How to Solve Tracing

To Do(s):

1. Always start from the driver code.
2. Whenever a new object/instance is created, create a separate table. If multiple instances of the same class are created, these too will have separate tables.
3. Whenever an object of a class is created, the constructor of that class must be executed. For example, as we executed the following code:
   q = Q5([5])
   The constructor of the Q5 class must be executed. Therefore, the following lines are automatically executed:

def __init__ (self, z):
self.sum = 1
self.x = 2
self.y = 3
self.z = z

4. Each variable will have its own column.
5. Instance variables, local variables of the methods will be inside the same table.
6. To distinguish between instance variable and class variables with same names, write self. In front of the instance variable.
7. If a method is called, jump to that method. When the method is executed, jump back to the point where that method was called. Depending on the nature of the method (having return statement or not), you may have to put a specific value received from the method call. For example:

methodA(a,b):
    return a+b
 	
x= (100 - methodA (5,6))/2
print(methodA(10,20))
	methodA(a,b):
	    print(a,b)
	
	methodA (5,6)
	z =5

Here, we can assume that methodA returns something which will return something that will replace methodA (5,6) and then the arithmetic operation can be performed. In this case, 11 will be returned and we shall perform the following operation: x= (100 – 11)/2 Same goes for the next line. 30 is returned and printed.	Here, methodA does not need to return anything. Therefore, after finishing methodA execution, we jump back to the point where the method had been called from. Now we can go to the next line and execute z= 5.

8. Whenever you create an object or declare a list/dictionary, along with the value, also specify a made-up location of your choosing. This will help you keep track.

```
Class X:
    def __init__(self, x, y, z=[ ]):
          self.z = z

x1= X("Hello", 100, [10,20])

x2= X("Yo", 10)
```

Trace Table:

x1 (London)		
x	y	z or self.z (Las Vegas)
Hello	100	[10,20]

x2 (Canada)		
x	y	z or self.z (Brooklyn)
Yo	10	[ ]

Things to Watch Out For

1. **Method Overloading:**
Depending on the argument and parameter variance, we may need to take different branches.

```
def methodB(self, mg0, mg1, mg2 = None):
        if mg2 == None:
            x, y = 5, 6
            y = self.sum + mg1.content
            self.y = y + mg1.content
            x = self.x + 7 +mg1.content
            self.sum = self.sum + x + y
            self.x = mg1.content + x +8
            mg0[0]+=2
            print(x, " ", y," ", self.sum)
            return y
        else:
            x = 1
            self.y += mg2[0].content
            mg2[0].content = self.y + mg1.content
            x = x + 4 + mg1.content
            self.sum += x + self.y
            mg1.content = self.sum - mg2[0].content
            print(self.x, " ",self.y," ", self.sum)
            return self.sum
y = self.methodB(self.z, msg[0]) + self.y
x = y + self.methodB(self.z, msg[0], msg)
```

Here, after entering method, if we pass 3 arguments, we have to execute the else block as mg2 != None. On the other hand, if we pass 2 arguments, if block would be executed, as the default value of mg2 is None.
For functions with any number of arguments (*variable), the position of the arguments passed, must be noticed carefully. *variables can take any number of arguments and all the arguments are kept inside a tuple.

def B(x, *y):
  print(x, y)
B(5,6,7,8)
B(5)
	def B(x, *y, z=0):
	  print(x, y, z)
	B(5,6,7,8)
	B(5)
		def B(x,*y,z=0):
		  print(x,y,z)
		B(5,6,7,8)
		B(5,z=100)

Output: 5 (6, 7, 8) 5 ()	Output: 5 (6, 7, 8) 0 5 () 0	Output: 5 (6, 7, 8) 0 5 () 100
Explanation: The first argument is passed onto x. Since y can take any number of arguments, the subsequent arguments are all pass inside the tuple of y.	Explanation: The first argument is passed onto x. Since y can take any number of arguments, z will never receive an argument unless we specify which variable should receive which argument (keyword arguments).	Explanation: Here while method calling, since we specified that z=100, it means that z in methodB will receive 100.

