Unit I: -

2022-23(CT)

1. Explain any two sequential data types in python with proper example.

Ans:-

1. **Lists**:
   - A list is an ordered collection of items, where each item can be of any data type (e.g., numbers, strings, other lists, etc.).
   - Key characteristics:
     - **Ordered**: The order of elements in a list matters.
     - **Mutable**: You can modify the elements of a list after its creation.
     - **Indexed**: You can access individual elements using their index (starting from 0).
     - **Dynamic size**: Lists can grow or shrink as needed during program execution.
   - Example:

     ```
     # Creating a list of programming languages
     programming_languages = ["Python", "Java", "JavaScript", "C++", "Ruby"]

     # Accessing elements by index
     print(programming_languages[0])  # Output: Python
     print(programming_languages[2])  # Output: JavaScript
     ```
2. **Tuples**:
   - A tuple is similar to a list, but with a few key differences:
     - **Immutable**: Once you create a tuple, you cannot change its elements.
     - **Used for fixed collections**: Tuples are often used to represent related values that shouldn't be modified.
   - Example:

     ```
     # Creating a tuple of RGB color values
     white = (255, 255, 255)
     black = (0, 0, 0)

     # Accessing elements by index
     print(white[0])  # Output: 255
     print(black[1])  # Output: 0
     ```

Remember that choosing between lists and tuples depends on your specific use case. If you need a collection that can be modified, use a list. If you want to ensure immutability, use a tuple. Both data types are powerful tools for organizing and managing data in Python! If you have any more questions or need further clarification, feel free to ask! □□□ .

2. Write a program to find largest of three numbers without using max() function.

Ans:
```
n1, n2, n3 = input("Enter Three numbers:").split()

n1 = int(n1)
n2 = int(n2)
n3 = int(n3)
lar = ((n1+n2)+abs(n1-n2))//2
lar = ((lar+n3)+abs(lar-n3))//2
sml = ((n1+n2)-abs(n1-n2))//2
sml = ((sml+n3)-abs(sml-n3))//2
print(lar)
print(sml)
```

3. With suitable examples explain the membership and identity operators.

Ans:

Membership Operators:

Python provides two membership operators to check or validate the membership of a value within a sequence (such as strings, lists, or tuples):

1. **`in` Operator:**
   - The `in` operator checks if a character, substring, or element exists in a sequence.
   - It evaluates to `True` if it finds the specified element in the sequence; otherwise, it returns `False`.
   - Examples:
     - `'G' in 'GeeksforGeeks'` evaluates to `True`.
     - `'g' in 'GeeksforGeeks'` (since Python is case-sensitive) evaluates to `False`.
     - `'Geeks' in ['Geeks', 'For', 'Geeks']` evaluates to `True`.
     - `10 in [10000, 1000, 100, 10]` evaluates to `True`.
     - `3 in {1: 'Geeks', 2: 'For', 3: 'Geeks'}` evaluates to `True`.
   - The execution speed of the `in` operator depends on the target object's type. For lists, the average time complexity is O(n), while for sets, it's O(1).
2. **`not in` Operator:**
   - The `not in` operator evaluates to `True` if it does not find a variable in the specified sequence and `False` otherwise.
   - Examples:
     - `x = 24, y = 20`
       - `x not in [10, 20, 30, 40, 50]` evaluates to `True`.
       - `y in [10, 20, 30, 40, 50]` evaluates to `True`.

Identity Operators:

Identity operators compare objects based on their data type and memory location. They help determine if two objects are the same object in memory:

1. **`is` Operator:**
   - The `is` operator evaluates to `True` if the variables on either side of the operator point to the same object in memory.
   - Example:
   - `x = 5`
   - `y = 5`
   - `print(x is y)  # Output: True`
2. **`is not` Operator:**
   - The `is not` operator evaluates to `True` if the variables do not point to the same object in memory.
   - Example:
   - `a = [1, 2, 3]`
   - `b = a`
   - `print(a is b)  # Output: True`

Remember that identity operators are useful for comparing objects' memory locations, while membership operators check if a value exists within a sequence. If you have any further questions or need more examples, feel free to ask!

28 Jul 2022

1. What is variable? What are the rules and conventions for declaring a variable?

Ans: A **variable** in programming is a named storage location that holds a value or data. Variables are essential for storing and manipulating data within computer programs. Here are some key points about variables:

1. **Named Storage Location**: A variable has a name (also called an **identifier**) that allows us to refer to it independently of the actual data it holds.
2. **Value**: The value stored in a variable can change during the execution of a program. Variables act as placeholders for data.
3. **Declaration of Variables**:
   - When declaring a variable, we specify its **type** and **name** before using it in the program.
   - The syntax for declaring variables varies across programming languages, but the fundamental concept remains consistent.
4. **Initialization of Variables**:
   - Initialization involves assigning an initial value to a declared variable.
   - The syntax for variable initialization also varies based on the programming language.
5. **Types of Variables**:

   - **Global Variables**: Declared outside any function or block, accessible throughout the entire codebase.
   - **Local Variables**: Declared within a specific function, block, or scope, and only accessible within that context.
6. **Naming Conventions**:
   - Follow these conventions when naming variables:
     - Start with a letter or underscore.
     - Avoid starting with a number.
     - Use alphanumeric characters and underscores (A-z, 0-9, and _).
     - Be case-sensitive (e.g., `name`, `Name`, and `NAME` are different variables).
     - Avoid using reserved words (keywords) as variable names.

Here's an example of declaring and initializing variables in Python:

```
# Valid variable names
geeks = 1
Geeks = 2
Ge_e_ks = 5
_geeks = 6
geeks_ = 7
_GEEKS_ = 8

print(geeks, Geeks, Ge_e_ks)
print(_geeks, geeks_, _GEEKS_)
```

Remember that variables play a crucial role in programming, allowing us to store and manipulate data effectively. If you have any further questions or need more examples, feel free to ask! ☐

2. ExplaFin in detail the identity and membership operators in python with appropriate example.
3. What is the output of the following expressions
   a) 54//17
   b) -17//10
   c) 2==3-1
   d) 4+5*3//5
   e) [1,2,3,4,5][-1]
   f) (1,2,3)*3

10 Nov 2022

1. What are list in python? How to define and access the elements of list?

Ans: A **list** in Python is an ordered collection of items (or elements). It allows you to store multiple values in a single variable. Lists are versatile and can hold elements of different data types, such as numbers, letters, strings, and even nested lists.

