

# VU LEARNING HUB

[vulearninghub.xyz](https://vulearninghub.xyz)

**HIGHLIGHTED HANDOUT**

## CS306

Introduction to Python

**CS306 by VU Learning HUB**

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

# Notes for CS306 – Introduction to Python

(mid term preparation notes)

---

## CS306 – Introduction to Python (Handout)

Credit Hours: 3 (Theory)

Offered by: Virtual University of Pakistan

Course Type: Core Course (e.g., BSCS, ADP in Business Analytics)

---

### Course Objectives

- Understand the basics of Python programming.
  - Develop logic-building and problem-solving skills.
  - Use Python to perform data manipulation, file handling, and object-oriented programming.
- 

### Course Outline

#### 1. Introduction to Python

- History and evolution
  - Features of Python
  - Installation and environment setup
  - Writing your first Python program
- 

#### 2. Python Basics

1. Keywords and identifiers
  2. Variables and constants
  3. Data types: int, float, str, bool
  4. Type casting
  5. Basic input/output
  6. Comments
-

### 3. Operators and Expressions

1. Arithmetic, Assignment, Comparison
  2. Logical, Identity, Membership
  3. Operator precedence and associativity
- 

### 4. Conditional Statements

- if, if-else, if-elif-else
  - Nested conditions
- 

### 5. Loops and Iteration

- while and for loops
  - range() function
  - Nested loops
  - Loop control: break, continue, pass
- 

### 6. Functions

1. Defining and calling functions
  2. Arguments and return values
  3. Default and keyword arguments
  4. Variable scope
  5. Recursion
- 

### 7. Data Structures

#### *Lists*

- Creating, indexing, slicing
- Built-in functions (append, remove, sort, etc.)

#### *Tuples*

- Immutable lists
- Tuple operations

#### *Dictionaries*

- Key-value pairs

- Accessing, updating, deleting elements

### Sets

- Unordered collections
  - Set operations (union, intersection, etc.)
- 

## 8. String Manipulation

- Indexing and slicing
  - String methods (upper, lower, split, join, find, etc.)
  - Formatting strings
- 

## 9. File Handling

- Opening and closing files
  - Read and write operations
  - Working with with statement
  - Handling text and binary files
- 

## 10. Exception Handling

1. try, except, finally
  2. Raising exceptions
  3. Built-in exception types
- 

## 11. Object-Oriented Programming

- Classes and objects
  - Constructors (\_\_init\_\_)
  - Instance vs class variables
  - Inheritance and polymorphism
  - Encapsulation
- 

## 12. Modules and Packages

1. Importing built-in and custom modules

2. `__name__ == "__main__"` idiom

3. Creating packages with `__init__.py`

---

### 13. Python Standard Library Overview

- `math`, `random`, `datetime`, `os`, `sys`
  - `json` and `pickle` for data serialization
  - `re` for regular expressions
- 



### Assessment Breakdown

- Quizzes / Assignments / GDBs: 20%
  - Midterm Exam: 20–30% 
  - Final Exam: 50–60%
- 



### Recommended Tools

- Python (version 3.x)
  - IDLE / PyCharm / VS Code
  - Jupyter Notebooks (for experimentation)
- 



### Useful Resources

1. CS306 Playlist on VU YouTube Channel
  2. Official Python Docs
  3. W3Schools Python
- 

## CS306 – Lecture 1: Introduction to Python Programming

---

## Lecture Objectives

- Understand the evolution and background of Python.
  - Recognize the features that make Python a popular programming language.
  - Learn how to install Python and run your first program.
  - Know the basic structure and syntax of a Python script.
- 

## 1. What is Python?

Python is a **high-level, interpreted**, and **general-purpose** programming language. It was created by **Guido van Rossum** and released in **1991**.

---

## 2. History and Evolution of Python

| Version | Year | Highlights |
|---------|------|------------|
|---------|------|------------|

|            |      |                               |
|------------|------|-------------------------------|
| Python 1.0 | 1991 | First release, basic features |
|------------|------|-------------------------------|

|            |      |                                         |
|------------|------|-----------------------------------------|
| Python 2.0 | 2000 | List comprehensions, garbage collection |
|------------|------|-----------------------------------------|

|            |      |                                       |
|------------|------|---------------------------------------|
| Python 3.0 | 2008 | Major update, not backward compatible |
|------------|------|---------------------------------------|

**Note:** Python 3.x is the current and recommended version. Python 2 reached end of life in 2020. 

---

## 3. Key Features of Python

- **Simple and easy to learn:** Python's syntax is close to English.
  - **Interpreted:** Code is executed line-by-line.
  - **Dynamically typed:** No need to declare variable types.
  - **High-level language:** Abstracts low-level details.
  - **Object-Oriented:** Supports classes and objects.
  - **Extensive standard library:** Ready-to-use modules.
  - **Platform Independent:** Runs on Windows, Mac, Linux.
  - **Community Support:** Strong developer community.
-

## 4. Installing Python

 Official Website:

Visit <https://www.python.org>

 Installation Steps (Windows):

7. Download Python (prefer Python 3.x).
8. Run the installer.
9. Check the box **“Add Python to PATH”**.
10. Click **Install Now**.

 Verify Installation:

Open Command Prompt and type:

```
python --version
```

Or:

```
python
```

---

## 5. Writing and Running Python Code

 Methods:

### 4. Interactive Mode:

- Open Command Prompt or Terminal
- Type `python` to enter interactive mode
- Example:
- `>>> print("Hello, Python!")`

### 5. Script Mode:

- Write code in a file with `.py` extension
  - Save and run using:
  - `python filename.py`
- 

## 6. Your First Python Program

Example:

```
# This is a comment  
print("Hello, world!")
```

Output:

```
Hello, world!
```

---

## 7. Comments in Python

- Used to explain code
- Not executed by interpreter

### Single-line comment:

```
# This is a comment
```

### Multi-line (using triple quotes):

```
"""
This is a
multi-line comment
"""
```

---

## Summary

- Python is an easy-to-learn, powerful programming language.
- It supports multiple paradigms like procedural and object-oriented programming.
- You can use interactive or script mode to write Python code.
- Python 3.x is the recommended version.

---

## Practice Tasks

6. Install Python 3 from [python.org](https://python.org).
7. Run Python in interactive mode and type:
8. `print("Welcome to CS306!")`
9. Create and save a `.py` file that prints your name.

---

## CS306 – Lecture 2: Python Programming Basics

### Lecture Objectives

- Understand the concept of variables and data types in Python.
- Learn about identifiers and naming rules.
- Practice basic input and output operations.
- Distinguish between comments and docstrings.

## 1. Identifiers in Python

### Definition:

Identifiers are the names used to identify variables, functions, classes, modules, or other objects.

### Rules for Naming Identifiers:

- Can contain letters (a–z, A–Z), digits (0–9), and underscores (\_)
- Must begin with a letter or underscore (not a digit)
- Cannot be a Python **keyword** (e.g., `if`, `while`, `class`, etc.)
- Case-sensitive: `Name`, `name`, and `NAME` are different

 Good practice: Use meaningful names (e.g., `student_name` instead of `x`)

---

## 2. Python Keywords

Reserved words in Python that cannot be used as identifiers.

### Example Keywords Example Keywords

|                     |                          |
|---------------------|--------------------------|
| <code>if</code>     | <code>else</code>        |
| <code>for</code>    | <code>while</code>       |
| <code>def</code>    | <code>return</code>      |
| <code>import</code> | <code>from</code>        |
| <code>class</code>  | <code>try, except</code> |

Use `help("keywords")` in the Python shell to see the full list.

---

## 3. Variables in Python

### Definition:

A variable is a **container** for storing data values.

```
name = "Ali"
age = 20
```

Python uses **dynamic typing**, so you don't need to declare data types explicitly.

#### Variable Assignment:

- = is the **assignment operator**
  - Multiple assignments:
    - `x = y = z = 10` # Same value to multiple variables
    - `a, b = 5, "Hello"` # Multiple values to multiple variables
- 

## 4. Data Types in Python

| Data Type | Example     | Description                 |
|-----------|-------------|-----------------------------|
| int       | 5, -12      | Integer values              |
| float     | 3.14, -2.5  | Decimal numbers             |
| str       | "Python"    | Text (string)               |
| bool      | True, False | Boolean values              |
| list      | [1, 2, 3]   | Ordered, mutable sequence   |
| tuple     | (1, 2, 3)   | Ordered, immutable sequence |
| dict      | {"a": 1}    | Key-value pairs             |
| set       | {1, 2, 3}   | Unordered, unique elements  |

Use the `type()` function to check the data type of a variable.

```
a = 10
print(type(a))    # <class 'int'>
```

---

## 5. Input and Output Functions

 Output: `print()`

Displays data to the screen.

```
print("Welcome to Python!")
name = "Ali"
print("Hello", name)
```

 Input: `input()`

Takes input from the user as a **string** by default.

```
name = input("Enter your name: ")
age = input("Enter your age: ")
print("Hi", name, "you are", age, "years old.")
```

Convert input to number using `int()` or `float()` if needed.

```
num = int(input("Enter a number: "))
```

---

## 6. Comments and Documentation Strings

### Comments:

- Used to describe and explain code
- Ignored during execution

```
# This is a single-line comment
```

### Docstrings:

- Multi-line documentation used for functions, classes, or modules
- Placed just after definition
- Use triple quotes `""" """`

```
def greet():
    """
    This function prints a greeting message.
    """
    print("Hello from Python!")
```

Use `help(function_name)` to see the docstring.

---

## 7. Practice Examples

```
# Program to take name and age and display a message
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print("Welcome", name, "you are", age, "years old.")
# Using multiple assignments
x, y, z = 10, 20, 30
print(x, y, z)
```

---

## Summary

- Identifiers name variables, functions, etc., and must follow naming rules.
- Variables store values and Python handles their data types dynamically.
- Use `input()` and `print()` for basic interaction.
- Comments and docstrings improve code readability and documentation.

## Homework / Self-Practice

### 4. Write a Python script to:

- Take your name and birth year.
- Calculate and display your age.

### 5. Identify invalid variable names:

- `2nd_name`, `first-name`, `_value`, `class`

### 6. Write a docstring for a function that adds two numbers.

---

---

## CS306 – Lecture 3: Operators and Expressions in Python

---

### Lecture Objectives

- Understand the different types of operators in Python.
  - Learn about operator precedence and associativity.
  - Use expressions in practical programming tasks.
  - Explore type conversion techniques.
- 

### 1. What is an Operator?

An **operator** is a symbol or function that performs operations on variables and values (operands).

