TOP 25

ADVANCE LEVEL

PYTHON

INTERVIEW QUESTIONS

Curated by tutort academy

NOTE

These are advance level question that are actually acted these days, and you can't answer these if you don't learn these before.

QUESTION 1

What is the Global Interpreter Lock (GIL) in Python, and how does it impact multi-threading?

The Global Interpreter Lock (GIL) is a mutex (short for mutual exclusion) that allows only one thread to execute Python bytecode at a time, even on multi-core processors. This limitation affects multi-threading in Python because it prevents true parallel execution of threads, limiting the performance benefits of multi-core CPUs.

While the GIL makes Python threads less efficient for CPU-bound tasks, it is not a problem for I/O-bound tasks, as threads can yield the GIL during I/O operations. To achieve parallelism in CPU-bound tasks, you can use the multiprocessing module, which creates separate processes, each with its own Python interpreter and memory space.

Example demonstrating GIL impact:

```
import threading
def count_up():
for _ in range(1000000):
pass
def count_down():
for _ in range(1000000):
pass
thread1 = threading.Thread(target=count_up)
thread2 = threading.Thread(target=count_down)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

QUESTION 2

Explain the differences between shallow copy and deep copy in Python, and how can you create them?

In Python, a shallow copy and a deep copy are two ways to duplicate objects:

Shallow Copy: A shallow copy creates a new object but does not create copies of nested objects within the original object. Instead, it references the same nested objects. You can create a shallow copy using the copy module's copy() function or by slicing.

```
import copy
original_list = [[1, 2, 3], [4, 5, 6]]
shallow_copied_list = copy.copy(original_list)
```

Deep Copy: A deep copy creates a completely independent copy of the original object and all its nested objects. You can create a deep copy using the copy module's deepcopy() function.

```
import copy
original_list = [[1, 2, 3], [4, 5, 6]]
deep_copied_list = copy.deepcopy(original_list)
```

QUESTION 3

What is the purpose of Python decorators? Provide an example of how to use them.

Python decorators are a powerful feature that allows you to modify or extend the behavior of functions or methods without changing their source code. They are often used for tasks such as logging, authentication, and memoization.

Here's an example of a simple decorator that logs the execution time of a function:

```
import time
def timing_decorator(func):
def wrapper(*args, **kwargs):
start_time = time.time()
result = func(*args, **kwargs)
end_time = time.time()
print(f"{func.__name__} took {end_time - start_time}
seconds to execute.")
return result
return wrapper
@timing_decorator
def some_function():
# Your code here
time.sleep(2)
some_function()
```

Subhadip Chowdhury

From AMERICAN EXPRESS → To amazon

Placed with **100% Hike**

QUESTION 4

What is a generator in Python? How does it differ from a regular function?

A generator in Python is a type of iterable, but it doesn't store all its values in memory at once. Instead, it yields values one at a time, which makes it memory-efficient for working with large datasets or infinite sequences. Generators are defined using functions with the yield keyword.

Here's an example of a simple generator function that generates Fibonacci numbers:

```
def fibonacci_generator():
a, b = 0, 1
while True:
yield a
a, b = b, a + b
# Usage:
fib_gen = fibonacci_generator()
for _ in range(10):
print(next(fib_gen))
```

Regular functions, on the other hand, compute and return a value using the return statement, and they do not retain their state between calls.

QUESTION 5

Explain the purpose of the Python __init__ and __del__ methods in classes.

The **__init__** and **__del__** methods are special methods in Python classes:

__init__: This method is the constructor of a class and is called when an instance of the class is created. It initializes the attributes and sets up the object's initial state.

```
class MyClass:
def __init__(self, x):
self.x = x
obj = MyClass(5) # Calls __init__ with x=5
```

__del__: This method is called when an object is about to be destroyed and its memory is being deallocated. It can be used to perform cleanup operations or release resources associated with the object.

