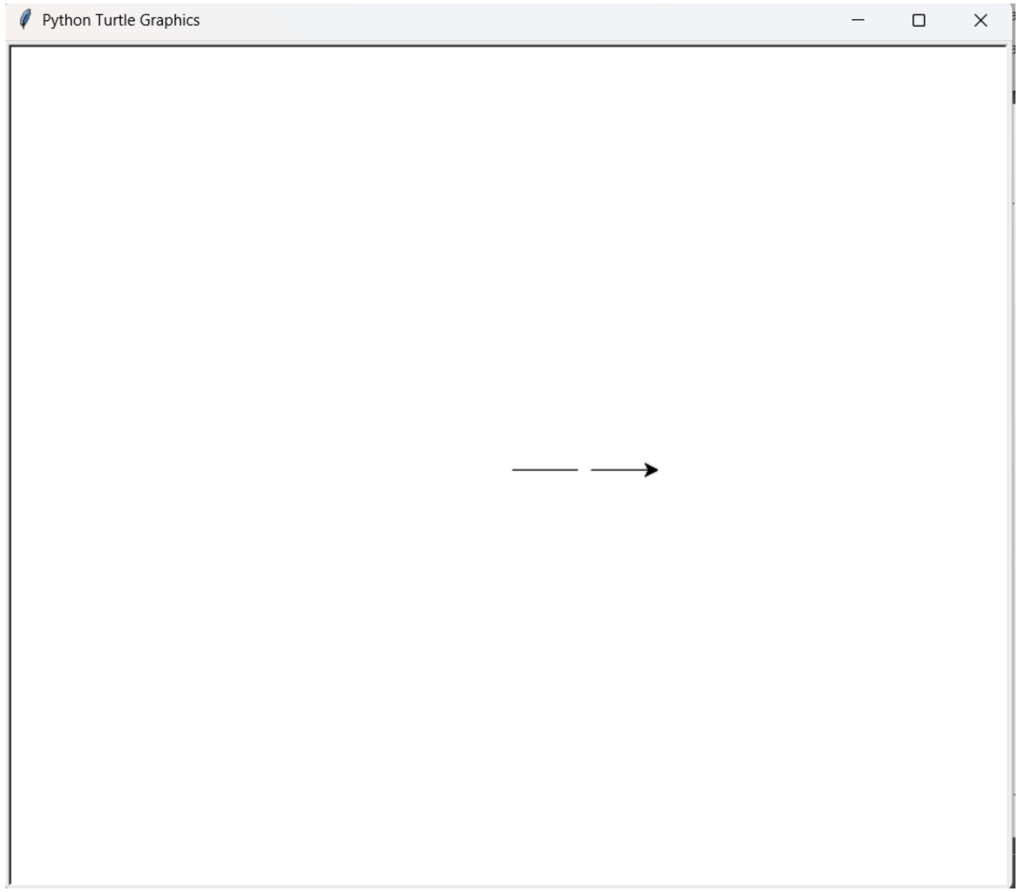
- pendown()
  - Purpose: Lowers the pen, so the turtle draws a line when it moves.
  - Parameters: None.
  - Usage: Resumes drawing after penup().

c. Changing Turtle Attributes

SLO 3.2.3 A

- penup()/pendown()
  - Example:

```
import turtle
t = turtle.Turtle()

t.forward(50)

t.penup()
t.forward(10)
t.pendown()

t.forward(50)

turtle.done()
```



c. Changing Turtle Attributes

SLO 3.2.3 A

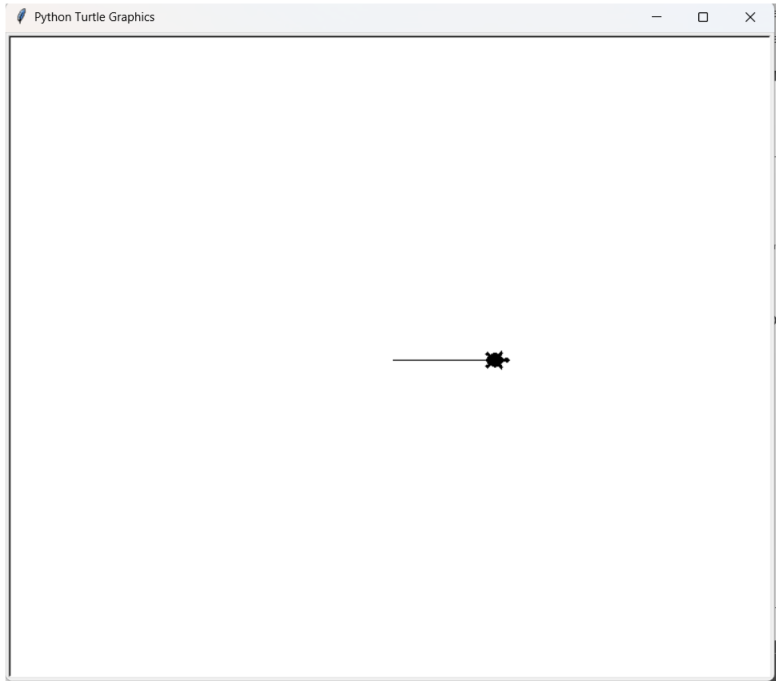
- speed(speed)
  - Purpose: Sets or returns the turtle’s drawing speed.
  - Parameters: speed (int or string) – 0 (fastest), 1-10 (slow to fast), or strings like "fastest", "fast", "normal", "slow", "slowest".
  - Usage: Controls animation speed; 0 disables animation for instant drawing.

SLO 3.2.3 A

c. Changing Turtle Attributes

- shape(name)
  - Purpose: Sets or returns the turtle's shape.
  - Parameters: name (string) – shape name (e.g., "arrow", "turtle", "circle", "square", "triangle", "classic").
  - Usage: Changes the turtle's appearance for stamps or visual feedback.
  - Example:

```
import turtle
t = turtle.Turtle()

t.shape("turtle")

t.forward(100)


turtle.done()
```



d. Preset Shapes

SLO 3.2.3 A

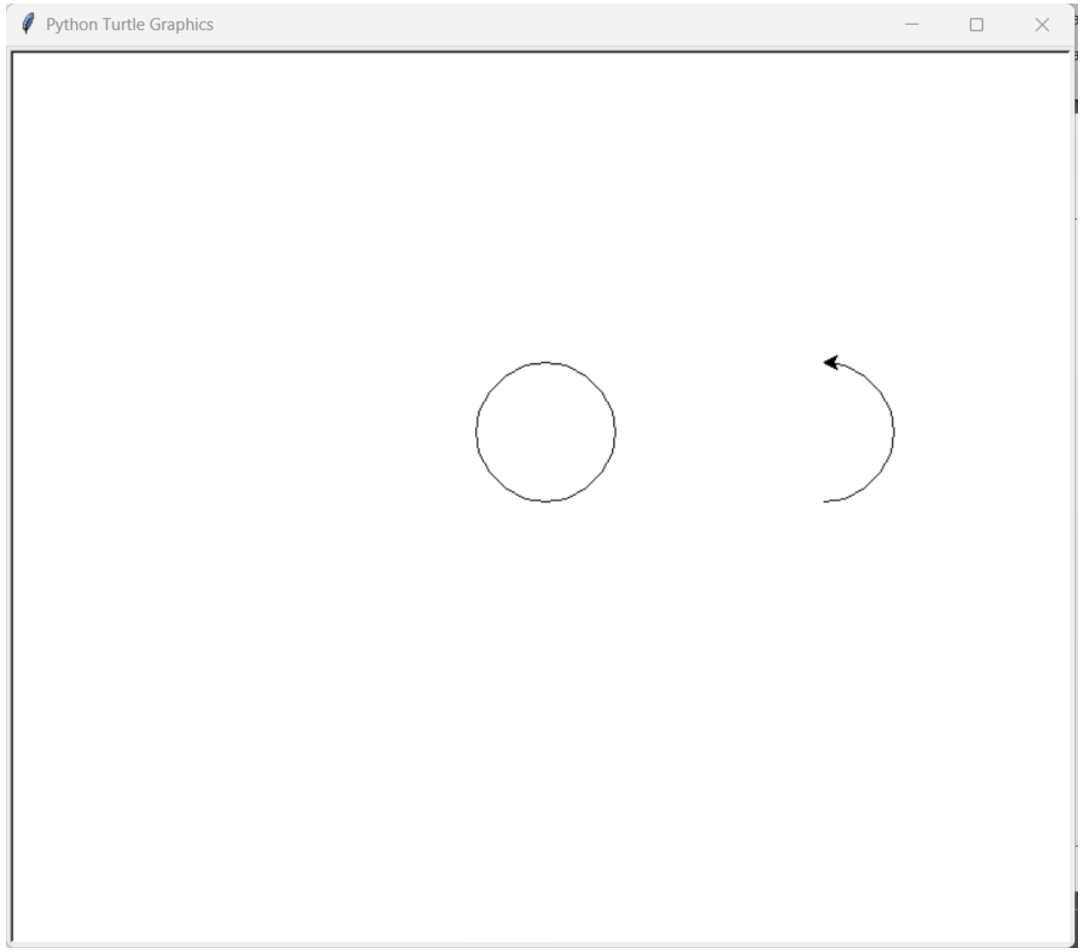
- circle(radius, extent=None, steps=None)
  - Purpose: Draws a circle or arc with the specified radius.
  - Parameters:
    - radius (float) – radius of the circle (positive for counterclockwise, negative for clockwise).
    - extent (float, optional) – angle of the arc (e.g., 180 for a semicircle).
    - steps (int, optional) – number of steps to approximate the circle (for polygons).
  - Usage: Draws a full circle or partial arc; can be filled if used with begin_fill() and end_fill().

SLO 3.2.3 A

d. Preset Shapes

- circle(radius, extent=None, steps=None)
  - Example

```
import turtle
t = turtle.Turtle()

# Draw a circle with radius 50 pixels
t.circle(50)

t.penup()
t.forward(200)
t.pendown()

# Draws a semicircle
t.circle(50, 180)

turtle.done()
```



d. Preset Shapes

SLO 3.2.3 A

- dot(size=None, *color)
  - Purpose: Draws a filled circular dot at the turtle's current position. Does not change the turtle's position or orientation.
  - Parameters:
    - size (optional, integer or float):
      - Specifies the diameter of the dot in pixels.
      - If None, the dot's size is calculated as pensize + 4 (the current pen size plus 4 pixels).
      - Must be a positive number; larger values create bigger dots.
    - *color (optional, string or RGB tuple):
      - Specifies the fill color of the dot.
      - Can be a color name (e.g., "red", "blue") or an RGB tuple (e.g., (1, 0, 0) for red).
      - If not provided, the dot uses the turtle's current fillcolor (set via fillcolor() or color()).

3.3 Libraries

SLO	Students should be able to	Cognitive Level
3.3.1	describe the purpose of libraries;	R
3.3.2	write the syntax of installing a library in Python language;	A
3.3.3	write a Python program that imports the datetime module and uses the following arguments: a. %A, b. %B, c. %C, d. %D, e. %H, f. %I, g. %S, h. %Y;	A

SLO 3.3.1 U

describe the purpose of libraries;

SLO 3.3.1 U

Describe the purpose of Libraries

- Libraries in Python are collections of pre-written code, functions, and modules that extend the language's capabilities, allowing developers to perform specific tasks without writing code from scratch.
- Their primary purpose is to simplify and accelerate development by providing reusable, tested, and optimized tools for common operations.

Describe the purpose of Libraries

SLO 3.3.1 U

Here’s a breakdown of their purpose:

- Code Reusability: Libraries provide pre-built functions and classes, enabling developers to reuse code for tasks like data processing, file handling, or complex computations, saving time and effort.
- Modularity: They organize code into modules, making it easier to manage and integrate specific functionalities (e.g., math for mathematical operations or requests for HTTP requests).
- Efficiency: Libraries are often optimized for performance, offering efficient implementations of algorithms or data structures (e.g., numpy for numerical computations).

SLO 3.3.1 U

Describe the purpose of Libraries

- Specialized Functionality: They provide tools for specific domains, such as:
  - Data Science: pandas for data manipulation, scikit-learn for machine learning.
  - Web Development: flask or django for building web applications.
  - Networking: socket or requests for network communication.
  - Graphics/GUI: pygame or tkinter for graphical interfaces.
- Community-Driven Development: Many libraries are open-source, maintained by communities, ensuring regular updates, bug fixes, and broad compatibility (e.g., libraries hosted on PyPI).
- Simplifying Complex Tasks: Libraries abstract complex logic, allowing developers to focus on high-level tasks. For example, matplotlib simplifies plotting, and beautifulsoup makes web scraping intuitive.

SLO 3.3.2 A

write the syntax of installing a library in Python language;

SLO 3.3.2 A

Syntax of installing a Library in Python

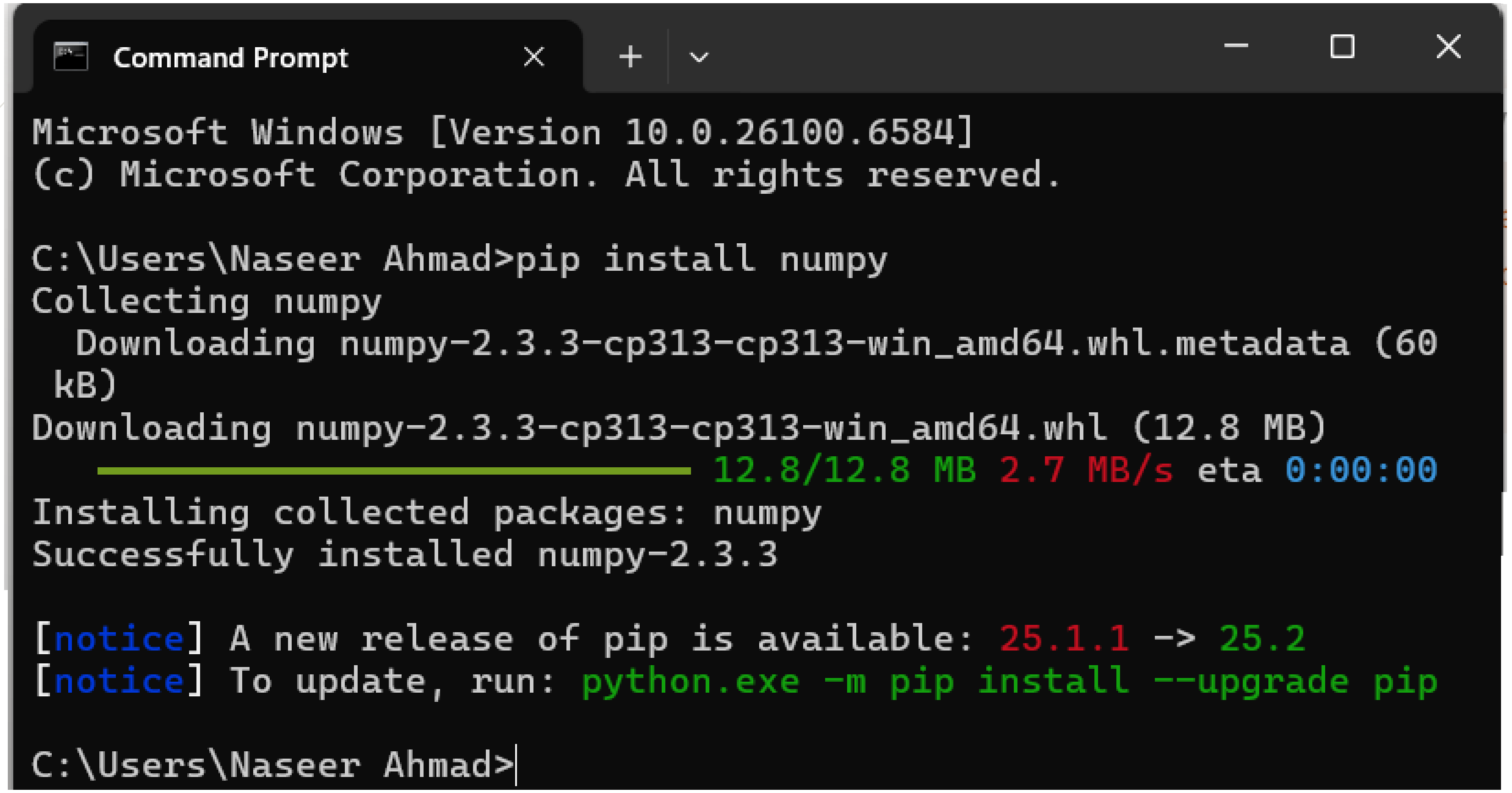
- To install a library in Python, you typically use the pip package manager. Below is the syntax for installing a Python library:

```
pip install library_name
```

- library_name: Replace with the name of the library (e.g., numpy, pandas, requests).
- Run this command in your terminal, command prompt, or shell.
- Ensure you have pip installed (it comes with Python by default in most cases).

