

Solutions

4.1 Classes and Instances

Reference example

```
class Counter:
    def __init__(self, start):
        self.value = start
    def increment(self):
        self.value += 1
```

```
a = Counter(2)
b = Counter(10)
a.increment()
print(a.value, b.value)
```

4.1.1 - Explain

What role does `__init__` play when `Counter(5)` is evaluated?

It initializes the new instance state using the supplied argument. It should not return an alternate object or a non-None result.

4.1.2 - Write code

Create a `Box` class whose constructor stores width and whose `area` method returns width squared. Show `Box(4).area()`.

Store instance state through `self`.

```
class Box:
    def __init__(self, width):
        self.width = width
    def area(self):
        return self.width ** 2

print(Box(4).area())
```

Expected output

16

Solutions

4.2 Instance Versus Class Attributes

Reference example

```
class Bag:
    def __init__(self):
        self.items = []

a = Bag()
b = Bag()
a.items.append("book")
print(a.items)
print(b.items)
```

4.2.1 - Trace

Determine the exact output and explain the state changes.

```
class Item:
    category = "tool"

a = Item()
b = Item()
a.category = "book"
print(a.category, b.category)
```

Output

book tool

The instance assignment shadows the class attribute for a only.

4.2.2 - Explain

Why can items = [] in the class body cause unexpected sharing between instances?

The list is stored on the class and may be reached by multiple instances. Mutating it does not create an independent list for each instance.

Solutions

4.3 Methods and Encapsulation

Reference example

```
class Wallet:
    def __init__(self):
        self._balance = 0
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
    def balance(self):
        return self._balance

w = Wallet()
w.deposit(8)
w.deposit(-2)
print(w.balance())
```

4.3.1 - Explain

Why is directly changing `_balance` weaker than using `deposit` for normal client code?

It bypasses the method validation and couples the caller to internal representation. Respect the public interface even when the language permits direct access.

4.3.2 - Write code

Write a `Celsius` class with a method returning Fahrenheit, and show the result for 0 degrees Celsius.

Use the conversion inside an instance method.

```
class Celsius:
    def __init__(self, value):
        self.value = value
    def fahrenheit(self):
        return self.value * 9 / 5 + 32

print(Celsius(0).fahrenheit())
```

Expected output

32.0

Solutions

4.4 Special Methods

Reference example

```
class Playlist:
    def __init__(self, songs):
        self.songs = list(songs)
    def __len__(self):
        return len(self.songs)
    def __str__(self):
        return f"Playlist({len(self)} songs)"

p = Playlist(["A", "B"])
print(len(p))
print(p)
```

4.4.1 - Explain

What return type should `__len__` produce, and what should `__str__` return?

`__len__` must supply an appropriate nonnegative integer length. `__str__` returns a string representation, not a printed side effect.

4.4.2 - Write code

Create a `Label` class whose string representation is its stored text, then print `Label("ready")`.

Return the text from `__str__`.

```
class Label:
    def __init__(self, text):
        self.text = text
    def __str__(self):
        return self.text

print(Label("ready"))
```

Expected output

ready