Here are some key points about Python lists:

1. **Creating a List**:
    - You can create a list by enclosing elements inside square brackets (`[]`), separated by commas.
    - Example:
    - `# Creating a list of fruits`
    - `fruits = ["apple", "banana", "cherry"]`
2. **Accessing List Items**:
    - List items are indexed, starting from 0 (zero).
    - To access an item in a list, use the index operator (`[]`) followed by the index number.
    - Example:
    - `# Accessing the second item (index 1) in the list`
    - `print(fruits[1])  # Output: "banana"`
3. **Negative Indexing**:
    - Negative indexing starts from the end of the list (-1 refers to the last item, -2 refers to the second-to-last item, and so on).
    - Example:
    - `# Accessing the last item using negative index`
    - `print(fruits[-1])  # Output: "cherry"`
4. **Slicing (Range of Indexes)**:
    - You can specify a range of indexes to retrieve a sublist.
    - The syntax is `list[start:end]`, where `start` is inclusive, and `end` is exclusive.
    - Example:
    - `# Retrieving a sublist from index 2 (inclusive) to 5 (exclusive)`
    - `print(fruits[2:5])  # Output: ["cherry", "orange", "kiwi"]`
5. **Omitting Start or End Value**:
    - If you omit the `start` value, it defaults to the beginning of the list.
    - If you omit the `end` value, it defaults to the end of the list.
    - Example:
    - `# Retrieving items from the beginning up to index 4 (not including "kiwi")`
    - `print(fruits[:4])  # Output: ["apple", "banana", "cherry", "orange"]`
    -
    - `# Retrieving items from index 2 to the end`
    - `print(fruits[2:])  # Output: ["cherry", "orange", "kiwi", "melon", "mango"]`
6. **Checking Item Existence**:
    - To check if a specific item exists in a list, use the `in` keyword.
    - Example:
    - `if "apple" in fruits:`
    - `    print("Yes, 'apple' is in the fruits list")`

Remember that lists are powerful data structures in Python, allowing you to store, manipulate, and iterate over collections of data. If you have any more questions or need further clarification, feel free to ask! □

Sources:

1. W3Schools - Python Lists - Access List Items
2. Python Tutorial - Python List
3. GeeksforGeeks - Python Lists
4. Stack Overflow - How to access List elements[1][2][3][4]

2. Discuss the int(), float(), str() type conversion functions with examples.

Ans: Certainly! Let's explore the **type conversion functions** in Python: `int()`, `float()`, and `str()`. These functions allow you to convert data from one type to another. Here are examples of how each function works:

1. **`int()` Function**:
   - The `int()` function converts a value to an integer (whole number).
   - It can convert a string containing a valid integer or a floating-point number to an integer.
   - Example:
   - `number_str = "123"`
   - `integer_number = int(number_str)`
   - `print(integer_number)  # Output: 123`
2. **`float()` Function**:
   - The `float()` function converts a value to a floating-point number (decimal number).
   - It can convert an integer or a string containing a valid float to a float.
   - Example:
   - `float_str = "3.14"`
   - `float_number = float(float_str)`
   - `print(float_number)  # Output: 3.14`
3. **`str()` Function**:
   - The `str()` function converts a value to a string.
   - It can convert an integer, a float, or any other data type to a string.
   - Example:
   - `x = 2`
   - `x_str = str(x)`
   - `print(x_str)  # Output: "2"`

Remember that these type conversion functions are useful when you need to manipulate data of different types or when interacting with user input. If you have any more questions or need further clarification, feel free to ask! □

Sources:

1. Python Type Casting (int(), float() & str() functions) - Electrical Workbook
2. Understanding Python’s int() and float() Functions - Medium
3. Plotting and Programming in Python: Data Types and Type Conversion
4. Understanding Data Type Conversion (Type Casting) in Python
5. Type Conversion in Python - GeeksforGeeks[12345]

3. Write a python program to convert height in feet and inches to cm. [1 feet = 12 inch and 1 inch = 2.54 cm] (Sample input: 2 feet 7 inch Sample output: 78.74cm)

Ans: Certainly! Let’s write a Python program to convert height from feet and inches to centimeters. We’ll follow the given conversion factors:

- 1 foot = 12 inches
- 1 inch = 2.54 cm

Here’s the program:

```
def feet_and_inches_to_cm(feet, inches):
    # Convert feet to inches and add to the given inches
    total_inches = feet * 12 + inches

    # Convert total inches to centimeters
    cm = total_inches * 2.54

    return cm

# Sample input: 2 feet 7 inches
feet_input = 2
inches_input = 7

# Calculate the result
height_in_cm = feet_and_inches_to_cm(feet_input, inches_input)

# Display the result
print(f"{feet_input} feet {inches_input} inches is approximately
{height_in_cm:.2f} cm.")
```

When you run this program with the sample input (2 feet 7 inches), it will output:

```
2 feet 7 inches is approximately 78.74 cm.
```

Feel free to modify the input values as needed. If you have any more questions or need further assistance, feel free to ask! □

15 Jul 2023

1. What are lists in python? What is the difference between the list methods *append and extend*? Explain with suitable example.

Ans: A **list** in Python is an ordered collection of items (or elements). It allows you to store multiple values in a single variable. Lists are versatile and can hold elements of different data types, such as numbers, letters, strings, and even nested lists.

Here are some key points about Python lists:

1. **Creating a List**:
    - You can create a list by enclosing elements inside square brackets (`[]`), separated by commas.
    - Example:
    - `# Creating a list of fruits`
    - `fruits = ["apple", "banana", "cherry"]`
2. **Accessing List Items**:
    - List items are indexed, starting from 0 (zero).
    - To access an item in a list, use the index operator (`[]`) followed by the index number.
    - Example:
    - `# Accessing the second item (index 1) in the list`
    - `print(fruits[1])  # Output: "banana"`
3. **Difference Between `append()` and `extend()`**:
    - `.append()` adds a single element to the end of the list.
    - `.extend()` appends multiple elements from an iterable (such as another list) to the end of the list.
    - Examples:
    - `# Using append()`
    - `my_list = [1, 2, 3]`
    - `my_list.append([4, 5])`
    - `print(my_list)  # Output: [1, 2, 3, [4, 5]]`
    -
    - `# Using extend()`
    - `my_list = [1, 2, 3]`
    - `my_list.extend([4, 5])`
    - `print(my_list)  # Output: [1, 2, 3, 4, 5]`
    - Note that `.extend()` modifies the original list, whereas `.append()` adds a single item (which could be a list itself) to the end of the list.

Remember that lists are powerful data structures in Python, allowing you to store, manipulate, and iterate over collections of data. If you have any more questions or need further clarification, feel free to ask! □

Sources:

1. W3Schools - Python Lists
2. Stack Overflow - What is the difference between Python’s list methods append and extend?

3. Finxter - Python List append() vs extend()
4. GeeksforGeeks - append() and extend() in Python
5. FreeCodeCamp - Python List Append VS Python List Extend[1][2][3][4][5]

2. Explain python logical operators with suitable example.

Ans: Certainly! In Python, **logical operators** are used to combine or manipulate boolean expressions. These operators allow you to make decisions based on the truth values of multiple conditions. Let's explore the three main logical operators: `and`, `or`, and `not`, along with suitable examples:

1. **Logical AND (`and`)**:
    - The `and` operator returns `True` if both operands (expressions) are true; otherwise, it returns `False`.
    - Example:
    - `x = 5`
    - `y = 10`
    - `result = (x > 2) and (y < 15)`
    - `print(result)  # Output: True`
2. **Logical OR (`or`)**:
    - The `or` operator returns `True` if at least one of the operands is true; otherwise, it returns `False`.
    - Example:
    - `a = 20`
    - `b = 5`
    - `result = (a > 15) or (b > 10)`
    - `print(result)  # Output: True`
3. **Logical NOT (`not`)**:
    - The `not` operator negates the truth value of an expression.
    - It returns `True` if the expression is false, and vice versa.
    - Example:
    - `flag = False`
    - `inverted_flag = not flag`
    - `print(inverted_flag)  # Output: True`

Remember these key points:

- Logical operators are often used in conditional statements (such as `if`, `elif`, and `else`) to control program flow.
- Parentheses can be used to group expressions and control the order of evaluation.

