1. What is the result of the code, and explain?

>>> X = 'iNeuron'

>>>deffunc():

print(X)

>>>func()

Ans:- iNeuron

2. What is the result of the code, and explain?

>>> X = 'iNeuron'

>>>deffunc():

X = 'NI!'

>>>func()

>>>print(X)

Ans:- iNeuron

3. What does this code print, and why?

>>> X = 'iNeuron'

>>>deffunc():

X = 'NI'

print(X)

>>>func()

>>>print(X)

Ans:-iNeuron

iNeuron

4. What output does this code produce? Why?

>>> X = 'iNeuron'

>>>deffunc():

global X

X = 'NI'

>>>func()

>>>print(X)

Ans:-output = NI because x is defined by global keyword in function

5. What about this code—what’s the output, and why?

>>> X = 'iNeuron'

>>>deffunc():

X = 'NI'

def nested():

print(X)

nested()

>>>func()

>>> X

Ans:-iNeuron

‘iNeuron’

Because by calling nested function output is iNeuron and for x it will print its value ie: ‘iNeuron’

6. How about this code: what is its output in Python 3, and explain?

>>>deffunc():

X = 'NI'

def nested():

nonlocal X

X = 'Spam'

nested()

print(X)

>>>func()

Ans:- output:- spam

The following restrictions apply to the nonlocal instruction:

* the name that is declared in the nonlocal statement must already exist in the enclosing function at the time of the call in the nested function;
* when using the statement, names are looked up only in the scope of enclosing functions. The built-in and global scopes are not searched for nonlocal names, even if such names already exist in those scopes

7. Name three ways to retain state information in a Python function.

Ans:-State is **a behavioral design pattern that allows an object to change the behavior when its internal state changes**. The pattern extracts state-related behaviors into separate state classes and forces the original object to delegate the work to an instance of these classes, instead of acting on its own.

instance attributes, global variables, nonlocal variables, and function attributes can all be used for retaining state

An instance attribute is **a Python variable belonging to one, and only one, object**. This variable is only accessible in the scope of this object and it is defined inside the constructor function, \_\_init\_\_(self,..) of the class

Global variable

def **manageScore**():

global score

score = 0

def **hit**():

global score

score += 1

print(f"Just hit an enemy! Your score is {score} points.")

return hit

shoot1 = manageScore()

shoot 1()

Global variables are not the only way state retention may be achieved. The second option are nonlocal variables and closures. If we declare a variable **nonlocal** in the nested function, the program then works on the variable with that name which it finds in the enclosing function.

If the enclosing function returns another function and if that returned function remembers state from the enclosing scope, we call it closures.

Local variable

def **manageScore**():

score = 0

def **hit**():

nonlocal score

score += 1

print(f"Just hit an enemy! Your score is {score} points.")

return hit

function attributes

Another way of retaining state across function calls are function attributes. These are names that we attach to functions. Function attributes use the same dot notation as class or instance attributes. They work pretty much the same as nonlocal variables. One important difference is that they are accessible outside the nested function.

def **manageScore**():

def **hit**():

hit.score += 1

print(f"Just hit an enemy! Your score is {hit.score} points.")

hit.score = 0

return hit