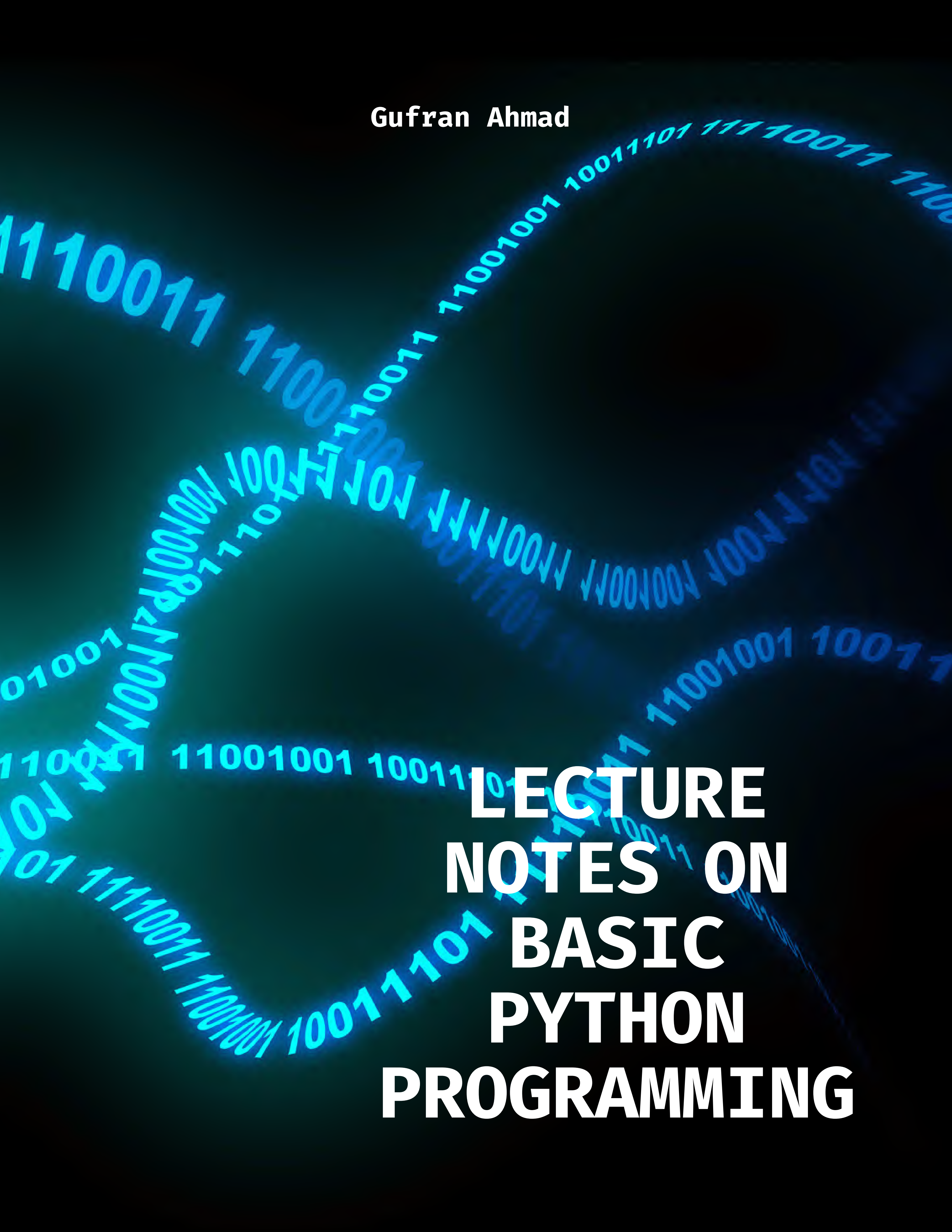

Gufran Ahmad

LECTURE NOTES ON BASIC PYTHON PROGRAMMING

Lecture 1: What is Programming? Why Python?

Key Topics:

- Definition and purpose of programming.
- Overview of Python as a programming language.
- Basics of Python syntax and operations.

Summary:

1. What is Programming?
   - Programming is instructing a computer to perform specific tasks using a language it understands.
   - Programs convert human-readable code into machine instructions using interpreting or compiling.
2. Why Python?
   - Python is user-friendly and versatile, designed for simplicity and readability.
   - It acts like an interpreted language, suitable for beginners yet powerful enough for advanced applications.

Key Code Examples:

1. "Hello, World!" Program:
   ```
   print("Hello, World!")
   ```
   Explanation:
   - A simple program to print "Hello, World!" on the screen.
   - Demonstrates how code is executed by converting to machine instructions.
2. Greeting Program with Comments:
   ```
   # Greeting
   print("Howdy!")  # Invitation
   print("Shall we play a game?")
   ```
   Explanation:
   - Comments (#) are ignored by the interpreter and are used to make the code understandable for humans.

Python Characteristics:

- Python supports interpreted and scripting capabilities.
- Offers high-level commands for automation.
- IDLE (Python's interactive shell) allows for immediate feedback during coding.

Lecture 2: Variables: Operations and Input/Output

Key Topics:

- Variables and their role in programming.
- Memory structure and variable types.
- Basic input/output operations.
- Performing operations on variables.

Summary:

1. Variables and Memory:
   - Variables are "boxes" in memory used to store data.
   - A variable is defined using an assignment operator =.
     Example:

     ```
     x = 3
     ```

     - Here, x is the variable name, and 3 is the value stored in x.
2. Data Types:
   - Common types include:
     - Integers (int): Whole numbers.
     - Floating-point numbers (float): Numbers with decimals.
     - Strings (str): Text enclosed in quotes (e.g., "Hello" or 'World').
   - Python determines the type based on the assigned value.
3. Basic Operations with Variables:
   - Arithmetic operations: +, -, *, /.
   - String concatenation: Adding strings with +.
   - Example:

     ```
     a = 5
     b = 3
     c = a + b  # Adds integers
     name = "John"
     greeting = "Hello, " + name  # Concatenates strings
     ```
4. Input and Output Operations:
   - Input is collected using input() and stored as a string.
     Example:

     ```
     user_input = input("Enter a value: ")
     print("You entered:", user_input)
     ```
   - Convert input to specific types using int() or float():

     ```
     a = int(input("Enter an integer: "))
     ```

```
b = float(input("Enter a floating-point number: "))
print("Sum:", a + b)
```

5. Common Pitfalls:
    - Input data is always a string unless explicitly converted.
    - Adding a string and a number results in an error:

    ```
    # Error example
    num = input("Enter a number: ")
    result = num + 5  # TypeError
    ```

Key Code Example: Adding Two Numbers

```
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
print("The sum is:", a + b)
```

Memory Visualization:

- When a = 5 and b = 3:

  ```
  Memory:
  a: 5
  b: 3
  ```

Lecture 3: Conditionals and Boolean Expressions

Key Topics:

- Using conditional statements (if, elif, else).
- Boolean expressions and comparisons.
- Nesting and logical operators.

