## 1. Create a Python program which uses a constructor and abstract class.

When an object is formed, a constructor, a specific method, is invoked. The characteristics of the object are initialised using it. The constructor method in Python is known as __init__() (GeeksforGeeks, 2019). For instance, a class that cannot be instantiated is considered abstract. It serves as an introductory class for other instantiable classes. A group of subclasses' shared interfaces is defined using abstract classes. Their use can facilitate inheritance and code reuse.

The brand-new ABC support framework is implemented in the pure Python standard library module abc. It includes decorators @abstractmethod and @abstractproperty and the metaclass ABCMeta (peps.python.org, N.D.).

The @abstractmethod decorator, which can be used to declare abstract methods, is likewise defined by the ABC module. It is impossible to instantiate a class with at least one method specified with this decorator, but it has yet to be overridden. Such methods can be called directly or indirectly from the overriding method in the subclass (docs.python.org, N.D.).

Here is an example of a Python program that demonstrates how to use an abstract class and constructor, based on examples from John, G. (2019):

```
from abc import ABC, abstractmethod

class Shape(ABC):
    """An abstract class representing a shape."""

    def __init__(self, name: str):
        """Initializes a shape."""
        self.name = name

    @abstractmethod
    def area(self) -> float:
        """Returns the area of the shape."""
        pass
```

```
class Circle(Shape):
    """A class representing a circle."""

    def __init__(self, name: str, radius: float):
        """Initializes a circle."""
        super().__init__(name)
        self.radius = radius

    def area(self) -> float:
        """Returns the area of the circle."""
        return 3.14 * self.radius ** 2

class Square(Shape):
    """A class representing a square."""

    def __init__(self, color, width, height):
        super().__init__(color)
        self.width = width
        self.height = height

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

circle = Circle("blue", 3)
print(f"Circle Area: {circle.area()}")

square = Square("red", 4, 5)
print(f"Square Area: {square.area()}")
```

This programme produces two shapes: a square and a circle. The areas of the shapes are then printed.

The class Shape is amorphous. It cannot, therefore, be instantiated directly. It must be subclassed instead. Subclasses of the Shape class include the Circle and Square classes. Both of them implement the abstract method area(). The shape's size is returned by this method.

The main() function produces two shapes: a circle and a square. The areas of the shapes are then printed.

## 2. Describe the concepts of polymorphism, aggregation and composition.

Object-oriented programming (OOP) ideas that explain the connections between objects include polymorphism, aggregation, and composition (Kanjilal, 2018).

Polymorphism is the ability of an object to take on different shapes. In object-oriented programming, polymorphism is achieved by overriding and inheritance. Thanks to inheritance, one object can take on the characteristics and functions of another object (Duggal, N. 2023). A subclass may override a method declared in a superclass to implement the method in the subclass's fashion. A Dog class, for instance, might be descended from a Pet class. Because of this, a Dog object may be used in any situation where a Pet object is anticipated.

A relationship between two objects is called an aggregattion in which one complete item contains another object's component. The portion object may exist independently from the whole thing (Azzam, 2020). For instance, an Engine object and a Car object may be aggregated. Although the Engine object can exist without the Car object, it is useless without it.

Since the component parts of an item cannot exist separately from the whole, the composition is a more potent form of aggregation (Visual-paradigm.com, 2019). For instance, an object such as a house might be made up of rooms. Without the House object, the Room objects are not possible.

## 3. Name variables according to the namespace within which they exist.

Lowercase letters are used to spell out the names of modules, variables, functions, and methods. When one of these names contains one or more embedded names, with the exception of modules, the embedded names are capitalised. Classes have names that adhere to the same rules but start with a capital letter. All letters in a variable that refers to a constant are capitalised, and any names that are embedded are separated by an underscore (Lambert, K. A & Osborne, M. 2010). In the table below are the examples of Python Naming Conventions:

| **Examples of Python Naming Conventions** | **Examples of Python Naming Conventions** |
|---|---|
| Type of Name Examples | Type of Name Examples |
| Variable salary, hoursWorked, isAbsent | Variable salary, hoursWorked, isAbsent |
| Constant ABSOLUTE_ZERO, INTEREST_RATE | Constant ABSOLUTE_ZERO, INTEREST_RATE |
| Function or method printResults, cubeRoot, isEmpty | Function or method printResults, cubeRoot, isEmpty |

Making your code readable relies heavily on the proper naming of variables. There is only one rule for naming variables: Make variables that explain their purpose and maintain a recurring theme across your code (curc.readthedocs.io, N.D.).

