# PYTHON CONTENT

## CORE PYTHON-VARIABLES

1. **Definition:**
   * A variable in Python ACTS AS A PLACE HOLDER or identifier that represents a memory location holding a value. It allows you to store and manipulate data in your programs.
2. **Variable Naming Rules:**
   * Variable names can contain letters (a-z, A-Z), numbers (0-9), and underscores (\_).
   * The name cannot start with a number.
   * Python is case-sensitive, so **myVar** and **myvar** are considered different variables.
3. **Data Types:**
   * Variables can store different types of data:
     + **int:** Integer, e.g., **x = 5**
     + **float:** Floating-point number, e.g., **y = 3.14**
     + **str:** String, e.g., **name = "John"**
     + **bool:** Boolean, e.g., **is\_true = True**
4. **Multiple Assignments:**
   * You can assign multiple variables in a single line: **a, b, c = 1, 2, 3**
5. **Variable Reassignment:**
   * You can change the value of a variable by assigning a new value to it.
   * Example: **x = 10**, then later **x = 20**
6. **Printing Variables:**
   * You can print the value of a variable using the **print** function.
   * Example: **print(x)**
7. **Constants:**
   * While Python doesn't have true constants, it is a convention to use uppercase letters for variable names whose values should not be changed.
8. **Common Variable Types:**
   * Lists, Tuples, Dictionaries, Sets - these are more complex data structures that can also be stored in variables.

# Variable assignment

x = 10

y = 3.14

name = "John"

is\_true = True

# Multiple assignments

a, b, c = 1, 2, 3

# Variable reassignment

x = 20

# Printing variables

print(x)

print(y)

print(name)

print(is\_true)

**Packing:**

* **Definition:** Packing refers to putting values into a single variable, often in the form of a tuple or a list.
* **Example:**

my\_packed\_tuple = 1, 2, 3 # Values are packed into a t-----------------uple

my\_packed\_list = [4, 5, 6] # Values are packed into a list

**Unpacking:**

* **Definition:** Unpacking refers to extracting values from a packed variable (like a tuple or a list) and assigning them in to separate variables.

# Unpacking a tuple

a, b, c = my\_packed\_tuple # Values from the tuple are unpacked into variables a, b, and c

# Unpacking a list

x, y, z = my\_packed\_list # Values from the list are unpacked into variables x, y, and z

**Combined Example:**

# Packing

my\_packed\_tuple = 1, 2, 3

my\_packed\_list = [4, 5, 6]

# Unpacking

a, b, c = my\_packed\_tuple

x, y, z = my\_packed\_list

print("Packed Tuple:", my\_packed\_tuple)

print("Unpacked Tuple:", a, b, c)

print("Packed List:", my\_packed\_list)

print("Unpacked List:", x, y, z)

In this example, values are first packed into a tuple and a list. Then, the values are unpacked into individual variables. Packing and unpacking are handy techniques, especially when working with functions that return multiple values or dealing with iterable data structures like tuples and lists.

1. **Local Variables:**
   * **Definition:** Variables declared within a function or a block of code have local scope. They are only accessible within that specific function or block.

**Example:**

def my\_function():

local\_variable = 10

print(local\_variable)

my\_function()

# print(local\_variable) # This would result in an error

1. **Global Variables:**
   * **Definition:** Variables declared outside any function or block have global scope. They can be accessed throughout the entire program, including within functions.
   * **Example:**

global\_variable = 20

def my\_function():

print(global\_variable)

my\_function()

print(global\_variable)

1. **Scope Rules:**
   * Local variables take precedence over global variables with the same name within the same scope. If a local variable has the same name as a global variable, the local variable shadows the global one within its scope.

Example:

x = 10 # Global variable

def my\_function():

x = 5 # Local variable with the same name

print(x)

my\_function()

print(x) # This prints the global variable, not the local one

1. **Using Global Variables Inside Functions:**
   * To modify a global variable from within a function, the **global** keyword is used.
   * Example:

global\_var = 30

def modify\_global():

global global\_var

global\_var += 5

modify\_global()

print(global\_var) # Output: 35

1. **Avoiding Global Variables:**
   * It is generally considered good practice to minimize the use of global variables when possible, as they can make the code more complex and harder to understand. Instead, prefer passing variables as arguments to functions and returning values.

## DATATYPES

In Python, a data type is a classification or category of data that specifies which type of value a variable can hold. It defines the operations that can be performed on the data and the methods by which the data interacts with other parts of a program. **DINAMICALLY**:Python is a dynamically-typed language, meaning that the interpreter automatically assigns a data type to a variable based on the value it is assigned. Here are some common data types in Python:

1. **int:** Integer type represents whole numbers POSITIVE ,NEGATIVE without any decimal points.
2. **float:** Float type represents real numbers and includes numbers with decimal points.
3. **str:** String type represents sequences of characters, such as text.
4. **list:** List is a mutable sequence type that can contain elements of different data types.
5. **tuple:** Tuple is an immutable sequence type similar to a list.
6. **set:** Set is an unordered collection of unique elements.
7. **dict:** Dictionary is an unordered collection of key-value pairs.
8. **bool:** Boolean type represents the truth values True or False.

In Python, **bool** is a built-in data type that represents boolean values. A boolean variable can have one of two possible values: **True** or **False**. Boolean values are often used in conditional statements and logical expressions to control the flow of a program.

1. **NoneType:** The type of the special constant **None**, often used to indicate the absence of a value or a null value.
2. **bytes:** Immutable sequence of bytes.
3. **bytearray:** Mutable sequence of bytes.
4. **complex:** Complex numbers with a real and imaginary part.

* **Data Type:** A data type is a classification that defines the type of data a variable can hold, specifying the operations that can be performed on it. Examples include integers, floats, strings, etc.
* **Variable:** A variable is a named storage location in a program that can hold data. It is associated with a data type, and its value can change during the execution of the program. Variables are instances of data types.

**TYPE CASTING**  
Type casting in Python refers to the process of converting a variable from one data type to another.

EXAMPLE

A=5

PRINT(FLOAT(A))

## OPERATORS

An operator in programming is a symbol or set of symbols that performs operations on operands like (values or variables).