Summary:

1. Conditional Statements:
   - Allow programs to make decisions based on conditions.
   - Syntax:

     ```
     if condition:
         # Code to execute if condition is True
     elif another_condition:
         # Code to execute if the previous condition is False
     else:
         # Code to execute if all conditions are False
     ```
   - Example:

     ```
     temp = 70
     if temp < 60:
         print("Turn on the heater.")
     elif temp > 80:
         print("Turn on the air conditioner.")
     else:
         print("Temperature is fine.")
     ```
2. Boolean Expressions:
   - Conditions evaluate to True or False.
   - Common comparison operators:
     - ==: Equal to
     - !=: Not equal to
     - <, <=: Less than, less than or equal
     - >, >=: Greater than, greater than or equal
   - Example:

     ```
     x = 10
     print(x > 5)   # True
     print(x == 10) # True
     print(x != 8)  # True
     ```
3. Logical Operators:
   - Combine multiple conditions:
     - and: All conditions must be True.

        - or: At least one condition must be True.
        - not: Negates a condition.
    - Example:

      ```
      age = 20
      if age > 18 and age < 65:
          print("Adult")
      ```
4. Nesting Conditionals:
    - if statements can be nested for more complex decisions.
    - Example:

      ```
      temp = 50
      if temp < 60:
          if temp < 50:
              print("It’s freezing!")
          else:
              print("It’s cold.")
      ```
5. Using elif:
    - Simplifies multiple if-else combinations:

      ```
      score = 85
      if score >= 90:
          print("Grade: A")
      elif score >= 80:
          print("Grade: B")
      elif score >= 70:
          print("Grade: C")
      else:
          print("Fail")
      ```

Key Code Example: Leap Year Checker

```
year = int(input("Enter a year: "))
if year % 400 == 0 or (year % 4 == 0 and year % 100 != 0):
    print("Leap year")
else:
    print("Not a leap year")
```

Exercises:

- Predict the output:

  ```
  a = 5
  b = 10
  if a > b:
      print("a is greater")
  else:
      print("b is greater")
  ```

Lecture 4: Basic Program Development and Testing

Key Topics:

- Steps for developing a program.
- Importance of iterative development and testing.
- Example program: Savings calculator.

Summary:

1. Program Development Steps:
   - Plan Ahead: Break down the program into smaller tasks before coding.
     - Example: For a savings goal program:
       1. Get user input.
       2. Perform calculations.
       3. Present results.
   - Write Comments: Use comments to outline the code structure before implementing logic.
   - Test Regularly: Test each logical section of code as it’s written to identify and fix errors early.
2. Iterative Development:
   - Develop programs step-by-step, ensuring that each stage works before proceeding.
   - Example:
     - Start by implementing user input.
     - Test if input is correctly stored.
     - Add calculations, then test.
     - Finally, format and display the results.
3. Error Handling:
   - Ensure the program handles invalid input gracefully:
     - Prevent division by zero.
     - Validate numerical inputs (e.g., positive numbers).

Key Code Example: Savings Calculator

1. Initial Version:

   ```
   balance = float(input("Enter the amount to save: "))
   payment = float(input("Enter the payment amount: "))
   num_payments = balance / payment
   print("You must make", num_payments, "payments.")
   ```
   Issue: Fails if payment is 0.
2. Improved Version with Error Handling:

   ```
   balance = float(input("Enter the amount to save: "))
   ```

```python
    if balance < 0:
        print("You already have enough saved!")
        balance = 0

    payment = float(input("Enter the payment amount: "))
    while payment <= 0:
        payment = float(input("Payment must be positive. Try again: "))

    num_payments = balance / payment
    print("You must make", num_payments, "payments.")
```

3. Advanced Version: Adding Rounded Results:

```python
import math

balance = float(input("Enter the amount to save: "))
if balance < 0:
    print("You already have enough saved!")
    balance = 0

payment = float(input("Enter the payment amount: "))
while payment <= 0:
    payment = float(input("Payment must be positive. Try again: "))

num_payments = math.ceil(balance / payment)
print("You must make", num_payments, "payments.")
```

Testing Process:

- Test with realistic and edge-case values (e.g., negative balance, zero payment, large values).
- Use print statements to verify intermediate values during development.

Iterative Improvement:

- Example Enhancements:
    1. Add input for current savings.
    2. Allow users to specify weekly, monthly, or yearly payments.

Lecture 5: Loops and Iterations

Key Topics:

- Types of loops: while and for.
- Repetition structures in programming.
- Avoiding infinite loops.
- Common use cases for loops.

Summary:

1. While Loops:
   - Used when the number of repetitions is unknown but depends on a condition.
   - Syntax:

     ```
     while condition:
         # Code to execute repeatedly
     ```
   - Example: Prompt for positive input:

     ```
     value = -1
     while value <= 0:
         value = int(input("Enter a positive number: "))
     ```
2. For Loops:
   - Used when the number of repetitions is known or defined by a sequence.
   - Syntax:

     ```
     for variable in range(start, stop, step):
         # Code to execute
     ```
   - Example: Print numbers from 1 to 5:

     ```
     for i in range(1, 6):
         print(i)
     ```
3. Infinite Loops:
   - A loop that never ends because the condition never becomes false.
   - Example of a common mistake:

     ```
     while True:
         print("This is an infinite loop!")
     ```
4. Combining Loops with Conditions:
   - Loops can include conditionals to control flow.
   - Example: Guessing game:

```python
secret = 7
guess = 0
while guess != secret:
    guess = int(input("Guess the number: "))
    if guess < secret:
        print("Too low!")
    elif guess > secret:
        print("Too high!")
print("Correct!")
```

5. Iterating Over Collections:
   - Loops can iterate over strings, lists, and other collections.
   - Example: Loop through a list:

```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Key Code Examples:

1. Average Age Calculator:

```python
num_people = int(input("How many people are there? "))
total_age = 0
for i in range(num_people):
    age = int(input(f"Enter age of person {i+1}: "))
    total_age += age
print("Average age:", total_age / num_people)
```

2. Countdown Using for Loop:

