1. Introduction to Python and its Features (simple, high-level, interpreted language).

* Python is a dynamic, high-level, free open source, and interpreted programming language. It supports object-oriented programming as well as procedural-oriented programming. In Python, we do not need to declare the type of variable because it is a dynamically typed language. For example, x = 10 Here, x can be anything such as String, int, etc. In this article we will see what characteristics describe the python programming language.
* Features in Python: 1. Free and Open Source: Python language is freely available at the official website and you can download it from the given download link below click on the Download Python keyword. Download Python Since it is open-source, this means that source code is also available to the public. So you can download it, use it as well as share it.
* 2. Easy to code: Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, Javascript, Java, etc. It is very easy to code in the Python language and anybody can learn Python basics in a few hours or days. It is also a developer-friendly language.
* 3. Easy to Read: As you will see, learning Python is quite simple. As was already established, Python’s syntax is really straightforward. The code block is defined by the indentations rather than by semicolons or brackets.
* 4. Object-Oriented Language: One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.
* 5. GUI Programming Support: Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in Python. PyQt5 is the most popular option for creating graphical apps with Python.
* 6. High-Level Language: Python is a high-level language. When we write programs in Python, we do not need to remember the system architecture, nor do we need to manage the memory.

7. Large Community Support: Python has gained popularity over the years. Our questions are constantly answered by the enormous Stack Overflow community. These websites have already provided answers to many questions about Python, so Python users can consult them as needed.

* 8. Easy to Debug: Excellent information for mistake tracing. You will be able to quickly identify and correct the majority of your program’s issues once you understand how to interpret Python’s error traces. Simply by glancing at the code, you can determine what it is designed to perform.
* 9. Python is a Portable language: Python language is also a portable language. For example, if we have Python code for Windows and if we want to run this code on other platforms such as Linux, Unix, and Mac then we do not need to change it, we can run this code on any platform.
* 10. Python is an integrated language: Python is also an integrated language because we can easily integrate Python with other languages like C, C++, etc.
* 11. Interpreted Language: Python is an Interpreted Language because Python code is executed line by line at a time. like other languages C, C++, Java, etc. there is no need to compile Python code this makes it easier to debug our code. The source code of Python is converted into an immediate form called bytecode.
* 12. Large Standard Library: Python has a large standard library that provides a rich set of modules and functions so you do not have to write your own code for every single thing. There are many libraries present in Python such as regular expressions, unit-testing, web browsers, etc.
* 13. Dynamically Typed Language: Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don’t need to specify the type of variable.
* 14. Frontend and backend development: With a new project py script, you can run and write Python codes in HTML with the help of some simple tags <py-script>, <py-env>, etc. This will help you do frontend development work in Python like javascript. Backend is the strong forte of Python it’s extensively used for this work cause of its frameworks like Django and Flask.
* 15. Allocating Memory Dynamically: In Python, the variable data type does not need to be specified. The memory is automatically allocated to a variable at runtime when it is given a value. Developers do not need to write int y = 18 if the integer value 15 is set to y. You may just type y=18.

1. History and evolution of Python.

* Python's evolution began in the late 1980s as a successor to the ABC programming language, with Guido van Rossum starting its development in 1989. The first release, Python 0.9.0, appeared in 1991. Key milestones include the release of Python 1.0 in 1994, 2.0 in 2000, and the major revision, Python 3.0, in 2008, which introduced significant changes not fully backward-compatible with earlier versions.
* The language has continued to evolve, with new features and libraries being added over time, solidifying its position as a versatile and popular programming language.

1. Advantages of using Python over other programming languages.

* Python offers several advantages over other programming languages, including its simplicity, versatility, extensive libraries, strong community support, and cross-platform compatibility. Its readability and ease of learning make it popular for both beginners and experienced developers. Python also excels in areas like data science, web development, and scripting, due to its rich ecosystem and frameworks.

