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

Python 3.8 introduced several new features and improvements, including:

1. **Assignment Expressions (the Walrus Operator):** This allows assigning values to variables within expressions, using the := operator.
2. **Positional-Only Parameters (/):** Functions can now define parameters that must be specified positionally and not as keyword arguments, improving clarity and flexibility.
3. **f-strings support =:** Debugging f-strings became easier with the ability to include expressions and their values directly.
4. **Parallel File System Cache for Compiled Bytecode:** This improved startup time for Python programs.
5. **continue in finally blocks:** The continue statement is now allowed within finally blocks, offering more control flow options.
6. **TypedDict:** This provides a way to define dictionaries with specific key types and value types, improving type hints.
7. **Performance improvements:** Python 3.8 also included various performance optimizations, making it faster than previous versions.

**2. What is monkey patching in Python?**

Monkey patching in Python refers to the technique of modifying or extending the behavior of existing code (such as modules, classes, or functions) at runtime without directly changing the original source code. This is usually done by reassigning attributes or methods dynamically.

**Why use monkey patching?**

* **Fixing bugs in external libraries:** You might use monkey patching to temporarily fix a bug in a third-party library while waiting for an official patch.
* **Adding functionality:** You can add new features to existing classes or modules without modifying their source code.
* **Testing:** Monkey patching is often used in testing to replace real implementations with mock objects to control behavior and isolate components under test.

**Example:**

Python

import module

def my\_new\_function():

# Your custom implementation

module.original\_function = my\_new\_function # Monkey patch

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

play\_circleeditcontent\_copy

**Cautions:**

* **Use sparingly:** Overusing monkey patching can make your code harder to understand and maintain, as the behavior might not be obvious from the original source.
* **Compatibility:** Be cautious when monkey patching external libraries, as updates to those libraries might break your patches.

Overall, monkey patching can be a powerful tool when used judiciously, but it's essential to be aware of its potential drawbacks and use it responsibly.

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

In Python, both shallow copy and deep copy create a new object from an existing one, but they differ in how they handle nested objects (objects that contain other objects, like lists or dictionaries).

**Shallow Copy:**

* **Creates a new object:** A new object is created.
* **Copies references:** For nested objects, only references to the original nested objects are copied. This means the new object and the original object will share the same nested objects.
* **Changes affect both:** If you modify a nested object in either the original or the shallow copy, the change will be reflected in both.
* **How to create:** Use the copy.copy() function or the object's copy() method (if available).

**Example:**

import copy

original\_list = [[1, 2], 3]

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

shallow\_copy[0][1] = 5 # Modifies the nested list in both

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

print(shallow\_copy) # Output: [[1, 5], 3]

**Deep Copy:**

* **Creates a new object:** A new object is created.
* **Recursively copies nested objects:** All nested objects are also recursively copied, creating entirely independent copies of the original object and its nested objects.
* **Changes don't affect each other:** Modifying a nested object in either the original or the deep copy will not affect the other.
* **How to create:** Use the copy.deepcopy() function.

**Example:**

Python

import copy

original\_list = [[1, 2], 3]

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

deep\_copy[0][1] = 5 # Modifies only the nested list in the deep copy

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

print(deep\_copy) # Output: [[1, 5], 3]

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

play\_circleeditcontent\_copy

**When to use which:**

* **Shallow copy:** Use when you need a quick copy of an object and don't intend to modify any nested objects.
* **Deep copy:** Use when you need a completely independent copy of an object, including all its nested objects, so that changes to one copy don't affect the other.

tuneshare

more\_vert

**4. What is the maximum possible length of an identifier?**

Technically, there is no strict limit on the length of identifiers in Python. However, the PEP 8 style guide, which is widely followed in the Python community, recommends a maximum line length of 79 characters. This indirectly limits the practical length of identifiers, as overly long identifiers can make code harder to read and maintain.

While you could technically create identifiers longer than 79 characters, it's generally discouraged for the sake of readability and adherence to standard practices.

**5. What is generator comprehension?**

In Python, a generator comprehension is a concise way to create a generator object. It's similar to a list comprehension, but instead of creating a list containing all the elements, it creates a generator that produces the elements one at a time as needed. This makes generator comprehensions more memory efficient, especially when dealing with large or infinite sequences.

**Syntax:**

Generator comprehensions use parentheses () instead of the square brackets [] used in list comprehensions. The general syntax is:

Python

(expression for item in iterable if condition)

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

play\_circleeditcontent\_copy

* expression: The operation to perform on each item.
* item: The variable representing each item in the iterable.
* iterable: The sequence (list, tuple, string, etc.) to iterate over.
* condition (optional): A filtering condition that determines which items are included.

**Example:**

Python

squares\_generator = (x\*\*2 for x in range(10))

for square in squares\_generator:

print(square) # Outputs: 0 1 4 9 16 25 36 49 64 81

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

play\_circleeditcontent\_copy

**Key differences from list comprehensions:**

* **Memory efficiency:** Generator comprehensions generate values on the fly, whereas list comprehensions store all elements in memory.
* **Laziness:** Generators produce values only when requested, while list comprehensions create the entire list upfront.
* **Syntax:** Generator comprehensions use parentheses (), while list comprehensions use square brackets [].

**When to use generator comprehensions:**

* When dealing with large datasets where storing all the elements in memory would be impractical.
* When you need to iterate over a sequence only once.
* When you want to create an infinite sequence (using an infinite iterable like itertools.count()).