1. Describe how interfaces support Python code design.

The interfaces of an instrumentation system are where the software and the outside environment collide. The input to determine whether a switch is open or closed, the voltage from a temperature sensor, or pulses from a magnetic sensor on a rotating shaft could all serve as simple interfaces. A data communications channel for interacting with a self-contained instrument or controller can also be considered an interface. In the case of an Ethernet port, it can even be a different computer system (Hughes, 2010).

According to PythonGuides.com (2020), the implementing class must provide a set of method signatures known as an interface. There are abstract methods in an interface. Since there is no implementation, the declaration will only apply to the abstract forms. 'Interface.Interface', the parent interface for all interfaces, is a subclass of 'interface' when defining an interface in Python. The classes that will inherit the interface will carry out the implementations. Python's interfaces differ from those of Java, C#, or C++. Writing ordered code involves implementing an interface.

In software engineering, interfaces are crucial. Updates and changes to the code base get harder to maintain as a programme expands. For instance, Python handles interfaces differently than most other languages and allows for a wide range of design complexity (Murphy, 2020).

Furthermore, Murphy (2020) explains that an interface is a general guide for creating classes. Interfaces define methods much like classes do. These methods are abstract in contrast to types. A form that the interface specifies is an abstract form. It needs to put the strategies into practice. Classes accomplish this, after which they implement the interface and provide the abstract interface methods with concrete meaning.

2. Write a Python program which applies interfaces.

Here is an illustration of how Python's abstract base classes (ABCs) can be used to implement interfaces based on examples from Python documentation (N.D.):

```
from abc import ABC, abstractmethod

# Define the interface
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

    @abstractmethod
    def calculate_perimeter(self):
        pass

# Create classes that follow the interface
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    def calculate_perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius ** 2

    def calculate_perimeter(self):
        return 2 * 3.14 * self.radius

# Create objects, then use them in different ways.
rectangle = Rectangle(4, 6)
circle = Circle(3)
shapes = [rectangle, circle]

for shape in shapes:
    print("Area:", shape.calculate_area())
    print("Perimeter:", shape.calculate_perimeter())
    print()
```

This program uses an abstract base class (ABC) from the ABC module to create an interface called Shape. The Shape interface has two abstract methods: Calculate_area and Calculate_Perimeter. To implement the Shape interface, any class must provide concrete implementations of these methods.

The next step is to create a Rectangle and Circle class that inherits from Shape and implement the required methods. These classes align with the Shape interface, providing the necessary functionalities to calculate the area and perimeter of their respective shapes.

By transforming Circles and Rectangles into objects and including them in the shapes list, we can implement the calculate_area and calculate_perimeter methods for each shape iteratively. Since both objects comply with the Shape interface, we can handle them in the same way, regardless of belonging to different classes.

Incorporating interfaces promotes a code structure that fosters abstraction, modularity, contract enforcement, and polymorphism. This helps improve the quality of the code and makes it easier to manage.

3. Define the metrics by which an object-oriented program can be assessed.

Object-oriented metrics are measurements used to evaluate the complexity, quality, and efficacy of software engineering methods, people, and products. These metrics are utilised to assess developers' productivity and skill level and the design, coding, and testing of object-oriented software systems (itskawal2000, 2021).

When assessing an object-oriented program, quality, maintainability, efficiency, and adherence to best practices. There are established notions of software metrics, and many metrics are available to measure product quality. As object-oriented analysis and design approaches become more prominent, it is important to start exploring object-oriented metrics for software quality (Rosenberg & Hyatt,1997).

Numerous metrics have been proposed to assess aspects of object-oriented software, including size, inheritance, cohesion, and coupling (Dagpinar & Jahnke, 2003).

By evaluating the critical research chosen on the systematic mapping, Saraiva (2014) claimed that all metrics were grouped based on the internal characteristics of the software connected to six points: Size, Inheritance, Coupling, Cohesion, Complexity, and Software Architecture Constraints. Based on the metric description found in the preliminary study that indicated it, the measurements were then linked to an internal software property.

Turan & Tanriover (2018) stated that it is tough to test complex code, and it is also challenging to keep up. Developers must follow the boundary values of metrics when writing code to ensure it is well-written, comprehensible, and maintainable. For instance, the following are some typical metrics for judging object-oriented programmes:

- The Maintainability Index (MI) is a valuable indicator for assessing how simple it is to maintain a programme. It considers several variables, including complexity, coupling, cohesiveness, and code duplication, to produce a numerical score. This rating aids in determining the program's capacity for alterations, bug fixes, and future improvements.