Syntax of installing a Library in Python

SLO 3.3.2 A



```
Microsoft Windows [Version 10.0.26100.6584]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Naseer Ahmad>pip install numpy
Collecting numpy
  Downloading numpy-2.3.3-cp313-cp313-win_amd64.whl.metadata (60
 kB)
Downloading numpy-2.3.3-cp313-cp313-win_amd64.whl (12.8 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.8/12.8 MB 2.7 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-2.3.3

[notice] A new release of pip is available: 25.1.1 -> 25.2
[notice] To update, run: python.exe -m pip install --upgrade pip

C:\Users\Naseer Ahmad>
```

SLO 3.3.2 A

Variations of installing a Library in Python

- Install a specific version:

  ```
  pip install library_name==version
  ```

  Example: pip install numpy==1.25.0
- Upgrade an existing library:

  ```
  pip install --upgrade library_name
  ```

  Example: pip install --upgrade pandas
- Install for a specific Python version (if multiple Python versions are installed):

  ```
  python3 -m pip install library_name
  py -m pip install library_name
  ```

Check Library in Python

SLO 3.3.2 A

- To check if a library is installed:

```
pip show library_name
```

- To list all installed libraries:

```
pip list
```

SLO 3.3.3 A

Write a Python program that imports the datetime module and uses the following arguments:

a. %A,
b. %B,
c. %C,
d. %D,
e. %H,
f. %I,
g. %S,
h. %Y;

SLO 3.3.3 A

Datetime Module

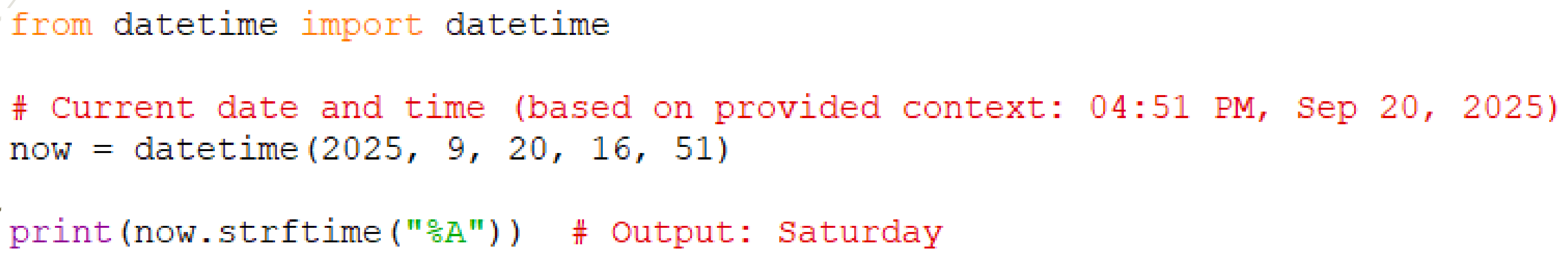
- The datetime module in Python provides classes for manipulating dates and times. It includes the strftime() method, which formats datetime objects into strings using specific format codes (like %A, %B, etc.)
- To use the module, import it:

```
import datetime
```

SLO 3.3.3 A

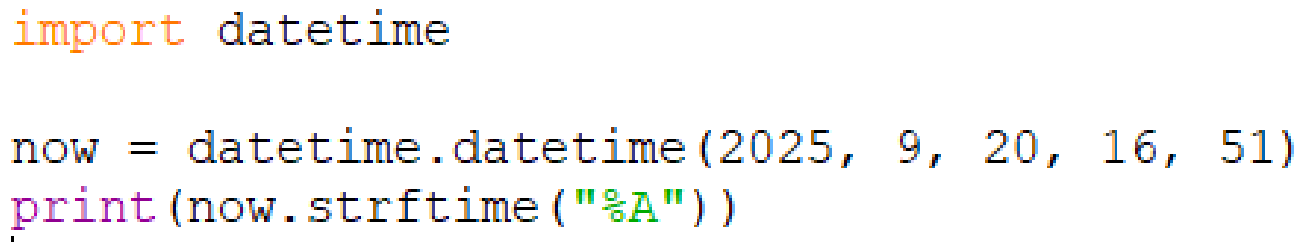
Datetime Module

- The strftime() method (string format time) converts a datetime object into a formatted string based on format codes.



```
from datetime import datetime

# Current date and time (based on provided context: 04:51 PM, Sep 20, 2025)
now = datetime(2025, 9, 20, 16, 51)

print(now.strftime("%A"))  # Output: Saturday
```

OR



```
import datetime

now = datetime.datetime(2025, 9, 20, 16, 51)
print(now.strftime("%A"))
```

Datetime Module

SLO 3.3.3 A

- %A: Full weekday name
  - Returns the full name of the day of the week.
  - Example: `print(now.strftime("%A"))  # Output: Saturday`
- %B: Full month name
  - Returns the full name of the month.
  - Example: `print(now.strftime("%B"))  # Output: September`
- %C: Century
  - Returns the century (year divided by 100, truncated to an integer) as a two-digit number.
  - Example: `print(now.strftime("%C"))  # Output: 20 (for 2025, century is 20th)`

Datetime Module

SLO 3.3.3 A

- %D: Date in MM/DD/YY format
  - Equivalent to %m/%d/%y (month/day/year with two-digit year).
  - Example: `print(now.strftime("%D"))  # Output: 09/20/25`
- %H: Hour (24-hour clock, 00-23)
  - Returns the hour in 24-hour format as a two-digit number.
  - Example: `print(now.strftime("%H"))  # Output: 16 (4 PM in 24-hour format)`
- %I: Hour (12-hour clock, 01-12)
  - Returns the hour in 12-hour format as a two-digit number.
  - Example: `print(now.strftime("%I"))  # Output: 04 (4 PM in 12-hour format)`

Datetime Module

SLO 3.3.3 A

- %S: Second
  - Returns the second as a two-digit number (00-59).
  - Example: `print(now.strftime("%S"))  # Output: 51`
- %Y: Year with century
  - Returns the full year, including the century, as a four-digit number.
  - Example: `print(now.strftime("%Y"))  # Output: 2025`

Datetime Module

SLO 3.3.3 A

```
from datetime import datetime

# Create a datetime object
current_time = datetime.now()

# Using the specified format codes
weekday = current_time.strftime("%A")  # Full weekday name
month = current_time.strftime("%B")    # Full month name
century = current_time.strftime("%C")  # Century
date_format = current_time.strftime("%D")  # MM/DD/YY format
hour_24 = current_time.strftime("%H")  # Hour (24-hour clock)
hour_12 = current_time.strftime("%I")  # Hour (12-hour clock)
seconds = current_time.strftime("%S")  # Seconds
year = current_time.strftime("%Y")     # Year with century

# Print individual outputs
print(f"Full Weekday Name (%A): {weekday}")
print(f"Full Month Name (%B): {month}")
print(f"Century (%C): {century}")
print(f"Date in MM/DD/YY (%D): {date_format}")
print(f"Hour (24-hour, %H): {hour_24}")
print(f"Hour (12-hour, %I): {hour_12}")
print(f"Seconds (%S): {seconds}")
print(f"Year (%Y): {year}")

# Combined formatted string
combined = current_time.strftime("Today is %A, %B %D, %H:%I:%S %Y")
print(f"\nCombined Format: {combined}")
```

```
Full Weekday Name (%A): Saturday
Full Month Name (%B): September
Century (%C): 20
Date in MM/DD/YY (%D): 09/20/25
Hour (24-hour, %H): 17
Hour (12-hour, %I): 05
Seconds (%S): 03
Year (%Y): 2025

Combined Format: Today is Saturday, September 09/20/25, 17:05:03 2025
```

3.4 Constants and Variables in Python Programming

SLO	Students should be able to	Cognitive Level
3.4.1	define the following basic data types in Python programming: a. char, b. str, c. int, d. float, e. bool;	R
3.4.2	differentiate between variable and constant;	U
3.4.3	differentiate between local and global variables;	U
3.4.4	write valid variable names based on the variable naming rules;	A
3.4.5	write a Python program to a. utilise different data types, b. demonstrate implicit and explicit type casting of variables;	A

SLO 3.4.1 R

define the following basic data types in Python programming:

a. char,
b. str,
c. int,
d. float,
e. bool;

SLO 3.4.1 R

Basic data types in Python programming

- In Python, data types define the kind of data a variable can hold.
- Python is dynamically typed, meaning you don't explicitly declare the data type when creating a variable.

1. char:
   - Definition: Python does not have a distinct char (character) data type like some other programming languages (e.g., C or Java). Instead, a single character is represented as a string (str) of length
   - Purpose: Used to store a single character, such as a letter, digit, or symbol.
   - Example:

```
char = 'A'  # A single character is a string in Python
print(char)  # Output: A
print(type(char))  # Output: <class 'str'>
```

Basic data types in Python programming

SLO 3.4.1 R

2. str (String):
   - Definition: A sequence of characters (letters, digits, symbols, or spaces) enclosed in single quotes (') or double quotes ("). Strings are immutable, meaning they cannot be changed after creation.
   - Purpose: Used to store and manipulate text, such as names, sentences, or any sequence of characters.
   - Example:

```
text = "Hello, World!"
print(text)  # Output: Hello, World!
print(type(text))  # Output: <class 'str'>
print(len(text))  # Output: 13 (length of string)
```

SLO 3.4.1 R

Basic data types in Python programming

3. int (Integer):

- Definition: Represents whole numbers (positive, negative, or zero) without a decimal point. Integers in Python have unlimited precision, meaning they can store arbitrarily large numbers.
- Purpose: Used for counting, indexing, or any numeric value without fractions.
- Example:

```
number = 42
negative = -10
print(number)  # Output: 42
print(type(number))  # Output: <class 'int'>
print(negative + 5)  # Output: -5
```

Basic data types in Python programming

SLO 3.4.1 R

4. float (Floating-Point Number):
   - Definition: Represents numbers with a decimal point or in scientific notation (e.g., 3.14 or 1.5e2). Floats are used for precise fractional values but have limited precision due to computer memory.
   - Purpose: Used for calculations involving fractions, measurements, or continuous values.
   - Example:

```
pi = 3.14159
scientific = 1.5e2  # 1.5 * 10^2 = 150.0
print(pi)  # Output: 3.14159
print(type(pi))  # Output: <class 'float'>
print(scientific)  # Output: 150.0
```

SLO 3.4.1 R

Basic data types in Python programming

5. bool (Boolean):
   - Definition: Represents one of two values: True or False. Booleans are used for logical operations and conditionals.
   - Purpose: Used in decision-making, loops, and logical expressions to control program flow.
   - Example:

```
is_active = True
is_empty = False
print(is_active)  # Output: True
print(type(is_empty))  # Output: <class 'bool'>
print(5 > 3)  # Output: True (result of comparison)
```

SLO 3.4.2 U

differentiate between variable and constant;

SLO 3.4.2 U

What is a Variable in Python

- A variable in Python is a named storage location in memory that holds a value.
- It acts as a container for storing data, which can be of various types (e.g., int, str, float, bool).
- Variables allow you to reference and manipulate data throughout your program.
- Python is dynamically typed, meaning you don't need to specify the data type when creating a variable, and the type is inferred from the assigned value.

SLO 3.4.2 U

What is a Variable in Python

- Key Characteristics:
  - Variables are created by assigning a value using the = operator.
  - The value of a variable can change during program execution.
  - Variable names are case-sensitive (e.g., age and Age are different).
  - Variable names must follow rules.

