1. What are the new features added in Python 3.8 version?

ANS: Python 3.8, released on October 14, 2019, introduced several new features and improvements. Some of the notable features added in Python 3.8 include:

A. \*\*Assignment Expressions (The Walrus Operator)\*\*:

- Introduced the `:=` assignment operator, also known as the "walrus operator."

- It allows you to assign values to variables as part of an expression.

- Example: `if (n := len(my\_list)) > 10:`

B. \*\*Positional-Only Parameters\*\*:

- Function parameters can now be designated as positional-only by using the `/` separator in the function definition.

- This restricts how arguments are passed to the function and can improve clarity in function calls.

- Example: `def my\_function(a, b, /, c, d):`

C. \*\*Syntax for Typed Dictionaries\*\*:

- Introduced a new syntax for specifying dictionary types with specific key-value pairs.

- Example: `my\_dict: dict[str, int] = {'a': 1, 'b': 2}`

D. \*\*f-strings Improvements\*\*:

- Added support for the `=` specifier in f-strings to display both the expression and the result.

- Example: `x = 10; print(f'{x=} {x\*2=}')`

E. \*\*Python Initialization Configuration (PEP 587)\*\*:

- The Python runtime can now be configured using a new `pyconfig.h` file, allowing platform-specific build options and module configurations.

F. \*\*Parallel Execution of `for` Loops\*\*:

- The `concurrent.futures` module introduced the `ThreadPoolExecutor` and `ProcessPoolExecutor` classes to perform parallel execution of `for` loops.

1. What is monkey patching in Python?

ANS: In Python, "monkey patching" refers to the practice of dynamically modifying or extending the behavior of a module, class, or function at runtime. It involves making changes to the code at runtime without altering the original source code. Monkey patching can be a powerful technique, but it should be used with caution, as it can lead to unexpected behavior and maintainability issues if not done carefully.

Monkey patching is typically used when you want to add new functionality to existing code, fix bugs temporarily, or work around limitations in third-party libraries without modifying the original codebase. It is often employed in testing, debugging, or prototyping scenarios.

Here are some examples of monkey patching:

A. \*\*Extending a Class Method\*\*:

You can add new methods or attributes to an existing class at runtime.

class MyClass:

def say\_hello(self):

return "Hello"

def new\_method(self):

return "New method!"

# Monkey patching: Adding a new method to MyClass

MyClass.new\_method = new\_method

obj = MyClass()

print(obj.say\_hello()) # Output: "Hello"

print(obj.new\_method()) # Output: "New method!"

B. \*\*Replacing a Function Implementation\*\*:

You can replace the implementation of an existing function with a new one.

def original\_function():

return "Original implementation"

def new\_function():

return "New implementation!"

# Monkey patching: Replacing original\_function with new\_function

original\_function = new\_function

print(original\_function()) # Output: "New implementation!"

C. \*\*Fixing a Bug in a Third-Party Library\*\*:

You can patch a bug in a third-party library by providing a temporary fix at runtime.

import third\_party\_library

def patched\_function():

# Temporary fix for the bug

# ...

# Monkey patching: Replacing the buggy function in the third-party library

third\_party\_library.problematic\_function = patched\_function

1. What is the difference between a shallow copy and deep copy?

ANS: In Python, a shallow copy and a deep copy are two different methods of creating copies of objects, especially when dealing with nested data structures like lists, dictionaries, or objects containing other objects. The key difference between them lies in how they handle nested objects or references within the original object being copied.

A. \*\*Shallow Copy\*\*:

- A shallow copy creates a new object, but it does not create new objects for the elements contained within the original object. Instead, it copies references to the nested objects.

- In other words, it creates a new top-level object but keeps references to the same objects as the original for nested elements. This means that changes made to nested elements in the copy will also affect the original and vice versa.

- Shallow copy can be performed using the `copy()` method or the `copy.copy()` function from the `copy` module.