1.ARITHMATIC OPERATOR

IN PYTHON , arithmetic refers to the fundamental mathematical operations that can be applied to numerical values. These operations include addition, subtraction, multiplication, division, modulus, exponentiation, and floor division

1. **Addition +:**
   * The addition operator combines two values to produce their sum.
   * Example: **a + b** adds the values of **a** and **b**.
2. **Subtraction -:**
   * The subtraction operator calculates the difference between two values.
   * Example: **a - b** subtracts the value of **b** from **a**.
3. **Multiplication \*:**
   * The multiplication operator multiplies two values to produce their product.
   * Example: **a \* b** multiplies the values of **a** and **b**.
4. **Division /:**
   * The division operator divides the value on the left by the value on the right, producing a floating-point result.
   * Example: **a / b** divides the value of **a** by **b**.
5. **Modulus %:**
   * The modulus operator returns the remainder of the division of the left operand by the right operand.
   * Example: **a % b** returns the remainder when **a** is divided by **b**.
6. **Exponentiation \*\*:**
   * The exponentiation operator raises the left operand to the power of the right operand.
   * Example: **a \*\* b** raises **a** to the power of **b**.
7. **Floor Division //:**
   * The floor division operator returns the quotient of the division, producing an integer result.
   * Example: **a // b** returns the integer result of dividing **a** by **b**.

### 2. Comparison Operators:

Comparison operators in Python are used to compare two values and return a Boolean result indicating whether the comparison is true or false. Here are the main comparison operators:

1. **Equal to (==):**
   * Returns **True** if the values on the left and right are equal.
   * Example: **a == b** returns **True** if **a** is equal to **b**.
2. **Not equal to (!=):**
   * Returns **True** if the values on the left and right are not equal.
   * Example: **a != b** returns **True** if **a** is not equal to **b**.
3. **Less than (<):**
   * Returns **True** if the value on the left is less than the value on the right.
   * Example: **a < b** returns **True** if **a** is less than **b**.
4. **Greater than (>):**
   * Returns **True** if the value on the left is greater than the value on the right.
   * Example: **a > b** returns **True** if **a** is greater than **b**.
5. **Less than or equal to (<=):**
   * Returns **True** if the value on the left is less than or equal to the value on the right.
   * Example: **a <= b** returns **True** if **a** is less than or equal to **b**.
6. **Greater than or equal to (>=):**
   * Returns **True** if the value on the left is greater than or equal to the value on the right.
   * Example: **a >= b** returns **True** if **a** is greater than or equal to **b**.

### 3. Logical Operators:

Logical operators in Python are used to perform logical operations on Boolean values.

1. **Logical AND (and):**
   * Returns **True** if both the left and right operands are true.
   * Example: **a and b** returns **True** if both **a** and **b** are true.
2. **Logical OR (or):**
   * Returns **True** if at least one of the left or right operands is true.
   * Example: **a or b** returns **True** if either **a** or **b** or both are true.
3. **Logical NOT (not):**
   * Returns the opposite of the operand's Boolean value. If the operand is **True**, **not** returns **False**, and vice versa.
   * Example: **not a** returns **True** if **a** is **False**.

### 4. Assignment Operators:

Assignment operators in Python are used to assign values to variables. They combine the operation of assigning a value to a variable with another operation, such as addition or subtraction. Here are some commonly used assignment operators:

1. **Assignment (=):**
   * Assigns the value on the right to the variable on the left.
   * Example: **a = 10** assigns the value **10** to the variable **a**.
2. **Addition Assignment (+=):**
   * Adds the value on the right to the current value of the variable on the left and assigns the result to the variable on the left.
   * Example: **a += 5** is equivalent to **a = a + 5**.
3. **Subtraction Assignment (-=):**
   * Subtracts the value on the right from the current value of the variable on the left and assigns the result to the variable on the left.
   * Example: **a -= 3** is equivalent to **a = a - 3**.
4. **Multiplication Assignment (\*=):**
   * Multiplies the current value of the variable on the left by the value on the right and assigns the result to the variable on the left.
   * Example: **a \*= 2** is equivalent to **a = a \* 2**.
5. **Division Assignment (/=):**
   * Divides the current value of the variable on the left by the value on the right and assigns the result to the variable on the left.
   * Example: **a /= 4** is equivalent to **a = a / 4**.
6. **Modulus Assignment (%=):**
   * Calculates the modulus of the current value of the variable on the left by the value on the right and assigns the result to the variable on the left.
   * Example: **a %= 3** is equivalent to **a = a % 3**.
7. **Exponentiation Assignment (\*\*=):**
   * Raises the current value of the variable on the left to the power of the value on the right and assigns the result to the variable on the left.
   * Example: **a \*\*= 2** is equivalent to **a = a \*\* 2**.
8. **Floor Division Assignment (//=):**
   * Performs floor division on the current value of the variable on the left by the value on the right and assigns the result to the variable on the left.
   * Example: **a //= 5** is equivalent to **a = a // 5**.

### 5. Identity Operators:

**Identity Operators** in Python are used to compare the memory locations of two objects.

* **is (Identity):**
  + Returns **True** if both operands refer to the same object.
  + Example: **a is b** returns **True** if **a** and **b** refer to the exact same object in memory.
* **is not (Non-identity):**
  + Returns **True** if both operands do not refer to the same object.
  + Example: **a is not b** returns **True** if **a** and **b** do not refer to the same object.

### 6.Membership Operators:

The **Membership Operators** in Python are used to test whether a value is a member of a sequence, such as a string, list, or tuple. There are two membership operators in Python:

* **in (Membership):**
  + Returns **True** if a value is found in the sequence.
  + Example: **x in myList** returns **True** if the value of **x** is present in the list **myList**.
* **not in (Not in Membership):**
  + Returns **True** if a value is not found in the sequence.
  + Example: **y not in mySet** returns **True** if the value of **y** is not present in the set **mySet**.

### Bitwise Operators:

**Bitwise Operators** in Python are used to perform operations on individual bits of integers. These operators work at the binary level, manipulating the binary representations of numbers. Here are the bitwise operators in Python:

* **& (Bitwise AND):**
  + Performs a bitwise AND operation.
  + Example: **a & b** performs a bitwise AND operation between the binary representations of **a** and **b**.
* **| (Bitwise OR):**
  + Performs a bitwise OR operation.
  + Example: **a | b** performs a bitwise OR operation between the binary representations of **a** and **b**.
* **^ (Bitwise XOR):**
  + Performs a bitwise XOR (exclusive OR) operation.
  + Example: **a ^ b** performs a bitwise XOR operation between the binary representations of **a** and **b**.
* **~ (Bitwise NOT):**
  + Inverts the bits of the operand.
  + Example: **~a** inverts all the bits in the binary representation of **a**.
* **<< (Left Shift):**
  + Shifts the bits of the left operand to the left by the number of positions specified by the right operand.
  + Example: **a << 2** shifts the bits of **a** two positions to the left.
* **>> (Right Shift):**
  + Shifts the bits of the left operand to the right by the number of positions specified by the right operand.
  + Example: **a >> 3** shifts the bits of **a** three positions to the right.