```python
for i in range(5, 0, -1):
    print(i)
print("Blast off!")
```

Differences Between while and for Loops:

Feature	while Loop	for Loop
Use Case	Unknown repetitions, conditional	Known repetitions or sequences
Syntax Simplicity	Requires manual control of variables	Built-in control over iteration

---

Lecture 6: Files and Strings

Key Topics:

- Handling files in Python.
- Reading from and writing to files.
- Strings and their operations.

Summary:

1. File Operations:
   - Files must be opened before use and closed after completing operations.
   - Modes for opening files:
     - "r": Read mode (default).
     - "w": Write mode (overwrites existing content).
     - "a": Append mode (adds to existing content).
   - Example:

     ```
     file = open("example.txt", "r")  # Open for reading
     content = file.read()           # Read the entire file
     print(content)
     file.close()                    # Close the file
     ```
2. Writing to Files:
   - Use "w" to write or "a" to append.
   - Example:

     ```
     with open("output.txt", "w") as file:
         file.write("Hello, World!")
     ```
     Note: Using with open ensures the file is automatically closed.
3. Reading from Files:
   - read(): Reads the entire file as a single string.
   - readline(): Reads one line at a time.
   - readlines(): Reads all lines into a list.
   - Example:

     ```
     with open("example.txt", "r") as file:
         for line in file:
             print(line.strip())  # Removes trailing newline
     characters
     ```
4. String Operations:
   - Common methods:
     - .strip(): Removes leading and trailing whitespace.

        - .split(): Splits a string into a list based on a delimiter.
        - .join(): Joins a list of strings into a single string.
    - Example:

      ```
      text = "  Hello, World!  "
      clean_text = text.strip()
      print(clean_text)  # "Hello, World!"
      ```
5. Combining Files and Strings:
    - Example: Word frequency counter.

      ```
      with open("document.txt", "r") as file:
          text = file.read()
          words = text.split()
          print("Number of words:", len(words))
      ```

Key Code Examples:

1. File Copy Program:

   ```
   with open("source.txt", "r") as source, open("destination.txt", "w") as destination:
       for line in source:
           destination.write(line)
   ```
2. Count Specific Words in a File:

   ```
   target_word = "Python"
   count = 0
   with open("document.txt", "r") as file:
       for line in file:
           words = line.split()
           count += words.count(target_word)
   print(f"The word '{target_word}' appears {count} times.")
   ```

Best Practices:

- Always close files or use with open for better resource management.
- Handle potential file errors with try and except:

  ```
  try:
      with open("missing_file.txt", "r") as file:
          content = file.read()
  except FileNotFoundError:
      print("File not found!")
  ```

Lecture 7: Operations with Lists

Key Topics:

- Basics of lists in Python.
- Common list operations (creation, indexing, slicing, and iteration).
- Modifying lists (adding, removing, updating elements).
- Useful built-in functions for lists.

Summary:

1. What is a List?
   - A list is an ordered, mutable collection of items.
   - Can store mixed data types, though typically stores similar data.
   - Created using square brackets:

     ```
     numbers = [1, 2, 3, 4]
     fruits = ["apple", "banana", "cherry"]
     mixed = [1, "apple", 3.5]
     ```
2. Accessing List Elements:
   - Use indexing (starting from 0):

     ```
     print(fruits[0])  # "apple"
     print(fruits[-1])  # "cherry"
     ```
   - Slicing: Extract portions of a list:

     ```
     print(numbers[1:3])  # [2, 3]
     print(numbers[:2])  # [1, 2]
     ```
3. Modifying Lists:
   - Adding items:
     - .append(): Adds an item at the end.

       ```
       fruits.append("date")
       ```
     - .insert(): Adds an item at a specific position.

       ```
       fruits.insert(1, "blueberry")
       ```
   - Removing items:
     - .remove(): Removes the first occurrence of a value.

       ```
       fruits.remove("banana")
       ```
     - .pop(): Removes an item by index (default: last item).

```
fruits.pop(0)  # Removes "apple"
```

- Updating items:

```
numbers[1] = 99  # Changes the second element to 99
```

4. Iterating Over Lists:
   - Use a for loop:

```
for fruit in fruits:
    print(fruit)
```

   - Use enumerate() for index and value:

```
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

5. Useful Built-in Functions:
   - len(): Returns the number of elements.
   - sorted(): Returns a sorted copy of the list.
   - sum(): Calculates the sum of numerical elements.
   - Example:

```
daily_high_temps = [83, 80, 73, 75, 79, 83, 86]
print(len(daily_high_temps))  # 7
print(sorted(daily_high_temps))  # [73, 75, 79, 80, 83, 83,
86]
print(sum(daily_high_temps))  # 559
```

6. Nested Lists:
   - Lists can contain other lists:

```
matrix = [[1, 2], [3, 4], [5, 6]]
print(matrix[1][0])  # 3
```

Key Code Examples:

1. Finding the Average of a List:

```
numbers = [10, 20, 30, 40]
avg = sum(numbers) / len(numbers)
print("Average:", avg)
```

2. Removing Duplicates from a List:

```
numbers = [1, 2, 2, 3, 3, 3, 4]
unique_numbers = list(set(numbers))
print(unique_numbers)  # [1, 2, 3, 4]
```

3. Flattening a Nested List:

```python
nested_list = [[1, 2], [3, 4], [5, 6]]
flat_list = [item for sublist in nested_list for item in sublist]
print(flat_list)  # [1, 2, 3, 4, 5, 6]
```

Lecture 8: Top-Down Design of a Data Analysis Program

Key Topics:

- Understanding top-down design methodology.
- Breaking down complex problems into manageable steps.
- Example: Designing a data analysis program.

Summary:

1. What is Top-Down Design?
   - A method of designing programs by breaking a larger problem into smaller, independent sub-problems (modules).
   - Each module focuses on a specific functionality, which can be implemented and tested separately.
2. Benefits of Top-Down Design:
   - Simplifies complex problems by dividing them into smaller tasks.
   - Encourages reusable, modular code.
   - Improves maintainability and debugging.
3. Steps in Top-Down Design:
   - Step 1: Understand the problem.
   - Step 2: Break the problem into logical steps or modules.
   - Step 3: Design and implement each module one by one.
   - Step 4: Integrate and test the complete solution.
4. Example Problem: Data Analysis Program
   - Goal: Analyze a dataset of numerical values to calculate statistics like mean, median, and variance.

   Top-Down Approach:
   - Module 1: Read the data.
   - Module 2: Perform calculations.
   - Module 3: Display the results.

