LECTURE ONE

**Introduction to Python and Jupyter Notebooks for Data Science**

Python is one of the most important and popular programming languages in the world, especially for data science. Its popularity stems from its ease of use, readability, and a syntax that closely resembles the English language, making it accessible for beginners and professionals alike1.

**Why Python for Data Science?**

* **Ease of Learning:** Python is considered easier to learn compared to other programming languages. Its syntax is straightforward, which helps new learners focus on problem-solving rather than complex language rules1.
* **Versatility:** Python supports multiple programming paradigms and has extensive libraries for data analysis, machine learning, and visualization.
* **Community Support:** Python has a large, active community, ensuring ample resources, tutorials, and third-party packages.

**Setting Up the Python Environment**

To start working with Python for data science, you need to install Python (preferably version 3.10 or newer) and set up an Integrated Development Environment (IDE). The recommended workflow involves using Jupyter Notebook, which provides an interactive web-based interface for writing and running code alongside documentation1.

**Key Tools**

* **Jupyter Notebook:** Allows you to write code and documentation (Markdown) in the same document. It’s ideal for data analysis, as you can run code snippets and immediately see results, making it easier to iterate and document your work1.
* **Anaconda:** A software distribution that simplifies the installation of Python and Jupyter, along with other data science tools like Spyder and R1.
* **Other IDEs:** While you can write Python code in command prompts or other editors like VS Code or MySQL Workbench (for SQL), Jupyter is preferred for its interactivity and ease of use in data science workflows1.

**Getting Started with Jupyter Notebook**

1. **Installation:** Install Anaconda, then launch Jupyter Notebook from the Anaconda Navigator1.
2. **Creating a Workspace:** Before starting, create a dedicated folder (e.g., "python\_data\_class") on your desktop to organize your work1.
3. **Launching Jupyter:** Open Jupyter Notebook, navigate to your working folder, and start a new notebook1.

**Notebook Structure**

* **Cells:** The notebook consists of cells, which can be either code cells (for Python code) or Markdown cells (for formatted text and documentation)1.
* **Running Code:** Write Python code in a code cell and run it to see the output immediately below the cell1.
* **Markdown Cells:** Use Markdown to add titles, explanations, and formatting. For example, use # for a large title, ## for subtitles, and plain text for explanations1.

**Basic Python Programming Concepts**

**Variables and Data Types**

* **Assignment:** Use the equals sign = to assign values to variables.
* **Common Data Types:**
  + String (str): Text data, e.g., "John"
  + Integer (int): Whole numbers, e.g., 23
  + Float (float): Decimal numbers, e.g., 3.14
  + Boolean (bool): True/False values1

**Input and Output**

* **Input:** Use input() to accept user input.
* **Output:** Use print() to display output to the user1.

**Example: Accepting User Input and Printing Output**

python

name = input("Enter your name: ")

**print**("Welcome to class,", name)

**Working with Jupyter Notebook Features**

* **Docstrings:** Jupyter provides easy access to documentation for functions and methods using docstrings. This helps you quickly inspect how a function works without leaving the notebook1.
* **Stopping Execution:** If a code cell is taking too long to run, you can stop it using the stop button in the notebook interface1.
* **Saving Work:** Notebooks are saved as .ipynb files in your working directory, making it easy to organize and revisit your work1.

**Best Practices**

* **Organization:** Always create and use a dedicated working folder for your projects.
* **Documentation:** Use Markdown cells to document your code, explain your logic, and make your work understandable for others and your future self.
* **Error Handling:** Read error messages carefully to debug your code efficiently. Errors are a normal part of programming and provide clues to fix issues1.
* **Consistency:** Type code accurately, as Python is case-sensitive and syntax errors can be common for beginners1.

**Example: Simple Python Program in Jupyter Notebook**

python

*# Accept user's name and year of birth, then calculate age*

name = input("Enter your name: ")

year\_of\_birth = int(input("Enter your year of birth: "))

current\_year = 2025

age = current\_year - year\_of\_birth

**print**("Hello", name + ", you are", age, "years old.")

LECTURE TWO

**Python Variables, Data Types, and Mathematical Operators**

**1. Recap: Variables and Data Types**

* **Variables** are containers for storing data values. In Python, you create a variable by assigning it a value with the = sign.
* **Data Types** commonly used in Python:
  + **Integers (int)**: Whole numbers (e.g., 5, -3)
  + **Floats (float)**: Numbers with decimals (e.g., 3.14, -0.5)
  + **Strings (str)**: Text (e.g., "hello")
  + **Booleans (bool)**: True or False

**2. Mathematical (Arithmetic) Operators in Python**

Mathematical operators allow you to perform calculations in your code. Here are the main ones:

| **Operator** | **Symbol** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- | --- |
| Addition | + | Adds values | 2 + 5 | 7 |
| Subtraction | - | Subtracts right from left | 6 - 2 | 4 |
| Multiplication | \* | Multiplies values | 3 \* 4 | 12 |
| Division | / | Divides left by right (always returns a float) | 10 / 2 | 5.0 |
| Floor Division | // | Divides and returns the integer part | 15 // 2 | 7 |
| Modulus | % | Returns the remainder | 15 % 2 | 1 |
| Exponentiation | \*\* | Raises left to the power of right | 3 \*\* 2 | 9 |

**Details and Examples**

* **Addition (+)**
  + Works with numbers and strings.
  + Example: print(2 + 5) outputs 7.
  + String concatenation: "John" + " " + "Doe" outputs "John Doe".
* **Subtraction (-)**
  + Works with numbers.
  + Example: print(6 - 2) outputs 4.
* **Multiplication (\*)**
  + Works with numbers and can repeat strings.
  + Example: print(3 \* 4) outputs 12.
  + Example: "Hi" \* 3 outputs "HiHiHi".
* **Division (/)**
  + Always returns a float, even if the result is a whole number.
  + Example: print(10 / 2) outputs 5.0.
* **Floor Division (//)**
  + Returns the integer part of the division (no decimals).
  + Example: print(15 // 2) outputs 7.
* **Modulus (%)**
  + Returns the remainder after division.
  + Example: print(15 % 2) outputs 1.
* **Exponentiation (\*\*)**
  + Raises a number to the power of another.
  + Example: print(3 \*\* 2) outputs 9.
  + **Square root**: print(9 \*\* (1/2)) outputs 3.0.
  + **Cube root**: print(27 \*\* (1/3)) outputs 3.0.