---

### 2. Types of Operators in Python

| Operator Type        | Description                    |
|----------------------|--------------------------------|
| Arithmetic Operators | Basic math operations          |
| Assignment Operators | Assign values to variables     |
| Comparison Operators | Compare values                 |
| Logical Operators    | Combine conditional statements |
| Bitwise Operators    | Work with bits                 |

| Operator Type        | Description                   |
|----------------------|-------------------------------|
| Membership Operators | Test membership in a sequence |
| Identity Operators   | Compare memory locations      |

---

### + 3. Arithmetic Operators

| Operator | Description    | Example | Result |
|----------|----------------|---------|--------|
| +        | Addition       | 5 + 3   | 8      |
| -        | Subtraction    | 5 - 2   | 3      |
| *        | Multiplication | 4 * 2   | 8      |
| /        | Division       | 10 / 2  | 5.0    |
| //       | Floor Division | 7 // 2  | 3      |
| %        | Modulus        | 7 % 2   | 1      |
| **       | Exponentiation | 2 ** 3  | 8      |

---

### 4. Assignment Operators

| Operator | Example | Equivalent |
|----------|---------|------------|
| =        | x = 5   | x = 5      |
| +=       | x += 3  | x = x + 3  |
| -=       | x -= 2  | x = x - 2  |
| *=       | x *= 4  | x = x * 4  |
| /=       | x /= 2  | x = x / 2  |
| %=       | x %= 3  | x = x % 3  |
| **=      | x **= 2 | x = x ** 2 |
| //=      | x //= 2 | x = x // 2 |

---

## 5. Comparison (Relational) Operators

| Operator | Description           | Example | Result |
|----------|-----------------------|---------|--------|
| ==       | Equal to              | 5 == 5  | True   |
| !=       | Not equal to          | 4 != 3  | True   |
| >        | Greater than          | 5 > 3   | True   |
| <        | Less than             | 3 < 4   | True   |
| >=       | Greater than or equal | 5 >= 5  | True   |
| <=       | Less than or equal    | 4 <= 5  | True   |

---

## 6. Logical Operators

Used to combine conditional statements.

| Operator | Description                      | Example          | Result |
|----------|----------------------------------|------------------|--------|
| and      | True if both conditions are True | x > 2 and x < 10 | True   |
| or       | True if at least one is True     | x < 2 or x > 10  | False  |
| not      | Reverses the result              | not (x > 5)      | False  |

---

## 7. Bitwise Operators (Basic Introduction)

Operate on binary digits of numbers.

| Operator   | Symbol | Example | Meaning             |
|------------|--------|---------|---------------------|
| AND        | &      | 5 & 3   | Binary AND          |
| OR         |        | 5   3   | Binary OR           |
| XOR        | ^      | 5 ^ 3   | Binary XOR          |
| NOT        | ~      | ~5      | Binary NOT          |
| Left Shift | <<     | 5 << 1  | Shift left by 1 bit |

| Operator | Symbol | Example | Meaning |
|----------|--------|---------|---------|
|----------|--------|---------|---------|

|             |    |        |                      |
|-------------|----|--------|----------------------|
| Right Shift | >> | 5 >> 1 | Shift right by 1 bit |
|-------------|----|--------|----------------------|

---

## 8. Membership Operators

Used to test for membership in a sequence (like string, list, tuple).

| Operator | Description                  | Example                   |
|----------|------------------------------|---------------------------|
| in       | True if value is present     | 'a' in 'apple' → True     |
| not in   | True if value is not present | 'z' not in 'apple' → True |

---

## 9. Identity Operators

Used to compare memory locations.

| Operator | Description                                    | Example    |
|----------|------------------------------------------------|------------|
| is       | True if two variables point to the same object | x is y     |
| is not   | True if not same object                        | x is not y |

---

## 10. Operator Precedence and Associativity

Operator precedence determines the order in which operations are performed.

### Example:

```
result = 10 + 5 * 2    # result = 10 + (5 * 2) = 20
```

### Precedence Order (High to Low):

4. () Parentheses
5. \*\* Exponentiation
6. \*, /, //, % Multiplication/Division
7. +, - Addition/Subtraction
8. ==, !=, >, <, >=, <= Comparisons
9. not
10. and

11. or

### Associativity:

- Most operators are **left-to-right**
- Exponentiation **\*\*** is **right-to-left**

---

## 11. Type Conversion (Type Casting)

### Implicit Conversion

Python automatically converts data types.

```
x = 5          # int
y = 2.5        # float
z = x + y      # float (7.5)
```

### Explicit Conversion

Manually convert types using functions like:

- `int()`
- `float()`
- `str()`

```
num = int("10")      # Converts string to integer
text = str(123)      # Converts number to string
```

---

## 12. Practice Examples

# Arithmetic example

```
a = 10
b = 3
print(a + b, a % b, a // b)
```

# Logical example

```
x = 5
print(x > 3 and x < 10)
```

# Type conversion

```
age = input("Enter your age: ")
print("Your age + 5 is:", int(age) + 5)
```

---

## Summary

- Python supports many operator types like arithmetic, assignment, comparison, logical, etc.
- Operator precedence affects how expressions are evaluated.

- Python allows both implicit and explicit type conversion.

---

### Homework / Self-Practice

4. Write a program that takes two numbers and prints:
  - Their sum, product, quotient, and remainder.
5. Create a program that:
  - Checks if a number is between 10 and 20.
6. Try out different assignment operators and see how values change.

---

## CS306 – Lecture 4: Conditional Statements in Python

### Lecture Objectives

- Understand decision-making in Python using conditional statements.
- Learn how to use `if`, `if-else`, and `if-elif-else`.
- Handle nested conditional logic.
- Understand indentation rules for blocks.

---

### 1. Introduction to Control Flow

In Python, control flow lets you execute certain parts of the code based on conditions. This is mainly done using:

- `if` statements
- `if-else` statements
- `if-elif-else` chains

---

### 2. The `if` Statement

The `if` statement is used to test a condition. If the condition is true, a block of code is executed.

Syntax

```
if condition:
    # block of code
```

### Example

```
x = 10
if x > 5:
    print("x is greater than 5")
```

---

## 3. The if-else Statement

Used to execute one block if the condition is true, and another if it is false.

### Syntax

```
if condition:
    # true block
else:
    # false block
```

### Example

```
age = 17
if age >= 18:
    print("You can vote")
else:
    print("You are not eligible to vote")
```

---

## 4. The if-elif-else Ladder

Used for multiple conditions.

### Syntax

```
if condition1:
    # block 1
elif condition2:
    # block 2
elif condition3:
    # block 3
else:
    # final block
```

### Example

```
marks = 85

if marks >= 90:
    print("Grade: A+")
elif marks >= 80:
    print("Grade: A")
elif marks >= 70:
    print("Grade: B")
else:
    print("Grade: C or below")
```

---

## 5. Nested if Statements

You can place one `if` statement inside another.

Example

```
num = 15
if num > 10:
    if num < 20:
        print("Number is between 10 and 20")
```

---

## 6. Indentation in Python

Python uses **indentation** (whitespace) to define code blocks.

- The default is 4 spaces.
- All code in the same block **must be aligned**.

Incorrect Example

```
if x > 5:
print("Wrong indentation") # ❌
```

Correct Example

```
if x > 5:
    print("Correct indentation") # ✅
```

---

## 7. Practical Examples

Example 1: Even or Odd Checker

```
number = int(input("Enter a number: "))
if number % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

Example 2: Voting Eligibility

```
age = int(input("Enter your age: "))
if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible")
```

Example 3: Largest of Three Numbers

```
a = 10
b = 20
c = 15

if a >= b and a >= c:
    print("a is largest")
elif b >= a and b >= c:
    print("b is largest")
else:
```

```
print("c is largest")
```

---

## 8. Common Mistakes to Avoid

- Forgetting colons : after if, elif, else
  - Misaligned indentation
  - Using = instead of == in conditions
- 

## 9. Real-World Use Case

### Login Validation

```
username = input("Enter username: ")
password = input("Enter password: ")

if username == "admin" and password == "12345":
    print("Login successful")
else:
    print("Invalid credentials")
```

---

## Summary

- if, if-else, and if-elif-else are used for decision-making.
  - Python relies on indentation, not braces {} like C/C++.
  - Nested conditions help manage more complex logic.
- 

## Homework / Self-Practice

1. Write a program that checks whether a number is positive, negative, or zero.
  2. Create a program that takes the percentage and assigns grades using if-elif-else.
  3. Develop a nested condition that checks:
    - If the user is adult (age  $\geq 18$ ),
    - If they are also a citizen, print “Eligible to vote”.
- 
- 

## CS306 – Lecture 5: Loops in Python (Part 1)

---

## Lecture Objectives

- Understand the need for loops.
  - Learn the syntax and use of the `while` loop.
  - Learn how to use the `for` loop.
  - Understand the use of `break`, `continue`, and `pass` statements.
  - Explore iteration through strings, lists, and ranges.
- 

### 1. What is a Loop?

A **loop** is a control structure that allows you to repeat a block of code **multiple times** until a condition is met.

Python supports two main types of loops:

- `while` loop
  - `for` loop
- 

### 2. The `while` Loop

Executes a block of code **as long as** the condition is `True`.

#### Syntax

```
while condition:
    # block of code
```

#### Example

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

#### Output

```
1
2
3
4
5
```

◆ Be careful: If the condition never becomes `False`, you will have an **infinite loop**.

---

### 3. The `for` Loop

Used to iterate over a **sequence** (like a list, tuple, string, or range).

#### Syntax

```
for variable in sequence:
    # block of code
```

#### Example

```
for i in range(1, 6):
    print(i)
```

#### Output

```
1
2
3
4
5
```

---

### 4. The `range()` Function

Often used with `for` loops to generate sequences of numbers.

#### Syntax

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

#### Examples

```
range(5)           # 0 to 4
range(1, 6)        # 1 to 5
range(0, 10, 2)    # 0, 2, 4, 6, 8
```

---

### 5. Iterating Through Data Structures

Iterate over a string

```
for letter in "Python":
    print(letter)
```

Iterate over a list

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

---

### 6. Loop Control Statements

#### `break`

Used to **exit** a loop prematurely.