```
class MyClass:
def __del__(self):
print("Object is being destroyed")
obj = MyClass()
del obj # Calls __del__ method
```

Note: The use of __del__ should be cautious, as Python's garbage collector often handles object destruction automatically, and you may not need to define this method in most cases.

QUESTION 6

What is a closure in Python, and why is it useful?

A closure is a nested function that remembers and has access to variables from its containing (enclosing) function's scope even after the containing function has finished executing. Closures are useful for creating function factories and maintaining state across multiple function calls.

Example of a closure:

```
def outer_function(x):
def inner_function(y):
return x + y
return inner_function
closure = outer_function(10)
result = closure(5) # result is 15
```

QUESTION 7

Explain the use of the super() function in Python and provide an example.

The super() function is used to call a method from the parent (super) class in the context of a subclass. It is often used to invoke the constructor of the parent class within the constructor of a subclass.

Example of using super():

```
class Parent:
def __init__(self, x):
self.x = x
class Child(Parent):
def __init__(self, x, y):
super().__init__(x)
self.y = y
child_obj = Child(10, 20)
print(child_obj.x) # Output: 10
print(child_obj.y) # Output: 20
```

QUESTION 8

What is method resolution order (MRO) in Python, and how is it determined?

Method Resolution Order (MRO) is the order in which Python looks for methods and attributes in a class hierarchy. It is determined using the C3 Linearization algorithm, which ensures a consistent and predictable order when multiple inheritance is involved.

```
Code- class A:
def method(self):
print("A method")
class B(A):
def method(self):
print("B method")
class C(A):
def method(self):
print("C method")
class D(B, C):
pass
d = D()
d.method() # Output: B method
```

QUESTION 9

What is the purpose of the async and await keywords in Python, and how are they used for asynchronous programming?

The async and await keywords are used to define and work with asynchronous code in Python, particularly in asynchronous functions (coroutines). async marks a function as asynchronous, and await is used to pause the execution of an asynchronous function until an asynchronous operation completes.

Example of using async and await:

```
import asyncio
async def fetch_data(url):
response = await asyncio.gather(
asyncio.to_thread(requests.get, url),
asyncio.to_thread(requests.get, url)
)
return response
# Usage:
result = asyncio.run(fetch_data("https://tutort.com"))
```

QUESTION 10

Explain the Global Interpreter Lock (GIL) in Python and its impact on multi-threading. How can you achieve parallelism in Python despite the GIL?

(Continuation from question 1) The Global Interpreter Lock (GIL) is a mutex that allows only one thread to execute Python bytecode at a time, limiting the efficiency of multi-threading for CPU-bound tasks. To achieve parallelism, you can use the multiprocessing module, which creates separate processes, each with its own Python interpreter and memory space.

Example using multiprocessing for parallelism:

```
import multiprocessing
def worker_function(x):
return x * x
if __name__ == "__main__":
pool = multiprocessing.Pool(processes=4)
result = pool.map(worker_function, [1, 2, 3, 4])
pool.close()
pool.join()
```

QUESTION 11

What is the Global Interpreter Lock (GIL) in Python, and how does it impact multi-threading?

(Continuation from question 1) The Global Interpreter Lock (GIL) is a mutex (short for mutual exclusion) that allows only one thread to execute Python bytecode at a time, even on multi-core processors. This limitation affects multi-threading in Python because it prevents true parallel execution of threads, limiting the performance benefits of multi-core CPUs.

While the GIL makes Python threads less efficient for CPU-bound tasks, it is not a problem for I/O-bound tasks, as threads can yield the GIL during I/O operations. To achieve parallelism in CPU-bound tasks, you can use the multiprocessing module, which creates separate processes, each with its own Python interpreter and memory space.

Example demonstrating GIL impact:

```
import threading
def count_up():
    for _ in range(1000000):
    pass
def count_down():
    for _ in range(1000000):
    pass
thread1 = threading.Thread(target=count_up)
thread2 = threading.Thread(target=count_down)
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

QUESTION 12

Explain the differences between shallow copy and deep copy in Python, and how can you create them?

(Continuation from question 2) In Python, a shallow copy and a deep copy are two ways to duplicate objects:

Shallow Copy: A shallow copy creates a new object but does not create copies of nested objects within the original object. Instead, it references the same nested objects. You can create a shallow copy using the copy module's **copy()** function or by slicing.

```
import copy
original_list = [[1, 2, 3], [4, 5, 6]]
shallow_copied_list = copy.copy(original_list)
```

Deep Copy: A deep copy creates a completely independent copy of the original object and all its nested objects. You can create a deep **copy** using the copy module's **deepcopy()** function.

```
import copy
original_list = [[1, 2, 3], [4, 5, 6]]
deep_copied_list = copy.deepcopy(original_list)
```

QUESTION 13

What is the purpose of Python decorators? Provide an example of how to use them.

(Continuation from question 3) Python decorators are a powerful feature that allows you to modify or extend the behavior of functions or methods without changing their source code. They are often used for tasks such as logging, authentication, and memoization.