**3. Type Casting (Converting Data Types)**

* When you use input() in Python, the value is always a string.
* To perform arithmetic, convert (cast) the string to an integer or float using int() or float().
* Example:

python

age = input("Enter your age: ")

**print**(type(age)) *# <class 'str'>*

age = int(age)

**print**(type(age)) *# <class 'int'>*

**4. Practical Examples**

**A. Calculating the Area of a Rectangle**

python

length = int(input("Enter the value of length: "))

breadth = int(input("Enter the value of breadth: "))

area = length \* breadth

**print**(f"The area of your rectangle is {area} cm squared")

* Always convert input to int or float before calculations, otherwise Python will throw an error if you try to multiply strings.

**B. Calculating the Area of a Triangle**

python

base = int(input("Enter the base: "))

height = int(input("Enter the height: "))

area = 0.5 \* base \* height

**print**(f"The area of the triangle is {area}")

**5. Common Mistakes and Tips**

* **Forgetting to convert input:** Always cast input to the correct type before calculations.
* **String concatenation vs. arithmetic:** Use + for both, but be mindful of types (numbers vs. strings).
* **Division always returns float:** Even 10 / 2 gives 5.0, not 5.

LECTURE THREE

**Practical Examples and Built-in Functions**

**1. Converting Days to Weeks and Days**

When you have a number of days and want to know how many weeks and extra days that represents, you can use simple arithmetic:

* **Weeks**: Divide the total days by 7 and take only the whole number (integer division).
* **Remaining Days**: Use the modulus operator to get the remainder.

**Python Implementation**

python

*# Accept user input for number of days*

days = int(input("Enter the number of days you want to convert: "))

*# Calculate weeks and remaining days*

weeks = days // 7

remaining\_days = days % 7

*# Output the result*

**print**(f"There are {weeks} week(s) and {remaining\_days} day(s) in {days} days.")

**Explanation:**

* // is the floor division operator, which gives the whole number part of the division.
* % is the modulus operator, which gives the remainder.

**Example:**

* If you enter 15, output will be: There are 2 week(s) and 1 day(s) in 15 days.

**2. Building a Currency Converter in Python**

A currency converter takes an amount in one currency and converts it to another using a given exchange rate.

**Steps:**

1. **Accept the base currency type (e.g., Dollar, Naira, Pound).**
2. **Accept the target currency type.**
3. **Accept the amount to convert.**
4. **Accept the exchange rate.**
5. **Calculate and display the converted amount.**

**Python Implementation**

python

*# Step 1: Accept base currency*

base\_currency = input("Enter the currency you want to convert from (e.g., Dollar): ")

*# Step 2: Accept amount*

amount = float(input(f"How many {base\_currency} do you want to convert? "))

*# Step 3: Accept target currency*

target\_currency = input(f"Enter the currency you want to convert your {amount} {base\_currency} to: ")

*# Step 4: Accept exchange rate*

rate = float(input(f"What is the current {base\_currency} to {target\_currency} exchange rate? "))

*# Step 5: Calculate and display result*

converted\_amount = amount \* rate

**print**(f"{amount} {base\_currency} is {converted\_amount} {target\_currency}.")

**Example:**

* If you want to convert 300 Dollars to Naira at a rate of 1300, output will be: 300 Dollar is 390000.0 Naira.

**3. Introduction to Built-in Python Functions**

Python has many built-in functions that make programming easier. Here are some important ones:

| **Function** | **Usage** | **Example** | **Description** |
| --- | --- | --- | --- |
| int() | int("5") | Converts a value to integer |  |
| float() | float("3.14") | Converts a value to float (decimal) |  |
| str() | str(100) | Converts a value to string |  |
| print() | print("Hello") | Outputs data to the screen |  |
| input() | input("Prompt") | Accepts user input as string |  |
| len() | len("Python") | Returns number of items/characters |  |
| sum() | sum([1]) | Returns the sum of items in an iterable |  |
| round() | round(3.14159, 2) | Rounds a number to specified decimals |  |
| range() | range(5) | Returns a sequence of numbers |  |

**Examples:**

* **Counting Letters in a Name**

python

name = input("Enter your name: ")

**print**(f"There are {len(name)} letters in your name.")

* **Summing Numbers in a List**

python

numbers = [1, 2, 3, 4]

**print**(f"The sum is {sum(numbers)}.")

**4. Understanding Function Parameters**

* **Function Structure:**  
  A function in Python is called by its name followed by parentheses. You put values (arguments) inside the parentheses.

**Example**:

python

**print**("Hello, World!") *# "Hello, World!" is the argument*

* **Parameters vs Arguments:**
  + *Parameter* is the variable listed inside the function definition.
  + *Argument* is the value sent to the function when it is called.

**5. Key Takeaways**

* Use // for integer division (e.g., weeks from days).
* Use % for the remainder (e.g., extra days).
* Always type-cast user input for calculations (int(), float()).
* Built-in functions like len(), sum(), print(), and input() simplify many tasks.
* Think through the logic step by step before coding.

LECTURE FOUR

**Loan App, Simple Interest, Conditional Logic, and Python Lists**

**1. Loan App: Calculating Simple Interest and Amount Payable**

**Overview**

A basic loan application program calculates the interest and total amount to be repaid based on the principal (amount borrowed), time (years), and a fixed interest rate. Additionally, if the loan amount exceeds a certain threshold (₦300,000), the borrower receives a 30% discount on the interest.

**Step-by-Step Logic**

1. **Accept User Inputs**
   * Principal (amount to borrow)
   * Number of years (duration of the loan)
2. **Set the Interest Rate**
   * The company determines a fixed interest rate (e.g., 3%).
3. **Calculate Simple Interest**
   * Formula:

Interest=(Principal×Rate×Time)/100

* + In the given context, since the rate is always 3% and time is in years, the formula simplifies to:

Interest=Principal×0.03×Years

1. **Apply Discount for Large Loans**
   * If the principal is greater than 300,000:
     + Apply a 30% discount to the interest (i.e., pay only 70% of the calculated interest).
     + Notify the user about the discount.
   * Otherwise, use the full interest.
2. **Calculate Total Amount Payable**
   * Total Amount} =Principal + Interest #(after discount if applicable)#
3. **Display Results**
   * Show the interest (and discount if any) and the total repayment amount.

