quiz 1

1. a clear statement of the original problem;
2. a different code example (or context) than that originally given in the quiz — the example must be similar in spirit to the original question, but of your own creation, and get to the point of the original quiz question;
3. printed output that clearly demonstrates any output and/or related topics (in doing so, you should include more print statements than were present in the quiz);
4. separate written (prose) discussion in which you clearly discuss:
   • why your original answer was incorrect;
   • how your new example, with printed output, demonstrates understanding of the concept(s).

Questions Corrected: 1 (0/2), 2 (2/3), 3, (2/3), 4 (4/6), 5 (0/2)

question 1

the original problem asks to give a higher level definition of what an int is:

in an intro course, we say that int is a function used to convert a string to its integer equivalent. briefly discuss why thinking of int as a function is inaccurate, presenting a better description of what is taking place.

```
five = int("5")
```

rewritten answer

int is a class or type constructor not just a function. this is because it constructs an integer object by calling the int class to make a new integer object. this means that int(”5”) is instantiating the integer object instead of just converting it to a string.

what went wrong

i didnt give a clear enough explanation on how int works more as a class that creates an int object.

question 2

the original problem asks to explain the difference between a class and an object with an example:

rewritten answer

a class is a blueprint that defines attributes and methods — variables that hold state and actions the class can do

```
class Car:
    def __init__(self, brand):
        self.brand = brand #attribute

    def honk(self):
        print("beep boop")
```

this is an example of a class describing what a car is and what it can do

on the other hand, objects are a specific instance of a class that is created during runtime — holds actual data

```
my_car = Car("honda")
my_car.honk()
print(my_car.brand)
```

this is an example of an object

what went wrong

the answer before was mostly right but lacked an explanation of how objects can still contain methods. the main difference between the two is that the object uses the class to store data while the class is used as a blueprint.

my new example shows a class, Car, that can save data, brand, and run a method within the class Car. by creating an object, my_car, it is an example of how the class is the blueprint for the object to be made.

question 3

the original problem asked to analyse the code below and explain some print statements:

consider the code implemented below

a) explain why “[1, 2, 3, 1]” is printed

b) should items on lines 1 and 2 instead be called data? why or why not?

```
def func(items: list[int]) → None:
    items[-1] = items[0]

def main() → None:
    data = [1, 2, 3, 4]
    func(data)
    print(data)

main()
```

rewritten answer

a)

initial list: data = [1, 2, 3, 4]

inside func(items):

items[-1] refers to the last element (index 4)

items[0] refers to the first element (index 1)

assignment in line 2 changes the last element to 1

result after func(data):

the original list data is now [1, 2, 3, 1]

this is because lists are mutable and func() changes the list

output of print(data): [1, 2, 3, 1]

b)

items is probably the better name here

it is a generic parameter name:

- items is a parameter of func() which means that it can accept any list — not only data
- naming it data would mean that it only works with the data variable which can be misleading

scope clarity:

- data is a variable in main() but items is local in func()

what went wrong

i didnt talk about how lists are mutable and this means that they can be changed within functions to affect variables in main. the list is modified in place because lists are mutable and passed by reference.

question 4

consider the class implementations below:

```
class Upper:
    __slots__ = ("_a", "_b")
    def __init__(self, one: int, two: int) → None:
        self._a = one
        self._b = two

    def setValues(self, one: int, two: int) → None:
        self._a = one + one
        self._b = two + two

    def __str__(self) → str: return f"{self._a}:{self._b}"

class Lower(Upper):
    __slots__ - ("_c", "_d")

    def __init__(self, one: int, two: int, thr: int) → None:
        super().__init__(one, two)
        self._c = thr
        self._d = thr + thr

    def setValues(self, one: int, two: int, thr: int) → None:
        self._a = one * one
        self._b = two * two
        self._c = thr * thr
        self._d = thr * thr * thr
```

a. what instance variables and methods would an Upper object have?

b. what instance variables and methods would a Lower object have?

c. after executing the code below, what would the values of the instance variables inside lower be? explain why.

d. when executing the code below, what get printed? explain why.

```
lower = Lower(11, 22, 33)
lower.setValues(1, 2, 3)
print(lower)
```

rewritten answers

a)

correct

b)

correct

c)

sets _a = 11, _b = 22

then sets specific variables

_c = 33 and _d = 33 + 33 = 66

results in _a = 11, _b = 22, _c =33, _d = 66

then lower.setValues(1, 2, 3) changes it

d)

setValues method is overwritten in Lower:

_a = 1 ∗ 1 = 1

_b = 2 ∗ 2 = 4

_c = 3 ∗ 3 = 9

_d = 3 ∗ 3 ∗ 3 = 27

print(lower)

calls inherited __str__ from Upper which only formats _a and _b

output: “1:4”

question 5

question about polymorphism and how it functions:

rewritten answer

yes, because polymorphism allows other class methods to be reused and repurposed — whether the object has set parameters regardless of the type.

question 6

question about creating a class

— can redo after i get pg 2 of the quiz