Here's an example of a simple decorator that logs the execution time of a function:

```
import time
    def timing_decorator(func):
    def wrapper(*args, **kwargs):
start_time = time.time()
result = func(*args, **kwargs)
    end_time = time.time()
print(f"{func.__name__} took {end_time - start_time}
seconds to execute.")
    return result
    return wrapper
@timing_decorator
    def some_function():
# Your code here
time.sleep(2)
        some_function()
```

QUESTION 14

What is a generator in Python? How does it differ from a regular function?

(Continuation from question 4) A generator in Python is a type of iterable, but it doesn't store all its values in memory at once. Instead, it yields values one at a time, which makes it memory-efficient for working with large datasets or infinite sequences. Generators are defined using functions with the **yield** keyword.

Here's an example of a simple generator function that generates Fibonacci numbers:

```
def fibonacci_generator():
a, b = 0, 1
while True:
yield a
a, b = b, a + b
# Usage:
fib_gen = fibonacci_generator()
for _ in range(10):
print(next(fib_gen))
```

Shaloni Gangrade

From AMERICAN EXPRESS → To TARGET

Placed with 100% Hike

QUESTION 15

Explain the purpose of the @staticmethod and @classmethod decorators in Python, and provide examples of when to use them.

The **@staticmethod** and **@classmethod** decorators are used to define methods that are associated with a class rather than an instance of the class.

- @staticmethod: This decorator defines a static method that doesn't depend on instance-specific data. Static methods can be called on the class itself, without creating an instance of the class.

```
class MathUtility:
@staticmethod
def add(x, y):
return x + y
result = MathUtility.add(5, 3)
```

- @classmethod: This decorator defines a class method that takes the class itself as its first argument. Class methods are often used for alternative constructors or to access and modify class-level attributes.

```
class MyClass:
    count = 0
def __init__(self, value):
    self.value = value
MyClass.count += 1
```

```
@classmethod
def get_instance_count(cls):
    return cls.count
    obj1 = MyClass(10)
    obj2 = MyClass(20)
count = MyClass.get_instance_count() # count is 2
```

QUESTION 16

Explain the purpose of the @property decorator in Python, and provide an example of its usage.

The @property decorator allows you to define a method as an attribute, making it accessible like an attribute while still executing custom logic when it is accessed.

Example of using **@property:**

```
class Circle:
def __init__(self, radius):
self._radius = radius
    @property
def radius(self):
return self._radius
    @radius.setter
def radius(self, value):
    if value < 0:
    raise ValueError("Radius
    cannot be negative")
self._radius = value
    @property
def area(self):
    return 3.14 * self._radius * self._radius
circle = Circle(5)
print(circle.radius) # Access radius as if it's an attribute
circle.radius = 7 # Use the setter to change radius
print(circle.area) # Access area as a computed property
```

QUESTION 17

What is the purpose of the __str__ and __repr__ methods in Python classes?

The __str__ and __repr__ methods are used to define how objects of a class should be represented as strings when they are printed or converted to a string.

__str__: This method should return a user-friendly, informal string representation of the object. It is called by the **str()** function and when using **print()**.

```
class MyClass:
 def __str__(self):
     return "This is a MyClass object"
```

__repr__: This method should return an unambiguous, formal string representation of the object, which can be used to recreate the object. It is called by the **repr()** function and can also be used for debugging.

```
class MyClass:
 def __init__(self, x):
     self.x = x
 def __repr__(self):
     return f"MyClass({self.x})"
```

QUESTION 18

Explain how context managers work in Python and provide an example of creating a custom context manager.

Context managers in Python are used to set up and tear down resources and ensure that they are properly managed, even in the presence of exceptions. They are typically used with the with statement.

Example of a custom context manager:

```
class MyContextManager:
 def __enter__(self):
     print("Entering the context")
     return self  # The object to be used within the context

 def __exit__(self, exc_type, exc_value, traceback):
     print("Exiting the context")
     # Optional: Handle exceptions and return True if handled

# Usage:
with MyContextManager() as cm:
 print("Inside the context")
```

QUESTION 19

What is the purpose of the __str__ and __repr__ methods in Python classes?

The __str__ and __repr__ methods are used to define how objects of a class should be represented as strings when they are printed or converted to a string.

- **__str__**: This method should return a user-friendly, informal string representation of the object. It is called by the str() function and when using print().

```
class MyClass:
    def __str__(self):
        return "This is a MyClass object"
```

- **__repr__**: This method should return an unambiguous, formal string representation of the object, which can be used to recreate the object. It is called by the repr() function and can also be used for debugging.

```
class MyClass:
    def __init__(self, x):
        self.x = x
    def __repr__(self):
        return f"MyClass({self.x})"
```

QUESTION 20

Explain how context managers work in Python and provide an example of creating a custom context manager.