Create a Variable in Python

SLO 3.4.2 U

- To create a variable, assign a value to a name using the = operator.
- The syntax is:

variable_name = value

```
# Creating variables with different data types
age = 25   # Integer
name = "Alice"   # String
height = 5.9   # Float
is_student = True   # Boolean
```

Ram

		A	l	i	c	e
25						
			5.9			
	True					

SLO 3.4.2 U

Variables vs. Constants in Python

- Variable
  - Definition: A variable is a named entity whose value can change during program execution.
  - Purpose: Used to store data that may be modified, such as counters, user input, or temporary results.
  - Example

```
counter = 0
counter = counter + 1  # Value changes to 1
print(counter)  # Output: 1
```

Variables vs. Constants in Python

SLO 3.4.2 U

- Constant
  - Definition: A constant is a named entity whose value is intended to remain unchanged throughout the program. In Python, there is no strict enforcement of constants (unlike languages like C++ or Java), but by convention, constants are defined using uppercase letters to indicate they should not be modified.
  - Purpose: Used for fixed values like mathematical constants (e.g., PI) or configuration settings that should remain consistent.
  - Example

```
PI = 3.14159  # Constant by convention (uppercase)
GRAVITY = 9.8
print(PI)  # Output: 3.14159
# PI = 3.14  # Avoid changing constants (not enforced by Python)
```

SLO 3.4.3 U

differentiate between local and global variables;

Local and Global variables

SLO 3.4.3 U

- Local Variables
  - Definition: Variables defined inside a function or a block of code (e.g., within a function's body).
  - Scope: Accessible only within the function or block where they are defined.
  - Lifetime: Exists only during the execution of the function or block. Once the function exits, the variable is destroyed.
  - Example:

```python
def my_function():
    local_var = 10  # Local variable
    print(local_var)

my_function()  # Output: 10
# print(local_var)  # Error: NameError, local_var is not defined
```

Local and Global variables

SLO 3.4.3 U

- Global Variables
  - Definition: Variables defined outside any function or block, typically at the module level.
  - Scope: Accessible throughout the entire module, including inside functions (unless shadowed by a local variable with the same name).
  - Lifetime: Exists for the entire duration of the program's execution.
  - Example:

```
global_var = 20  # Global variable

def my_function():
    print(global_var)  # Can access global variable

my_function()  # Output: 20
print(global_var)  # Output: 20
```

SLO 3.4.4 A

write valid variable names based on the variable naming rules;

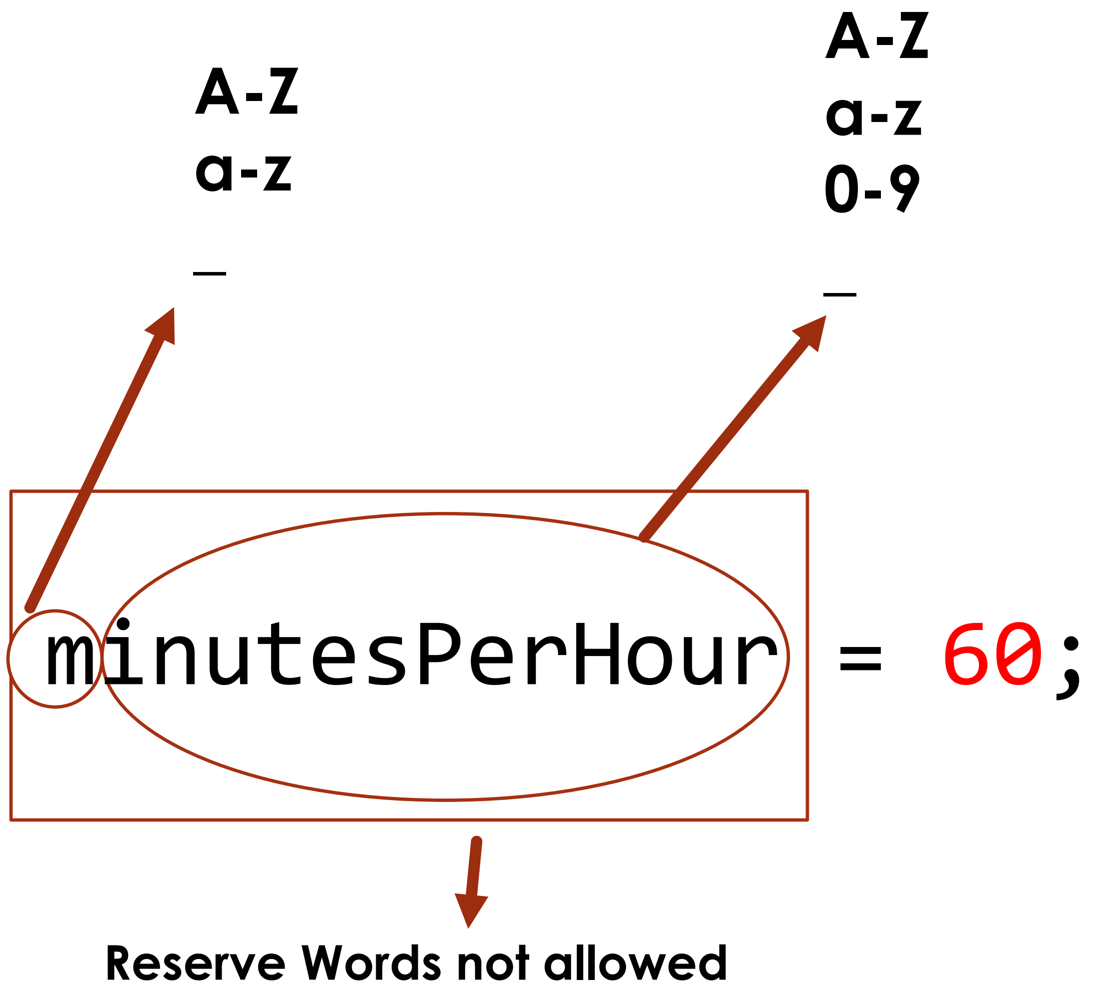
SLO 3.4.4 A

Valid variable names

- In Python, variable names must follow specific rules to be valid and should adhere to conventions for readability and maintainability.
- A variable name must begin with an alphabetic character (A-Z or a-z) or an underscore (_). It cannot start with a digit.
- After the first character, the variable name can contain letters (A-Z or a-z), digits (0-9), or underscores (_).
- A variable name cannot be a reserved keyword such as if, for, while, class, True, False.
- Variable names are case-sensitive, meaning age, Age, and AGE are treated as different variables.
- Spaces are not allowed in variable names.
- Only underscores (_) are allowed; other special characters (e.g., @, #, $) are not.

Valid variable names

SLO 3.4.4 A



```
minutesPerHour = 60;
```

SLO 3.4.5 A

write a Python program to

a. utilise different data types,

b. demonstrate implicit and explicit type casting of variables;

Utilise different data types

SLO 3.4.5 A

```python
# Define variables with different data types
name = "Alice"                                 # str: String for storing name
age = 25                                      # int: Integer for storing age
height = 5.8                                  # float: Floating-point number for height
is_student = True                             # bool: Boolean for student status

# Constants (by convention)
MAX_ATTEMPTS = 3                              # int: Constant for maximum attempts
PI = 3.14159                                  # float: Constant for mathematical value


# Print results
print(f"User Information:")
print(f"Name (str): {name}")
print(f"Age (int): {age}")
print(f"Height (float): {height} feet")
print(f"Is Student (bool): {is_student}")
print(f"PI (float): {PI}")
print(f"Attempts Left (int): {MAX_ATTEMPTS}")
```

```
User Information:
Name (str): Alice
Age (int): 25
Height (float): 5.8 feet
Is Student (bool): True
PI (float): 3.14159
Attempts Left (int): 3
```

SLO 3.4.5 A

Implicit and Explicit type casting of variables

- In Python, type casting (or type conversion) refers to converting a variable from one data type to another.
- Python supports two types of type casting: implicit and explicit.

1. **Implicit Type Casting**
   - Definition: Implicit type casting is when Python automatically converts one data type to another without explicit instructions from the programmer. This typically occurs during operations involving compatible data types to prevent data loss or ensure the operation is valid.
   - How it Works: Python promotes a "smaller" data type (e.g., int) to a "larger" or more precise data type (e.g., float) when performing operations.
   - Common Scenarios: Occurs in arithmetic operations or when combining certain types.
   - Key Point: Implicit casting is safe and handled by Python to avoid errors or loss of precision.

SLO 3.4.5 A

Implicit and Explicit type casting of variables

1. Implicit Type Casting

```
# Integer and float in an arithmetic operation
a = 5         # int
b = 2.5       # float
result = a + b  # Python converts 'a' to float for the operation
print(result)  # Output: 7.5
print(type(result))  # Output: <class 'float'>

# Integer and boolean (True = 1, False = 0)
c = 10        # int
d = True      # bool
result2 = c + d  # Python treats True as 1
print(result2)  # Output: 11
print(type(result2))  # Output: <class 'int'>
```

```
7.5
<class 'float'>
11
<class 'int'>
```

SLO 3.4.5 A

Implicit and Explicit type casting of variables

2. **Explicit Type Casting**

- Definition: Explicit type casting is when the programmer manually converts a variable from one data type to another using built-in functions like int(), float(), str(), or bool().
- How it Works: You explicitly call a function to convert the value to the desired type. This may lead to data loss if the conversion is not fully compatible (e.g., converting a float to an integer truncates the decimal part).
- Common Scenarios: Used when you need a specific type for an operation, user input handling, or formatting output.
- Key Point: Explicit casting gives you control but requires caution to avoid errors or unintended data loss.

Implicit and Explicit type casting of variables

SLO 3.4.5 A

2. Explicit Type Casting

- Built-in Functions for Explicit Type Casting
  - int(): Converts to an integer (truncates decimals for floats).
  - float(): Converts to a floating-point number.
  - str(): Converts to a string.
  - bool(): Converts to a boolean (True or False).

SLO 3.4.5 A

Implicit and Explicit type casting of variables

2. Explicit Type Casting

```
# Converting between different data types
number_str = "42"  # str
number_int = int(number_str)  # Convert string to integer
print(number_int)  # Output: 42
print(type(number_int))  # Output: <class 'int'>

number_float = float(number_str)  # Convert string to float
print(number_float)  # Output: 42.0
print(type(number_float))  # Output: <class 'float'>

value = 3.99  # float
value_int = int(value)  # Convert float to integer (truncates decimal)
print(value_int)  # Output: 3
print(type(value_int))  # Output: <class 'int'>

is_active = bool(1)  # Convert integer to boolean (non-zero = True)
print(is_active)  # Output: True
print(type(is_active))  # Output: <class 'bool'>

num = 100  # int
num_str = str(num)  # Convert integer to string
print(num_str + " points")  # Output: 100 points
print(type(num_str))  # Output: <class 'str'>
```

```
42
<class 'int'>
42.0
<class 'float'>
3
<class 'int'>
True
<class 'bool'>
100 points
<class 'str'>
```

3.5 Input Output Handling

SLO	Students should be able to	Cognitive Level
3.5.1	describe the purpose of using eval( ) in Python;	R
3.5.2	write a Python program to use the eval( ) function;	A
3.5.3	write a program to display a message and the value of the variable using the print( ) statement;	A
3.5.4	write a program for taking input during the execution of a program using the input( ) statement;	A

SLO 3.5.1 R

describe the purpose of using eval() in Python;

SLO 3.5.1 R

Using eval() in Python

- The eval() function in Python is a built-in function that evaluates a string as a Python expression and returns the result.
- It is used to dynamically execute Python code stored in a string, allowing for runtime evaluation of expressions.

Purpose of eval() in Python

- Dynamic Code Execution:
  - eval() allows you to execute Python expressions (e.g., arithmetic, logical, or function calls) stored as strings at runtime.
  - Useful when the expression to evaluate is not known until the program runs, such as user input or dynamically generated code.
- Converting Strings to Expressions:
  - Converts string representations of valid Python expressions (e.g., "2 + 3", "len('hello')") into their evaluated results (e.g., 5, 5).
  - Acts as a powerful tool for parsing and computing string-based expressions.

Using eval() in Python

SLO 3.5.1 R

Purpose of eval() in Python

- Flexibility in Calculations:
  - Enables dynamic computation, such as evaluating mathematical expressions provided by users or read from files.Example: A calculator program where users input expressions like "5 * 3 + 2".
- Accessing Variables and Functions:
  - eval() can access variables, functions, and objects in the current scope (local or global), allowing it to interact with the program's environment.

Syntax

```
eval(expression, globals=None, locals=None)
```

- expression: A string containing a valid Python expression.
- globals: A dictionary specifying the global namespace (optional).
- locals: A dictionary specifying the local namespace (optional).Returns: The result of the evaluated expression.

SLO 3.5.2 A

write a Python program to use the eval( ) function;

SLO 3.5.2 A

Write a Python program to use the eval( ) function

```python
# Simple arithmetic expression
expression = "2 + 3 * 4"
result = eval(expression)
print(result)  # Output: 14

# Using variables in eval
x = 10
expression = "x * 2"
result = eval(expression)
print(result)  # Output: 20
```

```
14
20
```

SLO 3.5.3 A

write a program to display a message and the value of the variable using the print() statement;

SLO 3.5.3 A

Printing in Python

- In Python, the print() function is used to display output to the console or terminal. It is a built-in function that allows you to output text, variables, or expressions, making it essential for debugging, displaying results, or providing user feedback.
- Syntax

```
print(*objects, sep=' ', end='\n', file=None, flush=False)
```

  - *objects: One or more items to print (e.g., variables, strings, numbers). Multiple objects are converted to strings and concatenated.
  - sep: Separator between multiple objects (default is a single space ' ‘).
  - end: String appended after the last object (default is a newline '\n’).
  - file: Output stream (default is sys.stdout for console; can redirect to a file).
  - flush: If True, forces output to be written immediately (default is False).

Printing in Python

SLO 3.5.3 A

- Examples of print()
  - Basic Usage:

```
print("Hello, World!")  # Output: Hello, World!
```

  - Multiple Objects:

```
name = "Alice"
age = 25
print(name, age)  # Output: Alice 25
```

  - Custom Separator and End:

```
print("A", "B", "C", sep="-", end="!\n")  # Output: A-B-C!
```

  - F-strings (Python 3.6+): Embed expressions inside string literals

```
name = "Bob"
print(f"Name: {name}")  # Output: Name: Bob
```

SLO 3.5.4 A

write a program for taking input during the execution of a program using the input( ) statement;

SLO 3.5.4 A

Input in Python

- In Python, the input() function is used to capture user input from the console or terminal. It allows programs to interact with users by accepting data (typically as strings) during runtime.
- Purpose: Reads a line of text entered by the user from the standard input (usually the keyboard) and returns it as a string.
- Syntax:

```
input(prompt="")
```

  - prompt: An optional string displayed to the user before input is collected (e.g., "Enter your name: ").
  - Returns: A string containing the user's input, including any trailing newline (which is stripped).
- Key Features:
  - Always returns a str type, requiring explicit type casting (e.g., int(), float()) for other data types.
  - Blocks program execution until the user presses Enter.

SLO 3.5.4 A

Input in Python

- Examples of input()
  - Basic Usage:

```
name = input("Enter your name: ")
print(f"Hello, {name}!")  # Output: Hello, Alice! (if user enters "Alice")
```

  - Type Casting Input:

```
age = int(input("Enter your age: "))  # Convert string to int

# Output: Next year, you'll be 26 (if user enters "25")
print(f"Next year, you'll be {age + 1}")
```

3.6 Operators in Python Programming

SLO	Students should be able to	Cognitive Level
3.6.1	differentiate between operator and operand;	U
3.6.2	describe the following types of operators: a. assignment operator, b. membership operators, c. arithmetic operators, d. bitwise operators, e. comparison (relational) operators, f. logical operators;	U
3.6.3	write a Python program using the following operators; a. arithmetic operators, b. bitwise operators, c. comparison (relational) operators, d. logical operators;	A

SLO 3.6.1 U

differentiate between operator and operand;

Differentiate between operator and operand

SLO 3.6.1 U

- Operator:
  - A symbol or keyword that performs a specific operation on one or more values.
  - Examples include:
    - Arithmetic operators: +, -, *, /, //, %, **
    - Comparison operators: ==, !=, >, <, >=, <=
    - Logical operators: and, or, not
    - Assignment operators: =, +=, -=, etc.
    - Example: In a + b, the + is the operator.
- Operand:
  - The value or variable on which the operator operates.
  - Operands can be literals, variables, or expressions.
  - Example: In a + b, a and b are operands.