Implementation Example:

1. Main Function:
   - Provides the overall structure and calls individual modules.

   ```
   def main():
       data = read_data()
       stats = calculate_statistics(data)
       display_results(stats)
   ```
2. Reading Data:

   ```
   def read_data():
       # Simulate reading data (replace with file input or user input
   as needed)
   ```

```python
        data = [10, 20, 30, 40, 50]
        return data
```

3. Calculating Statistics:

```python
   def calculate_statistics(data):
       mean = sum(data) / len(data)
       median = sorted(data)[len(data) // 2]
       variance = sum((x - mean) ** 2 for x in data) / len(data)
       return {"mean": mean, "median": median, "variance": variance}
```

4. Displaying Results:

```python
   def display_results(stats):
       print("Data Analysis Results:")
       print(f"Mean: {stats['mean']}")
       print(f"Median: {stats['median']}")
       print(f"Variance: {stats['variance']}")
```

5. Program Execution:

```python
   if __name__ == "__main__":
       main()
```

Output Example: When main() is executed:

```
Data Analysis Results:
Mean: 30.0
Median: 30
Variance: 200.0
```

Additional Improvements:

- Allow users to provide input data or read from a file.
- Add error handling for empty datasets.
- Include additional statistical measures (e.g., mode, standard deviation).

Lecture 9: Functions and Abstraction

Key Topics:

- Introduction to functions.
- Advantages of using functions.
- Function definition, arguments, and return values.
- Abstraction and code reuse.

Summary:

1. What is a Function?
   - A block of reusable code designed to perform a specific task.
   - Can take input (parameters), process it, and return an output.
   - Syntax:

     ```
     def function_name(parameters):
         # Function body
         return value
     ```
2. Benefits of Functions:
   - Reusability: Write once, use multiple times.
   - Abstraction: Simplifies complex operations by hiding details.
   - Maintainability: Easier to debug and update modular code.
3. Defining and Calling Functions:
   - Example:

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

     print(greet("Alice"))  # Output: Hello, Alice!
     ```
4. Arguments and Parameters:
   - Functions can take multiple arguments:

     ```
     def add(a, b):
         return a + b
     print(add(3, 5))  # Output: 8
     ```
   - Default arguments:

     ```
     def greet(name="Guest"):
         return f"Hello, {name}!"
     print(greet())  # Output: Hello, Guest!
     ```
5. Return Values:
   - Use return to send a value back to the caller.
   - Example:

```python
def square(num):
    return num ** 2
result = square(4)
print(result)  # Output: 16
```

6. Scope and Lifetime:
    - Variables inside functions are local to that function.
    - Global variables can be accessed inside functions using the global keyword (though discouraged).

Key Code Examples:

1. Function for Factorial Calculation:

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

2. Using Functions for Modular Code:
    - Program to calculate the area of a rectangle:

```python
def get_dimensions():
    length = float(input("Enter length: "))
    width = float(input("Enter width: "))
    return length, width

def calculate_area(length, width):
    return length * width

def display_area(area):
    print(f"The area is: {area}")

length, width = get_dimensions()
area = calculate_area(length, width)
display_area(area)
```

3. Abstraction with Nested Functions:
    - Example: Convert temperature between Celsius and Fahrenheit:

```python
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

def fahrenheit_to_celsius(fahrenheit):
    return (fahrenheit - 32) * 5/9
```

```python
def main():
    temp_c = 25
    temp_f = celsius_to_fahrenheit(temp_c)
    print(f"{temp_c}°C is {temp_f}°F")

if __name__ == "__main__":
    main()
```

Best Practices:

- Keep functions small and focused on a single task.
- Use descriptive names for functions and parameters.
- Avoid using global variables inside functions.

Lecture 10: Parameter Passing, Scope, and Mutable Data

Key Topics:

- Parameter passing in functions (pass-by-value vs. pass-by-reference).
- Scope and lifetime of variables.
- Mutable vs immutable data types.

Summary:

1. Parameter Passing:
    - In Python, parameters are passed by reference for mutable objects (e.g., lists, dictionaries) and by value for immutable objects (e.g., integers, strings).
    - Pass-by-Value: Changes to the parameter inside the function do not affect the original argument.
    - Pass-by-Reference: Changes to the parameter inside the function will affect the original argument.

    Example (Pass-by-Value with Immutable Types):

    ```
    def modify_number(num):
        num = 10  # Changes local copy, not the original

    n = 5
    modify_number(n)
    print(n)  # Output: 5 (no change)
    ```

    Example (Pass-by-Reference with Mutable Types):

    ```
    def modify_list(lst):
        lst.append(10)  # Modifies the original list

    my_list = [1, 2, 3]
    modify_list(my_list)
    print(my_list)  # Output: [1, 2, 3, 10] (list modified)
    ```

2. Scope of Variables:
    - Local Scope: Variables defined inside a function are local to that function.

      ```
      def foo():
          x = 10  # Local variable
          print(x)

      foo()  # Output: 10
      # print(x)  # This would cause an error (x is not defined
      outside foo)
      ```

- Global Scope: Variables defined outside all functions are global and can be accessed within functions, unless shadowed by a local variable.

  ```
  x = 5  # Global variable

  def foo():
      print(x)  # Accesses the global x

  foo()  # Output: 5
  ```
- Global Keyword: To modify a global variable inside a function, use the global keyword.

  ```
  x = 5

  def foo():
      global x
      x = 10  # Modify the global variable

  foo()
  print(x)  # Output: 10
  ```

3. Mutable vs Immutable Data Types:
   - Immutable Types: Once created, their values cannot be changed. Examples include integers, floats, and strings.
   - Mutable Types: Their values can be modified. Examples include lists, dictionaries, and sets.

   Example (Immutable):

   ```
   s = "Hello"
   s[0] = "h"  # This will raise a TypeError, since strings are immutable
   ```

   Example (Mutable):

   ```
   lst = [1, 2, 3]
   lst[0] = 99  # This works because lists are mutable
   print(lst)  # Output: [99, 2, 3]
   ```

4. Practical Example:
   - Passing mutable and immutable types to a function:

   ```
   def modify_data(x, y):
       x = 20  # Immutable, won't affect the original
       y.append(4)  # Mutable, will affect the original
       return x
   ```

```python
num = 5
lst = [1, 2, 3]

modify_data(num, lst)

print(num)  # Output: 5 (unchanged)
print(lst)  # Output: [1, 2, 3, 4] (modified)
```

Key Code Example:

1. Global vs Local Variable Example:

```python
count = 0  # Global variable

def increment():
    global count  # Declare the global variable
    count += 1

increment()
print(count)  # Output: 1
```

2. Function Modifying List (Mutable Data):

```python
def add_item_to_list(lst, item):
    lst.append(item)

my_list = [1, 2, 3]
add_item_to_list(my_list, 4)
print(my_list)  # Output: [1, 2, 3, 4] (list modified)
```

Best Practices:

- Avoid modifying global variables inside functions unless absolutely necessary.
- Use functions to encapsulate logic, and minimize side effects on mutable arguments.
- Be mindful of mutable vs immutable data types when passing arguments to functions.