1. Writing and executing your first Python program.

* Python Program to Print Hello World
* When we are just starting out with Python, one of the first programs we will learn is the classic “Hello, World!” program. It is a simple program that displays the message “Hello, World!” on the screen.
* Here is the “Hello World” program:
* print("Hello, World!")

1. Understanding Python’s PEP 8 guidelines.

* PEP 8 is Python's official style guide, providing a set of rules for writing readable and consistent Python code. It covers various aspects like naming conventions, code layout, whitespace usage, and more. Following PEP 8 makes code easier to read, maintain, and collaborate on.

1. Indentation, comments, and naming conventions in Python.

* Python relies on indentation to define code blocks, unlike languages that use braces. Consistent indentation is crucial; the standard is four spaces per level. Tabs can be used, but spaces are preferred for consistency across environments. Indentation errors will cause the program to fail.
* Comments in Python are used to explain code and are ignored by the interpreter. Single-line comments start with a # symbol. Multi-line comments, or docstrings, are enclosed in triple quotes (""" or '''). Docstrings are used for documentation purposes, while single-line comments explain specific parts of the code.
* Python naming conventions are outlined in PEP 8, the style guide for Python code. These conventions are used to improve code readability and consistency. Some common naming conventions include:
* Modules and packages: Use short, all-lowercase names, with underscores for modules but not for packages.
* Classes: Use CapWords (also known as CamelCase).
* Functions and variables: Use lowercase with underscores (snake\_case).
* Constants: Use all uppercase with underscores.

1. Writing readable and maintainable code.

* The Importance of Readable Code
* Consistent Naming Conventions. Choosing the right names for variables, functions, and classes is the first step towards readable code.
* Avoid Long Functions.
* Comment Wisely.
* Use Proper Indentation.
* Write Meaningful Tests.
* Avoid Deep Nesting.
* Modular Design.
* Meaningful Naming Conventions.

1. Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

* Python Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, Python data types are classes and variables are instances (objects) of these classes. The following are the standard or built-in data types in Python:
* Numeric – int, float, complex
* Sequence Type – string, list, tuple
* Mapping Type – dict
* Set Type – set, frozenset
* Datatypes: This code assigns variable ‘x’ different values of few Python data types – int, float, list, tuple, and string. Each assignment replaces the previous value, making ‘x’ take on the data type and value of the most recent assignment.
* # int, float, string, list, and set
* x = 50
* x = 60.5
* x = "Hello World"
* x = ["Tops", "for", "Technology"]
* x = ("Tops", "for", "Technology")
* 1. Numeric Data Types in Python
* The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python int, Python float and Python complex classes in Python.

1. Python variables and memory allocation.

* In Python, variables are dynamically typed and act as references to objects stored in memory. When a variable is assigned a value, memory is allocated to store the object, not the variable itself.
* Memory Allocation:
* Dynamic Typing: Python automatically determines the data type of a variable based on the assigned value. Memory is allocated accordingly.
* Heap Allocation: Objects are primarily stored in the heap, a region of memory for dynamic allocation.
* Reference Counting: Each object maintains a count of how many variables reference it. When this count drops to zero, the memory is eligible for garbage collection.
* Garbage Collection: Python's garbage collector reclaims memory occupied by objects that are no longer in use. It uses reference counting and generational garbage collection to do this.
* Memory Management: Python handles memory allocation and deallocation automatically, reducing the need for manual memory management.
* Object-specific allocators: Python uses object-specific allocators for memory allocation. If the object does not have one, and the size is greater than 512 bytes, it uses the raw memory allocator directly. Otherwise, it uses the object allocator.
* Variable Assignment:

When you assign a value to a variable, Python creates an object in memory and makes the variable refer to that object.

If you assign a different value to the same variable, it will now refer to a new object, and the old object is eligible for garbage collection if no other references exist.

The id() function can be used to see the memory address of an object.

Variables do not have a fixed data type, the type information is stored in the object itself.