```
for i in range(10):
    if i == 5:
        break
```

```
    print(i)
    continue
```

Skips the **current iteration** and continues with the next one.

```
for i in range(5):
    if i == 2:
        continue
    print(i)

pass
```

Does **nothing**; used as a placeholder.

```
for i in range(3):
    pass # placeholder for future code
```

---

## 7. Practical Examples

Example 1: Sum of first 10 numbers

```
total = 0
for i in range(1, 11):
    total += i
print("Sum:", total)
```

Example 2: Print even numbers between 1 to 20

```
for i in range(1, 21):
    if i % 2 == 0:
        print(i)
```

Example 3: Infinite while loop (with break)

```
while True:
    name = input("Enter your name (type 'exit' to stop): ")
    if name == "exit":
        break
    print("Hello", name)
```

---

## 8. Common Mistakes

- Forgetting to update the loop variable in a `while` loop.
  - Not using `:` after the loop condition.
  - Improper indentation.
- 

## Summary

- `while` loops are condition-based, `for` loops are sequence-based.
- `break` exits the loop, `continue` skips one iteration, `pass` does nothing.
- `range()` is useful for creating number sequences.

- Python allows iterating through strings, lists, and more.

---

## Homework / Self-Practice

1. Write a `while` loop to print numbers from 10 to 1.
  2. Create a `for` loop to print the multiplication table of any number entered by the user.
  3. Write a program to count how many vowels are in a given string.
  4. Print all odd numbers between 1 to 50 using a `for` loop.
- 
- 

## CS306 – Lecture 6: Loops in Python (Part 2) – Nested Loops and Patterns

---

### Lecture Objectives

- Understand **nested loops** in Python.
  - Learn how to construct **pattern-based outputs** using loops.
  - Practice logical thinking through loop exercises.
  - Reinforce the difference between `for` and `while` loops.
- 

### 1. Nested Loops

A **nested loop** is a loop inside another loop. Useful for:

- Printing patterns
- Working with 2D structures like matrices

#### Syntax

```
for i in range(n):           # Outer loop
    for j in range(m):       # Inner loop
        # Block of code
```

---

### 2. Simple Nested Loop Example

Example: Print a 3x3 grid of stars

```
for i in range(3):
    for j in range(3):
        print("*", end=" ")
    print()
```

Output:

```
* * *  
* * *  
* * *
```

- `end=" "` keeps printing on the same line.
  - `print()` after inner loop moves to the next line.
- 

### 3. Number Patterns

Example 1: Increasing Number Triangle

```
for i in range(1, 6):  
    for j in range(1, i + 1):  
        print(j, end=" ")  
    print()
```

Output:

```
1  
1 2  
1 2 3  
1 2 3 4  
1 2 3 4 5
```

---

Example 2: Star Triangle

```
for i in range(1, 6):  
    print(" *" * i)
```

Output:

```
*  
**  
***  
****  
*****
```

---

### 4. Logical Thinking with Loops

Nested loops are useful for problems involving rows and columns, such as:

- Grids
  - Tables
  - Matrix operations
  - Game boards
- 

### 5. Using Nested `while` Loops

Example: Print 3x3 number grid

```
i = 1
```

```
while i <= 3:
    j = 1
    while j <= 3:
        print(j, end=" ")
        j += 1
    print()
    i += 1
```

---

## 6. Multiplication Table (Using Nested Loops)

```
for i in range(1, 11):
    for j in range(1, 11):
        print(i * j, end="\t")
    print()
```

Output: A 10x10 multiplication table

---

## 7. Pattern Practice Examples

Example: Inverted Star Triangle

```
for i in range(5, 0, -1):
    print(" " * i)
```

Output:

```
*****
****
***
**
*
```

---

Example: Right-aligned Star Triangle

```
for i in range(1, 6):
    print(" " * (5 - i) + "*" * i)
```

Output:

```

*
 **
***
****
*****
```

---

## 8. Tips for Pattern Printing

- **Use outer loop** to control number of lines.
  - **Use inner loop** to control characters in each line.
  - Combine " " or numbers with spaces to format correctly.
  - Use end=" " or end="" to control line breaks.
-

## 9. Common Mistakes

- Confusing inner and outer loop variables.
  - Misusing `end=" "` and `print()`.
  - Forgetting to increment counters in `while` loops.
  - Improper indentation in nested loops.
- 

## Summary

- Nested loops allow repeated actions within repeated actions.
  - Pattern printing improves logical flow and nested loop mastery.
  - Loops can be used for both simple and advanced outputs.
- 

## Homework / Self-Practice

1. Write a program to print the following pattern:  
2. 1  
3. 2 2  
4. 3 3 3  
5. 4 4 4 4
  6. Print a hollow square using nested loops.
  7. Create a multiplication table from 1 to 20 using nested loops.
  8. Use nested `while` loops to print the pattern:  
9. \* \* \* \*  
10. \* \*  
11. \* \*  
12. \* \* \* \*
- 
- 

## CS306 – Lecture 7: Functions in Python (Part 1)

---

### Lecture Objectives

- Understand the concept of **functions** in Python.
  - Learn how to define and call functions.
  - Understand the use of **parameters** and **arguments**.
  - Explore **return values**.
  - Learn the benefits of using functions.
-

## 1. What is a Function?

A **function** is a block of organized, reusable code that performs a specific task.

Functions help to:

- Avoid repetition (code reusability)
  - Break a large program into smaller manageable parts
  - Improve clarity and debugging
- 

## 2. Defining a Function

Syntax

```
def function_name(parameters):  
    # block of code
```

Example

```
def greet():  
    print("Hello, Python!")
```

This defines the function, but **does not call it yet.**

---

## 3. Calling a Function

You **call** a function to execute the code inside it.

```
greet() # Output: Hello, Python!
```

---

## 4. Function with Parameters

Parameters are values passed **into** a function.

```
def greet(name):  
    print("Hello, ", name)
```

```
greet("Ali")
```

Output:

```
Hello, Ali
```

- "name" is a **parameter** in the function definition.
  - "Ali" is an **argument** passed during the function call.
-

## 5. Function with Multiple Parameters

```
def add(a, b):  
    print("Sum is", a + b)
```

```
add(5, 3)
```

Output:

```
Sum is 8
```

---

## 6. The `return` Statement

Functions can return a value using the `return` statement.

```
def square(x):  
    return x * x  
  
result = square(4)  
print(result)
```

Output:

```
16
```

- `return` sends a result **back to the caller**.
  - The result can be stored in a variable.
- 

## 7. Function vs Procedure

|           | Term | Returns Value? | Use Case                  |
|-----------|------|----------------|---------------------------|
| Function  | Yes  |                | When a result is expected |
| Procedure | No   |                | Just performs an action   |

**Note:** In Python, all functions are called "functions" — there's no separate keyword for "procedures". 

---

## 8. Why Use Functions?

- **Modularity:** Breaks program into smaller parts.
  - **Reusability:** Write once, use many times.
  - **Readability:** Clean and understandable code.
  - **Debugging:** Easy to test small pieces.
-

## 9. Practical Examples

Example 1: Find maximum of two numbers

```
def maximum(a, b):  
    if a > b:  
        return a  
    else:  
        return b
```

```
print(maximum(10, 20)) # Output: 20
```

Example 2: Calculate factorial

```
def factorial(n):  
    result = 1  
    for i in range(1, n+1):  
        result *= i  
    return result
```

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

---

## 10. Function Naming Rules

- Must begin with a letter or underscore
  - Cannot start with a number
  - Can include letters, numbers, underscores
  - Avoid using Python keywords
- 

## 11. Common Mistakes

- Forgetting parentheses when calling functions: `greet` vs `greet()`
  - Mismatched number of arguments
  - Not using `return` when a value is expected
  - Mixing up parameters and arguments
- 

## Summary

- A **function** is defined with `def` and called using its name with parentheses.
  - You can pass **parameters** and return **values**.
  - Functions help write **cleaner**, **reusable**, and **organized** code.
- 

## Homework / Practice Tasks

1. Define a function that takes a number and prints whether it is even or odd.

2. Write a function to calculate the area of a circle ( $\pi r^2$ ).
3. Create a function to convert Celsius to Fahrenheit.
4. Make a function that takes a name and age and prints a greeting message.

---

## CS306 – Lecture 8: Functions in Python (Part 2)

*Default Arguments, Keyword Arguments, Variable-Length Arguments*

---

### Lecture Objectives

- Understand **default arguments** in functions.
- Learn how to use **keyword arguments**.
- Explore **variable-length arguments** (\*args and \*\*kwargs).
- Reinforce flexibility in function definitions.

---

### 1. Default Arguments

Default arguments allow us to assign a **default value** to a parameter in case no value is provided during the function call.

Syntax:

```
def function_name(parameter=default_value):  
    # code block
```

Example:

```
def greet(name="Guest"):  
    print("Hello, ", name)
```

```
greet()           # Output: Hello, Guest  
greet("Ayesha")  # Output: Hello, Ayesha
```

Default arguments make parameters **optional**.

---

### 2. Rules for Default Arguments

- Default arguments must come **after** non-default ones.

 Valid:

```
def func(a, b=10): ...
```

✗ Invalid:

```
def func(a=10, b): ... # ✗ SyntaxError
```

---

### 🔑 3. Keyword Arguments

Keyword arguments are used when calling a function by **specifying parameter names**.

Example:

```
def student(name, age):  
    print("Name:", name)  
    print("Age:", age)
```

```
student(age=20, name="Ahmed")
```

Output:

```
Name: Ahmed  
Age: 20
```

Order does **not matter** when using keyword arguments.

---

### ✳️ 4. Variable-Length Arguments

Sometimes we don't know how many arguments will be passed. Python allows:

- `*args` → multiple **positional** arguments
  - `**kwargs` → multiple **keyword** arguments
- 

#### 📦 `*args`: Multiple Positional Arguments

Example:

```
def total(*numbers):  
    sum = 0  
    for num in numbers:  
        sum += num  
    print("Total:", sum)
```

```
total(5, 10, 15) # Output: Total: 30
```

`*args` collects extra positional arguments as a **tuple**.

---

### ◆ **\*\*kwargs: Multiple Keyword Arguments**

Example:

```
def show_details(**info):  
    for key, value in info.items():  
        print(key, ":", value)  
  
show_details(name="Sara", age=22, city="Lahore")
```

Output:

```
name : Sara  
age : 22  
city : Lahore
```

**\*\*kwargs** collects extra keyword arguments as a **dictionary**.

---

### 🔄 5. Combining All Types

Example:

```
def example(a, b=10, *args, **kwargs):  
    print("a =", a)  
    print("b =", b)  
    print("args =", args)  
    print("kwargs =", kwargs)  
  
example(1, 2, 3, 4, name="Ali", age=25)
```

Output:

```
a = 1  
b = 2  
args = (3, 4)  
kwargs = {'name': 'Ali', 'age': 25}
```

---

### 🌐 6. Benefits of Using These Features

- Create **more flexible** functions.
  - Handle **optional** and **unknown** inputs.
  - Build **reusable** utility functions.
- 

### ⚠️ 7. Common Mistakes

- Misplacing the order of arguments.
  - Using **\*args** or **\*\*kwargs** incorrectly (e.g., forgetting the **\*** or **\*\***).
  - Assigning default arguments before required ones.
-

## ✓ 8. Recap Table

| Type                  | Usage Example                   | Collected As |
|-----------------------|---------------------------------|--------------|
| Positional            | <code>func(1, 2)</code>         | As values    |
| Keyword               | <code>func(x=1, y=2)</code>     | As keys      |
| Default               | <code>def func(x=10)</code>     | Optional     |
| <code>*args</code>    | <code>def func(*args)</code>    | Tuple        |
| <code>**kwargs</code> | <code>def func(**kwargs)</code> | Dictionary   |

---

## Homework / Practice

1. Write a function that accepts variable numbers of arguments and returns their average.
  2. Define a function with default arguments for name and city.
  3. Use `**kwargs` to accept any student information and print it line by line.
  4. Combine `*args` and `**kwargs` in a single function and print the data.
- 
- 

## CS306 – Lecture 9: Recursion in Python

---

### Lecture Objectives

- Understand the concept of **recursion** in programming.
  - Learn how to write **recursive functions** in Python.
  - Study **base case** and **recursive case**.
  - Understand the importance of recursion and its application in solving problems.
- 

### 1. What is Recursion?

Recursion is a technique where a function **calls itself** in order to solve a problem. The problem is divided into smaller subproblems, and the function calls itself to solve them.

Key Components of a Recursive Function:

1. **Base case:** The condition that stops the recursion.
2. **Recursive case:** The part where the function calls itself.

## 2. Basic Structure of Recursion

General Syntax:

```
def recursive_function(parameters):  
    if base_case_condition:  
        return result # This is the base case.  
    else:  
        return recursive_function(smaller_problem) # Recursive call
```

---

## 3. Example: Factorial Function (Using Recursion)

The **factorial** of a number  $n$  is the product of all positive integers less than or equal to  $n$ :

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1 \quad n! = n \times (n-1) \times (n-2) \times \dots \times 1$$

Recursive Factorial Function:

```
def factorial(n):  
    if n == 1:  
        return 1 # Base case  
    else:  
        return n * factorial(n-1) # Recursive call
```

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

Explanation:

- **Base case:** If  $n == 1$ , return 1.
  - **Recursive case:** Multiply  $n$  by  $\text{factorial}(n-1)$ .
- 

## 4. Example: Fibonacci Sequence (Using Recursion)

The **Fibonacci sequence** is a series of numbers where each number is the sum of the two preceding ones. The sequence starts with 0 and 1:

$$F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2) \quad F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$$

Recursive Fibonacci Function:

```
def fibonacci(n):  
    if n <= 1:  
        return n # Base case  
    else:  
        return fibonacci(n-1) + fibonacci(n-2) # Recursive call
```

```
print(fibonacci(6)) # Output: 8
```

Explanation:

- **Base case:** If  $n \leq 1$ , return  $n$ .
  - **Recursive case:** Return the sum of `fibonacci(n-1)` and `fibonacci(n-2)`.
- 

## 5. Importance of Base Case

Without a **base case**, the recursion will continue infinitely, resulting in a **stack overflow error**. Always ensure that:

- A base case is defined.
  - The recursive calls move closer to the base case.
- 

## 6. Tracing Recursion

When tracing recursive calls, it's important to observe the following:

1. How the function call stack grows.
2. The base case condition.
3. How the function unwinds (returns back) after reaching the base case.

Example: Tracing `factorial(3)`

```
factorial(3)  # Calls factorial(2)
factorial(2)  # Calls factorial(1)
factorial(1)  # Returns 1
```

---

## 7. Recursive Function Efficiency

Recursion is elegant and easy to understand but can lead to inefficiency for large inputs. **Memoization** and **iterative solutions** are often preferred for optimizing recursive functions.

- **Memoization:** Store the results of subproblems to avoid redundant calculations.
  - **Iterative Solutions:** Replace recursion with loops for efficiency.
- 

## 8. Recursive vs Iterative Solutions

Example: Factorial (Iterative vs Recursive)

- **Iterative solution:**

```
def factorial_iterative(n):  
    result = 1  
    for i in range(1, n + 1):  
        result *= i  
    return result
```

- **Recursive solution:**

```
def factorial_recursive(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial_recursive(n-1)
```

---

## 9. Applications of Recursion

1. **Sorting Algorithms:** QuickSort, MergeSort, and other divide-and-conquer algorithms.
  2. **Tree and Graph Traversal:** In-order, pre-order, post-order traversal of trees.
  3. **Backtracking Algorithms:** Solving puzzles like Sudoku, N-Queens problem.
  4. **Mathematical Problems:** Factorial, Fibonacci, and other series.
- 

## 10. Common Recursion Mistakes

- Forgetting the **base case** (leading to infinite recursion).
  - Passing incorrect parameters that lead to incorrect results.
  - Using recursion unnecessarily when an iterative solution would be more efficient.
- 

## 11. Recursion Example: Sum of Natural Numbers

To calculate the sum of the first  $n$  natural numbers:

$$\text{Sum}(n) = n + \text{Sum}(n-1) \quad \text{Sum}(n) = n + \text{Sum}(n-1)$$

Recursive Function:

```
def sum_of_numbers(n):  
    if n == 1:  
        return 1  
    else:  
        return n + sum_of_numbers(n-1)  
  
print(sum_of_numbers(5))    # Output: 15
```

---

## Summary

- **Recursion** is a powerful programming technique where a function calls itself.
  - It consists of a **base case** and a **recursive case**.
  - Use recursion when the problem can be broken down into smaller subproblems.
  - Always ensure the recursion reaches a **base case** to avoid infinite loops.
- 

## Homework / Practice

1. Write a recursive function to find the power of a number ( $x^n$ ).
  2. Implement a recursive solution for finding the greatest common divisor (GCD) of two numbers.
  3. Create a recursive function to reverse a string.
  4. Solve the **Tower of Hanoi** problem using recursion.
- 
- 

## CS306 – Lecture 10: File Handling in Python

---

### Lecture Objectives

- Understand the concept of **file handling** in Python.
  - Learn how to **open**, **read**, and **write** files in Python.
  - Explore different **file modes** for file handling.
  - Handle **file exceptions** and ensure proper file closure.
- 

### 1. Introduction to File Handling

File handling in Python allows you to interact with files—read from them, write to them, and manipulate file contents. Files are categorized as either **text files** or **binary files**.

---

### 2. Opening Files in Python

Before performing any operations on a file (like reading or writing), you need to **open** the file using the `open()` function.

Syntax:

```
file_object = open("filename", "mode")
```

File Modes:

- **"r"**: Read (default mode). Opens the file for reading only.
- **"w"**: Write. Opens the file for writing (overwrites the file if it exists).
- **"a"**: Append. Opens the file for appending data.
- **"b"**: Binary mode (used for non-text files like images).
- **"x"**: Exclusive creation. Creates a new file but raises an error if the file already exists.

Example:

```
file = open("example.txt", "r") # Opens file in read mode
```

---

### 3. Reading from Files

Python provides several methods to read data from a file:

`read()`:

Reads the entire content of the file.

```
file = open("example.txt", "r")
content = file.read() # Reads the entire file
print(content)
file.close() # Always close the file after operations
readline():
```

Reads one line at a time.

```
file = open("example.txt", "r")
line = file.readline() # Reads one line
print(line)
file.close()
readlines():
```

Reads all the lines into a **list**.

```
file = open("example.txt", "r")
lines = file.readlines() # Returns a list of all lines
print(lines)
file.close()
```

---

### 4. Writing to Files

To write to a file, open it in **write ("w")** or **append ("a")** mode.

```
write():
```

Writes the string to the file (overwrites the file if it exists).

```
file = open("example.txt", "w")
file.write("Hello, World!\nThis is a test.")
file.close()

writelines():
```

Writes a list of strings to a file.

```
file = open("example.txt", "w")
lines = ["First line\n", "Second line\n"]
file.writelines(lines)
file.close()
```

---

## 5. Closing Files

It's crucial to always **close** a file after performing operations to free up system resources. 

Syntax:

```
file.close()
```

---

## 6. File Exceptions

When handling files, it's important to consider errors such as:

- The file not existing.
- Permission errors.
- File handling exceptions.

Python provides **exception handling** to deal with such issues using `try`, `except`, and `finally`.

Example:

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("The file does not exist!")
finally:
    file.close()
```

---

## 7. Working with File Paths

When opening a file, you can specify a **relative path** (from the current directory) or an **absolute path** (full path from the root).

Example:

```
file = open("/home/user/example.txt", "r") # Absolute path
```

---

## 8. File Operations Summary

| Mode | Description                                         | Example                             |
|------|-----------------------------------------------------|-------------------------------------|
| "r"  | Read (default)                                      | <code>open("file.txt", "r")</code>  |
| "w"  | Write (overwrites)                                  | <code>open("file.txt", "w")</code>  |
| "a"  | Append (adds to the end)                            | <code>open("file.txt", "a")</code>  |
| "b"  | Binary (used for non-text files)                    | <code>open("file.jpg", "rb")</code> |
| "x"  | Create a new file (raises error if the file exists) | <code>open("file.txt", "x")</code>  |

---

## 9. Context Manager (with Statement)

Using `with` statement helps ensure that the file is properly closed after the operations, even if an error occurs.

