

Python Institute

PCPP-32-101 Exam

PCPP1 - Certified Professional in Python Programming 1

**Questions & Answers
Demo**

Version: 4.0

Question: 1

Select the true statement about composition

- A. Composition extends a class's capabilities by adding new components and modifying the existing ones.
- B. Composition allows a class to be projected as a container of different classes
- C. Composition is a concept that promotes code reusability while inheritance promotes encapsulation.
- D. Composition is based on the has a relation: so it cannot be used together with inheritance.

Answer: B

Explanation:

Composition is an object-oriented design concept that models a has-a relationship. In composition, a class known as composite contains an object of another class known as component. In other words, a composite class has a component of another class¹.

1. Composition allows a class to be projected as a container of different classes.

Composition is a concept in Python that allows for building complex objects out of simpler objects, by aggregating one or more objects of another class as attributes. The objects that are aggregated are generally considered to be parts of the whole object, and the containing object is often viewed as a container for the smaller objects.

In composition, objects are combined in a way that allows for greater flexibility and modifiability than what inheritance can offer. With composition, it is possible to create new objects by combining existing objects, by using a container object to host other objects. By contrast, with inheritance, new objects extend the behavior of their parent classes, and are limited by that inheritance hierarchy.

Reference:

Official Python documentation on

Composition: <https://docs.python.org/3/tutorial/classes.html#composition>

GeeksforGeeks article on Composition vs Inheritance: <https://www.geeksforgeeks.org/composition-vs-inheritance-python/>

Real Python article on Composition and Inheritance: <https://realpython.com/inheritance-composition-python/>

Question: 2

Analyze the following snippet and select the statement that best describes it.

```
class OwnMath:
    pass

def calculate_value(numerator, denominator):
    try:
        value = numerator / denominator
    except ZeroDivisionError as e:
        raise OwnMath from e
    return value

calculate_value(4, 0)
```

- A. The code is an example of implicitly chained exceptions.
- B. The code is erroneous as the OwnMath class does not inherit from any Exception type class
- C. The code is fine and the script execution is not interrupted by any exception.
- D. The code is an example of explicitly chained exceptions.

Answer: D

Explanation:

In the given code snippet, an instance of OwnMath exception is raised with an explicitly specified `__cause__` attribute that refers to the original exception (ZeroDivisionError). This is an example of explicitly chaining exceptions in Python.

Question: 3

Analyze the following snippet and choose the best statement that describes it.

```
class Sword:
    var1 = 'weapon'

    def __init__(self):
        self.name = 'Excalibur'
```

- A. self.name is the name of a class variable.
- B. var1 is the name of a global variable
- C. Excalibur is the value passed to an instance variable

D. Weapon is the value passed to an instance variable

Answer: C

Explanation:

The correct answer is C. Excalibur is the value passed to an instance variable. In the given code snippet, self.name is an instance variable of the Sword class. When an instance of the Sword class is created with var1 = Sword('Excalibur'), the value 'Excalibur' is passed as an argument to the __init__ method and assigned to the name instance variable of the var1 object.

The code defines a class called Sword with an __init__ method that takes one parameter name. When a new instance of the Sword class is created with var1 = Sword('Excalibur'), the value of the 'Excalibur' string is passed as an argument to the __init__ method, and assigned to the self.name instance variable of the var1 object.

Reference:

Official Python documentation on Classes: <https://docs.python.org/3/tutorial/classes.html>

Question: 4

The following snippet represents one of the OOP pillars Which one is that?

```
class A:
    def run(self):
        print("A is running")

class B:
    def fly(self):
        print("B is flying")

class C:
    def run(self):
        print("C is running")

for element in A(), B(), C():
    element.run()
```

- A. Serialization
- B. Inheritance
- C. Encapsulation
- D. Polymorphism

Answer: C

Explanation:

The given code snippet demonstrates the concept of encapsulation in object-oriented programming. Encapsulation refers to the practice of keeping the internal state and behavior of an object hidden from the outside world and providing a public interface for interacting with the object. In the given code snippet, the `__init__` and `get_balance` methods provide a public interface for interacting with instances of the `BankAccount` class, while the `__balance` attribute is kept hidden from the outside world by using a double underscore prefix.

Question: 5

Analyze the following function and choose the statement that best describes it.

```
def my_decorator(coating):
    def level1_wrapper(my_function):
        def level2_wrapper(*args):
            our_function(*args)

        return level2_wrapper

    return level1_wrapper
```

- A. It is an example of a decorator that accepts its own arguments.
- B. It is an example of decorator stacking.
- C. It is an example of a decorator that can trigger an infinite recursion.
- D. The function is erroneous.

Answer: A

Explanation:

In the given code snippet, the `repeat` function is a decorator that takes an argument `num_times` specifying the number of times the decorated function should be called. The `repeat` function returns an inner function `wrapper_repeat` that takes a function `func` as an argument and returns another inner function `wrapper` that calls `func` `num_times` times.

The provided code snippet represents an example of a decorator that accepts its own arguments. The `@decorator_function` syntax is used to apply the `decorator_function` to the `some_function` function. The `decorator_function` takes an argument `arg1` and defines an inner function `wrapper_function` that takes the original function `func` as its argument. The `wrapper_function` then returns the result of calling `func`, along with the `arg1` argument passed to the `decorator_function`.

Here is an example of how to use this decorator with `some_function`:

```
@decorator_function("argument 1")
```

```
def some_function():  
    return "Hello world"
```

When `some_function` is called, it will first be passed as an argument to the `decorator_function`. The `decorator_function` then adds the string "argument 1" to the result of calling `some_function()` and returns the resulting string. In this case, the final output would be "Hello world argument 1".

Reference:

Official Python documentation on Decorators: <https://docs.python.org/3/glossary.html#term-decorator>