***Introduction to Python and its Features***

ANS).

**INTRODUCTION**

* Python is a programming language that is widely used in web applications, software development, data science, and machine learning (ML).
* Python is a versatile and popular programming language known for its readability and ease of use.
* Python is a dynamic, interpreted (bytecode-compiled) language.

**FEATURES**

1. **Readability:** Python's syntax is similar to English, making it easy to read and understand, even for beginners.
2. **Versatility:** Python can be used for a wide range of applications, including web development, data analysis, machine learning, and more.
3. **Large Standard Library**: Python comes with a rich set of built-in functions and modules that provide support for various tasks.
4. **Dynamic Typing:** Python automatically determines the data type of a variable at runtime, simplifying code writing.
5. **Cross-Platform:** Python runs on various operating systems like Windows, macOS and Linux.
6. **Object-Oriented Programming (OOP):** Python supports OOP principles like encapsulation, inheritance, and polymorphism.
7. **Open Source:** Python is free to use, distribute, and modify.

***History and evolution of Python***

ANS).

**HISTORY**

Python's history begins in the late 1980s with Guido van Rossum, a Dutch programmer, working on a hobby project at CWI in the Netherlands. He aimed to create a successor to the ABC programming language.

**EVOLUTION**

Python's evolution, starting in the late 1980s, has seen it grow from a successor to the ABC language to a widely used programming language. It began with version 0.9.0 in 1991 and continues to evolve, with Python 3.0 (released in 2008) marking a major transition. The language's evolution has been driven by the goal of making code more readable and developer productivity more advanced, with notable features and changes introduced in various versions.

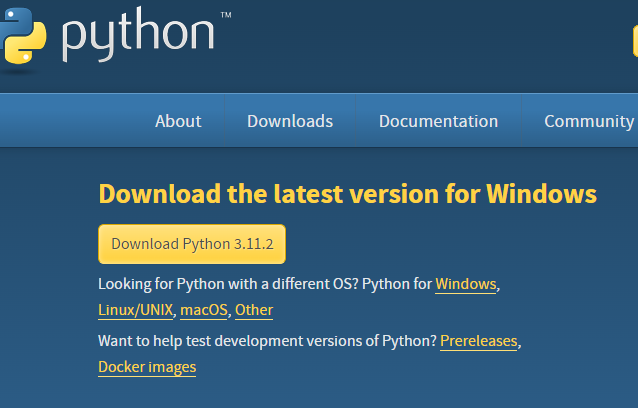
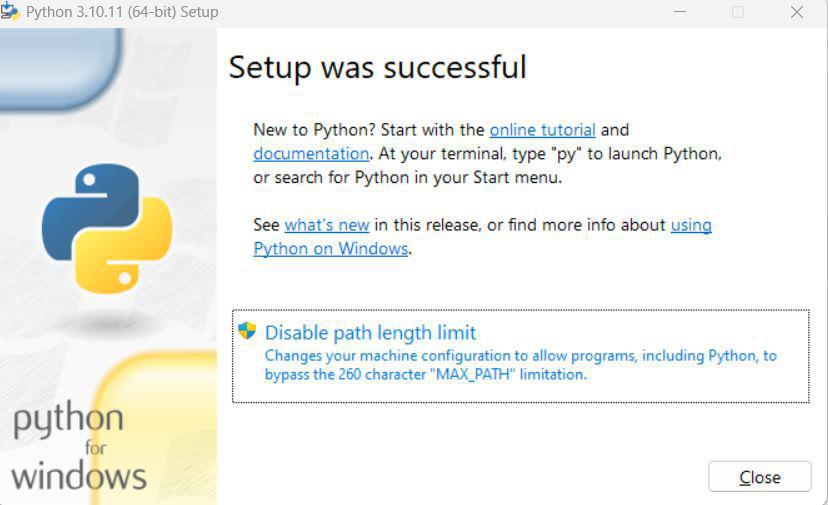
***Advantages of using Python over other programming languages***

Python offers numerous advantages over other programming languages due to its simplicity, versatility, extensive libraries, and strong community support. Its readability, easy-to-learn syntax, and open-source nature make it a popular choice for various applications, including web development, data science, and machine learning.