1. Python operators: arithmetic, comparison, logical, bitwise.

* Python operators are special symbols that perform operations on values and variables.
* Arithmetic Operators: These operators perform mathematical calculations.
* +: Addition (e.g., x + y)
* -: Subtraction (e.g., x - y)
* \*: Multiplication (e.g., x \* y)
* /: Division (e.g., x / y)
* %: Modulus (remainder of division) (e.g., x % y)
* //: Floor division (result rounded down to nearest whole number) (e.g., x // y)
* \*\*: Exponentiation (e.g., x \*\* y)
* Comparison Operators: These operators compare two values and return a boolean result (True or False).
* ==: Equal to (e.g., x == y)
* !=: Not equal to (e.g., x!= y)
* >: Greater than (e.g., x > y)
* <: Less than (eplay, x < y)
* >=: Greater than or equal to (e.g., x >= y)
* <=: Less than or equal to (e.g., x <= y)
* Logical Operators: These operators combine or modify Boolean expressions.
* and: Returns True if both operands are True (e.g., x and y)
* or: Returns True if at least one operand is True (e.g., x or y)
* not: Returns True if the operand is False, and vice versa (e.g., not x)
* Bitwise Operators: These operators perform operations on the binary representation of integers.
* &: Bitwise AND (e.g., x & y)
* |: Bitwise OR (e.g., x | y)
* ^: Bitwise XOR (e.g., x ^ y)
* ~: Bitwise NOT (e.g., ~x)
* <<: Left shift (e.g., x << y)
* >>: Right shift (e.g., x >> y)
* These operators allow for various operations in Python, from basic arithmetic to more complex logical and bit-level manipulations.

1. Introduction to conditional statements: if, else, elif.

* Conditional statements are programming structures that allow a program to execute different blocks of code based on whether a condition is true or false. The most common conditional statements are if, else, and elif.
* if statement: This statement executes a block of code only if a specified condition is true.
* Python
* if condition:

# code to execute if condition is true

* else statement: This statement executes a block of code if the condition in the preceding if statement is false.

Python

if condition:

# code to execute if condition is true

else:

# code to execute if condition is false

* elif statement: Short for "else if," this statement allows you to check multiple conditions in sequence. If the if condition is false, the program checks the elif condition, and so on.

Python

if condition1:

# code to execute if condition1 is true

elif condition2:

# code to execute if condition1 is false and condition2 is true

* else:

# code to execute if all conditions are false

* These statements can be used in various ways to control the flow of a program, allowing it to respond differently to different situations

1. Nested if-else conditions.

* Nested if-else statements in Python involve placing one if-else block inside another. This allows for checking multiple conditions in a hierarchical manner.
* Structure:
* The basic structure involves an outer if statement, and within its block, another if statement (or an if-else block). This nesting can be continued to multiple levels, although excessive nesting can make code harder to read.
* if condition1:

if condition2:

# Code to execute if both condition1 and condition2 are true

else:

# Code to execute if condition1 is true but condition2 is false

else:

# Code to execute if condition1 is false

* Example:

x = 10

y = 5

if x > 5:

if y > 2:

print("x is greater than 5 and y is greater than 2")

else:

print("x is greater than 5 but y is not greater than 2")

else:

print("x is not greater than 5")

1. Introduction to for and while loops.

* For and while loops are fundamental control flow structures in programming that allow code to be executed repeatedly. For loops are used when the number of iterations is known beforehand, while while loops are used when the number of iterations depends on a condition.
* For Loops:
* Purpose: Iterate over a sequence (like a list, tuple, or string) or a range of numbers.

Syntax:

for variable in sequence:

# Code to be executed repeatedly

# Iterate over a list of fruits

fruits = ["apple", "banana", "cherry"]

for fruit in fruits:

print(fruit)

* While Loops:

Purpose: Execute a block of code repeatedly as long as a condition is true.