Example:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# No need to call file.close(), it's automatically handled
```

---

## 10. Working with Binary Files

Binary files (e.g., images, audio files) require reading and writing in binary mode ("b").

Example: Reading a Binary File

```
with open("image.jpg", "rb") as file:
    content = file.read()
    print(content)
```

Example: Writing to a Binary File

```
with open("output.jpg", "wb") as file:
    file.write(content)
```

---

## Summary

- Use `open()` to work with files in Python.
  - Files can be opened in different modes ("`r`", "`w`", "`a`", etc.).
  - Always close files after operations, or use the `with` statement to ensure automatic closure.
  - Handle file exceptions using `try` and `except` blocks.
- 

## Homework / Practice

1. Write a program to count the number of words in a text file.
  2. Write a function to append new data to an existing file.
  3. Create a program that reads a file and prints each line in reverse order.
  4. Write a program that reads a binary file (e.g., an image) and saves it to another location.
- 
- 

## CS306 – Lecture 11: Python Libraries

---

### Lecture Objectives

- Understand what **libraries** are in Python.
  - Learn how to **import** and use external libraries in Python.
  - Explore some commonly used **standard libraries** in Python.
  - Learn about third-party libraries and how to install them using **pip**.
- 

### 1. Introduction to Python Libraries

In Python, a **library** is a collection of pre-written code that provides functionality for various tasks like file I/O, math operations, system operations, and web development. Libraries help avoid writing repetitive code and make programming faster and more efficient.

Two types of libraries:

1. **Standard Libraries:** These are built-in libraries that come with Python.
  2. **Third-Party Libraries:** Libraries developed by the community that can be installed via package managers like `pip`.
-

## 2. Importing Libraries

To use a library or module in Python, we need to import it.

Basic Syntax for Importing:

```
import library_name # Import the entire library
```

Example:

```
import math # Import the math library
print(math.sqrt(16)) # Output: 4.0
```

You can also import specific functions from a module:

```
from math import sqrt # Import only sqrt from the math library
print(sqrt(16)) # Output: 4.0
```

Or, you can import the library with an alias:

```
import math as m # Alias for math library
print(m.sqrt(16)) # Output: 4.0
```

---

## 3. Standard Libraries in Python

Python provides a wide range of **standard libraries** for various tasks. Some common ones include:

### 1. math:

Provides mathematical functions like `sqrt()`, `pow()`, `sin()`, etc.

Example:

```
import math
print(math.pi) # Output: 3.141592653589793
```

### 2. random:

Used for generating random numbers, choosing random elements from a list, etc.

Example:

```
import random
print(random.randint(1, 10)) # Generates a random integer between 1 and 10
```

### 3. time:

Used for working with time-related tasks, such as delays or formatting timestamps.

Example:

```
import time
print(time.time()) # Returns the current time in seconds since the epoch
```

#### 4. os:

Provides functions to interact with the operating system, such as file manipulation and process management.

Example:

```
import os
print(os.getcwd()) # Returns the current working directory
```

#### 5. sys:

Provides access to system-specific parameters and functions. Used for interacting with command-line arguments.

Example:

```
import sys
print(sys.version) # Prints the version of Python
```

#### 6. json:

Used for working with JSON (JavaScript Object Notation) data.

Example:

```
import json
data = '{"name": "John", "age": 30}'
parsed_data = json.loads(data) # Converts string to Python dictionary
print(parsed_data)
```

---

## 4. Third-Party Libraries

In addition to the standard libraries, Python also supports a vast collection of **third-party libraries** developed by the community. These libraries can be installed using Python's package manager `pip`.

Installing Third-Party Libraries with `pip`:

### 1. Installing a Library:

- o Run the following command in the terminal:

2. `pip install library_name`

3. **Example:** Installing the **requests** library (for making HTTP requests).

4. `pip install requests`

### 5. Using an Installed Library:

After installation, you can import and use the library in your Python code:

6. `import requests`

```
7. response = requests.get("https://api.github.com")
8. print(response.status_code)
```

Popular Third-Party Libraries:

- **NumPy**: For numerical computations.
- **Pandas**: For data manipulation and analysis.
- **Matplotlib**: For data visualization.
- **Django**: A web framework for building web applications.
- **Flask**: A lightweight web framework.
- **BeautifulSoup**: For web scraping and parsing HTML/XML.
- **TensorFlow**: For machine learning and neural networks.

---

## 5. Exploring and Managing Libraries

Checking Installed Libraries:

You can check the list of installed libraries using:

```
pip list
```

Uninstalling a Library:

To uninstall a library, use:

```
pip uninstall library_name
```

---

## 6. Virtual Environments

In Python, virtual environments allow you to create isolated environments for your projects. This helps manage dependencies and avoid conflicts between libraries in different projects.

Creating a Virtual Environment:

1. Install the `virtualenv` package:
2. `pip install virtualenv`
3. Create a new virtual environment:
4. `virtualenv myenv`
5. Activate the virtual environment:
  - On Windows:  
`myenv\Scripts\activate`
  - On macOS/Linux:  
`source myenv/bin/activate`
6. To deactivate the virtual environment:
7. `deactivate`

## 7. Using Documentation

Most libraries come with documentation that explains how to use their functions. You can access the official Python documentation here: [Python Standard Library](#).

To get information about a module or function in Python, use the `help()` function.

Example:

```
help(math)
help(math.sqrt)
```

---

## 8. Summary

- Python libraries allow you to extend Python's functionality without writing everything from scratch.
  - Use `import` to bring in libraries into your code.
  - Python's standard library includes modules for handling file I/O, system tasks, mathematics, and more.
  - Third-party libraries can be installed via `pip`, and popular libraries include `requests`, `numpy`, and `pandas`.
  - Use **virtual environments** to manage project dependencies effectively.
- 

## Homework / Practice

1. Install and use the **requests** library to fetch data from an API.
  2. Write a Python program that uses the **os** library to list all files in a directory.
  3. Use the **random** library to simulate rolling a dice.
  4. Create a Python program that reads a JSON file using the **json** library and extracts specific data.
- 
- 
- 

## CS306 – Lecture 12: Object-Oriented Programming (OOP) in Python

---

### Lecture Objectives

- Understand the concept of **Object-Oriented Programming (OOP)**.

- Learn about the **four main principles** of OOP: Encapsulation, Abstraction, Inheritance, and Polymorphism.
  - Understand how to implement these principles in Python using **classes** and **objects**.
  - Learn about **methods** and **attributes** in Python.
- 

## 1. Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects** and **classes**. OOP makes it easier to create modular, reusable, and maintainable code.

Key Concepts of OOP:

- **Classes**: Blueprints for creating objects.
  - **Objects**: Instances of a class.
  - **Attributes**: Variables that belong to a class or object.
  - **Methods**: Functions that belong to a class or object.
- 

## 2. Defining Classes and Objects

Classes

A class is a blueprint for creating objects. It defines the attributes and methods common to all objects of that class.

Syntax to Define a Class:

```
class ClassName:
    # constructor (__init__ method)
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    # method
    def method_name(self):
        pass
```

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

## Objects

An object is an instance of a class. When you create an object, Python calls the class constructor (`__init__` method) to initialize the object's attributes.

Creating an Object:

```
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
```

Now, you can access the attributes and methods of the objects:

```
print(person1.name) # Output: Alice
person2.greet()    # Output: Hello, my name is Bob and I am 25 years old.
```

---

## 3. Four Principles of OOP

### 1. Encapsulation

Encapsulation is the concept of **bundling** data (attributes) and methods that operate on the data into a single unit (class). It also involves controlling access to the data by restricting the use of the data outside the class.

*Example:*

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.__speed = 0 # Private attribute

    def accelerate(self):
        self.__speed += 5

    def get_speed(self):
        return self.__speed

    def set_speed(self, speed):
        if speed >= 0:
            self.__speed = speed
        else:
            print("Speed cannot be negative.")
```

Here, `__speed` is a private attribute, and we use methods to interact with it (`get_speed()` and `set_speed()`).

## 2. Abstraction

Abstraction is the concept of hiding the **complexity** of the system and exposing only the **essential features** to the user. This is done by defining **abstract classes** or using **abstract methods** that only provide a blueprint for other classes. 

In Python, abstraction is achieved through **abstract classes** using the `abc` module.

*Example:*

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        print("Woof!")
```

In this example, `Animal` is an abstract class with an abstract method `sound()`, which is implemented by the `Dog` class.

## 3. Inheritance

Inheritance allows a class to inherit attributes and methods from another class. This promotes **reusability** and allows the creation of **hierarchical relationships** between classes.

*Example:*

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")
```

Now, the `Dog` class inherits the `Animal` class and overrides the `speak()` method:

```
dog = Dog("Buddy")
dog.speak() # Output: Buddy barks.
```

## 4. Polymorphism

Polymorphism allows different classes to provide a **shared interface**, but the implementation can vary. It enables you to use a **single method** to represent different behaviors based on the object type.

*Example:*

```
class Cat(Animal):
    def speak(self):
        print(f"{self.name} meows.")

def make_sound(animal):
    animal.speak()

dog = Dog("Rex")
cat = Cat("Whiskers")

make_sound(dog) # Output: Rex barks.
make_sound(cat) # Output: Whiskers meows.
```

In this example, `make_sound()` uses polymorphism to call the `speak()` method, which behaves differently depending on whether the object is a `Dog` or `Cat`.

---

## 4. Methods in Python Classes

### Instance Methods

Instance methods are functions that operate on the attributes of an object. They are defined within a class and take the `self` parameter.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

circle1 = Circle(5)
print(circle1.area()) # Output: 78.5
```

### Class Methods

Class methods operate on the class itself rather than instances of the class. They are defined using the `@classmethod` decorator and take the `cls` parameter.

```
class Person:
    count = 0

    def __init__(self, name):
        self.name = name
        Person.count += 1

    @classmethod
    def get_count(cls):
        return cls.count

p1 = Person("Alice")
p2 = Person("Bob")
```

```
print(Person.get_count()) # Output: 2
```

### Static Methods

Static methods do not operate on the instance or the class. They are defined using the `@staticmethod` decorator and don't take `self` or `cls` parameters.

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

print(MathOperations.add(3, 5)) # Output: 8
```

---

## 5. Summary

- **Object-Oriented Programming (OOP)** organizes code around classes and objects.
- The four main principles of OOP are:
  - **Encapsulation**: Bundling data and methods together and restricting access to certain data.
  - **Abstraction**: Hiding implementation details and exposing only essential features. 
  - **Inheritance**: Reusing attributes and methods from another class.
  - **Polymorphism**: Using a shared interface with different implementations.
- Methods in Python classes can be **instance methods**, **class methods**, or **static methods**, each serving different purposes.