Context managers in Python are used to set up and tear down resources and ensure that they are properly managed, even in the presence of exceptions. They are typically used with the with statement.

Example of a custom context manager:

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self # The object to be used within the context
    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")
    # Optional: Handle exceptions and return True if handled
    # Usage:
    with MyContextManager() as cm:
        print("Inside the context")
```

Disha Patil

fractal

Tutort Academy is always ready to provide job assistance in various organizations such as MNCs and Startups, as well as help you with resume writing, mock interviews, LinkedIn profile creation, and everything else related to job search. They make you industry-ready.

QUESTION 21

Explain the purpose of the functools module in Python and provide an example of using its functions.

The functools module in Python provides higher-order functions and operations on callable objects. It includes functions like **partial**, **reduce**, and **wraps** for various purposes.

Example of using **functools.partial:**

```
from functools import partial
# Create a partial function with a fixed argument
        multiply_by_two = partial(lambda x, y: x * y, y=2)



result = multiply_by_two(5) # Equivalent to calling lambda x: x * 2 with x=5
```

QUESTION 22

What is monkey patching in Python, and when might it be used?

Monkey patching refers to the practice of dynamically modifying or extending the behavior of existing classes or modules at runtime. It can be used to add, replace, or modify functions or methods of existing code, often to fix bugs or add new features without modifying the original source code.

Example of monkey patching:

```
# Original class
class Calculator:
    def add(self, x, y):
        return x + y

# Monkey patching to add a new method
def multiply(self, x, y):
    return x * y

Calculator.multiply = multiply
# Usage
    calc = Calculator()
    result = calc.multiply(3, 4) # Result: 12
```

QUESTION 23

What are metaclasses in Python, and how are they different from regular classes?

Metaclasses in Python are classes that define the behavior of other classes, often referred to as "class factories." They control the creation and behavior of classes at the class level. Metaclasses are used for advanced purposes, such as enforcing coding standards, code generation, and custom class creation.

Example of defining a metaclass:

```
class MyMeta(type):
    def __init__(cls, name, bases, attrs):
        # Perform custom actions here
        pass
class MyClass(metaclass=MyMeta):
    def __init__(self, x):
        self.x = x
# MyClass is created with MyMeta as its metaclass
```

Sayan Banerjee

From ORACLE → To servicenow

Switch from Service Based Company

QUESTION 24

Explain the Global Interpreter Lock (GIL) in Python and its impact on multi-threading. How can you achieve parallelism in Python despite the GIL?

(Continuation from question 10) The Global Interpreter Lock (GIL) is a mutex that allows only one thread to execute Python bytecode at a time, limiting the efficiency of multi-threading for CPU-bound tasks. To achieve parallelism, you can use the **multiprocessing** module, which creates separate processes, each with its own Python interpreter and memory space.

Example using **multiprocessing** for parallelism:

```
import multiprocessing

def worker_function(x):
return x * x

if __name__ == "__main__":
pool = multiprocessing.Pool(processes=4)
result = pool.map(worker_function, [1, 2, 3, 4])
pool.close()
pool.join()
```

QUESTION 25

Explain the concept of metaprogramming in Python and provide an example of its use.

Metaprogramming in Python refers to the ability to write code that can manipulate or generate code dynamically at runtime. It allows you to create, modify, or analyze Python code programmatically, often using introspection and reflection. Metaprogramming is a powerful technique used in libraries, frameworks, and dynamic systems.

Example of metaprogramming using Python's **exec()** function to dynamically create and execute code:

```
# Dynamic code generation
variable_name = "x"
variable_value = 42
dynamic_code = f"{variable_name} = {variable_value}"

# Execute the dynamically generated code
exec(dynamic_code)

# Access the variable created dynamically
print(x) # Output: 42
```

In this example, we dynamically generate Python code as a string and execute it using the **exec()** function. This technique is useful in situations where code generation or dynamic behavior is required, but it should be used with caution to avoid security risks and maintainability issues.

tutort

Start Your Upskilling with us

Join our Job-Oriented Programs

Explore More

www.tutort.net

Watch us on Youtube

Read more on Quora

Explore our courses

Data Science & Machine Learning

Full Stack Data Science (AI & ML)

Follow us on

LinkedIn Instagram YouTube

Phone
+91-8712338901

E-mail
contact@tutort.net