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

>>> X = 'iNeuron'

>>> def func():

print(X)

>>> func()The output of this code will be:

```

iNeuron

```

When the `func()` function is called, it defines a new local variable `X` and sets its value to `'NI!'`. However, this variable is local to the function and does not affect the global variable `X` defined outside the function. Therefore, when the function finishes executing and `print(X)` is called, it prints the value of the global variable `X`, which is still `'iNeuron'`.

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

>>> X = 'iNeuron'

>>> def func():

X = 'NI!'

>>> func()

>>> print(X)

The output of this code will be "iNeuron".

In this code, a variable X is assigned the string "iNeuron" outside the function. Inside the function, a local variable X is created and assigned the string "NI!". When the function is called, it assigns the value "NI!" to the local variable X, but this variable is destroyed when the function completes its execution.

After the function call, the original value of X, which is "iNeuron", is printed using the print() statement outside the function. Therefore, the output of the print() statement will be "iNeuron

.

**3. What does this code print, and why?**

**>>> X = 'iNeuron'**

**>>> def func():**

**X = 'NI'**

**print(X)**

**>>> func()**

**>>> print(X)**

The code prints two different strings: 'NI' and 'iNeuron'.

Inside the function, the value of the variable X is changed to 'NI' with the assignment statement `X = 'NI'`. This new value of X is then printed with the `print()` function, producing the output 'NI'.

However, the value of X outside the function is still 'iNeuron'. When `print(X)` is called outside the function, it prints the value of X that is defined in the global scope, which is 'iNeuron'.

**4. What output does this code produce? Why?**

**>>> X = 'iNeuron'**

**>>> def func():**

**global X**

**X = 'NI'**

**>>> func()**

**>>> print(X)**

The output of this code will be `NI`. The `global` keyword is used inside the function `func()` to declare that the variable `X` is a global variable, rather than a local variable. When `func()` is called, it sets the global variable `X` to `'NI'`. When `print(X)` is called after the function call, it prints the new value of the global variable `X`, which is `'NI'`.

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

**>>> X = 'iNeuron'**

**>>> def func():**

**X = 'NI'**

**def nested():**

**print(X)**

**nested()**

**>>> func()**

**>>> X**

The code will output "NI" and then an UnboundLocalError. When `func()` is called, it defines a local variable `X` as 'NI'. Then it calls the `nested()` function which prints the value of the local variable `X`, which is 'NI'. However, when the `print(X)` statement is executed outside of the `func()` function, the local variable `X` is no longer in scope, so an UnboundLocalError is raised.

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

**>>> def func():**

**X = 'NI'**

**def nested():**

**nonlocal X**

**X = 'Spam'**

**nested()**

**print(X)**

**>>> func()**

In Python 3, the output of this code would be 'Spam'.

The func() function defines a nested() function, and X is defined as 'NI' within func(). nested() is then defined, and it uses the nonlocal keyword to declare that the variable X is defined in the enclosing scope (i.e., in func()). This means that when X is reassigned to 'Spam' within nested(), the variable X in func() is also reassigned to 'Spam'. Finally, func() prints the value of X, which is 'Spam' because of the nested() call.