Feel free to experiment with different expressions and logical operators to deepen your understanding! □

Sources:

1. Programiz - Python Operators (With Examples)
2. Python Numerical Methods - Logical Expressions and Operators
3. GeeksforGeeks - Python Logical Operators
4. W3Schools - Python Logical Operators[1234]

3. Write a python program that interchanges the first and last character of a given string.

Ans:

```
Name = "Vishal"

# lst = list(Name)
# temp = lst[0]
# lst[0] = lst[-1]
# lst[-1] = temp
# str = "".join(lst)
# print(str)

def Swap(Name):
    FC = Name[0]
    LC = Name[-1]
    return LC + Name[1:-1] + FC

print(Swap(Name))
```

Unit II:-

2022-23

1. With suitable program explain the working of while loop.

   Ans:-

Certainly! Let’s explore the working of a **while loop** in Python.

In Python, a **while loop** is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes `false`, the line immediately after the loop in the program is executed. Here’s the basic syntax:

```
while expression:
    # Statement(s) to be executed
```

- The loop will continue executing the statement(s) as long as the specified `expression` evaluates to `True`.
- It’s essential to ensure that the expression eventually becomes `False`, otherwise, the loop will run indefinitely.

Let’s illustrate this with an example. Suppose we want to print numbers from 1 to 3 using a while loop:

```
count = 1
while count < 4:
    print(count)
    count += 1
```

In this example:

1. We initialize the `count` variable to 1.
2. The `while` loop checks whether `count` is less than 4.
3. If `count` is less than 4, it executes the code block (printing the value of `count`) and increments `count`.
4. The loop continues until `count` becomes 4 (which satisfies the condition).

The output of the above program will be:

```
1
2
3
```

Remember to update the loop control variable (`count` in this case) within the loop to avoid an infinite loop. Otherwise, the loop will never end! □

2. Explain the following dictionary methods.
    a) dict.clear()
    b) dict.items()
    c) pop()
    d) dict.copy()

Ans: Certainly! Let’s explore each of the dictionary methods you’ve mentioned:

1. **`dict.clear()`**:
    - The `clear()` method removes all key-value pairs from a dictionary.
    - After calling `clear()`, the dictionary becomes empty.
    - Example:
    - `my_dict = {"name": "Alice", "age": 30}`
    - `my_dict.clear()`

  - `print(my_dict)  # Output: {}`
2. **`dict.items()`**:
    - The `items()` method returns a view of all key-value pairs in the dictionary as tuples.
    - Each tuple contains a key and its corresponding value.
    - Example:
    - `my_dict = {"name": "Bob", "city": "New York"}`
    - `items_view = my_dict.items()`
    - `print(items_view)  # Output: dict_items([('name', 'Bob'), ('city', 'New York')])`
3. **`pop(key[, default])`**:
    - The `pop()` method removes the specified key from the dictionary and returns its associated value.
    - If the key is not found, it raises a `KeyError` (unless a default value is provided).
    - Example:
    - `my_dict = {"name": "Charlie", "age": 25}`
    - `age_value = my_dict.pop("age")`
    - `print(age_value)  # Output: 25`
4. **`dict.copy()`**:
    - The `copy()` method creates a shallow copy of the dictionary.
    - It returns a new dictionary with the same key-value pairs.
    - Changes made to the original dictionary do not affect the copied dictionary.
    - Example:
    - `original_dict = {"fruit": "apple", "color": "red"}`
    - `copied_dict = original_dict.copy()`
    - `original_dict["color"] = "green"`
    - `print(copied_dict)  # Output: {'fruit': 'apple', 'color': 'red'}`

Remember these methods when working with dictionaries in Python. They are useful for managing and manipulating dictionary data. If you have any more questions or need further clarification, feel free to ask! □

Sources:

1. W3Schools - Python Dictionary Methods
2. Python Examples - Dictionary Methods
3. GeeksforGeeks - Python Dictionary Methods[1][2][3]

3. Write a program to find sum of digits in the string (Read a string alphabets and digits).

Ans:
```
def CalDigit(string):

    digit = 0
    for i in string:
        if i.isdigit():
            digit += int(i)
    return digit
string = input("Enter the string")
print(CalDigit(string))
```

28 Jul 2022

1. What is the purpose of else clause in a loop? Explain how else works with while and for loops, with examples.

Ans: Certainly! The `else` clause in loops introduces an intriguing concept: it executes a block of code only if the loop completes its iteration without encountering a `break` statement. This can be particularly useful when you want to perform an action only when a certain condition isn't met by any element in the loop[1]. Let's dive into the details and examples for both `while` and `for` loops:

1. **`else` with `while` loop**:
    - In a `while` loop, the `else` block executes after the loop body completes all iterations without encountering a `break`.
    - Here's an example using a `while` loop:
    - `i = 0`
    - `while i < 5:`
    - `    i += 1`
    - `    print("i =", i)`
    - `else:`
    - `    print("else block is executed")`

      Output:

      ```
      i = 1
      i = 2
      i = 3
      i = 4
      i = 5
      else block is executed
      ```

    - Explanation:
        - We declare `i = 0`.
        - The `while` loop continues as long as `i < 5`.
        - The value of `i` increments with each iteration.
        - The `else` block executes when the value of `i` becomes 5.
2. **`else` with `for` loop**:
    - Similarly, the `else` block in a `for` loop executes after the loop iterates through all elements without encountering a `break`.
    - Example using a `for` loop:
    - `l = [1, 2, 3, 4, 5]`
    - `for a in l:`
    - `    print(a)`

```
else:
    print("else block is executed")
```

Output:

```
1
2
3
4
5
else block is executed
```

- Explanation:
  - We have a list `l = [1, 2, 3, 4, 5]`.
  - The `for` loop iterates through each element (`a`) in the list.
  - The `else` block executes when the loop reads the last element of the list.

3. **`else` with the `break` statement**:
   - The `else` block after a `for` or `while` loop is executed only when the loop is **not** terminated by a `break` statement.
   - Example using a `while` loop:

```
def contains_even_number(lst):
    n = len(lst)
    i = 0
    while i < n:
        if lst[i] % 2 == 0:
            print("List contains an even number")
            break
        i += 1
    else:
        print("List does not contain an even number")

contains_even_number([1, 9, 8])  # For List 1
contains_even_number([1, 3, 5])  # For List 2
```

Output:

```
For List 1: List contains an even number
For List 2: List does not contain an even number
```

- Example using a `for` loop (same logic as above):

```
def contains_even_number(lst):
    for ele in lst:
        if ele % 2 == 0:
            print("List contains an even number")
            break
    else:
        print("List does not contain an even number")

contains_even_number([1, 9, 8])  # For List 1
contains_even_number([1, 3, 5])  # For List 2
```

Output is the same as above.

In summary, the `else` clause in loops provides a way to execute specific code when the loop completes all iterations without encountering a `break`. It’s a powerful construct for handling loop behavior in Python! □□[23].