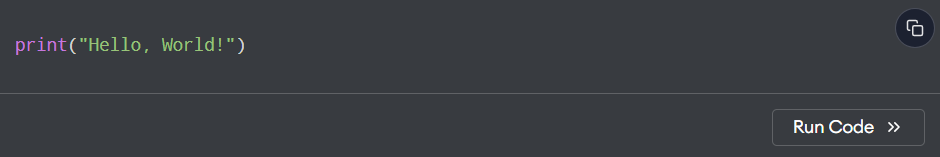
Python's vast ecosystem of libraries and frameworks also provides pre-built solutions for diverse tasks, including data analysis, machine learning, and web development. Furthermore, its cross-platform compatibility and open-source nature ensure portability and widespread accessibility.

The strong community support and numerous online resources further contribute to Python's appeal, making it a versatile and effective language for both beginners and experienced developers.

***Install Python for Windows***

1. Go to the python official website at <https://www.python.org/downloads/windows/>
2. Choose the latest version of Python releases for Windows. 
3. After choosing the correct released version, Click on the download Python.
4. Click on Install now, and you can add python.exe path. 
5. After the installation completed , You will find python is installed in your system . 
6. After a successful installation of Python, IDLE(Integrated Development and Learning Environment) will also be installed on our local computer alongside some of the packages. For simple programs, we can use IDLE.

***Writing and executing your first Python program***



Output:



***Understanding Python’s PEP 8 guidelines***

* PEP 8 is the official style guide for Python code, aiming to improve readability and consistency.
* provides rules for code formatting, naming conventions, and whitespace, among other things.
* Adhering to PEP 8 makes Python code easier to understand, maintain, and collaborate on.

Here's a breakdown of key aspects of PEP 8:

1. Code Layout
2. Naming Conventions
3. Whitespace
4. Comments
5. Other Recommendations

***Indentation, comments, and naming conventions in Python***

**indentation**

Python relies on indentation to define code blocks, unlike languages that use curly braces. Consistent indentation is crucial, typically using four spaces per level. Mixing tabs and spaces can cause errors.

**comments**

Comments are used to explain code and are ignored by the interpreter. Single-line comments start with #, while multi-line comments can be created using triple quotes ''' or """.

**naming conventions**

Naming conventions in Python promote readability and consistency. Variables and functions should use lowercase with words separated by underscores (snake\\_case). Classes should use CamelCase (CapWords). Constants should be in all uppercase with underscores.

***Writing readable and maintainable code***

Follow this rules

* Adhere to PEP 8 Guidelines
* Descriptive and Concise Variable Names
* Use List Comprehensions and Generator Expressions
* Leverage Python’s Built-in Functions and Libraries
* Follow the DRY Principle (Don't Repeat Yourself)
* Use Virtual Environments
* Write Unit Tests
* Use Meaningful Comments and Docstrings
* Handle Exceptions Gracefully
* Keep Your Code Modular

***Understanding data types***

**Integers**

The integer data type basically represents whole numbers (no fractional parts). The integer values jump from one value to another. There is nothing between 6 and 7. It could be asked why not make all your numbers floating point which allow for fractional parts.

**Floats**

a float is a data type used to represent numbers with decimal points (also known as floating-point numbers), allowing for fractional values and a wider range of numerical representation than integers.

**Strings**

a string is a fundamental data type used to represent sequences of characters, such as letters, numbers, and symbols. Strings are essential for handling textual data and are used in various programming tasks.

**Lists**

lists are a versatile and fundamental data type used to store collections of items.

**Tuples**

Tuples are a fundamental data type in Python used to store an ordered collection of items. They are similar to lists but with a key difference: tuples are immutable, meaning their contents cannot be changed after creation.

**Dictionaries**

Dictionaries are a fundamental data type in Python used for storing collections of data in key-value pairs. They are similar to real-world dictionaries, where you look up a word (the key) to find its definition (the value).

***Python variables and memory allocation***

**Python uses two main memory areas:**

**Heap:**

This is where objects are stored. When you create an object, like a list, a string, or a custom object, Python allocates memory on the heap to store that object.

**Stack:**

The stack is used for storing function call information and local variables within a function. When a function is called, a new stack frame is created to store the function's local variables.

