Python Knowledge

1. What is the difference between == and is in Python? Answer:

== checks for value equality. It determines if the values of two objects are equal. is checks for identity equality. It determines if two references point to the same object in memory.

---

2. How does Python's garbage collection work? Answer: Python uses automatic garbage collection to manage memory. It primarily uses reference counting and a cyclic garbage collector to detect and clean up unused objects.

Reference Counting: Each object has a reference count that increases when a reference to the object is created and decreases when a reference is deleted. Cyclic Garbage Collector: Handles cyclic references by identifying and collecting groups of objects that reference each other but are not reachable from the program.

---

3. What are Python decorators and how do you use them? Answer: Decorators are a way to modify or enhance functions or methods without changing their definition. They are typically used to add functionality to existing code in a reusable way.

Example of a decorator:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

---

4. What are list comprehensions and generator expressions? Provide examples.

Answer:

List Comprehensions: A concise way to create lists.

```
squares = [x**2 for x in range(10)]
print(squares)  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Generator Expressions: Similar to list comprehensions, but they return a generator object which yields items one at a time and uses less memory.

```
squares_gen = (x**2 for x in range(10))
print(list(squares_gen))  # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

---

5. Explain the use of self in Python classes. Answer: self is a reference to the current instance of the class and is used to access variables and methods associated with the class. It allows each object to keep track of its own data.

Example:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years
old.")

person = Person("Alice", 30)
person.greet()  # Output: Hello, my name is Alice and I am 30 years old.
```

---

6. What is the Global Interpreter Lock (GIL) in Python? Answer: The Global Interpreter Lock (GIL) is a mutex that protects access to Python objects, preventing multiple native threads from executing Python bytecodes at once. This means that even in a multi-threaded Python program, only one thread can execute Python code at a time. The GIL can be a limitation for CPU-bound programs but doesn't affect I/O-bound programs as much.

---

7. How do you handle exceptions in Python? Provide an example. Answer: Exceptions in Python are handled using try, except, else, and finally blocks. The try block contains the code that might raise an exception, the except block contains the code to handle the exception, the else block (optional) runs if no exceptions are raised, and the finally block (optional) contains code that always runs regardless of whether an exception was raised.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Caught an exception: {e}")
else:
    print("No exceptions were raised.")
finally:
    print("This block runs no matter what.")
```

Output:

```
Caught an exception: division by zero
This block runs no matter what.
```

---

8. What are the differences between staticmethod and classmethod in Python? Answer:

@staticmethod: A static method does not receive any implicit first argument. It behaves like a plain function that belongs to a class's namespace. @classmethod: A class method receives the class as its first argument, which is typically named cls. It can access and modify class state.

Example:

```
class MyClass:
    @staticmethod
    def static_method():
        return "This is a static method"

    @classmethod
    def class_method(cls):
        return f"This is a class method of {cls.__name__}"

print(MyClass.static_method())  # Output: This is a static method
print(MyClass.class_method())  # Output: This is a class method of MyClass
```

Additionally, you don't have to instantiate an object in order to call these methods.

---

9. What is a Python generator and how does it work? Answer: A generator in Python is a function that returns an iterator which we can iterate over (one value at a time). Generators are written using the yield statement instead of return. They are memory efficient because they produce items one at a time and only when required.

Example:

```

```

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

counter = count_up_to(5)
for number in counter:
    print(number)
```

Output:

```
1
2
3
4
5
```

---

10. Explain the concept of list slicing and provide an example. Answer: List slicing allows you to access a subset of a list by specifying a start, stop, and step index. The syntax for slicing is list[start:stop:step].

Example:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Slicing examples
print(numbers[2:7])      # Output: [2, 3, 4, 5, 6]
print(numbers[:5])       # Output: [0, 1, 2, 3, 4]
print(numbers[5:])       # Output: [5, 6, 7, 8, 9]
print(numbers[::2])      # Output: [0, 2, 4, 6, 8]
print(numbers[::-1])     # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

---

11. Explain the use of *args and **kwargs in function definitions. Answer:

*args: Allows a function to accept a variable number of positional arguments. It collects additional positional arguments as a tuple.

```
def func(*args):
    for arg in args:
        print(arg)

func(1, 2, 3)  # Output: 1 2 3
```

**kwargs: Allows a function to accept a variable number of keyword arguments. It collects additional keyword arguments as a dictionary.