Syntax:

* while condition:

# Code to be executed repeatedly

# Print numbers from 1 to 5

i = 1

while i <= 5:

print(i)

        i += 1

1. How loops work in Python.

* In Python, loops are used to execute a block of code repeatedly. There are two primary 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 a block of code for each item in the sequence.
* They are ideal for iterating over a known set of items.
* While Loops:
* while loops execute a block of code as long as a condition is true.
* The loop continues to execute until the condition becomes false.
* They are suitable when the number of iterations is not known in advance.
* Key Concepts:

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

* Loops are fundamental for iterating over collections like lists, tuples, and strings in programming. Here's how they work:
* For Loops:
* Iterating Directly: The most common way is to use a for loop to iterate directly over the elements of the collection.

my\_list = [1, 2, 3, 4]

for item in my\_list:

print(item) # Output: 1, 2, 3, 4

* Iterating with Indices: You can also iterate using indices with range(len(collection)).

my\_tuple = ("a", "b", "c")

for i in range(len(my\_tuple)):

print(my\_tuple[i]) # Output: a, b, c

* While Loops:

While loops can also be used, but you will need to manage the index manually.

my\_string = "hello"

i = 0

while i < len(my\_string):

print(my\_string[i]) # Output: h, e, l, l, o

       i += 1

1. Understanding how generators work in Python.

* Generators are a special type of function that allows you to create iterators. They produce values "on the fly," meaning they generate one value at a time when needed, instead of storing all values in memory like lists.
* Generators Work: When a generator function is called, it doesn't execute immediately. Instead, it returns a generator object.
* When next() is called on the generator object, the function executes until it encounters a yield statement.
* The value after yield is returned, and the function's state is saved.
* Subsequent next() calls resume execution from the last saved state until a StopIteration exception is raised.

1. Difference between yield and return.

* In finance, yield refers to the income an investment generates, usually expressed as a percentage, while return represents the total gain or loss, including both income and changes in the investment's value. Yield is a forward-looking measure of income, while return is a backward-looking measure of overall investment performance.

1. Understanding iterators and creating custom iterators.

* An iterator in Python is an object that holds a sequence of values and provide sequential traversal through a collection of items such as lists, tuples and dictionaries. The Python iterators object is initialized using the iter() method. It uses the next() method for iteration.

\_iter(): \_\_iter\_() method initializes and returns the iterator object itself.

\_next(): the \_\_next\_() method retrieves the next available item, throwing a StopIteration exception when no more items are available

Custom iterators in Python are created by implementing the iterator protocol, which involves defining the \_iter() and \_\_next() methods within a class. The \_\_iter() method should return the iterator object itself, and the \_\_next\_() method should return the next item in the sequence, raising a StopIteration exception when there are no more items.

* Creating a Custom Iterator: Define the Iterator Class: Create a new class that will be used as the custom iterator.

Implement \_iter\_(): This method should return the iterator object itself (usually self).

Implement \_next\_(): This method should return the next item in the sequence each time it's called.

* It also needs to raise a StopIteration exception when the sequence is exhausted, indicating that there are no more items.

(Optional) Add Additional Functionality: You can add more methods or attributes to your class to customize the iterator's behavior, such as resetting the iterator or accessing the current position.

1. Defining and calling functions in Python.

* Defining Functions: Functions in Python are defined using the def keyword, followed by the function name, parentheses (), and a colon :Arguments or parameters can be placed inside the parentheses. The code block that the function executes is indented below the function definition.
* def function\_name(parameter1, parameter2):

# Function body

return value

* Calling Functions: To call a function, you use the function's name followed by parentheses (). If the function takes arguments, you pass them inside the parentheses.