**Sample Python Implementation**

python

*# Step 1: Get user input*

principal = float(input("Enter the amount you want to borrow: "))

years = int(input("Enter the number of years: "))

*# Step 2: Set fixed interest rate*

rate = 3 # *3 percent*

*# Step 3: Calculate interest*

interest = (principal \* rate \* years) / 100

*# Step 4: Apply discount if applicable*

**if** principal > 300000:

discount = 0.3 \* interest *# 30% discount*

interest \*= 0.7 *# Pay only 70% of the interest*

**print**(f"You have gotten a 30% interest discount of {discount:.2f}")

**else**:

discount = 0

*# Step 5: Calculate total amount payable*

total\_amount = principal + interest

*# Step 6: Display results*

**print**(f"Your repayment amount is {total\_amount:.2f}")

**2. Conditional Logic in Python**

* **If-Else Statements:** Used to execute code blocks based on conditions.
* Only one block in an if-else chain will execute, depending on which condition is met.
* **Example:**

python

**if** principal > 300000:

*# Apply discount*

**else**:

*# No discount*

**3. Lists and Tuples in Python**

**Lists**

* **Definition:** Lists are collections of elements, enclosed in square brackets [ ].
* **Mutable:** Elements can be changed after creation.
* **Examples:**

python

fruits = ["mango", "orange", "cashew"]

numbers = [1, 2, 3, 4, 5]

**Tuples**

* **Definition:** Tuples are collections of elements, enclosed in parentheses ( ).
* **Immutable:** Elements cannot be changed after creation.
* **Example:**

python

states = ("Lagos", "Ogun", "Oyo")

**Key Differences**

| **Feature** | **List** | **Tuple** |
| --- | --- | --- |
| Syntax | [1,[2][3] | (1, 2, 3) |
| Mutability | Mutable (changeable) | Immutable (fixed) |
| Methods | Many | Few |

**Creating Lists**

* Directly: numbers = [1, 2, 3, 4,5]
* Using range() and list():

python

numbers = list(range(1, 11)) *# [1, 2, ..., 10]*

**Indexing and Slicing**

* **Indexing:** Access elements by position (starting from 0).

python

**print**(fruits[0]) *# 'mango'*

**print**(fruits[1]) *# 'orange'*

**print**(fruits[-1]) *# 'cashew' (last element)*

* **Slicing:** Get a sublist by specifying a range.

python

**print**(fruits[0:2]) *# ['mango', 'orange']*

**print**(fruits[:3]) *# ['mango', 'orange', 'cashew']*

**Forward and Backward Indexing**

* **Forward:** fruits[0] is first, fruits[1] is second, etc.
* **Backward:** fruits[-1] is last, fruits[-2] is second to last, etc.

**4. Practical Examples**

**A. List of Fruits**

python

fruits = ["mango", "orange", "cashew", "melon", "carrot"]

**print**(fruits[0:4]) *# ['mango', 'orange', 'cashew', 'melon']*

**B. List of Numbers Using Range**

python

numbers = list(range(1, 11)) *# [1, 2, 3, ..., 10]*

**print**(numbers)

**C. Accessing Elements**

python

**print**(fruits[0]) *# 'mango'*

**print**(fruits[-1]) *# 'carrot'*

**4. Dictionaries in Python**

**Definition**

* A dictionary is an unordered collection of key-value pairs.
* Keys are unique and used to access values.
* Defined using curly braces {}.

**Syntax and Example**

python

student = { "name": "John Doe", "age": 20,"department": "Computer Science"}

**Key Features**

* **Accessing Values:** Use the key inside square brackets.

python

**print**(student["name"]) *# Output: John Doe*

* **Adding/Updating:** Assign a value to a key.

python

student["level"] = 200

* **Removing:** Use del or pop().

python

**del** student["age"]

**Common Dictionary Methods**

| **Method** | **Description** |
| --- | --- |
| .keys() | Returns all keys |
| .values() | Returns all values |
| .items() | Returns all key-value pairs as tuples |
| .get(key) | Returns value for key, or None if missing |

**Use Cases**

* Storing structured data (e.g., user profiles, inventory, configurations).

**5. Sets in Python**

**Definition**

* A set is an unordered collection of unique elements (no duplicates).
* Defined using curly braces {} or the set() function.

**Syntax and Example**

python

fruits\_set = {"mango", "orange", "cashew", "mango"}

**print**(fruits\_set) *# Output: {'mango', 'orange', 'cashew'}*

**Key Features**

* **No Duplicates:** Each element appears only once.
* **Unordered:** No indexing or slicing.
* **Mutable:** You can add or remove elements.

**Common Set Methods**

| **Method** | **Description** |
| --- | --- |
| .add(x) | Adds element x to the set |
| .remove(x) | Removes x; error if not present |
| .discard(x) | Removes x; does nothing if not present |
| .union() | Returns the union of two sets |
| .intersection() | Returns common elements |
| .difference() | Returns elements in one set but not the other |

**Use Cases**

* Removing duplicates from a list
* Membership testing (checking if an item exists)
* Mathematical set operations

**6. Key Takeaways**

* Use conditional logic (if-else) to handle different scenarios in your programs.
* Lists are mutable and use square brackets; tuples are immutable and use parentheses.
* Indexing starts at 0 in Python; negative indices count from the end.
* Slicing allows you to extract sublists.
* Always test your code with various inputs to ensure all conditions work as expected.

LECTURE FIVE

**While Loops and Dictionaries (Part 1)**

**1. Introduction to the Program: Finding the Least Common Multiple (LCM)**

**Objective:** Write a Python program to find the LCM of two numbers.  
**What is LCM?**  
The **smallest positive integer** that is divisible by both input numbers.  
**Example:**

* LCM of 2 and 3 is **6** (6 is divisible by both 2 and 3).
* LCM of 2, 3, and 4 is **12** (12 is divisible by all three).

**2. Understanding the Problem**

**Common Misconceptions**

* **Myth 1:** LCM is always the product of the two numbers.
  + **Reality:** LCM(a, b) = (a × b) / GCD(a, b). For example, LCM(4, 6) = 12 (not 24).
* **Myth 2:** Smaller numbers can be LCM if they are less than the larger number.
  + **Reality:** LCM must be ≥ the larger input number. For example, LCM(4, 8) = 8 (not 4).

**Approach to Find LCM**

1. **Identify the greater number** between the two inputs.
2. **Start checking from the greater number upwards.**
3. **Find the first number divisible by both inputs** using a loop.

**3. Code Implementation**

python

*# Step 1: Accept user inputs*

num1 = int(input("Enter first number: "))

num2 = int(input("Enter second number: "))

*# Step 2: Determine the greater number*

greater = max(num1, num2)

*# Step 3: Find LCM using a while loop*

**while** True:

**if** greater % num1 == 0 **and** greater % num2 == 0:

lcm = greater

**break**

greater += 1

*# Step 4: Display the result*

**print**(f"LCM of {num1} and {num2} is {lcm}")

**4. Code Explanation**

**Key Components**

1. **while True Loop:**
   * Runs indefinitely until the LCM is found.
   * Checks divisibility using the modulus operator %.
2. **Condition Check:**
   * greater % num1 == 0 and greater % num2 == 0:
     + Ensures the current greater value is divisible by both numbers.
3. **Incrementing greater:**
   * If the condition fails, greater increases by 1, and the loop repeats.
4. **Breaking the Loop:**
   * The loop exits immediately when the LCM is found (break statement).