```
def func(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} = {value}")

func(a=1, b=2, c=3)  # Output: a = 1 b = 2 c = 3
```

---

12. Can an abstract class be instantiated? Answer:

No, an abstract class in Python cannot be instantiated even if it contains no abstract methods. The primary purpose of an abstract class is to serve as a base class that provides a common interface and possibly some common functionality for its subclasses. The abstract class itself is not meant to be instantiated directly.

In Python, the ABC class from the abc module is used to create abstract classes. Once a class inherits from ABC, it is considered an abstract class and cannot be instantiated, regardless of whether it has any abstract methods.

---

13. Explain `if __name__ == "__main__":`

`__name__`

- In Python, `__name__` is a special built-in variable that gets assigned a string value based on how the script is being executed. The value of `__name__` will be:
- `"__main__"` if the script is being run directly. The name of the module if the script is being imported.

`if __name__ == "__main__":`

- This line of code checks if the script is being run directly (i.e., not imported as a module). If the condition is true, the block of code under this statement will be executed.
- Why Use It?
    - Modularity: It allows you to write code that can be reused as a module without running certain parts of the code. For example, you can define functions and classes in a script and test them in the same script without executing the test code when the script is imported elsewhere.
    - Testing: It is often used to include test code or demo code that should only run when the script is executed directly, and not when it is imported.

14. What is Polymorphism? (Static vs. Dynamic Polymorphism) Types of Polymorphism

- **Compile-time Polymorphism (Static Polymorphism) - Overloading**: (In the same class) Achieved through method overloading (same method name with different parameters) and operator overloading (same operator with different meanings). Python does not support method overloading directly but supports operator overloading.

- **Run-time Polymorphism (Dynamic Polymorphism) - Overriding**: (In different classes) Achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.

Method Overriding (Run-time Polymorphism)

In this example, we have a base class Animal with a method make_sound. Two subclasses, Dog and Cat, override the make_sound method.

```
class Animal:
    def make_sound(self):
        raise NotImplementedError("Subclass must implement abstract
method")

class Dog(Animal):
    def make_sound(self):
        return "Bark"

class Cat(Animal):
    def make_sound(self):
        return "Meow"

def animal_sound(animal: Animal):
    print(animal.make_sound())

# Create instances
dog = Dog()
cat = Cat()

# Demonstrate polymorphism
animal_sound(dog)  # Output: Bark
animal_sound(cat)  # Output: Meow
```

---

15. What is overloading? In Python, overloading refers to the ability to define multiple methods with the same name but different implementations. There are two main types of overloading in programming: method overloading and operator overloading.

Method Overloading

- Python does not support traditional method overloading directly, as seen in statically-typed languages like Java or C++. Instead, Python handles method calls based on the number and types of arguments passed. You can achieve a similar effect using default arguments or by manually checking the arguments inside the method.

Operator Overloading

- Operator overloading allows you to define the behavior of operators (such as +, −, *, etc.) for your custom objects. This is done by defining special methods in your class.

Common Special Methods for Operator Overloading:

- `__add__(self, other)` for `+`
- `__sub__(self, other)` for `-`
- `__mul__(self, other)` for `*`
- `__truediv__(self, other)` for `/`
- `__eq__(self, other)` for `==`
- `__lt__(self, other)` for `<`
- `__le__(self, other)` for `<=`

---

16. What does the `@property` decorator do? The @property decorator in Python is used to define methods in a class that can be accessed like attributes. It allows you to define a method and then access it as if it were a simple attribute, which makes the code cleaner and more intuitive. This is especially useful for encapsulation and managing the access to instance variables.

```python
class Employee:
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name

    @property
    def full_name(self):
        return f"{self._first_name} {self._last_name}"

    @full_name.setter
    def full_name(self, name):
        first_name, last_name = name.split(" ")
        self._first_name = first_name
        self._last_name = last_name

    @full_name.deleter
    def full_name(self):
        self._first_name = None
        self._last_name = None

# Create an Employee instance
emp = Employee("John", "Doe")

# Access the full_name property
print(emp.full_name)  # Output: John Doe

# Set the full_name property
emp.full_name = "Jane Smith"
print(emp.full_name)  # Output: Jane Smith

# Delete the full_name property
del emp.full_name
print(emp.full_name)  # Output: None None
```

---

15. How do you make attribution or method private?

```
class MyClass:
    def __init__(self, public_value, private_value):
        self.public_value = public_value
        self.__private_value = private_value

    def get_private_value(self):
        return self.__private_value

# Create an instance
obj = MyClass(10, 20)

# Access public attribute
print(obj.public_value)  # Output: 10

# Access private attribute directly (will raise AttributeError)
# print(obj.__private_value)  # Uncommenting this line will raise an
AttributeError

# Access private attribute through a method
print(obj.get_private_value())  # Output: 20
```

---

17. What is a metaclass?

In Python, a metaclass is a class of a class that defines how a class behaves. Just as a class defines how instances of the class behave, a metaclass defines how classes behave. Classes themselves are instances of metaclasses.

Metaclasses are a powerful and advanced feature of Python's object-oriented programming, providing a way to customize class creation and behavior.

Basics of Metaclasses:

1. Class Creation: When a class is created, Python looks for a **metaclass** attribute. If it's not found, it defaults to type, which is the built-in metaclass. The metaclass controls how the class is created and initialized.
2. Customizing Class Creation: By defining a custom metaclass, you can control the creation, initialization, and even the inheritance of classes.