2. Explain the use of break and continue statement in for loop with suitable examples.

Ans: Certainly! Let’s explore the **`break`** and **`continue`** statements in the context of loops, specifically focusing on **`for`** loops. These statements allow you to control the flow of execution within a loop.

1. **`break` Statement**:
   - The **`break`** statement is used to **terminate** the loop immediately when a specific condition is met.
   - When encountered, the loop stops executing, and control moves to the code following the loop.
   - It’s often used to exit a loop early based on some condition.
   - Example using a **`for`** loop in Python:

   ```
   for i in range(10):
       print(i)
       if i == 5:
           break  # Exit the loop when i reaches 5
   ```

   Output:

   ```
   0
   1
   2
   3
   4
   5
   ```

2. **`continue` Statement**:
   - The **`continue`** statement is used to **skip** the current iteration and move to the next iteration of the loop.
   - When encountered, it jumps directly to the next iteration without executing the remaining code within the loop for the current iteration.
   - Example using a **`for`** loop in Python:

   ```
   for i in range(6):
       if i == 3:
           continue  # Skip printing when i is 3
       print(i)
   ```

   Output:

   ```
   0
   ```

```
1
2
4
5
```

3. **Differences** between **`break`** and **`continue`**:
    - **`break`**:
        - Used to exit the loop entirely.
        - Stops execution of the loop immediately.
        - Commonly used with **`switch`** statements and other loops (e.g., **`while`**, **`do-while`**).
        - Leftover iterations are not executed after encountering **`break`**.
    - **`continue`**:
        - Used to skip the current iteration and proceed to the next one.
        - Skips the remaining code for the current iteration.
        - Also used within loops (except **`switch`** statements).
        - Leftover iterations can still execute even after encountering **`continue`**.

Remember that both **`break`** and **`continue`** statements can be used in **`for`**, **`while`**, and **`do-while`** loops. They provide flexibility in controlling loop behavior based on specific conditions[1][2][3]. □□

3. Write a program to add digits of a number using while loop.

Ans:
```
num = 12345

sum = 0
while num!=0:
    rem = num % 10
    sum += rem
    num //=10
print(sum)
```

10 Nov 2022

1. Difference between continue and pass statements.

Ans: Certainly! Let's explore the differences between the **`continue`** and **`pass`** statements in Python:

1. **`continue`** **Statement**:
    - The **`continue`** statement is used within loops (such as **`for`**, **`while`**, or **`do-while`**) to skip the current iteration and proceed to the next iteration.

  - When encountered, it immediately jumps to the next iteration without executing the remaining code within the loop for the current iteration.
  - It's typically used when you want to skip specific iterations based on a condition.
  - Example using a **`for`** loop:

    ```
    for i in range(5):
        if i == 2:
            continue  # Skip printing when i is 2
        print(i)
    ```

    Output:

    ```
    0
    1
    3
    4
    ```

2. **`pass` Statement**:
   - The **`pass`** statement is a no-op (no operation) statement.
   - It does nothing when executed and serves as a placeholder.
   - It's often used when you need a syntactically correct block of code but don't want to execute any specific logic.
   - Example:

     ```
     def placeholder_function():
         pass  # Placeholder function with no implementation

     # You can define an empty class as well:
     class EmptyClass:
         pass
     ```

     In this case, both the function and the class are syntactically correct but do nothing.

3. **Key Differences**:
   - **`continue`**:
     - Skips the current iteration and proceeds to the next one.
     - Used within loops.
     - Useful for skipping specific iterations.
   - **`pass`**:
     - Does nothing; acts as a placeholder.
     - Used when you need a syntactically correct block but no actual logic.
     - Commonly used for stubs or empty classes/functions.

Remember that **`continue`** and **`pass`** serve different purposes: the former controls the flow of the loop, while the latter is simply a placeholder for future code. □□

2. What are the different loops available in python? Explain in brief.

Ans: Certainly! Python provides two main types of loops: **`for`** loops and **`while`** loops. Let's explore each of them briefly:

1. **`for` Loop**:
   - The **`for`** loop is used for sequential traversal. It iterates over a sequence (such as a list, tuple, set, or string) and executes a block of code for each item in that sequence.
   - Syntax:

     ```
     for item in sequence:
         # Code to execute for each item
     ```
   - Example (printing numbers from 1 to 5):

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

     Output:

     ```
     1
     2
     3
     4
     5
     ```
2. **`while` Loop**:
   - The **`while`** loop executes a block of statements repeatedly until a given condition is satisfied. It keeps running as long as the condition remains true.
   - Syntax:

     ```
     while condition:
         # Code to execute while condition is true
     ```
   - Example (printing "Hello Geek" three times):

     ```
     count = 0
     while count < 3:
         print("Hello Geek")
         count += 1
     ```

     Output:

     ```
     Hello Geek
     Hello Geek
     Hello Geek
     ```
3. **Nested Loops**:
   - You can also have loops within loops (nested loops). For example, using a **`for`** loop inside a **`while`** loop or vice versa.
   - Example (nested loop to print a pattern):

     ```
     for i in range(1, 4):
         for j in range(1, i + 1):
             print("*", end=" ")
         print()
     ```

Output:

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

Remember that both **`for`** and **`while`** loops are powerful tools for controlling program flow and handling repetitive tasks in Python! □□[12345].

3. Write a program to display the Fibonacci sequence up to nth term where n is provided by the user.

Ans:

```python
nterm = int(input("Terms.."))
count = 0
n1,n2 = 0,1

if nterm <=0:
    print("Enter the positive integer")

elif nterm == 1:
    print(n1)

else:
    while count < nterm:
        print(n1)
        nth = n1 + n2
        n1 = n2
        n2 = nth
        count+=1

```

15 Jul 2023

1. Explain if elif else statement in python with syntax, flowchart and example?

Ans: Certainly! Let’s dive into the details of **conditional statements** in Python, including the **`if`**, **`elif`**, and **`else`** constructs. I’ll provide you with the syntax, a flowchart, and examples to illustrate their usage.

1. `if` Statement:

The **`if`** statement is used to execute a block of code if a specific condition holds true. Here's the syntax:

```
if condition:
    # Code to execute if the condition is true
```

- The code block under the **`if`** statement runs only when the specified condition evaluates to **true**.
- Proper **indentation** (white space) is crucial in Python to delimit the block of code associated with the **`if`** statement.