Example of a shallow copy:

import copy

original\_list = [[1, 2, 3], [4, 5, 6]]

shallow\_copy\_list = copy.copy(original\_list)

shallow\_copy\_list[0][0] = 10

print(original\_list) # Output: [[10, 2, 3], [4, 5, 6]]

print(shallow\_copy\_list) # Output: [[10, 2, 3], [4, 5, 6]]

B. \*\*Deep Copy\*\*:

- A deep copy creates a new object and recursively creates new objects for all the elements contained within the original object, including nested objects.

- It ensures that the copied object is fully independent of the original object. Any changes made to the elements in the deep copy will not affect the original object, and vice versa.

- Deep copy can be performed using the `deepcopy()` function from the `copy` module.

Example of a deep copy:

import copy

original\_list = [[1, 2, 3], [4, 5, 6]]

deep\_copy\_list = copy.deepcopy(original\_list)

deep\_copy\_list[0][0] = 10

print(original\_list) # Output: [[1, 2, 3], [4, 5, 6]]

print(deep\_copy\_list) # Output: [[10, 2, 3], [4, 5, 6]]

1. What is the maximum possible length of an identifier?

ANS: In Python, the maximum possible length of an identifier is not explicitly defined. However, there are practical constraints that limit the length of an identifier.

The Python Language Reference states that an identifier is a sequence of letters, digits, and underscores, and it must start with a letter or an underscore. There is no set limit on the number of characters an identifier can have, but very long identifiers can negatively impact code readability and maintainability.

In Python 3.8 and earlier versions, the maximum length of an identifier was limited by the maximum size of a C string, which was typically 256 characters. However, starting from Python 3.9, the maximum size of an identifier has been increased to 2\*\*30 - 1 characters, which is a very large value and effectively allows extremely long identifiers.

While there is technically no fixed maximum length for identifiers in Python, it is generally recommended to keep identifiers concise and meaningful to improve code readability. A good practice is to follow the guidelines outlined in Python's PEP 8 style guide, which suggests using short, lowercase names for most variables and functions and using descriptive names that convey the purpose of the identifier.

1. What is generator comprehension?

ANS: Generator comprehension, also known as generator expression, is a concise and memory-efficient way to create generator objects in Python. It is similar to list comprehension but with one crucial difference: instead of creating a list, it creates a generator on-the-fly, which generates values one at a time as they are needed, rather than generating the entire sequence upfront.

The syntax for generator comprehension is similar to list comprehension, but instead of using square brackets (`[]`), it uses parentheses `()` to enclose the expression.

Here's the general syntax of a generator comprehension:

(generator\_expression for item in iterable if condition)

The generator comprehension iterates over the elements of the `iterable`, applies the given `condition` (if specified), and yields the `generator\_expression` for each item. It generates the values on-the-fly as you iterate over the generator, rather than creating a list containing all the values.

Example using list comprehension:

# List comprehension (creates a list)

squares\_list = [x\*\*2 for x in range(5)]

print(squares\_list) # Output: [0, 1, 4, 9, 16]

Example using generator comprehension:

# Generator comprehension (creates a generator)

squares\_generator = (x\*\*2 for x in range(5))

print(squares\_generator) # Output: <generator object <genexpr> at 0x7f1ee9c05150>

As shown in the example, the generator comprehension creates a generator object. To get the values, you can iterate over the generator using a loop or use functions like `next()` to get one value at a time:

# Iterating over the generator using a loop

for square in squares\_generator:

print(square)

# Using next() to get one value at a time

print(next(squares\_generator)) # Output: 0

print(next(squares\_generator)) # Output: 1

print(next(squares\_generator)) # Output: 4

# and so on...

The advantage of using generator comprehension is that it consumes less memory compared to list comprehension, especially when dealing with large or infinite sequences. Since the values are generated on-the-fly as you iterate over the generator, it saves memory by not storing the entire sequence in memory at once. This makes generator comprehension a valuable tool for efficient processing of large data sets or infinite sequences.