History of Python

Python is a widely used general-purpose, high-level programming language. It was initially designed by **Guido van Rossum** in **1991** and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Who invented Python?

In the late 1980s, history was about to be written. It was that time when working on Python started. Soon after that, Guido Van Rossum began doing its application-based work in December of 1989 at Centrum Wiskunde & Informatica (CWI) which is situated in the Netherlands. It was started as a hobby project because he was looking for an interesting project to keep him occupied during Christmas.
The programming language in which Python is said to have succeeded is ABC Programming Language, which had interfacing with the Amoeba Operating System and had the feature of exception handling. He had already helped create ABC earlier in his career and had seen some issues with ABC but liked most of the features. After that what he did was very clever. He had taken the syntax of ABC, and some of its good features. It came with a lot of complaints too, so he fixed those issues completely and created a good scripting language that had removed all the flaws. The inspiration for the name came from the BBC's TV Show – '**Monty Python's Flying Circus'**, as he was a big fan of the TV show and also he wanted a short, unique and slightly mysterious name for his invention and hence he named it Python! He was the "Benevolent dictator for life" (BDFL) until he stepped down from the position as the leader on 12th July 2018. For quite some time he used to work for Google, but currently, he is working at Dropbox.

Evolution of Python

The language was finally released in 1991. When it was released, it used a lot fewer codes to express the concepts, when we compare it with Java, C++ & C. Its design philosophy was quite good too. Its main objective is to provide code readability and advanced developer productivity. When it was released, it had more than enough capability to provide classes with inheritance, several core data types of exception handling and functions.
Following are the illustrations of different versions of Python along with the timeline.

*** **Python 3.12.3 is the latest stable version.**

Python :: Variables

1.What is a variable in Python?

In Python, a variable is a named location in the computer's memory that stores a value. It acts as a symbolic name for a value and allows programmers to manipulate data more conveniently.

Variables in Python are created by assigning a value to them using the assignment operator (`=`). The type of the variable is dynamically determined based on the value assigned to it.
Here is an example program that demonstrates the use of variables in Python:

```
# Define two variables
first_number = 5
second_number = 7

# Perform addition using variables
sum_result = first_number + second_number

# Print the result
print("Result = ", sum_result)
```

In this example, the variables `first_number` and `second_number` store the values 5 and 7, respectively. The sum of these two variables is calculated and stored in the variable `sum_result`. Finally, the program prints the result.

2.How do you declare a variable in Python?

In Python, you declare a variable by assigning a value to it using the assignment operator `=`. Python is a dynamically typed language, so you don't need to explicitly declare the type of the variable; it is determined based on the assigned value.
Here's a simple example:

```
# Declare and assign values to variables
my_variable = 10
my_string = "Hello, Python!"

# Print the variables
print("my_variable:", my_variable)
print("my_string:", my_string)
```

In this example, `my_variable` is assigned the integer value `10`, and `my_string` is assigned the string value `"Hello, Python!"`. The `print` statements then output the values of these variables.
Remember that variable names in Python are case-sensitive and can contain letters, numbers, and underscores, but they cannot start with a number. Descriptive and meaningful variable names are encouraged for better code readability.

3.Explain the rules for naming variables in Python.

In Python, variable names must follow certain rules to ensure proper functionality and code readability:

1. Valid Characters: Variable names can only contain letters (both uppercase and lowercase), numbers, and underscores. Special characters, spaces, or punctuation marks are not allowed.

2. Start with a Letter or Underscore: A variable name must begin with a letter (a-z, A-Z) or an underscore (_). It cannot start with a number.

3. Case-Sensitive: Python is case-sensitive, so `my_variable` and `My_Variable` would be considered different variables.
4. Reserved Words: Avoid using Python reserved words as variable names. Examples include `if`, `else`, `while`, `for`, `import`, and others.
5. Descriptive and Meaningful: Choose variable names that are descriptive and convey the purpose of the variable for better code readability.