Flowchart of the `if` Statement:

!Flowchart of if Statement

Example: Basic Conditional Check with `if` Statement

```
if 10 > 5:
    print("10 is greater than 5")
print("Program ended")
```

Output:

```
10 is greater than 5
Program ended
```

2. `if-else` Statement:

The **`if-else`** statement combines an additional block of code (the **`else`** block) that executes when the **`if`** condition is **false**. Here's the syntax:

```
if condition:
    # Executes this block if the condition is true
else:
    # Executes this block if the condition is false
```

Flowchart of the `if-else` Statement:

!Flowchart of if-else Statement

Example 1: Handling Conditional Scenarios with `if-else`

```
x = 3
if x == 4:
    print("Yes")
else:
    print("No")
```

Output:

```
No
```

3. `if-elif-else` Statement:

When you need to make a choice among more than two alternatives, you can use the **`if-elif-else`** statement. It allows you to handle multiple conditions. Here's the syntax:

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

Flowchart for a Python `if-elif-else` statement:

!Flowchart for if-elif-else Statement

Example 2: Nested `if-elif` Chain for Multiple Conditions

```
letter = "A"
if letter == "B":
    print("Letter is B")
else:
    if letter == "C":
        print("Letter is C")
    else:
        if letter == "A":
            print("Letter is A")
        else:
            print("Letter isn't A, B, or C")
```

Output:

```
Letter is A
```

In summary, conditional statements in Python allow you to control program flow based on specific conditions. Whether it's a simple **`if`**, a branching **`if-else`**, or a hierarchical **`if-elif-else`**, these constructs help you make decisions in your code! □□[1][2][3][4][5].

2. Write a program to display the Fibonacci sequence up to nth term where n is provided by the user.
3. Write a python program to count the number of vowels in a string provided by the user.

Ans: def is_vowel(letter):

```
    """Returns True if the letter is a vowel."""
    return letter in ["a", "e", "i", "o", "u"]


def count_vowels(string):
    """Counts the number of vowels in a string."""
    return len(list(filter(is_vowel, string)))


user_input = input("Enter a string: ")
print("Number of vowels:", count_vowels(user_input))
```

Unit III:-

28 Jul 2022

1. differentiate between the function with key word arguments and functions with default arguments

Ans: Certainly! Let’s differentiate between **keyword arguments** and **default arguments** in Python:

1. **Default Arguments**:
    - **Definition**: Default arguments are specified when defining a function. They allow you to set default values for function parameters.
    - **Purpose**: Default arguments make certain parameters optional when calling the function. If a value is not provided for a parameter, the default value is used.
    - **Example**:

    ```
    def greet_person(name, greeting="Hello"):
        complete_greeting = f"{greeting}, {name}!"
        return complete_greeting

    print(greet_person("Alice"))  # Uses default greeting ("Hello")
    print(greet_person("Bob", "Hi"))  # Custom greeting provided
    ```

    - In the example above, the `greeting` parameter has a default value of `"Hello"`. If no custom greeting is provided, the default greeting is used.
2. **Keyword Arguments**:

  - **Definition**: Keyword arguments are how you call a function. When calling a function, you can supply name-value pairs of `parameter_name=value` instead of just the value.
  - **Purpose**: Keyword arguments allow you to supply arguments in any order and explicitly specify which parameter each value corresponds to.
  - **Example**:

```
def calculate_power(base, exponent):
    return base ** exponent

result1 = calculate_power(2, 3)   # Positional arguments
result2 = calculate_power(exponent=3, base=2)   # Keyword
arguments

print(result1)   # 8
print(result2)   # 8
```

  - In the example above, both calls to `calculate_power` yield the same result. The second call uses keyword arguments to specify the parameters.

In summary:

- **Default arguments** are set when defining the function.
- **Keyword arguments** are how you call the function, allowing you to supply arguments in any order and explicitly specify parameter names[1][2][3].

2. what is inheritance? Demonstrate the implementation of multilevel inheritance

Ans: **Inheritance** is a fundamental concept in object-oriented programming (OOP) that allows you to create new classes based on existing classes. It enables code reuse by allowing a new class (called a **subclass** or **derived class**) to inherit properties (attributes and methods) from an existing class (called a **superclass**, **base class**, or **parent class**). In other words, inheritance forms a hierarchy of classes, similar to a family tree.

Here are some key points about inheritance:

- A subclass inherits all the attributes and methods of its superclass.
- The subclass can add its own specific attributes and methods.
- Inheritance promotes code organization, modularity, and extensibility.

Now, let's explore **multilevel inheritance** in Python. In multilevel inheritance, a class inherits from another class, which itself inherits from yet another class. It forms a chain of inheritance, allowing properties to flow from the topmost base class down to the derived class.

Example 1: Simple Multilevel Inheritance

Consider the following example:

```
class Base:
    def __init__(self, name, roll, role):
        self.name = name
        self.roll = roll
        self.role = role

class Intermediate(Base):
    def __init__(self, age, name, roll, role):
        super().__init__(name, roll, role)
        self.age = age

class Derived(Intermediate):
    def __init__(self, age, name, roll, role):
        super().__init__(age, name, roll, role)

    def print_data(self):
        print(f"The Name is: {self.name}")
        print(f"The Age is: {self.age}")
        print(f"The Role is: {self.role}")
        print(f"The Roll is: {self.roll}")

obj = Derived(21, "Lokesh Singh", 25, "Software Trainer")
obj.print_data()
```

Output:

```
The Name is: Lokesh Singh
The Age is: 21
The Role is: Software Trainer
The Roll is: 25
```

In this example:

- `Base` is the base class with common attributes (`name`, `roll`, and `role`).
- `Intermediate` inherits from `Base` and adds an `age` attribute.
- `Derived` inherits from `Intermediate`.

Example 2: Multilevel Inheritance with Method Overriding

Let’s consider another example involving method overriding:

```
class Shape:
    def area(self):
        raise NotImplementedError("Subclasses must implement area()")

class Rectangle(Shape):
    def __init__(self, length, width):
        self.width = width
        self.length = length

    def area(self):
        print(f"The Area of Rectangle is {self.length * self.width}")

class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)
    def area(self):
        print(f"The Area of Square is {self.length ** 2}")

rect = Rectangle(3, 4)
my_square = Square(4)
rect.area()
my_square.area()
```

Output:

```
The Area of Rectangle is 12
The Area of Square is 16
```

In this example:

- `Rectangle` inherits from `Shape` and overrides the `area()` method.
- `Square` inherits from `Rectangle` and further overrides the `area()` method.

Multilevel inheritance allows you to build complex class hierarchies while maintaining code reusability and flexibility. □□

References:

1. GeeksforGeeks: Multilevel Inheritance in Python
2. Naukri Code 360: Multilevel Inheritance in Python
3. TechBeamers: Python Multiple Inheritance Explained with Examples
4. Programiz: Python Multiple Inheritance (With Examples)[123456].