2. **Handing Pass by Reference:**

Whenever an immutable (int, string, float, tuple, etc.) value has been passed, it is called pass by value. For now, you can assume that a copy of the value is passed here. Let us consider the following code:

```
x= 5
y= x
x+= 1
```

After executing these lines, x=6 and y=5. Therefore, we can visualize that when y= x is performed, a copy of x's value is passed onto y and there is no connection between x and y. Which means, any changes to x, does not apply to y and vice-versa.

On the other hand, whenever a mutable (List, dictionary) value or an object has been passed, rather than sending a copy of the values, we actually send the reference/location of the value. Let us consider the following code:

```
def A(x):
  x [1] = 100
  print(x)
  return x

s= {1:50, 2:80}
print(s)
y=A(s)
print(s)
print(y)
```

```
def A(x):
  x [1] = 100
  print(x)
  return x

s= [10, 20, 30]
print(s)
y=A(s)
print(s)
print(y)
```

```
Output:
{1: 50, 2: 80}
{1: 100, 2: 80}
{1: 100, 2: 80}
{1: 100, 2: 80}
```

```
Output:
[10, 20, 30]
[10, 100, 30]
[10, 100, 30]
[10, 100, 30]
```

Here any changes made to s, x or y will have impact on all three. Therefore, passing a location and writing location name is way more convenient to keep track of the places you need to change.

s (Gulshan)	**x**	**y**
{1:50, 2:80}	Gulshan	Gulshan
{1:100, 2:80}		

As we are passing the location, we only need to change the value in the location Gulshan. Same goes for object passing.

3. **Position of the Method Call:**

self.y = self.y + self.methodB(self.z, msg[0])
y = self.methodB(self.z, msg[0]) + self.y

Here in the first line, the value of self.y must be fetched and written before calling methodB. The value of self.y may change in methodB, but we have already fetched the value before calling the method. On the other hand in the second line, self.y has been fetched after calling method, which forces us to use the most updated value of self.y after executing methodB.

Sample Tracing Problem

```python
class msgClass:
    def __init__(self):
        self.content = 0

class Q5:
    def __init__(self,z):
        self.sum = 1
        self.x = 2
        self.y = 3
        self.z = z

    def methodA(self):
        x, y = 1, 1
        msg = []
        myMsg = msgClass()
        myMsg.content = self.x
        msg.append(myMsg)
        msg[0].content = self.y + myMsg.content
        self.y = self.y + self.methodB(self.z, msg[0])
        y = self.methodB(self.z, msg[0]) + self.y
        x = y + self.methodB(self.z, msg[0], msg)
        self.sum = x + y + msg[0].content
        print(x," ", y," ", self.sum," ",self.z[0])

    def methodB(self, mg0, mg1, mg2 = None):
        if mg2 == None:
            x, y = 5, 6
            y = self.sum + mg1.content
            self.y = y + mg1.content
            x = self.x + 7 +mg1.content
            self.sum = self.sum + x + y
            self.x = mg1.content + x +8
            mg0[0]+=2
            print(x, " ", y," ", self.sum)
            return y
        else:
            x = 1
            self.y += mg2[0].content
            mg2[0].content = self.y + mg1.content
            x = x + 4 + mg1.content
            self.sum += x + self.y
            mg1.content = self.sum - mg2[0].content
            print(self.x, " ",self.y," ", self.sum)
            return self.sum

q = Q5([5])
q.methodA()
```

Solution

q (Dhaka)												
z or self.z (Gulshan)	**self.sum**	**self.x**	**self.y**	methodA				methodB				
[5]	1	2	3	**x**	**y**	**msg**	**myMsg**	**mg0**	**mg1**	**mg2**	**x**	**y**
[7]	21	27	11	1	1	[ ]	Mohakhali	Gulshan	Mohakhali	None	5	6
[9]	86	52	9	225	57	[Mohakhali]		Gulshan	Mohakhali	None	14	6
	168		31					Gulshan	Mohakhali	[Mohakhali]	5	6
	409		36								39	26
											1	
											46	

myMsg (Mohakhali)
self.content
0
2
5
5
41
127

Output
14 6 21
39 26 86
52 36 168
225 57 409 9