6. Avoid Single Underscore as the First Character: While technically allowed, using a single underscore as the first character in a variable name is often reserved for special cases (e.g., `_variable`). It's not recommended for regular variable names.

Here's an example demonstrating the correct and incorrect ways to name variables:

```
# Correct variable names
my_variable = 10
user_age = 25
total_sum = my_variable + user_age

# Incorrect variable names
second_variable = 5  # Starts with a number, not allowed
user age = 25        # Contains a space, not allowed
total-value = 30     # Contains a hyphen, not allowed
pass = 5             # Reserved word, not allowed
```

4.What is the purpose of the id() function in Python?

The `id()` function in Python is used to get the identity (unique identifier) of an object. This identity is a unique integer that represents the object. The `id()` function returns the memory address of the object in CPython, the default implementation of Python. Here's an example to illustrate the purpose of the `id()` function:

```
# Example usage of the id() function
my_list = [1, 2, 3]

# Get the identity of the list
list_id = id(my_list)

# Print the result
print("Identity of my_list:", list_id)
```

5.Differentiate between local and global variables in Python.

Local Variables:

```
# Local variables are defined within a specific function or block
def my_function():
    local_variable = 10
    print("Inside function:", local_variable)

# Calling the function to demonstrate the local variable
my_function()

# Attempting to access local_variable here would result in an error
```

In the above example, the variable `local_variable` is defined within the function `my_function()`. It is accessible only within the scope of that function. Once the function execution is complete, the local variable is destroyed, and attempting to access it outside the function would result in an error.

Global Variables:

```
# Global variables are defined outside of any function or block,
# making them accessible throughout the program
global_variable = 20

# Function accessing the global variable
def my_function():
    print("Inside function:", global_variable)
```

```
# Calling the function to demonstrate the global variable
my_function()

# Global variables are also accessible outside functions
print("Outside function:", global_variable)
```

In the second example, the variable `global_variable` is defined outside any function or block, making it a global variable. It is accessible throughout the entire program, both within and outside functions. The value of the global variable persists for the entire duration of the program.
These examples illustrate the differences in scope and lifetime between local and global variables in Python.

6.Explain the concept of variable scope in Python.

Variable scope in Python refers to the region of the code where a variable is accessible or can be modified. In Python, there are two main types of variable scope: `local` and `global`.
`Local scope` refers to the portion of code where a variable is declared within a function or block. Local variables are only accessible within that specific function or block and are not visible outside of it.
`Global scope` refers to the portion of code where a variable is declared outside of any function or block. Global variables are accessible from any part of the code, both inside and outside functions.

```
# Global variable
global_variable = 10

def example_function():
    # Local variable
    local_variable = 5

    # Accessing global variable within the function
    global_variable_inside_function = global_variable + 2

    # Printing local and modified global variables
    print("Local variable inside function:", local_variable)
    print("Modified global variable inside function:",
global_variable_inside_function)

# Call the function
example_function()
```

```
# Attempting to access local variable outside the function (will
result in an error)
# print("Attempt to access local variable outside function:",
local_variable)
```

In Python, nested scope occurs when a variable is declared within an inner block or function, making it accessible within that block as well as any nested blocks or functions:

```
# Global variable
global_variable = 10

def outer_function():
    # Outer function's local variable
    outer_variable = 5

    def inner_function():
        # Inner function's local variable
        inner_variable = 3

        # Accessing variables from outer scopes
        total_sum = outer_variable + inner_variable + global_variable

        # Printing variables
        print("Total sum inside inner function:", total_sum)

    # Call the inner function
    inner_function()

# Call the outer function
outer_function()
```

7.What are the built-in types for variables in Python?

In Python, there are several built-in data types that represent the different kinds of values variables can take. These built-in types include:

- `int`: Integer type, representing whole numbers.
- `float`: Floating-point type, representing decimal numbers.
- `str`: String type, representing text.
- `bool`: Boolean type, representing True or False values.
- `list`: List type, representing ordered, mutable sequences of elements.
- `tuple`: Tuple type, representing ordered, immutable sequences of elements.