***Python operators:***

**Arithmetic:**

Python operators are fundamental for performing mathematical calculations. Arithmetic operators are symbols used to perform mathematical operations on numerical values. Arithmetic operators include addition (+), subtraction (-), multiplication (\*), division (/), and modulus (%).

| **Operator** | **Description** | **Syntax** |
| --- | --- | --- |
| **+** | **Addition:**adds two operands | x + y |
| **–** | **Subtraction:**subtracts two operands | x – y |
| **\*** | **Multiplication:**multiplies two operands | x \* y |
| **/** | **Division (float):**divides the first operand by the second | x / y |
| **//** | **Division (floor):**divides the first operand by the second | x // y |
| **%** | **Modulus:**returns the remainder when the first operand is divided by the second | x % y |
| **\*\*** | **Power:**Returns first raised to power second | x \*\* y |

**Comparison**

**Comparison operators are used to compare two values:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Name | Example |  |
| == | Equal | x == y |  |
| != | Not equal | x != y |  |
| > | Greater than | x > y |  |
| < | Less than | x < y |  |
| >= | Greater than or equal to | x >= y |  |
| <= | Less than or equal to | x <= y |  |

**Logical**

**Logical operators are used to combine conditional statements:**

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Example |  |
| and | Returns True if both statements are true | x < 5 and  x < 10 |  |
| or | Returns True if one of the statements is true | x < 5 or x < 4 |  |
| not | Reverse the result, returns False if the result is true | not(x < 5 and x <10) |  |

**Bitwise**

Python bitwise operators are used to perform bitwise calculations on integers. The integers are first converted into binary and then operations are performed on each bit or corresponding pair of bits, hence the name bitwise operators. The result is then returned in decimal format.

| **Operator** | **Description** | **Syntax** |
| --- | --- | --- |
| **&** | Bitwise AND | x & y |
| **|** | Bitwise OR | x | y |
| **~** | Bitwise NOT | ~x |
| **^** | Bitwise XOR | x ^ y |
| **>>** | Bitwise right shift | x>> |
| **<<** | Bitwise left shift | x<< |

***Introduction to conditional statements : if, else, elif***

**If :**

If the simple code of block is to be performed if the condition holds true then the if statement is used. Here the condition mentioned holds then the code of the block runs otherwise not.

**Else :**

if...else statement is a control statement that helps in decision-making based on specific conditions. When the if condition is False. If the condition in the if statement is not true, the else block will be executed.

**Elif :**

 if-else conditions allow us to control the flow of execution based on certain conditions. While traditional if-else statements are usually written across multiple lines,  python offers a more compact and elegant way to express these conditions on a single line.

**Nested if-else conditions :**

if statement can also be checked inside other if statement. This conditional statement is called a nested if statement. This means that inner if condition will be checked only if outer if condition is true and by this, we can see multiple conditions to be satisfied.

***Introduction to for and while loop***

**For loop :**

Python For Loops are used for iterating over a sequence like lists, tuples, strings, and ranges.

* For loop allows you to apply the same operation to every item within loop.
* Using For Loop avoid the need of manually managing the index.
* For loop can iterate over any iterable object, such as dictionary, list or any custom iterators.

**While loop :**

Python 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.

***How loops work in Python***

In Python, loops are control structures that allow you to execute a block of code repeatedly. There are two main types of loops: for loops and while loops.

**For Loops:**

• for loops are used to iterate over a sequence (such as a list, tuple, string, or range) or other iterable objects.

• The loop executes the code block once for each item in the sequence.

• The syntax is for item in sequence:.

• For example:

for i in range(5):  
 print(i) # Output: 0, 1, 2, 3, 4

**While Loops:**

• while loops execute a block of code repeatedly as long as a condition is true.

• The loop continues until the condition becomes false.

• The syntax is while condition:.

• For example:

count = 0

while count < 5:

print(count)

count += 1 # Output: 0, 1, 2, 3, 4

**Loop Control Statements:**

• break: Terminates the loop prematurely.

• continue: Skips the current iteration and moves to the next.

• pass: Acts as a placeholder when a statement is syntactically required but no action is needed.

