Q1: What are Global, Protected and Private attributes?

Answer: In Python, the terms "Global," "Protected," and "Private" are not enforced access modifiers like in some other programming languages. However, certain naming conventions are commonly used to indicate the intended access level of attributes.

1. **Global attributes:** These are variables or attributes that can be accessed from anywhere within the program. They are typically declared outside of any class or function and can be used across different modules or classes.
2. **Protected attributes:** By convention, protected attributes are indicated by prefixing them with a single underscore (_). These attributes are intended to be accessible within the class and its subclasses. Although they can still be accessed from outside the class, it is considered a convention to treat them as non-public.
3. **Private attributes:** Private attributes are conventionally indicated by prefixing them with double underscores (__). These attributes are meant to be accessed only within the class itself. Python performs name mangling on these attributes to make them more difficult to access from outside the class. However, they can still be accessed using the mangled name if desired.

Remember that these access levels are more of a convention rather than strict enforcement. Python promotes a philosophy of "we're all consenting adults here," meaning that developers should respect naming conventions but ultimately have the freedom to access attributes as needed.

Q2: What is the use of 'self' in Python?

Answer: In Python, the self parameter is used within the definition of a class method to refer to the instance of that class. It is a convention to name this parameter self, although you can technically choose any valid variable name.
The self parameter allows instance methods to access and manipulate the attributes and methods of the class. When a method is called on an instance of a class, Python automatically passes the instance itself as the first argument to the method, which is why we need to include self as the first parameter in the method definition.
By using self, we can access instance variables and other methods of the class within the method implementation. It allows for proper encapsulation and enables each instance of a class to maintain its own state.

For example:

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def print_value(self):
        print(self.value)

obj = MyClass(42)
obj.print_value()  # Output: 42
```

In the example above, self is used to access the value attribute of the instance obj within the print_value method.

Q3: Are access specifiers used in Python?

Answer: In Python, access specifiers such as public, protected, and private are not enforced in the same way as in some other programming languages. Python follows a principle called "name mangling" for attributes and methods to achieve a limited form of encapsulation.
In Python, attributes or methods prefixed with a single underscore (_), such as _protected or _private, are considered conventionally "protected" or "internal" by convention. They indicate that the attribute or method should be treated as non-public, but it's ultimately up to the developer to respect this convention.
Similarly, attributes or methods prefixed with double underscores (__), such as __private, undergo name mangling, which changes their names to make them harder to access from outside the class. The mangled name includes the class name as a prefix to avoid conflicts with attributes of derived classes.
While these conventions provide a way to indicate intended access levels, they are not enforced by the language itself. In Python, there is an emphasis on readability, simplicity, and the "we're all consenting adults here" philosophy, which trusts developers to follow conventions and use attributes and methods responsibly.

Q4: Is it possible to call parent class without its instance creation?

Answer: No, it is not possible to call a parent class without creating an instance of it. In Python's object-oriented programming paradigm, you need to create an instance of a class in order to access its methods or attributes. The methods and attributes of a class are associated with instances of that class or its subclasses, and they require an instance to operate on specific data.
By creating an instance of a class, you establish an object context that allows you to call the methods or access the attributes of the class. Without an instance, there is no object context, and therefore, direct invocation of a parent class's methods or access to its attributes is not possible.

Q5: How is an empty class created in Python?

Answer: In Python, creating an empty class is quite simple. You can define an empty class by using the class keyword followed by the class name and an empty code block represented by a pair of parentheses:

```
class EmptyClass:
    pass
```

In the example above, EmptyClass is an empty class that doesn't have any attributes or methods defined within it. The pass statement is used as a placeholder to indicate that the class body is intentionally left empty. It is required because Python doesn't allow an empty code block.
Once you have defined an empty class, you can later add attributes, methods, or other members to it as needed. An empty class can serve as a foundation for further development or as a placeholder for future functionality.

Q6: How will you check if a class is a child of another class?

Answer: To check if a class is a child (subclass) of another class in Python, you can use the issubclass() function. By passing the class in question as the first argument and the potential parent class as the second argument to issubclass(), you can determine if the class is a subclass of the parent class.
The issubclass() function returns a boolean value: True if the class is indeed a subclass of the parent class, and False otherwise. This allows you to verify the inheritance relationship between classes without relying on code examples.

Q7: What is docstring in Python?

Answer: In In Python, a docstring is a string literal placed at the beginning of a module, function, class, or method definition. It serves as a documentation string that describes what the code does and provides information on how to use it. Docstrings are enclosed in triple quotes (either single or double quotes) and can span multiple lines.
Docstrings are used to provide documentation for code elements and serve as a form of inline documentation. They can be accessed through the __doc__ attribute of the respective object.
Docstrings are important for code readability, maintainability, and documentation generation tools. They help developers understand the purpose and usage of the code without inspecting the implementation details.

Q8: Is Python Object-Oriented or Functional Programming?

Answer: Python is a versatile programming language that supports multiple programming paradigms, including object-oriented programming (OOP) and functional programming (FP). It is not strictly limited to a single paradigm.
Python's OOP features allow you to define classes, create objects, and utilize concepts such as encapsulation, inheritance, and polymorphism. You can create and manipulate objects with their own attributes and methods, enabling you to build modular and reusable code structures.
On the other hand, Python also incorporates functional programming principles. It treats functions as first-class objects, allowing you to pass functions as arguments, return them from other functions, and store them in data structures. Python provides functional programming constructs such as lambda functions, map, filter, and reduce, which support functional programming concepts.
Python's support for both OOP and FP makes it a flexible language that can be used for various programming styles, depending on the requirements and preferences of the developer. You can choose to write code that is primarily object-oriented, primarily functional, or a combination of both paradigms.

Q9: What does an object() do?

Answer: In Python, the object() function returns a new empty object of the base class object. This function can be called with no arguments, and it creates a new instance of the object class.

The object class is the base class for all other classes in Python. It provides a set of default behaviors and methods that are inherited by all other classes. By creating an instance of object using object(), you can obtain a basic object with no additional attributes or methods specific to a particular class.

However, in most cases, you would not explicitly create instances of object() since it doesn't provide any additional functionality beyond what is already available by default in Python. Instances of object are typically used implicitly when creating instances of other classes, as all classes ultimately inherit from object by default.

Q10: What is the purpose of the super function in inheritance, and how is it used?

Answer: The super() function in Python is used to refer to the parent class or superclass in the context of inheritance. Its purpose is to enable method overriding while preserving the ability to call the overridden method in the parent class.

By using super(), you can access and invoke methods or attributes defined in the parent class from within the child class. This allows you to extend or customize the behavior of the parent class's methods in the child class without completely overriding them.

To use super(), you typically call it within a method of the child class and provide it with two arguments: the child class itself (or the class instance) and the specific instance of the child class. This ensures that the method is called on the appropriate instance and provides access to the parent class's methods.

In summary, the super() function facilitates method overriding and provides a convenient way to invoke parent class methods within the context of inheritance.

Q11: What is data abstraction?

Answer: In Data abstraction is a fundamental concept in object-oriented programming that focuses on the separation of the external behavior of an object from its internal implementation details. It allows the creation of abstract data types where the internal workings of the data are hidden, and only the essential operations or behaviors are exposed to the user.

Abstraction helps in managing complexity by providing a simplified and high-level view of the data or objects. It allows users to interact with objects using a well-defined interface without worrying about how the underlying implementation is structured or functioning.

In Python, data abstraction is achieved through the use of classes and objects. Classes define the structure and behavior of objects, while objects represent instances of those classes. By defining appropriate methods and attributes, classes can expose only the necessary functionality to interact with objects, hiding the implementation details.

By employing data abstraction, code can be modular, maintainable, and easier to understand. It promotes encapsulation, information hiding, and separation of concerns, leading to more robust and reusable code.