## IF STATEMENT

* if Statement:
  + Definition: The if statement is a control flow statement in Python that allows you to execute a block of code based on a specified condition. If the condition is true, the indented code block under the if statement is executed.
* elif Statement:
  + Definition: Short for "else if," the elif statement is used to check multiple conditions in a sequence. If the preceding if or elif conditions are false, the code block under the first true elif condition is executed.
* else Statement:
  + Definition: The else statement is used in conjunction with an if statement. If the conditions specified in the preceding if and elif statements are false, the code block under the else statement is executed.
* Nested if Statement:
  + Definition: A nested if statement is an if statement that appears inside another if statement. It allows for more complex conditional logic by checking multiple conditions in a hierarchical manner.
* Short Hand if:
  + Definition: The short-hand if is a concise way to write an if statement in a single line. It is often used when you need to assign a value or execute a simple statement based on a condition.
* Short Hand if-else:
  + Definition: The short-hand if-else is a concise way to write an if-else statement in a single line. It is useful when you want to assign different values or execute different statements based on a condition.
* Logical Operators:
  + Definition: Logical operators (and, or, not) are used to combine multiple conditions in a conditional statement. They allow you to create more complex conditions by specifying the relationship between multiple expressions.

## The pass Statement

The pass statement in Python is a no-operation statement.for example if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

## LOOPS

Certainly! In programming, a loop is a control structure that allows a sequence of instructions to be repeated multiple times. Loops are two types one is for loop another one is while loop.

There are two main types of loops:

* For Loop:
  + A for loop is used to iterate over a sequence of elements (e.g., a list, tuple, string, or range).
  + It is often used when the number of iterations is known in advance.
* While Loop:
  + A while loop is used to perform a block of code repeatedly until a certain condition is true.
  + It is suitable when the number of iterations is not known beforehand and depends on a certain condition.
* "break" statement: Used to exit a loop prematurely, stopping the execution of the loop.
* "continue" statement: Skips the rest of the current iteration and moves to the next iteration of the loop.
* *else:In Python, you can use the "while" loop with an "else" block. The "else" block is optional and is executed when the condition of the "while" loop becomes false. It is not executed if the loop is terminated by a "break" statement. The "else" block can be used to perform some additional actions after the loop has completed its iterations.*
* *range in Python is a built-in function used for generating sequences of numbers. It is often used with for loops to iterate over a specified range of values. The function takes parameters for start, stop, and step, allowing flexibility in defining the sequence.*

## FUNCTIONS

### FUNCTIONS

* Functions in Python are blocks of reusable code that perform a specific task.
* It is defined using the **def** keyword, followed by a name, parameters, and a code block.
* They help in organizing code and making it more modular.
* main advantage of using functions is that they promote code reusability and make the code easier to understand and maintain.

### CALLING A FUNCTION

To call a function in Python, you simply write the function name followed by parentheses. If the function requires any arguments, you can pass them inside the parentheses.

### USER DEFINED FUNCTION

Python allows you to define your own functions using the "def" keyword. You can define the function name, any required parameters, and the code block that will be executed when the function is called.

### PASS BY REFERENCE VS PASS BY VALUE

1. **Pass by Reference:**
   * In pass by reference, the function receives a reference or memory address of the actual variable, allowing modifications to affect the original variable.
   * IT IS MUTABLE
2. **Pass by Value:**
   * In pass by value, the function receives a copy of the actual variable's value,it does not allow modifications to affect the original variable.
   * IMMUTABLE

### TYPES OF ARGUMENTS

1. **Positional Arguments:**
   * Positional arguments are values passed to a function in the order in which parameters are defined.
2. **Keyword Arguments:**
   * Keyword arguments are values passed to a function with explicit parameter names, allowing flexibility in the order.
3. **Default Arguments:**
   * Default arguments have predefined values in a function definition and are used when a value for a parameter is not provided.
4. **Variable Number of Arguments (\*args and \*\*kwargs):Arbitrary**
   * **\*args** allows a function to accept a variable number of positional arguments.
   * **\*\*kwargs** allows a function to accept a variable number of keyword arguments.

### RETURN STATEMENT

The "return" statement is used to return a value from a function. It allows you to pass the result of the function back to the caller.

### SCOPE AND LIFETIME OF VARIABLES

The scope and lifetime of variables in Python depend on where they are defined. Variables defined inside a function have local scope and are only accessible within that function. Variables defined outside any function have global scope and can be accessed from anywhere in the code.

### NESTED FUNCTIONS

Python also allows you to nest functions, which means you can define a function inside another function. This allows you to create more complex functionality by combining multiple functions together.

### LAMBDA FUNCTIONS

Lambda functions in Python are also known anonymous functions that can be created using the lambda keyword. They are typically used for small, one-time operations and are written in a single line of code.

### DIFFERENCE BETWEEN LAMBDA AND DEF FUNCTION

The main difference between lambda functions and regular functions defined with the "def" keyword is that lambda functions can only contain a single expression, while regular functions can contain multiple statements.

### LAMBDA WITH FILTER(),MAP()

Lambda functions can be used with built-in functions like filter() and map(). IT IS USED to perform filtering and mapping operations on iterables.

### LAMBDA WITH LIST COMPREHENSION

You can also use lambda functions with list comprehension, It is used to create lists based on existing lists or other iterables.

### LAMBDA WITH IF ELSE

A lambda function with an if-else statement allows for conditional execution within a single expression.

### LAMBDA WITH MULTIPLE STATEMENTS

Lambda functions in Python are designed for conciseness and typically consist of a single expression. While it's possible to simulate multiple statements using a tuple and the **exec** function, it's not a recommended practice due to readability concerns.

### RECURSION IN PYTHON

Recursion in Python is a programming technique where a function calls itself, solving a problem by breaking it down into smaller instances. It involves defining a base case to terminate the recursive calls.

Top of Form

1. **Parameters**

* Parameters are variables that store data.

1. **Arguments:**

* Arguments are the actual values passed to the function when it is called.

## DATA STRUCTURES

A data structure is a way of organizing and storing data to perform operations efficiently. It defines the relationships and rules for organizing the data, allowing for easy retrieval, modification, and manipulation. Common data structures include arrays, linked lists, stacks, queues, trees, and graphs, each suited for specific tasks and algorithms.

### LIST DEFINITION

* A list is a data structure in python programming.
* It is a ordered collection of elements each element has a unique index,starting from 0,which allows for easy retrieval and manipulation.
* It is mutable that means we can modify the elements like added,removed or changed etc.
* A list is enclosed with square brackets and seperated by commas.