**Other Loop Concepts:**

• Nested loops: Loops inside other loops are used for iterating over multi-dimensional data structures.

• Loop else clause: Executes after a loop finishes normally (not terminated by break).

• Enumerate(): Adds a counter to an iterable object, useful for accessing both index and value.

• List comprehensions: A concise way to create lists using a loop inside square brackets.

***Using loops with collections (lists, tuples, etc.)***

Loops are fundamental for iterating over collections in Python, such as lists, tuples, and strings. Here's how you can use them:

**1. For Loops:**

* **Basic Iteration:** The for loop is used to iterate through each item in a collection.

Python

my\_list = [1, 2, 3, 4, 5]  
 for number in my\_list:  
 print(number)

* **Iterating with Index:** You can also use range and len to iterate through the index of elements.

Python

my\_list = ["apple", "banana", "cherry"]  
 for i in range(len(my\_list)):  
 print(i, my\_list[i])

Iterating through tuples.

Python

my\_tuple = ("red", "green", "blue")  
 for color in my\_tuple:  
 print(color)

* **Nested Loops:** You can have loops inside loops to iterate over nested collections.

Python

list\_of\_lists = [[1, 2], [3, 4], [5, 6]]  
 for inner\_list in list\_of\_lists:  
 for num in inner\_list:  
 print(num)

**2. While Loops:**

* **Iteration with Condition:** while loops continue executing as long as a condition is true.

Python

my\_list = [10, 20, 30]  
 i = 0  
 while i < len(my\_list):  
 print(my\_list[i])  
 i += 1

**3. List Comprehensions:**

* **Concise Iteration:** List comprehensions offer a more compact way to create new lists based on existing ones.

Python

my\_list = [1, 2, 3, 4]  
 squared\_list = [x\*\*2 for x in my\_list]  
 print(squared\_list)

Key Points

* **Lists:** Mutable, ordered collections.
* **Tuples:** Immutable, ordered collections.
* **Strings:** Iterable sequences of characters.
* **For loops**: are ideal for iterating over a known number of elements.
* **While loops**: are better when the number of iterations is not known in advance.
* **List comprehensions**: offer a concise way to create new lists based on existing ones.

***Understanding how generators work in Python***

Python generators are a powerful and memory-efficient way to create iterators. They are defined like regular functions but use the yield keyword instead of return.

Here's how they work:

Generator Function Definition: A function becomes a generator function if it contains at least one yield statement.

Python

def my\_generator():

yield 1

yield 2

yield 3

Generator Object Creation: When a generator function is called, it does not execute its body immediately. Instead, it returns a special type of iterator called a "generator object."

Python

gen\_obj = my\_generator()

**Lazy Evaluation and yield:**

The generator object can be iterated over, for example, using a for loop or by repeatedly calling the next() function.

When next() is called (or implicitly by a for loop), the generator function executes from its last paused state until it encounters a yield statement.

The value specified after yield is returned to the caller.

Crucially, the state of the generator function (local variables, execution point) is saved.

**Resuming Execution:**

On subsequent calls to next(), the generator function resumes execution from where it left off, continuing until the next yield statement or until the function finishes.

**StopIteration:**

When the generator function finishes executing (i.e., there are no more yield statements to execute), it implicitly raises a StopIteration exception, signaling the end of the iteration. This exception is handled automatically by for loops.

Key Benefits:

**Memory Efficiency:**

Generators produce values one at a time, on demand, rather than storing the entire sequence in memory. This is particularly beneficial when dealing with large or infinite datasets.