- `set`: Set type, representing an unordered collection of unique elements.
- `dict`: Dictionary type, representing a collection of key-value pairs.

Here's an example program demonstrating the use of these built-in types:

```
# Integer type
my_integer = 42

# Floating-point type
my_float = 3.14

# String type
my_string = "Hello, Python!"

# Boolean type
is_true = True

# List type
my_list = [1, 2, 3, 4, 5]

# Tuple type
my_tuple = (10, 20, 30)

# Set type
my_set = {1, 2, 3, 4, 5}

# Dictionary type
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

# Printing variables
print("Integer:", my_integer)
print("Float:", my_float)
print("String:", my_string)
print("Boolean:", is_true)
print("List:", my_list)
print("Tuple:", my_tuple)
print("Set:", my_set)
print("Dictionary:", my_dict)
```

The output of the provided example program would be:

```
Integer: 42
Float: 3.14
String: Hello, Python!
Boolean: True
```

```
List: [1, 2, 3, 4, 5]
Tuple: (10, 20, 30)
Set: {1, 2, 3, 4, 5}
Dictionary: {'name': 'John', 'age': 25, 'city': 'New York'}
```

8.How do you swap the values of two variables in Python without using a temporary variable?

Here's a Python code snippet that swaps the values of two variables without using a temporary variable:

```python
# Swapping variables without using a temporary variable
def swap_without_temp(a, b):
    a = a + b
    b = a - b
    a = a - b
    return a, b

# Example usage
x = 5
y = 10

x, y = swap_without_temp(x, y)

print("After swapping: x =", x, ", y =", y)
```

In this code, the values of `a` and `b` are swapped without using a temporary variable. The swap is achieved using arithmetic operations.
Here's an alternative Python code snippet that swaps the values of two variables without using a temporary variable:

```python
# Swapping variables without using a temporary variable
def swap_without_temp(a, b):
    a, b = b, a
    return a, b

# Example usage
x = 5
y = 10

x, y = swap_without_temp(x, y)
```

```
print("After swapping: x =", x, ", y =", y)
```

In this code, the values of `a` and `b` are swapped using a tuple unpacking assignment. This provides a concise and readable way to swap values in Python.

9.What is the difference between `==` and `is` when comparing variables?

When comparing variables in Python, `==` and `is` serve different purposes:
`==` checks for equality of values, while `is` checks for identity, i.e., if the variables refer to the same object in memory.

```
# Using == for value equality
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print("Using ==:")
print(a == b)  # True, as the values are the same
print(a == c)  # True, as the values are the same

# Using is for identity
print("\nUsing is:")
print(a is b)  # False, as they are different objects in memory
print(a is c)  # True, as both refer to the same object in memory
```

In the example, `==` compares the values of the lists, and it returns `True` if the values are the same. On the other hand, `is` checks if two variables refer to the same object in memory.

10.What is the purpose of the `del` statement in Python with respect to variables?

In Python, the `del` statement is used to delete variables or items from a collection (such as lists or dictionaries). It serves several purposes related to memory management and variable manipulation.
Here's an example program illustrating the use of `del`:

```
# Using del to delete variables
x = 10
y = [1, 2, 3]

print("Before deletion:")
print("x =", x)
print("y =", y)
```

```
# Deleting variables
del x
del y[0]

print("\nAfter deletion:")
# Uncommenting the next line would result in an error since 'x' is
deleted
# print("x =", x)
print("y =", y)
```

In this example, the `del` statement is used to delete the variable `x` and the first element of the list `y`. After deletion, attempting to access `x` would result in an error, while the list `y` reflects the removal of its first element.

11.Explain the concept of dynamic typing in Python.

In Python, dynamic typing is a concept where the type of a variable is determined at runtime, rather than at compile-time. This means that you can assign values of different types to the same variable during the execution of the program.