```
# Define a metaclass
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        cls_obj = super().__new__(cls, name, bases, dct)
        return cls_obj

# Define a class using the metaclass
class MyClass(metaclass=MyMeta):
    def __init__(self, value):
        self.value = value
```

```
    def show_value(self):
        print(self.value)

# Create an instance of MyClass
obj = MyClass(42)
obj.show_value()
```

Quiz Assessment

References:

- https://youtu.be/u9ZqynGOXI8?si=_-DJ3eRa_tCaVyvP
- https://github.com/techwithtim/Python-Quiz/tree/main

```
def q1():
    print("hello" * 5)
```

answer: "hellohellohellohellohello"

```
def q2():
    x, y = 4, 5
    y, x = x, y
    print(x)
    print(y)
```

answer: 5 4

```
def q3():
    z = 9
    lst = [0] * z
    print(lst[:-1])
```

answer: [0,0,0,0,0,0,0,0,0]

```
def q4():
    def help(x):
        return x % 2 == 0

    lst = [i**2 for i in range(10)]
    lst = filter(help, lst)
    print(list(lst)[::2])
```

Showing Work:

```
lst = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
filter(help, lst) -> [0, 4, 16, 36, 64]
list(lst)[::2] -> [0, 16, 64]
```

answer: `[0, 16, 64]`

```
def q5():
    z = 0
    for i in range(1, 10):
        if (i + 1 // 2) % 7 == 0:
            break
        else:
            z += int(i % 2 == 0)
            print(z)
    else:
        print('end')
```

answer: `0, 1, 1, 2, 2, 3, 3`

Thougths: You can put an `else` on a for loop and also the order of operations matters here: `(i + 1 // 2) = (i + (1 // 2)) = (i + 0)`

```
def q6():
    import math

    for i in range(1, 100):
        if math.sqrt(i) == (i - 5)**2:
            print(i)
            break
    else:
        print('no')
```

```
1,  2, 3,  4,  5,  6,  7,  8,  9
1,  4, 9,  16, 25, 36, 49, 64, 81
16, 1, 16, ...
```

answer: `no`

```
def q7():
    class A:
        def __init__(self, x):
            self.x = x
```

```
        def __eq__(self, o):
            return self.x == o.x

    a = A(2)
    b = A(2)
    print(a == b)
    print(a is b)
    print(b is a)
    print(a is a)
    print(a == a)
```

answer:

```
True
False
False
True
True
```

```
def q8():
    class A:
        def __init__(self, x, y):
            self.x = x
            self.y = y

        def print(self):
            print(self.x, self.y)

    class B(A):
        def print(self):
            print(self.y, self.x)

    class C(B):
        def __init__(self, x, y, z):
            super().__init__(x, y)
            self.z = z

    d = A(2, 4)
    e = B(4, 5)
    g = C(3, 4, 7)
    g.x = g.z
    d.print()
    e.print()
    g.print()
```

answer:

```
2, 4
5, 4
4, 7
```

```python
def q9():

    def pythonx(z):
        def q(x, y):
            x = y + z + x
            print(x)

        return q

    for i in range(10):
        func = x(i)
        func(i, i-1)
```

```python
def q10():
    def d(f):
        def w(*args, **kwargs):
            r = f(*args, **kwargs)
            r += 1
            return r

        return w

    @d
    def a(x):
        return x + 1

    print(a(5))
```

```python
def q11():
    print(*list(map(lambda x: chr(ord(x) + 1), ["a", "b", "c"])))
```

```python
def q12():
    x = 0.1
    y = 0.10000000000000001
    print(x == y)
```

```python
def q13():
    x = [True, 1, "a", "b", "2"]
    print(any(x))
```

```python
def q14():
    x = ["a", 1, 2, 3, 4]
    y = z = x
    z[1] = 7
    y[3] = 2
    x[2] = 9
    print(x)
    print(y)
    print(z)
```

```python
def q15():
    print(1 == True)
    print("1" == 1)
```

```python
def q16():
    x = b'1001'
    y = b'1010'
    z = x + y
    print(z)
```

```python
def q17():
    x = 0b1001
    y = 0b1010
    z = x + y
    print(z)
```