**List splitting**---- Splitting a list involves dividing it into multiple parts based on a specific criterion, such as a delimiter.

Access list items

Change list items

Add list items7

Remove list items

Loop lists

List comprehension

## List operators ~~ Python List Operations

The concatenation (+) and repetition (\*) operators work in the same way as they were working with the strings. The different operations of list are

1. Repetition
2. Concatenation
3. Length
4. Iteration
5. Membership

* **Repetition:** Repetition in lists refers to the duplication of elements, defined by multiplication operator (\*)
* **Concatenation:** Concatenation involves combining two or more lists using the concatenation operator (+), resulting in a new list that the elements from both lists in the specified order.
* **Length:** Length, represented by the len() function, provides the count of elements in a list, indicating the size or number of items contained within the list structure.
* **Iteration:** Iteration in lists involves sequentially accessing each element, typically accomplished using loops like 'for' or 'while' to perform operations on each item within the list.
* **Membership:** Membership, determined by the 'in' and 'not in' operators, checks whether an element is present or absent in a list, aiding in conditional evaluations or existence checks within the list structure.

Sort lists

Copy lists0

Join lists

List methods

Linked list

Shallow copy and deep copy

Nested list

**Indexing and Slicing:**

* Indexing involves accessing elements in a list,string, or tuple, each element as a unique index,starting from 0, It allows easy retrieval and manipulation.
* Slicing in Python involves extracting a portion of a sequence by specifying a range of indices. The syntax for slicing includes parameters like **start**, **stop**, and **step**, and it returns a new sequence based on the specified criteria.

**Access List Items:** Accessing list items involves retrieving specific elements by their index position within the list. This enables reading and manipulating individual elements as needed.we can access using indexing or slicing….

**Change List Items:** Changing list items involves modifying the value of an element at a specific index in the list. This allows for dynamic updates to the list content.indexin and slicing

**Add List Items:** Adding list items involves appending new elements to the end of a list or inserting elements at specified positions.

**Remove List Items:** Removing list items involves deleting elements based on their values or indices. This operation helps maintain a concise and updated list.

**Loop Lists:** Looping through lists involves iterating over each element in the list, performing operations or actions for each item. This is a fundamental technique for processing list contents.

**List Comprehension** , It is used to create lists based on existing lists or other iterables. It's a powerful tool for creating lists in a single line of code.

**List Operators:** List operators, such as **+** and **\***, facilitate concatenation and repetition, respectively. These operations offer flexibility in combining and duplicating list elements.

**Sort Lists:** Sorting lists involves arranging elements in ascending or descending order based on their values. This facilitates easier searching and retrieval.

**Copy Lists:** Copying lists allows the creation of independent duplicates, ensuring modifications to one list do not affect another. Understanding shallow and deep copy mechanisms is crucial.

* **shallow\_copy** is created using the **copy()** method, resulting in a new list with references to the same objects as the original list .
* **deep\_copy** is created using the **deepcopy()**method from the **copy** module, resulting in a completely independent copy of the original list and its nested structures .

**Join Lists:** Joining lists involves combining multiple lists into a single list. This can be achieved using concatenation or specialized functions.

**List Methods:** List methods are built-in functions that provide various operations like sorting, appending, and removing elements. These methods enhance the functionality and manipulation of lists.

| [append()](https://www.w3schools.com/python/ref_list_append.asp) | Adds an element at the end of the list |
| --- | --- |
| [clear()](https://www.w3schools.com/python/ref_list_clear.asp) | Removes all the elements from the list |
| [copy()](https://www.w3schools.com/python/ref_list_copy.asp) | Returns a copy of the list |
| [count()](https://www.w3schools.com/python/ref_list_count.asp) | Returns the number of elements with the specified value |
| [extend()](https://www.w3schools.com/python/ref_list_extend.asp) | Add the elements of a list (or any iterable), to the end of the current list |
| [index()](https://www.w3schools.com/python/ref_list_index.asp) | Returns the index of the first element with the specified value |
| [insert()](https://www.w3schools.com/python/ref_list_insert.asp) | Adds an element at the specified position |
| [pop()](https://www.w3schools.com/python/ref_list_pop.asp) | Removes the element at the specified position |
| [remove()](https://www.w3schools.com/python/ref_list_remove.asp) | Removes the item with the specified value |
| [reverse()](https://www.w3schools.com/python/ref_list_reverse.asp) | Reverses the order of the list |
| [sort()](https://www.w3schools.com/python/ref_list_sort.asp) | Sorts the list |

**Linked List:** It is a collection of nodes that are linked with each other. A node contains two things first is data and second is a link that connects it with another node.

**Nested List:** A nested list is a list containing other lists as elements. This structure allows for the representation of multi-dimensional data in a single list.

## STRING

Python strings

String indexing

Slicing strings

String comparison

Modify strings

Concatenate strings

Format strings

Escape characters

String methods

String operators

String conversion

**Python Strings:** Strings in Python are sequences of characters enclosed within single (' ') or double (" ") quotes, used to represent textual data. They are immutable, meaning their content cannot be modified once created.

**String Indexing:** String indexing involves accessing individual characters in a string using their position. Indexing starts at 0 for the first character, and negative indexing counts from the end.

**Slicing Strings:** Slicing allows extracting portions of a string by specifying a range of indices. The syntax is **string[start:stop:step]**, providing flexibility in substring extraction.

**String Comparison:** String comparison involves evaluating the equality or inequality of two strings using relational operators (**==**, **!=**, **<**, **>**, etc.). It considers the lexicographic order of characters.

**Modify Strings:** Modifying strings includes operations like changing individual characters, converting case with methods such as **upper()** and **lower()**, and replacing specific substrings.

**Concatenate Strings:** Concatenation involves combining multiple strings into a single string using the **+** operator. It creates a new string without modifying the original strings.

**Format Strings:** Formatting strings allows dynamic insertion of values into a template. This can be achieved using techniques like f-strings (**f"Hello, {name}"**) or the **format()** method.

**Escape Characters:** Escape characters, such as **\n** for newline and **\"** for double quote, allow the inclusion of special characters within a string. Raw strings (**r"raw string"**) ignore escape characters.