Here's an example program illustrating dynamic typing in Python:

```
# Dynamic typing example
variable = 10  # variable is an integer
print("Variable is an integer:", variable)

variable = "Hello"  # variable is now a string
print("Variable is a string:", variable)

variable = [1, 2, 3]  # variable is now a list
print("Variable is a list:", variable)
```

In this example, the variable `variable` is assigned different types of values - first an integer, then a string, and finally a list. The type of the variable is determined at runtime based on the assigned value.

12.How can you concatenate two strings in Python using variables?

In Python, you can concatenate two strings using the `+` operator or by using the `+=` shorthand.
Here's an example program demonstrating string concatenation in Python:

```
# String concatenation example
first_name = "John"
last_name = "Doe"

# Using the + operator
full_name = first_name + " " + last_name
print("Full name using + operator:", full_name)

# Using the += shorthand
greeting = "Hello, "
name = "Alice"
greeting += name
print("Greeting using += shorthand:", greeting)
```

In this example, two strings, `first_name` and `last_name`, are concatenated using the `+` operator to form the `full_name`. Additionally, a greeting string is concatenated with a name using the `+=` shorthand.

13.What is the difference between mutable and immutable objects in Python? Give examples.

In Python, objects can be categorized as either mutable or immutable based on whether their values can be changed after creation. Mutable objects allow modifications, while immutable objects do not.

Mutable Objects: Objects whose state or value can be changed after creation are considered mutable. Examples include lists and dictionaries.

```
# Mutable object example: List
original_list = [1, 2, 3]
print("Original list:", original_list)

# Modifying the list
original_list[0] = 99
print("Modified list:", original_list)
```

Immutable Objects: Objects whose state cannot be changed after creation are considered immutable. Examples include strings and tuples.

```
# Immutable object example: String
original_string = "Hello"
print("Original string:", original_string)

# Attempting to modify the string would result in an error
# Uncommenting the next line would raise an error
```

```
# original_string[0] = 'C'
```

In the example, the list (mutable object) can be modified by changing its elements, while attempting to modify the string (immutable object) would result in an error. Understanding the mutability of objects is crucial for effective programming and preventing unintended side effects.

14.Explain the use of the `global` keyword in Python.

In Python, the `global` keyword is used to indicate that a variable declared within a function should be treated as a global variable. It allows you to modify the value of a variable defined outside the current function scope.
Here's an example program demonstrating the use of the `global` keyword:

```
# Using the global keyword
global_variable = 10

def modify_global_variable():
    global global_variable  # Indicating that 'global_variable' is a
global variable
    global_variable = 20

modify_global_variable()

print("Modified global variable:", global_variable)
```

In this example, the function `modify_global_variable` uses the `global` keyword to indicate that it will modify the global variable `global_variable`. After calling the function, the value of the global variable is changed, and the updated value is printed.

15.What is the purpose of the `locals()` and `globals()` functions in Python?

In Python, the `locals()` and `globals()` functions are used to access local and global symbol tables, respectively. These functions return dictionaries representing the current local and global symbol tables, allowing you to view or modify variable names and their values.
Here's an example program illustrating the use of `locals()` and `globals()`:

```
# Using locals() and globals() functions
global_variable = 10

def example_function():
    local_variable = 20
```

```
    print("Local variables using locals():", locals())

example_function()
print("\nGlobal variables using globals():", globals())
```

In this example, the `example_function` prints the local variables using the `locals()` function. The global variables are printed outside the function using the `globals()` function. These functions are useful for debugging, introspection, and dynamic code generation scenarios.

16.How can you check the type of a variable in Python?

In Python, you can check the type of a variable using the `type()` function.
The `type()` function returns the type of the specified object.
Here's an example program demonstrating how to check the type of a variable:

```
# Checking the type of a variable using type()
variable_1 = 42
variable_2 = "Hello, Python!"
variable_3 = [1, 2, 3]

print("Type of variable_1:", type(variable_1))
print("Type of variable_2:", type(variable_2))
print("Type of variable_3:", type(variable_3))
```

In this example, the `type()` function is used to check the types of three different variables: an integer, a string, and a list. The output will display the respective types of these variables.