**5. Examples**

**Example 1:**

* **Input:** num1 = 2, num2 = 5
* **Steps:**
  + greater starts at 5.
  + Check 5 → 5 % 2 ≠ 0 → increment to 6.
  + Check 6 → 6 % 2 = 0 and 6 % 5 ≠ 0 → increment to 10.
  + Check 10 → 10 % 2 = 0 and 10 % 5 = 0 → **LCM = 10**.

**Example 2:**

* **Input:** num1 = 4, num2 = 6
* **Steps:**
  + greater starts at 6.
  + Check 6 → 6 % 4 ≠ 0 → increment to 12.
  + Check 12 → 12 % 4 = 0 and 12 % 6 = 0 → **LCM = 12**.

**6. Key Takeaways**

1. **LCM Definition:** The smallest number divisible by both inputs.
2. **Algorithm Steps:**
   * Start from the larger number.
   * Incrementally check divisibility.
3. **Efficiency:** This method works but may be slow for large numbers (optimize using GCD in future lessons).
4. **Loop Usage:** while True loops are powerful but require a clear exit condition (break).

LECTURE SIX

**While Loops & Dictionaries (Part 2)**

**1. While Loop Overview**

**Purpose:**

A **while loop** repeatedly executes a block of code **as long as a specified condition is True**. It is useful when the number of iterations is not known beforehand and depends on dynamic conditions.

**Difference from For Loop:**

| **Feature** | **For Loop** | **While Loop** |
| --- | --- | --- |
| Iteration Count | Fixed or known beforehand (e.g., range) | Runs until a condition becomes False or a break occurs |
| Use Case | When the number of iterations is known | When iterations depend on dynamic conditions |
| Syntax | for i in range(1, 6): | while condition: |

**Basic Syntax:**

**python**

**while** condition:

*# code block to execute repeatedly*

**Example: Print numbers from 1 to 5**

python

counter = 1

**while** True:

**if** counter <= 5:

**print**(counter)

counter += 1

**else**:

**break**

* The loop runs indefinitely (while True).
* The if condition checks if counter is within the limit.
* When counter exceeds 5, the loop breaks.

**Key Notes:**

* **Infinite loops:** Without a proper break or condition change, while loops can run forever.
* **Use cases:** Ideal for programs requiring continuous input or waiting for a specific event.

**2. Attendance Register Program Using While Loop**

**Objective:**

Create a program that registers employee attendance continuously until the manager stops it by entering a special password.

**Components:**

* **Data Structures:**
  + register: An empty list to store names of employees who attended.
  + password: A string representing the manager’s password to stop the program (e.g., "XYZ").

**Logic Flow:**

1. Use while True to keep accepting input indefinitely.
2. Prompt the user to enter their name.
3. If the entered name matches the password, stop the loop (break).
4. Otherwise, add the name to the register list.
5. Print a welcome message for each valid entry.

**Sample Code:**

python

register = []

password = "XYZ"

**while** True:

name = input("Enter your name: ")

**if** name == password:

**break**

**else**:

register.append(name)

**print**(f"Welcome to the office, {name}")

**Why input is inside the loop?**

* To allow multiple employees to enter their names one after another.
* The loop continues until the manager enters the password to stop it.

**3. Validating Staff Names in Attendance Register**

**Problem:**

Anyone can enter a name, which can lead to unauthorized entries.

**Solution:**

* Define a list staffs containing the names of official staff members.
* When a name is entered:
  + If it matches the password, stop the loop.
  + Else if the name is in staffs, add it to the register.
  + Else, print a warning that the person is not authorized but **do not stop the program**.

**Code Snippet:**

python

staffs = ['George', 'Sasha', 'Henry', 'Tina'] *# Official staff list*

register = []

password = "XYZ"

**while** True:

name = input("Enter your name: ")

**if** name == password:

**break**

**elif** name **in** staffs:

register.append(name)

**print**(f"Welcome to the office, {name}")

**else**:

**print**(f"Sorry, {name} is not authorized to enter.")

**Explanation:**

* The program only registers valid staff.
* Unauthorized names are rejected with a warning.
* The loop continues to accept entries until the manager stops it.

**4. Dictionaries in Attendance Management (Introduction)**

While lists store names sequentially, **dictionaries** can be used to store more detailed attendance information, such as the number of days attended or timestamps.

**Basic Dictionary Structure:**

**python**

attendance = {“George": 5,"Sasha": 3,"Henry": 4}

* Keys: Staff names
* Values: Number of days attended (or other relevant data)

**Example: Updating Attendance Count**

python

name = input("Enter your name: ")

**if** name **in** attendance:

attendance[name] += 1 *# Increment attendance count*

**else**:

attendance[name] = 1 *# First attendance entry*

LECTURE SEVEN

**Python Functions (Part 1)**

**1. Introduction to Functions**

**What Are Functions?**

Functions are **reusable blocks of code** designed to perform specific tasks. They help:

* **Reduce code repetition** (write once, use multiple times).
* **Organize code** into logical modules.
* **Simplify debugging** by isolating functionality.

**Types of Functions**

* **Built-in Functions:** Predefined in Python (e.g., print(), max(), len()).
* **User-defined Functions:** Created by programmers to solve custom problems.

| **Built-in Functions** | **User-defined Functions** |
| --- | --- |
| print("Hello") | def greet(name): print(name) |
| max(10, 20) | def calculate\_area(length, width) |

**Why Create Functions?**

* Perform tasks not covered by built-in functions.
* Avoid rewriting the same code (e.g., calculating age multiple times).
* Improve code readability and maintainability.

**2. Function Structure**

**Syntax**

python

**def** function\_name(parameter1, parameter2, ...):

*# Function body (code to execute)*

**return** value *# Optional*

**Components**

1. **def** Keyword: Declares a function.
2. **Function Name:** Unique identifier (e.g., greet, calculate\_age).
3. **Parameters:** Inputs that customize the function’s behavior.
4. **Colon (:):** Marks the start of the function body.
5. **Function Body:** Indented code block (executes when called).
6. **return Statement:** Sends a result back to the caller (optional).

**3. Parameters vs. Arguments**

* **Parameters:** Variables listed in the function definition.
* **Arguments:** Values passed to the function during a call.

**Example Analogy:**

* **Function:** A tea maker named Gary.
* **Parameter:** Amount of hot water (input).
* **Output:** Tea (customized based on the input).

**Example Code**

python

**def** greet(first\_name, last\_name): *# Parameters: first\_name, last\_name*

message = f"{first\_name} {last\_name}, how are you today?"