**String Methods:** String methods are built-in functions that operate on strings, including **len()** for length, **find()** for substring location, and **replace()** for replacing substrings.

| **Method** | **Description** |
| --- | --- |
| [capitalize()](https://www.w3schools.com/python/ref_string_capitalize.asp) | Converts the first character to upper case |
| [casefold()](https://www.w3schools.com/python/ref_string_casefold.asp) | Converts string into lower case |
| [center()](https://www.w3schools.com/python/ref_string_center.asp) | Returns a centered string |
| [count()](https://www.w3schools.com/python/ref_string_count.asp) | Returns the number of times a specified value occurs in a string |
| [encode()](https://www.w3schools.com/python/ref_string_encode.asp) | Returns an encoded version of the string |
| [endswith()](https://www.w3schools.com/python/ref_string_endswith.asp) | Returns true if the string ends with the specified value |
| [expandtabs()](https://www.w3schools.com/python/ref_string_expandtabs.asp) | Sets the tab size of the string |
| [find()](https://www.w3schools.com/python/ref_string_find.asp) | Searches the string for a specified value and returns the position of where it was found |
| [format()](https://www.w3schools.com/python/ref_string_format.asp) | Formats specified values in a string |
| format\_map() | Formats specified values in a string |
| [index()](https://www.w3schools.com/python/ref_string_index.asp) | Searches the string for a specified value and returns the position of where it was found |
| [isalnum()](https://www.w3schools.com/python/ref_string_isalnum.asp) | Returns True if all characters in the string are alphanumeric |
| [isalpha()](https://www.w3schools.com/python/ref_string_isalpha.asp) | Returns True if all characters in the string are in the alphabet |
| [isascii()](https://www.w3schools.com/python/ref_string_isascii.asp) | Returns True if all characters in the string are ascii characters |
| [isdecimal()](https://www.w3schools.com/python/ref_string_isdecimal.asp) | Returns True if all characters in the string are decimals |
| [isdigit()](https://www.w3schools.com/python/ref_string_isdigit.asp) | Returns True if all characters in the string are digits |
| [isidentifier()](https://www.w3schools.com/python/ref_string_isidentifier.asp) | Returns True if the string is an identifier |
| [islower()](https://www.w3schools.com/python/ref_string_islower.asp) | Returns True if all characters in the string are lower case |
| [isnumeric()](https://www.w3schools.com/python/ref_string_isnumeric.asp) | Returns True if all characters in the string are numeric |
| [isprintable()](https://www.w3schools.com/python/ref_string_isprintable.asp) | Returns True if all characters in the string are printable |
| [isspace()](https://www.w3schools.com/python/ref_string_isspace.asp) | Returns True if all characters in the string are whitespaces |
| [istitle()](https://www.w3schools.com/python/ref_string_istitle.asp) | Returns True if the string follows the rules of a title |
| [isupper()](https://www.w3schools.com/python/ref_string_isupper.asp) | Returns True if all characters in the string are upper case |
| [join()](https://www.w3schools.com/python/ref_string_join.asp) | Joins the elements of an iterable to the end of the string |
| [ljust()](https://www.w3schools.com/python/ref_string_ljust.asp) | Returns a left justified version of the string |
| [lower()](https://www.w3schools.com/python/ref_string_lower.asp) | Converts a string into lower case |
| [lstrip()](https://www.w3schools.com/python/ref_string_lstrip.asp) | Returns a left trim version of the string |
| [maketrans()](https://www.w3schools.com/python/ref_string_maketrans.asp) | Returns a translation table to be used in translations |
| [partition()](https://www.w3schools.com/python/ref_string_partition.asp) | Returns a tuple where the string is parted into three parts |
| [replace()](https://www.w3schools.com/python/ref_string_replace.asp) | Returns a string where a specified value is replaced with a specified value |
| [rfind()](https://www.w3schools.com/python/ref_string_rfind.asp) | Searches the string for a specified value and returns the last position of where it was found |
| [rindex()](https://www.w3schools.com/python/ref_string_rindex.asp) | Searches the string for a specified value and returns the last position of where it was found |
| [rjust()](https://www.w3schools.com/python/ref_string_rjust.asp) | Returns a right justified version of the string |
| [rpartition()](https://www.w3schools.com/python/ref_string_rpartition.asp) | Returns a tuple where the string is parted into three parts |
| [rsplit()](https://www.w3schools.com/python/ref_string_rsplit.asp) | Splits the string at the specified separator, and returns a list |
| [rstrip()](https://www.w3schools.com/python/ref_string_rstrip.asp) | Returns a right trim version of the string |
| [split()](https://www.w3schools.com/python/ref_string_split.asp) | Splits the string at the specified separator, and returns a list |
| [splitlines()](https://www.w3schools.com/python/ref_string_splitlines.asp) | Splits the string at line breaks and returns a list |
| [startswith()](https://www.w3schools.com/python/ref_string_startswith.asp) | Returns true if the string starts with the specified value |
| [strip()](https://www.w3schools.com/python/ref_string_strip.asp) | Returns a trimmed version of the string |
| [swapcase()](https://www.w3schools.com/python/ref_string_swapcase.asp) | Swaps cases, lower case becomes upper case and vice versa |
| [title()](https://www.w3schools.com/python/ref_string_title.asp) | Converts the first character of each word to upper case |
| [translate()](https://www.w3schools.com/python/ref_string_translate.asp) | Returns a translated string |
| [upper()](https://www.w3schools.com/python/ref_string_upper.asp) | Converts a string into upper case |
| [zfill()](https://www.w3schools.com/python/ref_string_zfill.asp) | Fills the string with a specified number of 0 values at the beginning |

**String Operators:** String operators include concatenation (**+**), repetition (**\***), and membership (**in**, **not in**). They provide various ways to manipulate and analyze strings.

**String Conversion:** String conversion involves converting other data types to strings using the **str()** function. This is useful for displaying non-string values in a textual format.

## TUPLE

* A Tuple is a data structure in python programming.
* It is a ordered collection of elements each element has a unique index,starting from 0,which allows for easy retrieval and manipulation.
* It is immutable that means we cannot modify the elements like added,removed or changed etc.
* A tuple is enclosed with paranthasis and seperated by commas.

Top of Form

1. **Access Tuples:**
   * Accessing elements in a tuple refers to retrieving values from the tuple using their position or index.
2. **Update Tuples:**
   * Updating tuples involves creating a new tuple with modified values, as tuples are immutable and cannot be changed in-place.
3. **Unpack Tuples:**
   * Unpacking tuple refers to extracting values from a packed tuple and assigning them to separate variables.
4. **Loop Tuples:**
   * Looping through tuples means iterating over each element in the tuple to perform some operation.
5. **Join Tuples:**
   * Joining tuples typically involves concatenating or combining multiple tuples to create a new tuple.
6. **Tuple Methods:**
   * Tuple methods are built-in functions that can be applied to tuples to perform specific operations. Examples include **count()** and **index()**.

| [count()](https://www.w3schools.com/python/ref_tuple_count.asp) | Returns the number of times a specified value occurs in a tuple |
| --- | --- |
| [index()](https://www.w3schools.com/python/ref_tuple_index.asp) | Searches the tuple for a specified value and returns the position of where it was found |

1. **Nested Tuples:**
   * Tuples can contain other tuples as elements, allowing for the creation of nested structures.
2. **Tuple Packing:**
   * The process of creating a tuple without using parentheses, where values are automatically packed into a tuple.
3. **Tuple Slicing:**
   * Extracting a portion of a tuple by specifying a range of indices.
4. **Tuple Comparison:**
   * Tuples can be compared element-wise, providing a means to determine their relative order.
5. **Length of a Tuple:**
   * Obtaining the number of elements in a tuple using the **len()** function.