17.What is variable unpacking in Python?

In Python, variable unpacking is a feature that allows you to assign multiple values to multiple variables in a single line. This can be done using iterable unpacking, which involves extracting elements from an iterable (e.g., a tuple, list, or string) and assigning them to individual variables.

Here's an example program illustrating variable unpacking:

```
# Variable unpacking in Python
coordinates = (3, 7)

# Unpacking the tuple into two variables
```

```
x, y = coordinates

print("Original tuple:", coordinates)
print("Unpacked variables - x:", x, ", y:", y)
```

In this example, the `coordinates` tuple is unpacked into two variables `x` and `y`. The values from the tuple are assigned to these variables in the order they appear in the tuple.

18.Discuss the concept of reference counting in Python variables.

In Python, reference counting is a memory management technique used to keep track of the number of references to an object. Every object in Python has a reference count, and when the count drops to zero, the memory occupied by the object is deallocated. The reference count is incremented when a new reference to the object is created and decremented when a reference goes out of scope or is explicitly deleted.

Here's an example program illustrating reference counting in Python:

```
# Reference counting example
variable_a = [1, 2, 3]  # Reference count of the list is 1

variable_b = variable_a  # Reference count is now 2

del variable_a  # Reference count is decremented to 1

variable_b = "Hello"  # Reference count of the list is now 0, and
memory is deallocated
```

In this example, the list `variable_a` is created with a reference count of 1.
When `variable_b` is assigned the value of `variable_a`, the reference count becomes 2.
Deleting `variable_a` reduces the reference count to 1, and when `variable_b` is reassigned, the reference count becomes 0, leading to the deallocation of memory.

19.How do you convert a variable from one data type to another in Python?

In Python, you can convert a variable from one data type to another using type conversion functions or methods. These functions include `int()`, `float()`, `str()`, etc.
Here's an example program demonstrating variable type conversion:

```
# Variable type conversion example
integer_variable = 42
```

```
float_variable = 3.14
string_variable = "123"

# Converting integer to float
float_from_integer = float(integer_variable)
print("Float from integer:", float_from_integer)

# Converting float to integer
integer_from_float = int(float_variable)
print("Integer from float:", integer_from_float)

# Converting string to integer
integer_from_string = int(string_variable)
print("Integer from string:", integer_from_string)
```

In this example, the variables `integer_variable`, `float_variable`, and `string_variable` are converted to different data types using the `float()` and `int()` functions. Type conversion allows you to change the data type of a variable as needed.

20.Explain the difference between shallow copy and deep copy in Python with regard to variables.

In Python, shallow copy and deep copy are two methods used to duplicate objects, such as lists or dictionaries. The difference lies in how they handle nested objects. Shallow copy creates a new object but does not create new objects for the elements within the container, while deep copy creates a completely independent copy with new objects for all nested elements.

Shallow Copy: Shallow copy creates a new object but does not create new objects for the elements within the container.

```
import copy

original_list = [1, [2, 3], 4]

shallow_copied_list = copy.copy(original_list)

# Modify the original list to demonstrate shallow copy behavior
original_list[1][0] = 'X'

print("Original list:", original_list)  # [1, ['X', 3], 4]
print("Shallow copied list:", shallow_copied_list)  # [1, ['X', 3], 4]
```

Deep Copy: Deep copy creates a completely independent copy with new objects for all nested elements.

```python
import copy

original_list = [1, [2, 3], 4]

deep_copied_list = copy.deepcopy(original_list)

# Modify the original list to demonstrate deep copy behavior
original_list[1][0] = 'Y'

print("Original list:", original_list)  # [1, ['Y', 3], 4]
print("Deep copied list:", deep_copied_list)  # [1, [2, 3], 4]
```

In the shallow copy example, modifying a nested element in the original list reflects the change in the shallow copied list. In the deep copy example, modifying the original list does not affect the deep copied list, showcasing the independence achieved through deep copy.