3. write a program to find factorial of number using a recursion function
Sent by you: write a program to find factorial of number using a recursion function

Ans: Certainly! Below is a Python program that calculates the factorial of a given number using a recursive function:

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

num = 7  # You can change this value to compute the factorial for a different
number

# Check if the number is negative
if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    print(f"The factorial of {num} is {recur_factorial(num)}")
```

Output:

```
The factorial of 7 is 5040
```

In this program:

- The `recur_factorial` function calculates the factorial recursively.
- We check if the number is negative or zero before computing the factorial.

Feel free to modify the `num` variable to find the factorial for any other positive integer! □□

References:

1. Programiz: Python Program to Find Factorial of Number Using Recursion
2. GeeksforGeeks: Python Program to Find the Factorial of a Number Using Recursion
3. Sanfoundry: Factorial Program in Python Using Recursion
4. GeeksforGeeks: Python Program for Factorial of a Number[1][2][3][4]

10 Nov 2023

1. explain the concept od method overriding with an example

Ans: Certainly! **Method overriding** is a fundamental concept in object-oriented programming (OOP) that allows a subclass (or child class) to provide its own implementation for a method that is already defined in its superclass (or parent class). When a method in a subclass has the same name, same parameters (or signature), and same return type (or sub-type) as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

Here's how method overriding works in Python:

1. **Defining a Superclass (Parent Class)**:
   - We start by defining a superclass with a method that we want to override.
   - The superclass provides a default implementation of the method.
   - Example:
   - `class Parent:`
   - `    def show(self):`
   - `        print("Inside Parent")`
2. **Creating a Subclass (Child Class)**:
   - Next, we create a subclass that extends the superclass.
   - The subclass overrides the method from the superclass by providing its own implementation.
   - Example:
   - `class Child(Parent):`
   - `    def show(self):`

    - `        print("Inside Child")`
3. **Method Invocation**:
    - When an object of the subclass calls the overridden method, Python uses the implementation from the subclass instead of the one in the superclass.
    - The version of the method executed depends on the type of the object being referred to (not the type of the reference variable).
    - Example:
    - `obj = Child()`
    - `obj.show()  # Output: "Inside Child"`

In summary, method overriding allows a subclass to provide its own specialized implementation of a method inherited from its superclass. It promotes flexibility, polymorphism, and code extensibility in OOP! □□

References:

1. GeeksforGeeks: Method Overriding in Python
2. Pickl.AI: Method Overloading & Overriding in Python
3. Learninbits: Method Overriding in Python: Mastering Polymorphism
4. Gyata: Mastering Method Overriding in Python
5. Javatpoint: Method Overriding in Python[1][2][3][4][5].

2. python code to check whether the given number is armstrong number using module

Ans:

15 Jul 2023

1. Describe function with default arguments with suitable program.

Ans: Certainly! In Python, you can define a function with default arguments. Default arguments allow you to specify default values for function parameters. If the function is called without providing a value for a specific argument, it will use the default value assigned to that argument. Let's explore this concept with an example:

Consider a function called `greet_person` that takes two arguments:

1. `name` (required): The name of the person.
2. `greeting` (optional, with a default value): The greeting message.

Here's how you can define the function with default arguments:

```
def greet_person(name, greeting="Hello"):
    """
    Greets a person with an optional custom greeting.
    Args:
        name (str): The name of the person.
        greeting (str, optional): The greeting message (default is "Hello").

    Returns:
        str: The complete greeting.
    """
    complete_greeting = f"{greeting}, {name}!"
    return complete_greeting

# Example usage:
print(greet_person("Alice"))  # Uses default greeting ("Hello")
print(greet_person("Bob", "Hi"))  # Custom greeting provided
```

Output:

```
Hello, Alice!
Hi, Bob!
```

In the example above:

- The `greet_person` function accepts a required argument (`name`) and an optional argument (`greeting`).
- The default value for `greeting` is set to `"Hello"`.
- When calling the function, you can either provide a custom greeting or use the default greeting.

Remember these points when working with default arguments:

1. The order of arguments matters when calling the function.
2. If you provide a value for an argument, it overrides the default value.
3. You can use both positional arguments (without specifying the parameter name) and keyword arguments (specifying the parameter name) when calling the function.

Feel free to experiment with different names and greetings to see how the function behaves! □□

References:

1. Default arguments in Python - GeeksforGeeks
2. How to Define a Function with Default Arguments in Python?
3. Mastering Default Arguments in Python: A Comprehensive Guide - Gyata
4. How to Use Python Default Parameters - Python Tutorial
5. Python Function Arguments (With Examples) - Programiz[12345].

2. Explain how to implement multiple inheritance in python.

Ans: Certainly! In Python, **multiple inheritance** allows a class to inherit from more than one parent class. This feature enables you to create versatile and complex class hierarchies. Let’s explore how to implement multiple inheritance using examples:

1. **Syntax for Multiple Inheritance**:
    - To create a class that inherits from multiple parent classes, specify the parent classes inside the parentheses after the child class name.
    - Syntax:

    ```
    class ChildClass(ParentClass1, ParentClass2, ...):
        # Body of the child class
    ```
2. **Example 1: Basic Multiple Inheritance**:
    - Suppose we have two parent classes: `Animal` and `Bird`.
    - The `Parrot` class inherits from both `Animal` and `Bird`.
    - Example:

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

    class Bird:
        def fly(self):
            print("Bird flies")

    class Parrot(Animal, Bird):
        pass

    parrot = Parrot()
    parrot.speak()  # Output: Animal speaks
    parrot.fly()    # Output: Bird flies
    ```
3. **Example 2: Method Overriding in Multiple Inheritance**:
    - When a method is overridden in both parent classes, the child class can choose which implementation to use.
    - Example:

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

    class Bird:
        def speak(self):
            print("Bird speaks")

    class Parrot(Animal, Bird):
        pass

    parrot = Parrot()
    parrot.speak()  # Output: Bird speaks (method from Bird class)
    ```
4. **Method Resolution Order (MRO)**:
    - Python uses the **C3 Linearization** algorithm to determine the order in which methods are called in multiple inheritance.
    - You can check the method resolution order using the `mro()` method or the `super()` function.
    - Example:

```
print(Parrot.mro())  # Output: [<class '__main__.Parrot'>, <class '__main__.Animal'>, <class '__main__.Bird'>, <class 'object'>]
```

Remember that multiple inheritance should be used carefully to avoid ambiguity and maintain code readability. Understanding method resolution order (MRO) is essential when dealing with complex class hierarchies! □□

References:

1. GeeksforGeeks: Multiple Inheritance in Python
2. Python Tutorial: Python Multiple Inheritance
3. Programiz: Python Multiple Inheritance (With Examples)
4. Machine Learning Tutorials: Multiple Inheritance in Python (With Examples)[1234].

3. Write a program to find the factorial of given number without recursion and with recursion.

Ans: Certainly! Below are Python programs to find the factorial of a given number both **without recursion** and **with recursion**:

1. **Without Recursion**:
   - We use a `while` loop to calculate the factorial iteratively.
   - Here's the code:

```
def factorial_without_recursion(n):
    fact = 1
    while n > 0:
        fact *= n
        n -= 1
    return fact

num = int(input("Enter a number: "))
print(f"Factorial of {num} (without recursion) is:", factorial_without_recursion(num))
```

2. **With Recursion**:
   - We use a recursive function to calculate the factorial.
   - Here's the code:

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

num = int(input("Enter a number: "))
print(f"Factorial of {num} (with recursion) is:", factorial_with_recursion(num))
```

Feel free to test these programs with different numbers! □□

References:

1. Sanfoundry: Python Program to Find the Factorial of a Number Without using Recursion
2. Tutorialspoint: Python Program to Find the Factorial of a Number Without using Recursion
3. Instanceofjava: Find Factorial without Recursion in Python
4. datagy.io: Python Factorial Function: Find Factorials in Python[1234].