It's typical to employ a naming convention that uses prefixes or suffixes to denote the namespace when naming variables according to the namespace in which they exist. This promotes clarity and prevents naming conflicts among several namespaces (Coghlan et. Al. 2001).

A namespace is a collection of symbolic names that are currently defined, along with details about the objects to which each name refers (Sturtz, J. 2020).

Built-in namespaces, global namespaces, local namespaces, and enclosing namespaces are the four different types of namespaces available in Python (Raj, 2021). For instance, here are these definitions:

The programme or module level is where global namespaces are defined. It includes the names of things defined in a module or the main programme. A global namespace is established when a programme begins and remains in place until the Python interpreter ends the application. The example below clarifies the idea of a global namespace:

```
myNum1 = 10
```

```
myNum2 = 10

def add(num1, num2):
        temp = num1 + num2
        return temp
```

A block of code or a function may be defined inside of another block of code or function. In these circumstances, the inner function or code block has access to the namespace of the outside function or code block. As a result, the inner function or code block is **enclosed** within the outer namespace. This is seen in the example that follows:

```
myNum1 = 10
myNum2 = 10

def add(num1, num2):
        temp = num1 + num2

        def print_sum():
        print(temp) return temp
```

A **local namespace** is created when defining a class, function, loop, or any other type of code block. The names assigned inside that function or block of code can only be accessed inside; they cannot be accessed outside of it. When a function or block of code is finished, the **local namespace** is formed and then deleted. Here is an example:

```
myNum1 = 10
myNum2 = 10


def add(num1, num2):
   temp = num1 + num2
   return temp
```

The names of built-in objects and functions are stored in a built-in namespace. It is produced when the Python interpreter is launched, remains active as long as the interpreter is open, and is deleted when the interpreter is shut down. Names of built-in data types, exceptions, and functions like print() and input() are all included in this list. The following describes how we can access every name in the built-in namespace:

```
builtin_names = dir(__builtins__)
for name in builtin_names:
    print(name)
```

**References:**

GeeksforGeeks. (2019). *__init__ in Python*. [online] Available at: https://www.geeksforgeeks.org/__init__-in-python/.

peps.python.org. (N.D.). *PEP 3119 – Introducing Abstract Base Classes | peps.python.org*. [online] Available at: https://peps.python.org/pep-3119/#the-abc-module-an-abc-support-framework [Accessed 11 Jun. 2023].

docs.python.org. (N.D.). *abc — Abstract Base Classes — Python 3.9.1 documentation*. [online] Available at: https://docs.python.org/3/library/abc.html.

John, G. (2019). *Abstract Base Classes in Python (abc)*. [online] Available at: https://www.tutorialspoint.com/abstract-base-classes-in-python-abc [Accessed 11 Jun. 2023].

Kanjilal, J. (2018). *Association, aggregation, and composition in OOP explained*. [online] InfoWorld. Available at: https://www.infoworld.com/article/3029325/exploring-association-aggregation-and-composition-in-oop.html.

Duggal, N. (2023). *Learn Polymorphism in Python with Examples | Simplilearn*. [online] Available at: https://www.simplilearn.com/polymorphism-in-python-article#what_is_polymorphism_in_python [Accessed 21 Jun. 2023].

Azzam, A. (2020). *Aggregation vs. Composition in Object Oriented Programming*. [online] Medium. Available at: https://medium.com/swlh/aggregation-vs-composition-in-object-oriented-programming-3fa4fd471a9f.

Visual-paradigm.com. (2019). *UML Association vs Aggregation vs Composition*. [online] Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/.

Lambert, K.A. and Osborne, M. (2010). *Fundamentals of python: from first programs through data structures*. Cengage Learning.

curc.readthedocs.io. (N.D.). *Coding best practices — Research Computing University of Colorado Boulder documentation*. [online] Available at: https://curc.readthedocs.io/en/latest/programming/coding-best-practices.html.

Van Rossum, G., Warsaw, B. & Coghlan, N. (2001). *PEP 8 – Style Guide for Python Code* | *peps.python.org*. [online] peps.python.org. Available at: https://peps.python.org/pep-0008/.

Sturtz, J. (2020). *Namespaces and Scope in Python – Real Python*. [online] realpython.com. Available at: https://realpython.com/python-namespaces-scope/.

Sturtz, J. (2021). *What is Namespace in Python?* [online] Available at: https://www.pythonforbeginners.com/basics/what-is-namespace-in-python#:~:text=What%20is%20a%20local%20namespace [Accessed 11 Jun. 2023].