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

A1. Python 3.8 introduced several new features and improvements that enhanced the language's capabilities. Here are some of the key features added in Python 3.8:

**1. Walrus Operator (:=)**

* The walrus operator allows assignment expressions inside expressions, meaning you can assign a value to a variable as part of an expression.
* **Example**:

if (n := len(some\_list)) > 10:

print(f"List is too long with {n} elements")

**2. Positional-Only Parameters**

* Python 3.8 introduced the ability to specify positional-only parameters in function definitions using the / syntax. These parameters must be specified positionally and cannot be passed as keyword arguments.
* **Example**:

def func(a, b, /, c, d):

return a + b + c + d

func(1, 2, c=3, d=4) # Valid

func(a=1, b=2, c=3, d=4) # Raises TypeError

**3. f-String Improvements**

* Python 3.8 enhanced f-strings with the ability to include the = sign to print both the expression and its value.
* **Example**:

x = 10

print(f'{x=}') # Output: x=10

**4. Syntax Warning for Deprecated Features**

* Python 3.8 introduces a new warning category, SyntaxWarning, which can be used to warn about syntactically correct but potentially problematic code.

**5. \_\_import\_\_ Function Improvements**

* The \_\_import\_\_ function was updated to allow the level argument to accept zero for explicit relative imports.

**6. New Syntax for Assignment Expressions in List Comprehensions**

* The walrus operator can now be used inside list comprehensions and other similar constructs to create more concise and readable code.
* **Example**:

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

print([square := x\*\*2 for x in numbers if square > 10])

**7. Shared Memory for Multiprocessing**

* The multiprocessing module now supports shared memory, allowing different processes to share data without serialization, improving efficiency.
* **Example**:

from multiprocessing import shared\_memory

shm = shared\_memory.SharedMemory(create=True, size=10)

**8. New math Functions**

* Python 3.8 introduced new functions in the math module, such as math.isqrt() (integer square root), math.prod() (product of an iterable), math.perm() (number of permutations), and math.comb() (number of combinations).
* **Example**:

import math

print(math.isqrt(16)) # Output: 4

print(math.prod([1, 2, 3, 4])) # Output: 24

**9. Type Hinting Enhancements**

* Python 3.8 added support for typing in more advanced ways, such as TypedDict and final variables using typing.Final.
* **Example**:

from typing import TypedDict, Final

class Point(TypedDict):

x: int

y: int

PI: Final = 3.14159

**10. re Module Enhancements**

* The re module received optimizations and new features, such as the addition of re.fullmatch().

**11. Improved Debugging with faulthandler**

* The faulthandler module now includes the ability to dump tracebacks on timeout, which is useful for debugging hanging processes.

**12. Other Improvements**

* Many other smaller improvements, optimizations, and standard library updates were included in Python 3.8. These include performance improvements, new modules, and extended functionality in existing modules.

1. What is monkey patching in Python?

A2. **Monkey Patching in Python**

**Monkey patching** is the term used for dynamically modifying a class or module at runtime. This means you can change the behavior of existing code without altering the original source.

**How it works**

* **Import the module or class:** You need to import the module or class you want to modify.
* **Replace the function or attribute:** Assign a new function or value to the attribute of the module or class you want to change.

**Example**

Python

import math

def my\_new\_sqrt(x):

return x \* x

# Monkey patch the sqrt function

math.sqrt = my\_new\_sqrt

print(math.sqrt(4)) # Output: 16

**Caution**

While monkey patching can be useful in certain scenarios, it's generally considered a bad practice. It can make code harder to understand, maintain, and test. It's often better to use inheritance or composition to extend functionality rather than modifying existing code directly.

**Common Use Cases (with caution):**

* **Testing:** Temporarily modifying behavior for testing purposes.
* **Debugging:** Quickly fixing issues in third-party libraries.
* **Experimentation:** Trying out different implementations without modifying the original code.

**Alternatives to Monkey Patching:**

* **Inheritance:** Create a subclass and override methods.
* **Composition:** Create a new class that delegates to the original class.
* **Decorator patterns:** Modify function behavior without altering the original function.

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

A3. **Shallow Copy vs. Deep Copy**

**Shallow Copy**

* Creates a new object but copies only the references to the original object's data.
* Changes to the original object will be reflected in the copy.
* Often faster than deep copy.

**Deep Copy**

* Creates a new object and recursively copies all nested objects.
* Changes to the original object will not affect the copy.
* Generally slower than shallow copy due to the overhead of creating new objects.

**Example:**

Python

import copy

# Shallow copy

list1 = [1, [2, 3]]

list2 = list1.copy() # Shallow copy

# Deep copy

list3 = copy.deepcopy(list1)

# Modifying the original list

list1[1][0] = 4

print(list1) # Output: [1, [4, 3]]

print(list2) # Output: [1, [4, 3]] (shallow copy, changes reflected)

print(list3) # Output: [1, [2, 3]] (deep copy, no changes)

**Key points:**

* For immutable objects like strings, numbers, and tuples, shallow and deep copies are essentially the same.
* For mutable objects like lists, dictionaries, and custom classes, the difference becomes significant.
* Choose shallow copy when performance is critical and you don't need independent copies.
* Choose deep copy when you need to ensure that changes to one object don't affect the other.

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

A4. **The maximum length of an identifier in Python is theoretically unlimited.**

However, **PEP 8, the Python style guide**

, recommends a maximum line length of 79 characters, which indirectly limits the practical length of identifiers. Extremely long identifiers can hinder readability and maintainability.

It's essential to strike a balance between descriptive identifier names and code clarity. While Python doesn't impose a strict limit, adhering to style guidelines is crucial for writing clean and maintainable code.

1. What is generator comprehension?

A5. **Generator Comprehension**

**Generator comprehension** is a concise way to create generators in Python. It's similar to list comprehension but returns a generator object instead of a list.

**Syntax:**

Python

(expression for item in iterable if condition)

Use code [with caution.](/faq#coding)

* expression: The value to yield for each item in the iterable.
* item: The iteration variable.
* iterable: Any iterable object (like a list, tuple, string, or range).
* if condition (optional): A conditional expression to filter items.

**Example:**

Python

even\_numbers = (num for num in range(10) if num % 2 == 0)

print(even\_numbers) # Output: <generator object at ...>

for num in even\_numbers:

print(num) # Output: 0, 2, 4, 6, 8

**Key Points:**

* Generator comprehensions are memory-efficient as they generate values on-the-fly.
* They are often used for large datasets or infinite sequences.
* The yield keyword is implicitly used behind the scenes.