Unit IV

28 Jul 2022

1. Discuss the following list functions.
    a) Len()
    b) Sum()
    c) Any()
    d) All()
    e) Sorted

Ans: Certainly! Let's discuss each of the mentioned list functions:

1. **`len()`**:
    - The `len()` function returns the number of elements (items) in a given iterable (such as a list, tuple, string, or dictionary).
    - Example:

    - `my_list = [10, 20, 30, 40, 50]`
    - `print("Length of the list:", len(my_list))  # Output: 5`
2. **`sum()`**:
    - The `sum()` function calculates the sum of all elements in an iterable (such as a list, tuple, or set).
    - Example:
    - `my_numbers = [1, 2, 3, 4, 5]`
    - `total_sum = sum(my_numbers)`
    - `print("Sum of the numbers:", total_sum)  # Output: 15`
3. **`any()`**:
    - The `any()` function returns `True` if at least one element in the iterable is `True`. Otherwise, it returns `False`.
    - Example:
    - `my_bools = [True, False, True, False]`
    - `result = any(my_bools)`
    - `print("Any True value in the list?", result)  # Output: True`
4. **`all()`**:
    - The `all()` function returns `True` if all elements in the iterable are `True`. Otherwise, it returns `False`.
    - Example:
    - `my_bools = [True, False, True, True]`
    - `result = all(my_bools)`
    - `print("All True values in the list?", result)  # Output: False`
5. **`sorted()`**:
    - The `sorted()` function returns a new sorted list (or other iterable) based on the elements of the original iterable.
    - It does not modify the original list.
    - Example:
    - `my_list = [5, 2, 8, 1, 3]`
    - `sorted_list = sorted(my_list)`
    - `print("Sorted list:", sorted_list)  # Output: [1, 2, 3, 5, 8]`

Remember to use these functions appropriately based on your specific requirements! □□

2. How to handle an exception using try except block? Explain with the help of a program.

Ans: Certainly! Exception handling in Python allows you to gracefully handle errors or exceptions that may occur during program execution. The `try` and `except` blocks are used for this purpose. Let me explain how to handle exceptions using a program:

1. **Syntax of `try` and `except`**:
    - The `try` block contains the code that might raise an exception.
    - The `except` block contains the code to handle the exception if it occurs.
    - Example:
    - `try:`

```
    # Code that may cause an exception
    num = int(input("Enter a number: "))
    result = 10 / num
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero")
except ValueError:
    print("Error: Invalid input (not an integer)")
```

2. **Explanation**:
   - In the example above:
     - The `try` block attempts to get user input, convert it to an integer, and perform a division.
     - If the user enters zero (causing a `ZeroDivisionError`) or a non-integer value (causing a `ValueError`), the corresponding `except` block is executed.
     - You can add more `except` blocks for other specific exceptions as needed.
3. **Common Exceptions**:
   - Some common exceptions include:
     - `ZeroDivisionError`: Raised when dividing by zero.
     - `ValueError`: Raised when an invalid value is used (e.g., converting a non-integer string to an integer).
     - `TypeError`: Raised when an operation is performed on an inappropriate data type.
     - `NameError`: Raised when a local or global name is not found.
     - …and many more.
4. **Handling Multiple Exceptions**:
   - You can handle multiple exceptions by specifying multiple `except` blocks.
   - Example:

     ```
     try:
         # Code that may cause exceptions
         num = int(input("Enter a number: "))
         result = 10 / num
         print(f"Result: {result}")
     except ZeroDivisionError:
         print("Error: Cannot divide by zero")
     except ValueError:
         print("Error: Invalid input (not an integer)")
     except Exception as e:
         print(f"Error: {e}")
     ```
5. **Using `else` and `finally`**:
   - You can also use an `else` block (executed if no exceptions occur) and a `finally` block (always executed, regardless of exceptions).
   - Example:

     ```
     try:
         # Code that may cause exceptions
         num = int(input("Enter a number: "))
         result = 10 / num
         print(f"Result: {result}")
     except ZeroDivisionError:
         print("Error: Cannot divide by zero")
     except ValueError:
         print("Error: Invalid input (not an integer)")
     else:
     ```

```
o       print("No exceptions occurred")
o   finally:
o       print("Execution completed")
```

Remember to handle exceptions appropriately based on your specific use case! □□

References:

1. W3Schools: Python Try Except
2. DataMentor: Python try...except (Exception Handling) (With Examples)
3. Programiz: Python Exception Handling (With Examples)
4. Python Tutorial: Python Try Except Block[1234].

3. Write a python program to check whether a string is palindrome.

Ans:
```
def is_palindrome(s):

   s = s.lower().replace(" ", "")  # Make it suitable for caseless comparison

   return s == s[::-1]



my_str = "A man a plan a canal Panama"

if is_palindrome(my_str):

   print("The string is a palindrome.")

else:

   print("The string is not a palindrome.")
```

10 Nov 2022

1. List the various dictionary methods and explain any 3 of them.
2. Explain the except clause with multiple exceptions.
3. Write a program to use max(),min() and sorted() methods in tuples.

15 Jul 2023

1. Explain how to create, raise and handle user defined exceptions in python.

Ans: Certainly! In Python, you can create, raise, and handle user-defined exceptions. Let’s break down each step:

1. **Creating User-Defined Exceptions**:
    - To create a custom exception, you need to define a new class that inherits from the built-in `Exception` class (or any of its subclasses).
    - You can name your custom exception class as you like, but it’s a good practice to end the name with “Error” (similar to built-in exceptions).
    - Example:

    ```
    class CustomError(Exception):
        """Custom exception for specific scenarios."""
        pass
    ```
2. **Raising User-Defined Exceptions**:
    - You can raise your custom exception using the `raise` keyword when a specific condition is met.
    - Example:

    ```
    def validate_age(age):
        if age < 18:
            raise CustomError("Age must be 18 or older")
        else:
            print("Valid age")

    try:
        user_age = int(input("Enter your age: "))
        validate_age(user_age)
    except CustomError as e:
        print(f"Error: {e}")
    ```
3. **Handling User-Defined Exceptions**:
    - To handle your custom exception, use the `try` and `except` blocks.
    - In the `except` block, catch your custom exception and provide appropriate handling code.
    - Example:

    ```
    try:
        user_age = int(input("Enter your age: "))
        validate_age(user_age)
    except CustomError as e:
        print(f"Error: {e}")
    else:
        print("No exceptions occurred")
    finally:
        print("Execution completed")
    ```

Remember to customize your exception classes based on the specific scenarios you want to handle! □□

References:

1. GeeksforGeeks: User-defined Exceptions in Python with Examples
2. Analytics Vidhya: The Complete Guide to User-Defined Exceptions in Python
3. PythonForBeginners.com: User-Defined Exceptions in Python
4. Machine Learning Tutorials: Python User-Defined Exceptions Tutorial (With Examples)[1234].