Lecture 11: Error Types, Systematic Debugging, Exceptions

Key Topics:

- Types of errors in Python.
- Debugging techniques.
- Using try, except, and finally for handling exceptions.
- Writing robust code that handles errors gracefully.

Summary:

1. Types of Errors:
    - Syntax Errors: Mistakes in the structure of the code (e.g., missing parentheses).

      ```
      # Syntax Error
      print("Hello, World!"
      ```
    - Runtime Errors: Errors that occur during execution (e.g., dividing by zero, file not found).

      ```
      # Runtime Error
      x = 10 / 0  # Division by zero error
      ```
    - Logical Errors: The program runs but produces incorrect results due to a flaw in the logic.

      ```
      # Logical Error
      total = 10 + "5"  # Cannot add integer to string
      ```
2. Systematic Debugging:
    - Print Debugging: Insert print statements to inspect variable values at different stages.

      ```
      def calculate_area(radius):
          print(f"Radius: {radius}")
          return 3.14 * radius ** 2

      print(calculate_area(5))
      ```
    - Using a Debugger: Many IDEs (e.g., PyCharm, VSCode) come with debuggers that allow stepping through code line by line to examine variable states and flow.
    - Test-Driven Development (TDD): Writing tests first to define expected behavior before writing actual code.
3. Exception Handling (try, except, finally):
    - try block: Contains code that may raise an exception.
    - except block: Handles the exception if it occurs.

- finally block: Executes code that runs regardless of whether an exception occurs, often used for cleanup.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid input! Please enter a number.")
else:
    print(f"The result is: {result}")
finally:
    print("Execution finished.")
```

Explanation:

- If the user enters 0, a ZeroDivisionError is caught.
- If the user enters non-numeric input, a ValueError is caught.
- The finally block always runs, regardless of the outcome.

4. Common Python Exceptions:
    - ZeroDivisionError: Raised when dividing by zero.
    - ValueError: Raised when a function receives an argument of the correct type but inappropriate value.
    - IndexError: Raised when accessing an invalid index in a list or tuple.
    - KeyError: Raised when accessing a non-existent key in a dictionary.
    - FileNotFoundError: Raised when trying to access a file that doesn't exist.

Key Code Example:

1. Handling Multiple Exceptions:

```
try:
    file = open("nonexistent_file.txt", "r")
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("You do not have permission to open this file.")
```

2. Gracefully Handling User Input:

```
def get_integer():
    while True:
        try:
```

```
        return int(input("Please enter an integer: "))
    except ValueError:
        print("That's not an integer. Try again.")

print(get_integer())  # Will keep asking until a valid integer is
entered.
```

Best Practices:

- Use specific exception types rather than catching all exceptions with a generic except block.
- Always handle exceptions gracefully and provide feedback to the user.
- Use finally for essential cleanup tasks, such as closing files or releasing resources.

Lecture 12: Python Standard Library, Modules, Packages

Key Topics:

- Introduction to Python's standard library.
- Using modules to extend functionality.
- Creating and using Python packages.
- Importing and organizing code for reuse.

Summary:

1. Python Standard Library:
   - Python comes with a rich standard library that includes many modules and packages for various tasks like file handling, data manipulation, math functions, and more.
   - Modules can be imported to access functions and variables defined in them.
   - Example:

     ```
     import math
     print(math.sqrt(16))  # Output: 4.0
     ```
2. Using Modules:
   - Importing a module:

     ```
     import module_name  # Importing an entire module
     ```
   - Using functions from a module:

     ```
     import math
     print(math.pow(2, 3))  # Output: 8.0 (2 raised to the power of 3)
     ```
   - Importing specific functions from a module:

     ```
     from math import sqrt
     print(sqrt(25))  # Output: 5.0
     ```
3. Creating Your Own Modules:
   - A module is simply a Python file (with a .py extension) that contains functions, classes, or variables.
   - Example:
     1. my_module.py:

        ```
        def greet(name):
            return f"Hello, {name}!"
        ```
     2. main.py (uses my_module):

        ```
        import my_module
        ```

```
print(my_module.greet("Alice"))  # Output: Hello,
Alice!
```

4. Packages:
   - A package is a collection of modules stored in a directory that includes an __init__.py file.
   - Example:
     - Directory structure:

       ```
       mypackage/
           __init__.py
           module1.py
           module2.py
       ```
   - To import from a package:

     ```
     from mypackage import module1
     from mypackage.module2 import function_name
     ```
5. Popular Python Standard Library Modules:
   - os: Interacting with the operating system (e.g., file operations, environment variables).

     ```
     import os
     print(os.getcwd())  # Output: current working directory
     ```
   - sys: Access system-specific parameters (e.g., command-line arguments).

     ```
     import sys
     print(sys.argv)  # Output: list of command-line arguments
     ```
   - datetime: Handling dates and times.

     ```
     from datetime import datetime
     print(datetime.now())  # Output: current date and time
     ```
   - random: Generating random numbers.

     ```
     import random
     print(random.randint(1, 10))  # Output: Random number between
     1 and 10
     ```
   - json: Working with JSON data.

     ```
     import json
     data = {"name": "Alice", "age": 30}
     json_data = json.dumps(data)  # Convert dictionary to JSON
     string
     print(json_data)
     ```
6. Using pip for External Libraries:

- Install external packages from the Python Package Index (PyPI) using the pip command.
  bash

  ```bash
  pip install requests  # Installs the 'requests' module
  ```

Key Code Example:

1. Using math and random Modules Together:

   ```python
   import math
   import random

   # Generate a random number and compute its square root
   num = random.randint(1, 100)
   print(f"Random number: {num}")
   print(f"Square root: {math.sqrt(num)}")
   ```
2. Creating a Custom Module:
   - math_operations.py:

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

     def subtract(a, b):
         return a - b
     ```
   - main.py:

     ```python
     import math_operations

     result = math_operations.add(10, 5)
     print(f"Addition result: {result}")  # Output: 15
     ```

Best Practices:

- Module Naming: Use clear and descriptive names for your modules and functions.
- Keep Functions Focused: Each function should do one thing and do it well.
- Avoid Using from module import *: It can lead to namespace pollution and unclear code.

Lecture 13: Game Design with Functions

Key Topics:

- Designing simple games in Python using functions.
- Breaking down game logic into manageable functions.
- Example: A number-guessing game.

Summary:

1. Game Design Principles:
    - Modularity: Break down the game into small, manageable tasks using functions.
    - Interaction: Functions can be used to handle user input, game logic, and output.
    - Game Flow: Use functions to control the flow of the game, such as starting the game, processing the player's actions, and ending the game.
2. Example Game: Number Guessing Game
    - A simple game where the computer selects a random number, and the player guesses it.

    Game Components:
    - Generating a random number: Use random to select a number.
    - Getting user input: Use input() to prompt the user for their guess.
    - Checking the guess: Compare the guess with the correct answer.
    - Game loop: Keep the game running until the player guesses the correct number.

Step-by-Step Breakdown:

1. Game Setup:
    - Import necessary modules (random for random number generation).
    - Define a function to start the game and generate a random number.
2. Main Game Function:
    - Use a loop to continuously prompt the user for guesses until they are correct.
3. Game Feedback:
    - Provide feedback on the user's guess (whether it's too high, too low, or correct).

Example Code:

```
import random
```

```python
def generate_number():
    """Generate a random number between 1 and 100."""
    return random.randint(1, 100)

def get_user_guess():
    """Prompt the user to guess a number."""
    guess = int(input("Guess a number between 1 and 100: "))
    return guess

def give_feedback(guess, number):
    """Provide feedback on whether the guess is too high, too low, or
correct."""
    if guess < number:
        print("Too low!")
    elif guess > number:
        print("Too high!")
    else:
        print("Congratulations! You guessed the number!")

def play_game():
    """Main game loop."""
    number = generate_number()  # Generate a random number
    guessed_correctly = False

    while not guessed_correctly:
        guess = get_user_guess()  # Get user's guess
        if guess == number:
            give_feedback(guess, number)
            guessed_correctly = True
        else:
            give_feedback(guess, number)

# Start the game
if __name__ == "__main__":
    play_game()
```

Explanation of Functions:

- generate_number(): This function generates a random number between 1 and 100.
- get_user_guess(): Prompts the user for a guess and returns it.
- give_feedback(): Checks if the guess is too high, too low, or correct and prints feedback accordingly.

- play_game(): Controls the main game flow, repeatedly asking for guesses until the player guesses the correct number.

Game Flow:

1. The game starts and a random number is generated.
2. The user is asked to guess the number.
3. The feedback (too high/low/correct) is provided after each guess.
4. The loop continues until the user guesses correctly.

Extending the Game:

- Adding a Guess Counter: Keep track of how many guesses the player makes.
- Adding Difficulty Levels: Allow the player to select a difficulty (e.g., a number range of 1–100 for easy or 1–1000 for hard).
- Allowing Multiple Players: Extend the game to allow more than one player to guess alternately.

Key Design Points:

- Modularity: Each part of the game (number generation, user input, feedback, game loop) is separated into its own function.
- User Interaction: The game continuously interacts with the user by prompting for input and providing feedback.
- Control Flow: The game logic (checking guesses and providing feedback) is controlled by loops and conditionals, making it easy to follow and extend.

Lecture 14: Bottom-Up Design, Turtle Graphics, Robotics

Key Topics:

- Bottom-up design methodology.
- Introduction to Turtle Graphics for drawing and visual programming.
- Using Python in robotics applications.

Summary:

1. Bottom-Up Design:
    - Bottom-up design focuses on building small, simple components first, which are then integrated into a larger, more complex system.
    - This approach contrasts with top-down design, where you start with the overall system and break it down.
    - Advantages:
        - Easier to understand and implement smaller components.
        - Debugging is simpler since you work with smaller sections at a time.
        - Flexibility in modifying individual components without affecting the whole system.
2. Steps in Bottom-Up Design:
    - Step 1: Start with the smallest building blocks or functions.
    - Step 2: Build and test these blocks thoroughly.
    - Step 3: Combine blocks to form larger modules.
    - Step 4: Integrate these modules into the final system.

   Example:
    - Start by writing simple functions for individual components (e.g., movement for a robot).
    - Test these functions, then combine them to control more complex robot behaviors.

---

3. Turtle Graphics:
    - Turtle Graphics is a Python library used to introduce programming concepts by controlling a "turtle" that moves around the screen and draws lines.
    - It is a great tool for visual learners and young programmers.
4. Basic Turtle Operations:
    - Importing the Library:

      ```
      import turtle
      ```
    - Creating a Turtle Object:

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

- Basic Commands:
  - forward(distance): Moves the turtle forward by the given distance.
  - left(angle): Turns the turtle left by the specified angle.
  - right(angle): Turns the turtle right by the specified angle.
  - penup(): Lifts the pen, preventing drawing.
  - pendown(): Lowers the pen to start drawing.
  - color(color_name): Changes the pen color.

Example: Drawing a Square

```
import turtle

t = turtle.Turtle()

for _ in range(4):  # Loop to draw 4 sides
    t.forward(100)  # Move forward 100 units
    t.left(90)      # Turn left 90 degrees

turtle.done()  # Keeps the window open
```

5. Robotics with Python:
   - Python is widely used in robotics due to its simplicity and extensive libraries.
   - Libraries like RPi.GPIO (for Raspberry Pi) allow you to interface with hardware (e.g., motors, sensors).

   Example: Simple motor control with Raspberry Pi:

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

motor_pin = 17  # GPIO pin for motor
GPIO.setup(motor_pin, GPIO.OUT)

# Turn motor on
GPIO.output(motor_pin, GPIO.HIGH)
time.sleep(2)  # Run motor for 2 seconds

# Turn motor off
GPIO.output(motor_pin, GPIO.LOW)

GPIO.cleanup()  # Clean up GPIO setup
```

6. Combining Bottom-Up Design with Turtle and Robotics:
    - You can apply the bottom-up approach in building a robot that performs tasks like moving, detecting objects, and drawing patterns.
    - For example, start by designing basic functions like moving the robot in different directions, then add functions to control sensors, and finally integrate everything into a robot that performs complex tasks like drawing on the ground.

Key Code Example:

1. Turtle Drawing a Star:

```
import turtle

t = turtle.Turtle()
t.color("blue")

for _ in range(5):
    t.forward(100)  # Draw a side of the star
    t.right(144)    # Turn 144 degrees to form the star

turtle.done()
```

2. Using Bottom-Up Design for a Simple Robot:
    - Step 1: Write individual functions to control the robot’s movements:

    ```
    def move_forward(distance):
        # Code to move the robot forward
        pass

    def turn_left(angle):
        # Code to turn the robot left
        pass
    ```

    - Step 2: Combine the functions to create more complex behaviors, such as drawing a square:

    ```
    def draw_square():
        for _ in range(4):
            move_forward(100)
            turn_left(90)
    ```

    - Step 3: Integrate these behaviors into a larger program, like a robot that draws shapes or navigates an area.

Best Practices:

- Use bottom-up design when creating complex robotic systems or applications.
- Start with small, testable components and gradually combine them into larger systems.
- In robotics, always test your hardware connections (e.g., motors, sensors) separately before integrating them into the main program.

