**PYTHON ADVANCE ASSIGNMENT\_15**

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

Python 3.8, which was released in October 2019, introduced several new features and improvements, including:

Assignment expressions: a new syntax for performing an assignment and returning the value of the assignment in a single expression, using the := operator.

Positional-only parameters: a new syntax for defining function parameters that can only be passed positionally, and not by keyword.

f-strings now support = for debugging: f-strings can now be used to display the value of an expression using the = specifier, which can be useful for debugging.

The walrus operator (:=) is used to assign a value to a variable while testing it in a single line of code.

The Typing module has been improved to support standard types such as List, Dict, Tuple, and Set more directly.

The importlib.metadata module has been added to provide an interface for accessing metadata about packages that are installed with pip, including version numbers, authors, and other information.

Python 3.8 also includes several performance optimizations and other smaller improvements, such as faster f-strings, improved traceback messages, and better handling of UTF-8 encoded text files.

These are just a few of the new features and improvements introduced in Python 3.8. For a full list of changes and additions, you can refer to the Python 3.8 documentation.

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

Monkey patching is a technique in Python where you can modify or extend the behavior of existing objects or modules at runtime, by modifying their attributes or methods. It involves dynamically replacing a method or attribute on an object with a new implementation, allowing you to change the behavior of an object without changing its source code.

For example, let's say you have a module that defines a function called add that adds two numbers together:

# mymodule.py

def add(x, y):

return x + y

Now let's say you want to modify the behavior of the add function in a specific situation. Instead of modifying the mymodule.py file, you can use monkey patching to change the implementation of the add function at runtime, like this:

import mymodule

def new\_add(x, y):

return x \* y

mymodule.add = new\_add

This code replaces the original add function with the new implementation, which multiplies the two numbers instead of adding them.

Monkey patching can be useful in certain situations, such as when you want to temporarily modify the behavior of a third-party library or when you need to test a specific edge case that is difficult to reproduce otherwise. However, it can also make your code harder to read and maintain, and can lead to unexpected behavior if not used carefully. It's important to use monkey patching judiciously and document any changes you make to existing code.

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

In Python, a copy of an object can be created using either a shallow copy or a deep copy. The main difference between the two is how they handle the copying of nested objects.

A shallow copy creates a new object, but does not create new copies of the nested objects. Instead, the copy points to the same nested objects as the original. In other words, any changes made to the nested objects will be reflected in both the original and the copy. Shallow copy can be created using the copy() method or the [:] slice operator.

Here's an example of creating a shallow copy:

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

shallow\_copy = original\_list.copy()

shallow\_copy[2][0] = 'changed'

print(original\_list) # [1, 2, ['changed', 4]]

print(shallow\_copy) # [1, 2, ['changed', 4]]

In contrast, a deep copy creates a new object and also creates new copies of all the nested objects. This means that any changes made to the nested objects in the original will not be reflected in the copy, and vice versa. Deep copy can be created using the copy.deepcopy() function from the copy module.

Here's an example of creating a deep copy:

import copy

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

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

deep\_copy[2][0] = 'changed'

print(original\_list) # [1, 2, [3, 4]]

print(deep\_copy) # [1, 2, ['changed', 4]]

As we can see, the original list is unchanged by the modification to the nested object in the deep copy.

In general, shallow copies are faster and use less memory than deep copies, but they may not always be appropriate if you need to make independent copies of nested objects. The choice between shallow and deep copy depends on the specific use case and how the copied objects will be used.

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

In Python, the maximum possible length of an identifier is technically unlimited. However, it's generally recommended to keep identifiers (such as variable names, function names, and class names) reasonably short and descriptive for readability.

According to the Python language reference, an identifier is defined as a sequence of one or more letters, underscores, and digits. The first character of an identifier must be a letter or an underscore. Identifiers are case-sensitive, meaning that foo and Foo are considered to be two different identifiers.

There is no specific maximum length for identifiers in Python, but very long identifiers can make your code harder to read and maintain. PEP 8, the official Python style guide, recommends limiting the length of identifiers to 79 characters or fewer. In practice, most Python code uses identifiers that are much shorter than that.

In addition to length, it's also important to choose descriptive and meaningful names for your identifiers. This can help make your code more understandable and maintainable over time.

**5. What is generator comprehension?**

In Python, a generator comprehension is a compact and memory-efficient way to create a generator object using a single line of code. It is similar to a list comprehension, but instead of creating a list, it creates a generator that can be used to generate values on the fly, as needed.

The syntax for a generator comprehension is very similar to that of a list comprehension, with the main difference being the use of parentheses instead of square brackets. Here's an example:

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

This code creates a generator that yields the square of each number from 0 to 9, one at a time, as the generator is iterated over. Note that the generator comprehension is enclosed in parentheses, rather than square brackets as in a list comprehension.

Like a list comprehension, a generator comprehension can include any number of for and if clauses to filter and transform the values being generated. For example:

even\_squares = (x\*\*2 for x in range(10) if x % 2 == 0)

This generator comprehension generates only the squares of even numbers in the range 0 to 9.

The main advantage of using a generator comprehension instead of a list comprehension is that it is more memory-efficient, especially for large data sets. Because a generator yields values on-the-fly, it only stores one value in memory at a time, rather than creating a list of all values in memory at once. This can be a significant performance advantage when dealing with large amounts of data or when memory is limited.