Python

* result = function\_name(argument1, argument2)

1. Function arguments (positional, keyword, default).

* Function arguments in Python can be categorized into three main types: positional, keyword, and default arguments.
* Positional Arguments: Positional arguments are defined by their order or position in the function call. The first argument in the function call corresponds to the first parameter in the function definition, the second to the second, and so on.
* Python
* def my\_function(a, b, c):
* print(f"a: {a}, b: {b}, c: {c}")
* my\_function(1, 2, 3) # Output: a: 1, b: 2, c: 3
* Keyword Argument: Keyword arguments are identified by their parameter names when calling a function. This allows you to pass arguments out of order, making the code more readable and less prone to errors.
* Python
* def my\_function(a, b, c):
* print(f"a: {a}, b: {b}, c: {c}")
* my\_function(c=3, b=2, a=1) # Output: a: 1, b: 2, c: 3
* Default Arguments: Default arguments are assigned default values in the function definition. If the function is called without providing a value for that argument, the default value is used.
* def my\_function(a, b=2, c=3):
* print(f"a: {a}, b: {b}, c: {c}")
* my\_function(1) # Output: a: 1, b: 2, c: 3

my\_function(1, 4) # Output: a: 1, b: 4, c: 3

my\_function(1, c=5) # Output: a: 1, b: 2, c: 5

1. Scope of variables in Python.

* In Python, the scope of a variable determines where it can be accessed within the code. There are several types of variable scopes in Python:
* 1. Local Scope: Variables defined inside a function have local scope.
* They can only be accessed within that function. When a function call ends, the local variables are destroyed. If a variable with the same name exists outside the function, it will be treated as a separate variable.
* 2. Global Scope:
* Variables defined outside any function or block have global scope.

Global variables can be accessed from anywhere within the program, including inside functions.

If a variable with the same name is defined locally inside a function, the local variable takes precedence within that function.

* 3. Enclosing Scope:
* When a function is defined inside another function, the inner function has access to variables in the outer function's scope.

This is known as the enclosing scope.

Nonlocal keyword is used to modify variables in the enclosing scope.

* 4. Built-in Scope:
* Python has a set of predefined built-in functions and constants.

These have built-in scope and are always available in the program.

1. Built-in methods for strings, lists, etc.

* Python string methods is a collection of in-built Python functions that operates on strings. Every string method in Python does not change the original string instead returns a new string with the changed attributes. Python string is a sequence of Unicode characters that is enclosed in quotation marks. In this article, we will discuss the in-built string functions i.e. the functions provided by Python to operate on strings.
* Case Changing of Python String Methods
* lower(): Converts all uppercase characters in a string into lowercase
* upper(): Converts all lowercase characters in a string into uppercase
* title(): Convert string to title case
* swapcase(): Swap the cases of all characters in a string
* capitalize(): Convert the first character of a string to uppercase
* List of String Methods in Python

Here is the list of in-built Python string methods, that you can use to perform actions on string:

capitalize() Converts the first character of the string to a capital (uppercase) letter

* casefold() Implements caseless string matching
* center() Pad the string with the specified character.
* count() Returns the number of occurrences of a substring in the string.
* encode() Encodes strings with the specified encoded scheme
* endswith() Returns “True” if a string ends with the given suffix
* index() Returns the position of the first occurrence of a substring in a string
* isalnum() Checks whether all the characters in given string is alphanumeric or not a
* isalpha() Returns “True” if all characters in the string are alphabets
* isdecimal() Returns true if all characters in a string are decimal
* isdigit() Returns “True” if all characters in the string are digits
* isidentifier() Check whether a string is a valid identifier or not
* islower() Checks if all characters in the string are lowercase
* isnumeric() Returns “True” if all characters in the string are numeric characters
* isspace() Returns “True” if all characters in the string are whitespace characters
* istitle() Returns “True” if the string is a title cased string
* isupper() Checks if all characters in the string are uppercase
* join() Returns a concatenated String
* lower() Converts all uppercase characters in a string into lowercase
* replace() Replaces all occurrences of a substring with another substring
* rfind() Returns the highest index of the substring

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

* Python supports the following control statements:
* Break statement
* Continue statement
* Pass statement
* Break Statement in Python: The break statement in Python is used to exit or “break” out of a loop (either a for or while loop) prematurely, before the loop has iterated through all its items or reached its condition. When the break statement is executed, the program immediately exits the loop, and the control moves to the next line of code after the loop.
* Continue Statement in Python
* Python Continue statement is a loop control statement that forces to execute the next iteration of the loop while skipping the rest of the code inside the loop for the current iteration only, i.e. when the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped for the current iteration and the next iteration of the loop will begin.
* Pass Statement in Python: Pass statement in Python is a null operation or a placeholder. It is used when a statement is syntactically required but we don’t want to execute any code. It does nothing but allows us to maintain the structure of our program.