Lecture 15: Event‑Driven Programming

Key Topics:

- Introduction to event-driven programming.
- How events trigger actions in Python.
- Practical examples: GUI applications and event handling.

Summary:

1. What is Event-Driven Programming?
   - Event-driven programming is a paradigm where the flow of the program is determined by user actions (events), sensor inputs, or messages from other programs.
   - Events such as mouse clicks, key presses, or sensor readings trigger certain actions or functions in response.
   - Common in graphical user interfaces (GUIs), robotics, and networked applications.
2. Key Concepts in Event-Driven Programming:
   - Event Loop: The core of event-driven programming. The event loop listens for events and calls the corresponding event handler functions when they occur.
   - Event Handlers: Functions that define the actions to take when specific events occur.
   - Callbacks: Functions passed as arguments to other functions or components that are invoked when an event happens.
3. Event-Driven Programming in Python:
   - Python libraries such as Tkinter (for GUIs) and pygame (for games) are examples of event-driven frameworks.
   - The typical flow involves setting up an event handler and then entering the main event loop where the program waits for events to process.

---

4. Basic Example of Event-Driven Programming:
   - GUI with Tkinter: Tkinter is a Python library for creating simple GUI applications that is based on event-driven programming. We define the actions to take (event handlers) and associate them with widgets (e.g., buttons).

   Example: A simple GUI program with a button that triggers an event when clicked:

```
import tkinter as tk

def on_button_click():
    print("Button clicked!")
```

```
# Create the main window
window = tk.Tk()
window.title("Event-Driven Example")

# Create a button and associate it with the event handler
button = tk.Button(window, text="Click me",
command=on_button_click)
button.pack()

# Start the event loop
window.mainloop()
```

Explanation:

- window.mainloop() starts the event loop, which listens for user actions.
- on_button_click is called when the user clicks the button.

5. Event Loop:
    - The event loop is an ongoing process that waits for events (such as a button click or a mouse movement) and triggers the appropriate actions. In the example above, the event loop listens for events in the Tkinter window.

Example of Event-Driven Game (Using pygame):

- In games, event-driven programming is used to detect user inputs like key presses or mouse clicks and update the game accordingly.

Example: Handling keyboard events in a simple pygame program:

```
import pygame

pygame.init()
screen = pygame.display.set_mode((400, 300))
clock = pygame.time.Clock()

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:  # Close window
            running = False
        if event.type == pygame.KEYDOWN:  # Key press event
            if event.key == pygame.K_LEFT:
                print("Left arrow key pressed")
            if event.key == pygame.K_RIGHT:
                print("Right arrow key pressed")
```

```
    screen.fill((0, 0, 0))  # Fill screen with black
    pygame.display.flip()  # Update the screen

    clock.tick(60)  # 60 frames per second

pygame.quit()
```

Explanation:

- The pygame.event.get() function returns a list of all events, and we check for specific types of events, such as QUIT (window close) and KEYDOWN (key press).
- The game updates in a loop, handling events and rendering the screen each frame.

---

6. Event-Driven Robotics:
   - In robotics, event-driven programming can be used to handle inputs from sensors, such as detecting an object or reaching a destination.
   - For example, a robot may wait for a button press or sensor reading (an event) to start a task or navigate.

Key Code Example:

1. Event-Driven Button Example with Tkinter:

```
import tkinter as tk

def on_button_click():
    print("Button was clicked!")

def on_key_press(event):
    print(f"Key pressed: {event.char}")

# Setup the window
window = tk.Tk()
window.title("Event-Driven Programming")

# Create a button and associate it with a function
button = tk.Button(window, text="Click Me",
command=on_button_click)
button.pack()

# Bind key press event to a function
window.bind("<KeyPress>", on_key_press)

# Start the event loop
```

```
window.mainloop()
```

Explanation:

- window.bind("<KeyPress>", on_key_press) listens for any key press and triggers the on_key_press function.

Best Practices:

- Keep event handlers simple: Event handlers should only contain the logic needed to handle the event. Long or complex logic should be moved to other functions.
- Avoid blocking the event loop: Event loops should be non-blocking to ensure the program remains responsive to user inputs.
- Use try and except: Handle potential errors within event handlers to prevent the program from crashing.

Lecture 16: Visualizing Data and Creating Simulations

Key Topics:

- Introduction to data visualization.
- Tools for visualizing data in Python (e.g., matplotlib, seaborn).
- Simulating real-world processes (e.g., Monte Carlo simulations, physics simulations).

Summary:

1. Introduction to Data Visualization:
    - Data visualization is the graphical representation of data and information.
    - Helps in understanding patterns, trends, and outliers within datasets.
    - Python provides several libraries for data visualization, with matplotlib and seaborn being the most popular.
2. Why Data Visualization?
    - Allows for quick insights into large datasets.
    - Makes data more accessible and interpretable.
    - Improves decision-making by visualizing relationships between variables.

---

3. Using matplotlib for Data Visualization:
    - matplotlib is a widely used library for creating static, animated, and interactive visualizations in Python.
    - The most commonly used module is pyplot, which provides a MATLAB-like interface for creating various types of plots.

   Common Plots in matplotlib:
    - Line Plot: Displays data trends over a continuous range.
    - Bar Chart: Compares discrete categories.
    - Histogram: Displays frequency distribution of numerical data.
    - Scatter Plot: Shows relationships between two continuous variables.

   Example: Line plot using matplotlib:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

plt.plot(x, y)
plt.title("Line Plot Example")
plt.xlabel("X axis")
```

```
plt.ylabel("Y axis")
plt.show()
```

Explanation:

- plt.plot(x, y) creates the line plot using x and y values.
- plt.title, plt.xlabel, and plt.ylabel add the plot title and axis labels.
- plt.show() displays the plot in the default viewer.

4. Advanced Plotting with seaborn:
    - seaborn is built on top of matplotlib and simplifies creating attractive and informative statistical graphics.
    - It integrates with pandas for easier data handling and supports complex visualizations.

Example: Scatter plot using seaborn:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load a sample dataset from seaborn
tips = sns.load_dataset("tips")

sns.scatterplot(x="total_bill", y="tip", data=tips)
plt.title("Total Bill vs Tip")
plt.show()
```

Explanation:

- sns.scatterplot creates a scatter plot with the total_bill on the x-axis and tip on the y-axis from the tips dataset.
- The sns library automatically handles aesthetics and provides better-looking plots than matplotlib by default.