---

## Homework / Practice

1. Create a class `BankAccount` with attributes `account_number` and `balance`. Implement methods to deposit, withdraw, and check balance.
2. Create a `Shape` class with an abstract method `area()`. Implement classes for `Rectangle` and `Circle` that inherit from `Shape` and override the `area()` method.
3. Write a program to implement polymorphism using a base class `Vehicle` and derived classes `Car` and `Bike`.

---

## CS306 – Lecture 13: File Handling in Python

### Lecture Objectives

- Understand the concept of **file handling** in Python.

- Learn how to open, read, write, and close files in Python.
- Explore various modes for file operations in Python.
- Learn about context managers for safe file handling.

---

## 1. Introduction to File Handling

File handling in Python allows you to read from and write to files. This is essential for working with data stored in files like text files, CSVs, and log files. Python provides several built-in functions for handling files efficiently. 

Common File Operations:

- **Opening** a file.
- **Reading** from a file.
- **Writing** to a file.
- **Closing** a file.

---

## 2. Opening a File

Before performing any operation on a file, you must **open** it using Python's built-in `open()` function. This function returns a file object, which provides methods and attributes to perform file operations.

Syntax:

```
file_object = open("filename", "mode")
```

Parameters:

- "filename": The name of the file to be opened (can include the file path).
- "mode": The mode in which the file is opened (read, write, append, etc.).

Modes of File Operations:

- 'r': Read (default mode). Opens the file for reading.
- 'w': Write. Opens the file for writing. Creates a new file or truncates an existing file.
- 'a': Append. Opens the file for writing. Data is added at the end.
- 'b': Binary mode (used with other modes like 'rb' or 'wb' for binary files).
- 'x': Exclusive creation. Creates a new file and returns an error if the file exists.
- 't': Text mode (default mode).

Example:

```
file = open("example.txt", "r")
```

---

### 3. Reading from a File

Once a file is opened in read mode ('r'), you can use different methods to read its contents.

Methods to Read a File:

1. **read()**: Reads the entire file content.
2. `file_content = file.read()`
3. `print(file_content)`
4. **readline()**: Reads one line from the file at a time.
5. `line = file.readline()`
6. `print(line)`
7. **readlines()**: Reads all the lines in a file and returns a list of lines.
8. `lines = file.readlines()`
9. `print(lines)`

Example: Reading a File Line by Line

```
file = open("example.txt", "r")
for line in file:
    print(line, end="") # end="" prevents extra newline
file.close()
```

---

### 4. Writing to a File

To write to a file, you must open the file in write ('w'), append ('a'), or exclusive creation ('x') mode. If the file doesn't exist, Python will create it.

Writing Text to a File:

1. **write()**: Writes a string to the file.
  2. `file = open("example.txt", "w")`
  3. `file.write("Hello, world!")`
  4. `file.close()`
  5. **writelines()**: Writes a list of strings to the file.
  6. `lines = ["First line\n", "Second line\n", "Third line\n"]`
  7. `file = open("example.txt", "w")`
  8. `file.writelines(lines)`
  9. `file.close()`
- 

### 5. Closing a File

It is important to **close** a file after performing operations to ensure data is saved and resources are released.

Syntax:

```
file.close()
```

Failure to close a file can lead to memory leaks or corrupted data.

---

## 6. Using Context Manager (with Statement)

Python provides a convenient way to handle file operations using a **context manager**. This ensures that the file is properly closed after its operations are completed, even if an exception occurs.

Syntax:

```
with open("filename", "mode") as file:
    # File operations go here
    content = file.read()
```

The context manager automatically handles the opening and closing of the file.

Example:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# No need to manually close the file
```

---

## 7. File Handling Example

Let's combine the concepts we've learned so far in a practical example. In this example, we will read from one file, modify its contents, and then write the updated content to a new file.

Example:

```
# Reading from the file
with open("source.txt", "r") as file:
    lines = file.readlines()

# Modifying the content
lines.append("This is a new line.\n")

# Writing to a new file
with open("destination.txt", "w") as file:
    file.writelines(lines)
```

---

## 8. File Handling with Binary Files

In addition to handling text files, Python also supports **binary file operations**. Binary files store data in a format that is not human-readable (e.g., images, videos).

To open a binary file, use the 'b' mode in conjunction with other modes like 'rb' or 'wb'.

Example (Reading a Binary File):

```
with open("image.jpg", "rb") as file:
    data = file.read()
    print(data)
```

Example (Writing to a Binary File):

```
with open("new_image.jpg", "wb") as file:
    file.write(data)
```

---

## 9. Summary

- File handling in Python involves opening, reading, writing, and closing files.
  - You can open files in various modes, such as 'r', 'w', and 'a'.
  - Use `read()`, `readline()`, and `readlines()` to read from a file.
  - Use `write()` and `writelines()` to write to a file.
  - Always close a file after operations or use the `with` statement for automatic file management.
  - Binary file handling is done using 'rb' and 'wb' modes for reading and writing binary data.
- 

## Homework / Practice

1. Write a Python program that reads a file, counts the number of words, and then writes the word count to a new file.
  2. Create a Python program that reads a CSV file and displays its content.
  3. Write a program that reads a text file and creates a new file containing the reversed content of the original file.
  4. Write a program to append new text to an existing file without overwriting its content.
- 
- 

## CS306 – Lecture 14: Exception Handling in Python

---

### Lecture Objectives

- Understand what exceptions are and why they occur.
  - Learn how to handle exceptions in Python using `try`, `except`, `else`, and `finally`.
  - Explore common built-in exceptions and how to raise custom exceptions.
  - Understand the importance of exception handling in making Python programs more robust.
-

## 1. Introduction to Exceptions

An **exception** is an event that disrupts the normal flow of a program. When Python encounters an error, it raises an exception to indicate that something went wrong.

Types of Errors in Python:

1. **Syntax Errors:** Occur when Python can't interpret the code (e.g., a typo, missing colon).
2. **Runtime Errors:** Occur during the execution of a program (e.g., dividing by zero, accessing a non-existent file).
3. **Logical Errors:** Occur when the program runs but doesn't do what the programmer intended.

---

## 2. Exception Handling in Python

Python uses the `try`, `except`, `else`, and `finally` blocks to handle exceptions. The goal of exception handling is to **gracefully** handle errors and allow the program to continue executing without crashing.

Syntax of Exception Handling:

```
try:
    # Code that may raise an exception
    risky_code()
except ExceptionType:
    # Code to handle the exception
    handle_exception()
else:
    # Code to execute if no exception occurs
    successful_execution()
finally:
    # Code that will always execute, regardless of exception
    cleanup()
```

Components:

- **try block:** Contains code that might raise an exception.
- **except block:** Catches the exception and handles it.
- **else block:** Executes if no exception occurs in the `try` block.
- **finally block:** Always executes, regardless of whether an exception occurred or not.

---

## 3. try and except Blocks

- **try:** Code that may raise an exception is placed inside the `try` block.
- **except:** Code inside the `except` block executes when the specified exception occurs.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input. Please enter an integer.")
```

In this example:

- The `ZeroDivisionError` will be raised if the user enters 0 as input.
- The `ValueError` will be raised if the input is not an integer.

---

## 4. Handling Multiple Exceptions

Python allows multiple `except` blocks to handle different exceptions separately.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input. Please enter an integer.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

Here, the `Exception` block catches any other errors that were not explicitly handled.

---

## 5. The `else` Block

The `else` block executes if no exception occurs in the `try` block. This is useful for code that should run only when the `try` block is successful.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Invalid input. Please enter an integer.")
else:
    print(f"Result: {result}")
```

In this example:

- If no exception occurs, the result is printed in the `else` block.

---

## 6. The `finally` Block

The `finally` block always executes, whether an exception is raised or not. It is useful for cleaning up resources, such as closing files or releasing database connections.

Example:

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    file.close()
    print("File has been closed.")
```

Here, the `finally` block ensures that the file is closed, even if an exception occurs.

---

## 7. Common Built-in Exceptions

Python has several built-in exceptions that can be used in exception handling:

- **`ZeroDivisionError`**: Raised when attempting to divide by zero.
- **`FileNotFoundError`**: Raised when trying to open a file that doesn't exist.
- **`ValueError`**: Raised when a function receives an argument of the correct type but inappropriate value.
- **`IndexError`**: Raised when trying to access an index that is out of range in a list or tuple.
- **`KeyError`**: Raised when trying to access a key that does not exist in a dictionary.

---

## 8. Raising Exceptions

Sometimes, you may want to raise your own exceptions. This can be done using the `raise` keyword.

Syntax:

```
raise ExceptionType("Error message")
```

You can raise built-in exceptions or define your own custom exceptions.

Example:

```
x = -1
if x < 0:
    raise ValueError("Value cannot be negative!")
```

In this example, a `ValueError` is raised if `x` is less than 0.

---

## 9. Custom Exceptions

You can define your own exceptions by inheriting from the built-in `Exception` class.

Example:

```
class NegativeValueError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Using the custom exception
x = -5
if x < 0:
    raise NegativeValueError("Negative values are not allowed!")
```

In this example, a custom exception `NegativeValueError` is created, and it is raised when `x` is negative.

---

## 10. Best Practices for Exception Handling

- Only catch exceptions you can handle. Don't use a generic `except` unless necessary.
  - Use the `finally` block for cleanup operations.
  - Don't use exceptions for control flow (i.e., don't rely on exceptions as part of the normal program logic).
  - Raise meaningful custom exceptions with clear error messages.
- 

## 11. Summary

- **Exceptions** are errors that disrupt the normal flow of a program.
- You can handle exceptions in Python using `try`, `except`, `else`, and `finally` blocks.
- The `else` block is executed if no exception occurs in the `try` block.
- The `finally` block always runs, regardless of whether an exception was raised or not.
- You can raise custom exceptions using the `raise` keyword.
- Exception handling helps make programs more robust by preventing crashes and providing useful error messages.