**print**(message)

greet("John", "Doe") *# Arguments: "John", "Doe"*

**4. Creating and Calling Functions**

**A. Greeting Function**

python

**def** greet(first\_name, last\_name):

message = f"{first\_name} {last\_name}, how are you today?"

**print**(message)

*# Calling the function*

greet("John", "Doe") *# Output: John Doe, how are you today?*

**B. Using Keyword Arguments**

Explicitly naming arguments improves readability:

python

greet(first\_name="Jane", last\_name="Doe")

**5. Returning Values**

Functions can return results instead of printing them directly.

**Example 1: Returning a Greeting**

python

**def** greet(first\_name, last\_name):

**return** f"{first\_name} {last\_name}, how are you today?"

message = greet("John", "Doe")

**print**(message) *# Output: John Doe, how are you today?*

**Example 2: Calculating Age**

python

**def** calculate\_age(birth\_year, current\_year):

age = current\_year - birth\_year

**return** age

*# Calling the function*

age = calculate\_age(1990, 2025)

**print**(f"Age: {age}") *# Output: Age: 35*

**6. Key Concepts**

| **Concept** | **Description** |
| --- | --- |
| **Parameters** | Inputs defined in the function (e.g., birth\_year). |
| **Arguments** | Actual values passed during a function call (e.g., 1990). |
| **return** | Sends a value back to the caller. If omitted, the function returns None. |
| **Scope** | Variables inside a function are local and not accessible outside it. |

**7. Best Practices**

1. **Descriptive Names:** Use names like calculate\_area instead of func1.
2. **Modularity:** Break complex tasks into smaller functions.
3. **Docstrings:** Add comments to explain the function’s purpose.

python

**def** greet(first\_name, last\_name):

"""Returns a greeting message for the given name."""

**return** f"Hello, {first\_name} {last\_name}!"

**8. Common Mistakes**

* **Missing Colons:** Forgetting : after the function definition.
* **Incorrect Indentation:** Code not indented under def.
* **Unused Parameters:** Defining parameters but not using them.

LECTURE EIGHT

**Python Functions (Part 2)**

**1. Overview**

This lesson expands on Python functions, focusing on:

* Using loops (while, for) inside functions.
* Calculating the **Least Common Multiple (LCM)**.
* Checking if a number is **prime**.
* Nesting functions (functions inside functions).
* Practical use of the **modulus operator %**.
* Emphasizing hands-on practice for mastery.

**2. Finding the Least Common Multiple (LCM)**

**Concept**

The LCM of two integers is the **smallest positive integer divisible by both**.  
**Example:**

* LCM(3, 5) = 15
* LCM(2, 24) = 24

**Algorithm**

1. Start with the **greater number** between the two inputs.
2. Use a while True loop to check divisibility:
   * If the current number is divisible by both inputs, return it as the LCM.
   * Otherwise, increment by 1 and repeat.

**Code Implementation**

python

**def** get\_lcm(num1, num2):

greater = max(num1, num2)

**while** True:

**if** greater % num1 == 0 **and** greater % num2 == 0:

**return** greater

greater += 1

*# Example Usage*

**print**(get\_lcm(3, 5)) *# Output: 15*

**Key Notes**

* The loop starts at the larger number to minimize iterations.
* Use % to check divisibility (remainder = 0).
* Avoid breaking the loop prematurely; let it run until the LCM is found.

**3. Prime Number Check**

**Definition**

A **prime number** is a number > 1 that has no divisors other than 1 and itself.  
**Examples:**

* 3 is prime (divisible by 1 and 3).
* 4 is not prime (divisible by 1, 2, 4).

**Function Logic**

1. If the number ≤ 1, return False.
2. Use a for loop to check divisors from 2 to (number – 1).
3. If any divisor is found, the number is **not prime** (return False).
4. If no divisors are found, the number is **prime** (return True).

**Code Implementation**

python

**def** check\_prime(number):

**if** number <= 1:

**return** False

**for** n **in** range(2, number):

**if** number % n == 0:

**return** False

**return** True

*# Example Usage*

**print**(check\_prime(13)) *# Output: True*

**Efficiency Note**

* For simplicity, the loop runs up to (number – 1).
* Optimized approach: Check up to √number (e.g., range(2, int(number\*\*0.5) + 1)).

**4. Modulus Operator %**

**Purpose**

* Returns the **remainder** after division.
* Critical for checking divisibility.

**Examples**

* 6 % 2 = 0 → 6 is divisible by 2.
* 7 % 2 = 1 → 7 is not divisible by 2.

**Use Cases**

* **LCM:** Check if a number is divisible by both inputs.
* **Prime Check:** Determine if a number has factors.

**5. Nested Functions**

Functions can be defined inside other functions for modularity.

**Example**

python

**def** outer\_function():

**def** inner\_function():

**print**("This is a nested function.")

inner\_function()

outer\_function() *# Output: This is a nested function.*

**6. Programming Mindset & Best Practices**

1. **Practice Relentlessly:**
   * Code daily to internalize concepts.
2. **Embrace Failure:**
   * Errors are learning opportunities.
3. **Avoid Overthinking:**
   * Start with simple implementations; optimize later.
4. **Incremental Learning:**
   * Build on previous lessons (e.g., loops → functions).

LECTURE NINE

**Python Lambda Functions**

**1. What is a Lambda Function?**

* **Lambda functions** are also known as **anonymous functions** or **nameless functions** in Python.
* They are small, one-line functions defined without using the standard **def** keyword.
* Lambda functions are used for quick, simple operations where a full function definition would be unnecessarily verbose.
* They are often used as arguments to higher-order functions (like map(), filter(), and sorted()), or for short, throwaway operations.

**2. Syntax of a Lambda Function**

python

**lambda** parameters: expression

* **lambda** is the keyword that creates the function.
* **parameters**: One or more input variables (comma-separated).
* **expression**: A single expression whose result is automatically returned.

**Key Points:**

* Lambda functions can take any number of arguments, but only **one expression**.
* The result of the expression is returned implicitly (no need for a return statement).
* Lambda functions cannot contain multiple statements or commands (no loops, no assignments).

**3. Comparison: Normal Function vs. Lambda Function**

**Normal Function:**

python

**def** add(a, b):

**return** a + b

**print**(add(3, 5)) *# Output: 8*

**Equivalent Lambda Function:**

python

add = **lambda** a, b: a + b

**print**(add(3, 5)) *# Output: 8*

* Here, add is a variable that refers to the lambda function.
* The lambda function takes two inputs (a and b) and returns their sum.

**4. Common Use Cases for Lambda Functions**

**A. Using Lambda with map()**

Applies a function to every item in an iterable.