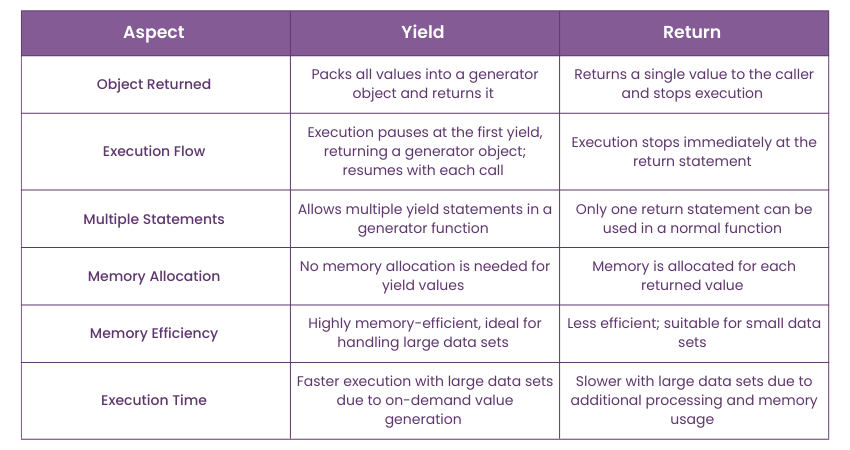
**Lazy Evaluation:**

Values are computed only when needed, which can improve performance in scenarios where not all values in a sequence are required.

**Clean Code:**

They provide a concise and elegant way to create iterators without needing to implement the full iterator protocol (\_\_iter\_\_() and \_\_next\_\_() methods).

***Difference between yield and return***



***Understanding iterators and creating custom iterator***

an iterator is an object that implements the iterator protocol, which consists of the \_\_iter\_\_() and \_\_next\_\_() methods. Iterators allow sequential access to elements within a collection without exposing the underlying data structure.

**Understanding Iterators:**

**Iterable:**

An object that can be iterated over (e.g., lists, tuples, strings, dictionaries). An iterable returns an iterator when passed to the iter() function.

**Iterator:**

An object that maintains its internal state during iteration, remembers its position, and can return the next element in a sequence using the \_\_next\_\_() method. When there are no more elements, \_\_next\_\_() raises a StopIteration exception.

**Creating Custom Iterators:**

To create a custom iterator, define a class that implements the \_\_iter\_\_() and \_\_next\_\_() methods:

**\_\_iter\_\_(self):**

This method should return the iterator object itself. It is called when you use the iter() function or when the iterator is used in a for loop.

**\_\_next\_\_(self):**

This method should return the next item in the sequence. If no more items exist, it must raise a StopIteration exception to signal the end of the iteration.

Example of a Custom Iterator:

Python

class MyCustomIterator:

def \_\_init\_\_(self, limit):

self.current = 0

self.limit = limit

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.current < self.limit:

value = self.current

self.current += 1

return value

else:

raise StopIteration

# Using the custom iterator

my\_iterator = MyCustomIterator(5)

for num in my\_iterator:

print(num)

This example demonstrates a custom iterator that generates numbers from 0 up to a specified limit. The \_\_init\_\_ method initializes the starting point and the limit. The \_\_iter\_\_ method returns self, indicating that the instance itself is the iterator. The \_\_next\_\_ method checks if the current number is within the limit, returns the value, increments the counter, and raises StopIteration when the limit is reached.

***Defining and calling functions in Python***

**1. Defining a Function:**

A function in Python is defined using the def keyword, followed by the function name, parentheses (), and a colon :. Any parameters the function accepts are listed inside the parentheses. The code block belonging to the function must be indented. An optional return statement can be used to send a value back to the caller.

Python

def my\_function(parameter1, parameter2):

# Code block of the function

result = parameter1 + parameter2

return result

**2. Calling a Function:**

To execute the code within a defined function, it must be called. This is done by writing the function's name followed by parentheses (). If the function requires arguments, they are passed inside the parentheses during the call.

Python

# Calling the function without arguments (if none are defined)

my\_function\_without\_args()

# Calling the function with arguments

value1 = 10

value2 = 20

returned\_value = my\_function(value1, value2)

print(returned\_value) # This will print 30

***Function arguments***

Function arguments in programming languages like Python can be categorized by how they are passed and defined:

**Positional Arguments:** These arguments are matched to parameters based on their order or position in the function call. The first argument in the call corresponds to the first parameter in the function definition, the second to the second, and so on.

Python

def greet(name, message):

print(f"Hello, {name}! {message}")

greet("Alice", "How are you?") # "Alice" is positional for 'name', "How are you?" for 'message'

**Keyword Arguments:** These arguments are passed by explicitly naming the parameter they correspond to in the function call, using the syntax parameter\_name=value. This allows for flexibility in the order of arguments and can improve code readability.