## Homework / Practice

1. Write a Python program that reads a file, counts the number of lines, and handles the case when the file does not exist.
  2. Create a custom exception class called `InvalidAgeError` and raise it when the user enters an age less than 0.
  3. Write a program that takes user input and handles the exception if the input is not an integer.
  4. Modify a file read program to include exception handling, and ensure that the file is closed properly in all situations.
- 
- 

## CS306 – Lecture 15: Object-Oriented Programming (OOP) in Python – Part 1

---

### Lecture Objectives

- Understand the basic concepts of Object-Oriented Programming (OOP).
  - Learn how to define and create classes and objects in Python.
  - Understand attributes and methods.
  - Learn the concept of `self` and constructors (`__init__` method).
- 

### 1. What is Object-Oriented Programming (OOP)?

**OOP** is a programming paradigm based on the concept of “**objects**”, which can contain **data** (attributes) and **functions** (methods).

Key Concepts of OOP:

- **Class**: A blueprint for creating objects.
  - **Object**: An instance of a class.
  - **Attribute**: A variable bound to the object.
  - **Method**: A function bound to the object.
-

## 2. Classes and Objects in Python

Creating a Class:

```
class Student:
    pass
```

This defines an empty class named `Student`.

Creating an Object:

```
s1 = Student()
```

Here, `s1` is an **object** (or instance) of the `Student` class.

---

## 3. The `__init__()` Method (Constructor)

The `__init__()` method is automatically called when a new object is created. It is used to initialize attributes.

Example:

```
class Student:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no
```

Creating Objects:

```
s1 = Student("Ali", 123)
s2 = Student("Sara", 456)
```

Accessing Attributes:

```
print(s1.name)      # Output: Ali
print(s2.roll_no)   # Output: 456
```

---

## 4. The `self` Keyword

The `self` keyword refers to the current instance of the class. It must be the first parameter of any method in the class.

Example:

```
class Student:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, my name is {self.name}")
```

Usage:

```
s = Student("Ayesha")
s.greet()  # Output: Hello, my name is Ayesha
```

---

## 5. Adding Methods to a Class

You can add multiple methods to perform actions related to the object.

Example:

```
class Calculator:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add(self):
        return self.a + self.b

    def subtract(self):
        return self.a - self.b
```

Using Methods:

```
calc = Calculator(10, 5)
print(calc.add())      # Output: 15
print(calc.subtract()) # Output: 5
```

---

## 6. Class vs Instance Attributes

- **Instance Attributes:** Defined in the `__init__()` method and are unique to each object.
- **Class Attributes:** Defined outside any methods and shared across all instances.

Example of Class Attribute:

```
class Dog:
    species = "Canine" # class attribute

    def __init__(self, name):
        self.name = name # instance attribute
```

Access:

```
d = Dog("Buddy")
print(d.name)      # Output: Buddy
print(d.species)   # Output: Canine
```

---

## 7. Dynamic Attributes

Attributes can be added to an object at runtime.

```
d.age = 5
print(d.age)    # Output: 5
```

However, this is not a common practice and may lead to inconsistent object structures.

---

## 8. The `__str__()` Method

The `__str__()` method is used to define how an object is represented as a string.

Example:

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"{self.title} by {self.author}"
```

Usage:

```
b = Book("Python 101", "John")
print(b)  # Output: Python 101 by John
```

---

## 9. Practice Example

Problem:

Create a class `BankAccount` with attributes: account holder name, account number, and balance. Include methods to deposit, withdraw, and check balance.

Solution:

```
class BankAccount:
    def __init__(self, name, account_no, balance=0):
        self.name = name
        self.account_no = account_no
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient balance!")
        else:
            self.balance -= amount

    def check_balance(self):
        print(f"Current balance: {self.balance}")
```

---

## 10. Summary

- Python supports object-oriented programming using classes and objects.
- The `__init__()` method initializes object attributes.
- `self` refers to the current object.
- Attributes can be either instance-level or class-level.
- Methods define the behavior of objects.

- The `__str__()` method defines string representation of an object.

---

## Homework / Practice

1. Create a class `Car` with attributes: make, model, and year. Add a method to display car information.
  2. Define a class `Rectangle` with attributes length and width, and methods to calculate area and perimeter.
  3. Create a class `Person` with attributes name and age, and a method `is_adult()` that returns `True` if `age >= 18`.
- 
- 

## CS306 – Lecture 16: Object-Oriented Programming in Python – Part 2 (Inheritance and Polymorphism)

---

### Lecture Objectives

- Understand the concept of **inheritance** in Python.
  - Learn how to create **subclasses**.
  - Explore the use of the `super()` function.
  - Understand **polymorphism** and method overriding.
- 

### 1. Inheritance in Python

**Inheritance** allows a class (child/subclass) to inherit attributes and methods from another class (parent/superclass).

Benefits of Inheritance:

- Code reusability
- Better organization of code
- Supports extensibility

Syntax:

```
class Parent:
    # parent class code

class Child(Parent):
```

```
# child class code
```

---

## 2. Example: Simple Inheritance

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    pass

d = Dog()
d.speak() # Output: Animal speaks
```

Here, Dog inherits the `speak()` method from Animal.

---

## 3. Overriding Methods

A child class can override methods of its parent class.

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

d = Dog()
d.speak() # Output: Dog barks
```

---

## 4. The `super()` Function

The `super()` function is used to call methods from the parent class.

Example:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound")

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

    def speak(self):
```

```
super().speak()  
print(f"{self.name} barks")
```

Usage:

```
d = Dog("Buddy", "Labrador")  
d.speak()
```

### Output:

```
Buddy makes a sound  
Buddy barks
```

---

## 5. Types of Inheritance in Python

### 1. Single Inheritance

- One child class inherits from one parent class.

### 2. Multiple Inheritance

- One child class inherits from multiple parent classes.

### 3. Multilevel Inheritance

- A class is derived from a derived class.

### 4. Hierarchical Inheritance

- Multiple child classes inherit from a single parent.
- 

## 6. Example: Multiple Inheritance

```
class Father:  
    def skills(self):  
        print("Gardening, Programming")  
  
class Mother:  
    def skills(self):  
        print("Cooking, Art")  
  
class Child(Father, Mother):  
    def skills(self):  
        Father.skills(self)  
        Mother.skills(self)  
        print("Sports")  
  
c = Child()  
c.skills()
```

### Output:

```
Gardening, Programming  
Cooking, Art  
Sports
```

---

## 7. Polymorphism

Polymorphism allows different classes to define methods with the same name, but possibly different behaviors.

Example:

```
class Bird:
    def fly(self):
        print("Bird can fly")

class Airplane:
    def fly(self):
        print("Airplane flies in the sky")

# Polymorphism using function
def fly_test(entity):
    entity.fly()

b = Bird()
a = Airplane()

fly_test(b) # Output: Bird can fly
fly_test(a) # Output: Airplane flies in the sky
```

---

## 8. The `isinstance()` Function

Used to check if an object belongs to a specific class or its subclass.

```
print(isinstance(d, Dog)) # True
print(isinstance(d, Animal)) # True
```

---

## 9. Real-World Example

Problem:

Model a `Person` class and derive a `Student` class that adds student-specific attributes and behavior.

Code:

```
class Person:
    def __init__(self, name):
        self.name = name

    def introduce(self):
        print(f"Hi, I'm {self.name}")

class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)
        self.student_id = student_id
```

```
def introduce(self):  
    print(f"Hi, I'm {self.name}, my student ID is {self.student_id}")
```

Usage:

```
s = Student("Aisha", "VU1234")  
s.introduce() # Output: Hi, I'm Aisha, my student ID is VU1234
```

---

## 10. Summary

- **Inheritance** allows reuse of code from parent classes.
- **Method Overriding** lets you modify inherited behavior.
- The `super()` function accesses methods from the parent class.
- **Polymorphism** enables different classes to use the same method names with different implementations.
- Python supports **multiple types of inheritance**, including multiple and multilevel.

---

## Homework / Practice

1. Create a class `Employee` with attributes `name` and `salary`. Create a subclass `Manager` that adds a `department`.
2. Model a `Vehicle` class and derive `Car` and `Bike` classes with specific attributes.
3. Demonstrate method overriding using a `Shape` class and `Circle` and `Square` subclasses.

---

## CS306 – Lecture 17: Encapsulation and Data Hiding in Python

### Lecture Objectives

- Understand the concept of **encapsulation** in object-oriented programming.
- Learn how to implement **data hiding** in Python.
- Differentiate between **public**, **protected**, and **private** attributes.
- Use **getter and setter** methods to control access.

---

### 1. What is Encapsulation?

**Encapsulation** is the process of bundling data (attributes) and methods that operate on that data into a single unit — a class.

It also helps in **restricting direct access** to some of an object's components, which is known as **data hiding**.

---

## 2. Importance of Encapsulation

- Controls access to the internal state of objects.
  - Improves code maintainability and flexibility.
  - Protects object integrity by preventing unintended changes.
- 

## 3. Access Modifiers in Python

Python uses **conventions** (not strict rules) to denote access control:

### **Modifier Syntax**

### **Access Level**

Public     `var`     Accessible anywhere

Protected   `_var`     Accessible in subclass

Private     `__var`     Name mangled, internal use only

---

## 4. Public Members

Accessible from anywhere.

```
class Person:
    def __init__(self, name):
        self.name = name   # public

p = Person("Ali")
print(p.name)   # Output: Ali
```

---

## 5. Protected Members

Convention: prefix with a single underscore `_`.

```
class Person:
    def __init__(self, name):
        self._name = name   # protected

class Student(Person):
    def display(self):
```

```
print("Name:", self._name)

s = Student("Sara")
s.display() # Output: Name: Sara
```

**Note:** It's still accessible, but considered for internal use or subclassing. 

---

## 6. Private Members

Prefix with double underscore `__`. Python does **name mangling**, making the variable harder to access from outside.

```
class Person:
    def __init__(self, name):
        self.__name = name # private

p = Person("Usman")
# print(p.__name) # Error: AttributeError

# Access using name mangling
print(p._Person__name) # Output: Usman
```

---

## 7. Getter and Setter Methods

Used to **safely access or modify private attributes**.