SLO 3.6.2 U

describe the following types of operators:

a. assignment operator,
b. membership operators,
c. arithmetic operators,
d. bitwise operators,
e. comparison (relational) operators,
f. logical operators;

Types of Operators: Assignment Operator

SLO 3.6.2 U

- Purpose: Assigns a value (or the result of an expression) to a variable or modifies an existing variable's value.
- Operators:
  - = : Assigns a value to a variable.
  - +=, -=, *=, /=, //=, %=, **=, etc. : Compound assignment operators that combine an arithmetic operation with assignment.
- How it works: The right-hand operand (value or expression) is evaluated and assigned to the left-hand operand (variable).
- Example:

```
x = 10          # Assigns 10 to x
x += 5          # Equivalent to x = x + 5, so x becomes 15
print(x)        # Output: 15
```

Types of Operators: Membership Operators

SLO 3.6.2 U

- Purpose: Tests whether a value is present in a sequence (e.g., list, string, tuple, set, dictionary) or not.
- Operators:
  - in : Returns True if the value is found in the sequence.
  - not in : Returns True if the value is not found in the sequence.
- How it works: The left-hand operand is the value to check, and the right-hand operand is the sequence.
- Example:

```
my_list = [1, 2, 3, 4]
print(2 in my_list)        # Output: True
print(5 not in my_list)    # Output: True
```

Types of Operators: Arithmetic Operators

SLO 3.6.2 U

- Purpose: Perform mathematical operations on numeric operands (integers, floats, etc.).
- Operators:
  - + : Addition
  - - : Subtraction
  - * : Multiplication
  - / : Division (returns float)
  - // : Floor division (returns integer, rounds down)
  - % : Modulus (returns remainder)
  - ** : Exponentiation (power)
- How it works: Operates on two operands (except unary + or -) to produce a result.

Types of Operators: Arithmetic Operators

SLO 3.6.2 U

- Example:

```
a, b = 10, 3
print(a + b)     # Output: 13
print(a / b)     # Output: 3.333...
print(a // b)    # Output: 3
print(a % b)     # Output: 1
print(a ** b)    # Output: 1000
```

Types of Operators: Bitwise Operators

SLO 3.6.2 U

- Purpose: Perform operations on the binary representations of integers.
- Operators:
  - & : Bitwise AND
  - | : Bitwise OR
  - ^ : Bitwise XOR
  - ~ : Bitwise NOT (unary, inverts bits)
  - << : Left shift (shifts bits left)
  - >> : Right shift (shifts bits right)
- How it works: Operates on the binary digits of the operands. For example, 5 & 3 compares binary 101 (5) and 011 (3) to produce 001 (1).

SLO 3.6.2 U

Types of Operators: Bitwise Operators

- Example:

```
a, b = 5, 3  # 5 = 101, 3 = 011 in binary
print(a & b)  # Output: 1 (101 & 011 = 001)
print(a | b)  # Output: 7 (101 | 011 = 111)
print(a ^ b)  # Output: 6 (101 ^ 011 = 110)

# Output: -6 (inverts bits of 101, considering two's complement)
print(~a)

print(a << 1) # Output: 10 (101 << 1 = 1010)
print(a >> 1) # Output: 2 (101 >> 1 = 010)
```

Types of Operators: Comparison (Relational) Op.

SLO 3.6.2 U

- Purpose: Compare two operands and return a boolean (True or False) based on the condition.
- Operators:
  - == : Equal to
  - != : Not equal to
  - > : Greater than
  - < : Less than
  - >= : Greater than or equal to
  - <= : Less than or equal to
- How it works: Compares the values of the left and right operands.

SLO 3.6.2 U

Types of Operators: Comparison (Relational) Op.

- Example:

```
x, y = 10, 20
print(x == y)    # Output: False
print(x != y)    # Output: True
print(x < y)     # Output: True
print(x >= y)    # Output: False
```

Types of Operators: Logical Operators

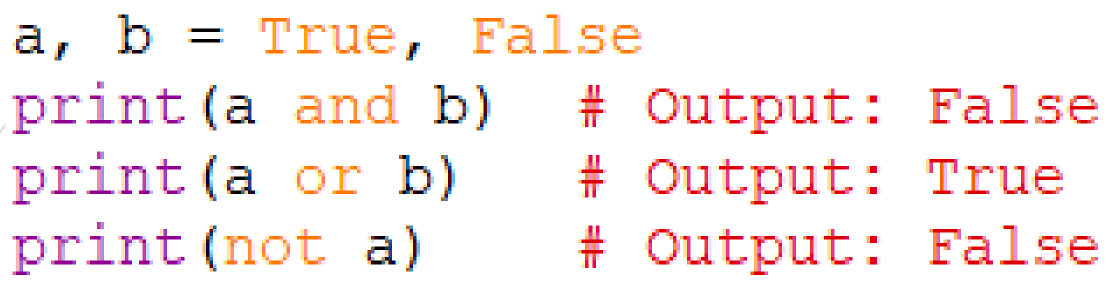
SLO 3.6.2 U

- Purpose: Combine boolean expressions or values to produce a boolean result.
- Operators:
  - and : Returns True if both operands are True.
  - or : Returns True if at least one operand is True.
  - not : Returns the opposite boolean value of the operand (unary).
- How it works: Evaluates operands in a boolean context. Python uses short-circuit evaluation (e.g., in and, if the first operand is False, the second is not evaluated)

SLO 3.6.2 U

Types of Operators: Logical Operators

- Example:



```python
a, b = True, False
print(a and b)   # Output: False
print(a or b)    # Output: True
print(not a)     # Output: False
```

SLO 3.6.3 A

write a Python program using the following operators;

a. arithmetic operators,
b. bitwise operators,
c. comparison (relational) operators,
d. logical operators;

SLO 3.6.3 A

write a Python program using the arithmetic operators

```python
# Global variables
num1 = 10
num2 = 3

# Local variables (within the main block)
result_add = num1 + num2
result_sub = num1 - num2
result_mul = num1 * num2
result_div = num1 / num2
result_floor_div = num1 // num2
result_mod = num1 % num2
result_exp = num1 ** num2

# Display results
print(f"Performing arithmetic operations with num1 = {num1} and num2 = {num2}\n")
print(f"Addition: {num1} + {num2} = {result_add}")
print(f"Subtraction: {num1} - {num2} = {result_sub}")
print(f"Multiplication: {num1} * {num2} = {result_mul}")
print(f"Division: {num1} / {num2} = {result_div}")
print(f"Floor Division: {num1} // {num2} = {result_floor_div}")
print(f"Modulus: {num1} % {num2} = {result_mod}")
print(f"Exponentiation: {num1} ** {num2} = {result_exp}")
```

write a Python program using the bitwise operators

SLO 3.6.3 A

```
#Global variables
num1 = 10  # Binary: 1010
num2 = 3   # Binary: 0011

# Local variables (within the main block)
result_and = num1 & num2   # Bitwise AND
result_or = num1 | num2    # Bitwise OR
result_xor = num1 ^ num2   # Bitwise XOR
result_not_num1 = ~num1    # Bitwise NOT (on num1)
result_left_shift = num1 << 2  # Left shift num1 by 2
result_right_shift = num1 >> 2 # Right shift num1 by 2

# Display results
print(f"Bitwise operations with num1 = {num1} and num2 = {num2}\n")
print(f"AND: {num1} & {num2} = {result_and}")
print(f"OR: {num1} | {num2} = {result_or}")
print(f"XOR: {num1} ^ {num2} = {result_xor}")
print(f"NOT (~num1): ~{num1} = {result_not_num1}")
print(f"Left Shift: {num1} << 2 = {result_left_shift}")
print(f"Right Shift: {num1} >> 2 = {result_right_shift}")
```

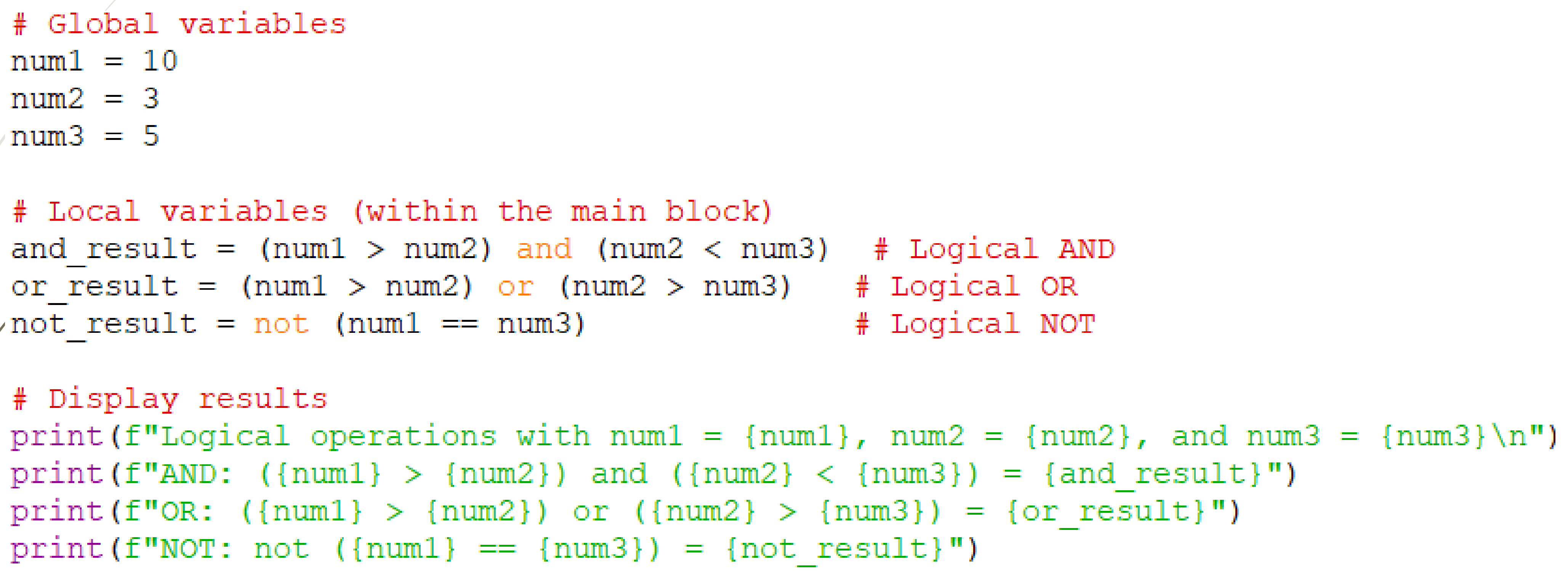
SLO 3.6.3 A

write a Python program using the comparison (relational) operators

```python
# Global variables
num1 = 10
num2 = 3

# Local variables (within the main block)
equal = num1 == num2  # Equal to
not_equal = num1 != num2  # Not equal to
greater = num1 > num2  # Greater than
less = num1 < num2  # Less than
greater_equal = num1 >= num2  # Greater than or equal to
less_equal = num1 <= num2  # Less than or equal to

# Display results
print(f"Comparison operations with num1 = {num1} and num2 = {num2}\n")
print(f"Equal: {num1} == {num2} = {equal}")
print(f"Not Equal: {num1} != {num2} = {not_equal}")
print(f"Greater: {num1} > {num2} = {greater}")
print(f"Less: {num1} < {num2} = {less}")
print(f"Greater or Equal: {num1} >= {num2} = {greater_equal}")
print(f"Less or Equal: {num1} <= {num2} = {less_equal}")
```

SLO 3.6.3 A

write a Python program using the logical operators



```python
# Global variables
num1 = 10
num2 = 3
num3 = 5

# Local variables (within the main block)
and_result = (num1 > num2) and (num2 < num3)  # Logical AND
or_result = (num1 > num2) or (num2 > num3)    # Logical OR
not_result = not (num1 == num3)               # Logical NOT

# Display results
print(f"Logical operations with num1 = {num1}, num2 = {num2}, and num3 = {num3}\n")
print(f"AND: ({num1} > {num2}) and ({num2} < {num3}) = {and_result}")
print(f"OR: ({num1} > {num2}) or ({num2} > {num3}) = {or_result}")
print(f"NOT: not ({num1} == {num3}) = {not_result}")
```

3.7 Selection Statements in Python Programming

SLO	Students should be able to	Cognitive Level
3.7.1	describe **if**, **if-else**, and **elif** statements;	U
3.7.2	write a Python program using the following statements: a. if, b. if-else, c. if-elif-else, d. nested if, e. pass;	A

SLO 3.7.1 R

describe if, if-else, and elif statements;

SLO 3.7.1 R

if statement

- The if statement checks a condition.
- If the condition is True, the block of code inside it runs.
- If it's False, the block is skipped.
- Syntax:

```
if condition:
    # code to execute if condition is true
```

If-else statement

SLO 3.7.1 R

- The if-else statement adds an alternative path —
- if the condition is True, one block runs;
- if it’s False, the other block runs.
- Syntax:

```
if condition:
    # code to execute if condition is true
else:
    # code if condition is false
```

elif statement

SLO 3.7.1 R

- The elif (short for “else if”) allows checking multiple conditions in sequence.
- As soon as one condition is True, its block runs and the rest are skipped.
- Syntax:

```
if condition1:
    # code if condition1 is true
elif condition2:
    # code if condition2 is true
else:
    # code if all conditions are false
```

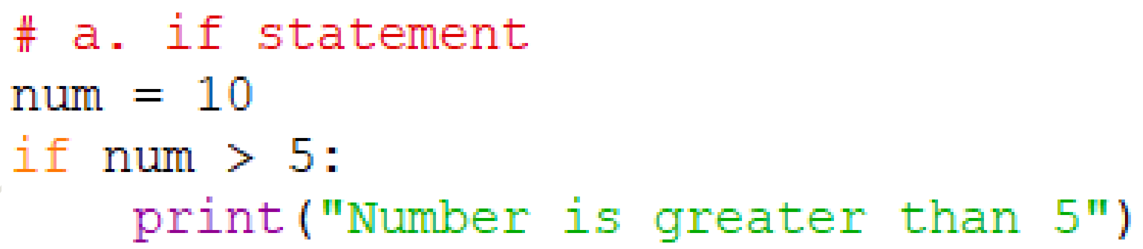
SLO 3.7.2 A

write a Python program using the following statements:

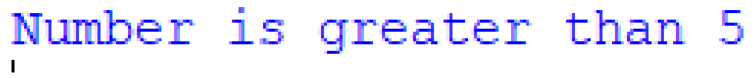
a. if,
b. if-else,
c. if-elif-else,
d. nested if,
e. pass;

write a Python program using the if statements

SLO 3.7.2 A



```python
# a. if statement
num = 10
if num > 5:
    print("Number is greater than 5")
```



```
Number is greater than 5
```

write a Python program using the if-else statements

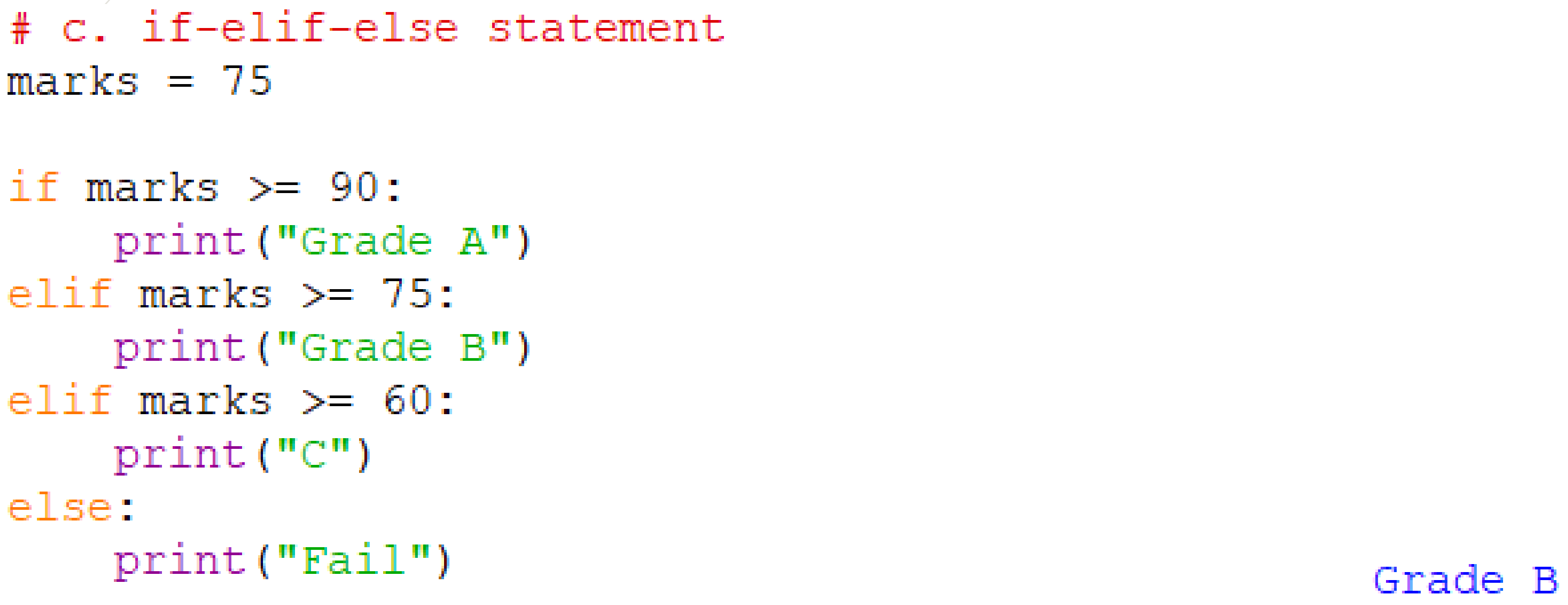
SLO 3.7.2 A

```python
# b. if-else statement
num = 3
if num % 2 == 0:
    print("Number is even")
else:
    print("Number is odd")
```

```
Number is odd
```

SLO 3.7.2 A

write a Python program using the if-elif-else statements



```python
# c. if-elif-else statement
marks = 75

if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 60:
    print("C")
else:
    print("Fail")
```

```
Grade B
```

Nested If

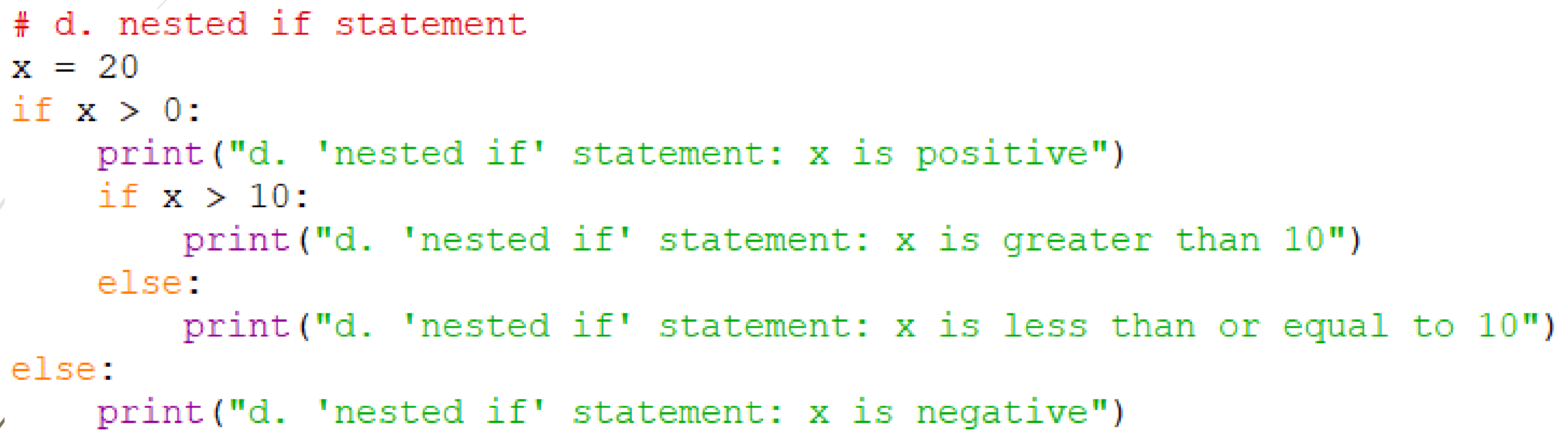
SLO 3.7.2 A

- A nested if statement in Python means putting one if statement inside another.
- It’s used when you need to check multiple conditions where one depends on another — like saying “If this is true, then check that.”
- Syntex:

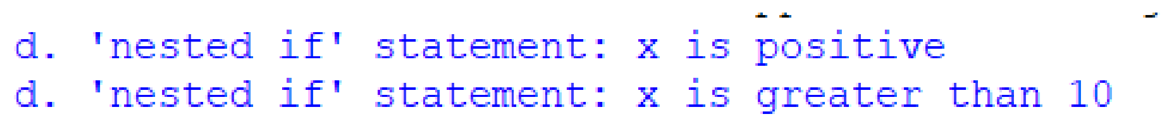
```
if condition1:
    # outer if block
    if condition2:
        # inner if block
        # executes only if both conditions are true
```

SLO 3.7.2 A

write a Python program using the nested if statements



```python
# d. nested if statement
x = 20
if x > 0:
    print("d. 'nested if' statement: x is positive")
    if x > 10:
        print("d. 'nested if' statement: x is greater than 10")
    else:
        print("d. 'nested if' statement: x is less than or equal to 10")
else:
    print("d. 'nested if' statement: x is negative")
```



```
d. 'nested if' statement: x is positive
d. 'nested if' statement: x is greater than 10
```

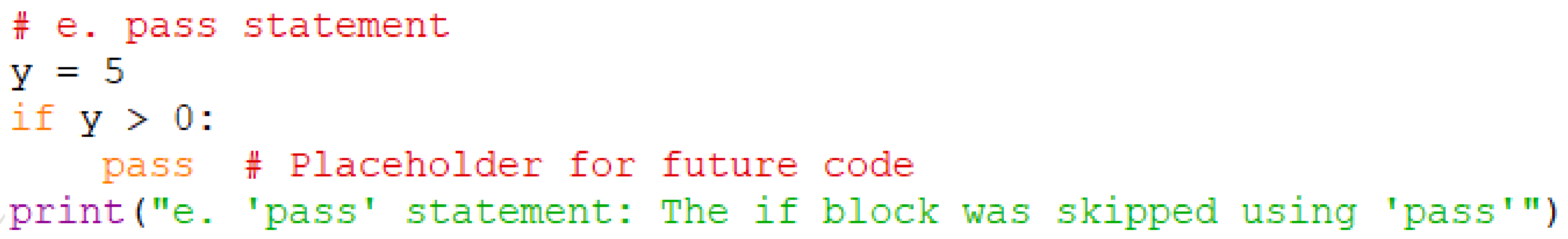
pass statement

SLO 3.7.2 A

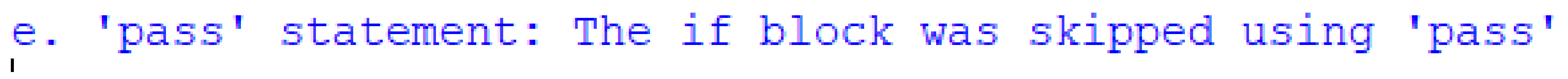
- The pass statement in Python is a null operation — it does nothing when executed.
- It's mainly used as a placeholder in code where a statement is syntactically required but you don't want any action to occur yet.
- Purpose of pass
  - To avoid syntax errors in empty code blocks.
  - To create placeholders for future code (like "to be implemented later").
  - Useful in functions, loops, classes, or conditionals where you haven't written logic yet.
- Syntex: `pass`

write a Python program using the pass statements

SLO 3.7.2 A



```python
# e. pass statement
y = 5
if y > 0:
    pass  # Placeholder for future code
print("e. 'pass' statement: The if block was skipped using 'pass'")
```



```
e. 'pass' statement: The if block was skipped using 'pass'
```

3.8 Repetition (Loop) in Python Programming

SLO	Students should be able to	Cognitive Level
3.8.1	describe loop and its types in Python programming;	U
3.8.2	describe the structure of **for** and **while** loop;	U
3.8.3	write a Python program using the following loops: a. for, b. while, c. nested for;	A
3.8.4	describe the **break**, **continue**, and **exit** statements in Python;	U
3.8.5	write a Python program using the following statements: a. break, b. continue, c. exit( ) function;	A

SLO 3.8.1 U

describe loop and its types in Python programming;

Prepared By: Naseer Ahmad (Nusrat Jahan Boys College) 0333-9790903

Loop and its types in Python Programming

SLO 3.8.1 U

- In Python, a loop is a control structure that allows you to execute a block of code repeatedly based on a condition or a sequence.
- Loops are essential for automating repetitive tasks and iterating over data structures like lists, tuples, or strings.
- Python provides two primary types of loops:
    - **for loop**
    - **while loop**
- Additionally, Python supports loop control statements to modify loop behavior.

SLO 3.8.1 U

Loop and its types in Python Programming

- **for loop**
  - A for loop in Python is a control structure used to iterate over a sequence (e.g., list, tuple, string, dictionary, set, or range) to execute a block of code for each element in the sequence. It is typically used when the number of iterations is known or when you need to process each item in an iterable.
- **while loop**
  - A while loop in Python is a control structure that repeatedly executes a block of code as long as a specified condition evaluates to True. It is used when the number of iterations is unknown and depends on a condition being met

SLO 3.8.2 U

describe the structure of for and while loop;

Structure of for loop

SLO 3.8.2 U

- A for loop in Python is a control structure used to iterate over a sequence or iterable object, such as a list, tuple, string, dictionary, set, or range() object.
- It allows you to execute a block of code repeatedly for each element in the sequence.
- The for loop is particularly useful when the number of iterations is known or when you need to process each item in an iterable.

Structure of for loop

SLO 3.8.2 U

- Syntex:

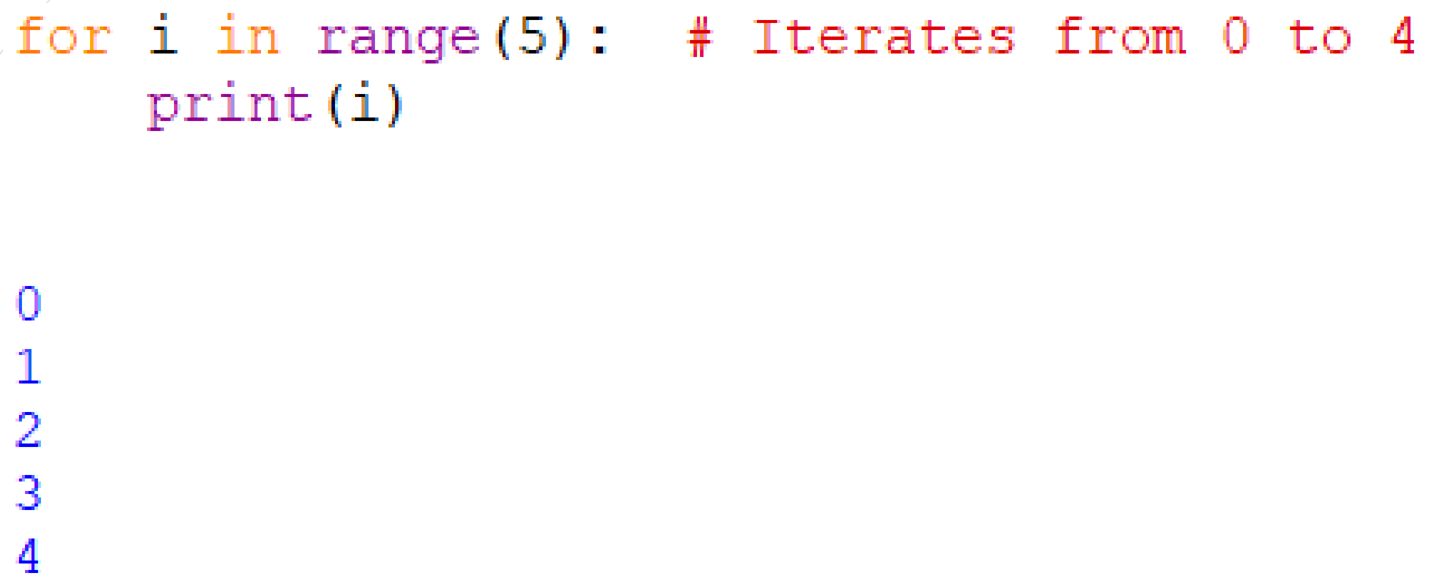
```
for variable in sequence:
    # Code block to execute
else:
    # code to run if the loop finishes without 'break'
```

- variable: A temporary variable that takes on the value of each element in the sequence during each iteration. You can name this variable anything meaningful (e.g., item, number, or char).
- sequence: An iterable object, such as a list, tuple, string, set, dictionary, or range() object, that provides the elements to iterate over.
- Code block: The indented code that executes for each iteration. Indentation (usually 4 spaces or 1 tab) defines the scope of the loop.