1. 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.

1. Basic operations: concatenation, repetition, string methods (upper(), lower(), etc.).

* Concatenation: Concatenation is the process of joining two or more strings together to create a new string. The most common way to do this is using the + operator.

string1 = "Hello"

string2 = "World"

result = string1 + " " + string2

print(result) # Output: Hello World

* Repetition: Repetition is the process of creating a new string by repeating an existing string a specified number of times. The \* operator is used for this purpose.

string = "abc"

result = string \* 3

print(result) # Output: abcabcabc

* String Methods: String methods are built-in functions that can be used to manipulate strings. Some of the most commonly used methods include:
* upper(): Converts all characters in a string to uppercase.

string = "hello"

result = string.upper()

print(result) # Output: HELLO

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

string = "WORLD"

result = string.lower ()

print(result) # Output: world

1. String slicing.

* String slicing in Python is a way to get specific parts of a string by using start, end and step values. It’s especially useful for text manipulation and data parsing.
* Let’s take a quick example of string slicing:
* s = "Hello, Python!"
* Print (s [0:5])
* Output
* Hello

1. 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.
* Python programming language also supports Functional Programming paradigms without the support of any special features or libraries
* Pure Functions
* As Discussed above, pure functions have two properties.
* It always produces the same output for the same arguments. For example, 3+7 will always be 10 no matter what.
* It does not change or modifies the input variable.
* The second property is also known as immutability. The only result of the Pure Function is the value it returns. They are deterministic. Programs done using functional programming are easy to debug because pure functions have no side effects or hidden I/O. Pure functions also make it easier to write parallel/concurrent applications. When the code is written in this style, a smart compiler can do many things – it can parallelize the instructions, wait to evaluate results when needing them, and memorize the results since the results never change as long as the input doesn’t change.

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

* Map (), filter (), and reduce () are higher-order functions commonly used for processing data in various programming languages. They operate on collections (like arrays or lists) and provide a concise way to transform, select, and aggregate data.
* Map (): The map () function applies a given function to each element of a collection, creating a new collection with the transformed elements.
* Purpose: Transform each element of an array.
* Example:
* numbers = [1, 2, 3, 4]
* squared\_numbers = list (map (lambda x: x\*\*2, numbers)) # Output: [1, 4, 9, 16]
* Filter (): The filter () function creates a new collection containing only the elements from the original collection that satisfy a given condition (predicate).
* Purpose: Select elements that match a condition.
* Reduce (): The reduce () function applies a function cumulatively to the elements of a collection, reducing it to a single value. It takes an initial value and a function that combines the accumulator (result so far) with the current element.
* Purpose: Combine all elements into a single value.

1. Introduction to closures and decorators.

* Closures and decorators are powerful Python tools for writing reusable and flexible code. Closures allow inner functions to remember and access variables from their enclosing scope, even after the outer function has finished executing. Decorators, built upon closures, enhance or modify the behavior of functions by wrapping them with additional functionality.
* Closures: A closure is a nested function that remembers the values of its enclosing scope's variables, even after the outer function is no longer running.
* How it works: When a function is defined within another function, it can access and modify variables from the outer function's scope. This allows the inner function to retain information about the outer function's state.
* Decorators: A decorator is a function that takes another function as input, modifies its behavior, and returns a new function.
* Decorators typically use closures to maintain state and add functionality to the decorated function. They are defined using the @ syntax, which is a shorthand for calling a decorator function on another function.