Python

def create\_user(name, email, age):

print(f"User: {name}, Email: {email}, Age: {age}")

create\_user(email="test@example.com", name="Bob", age=30) # Order doesn't matter with keywords

**Default Arguments:** These are parameters in a function definition that are assigned a default value. If a value for such a parameter is not provided during the function call, the default value is used. If a value is provided, it overrides the default. Default arguments must be defined after any non-default (positional) arguments in the function signature.

Python

def send\_email(to\_address, subject="No Subject", body=""):

print(f"Sending email to: {to\_address}")

print(f"Subject: {subject}")

print(f"Body: {body}")

send\_email("user@example.com") # Uses default subject and body

send\_email("another@example.com", subject="Important Update") # Overrides subject

***Scope of variables in Python***

The scope of a variable in Python refers to the region of the program where that variable is accessible and recognized. Python follows the LEGB rule for determining variable scope: Local, Enclosing, Global, and Built-in.

**Local Scope:** Variables defined inside a function have local scope. They are only accessible within that specific function and cease to exist once the function finishes execution.

Python

def my\_function():

local\_var = "I am local"

print(local\_var)

my\_function()

# print(local\_var) # This would raise a NameError

**Enclosing (Nonlocal) Scope:** This applies to variables in nested functions. An inner function can access variables from its immediate outer (enclosing) function, even though those variables are not global. The nonlocal keyword is used to modify a variable in the enclosing scope from within a nested function.

Python

def outer\_function():

enclosing\_var = "I am in the enclosing scope"

def inner\_function():

nonlocal enclosing\_var

enclosing\_var = "I am modified by the inner function"

print(enclosing\_var)

inner\_function()

print(enclosing\_var)

outer\_function()

**Global Scope:** Variables defined outside of any function or class have global scope. They are accessible from anywhere in the program, including within functions, unless a local variable with the same name is defined within that function. The global keyword is used to modify a global variable from within a function.

Python

global\_var = "I am global"

def another\_function():

print(global\_var) # Accessing global\_var

global global\_var

global\_var = "Global modified in function"

another\_function()

pri8////////////////////////nt(global\_var)

**Built-in Scope:** This is the broadest scope and includes all pre-defined functions and keywords that are always available in Python, such as print(), len(), str(), True, False, etc. These are accessible from any part of the program.

***Built-in methods for strings, lists, etc***

**Strings in Python:**

A string is a sequence of characters that can be a combination of letters, numbers, and special characters. It can be declared in python by using single quotes, double quotes, or even triple quotes. These quotes are not a part of a string, they define only starting and ending of the string. Strings are immutable, i.e., they cannot be changed. Each element of the string can be accessed using indexing or slicing operations.

# Assigning string to a variable

a = 'This is a string'

print (a)

b = "This is a string"

print (b)

c= '''This is a string'''

print (c)

Output:

This is a string

This is a string

This is a string

**Lists in Python:**

Lists are one of the most powerful data structures in python. Lists are sequenced data types. In Python, an empty list is created using list() function. They are just like the arrays declared in other languages. But the most powerful thing is that list need not be always homogeneous. A single list can contain strings, integers, as well as other objects. Lists can also be used for implementing stacks and queues. Lists are mutable, i.e., they can be altered once declared. The elements of list can be accessed using indexing and slicing operations.

# Declaring a list

L = [1, "a" , "string" , 1+2]

print L

#Adding an element in the list

L.append(6)

print L

#Deleting last element from a list

L.pop()

print (L)

#Displaying Second element of the list

print (L[1])

The output is:

[1, 'a', 'string', 3]

[1, 'a', 'string', 3, 6]

[1, 'a', 'string', 3]

a

**Tuples in Python:** A tuple is a sequence of immutable Python objects. Tuples are just like lists with the exception that tuples cannot be changed once declared. Tuples are usually faster than lists.

tup = (1, "a", "string", 1+2)

print(tup)

print(tup[1])

The output is :

(1, 'a', 'string', 3)

a

***Understanding the role of break, continue, and pass in Python loops***

break, continue, and pass are control flow statements used within loops to modify their execution behavior.

break statement:

The break statement immediately terminates the loop in which it is encountered.

Execution then proceeds to the statement immediately following the loop.

It is used when a specific condition is met, and further iterations of the loop are no longer necessary.