2. Write a python function that takes two lists and returns True if they have at least one common member.

```
Ans: def have_common_element(list1, list2):
    # Convert lists to sets for efficient intersection check
    set1 = set(list1)
    set2 = set(list2)

    # Check if the intersection of sets is not empty
    return bool(set1 & set2)

# Example usage:
list_a = [1, 2, 3, 4, 5]
list_b = [5, 6, 7, 8, 9]

if have_common_element(list_a, list_b):
    print("The lists have at least one common element.")
else:
    print("The lists do not have any common elements.")
```

3. Write a python program to handle a ZeroDivisionError exception when dividing a number by zero.

```
Ans: try:
    div = 130//0
    print("The Division is ",div)
except ZeroDivisionError as e:
    print(e)
    print("You can't divide by zero!")
```

Unit V

28 Jul 2022

1. Discuss the following methods associate with the file object.
   a. Read()
   b. Readline()
   c. Readlines()

Ans: Certainly! Let's discuss each of the mentioned methods associated with the file object in Python:

1. **`read()`**:
   - The `read()` method reads the entire content of a file and returns it as a single string.
   - It reads from the current file position up to the end of the file.
   - Example:
   - `with open("my_file.txt", "r") as file:`
   - `    content = file.read()`
   - `    print(content)`
2. **`readline()`**:
   - The `readline()` method reads a single line from the file.
   - It moves the file pointer to the next line.
   - Example:
   - `with open("my_file.txt", "r") as file:`
   - `    line = file.readline()`
   - `    print(line)`
3. **`readlines()`**:
   - The `readlines()` method reads all lines from the file and returns them as a list of strings.
   - Each line is an element in the list.
   - Example:

```
with open("my_file.txt", "r") as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())  # Remove newline characters
```

Remember to replace `"my_file.txt"` with the actual file path you want to read from! □□

References:

1. W3Schools: Python File Methods - `read()`
2. W3Schools: Python File Methods - `readline()`
3. W3Schools: Python File Methods - `readlines()` [1].

2. Write a python block to create the table StudentInfo with suitable attributes/columns in MySQL and then write a python program to insert some records in the table and fetch all rows from the table.

Ans:

```
import pyodbc



conn_str = (
    "Driver={ODBC Driver 17 for SQL Server};"
    "Server=VJ\SQLEXPRESS;"
    "Database=vishal;"
    "Trusted_Connection=yes;"
)

conn = pyodbc.connect(conn_str)

cursor = conn.cursor()

cursor.execute("DROP TABLE IF EXISTS DemoTable")


cursor.execute("CREATE TABLE DemoTable (Name varchar(250),Roll_No int, Class
varchar(25))")

# cursor.execute("INSERT INTO DemoTable(Name,Roll_No,Class)
VALUES('Vishal',26,'FYMCA')")
# cursor.execute("SELECT * FROM DemoTable")
# row = cursor.fetchone()
# print(row)
```

```
Val = [('Vishal',26,'FYMCA'),('Abhi',27,'FYMCA'),('Ashish',25,'FYMCA')]
cursor.executemany("INSERT INTO DemoTable (Name,Roll_No,Class)
VALUES(?,?,?)",Val)
cursor.execute("SELECT * FROM Demotable")
rows = cursor.fetchall()
for row in rows:
    print(row)

cursor.close()
conn.close()
```

3. Write a program to append one file into another.

```
Ans: # 3. Write a program to append one file into another.


try:
    f1 = open("File1.txt","r")
    f2 = open("File2.txt","a")
    data = f1.read()
    f2.write(data)
    f1.close()
    f2.close()
    f2 = open("File2.txt","r")
    print(f2.read())
except FileNotFoundError:
    print("File not Found!")
```

10 Nov 2022

1. Describe the different access modes of the file manipulation.

Ans: Certainly! In Python, file access modes determine how a file is opened and what operations can be performed on it. When you open a file using the `open()` function, you can specify the file mode as the second argument. Below are the different file access modes in Python:

1. **Read Mode (`'r'`):**
    - Opens the file for reading only.
    - Raises a `FileNotFoundError` if the file does not exist.
    - Example:
    - `with open("my_file.txt", "r") as file:`
    - `    content = file.read()`

    - `    print(content)`
2. **Write Mode (`'w'`)**:
    - Opens the file for writing only.
    - Truncates the file to zero length if it already exists.
    - Creates a new file if it does not exist.
    - Example:
    - `with open("my_file.txt", "w") as file:`
    - `    file.write("Hello, world!")`
3. **Append Mode (`'a'`)**:
    - Opens the file for appending (writing at the end).
    - Inserts data at the end of the file.
    - Creates a new file if it does not exist.
    - Example:
    - `with open("my_file.txt", "a") as file:`
    - `    file.write("\nAppended line")`
4. **Read and Write Mode (`'r+'`)**:
    - Opens the file for both reading and writing.
    - Raises a `FileNotFoundError` if the file does not exist.
    - Example:
    - `with open("my_file.txt", "r+") as file:`
    - `    content = file.read()`
    - `    file.write("\nAdditional line")`
5. **Binary Mode (`'b'`)**:
    - Used in combination with other modes (e.g., `'rb'`, `'wb'`).
    - Opens the file in binary mode (for reading or writing binary data).
    - Example:
    - `with open("my_binary_file.bin", "rb") as file:`
    - `    binary_data = file.read()`

Remember to choose the appropriate mode based on your specific file manipulation needs! □□

References:

1. GeeksforGeeks: File Mode in Python
2. llego.dev: A Comprehensive Guide to File Modes in Python
3. GeeksforGeeks: Reading and Writing to Text Files in Python
4. W3Schools: Python File Handling
5. EyeHunts: Python File Modes (Open, Write, Append)[12345].

2. Write a python program to create a table and insert some records in that table. Then selects all rows from the table and display the records.
3. Write a program to copy contents of one file into another.

15 Jul 2023

1. How to perform open, read, write, and close operations on a text file?

Ans: Certainly! Let's discuss how to perform open, read, write, and close operations on a text file in Python:

1. **Opening a File**:
    - To open a file, you can use the `open()` function. It returns a file object that allows you to interact with the file.
    - Syntax:

    -
      ```
      file = open("filename.txt", "mode")
      ```
    - The `filename.txt` is the name of the file you want to open, and the `mode` specifies the purpose of opening the file (e.g., read, write, append, etc.).
2. **Reading from a File**:
    - To read the contents of a file, you can use methods like `read()`, `readline()`, or `readlines()`.
    - Example using `read()`:

    -
      ```
      with open("my_file.txt", "r") as file:
          content = file.read()
          print(content)
      ```
3. **Writing to a File**:
    - To write data to a file, you can use methods like `write()` or `writelines()`.
    - Example using `write()`:

    -
      ```
      with open("my_file.txt", "w") as file:
          file.write("Hello, world!")
      ```
4. **Closing a File**:
    - Always close the file after you're done with it using the `close()` method.
    - Example:

    -
      ```
      with open("my_file.txt", "r") as file:
          content = file.read()
          print(content)
      # File is automatically closed when the block exits
      ```

Remember to replace `"filename.txt"` with the actual file name you want to work with! □□

References:

1. Programiz: Python File Input/Output
2. FreeCodeCamp: Python Write to File – Open, Read, Append, and Other File Handling Functions Explained
3. GeeksforGeeks: Reading and Writing to Text Files in Python
4. KoderHQ: Basic Python File Operations Tutorial[1234].

2. Write a program to display content of a text file on the monitor by reading it word by word.

Ans:

```
f1 = open("File2.txt","r")

for line in f1:
    # Split the line into words
    words = line.split()
    for word in words:
        print(word)
```

3. Write a python program to create a table and insert some records in that table. Finally selects all rows from the table and display the records. Use suitable names for data base and table.