Structure of for loop

SLO 3.8.2 U

- Example:



```
for i in range(5):  # Iterates from 0 to 4
    print(i)
```

```
0
1
2
3
4
```

range()

SLO 3.8.2 U

- The range() function in Python is a built-in function used to generate a sequence of numbers, commonly employed in for loops to iterate a specific number of times or over a range of values.
- It is particularly useful for creating sequences of integers in a concise and memory-efficient way, as it generates numbers on demand rather than storing them as a list.
- The range() function returns a range object, which is an immutable sequence type that can be iterated over, typically in a for loop.

range()

SLO 3.8.2 U

- Syntax of range()

```
range(start, stop, step)
```

- start (optional): The starting number of the sequence (inclusive). Defaults to 0 if not specified.
- stop (required): The end number of the sequence (exclusive, meaning the sequence stops before this number).
- step (optional): The increment (or decrement) between numbers in the sequence. Defaults to 1 if not specified.

range()

SLO 3.8.2 U

Forms of range()

The range() function can be called in three ways, depending on the number of arguments provided:

- range(stop): Generates numbers from 0 to stop-1 with a step of 1.
- range(start, stop): Generates numbers from start to stop-1 with a step of 1.
- range(start, stop, step): Generates numbers from start to stop-1, incrementing by step.

For loop

SLO 3.8.2 U

```
for i in range(5):  # Generates 0, 1, 2, 3, 4
    print(i)

for i in range(2, 6):  # Generates 2, 3, 4, 5
    print(i)


for i in range(1, 10, 2):  # Generates 1, 3, 5, 7, 9
    print(i)


for i in range(10, 0, -2):  # Generates 10, 8, 6, 4, 2
    print(i)
```

SLO 3.8.2 U

Structure of while loop

- A while loop in Python is a control structure that repeatedly executes a block of code as long as a specified condition evaluates to True.
- It is used when the number of iterations is unknown and depends on a condition being met, making it ideal for situations where the loop's termination depends on dynamic or external factors.

Structure of while loop

SLO 3.8.2 U

- Syntax of a While Loop

```
while condition:
    # Code block to execute
else:
    # code to run if the loop finishes without 'break'
```

- condition: An expression that evaluates to True or False. The loop continues as long as the condition is True.
- Code block: The indented code (usually 4 spaces) that runs for each iteration while the condition remains True.

SLO 3.8.2 U

Structure of while loop

- How a While Loop Works
  - The condition is evaluated.
  - If the condition is True, the code block executes.
  - After each iteration, the condition is re-evaluated.
  - The loop stops when the condition becomes False or a break statement is encountered.
  - An optional else block can execute when the loop completes normally (i.e., without a break).
- Key Features
  - Condition-Driven: Continues until the condition is False.
  - Manual Control: Requires the programmer to update variables within the loop to ensure the condition eventually becomes False, avoiding infinite loops.
  - Flexible: Useful for scenarios like user input validation, waiting for events, or processing until a specific state is reached.
  - Control Statements: Supports break, continue, and else for additional control.

while loop

SLO 3.8.2 U

```
count = 1
while count <= 5:
    print(count)
    count += 1  # Increment to avoid infinite loop
```

```
1
2
3
4
5
```

SLO 3.8.3 A

write a Python program using the following loops:

a. for,
b. while,
c. nested for;

write a Python program using for loop

SLO 3.8.3 A

```python
# Program to calculate the factorial of a number using a for loop

# Get input from the user
num = int(input("Enter a non-negative integer to calculate its factorial: "))

# Check for negative numbers
if num < 0:
    print("Factorial is not defined for negative numbers!")
else:
    # Initialize factorial to 1
    factorial = 1

    # Calculate factorial using a for loop
    for i in range(1, num + 1):
        factorial *= i

    # Print the result
    print(f"The factorial of {num} is: {factorial}")
```

```
Enter a non-negative integer to calculate its factorial: 5
The factorial of 5 is: 120
```

write a Python program using while loop

SLO 3.8.3 A

```python
# Program to print the table of 2 using a while loop

num = 1  # start from 1

while num <= 10:  # loop until 10
    print(f"2 x {num} = {2 * num}")
    num += 1  # increment the counter
```

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

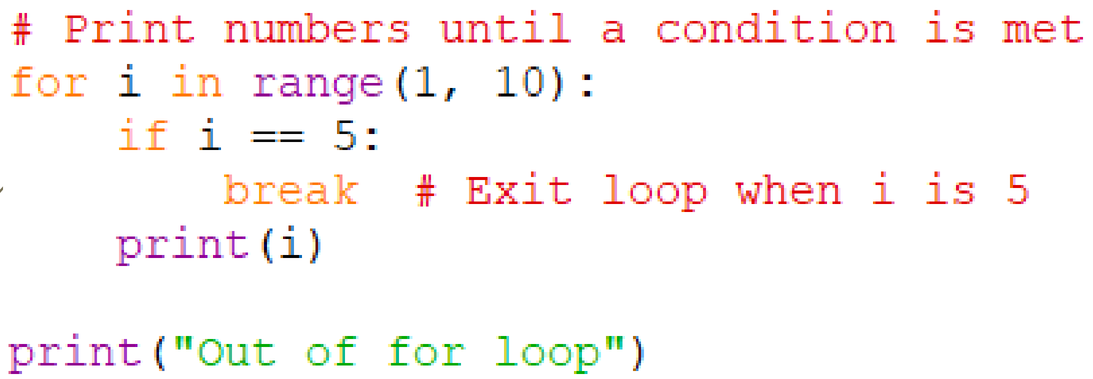
SLO 3.8.4 U

describe the break, continue, and exit statements in Python;

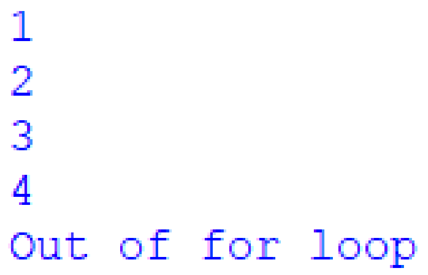
SLO 3.8.4 U

break statement in loop

- The break statement in Python is used to exit a loop prematurely when a certain condition is met, stopping the loop entirely.
- It can be used in both for and while loops.



```python
# Print numbers until a condition is met
for i in range(1, 10):
    if i == 5:
        break  # Exit loop when i is 5
    print(i)

print("Out of for loop")
```



```
1
2
3
4
Out of for loop
```

SLO 3.8.4 U

continue statement in loop

- The continue statement in Python is used to skip the rest of the code inside a loop for the current iteration and move to the next iteration.
- It works in both for and while loops, unlike break, which exits the loop entirely.

```python
# Print only odd numbers from 1 to 10
for i in range(1, 11):
    if i % 2 == 0:  # Check if number is even
        continue     # Skip even numbers
    print(i)
```

```
1
3
5
7
9
```

SLO 3.8.4 U

exit() function in python

- In Python, the exit() function is used to terminate the program immediately.
- It is part of the sys module and is typically used to exit a script or program when a certain condition is met.
- To use it, you need to import the sys module.
- There's also a built-in exit() function (and its alias quit()) available in the Python interactive shell, mainly for exiting the interpreter, but it's not recommended for use in scripts.

```
import sys

# Check for a specific number and exit
for i in range(1, 10):
    if i == 5:
        print(f"Reached {i}, exiting program.")
        sys.exit()   # Terminate the program
    print(i)
```

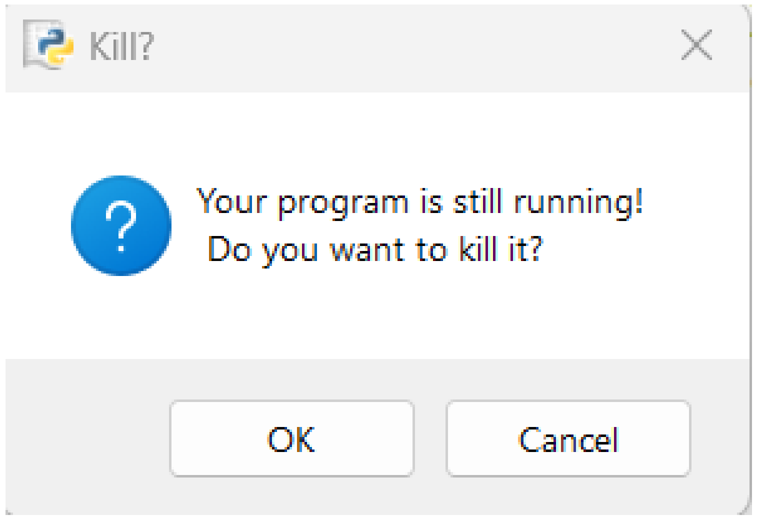
```
1
2
3
4
Reached 5, exiting program.
```

exit() function in python

SLO 3.8.4 U

```python
# Check for a specific number and exit
for i in range(1, 10):
    if i == 5:
        print(f"Reached {i}, exiting program.")
        exit()  # Terminate the program
    print(i)
```

```
1
2
3
4
Reached 5, exiting program.
```


Kill?

Your program is still running!
Do you want to kill it?

SLO 3.8.5 A

write a Python program using the following statements:

a. break,
b. continue,
c. exit() function;

SLO 3.8.5 A

write a program using break, continue, exit()

```
for i in range(1, 11):
    if i == 8:
        print("Reached 8, exiting program.")
        exit()  # Terminates the entire program
    if i % 2 == 0:
        print(f"{i} is even, continuing to next number.")
        continue  # Skips the rest of the loop body for this iteration
    if i == 5:
        print("Reached 5, breaking the loop.")
        break  # Exits the loop entirely
    print(f"Processing odd number: {i}")
print("This line will not be reached due to exit().")
```

```
Processing odd number: 1
2 is even, continuing to next number.
Processing odd number: 3
4 is even, continuing to next number.
Reached 5, breaking the loop.
This line will not be reached due to exit().
```

3.9 Debugging in Python

SLO	Students should be able to	Cognitive Level
3.9.1	describe the importance of debugging;	U
3.9.2	explain different types of bugs (errors) in Python code;	U
3.9.3	explain the process of finding bugs using an Integrated Development Environment (IDE);	U
3.9.4	explain breakpoints to pause execution;	U
3.9.5	apply the process of debugging using the print statement and assert keyword in Python;	A
3.9.6	apply the process of identifying and resolving bugs using the Python debugger (pdb).	A

SLO 3.9.1 U

describe the importance of debugging;

SLO 3.9.1 U

Importance of Debugging in Python

- Debugging in Python is critical for:
  - Ensuring correctness: Fixes errors (e.g., wrong break or continue usage) so the program works as intended.
  - Saving time: Catches issues early, preventing bigger problems.
  - Improving quality: Leads to cleaner, efficient code.
  - Enhancing learning: Deepens understanding of Python's flow (e.g., exit() behavior).
  - Preventing failures: Ensures reliability in production.
  - Aiding collaboration: Makes code consistent for teams.
  - Tools like print(), pdb, or IDE debuggers help identify issues in control flow.

SLO 3.9.2 U

explain different types of bugs (errors) in Python code;

Types of bugs (errors) in Python

SLO 3.9.2 U

- In Python programming, bugs (errors) are issues that prevent a program from running correctly or producing the expected output.
- These errors can be broadly categorized into three main types: syntax errors, runtime errors, and logical errors.

SLO 3.9.2 U

Syntax Error

- Definition: Errors caused by incorrect Python syntax, preventing the code from being parsed or executed.
- Why they occur: Violating Python's grammar rules, such as missing punctuation, incorrect indentation, or invalid keywords.
- Impact: The program fails to run, and Python raises a SyntaxError with a specific line number.
- Example (related to your code):

```
for i in range(1, 11)
    print(i)  # Missing colon (:) after range
```

Runtime Errors

SLO 3.9.2 U

- Definition: Errors that occur during program execution, causing the program to crash or terminate unexpectedly.
- Why they occur: Issues like dividing by zero, accessing undefined variables, or calling exit() inappropriately.
- Impact: Python raises exceptions (e.g., ZeroDivisionError, NameError), and the program stops unless handled with try-except.
- Example (related to your code):

```
for i in range(1, 11):
    if i == 8:
        exit(undefined_var)  # Undefined variable
    print(i)
```

Logical Errors

SLO 3.9.2 U

- Definition: Errors where the code runs without crashing but produces incorrect or unexpected results due to flawed logic.
- Why they occur: Incorrect algorithms, wrong conditions, or misuse of control structures like break or continue.
- Impact: The program completes but gives wrong output, often requiring debugging to identify.
- Example (related to your code):

```python
for i in range(1, 11):
    if i % 2 == 0:
        print(f"{i} is odd, continuing.")  # Wrong logic: even numbers labeled as odd
        continue
    print(f"Processing number: {i}")
```

SLO 3.9.2 U

Semantic Errors

- Definition: Errors where the code runs and produces output, but it doesn't align with the programmer's intended meaning or purpose.
- Why they occur: Misunderstanding how Python features work or using them in ways that don't match the intended functionality.
- Impact: The program works but doesn't achieve the desired goal, often harder to spot than logical errors.
- Example (related to your code):

```python
import sys
for i in range(1, 11):
    if i == 8:
        sys.exit()  # Intended to exit loop, but exits entire program
    print(i)
```

SLO 3.9.3 U

explain the process of finding bugs using an Integrated Development Environment (IDE);

Finding Bugs Using IDLE

SLO 3.9.3 U

1. Run Your Program
   - Open your Python file in IDLE (File → Open → Select your .py file).
   - Run it using:
     Run → Run Module or press F5.
   - If there's an error, IDLE will display a traceback message in the Python Shell.
2. Read the Error Message (Traceback)
   - When Python encounters an error, it shows:
     - The line number
     - The type of error
     - A short message explaining what went wrong