Python

for i in range(5):

if i == 3:

break # Exit the loop when i is 3

print(i)

# Output:

# 0

# 1

# 2

continue statement:

The continue statement skips the rest of the code within the current iteration of the loop.

It then moves control to the beginning of the loop for the next iteration.

It is used when a specific condition is met, and the current iteration should be bypassed, but the loop needs to continue with subsequent iterations.

Python

for i in range(5):

if i == 2:

continue # Skip printing when i is 2

print(i)

# Output:

# 0

# 1

# 3

# 4

pass statement:

The pass statement is a null operation; it does nothing.

It acts as a placeholder where a statement is syntactically required but no action is desired or implemented yet.

It is often used when creating code skeletons or when a block of code will be filled in later to avoid syntax errors.

Python

for i in range(3):

if i == 1:

pass # Do nothing when i is 1

else:

print(i)

# Output:

# 0

# 2

***Understanding how to access and manipulate strings***

One common operation in string manipulation is concatenation, which involves combining multiple strings together. For example, if we have two strings "Hello" and "World", concatenating them would result in the string "Hello World". Another important aspect of string manipulation is splitting.

***Basic operations: concatenation, repetition, string methods (upper(), lower()***

Basic operations on strings include concatenation, repetition, and the use of various string methods.

**Concatenation:**

Concatenation involves joining two or more strings together to form a single, longer string. In many programming languages, this is achieved using the + operator.

Python

string1 = "Hello"

string2 = " World"

concatenated\_string = string1 + string2

print(concatenated\_string) # Output: Hello World

**Repetition:**

Repetition involves creating a new string by repeating an existing string a specified number of times. This is often done using the \* operator.

Python

original\_string = "abc"

repeated\_string = original\_string \* 3

print(repeated\_string) # Output: abcabcabc

**String Methods:**

String methods are built-in functions that can be called on string objects to perform various manipulations. Common examples include:

upper(): Converts all characters in a string to uppercase.

Python

text = "example"

uppercase\_text = text.upper()

print(uppercase\_text) # Output: EXAMPLE

**lower()**: Converts all characters in a string to lowercase.

Python

text = "EXAMPLE"

lowercase\_text = text.lower()

print(lowercase\_text) # Output: example

**capitalize()** : Capitalizes the first character of the string and converts the rest to lowercase.

Python

text = "hello world"

capitalized\_text = text.capitalize()

print(capitalized\_text) # Output: Hello world

**title()** : Capitalizes the first letter of each word in the string.

Python

text = "this is a title"

title\_text = text.title()

print(title\_text) # Output: This Is A Title

**strip()** : Removes leading and trailing whitespace characters.

Python

text = " padded string "

stripped\_text = text.strip()

print(stripped\_text) # Output: padded string

***String slicing***

String slicing is a method used to extract a specific portion or substring from a larger string. It allows access to a sequence of characters within the original string by specifying a range of indices.

**Key Concepts of String Slicing:**

**Syntax:**

The general syntax for string slicing is string[start:stop:step].

**start:** The index where the slice begins (inclusive). If omitted, it defaults to the beginning of the string (index 0).

**stop:** The index where the slice ends (exclusive). The character at this index is not included in the resulting substring. If omitted, it defaults to the end of the string.

**step:** The interval between characters to be included in the slice (optional). If omitted, it defaults to 1, meaning consecutive characters are selected. A negative step value can be used to slice in reverse.

**Indexing:**

Strings are indexed starting from 0 for the first character. Negative indices can be used to count from the end of the string, where -1 refers to the last character, -2 to the second to last, and so on.

**Immutability:**

In many programming languages (like Python), strings are immutable. This means that string slicing does not modify the original string; instead, it returns a new string containing the extracted portion.

Examples (Python):

Python

my\_string = "Hello, World!"

# Basic slice from index 0 up to (but not including) index 5

slice1 = my\_string[0:5] # "Hello"

# Slice from index 7 to the end of the string

slice2 = my\_string[7:] # "World!"

# Slice from the beginning up to (but not including) index 5

slice3 = my\_string[:5] # "Hello"

# Slice with a step of 2

slice4 = my\_string[0:13:2] # "Hlo ol!"