**python**

numbers = [1, 2, 3, 4]

squared = list(map(**lambda** x: x \*\* 2, numbers))

**print**(squared) *# Output: [1, 4, 9, 16]*

**B. Using Lambda with filter()**

Filters items in an iterable based on a condition.

**python**

numbers = [1, 2, 3, 4, 5, 6]

even = list(filter(**lambda** x: x % 2 == 0, numbers))

**print**(even) *# Output: [2, 4, 6]*

**C. Using Lambda with sorted()**

Custom sorting based on a key.

python

names = ["Alice", "Bob", "Charlie", "David"]

sorted\_names = sorted(names, key=**lambda** x: len(x))

**print**(sorted\_names) *# Output: ['Bob', 'Alice', 'David', 'Charlie']*

**5. More Lambda Examples**

**A. Lambda with No Parameters**

python

greet = **lambda**: "Hello, World!"

**print**(greet()) *# Output: Hello, World!*

**B. Lambda with One Parameter**

python

square = **lambda** x: x \* x

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

**C. Lambda with Multiple Parameters**

python

multiply = **lambda** a, b, c: a \* b \* c

**print**(multiply(2, 3, 4)) *# Output: 24*

**6. Limitations of Lambda Functions**

* **Single Expression Only:** Cannot contain multiple statements, assignments, or complex logic.
* **Readability:** Overusing lambda functions can make code harder to understand.
* **No Name:** Lambdas are anonymous unless assigned to a variable, so debugging can be trickier.

**7. When to Use Lambda Functions**

* When you need a simple, short function for a limited purpose.
* As arguments to functions that expect a function (like map, filter, sorted, etc.).
* When defining a function in-place is more readable than defining it elsewhere with def.

**When NOT to use:**

* For complex operations (use def instead for clarity).

**8. Practice Exercises**

1. **Write a lambda function to cube a number.**

python

cube = **lambda** x: x \*\* 3

**print**(cube(3)) *# Output: 27*

1. **Use lambda with filter to get all numbers divisible by 3 in a list.**

python

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

divisible\_by\_3 = list(filter(**lambda** x: x % 3 == 0, numbers))

**print**(divisible\_by\_3) *# Output: [3, 6, 9]*

1. **Sort a list of tuples by the second item using lambda.**

python

pairs = [(1, 3), (2, 2), (4, 1)]

sorted\_pairs = sorted(pairs, key=**lambda** x: x[1])

**print**(sorted\_pairs) *# Output: [(4, 1), (2, 2), (1, 3)]*

LECTURE TEN

**Exception Handling in Python**

**1. Introduction to Exceptions**

An **exception** is an error that occurs during program execution, often due to invalid input or unexpected conditions. Exceptions disrupt normal program flow and can cause crashes if unhandled.

**Examples of Common Exceptions:**

* ZeroDivisionError: Dividing by zero.
* ValueError: Invalid input (e.g., converting a string to an integer).
* TypeError: Incorrect data type (e.g., adding a string and an integer).
* FileNotFoundError: Accessing a non-existent file.

**2. Why Handle Exceptions?**

* **Prevent crashes:** Users might enter invalid data (e.g., text instead of numbers).
* **Improve user experience:** Provide clear error messages instead of cryptic tracebacks.
* **Maintain program flow:** Allow the program to recover gracefully and continue running.

**3. Basic Syntax of Exception Handling**

Use **try** and **except** blocks to catch and handle exceptions:

python

**try**:

*# Code that might raise an exception*

**except** ExceptionType:

*# Code to handle the exception*

**Example: Handling ZeroDivisionError**

python

**try**:

result = 5 / 0

**except** ZeroDivisionError:

**print**("Error: Cannot divide by zero!")

result = 0 *# Assign a default value*

**print**("Result:", result) *# Output: Result: 0*

**4. Handling Multiple Exceptions**

Catch specific exceptions and handle them differently:

python

**try**:

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

result = 10 / num

**except** ZeroDivisionError:

**print**("Error: Division by zero!")

**except** ValueError:

**print**("Error: Invalid input (not a number)!")

**5. The else and finally Clauses**

* **else:** Runs if no exceptions occur.
* **finally:** Always runs, regardless of exceptions (useful for cleanup tasks).

python

**try**:

file = open("data.txt", "r")

**except** FileNotFoundError:

**print**("Error: File not found!")

**else**:

**print**("File content:", file.read())

file.close()

**finally**:

**print**("Execution complete.")

**6. Example: Input Validation**

python

**while** True:

**try**:

age = int(input("Enter your age: "))

**break**

**except** ValueError:

**print**("Invalid input! Enter a number.")

**print**("Your age is:", age)

**7. Common Exceptions in Python**

| **Exception** | **Cause** |
| --- | --- |
| ZeroDivisionError | Dividing by zero. |
| ValueError | Invalid argument type (e.g., int("text")). |
| TypeError | Incompatible operation (e.g., "5" + 3). |
| FileNotFoundError | Accessing a file that doesn’t exist. |
| IndexError | Accessing a list index out of range. |
| KeyError | Accessing a non-existent dictionary key. |

**8. Best Practices**

1. **Be specific:** Catch specific exceptions (e.g., except ValueError instead of a generic except).
2. **Avoid bare except:** Always specify the exception type to avoid masking errors.
3. **Use finally for cleanup:** Close files or release resources here.
4. **Provide helpful messages:** Guide users to correct their input.

**Practice Exercise:**  
Write a program that asks for two numbers, divides them, and handles both ZeroDivisionError and ValueError.

python

**try**:

num1 = int(input("Enter first number: "))

num2 = int(input("Enter second number: "))

result = num1 / num2

**except** ZeroDivisionError:

**print**("Error: Cannot divide by zero!")

**except** ValueError:

**print**("Error: Invalid input (not a number)!")

**else**:

**print**(f"{num1} / {num2} = {result}")

**Key Takeaway:**  
Exception handling ensures your program remains robust and user-friendly, even when unexpected errors occur.

LECTURE ELEVEN

**Python List Comprehension and Functions with Unlimited Arguments**

**1. Overview of Topics**

* **List Comprehension:** A concise, Pythonic way to create and transform lists.
* **Functions with Unlimited Arguments (\*args):** Flexible functions that accept any number of positional arguments.
* **Brief mentions:** Exception handling, lambda functions.

**2. Functions with Unlimited Arguments (\*args)**

**Concept**

* Sometimes, you want to write a function that can accept any number of arguments.
* In Python, this is done using \*args in the function definition.
* All extra positional arguments passed to the function are collected into a tuple named args (or any name you choose).