**12.Tuple Comprehension:**

* Creating tuples using a concise and expressive syntax similar to list comprehensions.

## SET

### SET

A set in Python is an unordered collection of unique elements. You can access, add, and remove items from a set. Sets support iteration, and operations like union and intersection, and have built-in methods for various set-related tasks.

A SET IS A UNORDERED COLLECTION OF ELEMENTS AND IT IS MUTABLE THAT MEANS WE CAN MODIFY THEM AFTER CREATION BY ADDING OR REMOVING ELEMENTS AND IT DOES NOT ALLOW DUPLICATES AND IS ENCLOSED WITH FLOWER BRACKETS……..

1. **Access Set Items:**
   * Iterating through a set to retrieve and work with each element using loops or the **in** keyword.
2. **Add Set Items:**
   * Inserting new elements into a set using the **add()** method.
3. **Remove Set Items:**
   * Deleting elements from a set using methods like **remove()** or **discard()**.
4. **Loop Sets:**
   * Iterating through the elements of a set using loops to perform operations on each element.
5. **Join Sets:**
   * Combining two or more sets to create a new set containing all unique elements, often using methods like **union()**.
6. **Set Methods:**
   * Utilizing built-in methods for set operations such as **intersection()**, **difference()**, and **symmetric\_difference()** to manipulate sets.

| [add()](https://www.w3schools.com/python/ref_set_add.asp) | Adds an element to the set |
| --- | --- |
| [clear()](https://www.w3schools.com/python/ref_set_clear.asp) | Removes all the elements from the set |
| [copy()](https://www.w3schools.com/python/ref_set_copy.asp) | Returns a copy of the set |
| [difference()](https://www.w3schools.com/python/ref_set_difference.asp) | Returns a set containing the difference between two or more sets |
| [difference\_update()](https://www.w3schools.com/python/ref_set_difference_update.asp) | Removes the items in this set that are also included in another, specified set |
| [discard()](https://www.w3schools.com/python/ref_set_discard.asp) | Remove the specified item |
| [intersection()](https://www.w3schools.com/python/ref_set_intersection.asp) | Returns a set, that is the intersection of two other sets |
| [intersection\_update()](https://www.w3schools.com/python/ref_set_intersection_update.asp) | Removes the items in this set that are not present in other, specified set(s) |
| [isdisjoint()](https://www.w3schools.com/python/ref_set_isdisjoint.asp) | Returns whether two sets have a intersection or not |
| [issubset()](https://www.w3schools.com/python/ref_set_issubset.asp) | Returns whether another set contains this set or not |
| [issuperset()](https://www.w3schools.com/python/ref_set_issuperset.asp) | Returns whether this set contains another set or not |
| [pop()](https://www.w3schools.com/python/ref_set_pop.asp) | Removes an element from the set |
| [remove()](https://www.w3schools.com/python/ref_set_remove.asp) | Removes the specified element |
| [symmetric\_difference()](https://www.w3schools.com/python/ref_set_symmetric_difference.asp) | Returns a set with the symmetric differences of two sets |
| [symmetric\_difference\_update()](https://www.w3schools.com/python/ref_set_symmetric_difference_update.asp) | inserts the symmetric differences from this set and another |
| [union()](https://www.w3schools.com/python/ref_set_union.asp) | Return a set containing the union of sets |
| [update()](https://www.w3schools.com/python/ref_set_update.asp) | Update the set with the union of this set and others |

## DICTIONARIES

### Dictionaries

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered\*, changeable and do not allow duplicates.

A DICTIONARY IS A ORDERED COLLECTION OF ELEMENTS AND IT IS MUTABLE THAT MEANS WE CAN MODIFY THEM AFTER CREATION AND IT DOES NOT ALLOW DUPLICATES AND IS ENCLOSED WITH CURLY BRACKETS IT IS USED TO STORE IN KEY VALUE PAIRS….……..

1. **Access Items:**
   * Retrieving values from a dictionary using their corresponding keys.
2. **Change Items:**
   * Modifying the values of existing key-value pairs in a dictionary.
3. **Add Items:**
   * Inserting new key-value pairs into a dictionary.
4. **Remove Items:**
   * Deleting key-value pairs from a dictionary using methods like **pop()** or **del**.
5. **Loop Dictionaries:**
   * Iterating through key-value pairs in a dictionary using loops.
6. **Copy Dictionaries:**
   * Creating a shallow or deep copy of a dictionary using the **copy()** method.
7. **Nested Dictionaries:**
   * Creating dictionaries within dictionaries to represent hierarchical or nested data structures.
8. **Dictionary Methods:**
   * Utilizing built-in methods like **keys()**, **values()**, and **items()** to retrieve keys, values, and key-value pairs, respectively.

| **Method** | **Description** |
| --- | --- |
| [clear()](https://www.w3schools.com/python/ref_dictionary_clear.asp) | Removes all the elements from the dictionary |
| [copy()](https://www.w3schools.com/python/ref_dictionary_copy.asp) | Returns a copy of the dictionary |
| [fromkeys()](https://www.w3schools.com/python/ref_dictionary_fromkeys.asp) | Returns a dictionary with the specified keys and value |
| [get()](https://www.w3schools.com/python/ref_dictionary_get.asp) | Returns the value of the specified key |
| [items()](https://www.w3schools.com/python/ref_dictionary_items.asp) | Returns a list containing a tuple for each key value pair |
| [keys()](https://www.w3schools.com/python/ref_dictionary_keys.asp) | Returns a list containing the dictionary's keys |
| [pop()](https://www.w3schools.com/python/ref_dictionary_pop.asp) | Removes the element with the specified key |
| [popitem()](https://www.w3schools.com/python/ref_dictionary_popitem.asp) | Removes the last inserted key-value pair |
| [setdefault()](https://www.w3schools.com/python/ref_dictionary_setdefault.asp) | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| [update()](https://www.w3schools.com/python/ref_dictionary_update.asp) | Updates the dictionary with the specified key-value pairs |

## OOPS:

In Python, Object-Oriented Programming (OOP) involves creating classes and objects to model real-world entities, promoting concepts like inheritance, polymorphism, and encapsulation. Data abstraction etc.