# Reverse the string using slicing

reversed\_string = my\_string[::-1] # "!dlroW ,olleH"

***How functional programming works in Python***

Functional programming is a programming paradigm in which we try to bind everything in a pure mathematical functions style. It is a declarative type of programming style. Its main focus is on " what to solve" in contrast to an imperative style where the main focus is "how to solve". It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.

**Concepts of Functional Programming**

Any Functional programming language is expected to follow these concepts.

**Pure Functions:** These functions have two main properties. First, they always produce the same output for the same arguments irrespective of anything else. Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.

**Recursion:** There are no "for" or "while" loop in functional languages. Iteration in functional languages is implemented through recursion.

**Functions are First-Class and can be Higher-Order:** First-class functions are treated as first-class variables. The first-class variables can be passed to functions as a parameter, can be returned from functions, or stored in data structures.

**Variables are Immutable:** In functional programming, we can’t modify a variable after it’s been initialized. We can create new variables – but we can’t modify existing variables.

***Using map(), reduce(), and filter() functions for processing data***

The map(), reduce(), and filter() functions are higher-order functions commonly used in functional programming paradigms for efficient data processing. They offer a concise and declarative way to manipulate collections of data without explicit loops.

**map():** This function applies a given function to each item in an iterable (like a list, tuple, or array) and returns a new iterable containing the results. It is used for transformation, where you want to change each element in a collection based on a specific rule.

Python

numbers = [1, 2, 3, 4]

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

# squared\_numbers will be [1, 4, 9, 16]

**filter():** This function constructs a new iterable from elements of an existing iterable for which a given function returns True. It is used for selection, where you want to extract specific elements from a collection based on a condition.

Python

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

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

# even\_numbers will be [2, 4, 6]

**reduce():** This function applies a rolling computation to sequential pairs of values in an iterable, reducing the iterable to a single cumulative value. It is used for aggregation, where you want to combine elements of a collection into a single result. In Python, reduce() is part of the functools module.

Python

from functools import reduce

numbers = [1, 2, 3, 4]

product = reduce(lambda x, y: x \* y, numbers)

# product will be 24 (1 \* 2 \* 3 \* 4)

These functions can be combined to create powerful and flexible data processing pipelines, enabling operations like filtering data, transforming the filtered data, and then aggregating the transformed results.

***Introduction to closures and decorators***

Closures and decorators are powerful features in Python that allow for more advanced and flexible code patterns. Understanding these concepts can greatly enhance your ability to write clean, efficient, and reusable code.

**Why Python decorators rather than closures?**

Python decorators are preferred over closures for their readability, reusability, and flexibility. Decorators clearly convey the intent of modifying a function with the @decorator\_name syntax, keeping the code clean and focused. They promote modularity, allowing the same decorator to be easily reused across multiple functions. Decorators can also preserve the original function's metadata, which is important for debugging. Additionally, decorators can be parameterized and stacked to compose various behaviors efficiently, making them a more powerful and versatile tool compared to closures.

**Closures in Python**

A closure in Python occurs when a nested function captures the local variables from its enclosing scope. This allows the nested function to access these variables even after the outer function has finished executing.

**How Closures Work**

Closures are created when:

There is a nested function.

The nested function references a value in its enclosing scope.

The enclosing function returns the nested function.

Here's an example to illustrate closures:

# code

def outer\_function(msg):

message = msg

def inner\_function():

print(message)

return inner\_function

​

closure = outer\_function("Hello, World!")

closure() # Output: Hello, World!

In this example, inner\_function is a closure that captures the message variable from its enclosing scope, outer\_function

**Decorators in Python**

Python Decorators are a powerful and expressive tool in Python that allows you to modify the behavior of a function or method. They are often used to add "wrapping" functionality to existing functions in a clean and readable way.

**How Decorators Work**

A decorator is a function that takes another function as an argument, adds some kind of functionality, and returns a new function. Here's a basic example of a decorator:

# code

def simple\_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func()

print("Something is happening after the function is called.")

return wrapper

​

@simple\_decorator

def say\_hello():

print("Hello!")

​

say\_hello()

Output:

Something is happening before the function is called.

Hello!

Something is happening after the function is called.