- The Cyclomatic Complexity (CC) metric is used to assess the structural complexity of code. It is calculated by counting the code pathways that make up a program's flow. The number of controls affects the possible CC value flows based on different inputs. Code with complex control flows is more challenging and requires additional testing for appropriate code coverage. A probable code issue is indicated by high cyclomatic complexity.

- The "Depth of Inheritance" is the number of class definitions that reach the base of the class hierarchy. Tracing the purposes or redefinitions of specific methods and fields on a deeper scale could be more challenging. The complexity of the code may increase as the number of types in an inheritance hierarchy increase.

On the other hand, a high inheritance depth may also indicate substantial code reuse. It can be challenging to decide what an ideal depth of inheritance is.

- Class Coupling is a programme design evaluation criterion, which analyses the relationship to various classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields declared on external styles and attribute decorating. High cohesion and low coupling for the types and methods are characteristics of well-designed software. High coupling denotes a complicated design that is difficult to reuse and maintain due to its extensive interdependencies on other types. A class will have zero class coupling if it does not reference other classes. The class coupling will be increased by referencing different courses in our class, such as developing complex-type properties.

Thanks to the SOLID design principles, we can construct flexible, reusable, and maintainable software. In the acronym SOLID, each letter stands for a distinct tenet (Abba, 2022). The acronym's notes are defined as follows:

- S: Single responsibility principle.
- O: Open–closed principle.
- L: Liskov substitution principle.
- I: Interface segregation principle.
- D: Dependency inversion principle.

According to Abba (2022), the program's compliance with core object-oriented design concepts such as encapsulation, inheritance, polymorphism, and SOLID principles is evaluated by compliance with design principles. These rules can be broken, resulting in difficult-to-understand, difficult-to-maintain, and difficult-to-extend code.

As Forrest (2023) stated, metrics are essential in gaining valuable insights into object-oriented programs' quality, maintainability, and efficiency. It is crucial to approach the selection, creation, and utilisation of software metrics systematically and thoroughly as they provide various benefits, including:

- Early detection of potential issues in the program.
- Monitoring the program's progress and identifying areas for improvement.
- Comparing different programs and identifying any modifications made.
- Defending the program's development and maintenance costs.

References:

Hughes, J.M. (2010). *Real World Instrumentation with Python: Automated Data Acquisition and Control Systems*. " O'Reilly Media, Inc.".

PythonGuides.com (2020). *Introduction To Python Interface - Python Guides*. [online] Available at: https://pythonguides.com/python-interface/#:~:text=An%20interface%20is%20a%20collection%20of%20method%20signatures [Accessed 11 Jun. 2023].

Murphy, W. (2020). *Implementing an Interface in Python – Real Python*. [online] realpython.com. Available at: https://realpython.com/python-interface/.

Python documentation. (N.D.). *abc — Abstract Base Classes*. [online] Available at: https://docs.python.org/3/library/abc.html#:~:text=This%20module%20provides%20the%20infrastructure [Accessed 12 Jun. 2023].

itskawal2000 (2021). *Object Oriented Metrices in Software Engineering*. [online] Available at: https://www.geeksforgeeks.org/object-oriented-metrices-in-software-engineering/.

Rosenberg, L.H. and Hyatt, L.E. (1997). Software quality metrics for object-oriented environments. *Crosstalk journal*, *10*(4), pp.1-6.

Dagpinar, M. & Jahnke, J.H. (2003). Predicting maintainability with object-oriented metrics-an empirical comparison. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* (pp. 155-155). IEEE Computer Society.

Saraiva, J. de A.G. (2014). *Classifying metrics for assessing object-oriented software maintainability: a family of metrics' catalogs*. [online] repositorio.ufpe.br. Available at: https://repositorio.ufpe.br/handle/123456789/12152 [Accessed 16 Jun. 2023].

Turan, O. & TANRIÖVER, Ö.Ö. (2018). An Experimental Evaluation of the Effect of SOLID Principles to Microsoft VS Code Metrics. *AJIT-e: Bilişim Teknolojileri Online Dergisi*, *9*(34), pp.7-24.

Abba, I. V. (2022). *SOLID Definition – the SOLID Principles of Object-Oriented Design Explained*. [online] Available at: https://www.freecodecamp.org/news/solid-principles-single-responsibility-principle-explained/.

Forrest, G. (2011). *The Importance of Implementing Effective Metrics*. [online] isixsigma.com. Available at: https://www.isixsigma.com/metrics-methodology/importance-implementing-effective-metrics/.