5. Simulations in Python:
    - Simulations are often used to model and predict complex systems, often involving randomness or uncertainty.
    - Monte Carlo simulations are a popular method for simulating processes using random sampling to obtain numerical results.
6. Monte Carlo Simulation Example:
    - Example: Estimating the value of π using a Monte Carlo simulation:

```
import random
import matplotlib.pyplot as plt

inside_circle = 0
total_points = 10000

x_inside = []
```

```
y_inside = []
x_outside = []
y_outside = []

for _ in range(total_points):
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)

    if x**2 + y**2 <= 1:
        inside_circle += 1
        x_inside.append(x)
        y_inside.append(y)
    else:
        x_outside.append(x)
        y_outside.append(y)

pi_estimate = (inside_circle / total_points) * 4

# Plot the points inside and outside the circle
plt.scatter(x_inside, y_inside, color='blue', s=1)
plt.scatter(x_outside, y_outside, color='red', s=1)
plt.gca().set_aspect('equal', adjustable='box')
plt.title(f"Monte Carlo Simulation Estimate of Pi: {pi_estimate}")
plt.show()
```

Explanation:

- This simulation randomly generates points within a square of side length 2.
- Points that fall within a circle inscribed in that square are counted.
- The ratio of points inside the circle to the total points approximates the value of π, which is multiplied by 4 for the final estimate.
- The blue points represent those inside the circle, and the red points represent those outside.

---

7. Simulating Real-World Processes:
    - Python can be used to simulate real-world phenomena such as the spread of diseases (epidemic modeling), financial markets, or physical processes (like the motion of particles or fluids).
    - These simulations often rely on random number generation and iterative models to approximate the system behavior over time.

Best Practices:

- Clear and descriptive visualizations: Always label your axes, include a title, and use appropriate colors to represent your data.
- Data handling: Clean and preprocess your data before visualizing it to ensure accurate and meaningful insights.
- Use appropriate plots: Choose the right plot type for the data you’re working with (e.g., use a scatter plot for relationships, a bar chart for comparisons, etc.).

Lecture 17: Final Project: Putting It All Together

Key Topics:

- Integrating the concepts learned throughout the course into a comprehensive project.
- Planning, developing, and testing a Python-based project.
- Final project idea: Developing a simple Python-based application or simulation.

Summary:

1. Project Overview:
    - This final project will allow you to apply the concepts you've learned in previous lectures. The goal is to build a functional program that demonstrates the use of Python's basic features, including data structures, functions, event handling, visualization, and even simulations.
    - The project could be a simple game, a data visualization tool, a simulation, or a combination of multiple topics.
2. Steps for Project Development:
    - Step 1: Define the Project Scope:
        - Decide what your final project will be. Consider what interests you most—game development, data analysis, simulations, etc.
        - Example ideas include:
            - A number-guessing game with advanced features (difficulty levels, hints).
            - A data visualization tool that reads from a dataset and generates various plots.
            - A Monte Carlo simulation to model a real-world process, such as stock market fluctuations.
    - Step 2: Break the Project into Functions/Modules:
        - Use modular programming and break down the tasks into smaller functions or modules.
        - Example for a guessing game:
            - generate_number() – to generate a random number.
            - get_user_input() – to get the user's guess.
            - check_guess() – to compare the guess with the target number.
            - provide_feedback() – to give feedback (too high, too low).
    - Step 3: Build the Project Iteratively:
        - Start by building the basic functionality.

            ▪ Add new features step by step, testing each one as you go.
        ○ Step 4: Use Testing and Debugging:
            ▪ Test early and often. Break your project into small components, and test them one by one.
            ▪ Use print debugging and tools like assert statements to ensure everything works as expected.
            ▪ Debug any issues and ensure all errors are handled gracefully.

3. Project Example: Number Guessing Game with Features
    - Basic Requirements:
        - The user should guess a randomly generated number within a certain range (e.g., 1 to 100).
        - Provide feedback on whether the guess is too high, too low, or correct.
        - Add a counter to track the number of guesses.
    - Advanced Features:
        - Allow the user to select a difficulty level (easy, medium, hard).
        - Offer hints after a certain number of incorrect guesses.
        - Show a leaderboard with the best times or fewest guesses.

Example: Advanced Number Guessing Game with Features

```
import random

def generate_number(range_start, range_end):
    """Generate a random number within the given range."""
    return random.randint(range_start, range_end)

def get_user_input():
    """Prompt the user to enter a guess."""
    try:
        guess = int(input("Enter your guess: "))
        return guess
    except ValueError:
        print("Invalid input. Please enter a number.")
        return get_user_input()

def check_guess(guess, target_number):
    """Compare the guess with the target number and return feedback."""
    if guess < target_number:
```

```
        return "Too low!"
    elif guess > target_number:
        return "Too high!"
    else:
        return "Correct!"

def play_game():
    """Main game loop."""
    print("Welcome to the Number Guessing Game!")
    difficulty = input("Choose difficulty (easy, medium, hard):
").lower()

    if difficulty == "easy":
        range_start, range_end = 1, 50
    elif difficulty == "medium":
        range_start, range_end = 1, 100
    else:
        range_start, range_end = 1, 200

    target_number = generate_number(range_start, range_end)
    guesses = 0
    guessed_correctly = False

    while not guessed_correctly:
        guess = get_user_input()
        guesses += 1
        feedback = check_guess(guess, target_number)
        print(feedback)

        if feedback == "Correct!":
            guessed_correctly = True

    print(f"You guessed the number in {guesses} attempts!")

if __name__ == "__main__":
    play_game()
```

Explanation:

- The game allows the player to choose a difficulty level, which adjusts the range of numbers.
- The player is prompted for guesses, and feedback is provided if the guess is too high, too low, or correct.
- The game tracks the number of guesses and ends once the correct number is guessed.

Project Evaluation Criteria:

- Functionality: Does the project meet the objectives and work as intended?
- Code Quality: Is the code readable, well-organized, and efficient?
- Creativity: Does the project include additional features or enhancements?
- Testing and Debugging: Is the program tested for edge cases and errors? Does it handle exceptions gracefully?

Additional Project Ideas:

- Simulations:
  - Simulate a physical system (e.g., projectile motion, bouncing ball).
  - Monte Carlo simulations for risk analysis or optimization problems.
- Data Visualization:
  - Create a dashboard that visualizes a dataset using matplotlib and seaborn.
  - Build a Python program that fetches data from an API and visualizes it (e.g., weather data).
- Games and Interactive Applications:
  - Develop a text-based RPG (role-playing game) using functions to manage different parts of the game.
  - Build a graphical game using pygame or a simple GUI using Tkinter.

Best Practices for the Final Project:

- Plan the Project: Start with a clear design and breakdown of features.
- Document Your Code: Include comments and docstrings to explain the logic and steps.
- Test Early and Often: Ensure each part of your project works independently before combining it into a final version.
- Keep It Simple: Focus on completing a small, functional project. You can always expand it later with more features.
- Version Control: Use version control (e.g., Git) to keep track of changes and versions.