**Syntax Example**

python

**def** function\_name(\*args):

*# args is a tuple of all positional arguments*

**pass**

**Example: Average Function**

python

**def** average(\*nums):

nums = list(nums) *# Optional: convert to list for modification*

**return** sum(nums) / len(nums)

**print**(average(3, 4, 5, 6, 7)) *# Output: 5.0*

**print**(average(1, 2)) *# Output: 1.5*

**Key Points:**

* \*nums collects all arguments into a tuple.
* Converting to a list is only necessary if you need to modify the sequence.
* You can pass any number of arguments; the function will handle them all.

**3. Introduction to List Comprehension**

**What is List Comprehension?**

* A **concise way to create lists** by applying an expression to each item in an iterable (like a list or range).
* It can include conditional logic (if statements).
* It is more readable and efficient than traditional for-loops.

**General Syntax**

python

[expression **for** item **in** iterable **if** condition]

**4. Traditional Filtering: Even Numbers Example**

**Traditional Approach:**

python

numbers = list(range(1, 11))

evens = []

**for** n **in** numbers:

**if** n % 2 == 0:

evens.append(n)

**print**(evens) *# Output: [2, 4, 6, 8, 10]*

**5. Equivalent List Comprehension for Even Numbers**

python

numbers = list(range(1, 11))

evens = [x **for** x **in** numbers **if** x % 2 == 0]

**print**(evens) *# Output: [2, 4, 6, 8, 10]*

**Explanation:**

* x iterates through each element in numbers.
* if x % 2 == 0 filters only even numbers.
* The result is a new list with only even numbers.

**6. List Comprehension with If-Else Logic**

You can include an **if-else** statement inside the list comprehension to transform items conditionally.

**Syntax:**

python

[<true\_value> **if** <condition> **else** <false\_value> **for** item **in** iterable]

**Example:**

python

results = ['ODD' **if** x % 2 != 0 **else** 'EVEN' **for** x **in** numbers]

**print**(results)

*# Output: ['ODD', 'EVEN', 'ODD', 'EVEN', 'ODD', 'EVEN', 'ODD', 'EVEN', 'ODD', 'EVEN']*

**Key Points:**

* The if-else comes before the for loop in the comprehension.
* This allows you to perform different transformations based on the condition.

**7. List Comprehension for Simple Transformations**

**Example: Multiply every number by 2**

python

doubled = [x \* 2 **for** x **in** numbers]

**print**(doubled) *# Output: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]*

**8. Exercise Examples Using List Comprehension**

**8.1 Find Multiples of 4 between 1 and 40**

python

multiples\_of\_4 = [x **for** x **in** range(1, 41) **if** x % 4 == 0]

**print**(multiples\_of\_4) *# Output: [4, 8, 12, 16, 20, 24, 28, 32, 36, 40]*

**8.2 Square All Odd Numbers between 1 and 20**

python

squares\_of\_odds = [x\*\*2 **for** x **in** range(1, 21) **if** x % 2 != 0]

**print**(squares\_of\_odds) *# Output: [1, 9, 25, 49, 81, 121, 169, 225, 289, 361]*

**8.3 Label Numbers as "High" or "Low" Based on Value**

python

labels = ['High' **if** x > 5 **else** 'Low' **for** x **in** range(1, 11)]

**print**(labels) *# Output: ['Low', 'Low', 'Low', 'Low', 'Low', 'Low', 'High', 'High', 'High', 'High']*

**10. Key Takeaways**

* **List comprehensions** make your code more concise and readable.
* **\*args** lets you write functions that accept any number of positional arguments.
* Use list comprehensions for filtering, mapping, and conditional transformations.
* Practice writing both traditional loops and list comprehensions to understand their advantages.

LECTURE TWELVE

**DS PY OOP PART 1**

**Introduction to Object-Oriented Programming (OOP)**

**What is OOP?**

* **Object-Oriented Programming (OOP)** is a programming paradigm where everything is modeled as **objects**.
* Objects combine **data** (attributes) and **behavior** (methods/functions).
* In Python (and many other languages), **almost everything is an object**, including variables.

**Why Use OOP?**

* Helps you become a **better programmer** by thinking in terms of real-world entities.
* Organizes code into **manageable, reusable pieces** (classes and objects).
* Makes code **structured, easier to maintain, debug, and extend**.
* Encourages **encapsulation** — bundling data and methods that operate on that data within one unit.

**Key Concepts and Terminology**

| **Term** | **Definition** |
| --- | --- |
| **Class** | A blueprint or template for creating objects. Represents a real-world entity or concept. |
| **Object** | An instance of a class with specific attributes and behaviors. |
| **Attributes** | Data or characteristics of an object (e.g., color, model, age). |
| **Methods** | Functions defined inside a class that describe behaviors (e.g., start(), bark()). |
| **Constructor** (\_\_init\_\_) | Special method to initialize an object's attributes when it is created. |
| **Instance** | A unique object created from a class. |

**Class Structure in Python**

python

**class** ClassName:

**def** \_\_init\_\_(self, parameters):

self.attribute1 = value1

self.attribute2 = value2

*# Initialize other attributes*

**def** method1(self):

*# Code for behavior 1*

**def** method2(self, param):

*# Code for behavior 2*

* **Class names** start with a capital letter by convention.
* The first parameter of all methods is self, which refers to the current instance of the class.
* The constructor method is always named \_\_init\_\_ with double underscores before and after.
* Attributes are defined inside \_\_init\_\_ using self.attribute = value.

**Example 1: Dog Class**

**Attributes:**

* name
* color
* age

**Method:**

* bark(): prints a message showing the dog is barking.

python

**class** Dog:

**def** \_\_init\_\_(self, name, color, age):

self.name = name

self.color = color

self.age = age

**def** bark(self):

**print**(f"{self.name} is barking")

*# Creating instances (objects)*

dog1 = Dog("Terry", "Black", 5)

dog2 = Dog("Jack", "Red", 3)

dog1.bark() *# Output: Terry is barking*

dog2.bark() *# Output: Jack is barking*

**Benefits:**

* Easily create multiple dog objects without rewriting code.
* Encapsulates behavior (bark) and attributes in one class.
* Demonstrates how objects are instances of a class.

**Example 2: Car Class (Detailed)**

**Attributes:**

* maker (manufacturer)
* model
* year
* color
* speed (starts at 0)
* engine\_on (engine state, default off)

**Methods (Behaviors):**

* start(): Turns the engine on if it was off and speed is 0.
* accelerate(d=1): Increases speed based on gear d (default gear 1).
* brake(intensity="soft"): Applies brake, reducing speed by intensity.
* park(): Stops the car and turns off engine if speed is 0.
* kick(): Optional alias for starting the car.

**Important Rules:**

* Engine must be off and speed 0 to start.
* Can only accelerate if engine is on.
* Brake reduces speed:
  + Soft brake reduces speed by 50%.
  + Hard brake brings speed to 0.
* To park, car must not be moving.