Example:

```
class BankAccount:
    def __init__(self):
        self.__balance = 0

    def get_balance(self):
        return self.__balance

    def set_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid amount")

acc = BankAccount()
acc.set_balance(1000)
print(acc.get_balance()) # Output: 1000
```

---

## 8. Properties (Alternative to Getters/Setters)

Python provides a built-in `@property` decorator.

```
class BankAccount:
    def __init__(self):
        self.__balance = 0

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid amount")

acc = BankAccount()
acc.balance = 500
print(acc.balance)  # Output: 500
```

---

## 9. Practical Example

Problem: Model a `Student` class with name and GPA, ensuring GPA is validated (0.0 to 4.0).

```
class Student:
    def __init__(self, name):
        self.__name = name
        self.__gpa = 0.0

    def set_gpa(self, gpa):
        if 0.0 <= gpa <= 4.0:
            self.__gpa = gpa
        else:
            print("Invalid GPA")

    def get_gpa(self):
        return self.__gpa

s = Student("Maria")
s.set_gpa(3.5)
print(s.get_gpa())  # Output: 3.5
```

---

## 10. Summary

- **Encapsulation** keeps data safe from unintended access or modification.
  - Python supports **public**, **protected**, and **private** attributes (by convention).
  - Use **getter/setter** methods or **@property decorators** to access or update private variables safely.
  - Data hiding improves program security and flexibility.
-

## Homework / Practice

1. Create a `Car` class with private attributes `model` and `price`. Add getter and setter methods.
  2. Implement a `Rectangle` class that hides its `length` and `width` and calculates area through a public method.
  3. Use `@property` to manage access to a `Student` class's percentage attribute.
- 
- 

## CS306 – Lecture 18: Working with Files in Python

---

### Lecture Objectives

- Understand how to **create, open, read, and write** files in Python.
  - Learn about different **file modes**.
  - Use the **with statement** to handle files safely.
  - Handle **file exceptions**.
- 

### 1. Introduction to File Handling

File handling is used to store data **permanently** into files (external storage like `.txt`, `.csv`, etc.).

Python provides built-in functions for:

- Creating files
  - Writing to files
  - Reading from files
  - Appending data
  - Closing files
- 

### 2. Opening Files with `open()`

Syntax:

```
file_object = open("filename", "mode")
```

### Common File Modes:

| Mode | Description                                    |
|------|------------------------------------------------|
| 'r'  | Read (default), error if file doesn't exist    |
| 'w'  | Write, creates file if not exists (overwrites) |
| 'a'  | Append, creates if not exists                  |
| 'b'  | Binary mode                                    |
| '+'  | Read & write mode                              |

---

### 3. Writing to a File

Example:

```
file = open("data.txt", "w")
file.write("Hello, world!")
file.close()
```

Creates or overwrites `data.txt` with the content.

---

### 4. Reading from a File

```
file = open("data.txt", "r")
content = file.read()
print(content)
file.close()
```

Other Read Methods:

- `read()`: Reads entire file
  - `readline()`: Reads one line
  - `readlines()`: Reads all lines as a list
- 

### 5. Appending to a File

```
file = open("data.txt", "a")
file.write("\nNew line added.")
file.close()
```

---

### 6. Using `with` Statement (Best Practice)

Automatically closes the file after execution.

```
with open("data.txt", "r") as file:  
    print(file.read())
```

For Writing:

```
with open("data.txt", "w") as file:  
    file.write("Overwritten content.")
```

---

## 7. File Not Found Error

Always check if the file exists when using read mode.

```
try:  
    with open("nonexistent.txt", "r") as f:  
        print(f.read())  
except FileNotFoundError:  
    print("File does not exist.")
```

---

## 8. Example: Write & Read Student Records

Writing:

```
with open("students.txt", "w") as file:  
    file.write("Ali, 85\n")  
    file.write("Sara, 92\n")
```

Reading:

```
with open("students.txt", "r") as file:  
    for line in file:  
        print(line.strip())
```

---

## 9. Working with Binary Files

```
with open("image.jpg", "rb") as f:  
    data = f.read()
```

```
with open("copy.jpg", "wb") as f:  
    f.write(data)
```

---

## 10. Summary

- Python provides `open()` for file handling with multiple modes (`r`, `w`, `a`, etc.).
  - Always **close** the file or use `with` to do it automatically.
  - Use **exception handling** to manage file errors.
  - Files can be read as full text, line-by-line, or into lists.
  - Binary files use `'rb'` and `'wb'` modes.
-

## Homework / Practice

1. Write a Python program to create a file called `marks.txt` and store 5 students' names and marks.
  2. Read from the `marks.txt` file and print each student's result.
  3. Use the `with` statement to append new student data to `marks.txt`.
- 
- 

## CS306 – Lecture 19: Exception Handling in Python

---

### Lecture Objectives

- Understand what **exceptions** are and why they occur.
  - Learn how to **handle exceptions** using `try`, `except`, `finally`, and `else`.
  - Explore **built-in exception types**.
  - Raise custom exceptions using `raise`.
- 

### 1. What is an Exception?

An **exception** is an error that occurs **during the execution of a program**, disrupting its normal flow.

Example:

```
x = 10 / 0 # ZeroDivisionError
```

Without handling, the program crashes.

---

### 2. Exception Handling Syntax

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the exception
```

Example:

```
try:
    a = int(input("Enter a number: "))
    b = 10 / a
    print("Result:", b)
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

### 3. Catching Multiple Exceptions

```
try:
    x = int("abc")
except ValueError:
    print("Invalid input.")
except ZeroDivisionError:
    print("Division error.")
```

You can also combine them:

```
try:
    # some code
except (ValueError, ZeroDivisionError) as e:
    print("Error:", e)
```

---

### 4. Using else Clause

Executes if no exception occurs.

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid number.")
else:
    print("Square is", num ** 2)
```

---

### 5. Using finally Clause

Always executes, regardless of whether an exception occurred.

```
try:
    file = open("sample.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    print("Cleaning up.")
```

---

### 6. Raising Exceptions Manually

You can raise exceptions using `raise`.

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative.")
```

---

## 7. Common Built-in Exceptions

| Exception         | Description              |
|-------------------|--------------------------|
| ZeroDivisionError | Division by zero         |
| ValueError        | Invalid value            |
| TypeError         | Wrong type of data       |
| FileNotFoundError | File not found           |
| IndexError        | List index out of range  |
| KeyError          | Dictionary key not found |

---

## 8. Example: File Handling with Exception

```
try:
    with open("marks.txt", "r") as file:
        data = file.read()
        print(data)
except FileNotFoundError:
    print("The file was not found.")
finally:
    print("File operation complete.")
```

---

## 9. Summary

- Use `try...except` to catch and handle exceptions.
  - Use `else` for code that runs if no exception occurs.
  - Use `finally` to execute cleanup code regardless of exceptions.
  - Raise exceptions manually using `raise`.
  - Knowing built-in exceptions helps debug effectively.
- 

## Homework / Practice

1. Write a program that asks the user to input a number and handles:
    - `ValueError` if input is not a number.
    - `ZeroDivisionError` if user enters 0.
  2. Read a file called `data.txt`, handle missing file errors and display contents.
  3. Write a custom function that raises a `ValueError` if input is not in a valid range.
- 
-

## CS306 – Lecture 20: Working with Modules in Python

### Lecture Objectives

- Understand the concept of **modules** in Python.
- Learn how to **import** and use **built-in** and **user-defined** modules.
- Discover the use of the **dir()** function.
- Understand the **`__name__ == "__main__"`** idiom.
- Explore the **math** and **random** modules.

### 1. What is a Module?

A **module** is a file containing Python code (functions, classes, variables) that can be reused in other programs.

Any `.py` file is a **module**.

### 2. Importing Modules

Syntax:

```
import module_name
```

Example:

```
import math
print(math.sqrt(16))    # Output: 4.0
```

Import Specific Items:

```
from math import sqrt
print(sqrt(25))
```

Import with Alias:

```
import math as m
print(m.pi)
```

### 3. Common Built-in Modules

| Module                | Description                |
|-----------------------|----------------------------|
| <code>math</code>     | Mathematical operations    |
| <code>random</code>   | Generate random numbers    |
| <code>datetime</code> | Date and time manipulation |

| Module | Description                       |
|--------|-----------------------------------|
| os     | Interacting with operating system |
| sys    | System-specific parameters        |

---

#### 4. The `math` Module

Common Functions:

| Function                    | Description             |
|-----------------------------|-------------------------|
| <code>math.sqrt(x)</code>   | Square root             |
| <code>math.pow(x, y)</code> | x raised to the power y |
| <code>math.floor(x)</code>  | Round down              |
| <code>math.ceil(x)</code>   | Round up                |
| <code>math.pi</code>        | Value of $\pi$          |

Example:

```
import math
print(math.floor(4.7)) # Output: 4
```

---

#### 5. The `random` Module

Used for generating random numbers.

Example:

```
import random
print(random.randint(1, 10)) # Random int between 1 and 10
```

Other Functions:

| Function                          | Description               |
|-----------------------------------|---------------------------|
| <code>random.random()</code>      | Float between 0.0 to 1.0  |
| <code>random.choice(list)</code>  | Randomly select from list |
| <code>random.shuffle(list)</code> | Shuffle list              |

---

#### 6. Creating Your Own Module

Suppose you create a file named `mytools.py`:

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

You can use it in another file:

```
import mytools  
print(mytools.greet("Ali"))
```

---

## 7. Using `dir()` Function

Lists all functions, classes, and variables defined in a module.

```
import math  
print(dir(math))
```

---

## 8. `__name__ == "__main__"`

Used to separate module usage and script execution.

```
# file: greet.py  
def hello():  
    print("Hello!")  
  
if __name__ == "__main__":  
    hello()
```

When `greet.py` is imported in another file, `hello()` won't run unless called directly.

---

## 9. Summary

- Modules organize and reuse code across programs.
  - Use `import`, `from ... import`, or aliases to access module content.
  - Python includes many powerful built-in modules (`math`, `random`, etc.).
  - You can create your own modules by saving Python code in `.py` files.
  - `__name__ == "__main__"` ensures code only runs when executed directly.
- 

## Homework / Practice

### 1. Import the `math` module and display:

- Square root of 100
- Value of pi
- Result of `math.pow(2, 5)`

2. Create a module named `mycalc.py` with two functions: `add(x, y)` and `subtract(x, y)`. Import and use them in another script.
  3. Use the `random` module to create a simple dice roll simulation.
-