Finding Bugs Using IDLE

SLO 3.9.3 U

```
print(x)
```

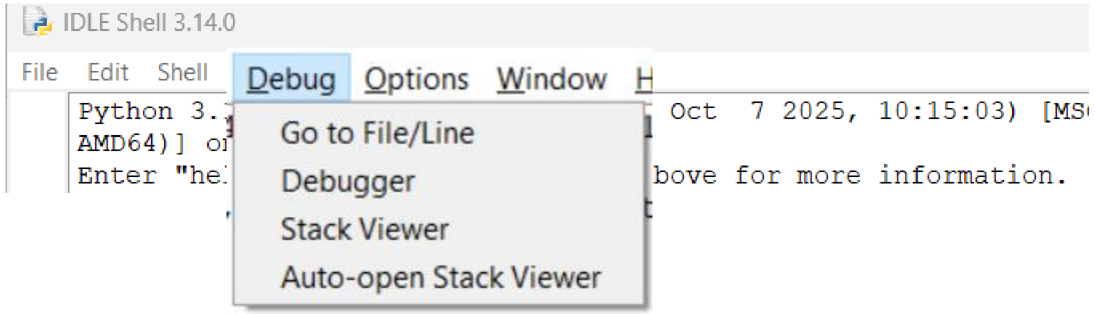
```
Traceback (most recent call last):
  File "C:/pycode/test.py", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

- Explanation:
  - File name: test.py
  - Line number: 1
  - Error: NameError
  - Reason: x is not defined

SLO 3.9.3 U

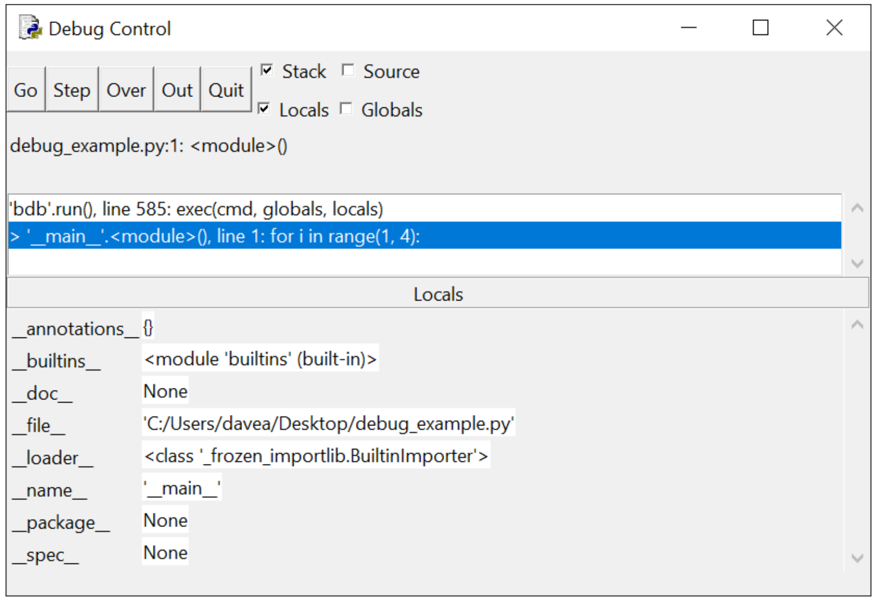
Use the IDLE Debugger

- IDLE includes a debugger that lets you run your program step by step and inspect variables.



```
Python 3.         Oct  7 2025, 10:15:03) [MS
AMD64)] on
Enter "he         bove for more information.
```

- To enable it:
  - Go to Debug → Debugger in the menu.
  - A new window opens showing options:
    - Step – execute one line at a time
    - Over – skip over function calls
    - Out – step out of a function
    - Go – continue execution
    - Quit – stop debugging



[✓ Stack] Source
[✓ Locals] Globals

```
debug_example.py:1: <module>()

'bdb'.run(), line 585: exec(cmd, globals, locals)
> '__main__'.<module>(), line 1: for i in range(1, 4):
```

__annotations__	{}
__builtins__	<module 'builtins' (built-in)>
__doc__	None
__file__	'C:/Users/davea/Desktop/debug_example.py'
__loader__	<class '_frozen_importlib.BuiltinImporter'>
__name__	'__main__'
__package__	None
__spec__	None

Use the IDLE Debugger

SLO 3.9.3 U

- How to use it:
  - Set breakpoints by clicking beside a line number (or pressing Ctrl + B).
  - Run your program with F5.
  - The debugger will pause at your breakpoints so you can check variable values and program flow.

SLO 3.9.4 U

explain breakpoints to pause execution;

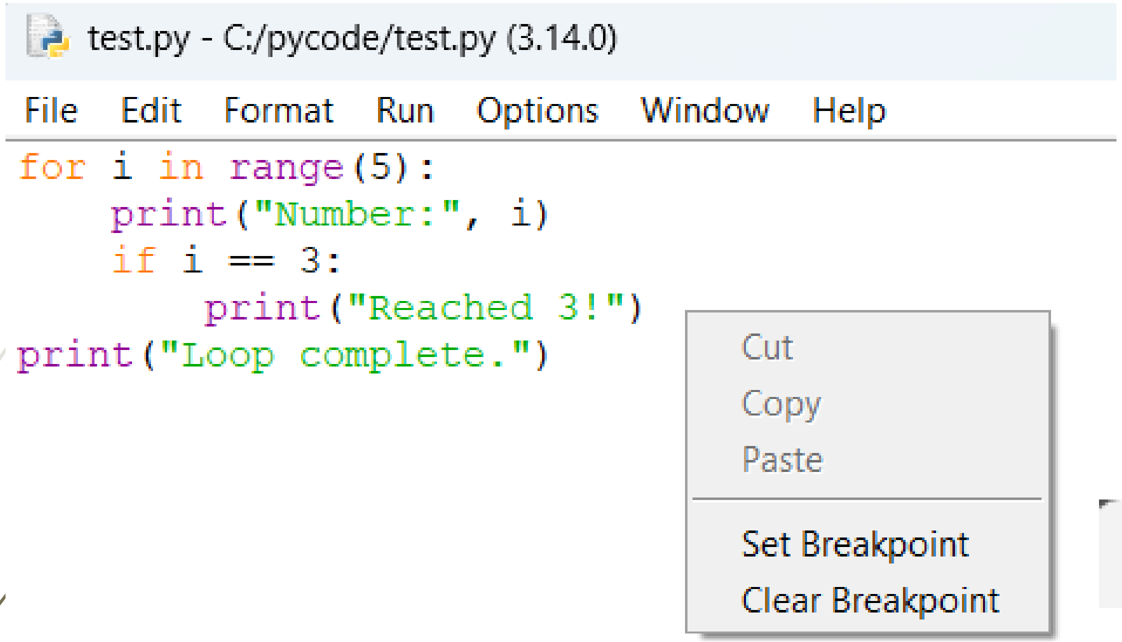
SLO 3.9.4 U

Breakpoints in python

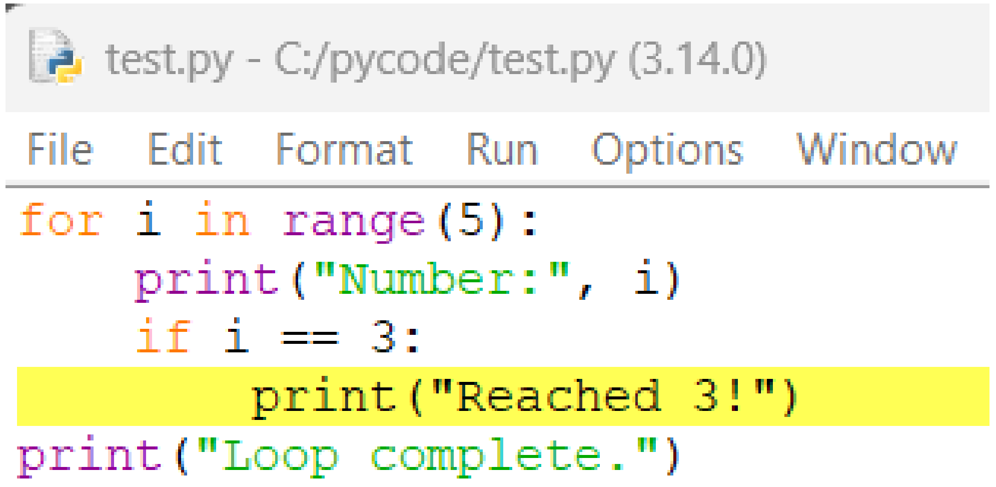
- Breakpoints are one of the most powerful features in Python IDLE for debugging.
- They let you pause your program at a specific line so you can inspect variables, test logic, and step through code one line at a time.
- Right click on any line to set/clear breakpoint

SLO 3.9.4 U

Breakpoints in python


test.py - C:/pycode/test.py (3.14.0)
File Edit Format Run Options Window Help
Cut
Copy
Paste
Set Breakpoint
Clear Breakpoint

```
for i in range(5):
    print("Number:", i)
    if i == 3:
        print("Reached 3!")
print("Loop complete.")
```


test.py - C:/pycode/test.py (3.14.0)
File Edit Format Run Options Window

```
for i in range(5):
    print("Number:", i)
    if i == 3:
        print("Reached 3!")
print("Loop complete.")
```

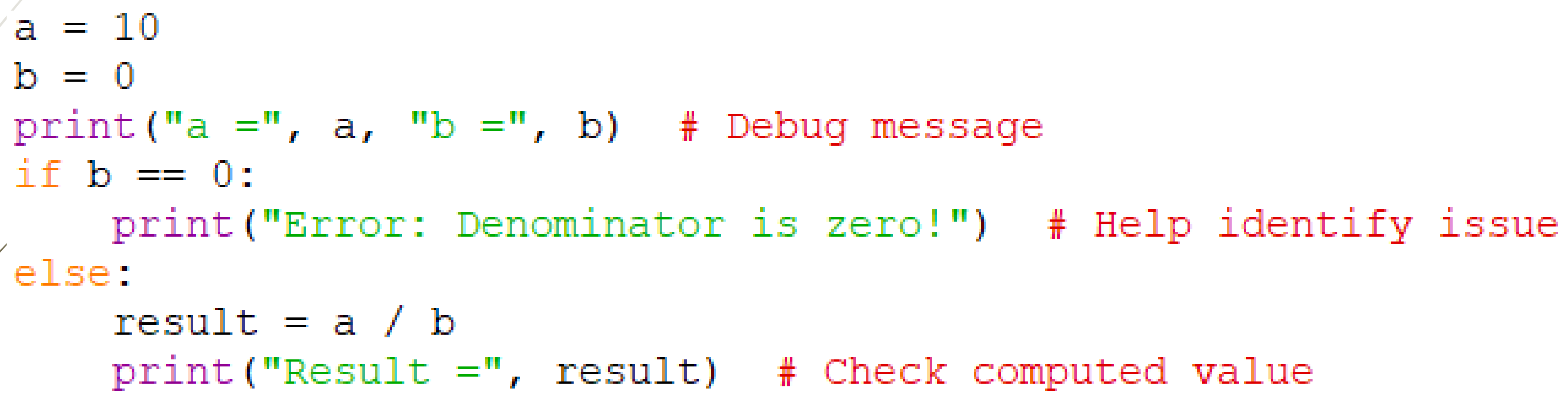
SLO 3.9.5 A

apply the process of debugging using the print statement and assert keyword in Python;

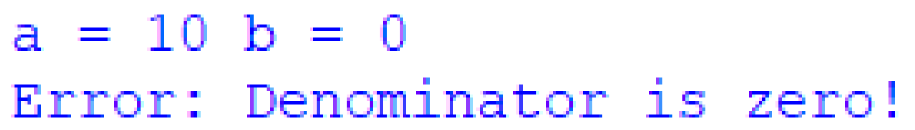
SLO 3.9.5 A

1. Debugging with print() Statements

- The simplest and most common way to find bugs is by using print() to display variable values and check the program's flow.



```python
a = 10
b = 0
print("a =", a, "b =", b)  # Debug message
if b == 0:
    print("Error: Denominator is zero!")  # Help identify issue
else:
    result = a / b
    print("Result =", result)  # Check computed value
```



```
a = 10 b = 0
Error: Denominator is zero!
```

SLO 3.9.5 A

2. Debugging with assert Keyword

- The assert statement is used to test conditions that must be true while your program runs.
- If the condition is False, it raises an AssertionError and optionally shows a message.
- Syntex:

```
assert condition, "Error message"
```

2. Debugging with assert Keyword

SLO 3.9.5 A

- The assert statement is used to test conditions that must be true while your program runs.
- If the condition is False, it raises an AssertionError and optionally shows a message.
- Syntex:

```
assert condition, "Error message"
```

SLO 3.9.5 A

2. Debugging with assert Keyword

```python
a = 10
b = 0
print("a =", a, "b =", b)   # Debug message
assert b != 0, "Denominator must not be zero"
result = a / b
print("Result =", result)   # Check computed value
```

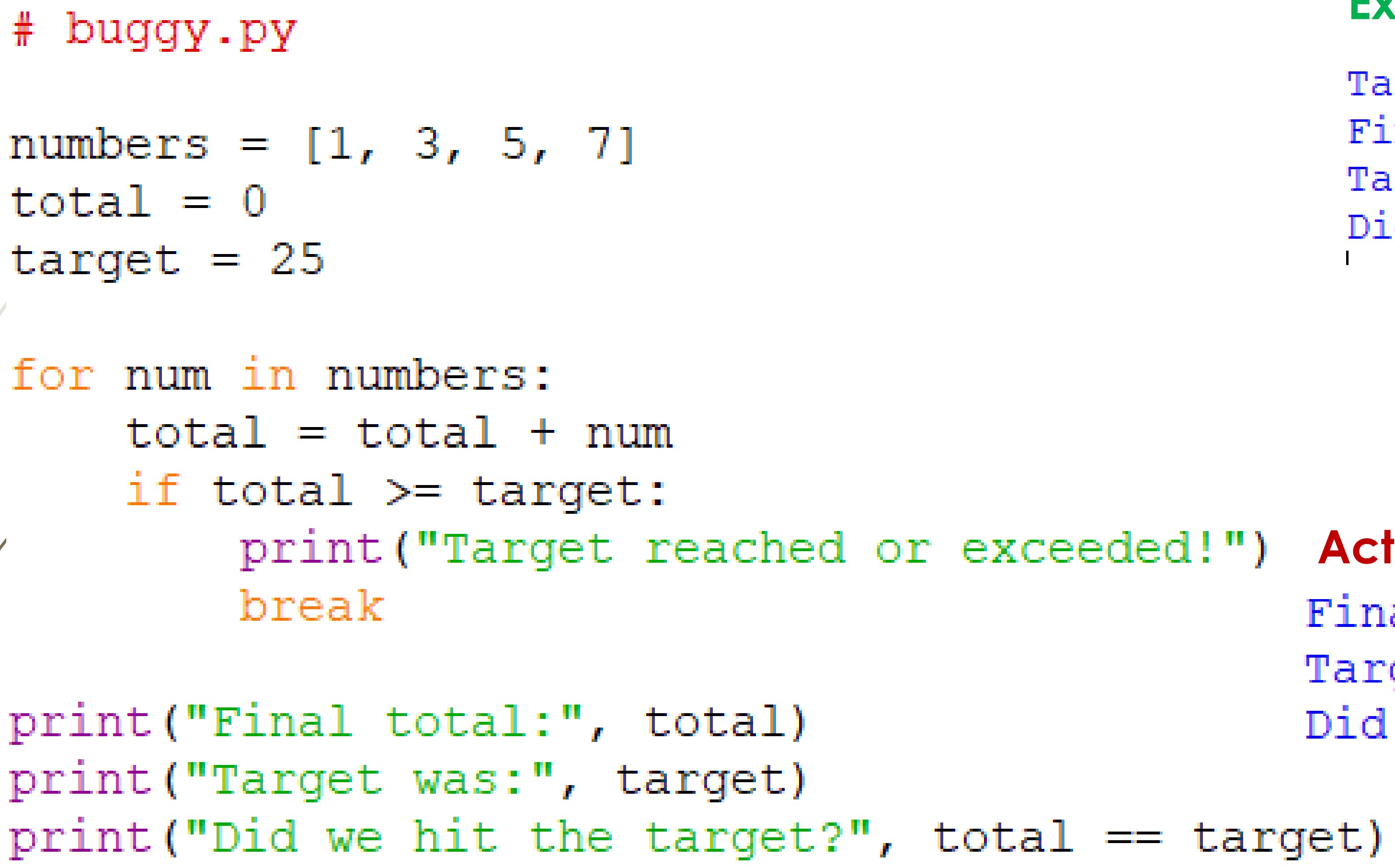
```
a = 10 b = 0
Traceback (most recent call last):
  File "C:/pycode/test.py", line 4, in <module>
    assert b != 0, "Denominator must not be zero"
AssertionError: Denominator must not be zero
```

SLO 3.9.6 A

apply the process of identifying and resolving bugs using the Python debugger (pdb).

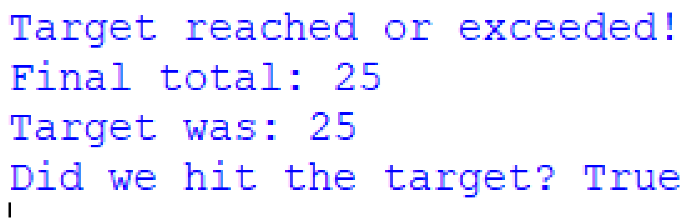
Example Script – With a Hidden Bug

SLO 3.9.6 A



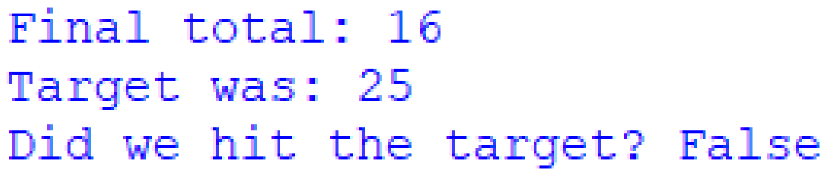
```python
# buggy.py