### 1. Class Definition:

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

#### **Attributes and Methods:**

* **Attributes:** These Attributes represent the characteristics or properties of the object. They are variables that store data.
* **Methods:** Methods are functions within a class or behavior of the class.

#### **Constructor (**\_\_init\_\_ **method):**

* The **\_\_init\_\_** method is a special method used for initializing object attributes. It is called when an object is created.

#### **Instance and Class Variables static:**

* **Instance Variables:** These variables are specific to each instance of the class. They are defined inside the **\_\_init\_\_** method.
* **Class Variables:** These variables are shared among all instances of the class. They are defined outside any method in the class.

#### **Instance and Class Methods:**

* **Instance Methods:** **Instace methods can access both class and instance variables**
* **Class Methods:** **Class methods can access only for class variables.**

### 2. Object Instantiation:

* **Definition:** Objects are instances of classes. They represent real-world entities and have attributes (characteristics) and methods (behaviors) of the class….

#### **Creating Objects:**

* **Creating Objects:** Objects are instances of a class. Creating an object involves calling the class as if it were a function, which is known as instantiation.

#### **Initializing Object Attributes:**

* **Initializing Object Attributes:** The **\_\_init\_\_** method in a class is used to initialize the attributes of an object when it is created. It sets the initial state of the object.

#### **Accessing Object Attributes:**

* **Accessing Object Attributes:** Object attributes are accessed using dot notation. This involves specifying the object followed by a dot and then the attribute name.

#### **Invoking Object Methods or calling:**

* **Invoking Object Methods:** Methods are functions defined within a class. Invoking a method on an object means calling that method on the object.

### Inheritance:

* **INHERITANCE IS THE PROCESS OF CREATING A NEW CLASS FROM AN EXISTING CLASS**
* **INHERITANCE LEADS TO THE CONCEPT OF REUSABILITY WHICH MEANS THE FEATURES OF THE BASE CLASS CAN BE REUSED IN THE DERIVED CLASS WITHOUT MODIFYING IT**
* **Inheritance include SINGLE,MUTLILEVEL,MULTIPLE,HIERARCHICAL,HYBRID**

1. **Single Inheritance:**
   * **Definition:** Single inheritance is a type of inheritance where a subclass inherits from only one superclass, forming a linear relationship.
2. **Multiple Inheritance:**
   * **Definition:** Multiple inheritance is a type of inheritance where a subclass inherits from more than one superclass, forming a multiple relationship.
3. **Multilevel Inheritance:**
   * **Definition:** Multilevel inheritance is a type of inheritance where a subclass inherits from another subclass, creating a chain formation of inheritance.
4. **Hierarchical Inheritance:**
   * **Definition:** Hierarchical inheritance is a type of inheritance where multiple subclasses inherit from a common superclass, forming a tree-like structure.
5. **Hybrid (or Virtual) Inheritance:**
   * **Definition:** Hybrid (or Virtual) inheritance is a combination of multiple types of inheritance ,providing flexibility in class relationships.

### 4. Encapsulation:

* **Definition:** "Encapsulation is the bundling of data and methods into a single unit of a class.
* **Encapsulation leads to the concept of data hiding which means data is prevented from an external access**

#### **Access Modifiers (Public, Private, Protected):**

* **Access Modifiers:** Access modifiers are used to control the visibility of attributes and methods in a class.
  + **Public:** Accessible from anywhere.
  + **Private:** Accessible only within the class.
  + **Protected:** Accessible within the class and its subclasses.

#### **Property Decorators:**

* **Property Decorators:** Property decorators in Python are used to define getter, setter, and deleter methods for class attributes. They provide a way to encapsulate attribute access and modification.

#### **Getter and Setter Methods:**

* **Getter and Setter Methods:** Getter methods retrieve the value of an attribute, while setter methods set the value of an attribute. They are used to encapsulate the access and modification of class attributes.

### 5. Polymorphism:

* **POLY MEANS MORE THAN ONE MORPHISM MEANS DIFFERENT FORM**….
* **Definition:** Polymorphism is the ability of a class to take on multiple forms. In programming, it allows objects of different types to be treated as objects of a common base type.

#### **Method Overloading:**

* Method overloading is a feature in object-oriented programming that allows a class to have multiple methods with the same name but different parameters.
* It Occurs within a single class
* No relationship required between the overloaded methods
* It is a form of compile-time polymorphism

#### **Method Overriding:**

* Method overriding is a feature in object-oriented programming that allows a class to have method in the child class with the same name and parameters as a method in the parent class
* Method Overriding Occurs between a parent and child class
* Requires an inheritance relationship between the parent and child classes.
* It is a form of runtime polymorphism

### 6. Abstraction:

Data abstraction is the process of hiding the complex implementation details and showing only the essential features of an object. It involves the use of abstract classes and abstract methods.

## Abstract Classes:

## Abstract classes are blueprints for subclasses, providing a common structure and required methods.

## Abstract classes can have both regular methods with implementations and abstract methods without implementations.

## Abstract Methods:

## Abstract methods are methods declared in an abstract class that do not have an implementation.

## Subclasses of the abstract class must provide an implementation for these abstract methods.

## Abstract methods are defined using the `@abstractmethod` decorator.

## Interfaces:

* Interfaces in Python are a way to define a contract or a set of methods that a class must implement.
* They provide a way to achieve polymorphism and enforce consistency in object-oriented programming.

## FILE HANDELING

File handling in programming refers to the process of working with files, which includes creating, reading, updating, and deleting data from files. This is a crucial aspect of many applications as it allows them to interact with external storage, such as hard drives, to store and retrieve information.

1. **Opening and Closing Files:**
   * To work with files, you use the **open()** function. It takes two parameters: the file name and the mode ('r' for reading, 'w' for writing, 'a' for appending, etc.).
   * It's crucial to close files after you've finished working with them using the **close()** method. Using the **with** statement (**with open(...) as file:**) is a recommended way to automatically close the file when done.
2. **Reading from Files:**
   * Reading from files can be done using methods like **read()**, which reads the entire content, or **readline()**, which reads one line at a time. The **readlines()** method can be used to read all lines into a list.
3. **Writing to Files:**
   * To write to a file, use the **write()** method or **writelines()** to write a sequence of lines.
4. **Appending to Files:**
   * If you want to add content to an existing file without overwriting it, open the file in append mode ('a').
5. **Modes:**
   * The **open()** function supports various modes such as 'r' (read), 'w' (write), 'a' (append), 'b' (binary), and combinations like 'rb' or 'r+'.
6. **Handling Exceptions:**
   * It's good practice to handle exceptions when working with files. Common exceptions include **FileNotFoundError** if the file doesn't exist and **PermissionError** if there are permission issues.