**Sample Code**

python

**class** Car:

**def** \_\_init\_\_(self, maker, model, year, color):

self.maker = maker

self.model = model

self.year = year

self.color = color

self.speed = 0

self.engine\_on = False

**def** start(self):

**if** **not** self.engine\_on **and** self.speed == 0:

self.engine\_on = True

**print**(f"{self.maker} {self.model} is starting.")

**else**:

**print**(f"Cannot start {self.maker} {self.model}; engine is already on or car is moving.")

**def** accelerate(self, d=1):

**if** **not** self.engine\_on:

**print**("Start the car first!")

**return**

**if** d == 1:

self.speed += 50

**elif** d == 2:

self.speed += 100

**else**:

**print**("Invalid gear; no acceleration.")

**return**

**print**(f"{self.maker} {self.model} is moving at {self.speed} mph.")

**def** brake(self, intensity="soft"):

**if** self.speed <= 0:

**print**(f"{self.maker} {self.model} has already stopped.")

**return**

intensity = intensity.lower()

**if** intensity == "soft":

self.speed -= self.speed \* 0.5 *# Reduce speed by 50%*

**print**(f"{self.maker} {self.model} is braking softly, speed now {self.speed} mph.")

**elif** intensity == "hard":

self.speed = 0

**print**(f"{self.maker} {self.model} has stopped.")

**else**:

**print**("Unknown brake intensity.")

**def** park(self):

**if** self.speed > 0:

**print**("Cannot park while the car is moving!")

**elif** self.engine\_on:

self.engine\_on = False

**print**(f"{self.maker} {self.model} is parked and engine is off.")

**else**:

**print**(f"{self.maker} {self.model} is already parked.")

**def** kick(self):

*# Alias for start*

self.start()

*# Creating an instance*

my\_car = Car("Lexus", "RX 350", 2018, "Black")

*# Usage example*

my\_car.start() *# Starts the car if possible*

my\_car.accelerate() *# Accelerate at default gear 1 (50 mph)*

my\_car.brake() *# Soft brake, reduces speed by 50%*

my\_car.park() *# Attempts to park the car*

**Summary of OOP Benefits in These Examples**

* **Encapsulation**: Data (attributes) and behavior (methods) are bundled inside classes.
* **Reusability**: Classes can be reused to create many objects without rewriting code.
* **Abstraction**: Users interact with objects through methods without worrying about internal details.
* **Modularity**: Code is organized into logical units (classes), easier to maintain and extend.

LECTURE THIRTHEEN

**Object-Oriented Programming in Python (Advanced Concepts)**

This lesson builds upon foundational OOP concepts in Python by demonstrating how to create a simple **Calculator** class with practical features such as constructors, methods, exception handling, state management, and user interaction. Below is a detailed, comprehensive guide covering all key points and implementations.

**1. Creating a Class in Python**

* Use the class keyword to define a class.
* Classes are blueprints for creating objects (instances).
* Example:

python

**class** Calculator:

**pass** *# Placeholder for class body*

**2. Constructor (\_\_init\_\_ method)**

* Special method called automatically when an instance is created.
* Used to initialize instance attributes.
* Not mandatory but essential when you want to set initial state or properties.
* For the Calculator class, we initialize an empty list history to track operations.

**Example:**

python

**class** Calculator:

**def** \_\_init\_\_(self):

self.history = [] *# Initialize history list to keep track of operations*

**3. Defining Methods for Calculator Operations**

The Calculator supports four basic arithmetic operations:

* **add(a, b)**: Returns the sum of a and b.
* **minus(a, b)**: Returns the difference (a - b).
* **multiply(a, b)**: Returns the product.
* **divide(a, b)**: Returns the quotient, with special handling for division by zero.

Each method:

* Computes the result.
* Appends a descriptive string to self.history.
* Returns the result.

**Example:**

python

**def** add(self, a, b):

result = a + b

self.history.append(f"{a} + {b} = {result}")

**return** result

**4. Handling Division & ZeroDivisionError**

* Division by zero causes a runtime error.
* To avoid this, check if divisor b is zero before division.
* If zero, return a friendly message instead of raising an error.

**Example:**

python

**def** divide(self, a, b):

**if** b == 0:

self.history.append(f"{a} / {b} = Cannot divide by zero")

**return** "Cannot divide by zero"

**else**:

result = a / b

self.history.append(f"{a} / {b} = {result}")

**return** result

**5. Maintaining Operation History**

* Use an instance attribute self.history (a list) to store all performed operations.
* Each method appends a string describing the operation and its result.
* This allows tracking and reviewing past calculations.

**6. Implementing a log Method to Retrieve History**

* The log method returns the stored history.
* It accepts an optional parameter show (default 0):
  + If show == 0: Return the **entire** history list.
  + If show > 0: Return the **last show entries** from history.

**Example:**

python

**def** log(self, show=0):

**if** show == 0:

**return** self.history

**else**:

**return** self.history[-show:]

**7. Clearing History with Confirmation**

* The clear method interacts with the user to confirm clearing history.
* Steps:
  + Prompt user with input: "Are you sure you want to clear history? (yes/no): "
  + If user inputs "yes" (case-insensitive): clear self.history and print confirmation.
  + If "no": do nothing.
  + For any other input: print "Wrong input".

**Example:**

python

**def** clear(self):

decision = input("Are you sure you want to clear history? (yes/no): ").lower()

**if** decision == "yes":

self.history.clear()

**print**("History cleared")

**elif** decision == "no":

**pass**

**else**:

**print**("Wrong input")

**Complete Example: Calculator Class with All Features**

python

**class** Calculator:

**def** \_\_init\_\_(self):

self.history = []

**def** add(self, a, b):

result = a + b

self.history.append(f"{a} + {b} = {result}")

**return** result

**def** minus(self, a, b):

result = a - b

self.history.append(f"{a} - {b} = {result}")

**return** result

**def** multiply(self, a, b):

result = a \* b

self.history.append(f"{a} \* {b} = {result}")

**return** result

**def** divide(self, a, b):

**if** b == 0:

self.history.append(f"{a} / {b} = Cannot divide by zero")

**return** "Cannot divide by zero"

**else**:

result = a / b

self.history.append(f"{a} / {b} = {result}")

**return** result

**def** log(self, show=0):

**if** show == 0:

**return** self.history

**else**:

**return** self.history[-show:]

**def** clear(self):

decision = input("Are you sure you want to clear history? (yes/no): ").lower()

**if** decision == "yes":

self.history.clear()

**print**("History cleared")

**elif** decision == "no":

**pass**

**else**:

**print**("Wrong input")