numbers = [1, 3, 5, 7]
total = 0
target = 25

for num in numbers:
    total = total + num
    if total >= target:
        print("Target reached or exceeded!")
        break

print("Final total:", total)
print("Target was:", target)
print("Did we hit the target?", total == target)
```

Expected:



```
Target reached or exceeded!
Final total: 25
Target was: 25
Did we hit the target? True
```

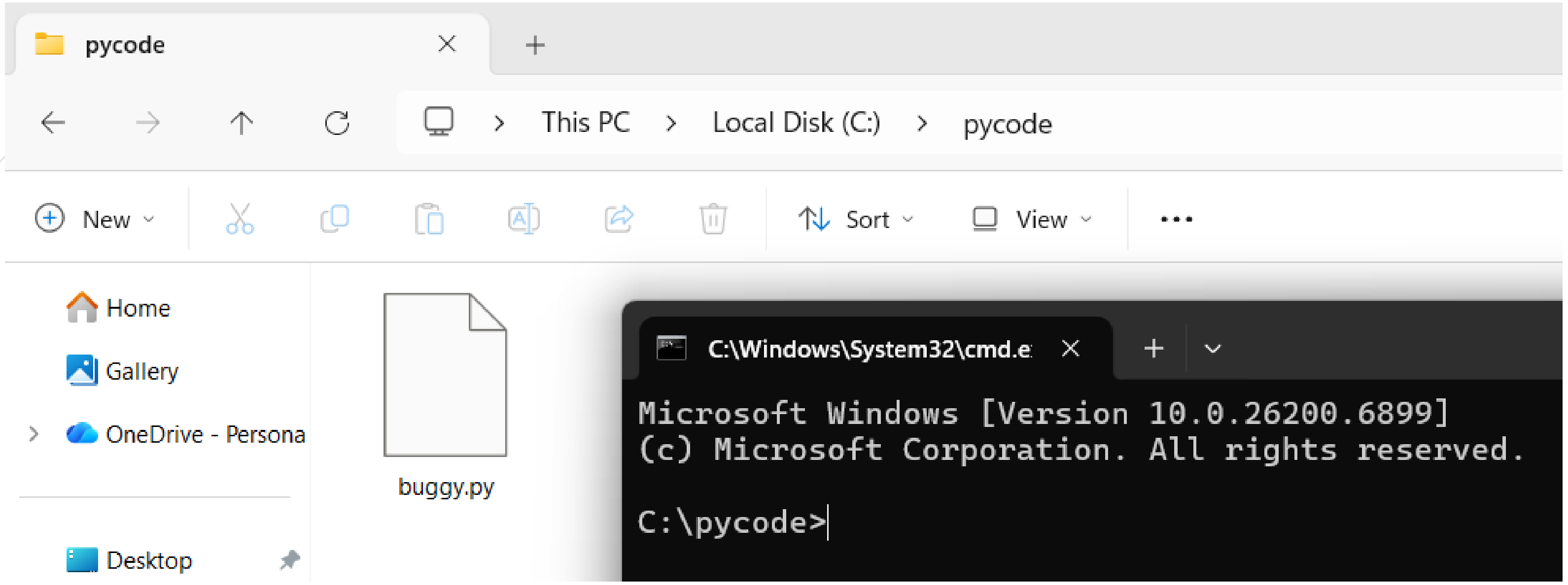
Actual:



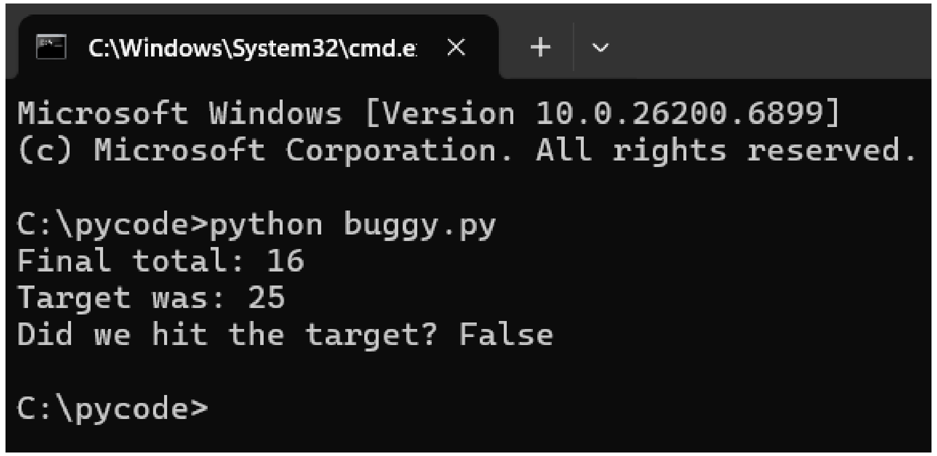
```
Final total: 16
Target was: 25
Did we hit the target? False
```

Run Script from Terminal

SLO 3.9.6 A



```
Microsoft Windows [Version 10.0.26200.6899]
(c) Microsoft Corporation. All rights reserved.

C:\pycode>
```



```
Microsoft Windows [Version 10.0.26200.6899]
(c) Microsoft Corporation. All rights reserved.

C:\pycode>python buggy.py
Final total: 16
Target was: 25
Did we hit the target? False

C:\pycode>
```

SLO 3.9.6 A

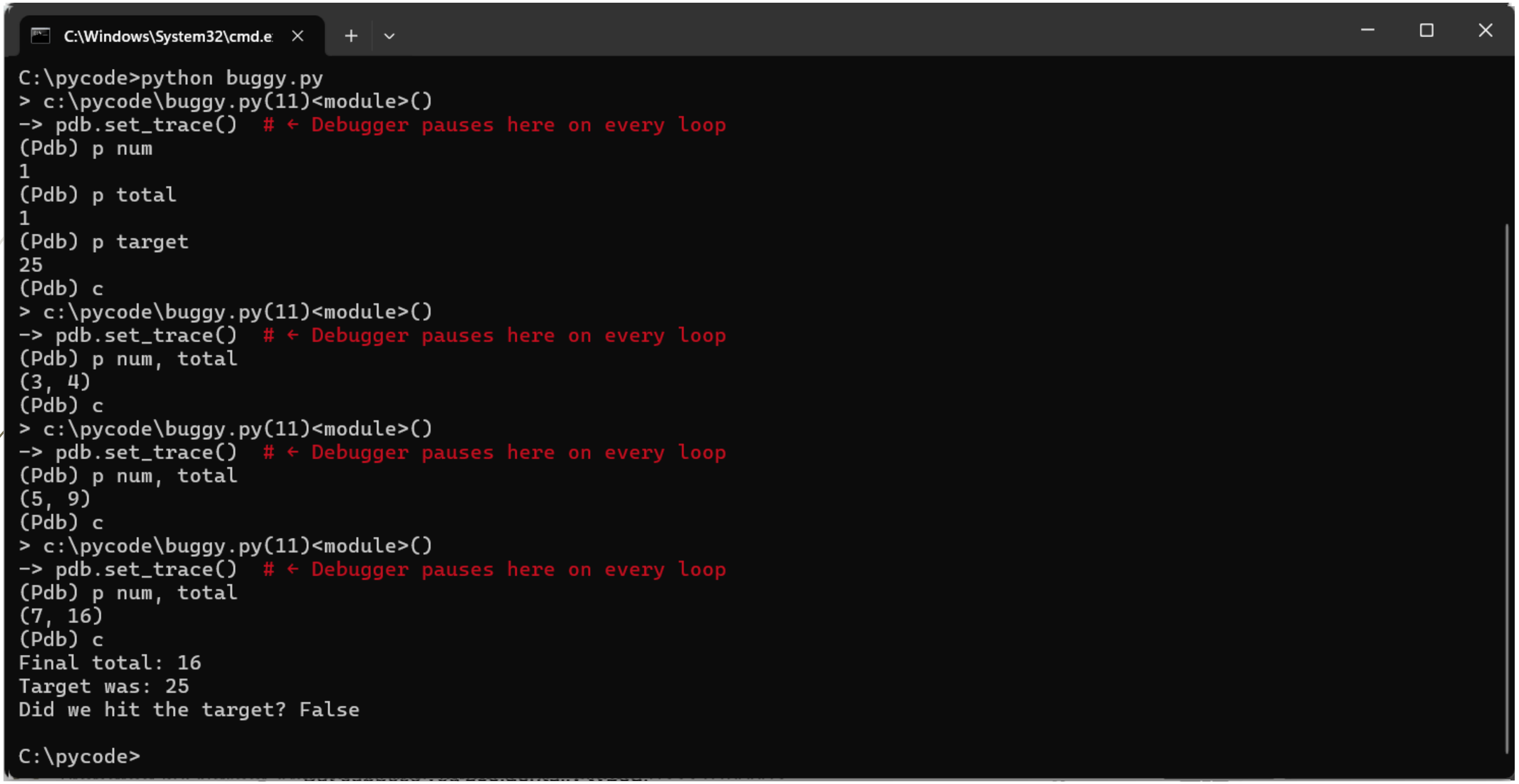
Debug

- pdb.set_trace() (Manual Breakpoint)
- Add this line where you suspect the bug (e.g., inside the loop):

```
# buggy.py

import pdb   # ← Add at top

numbers = [1, 3, 5, 7]
total = 0
target = 25

for num in numbers:
    total = total + num
    pdb.set_trace()   # ← Debugger pauses here on every loop
    if total >= target:
        print("Target reached or exceeded!")
        break

print("Final total:", total)
print("Target was:", target)
print("Did we hit the target?", total == target)
```

Debug

SLO 3.9.6 A



```
C:\pycode>python buggy.py
> c:\pycode\buggy.py(11)<module>()
-> pdb.set_trace()  # ← Debugger pauses here on every loop
(Pdb) p num
1
(Pdb) p total
1
(Pdb) p target
25
(Pdb) c
> c:\pycode\buggy.py(11)<module>()
-> pdb.set_trace()  # ← Debugger pauses here on every loop
(Pdb) p num, total
(3, 4)
(Pdb) c
> c:\pycode\buggy.py(11)<module>()
-> pdb.set_trace()  # ← Debugger pauses here on every loop
(Pdb) p num, total
(5, 9)
(Pdb) c
> c:\pycode\buggy.py(11)<module>()
-> pdb.set_trace()  # ← Debugger pauses here on every loop
(Pdb) p num, total
(7, 16)
(Pdb) c
Final total: 16
Target was: 25
Did we hit the target? False

C:\pycode>
```

Debug (Manual Breakpoint)

SLO 3.9.6 A

```python
# buggy.py

import pdb   # ← Add at top

numbers = [1, 3, 5, 7, 9]
total = 0
target = 25

for num in numbers:
    total = total + num
    #pdb.set_trace()   # ← Debugger pauses here on every loop
    if total >= target:
        print("Target reached or exceeded!")
        break

print("Final total:", total)
print("Target was:", target)
print("Did we hit the target?", total == target)
```

```
Target reached or exceeded!
Final total: 25
Target was: 25
Did we hit the target? True
```

SLO 3.9.6 A

Full List of pdb Commands

Command	Shortcut	Description
`list`	`l`	Show current code
`list .`	`l .`	Show current line
`list 5,10`		Show lines 5 to 10
`next`	`n`	Execute **next line** (step over)
`step`	`s`	Step **into** function calls
`continue`	`c` or `cont`	Continue until breakpoint or end
`break`	`b`	Set breakpoint: `b 10` → line 10
`break function_name`		Break at function
`tbreak`	`t`	Temporary breakpoint (removed after hit)
`clear`	`cl`	Clear breakpoint: `cl 10`
`break`	`b`	List all breakpoints
`print`	`p`	Print variable: `p total`

SLO 3.9.6 A

Full List of pdb Commands

Command	Shortcut	Description
`print`	`p`	Print variable: `p total`
`pp`		Pretty-print (for lists/dicts)
`whatis`		Show type: `whatis total`
`display`		Auto-display expression every step
`undisplay`		Stop auto-display
`jump`	`j`	Set next line to execute (dangerous!)
`return`	`r`	Continue until current function returns
`until`	`unt`	Continue until line > current or loop exit
`where`	`w`	Show full stack trace
`up`	`u`	Move up in call stack
`down`	`d`	Move down in call stack

SLO 3.9.6 A

Full List of pdb Commands

Command	Shortcut	Description
`args`	`a`	Show function arguments (if in function)
`interact`		Start interactive Python shell at current point
`quit`	`q`	Exit debugger
`help`	`h`	Show help
`help next`		Help for specific command

ANY Questions?

Thank You!