## EXPECTION HANDELING

Exception handling is also known as error handling is a critical aspect of programming that involves dealing with errors and unexpected situations.

* **Try Block:** Contains code that may raise an exception.
* **Except Block:** Catches and handles exceptions raised in the try block.
* **Else Block:** Executes if no exceptions occurred in the try block.
* **Finally Block:** Always executes, regardless of exception occurred or not. It is useful for cleanup operations.

1. **Handling Multiple Exceptions:**
   * You can use a single **except** block to handle multiple exceptions or use a tuple in the **except** statement.
2. **Generic Exception Handling or hierarchy:**
   * Using a generic **except** block (**except Exception as e:**) catches any exception. However, it's generally better to be specific about the exceptions you catch.
3. **Raising Exceptions:**
   * You can use the **raise** statement to manually raise an exception. This can be useful in situations where you want to indicate an error based on certain conditions.
4. **Custom Exceptions:**
   * You can create custom exceptions by defining a new class that inherits from the built-in **Exception** class. This allows you to raise and handle exceptions specific to your application.
5. **Exception Chaining or propagation:**
   * Python allows exceptions to be chained using the **from** keyword. This can be useful for preserving the context of an original exception while raising a new one.

## ITERATORS

An iterator is an object that defines how to access a sequence of elements, one at a time, using the **\_\_iter\_\_** method to obtain the iterator itself and the **\_\_next\_\_** method to retrieve the next element in the sequence. When there are no more elements, it raises the **StopIteration** exception.

Certainly! Here are definitions for the mentioned Python iterator subtopics:

1. **iter() and next():**
   * **iter()**: A built-in function in Python used to obtain an iterator from an iterable.
   * **next()**: A built-in function that retrieves the next item from an iterator.
2. **StopIteration Exception:**
   * **StopIteration**: An exception raised by the **\_\_next\_\_()** method to signal the end of iteration.
3. **Iterating with for Loop:**
   * The **for** loop in Python internally uses iterators. It calls **iter()** on an iterable and repeatedly calls **next()** to get the next item until **StopIteration** is raised.
4. **Creating Custom Iterators:**
   * Creating custom iterators involves implementing the **\_\_iter\_\_()** and **\_\_next\_\_()** methods in a class.
5. **itertools Module:**
   * The **itertools** module in Python provides tools for working with iterators, including functions like **count()**, **cycle()**, and **chain()**.
6. **Generator Functions:**
   * Generator functions are a concise way to create iterators using the **yield** keyword to produce a series of values lazily.
7. **Built-in Functions for Iterators:**
   * Certain built-in functions like **all()**, **any()**, and **sum()** work with iterators, taking advantage of the iterator protocol.

## GENERATORS

Generators are used to create iterators. And it is a special type of iterator are created using a generator function. Generators are defind with **yield** keyword to produce a sequence of values lazily, allowing for efficient memory usage.

1. **yield Statement:**
   * The **yield** statement is used in generator functions to produce a value and temporarily suspend the function's state. It facilitates the creation of iterators.
2. **Lazy Evaluation:**
   * Generators implement lazy evaluation, generating values on-demand rather than creating the entire sequence upfront. This can be memory-efficient for large datasets.
3. **Generator Expressions:**
   * Generator expressions provide a concise way to create generators using a syntax similar to list comprehensions, using parentheses instead of square brackets.
4. **next() Function:**
   * The **next()** function retrieves the next value from a generator. It automatically resumes the generator function until the next **yield** statement is encountered.
5. **Infinite Generators:**
   * Generators can represent infinite sequences, allowing for the creation of sequences without the need to store all values in memory.
6. **StopIteration Exception:**
   * When a generator function completes, it raises a **StopIteration** exception, indicating the end of the sequence. This exception is typically handled automatically by Python.
7. **Generator State:**
   * Generators maintain their state between successive calls. The state includes local variables and the position of the **yield** statement, allowing the function to resume execution where it left off.

## DECORATORS

A decorator in Python is a function that takes another function as input, it extends or modifies its behavior of the function, and returns a new function. Decorators are defined using the **@decorator** allowing them to enhance the functionality of functions or methods.

1. **Function Decorators:**
   * Function decorators are a way to modify or extend the behavior of functions without modifying their code directly. They are applied using the **@decorator** syntax.
2. **Decorator Syntax:**
   * Decorators use the **@decorator** syntax, where **decorator** is a function that takes another function as input and typically returns a new function.
3. **Common Use Cases:**
   * Decorators are commonly used for tasks such as logging, timing, access control, and caching. They help separate concerns and enhance code modularity.
4. **Chaining Decorators:**
   * Multiple decorators can be applied to a single function, forming a chain of decorators. The order in which decorators are applied can affect the overall behavior.
5. **Class Decorators:**
   * Class decorators are applied to classes and can modify or extend their behavior.
6. **Built-in Decorators:**
   * Python includes several built-in decorators, such as **@staticmethod**, **@classmethod**, and **@property**. These decorators provide specific functionality for methods in classes.

# Coding questions

1. [Write a program to reverse an integer in Python.](https://quescol.com/interview-preparation/python-program-to-reverse-a-number)
2. [Write a program in Python to check whether an integer is Armstrong number or not.](https://quescol.com/interview-preparation/armstrong-program-in-python)

It is a number that is equal to the sum of its own digits raised to the power of the number of digits.

1. [Write a program in Python to check given number is prime or not.](https://quescol.com/interview-preparation/python-program-to-check-prime-number)

In Mathematical terms, A prime number is a natural number greater than 1 that can be divided by only 1 and the number itself.

1. [Write a program in Python to print the Fibonacci series using iterative method.](https://quescol.com/interview-preparation/fibonacci-series-in-python-program)

A Fibonacci series is a series in which next number is a sum of previous two numbers.

1. [Write a program in Python to print the Fibonacci](https://quescol.com/interview-preparation/fibonacci-series-in-python-program-using-recursive-method) [series using recursive method.](https://quescol.com/interview-preparation/fibonacci-series-in-python-program-using-recursive-method)
2. [Write a program in Python to check whether a number is palindrome or not using iterative method.](https://quescol.com/interview-preparation/palindrome-python-program-using-iterative-method)

A Palindrome number is a number which reverse is equal to the original number means number itself.