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

```
>>> X = 'iNeuron'
>>> def func():
print(X)
>>> func()
```

The result of the code will be "iNeuron". The variable `X` is defined outside the function `func()` with the value "iNeuron". When the function is called and the `print(X)` statement is executed, it looks for the value of `X` in the enclosing scope, which in this case is the global scope. Since `X` is defined globally as "iNeuron", the function prints "iNeuron".

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

```
>>> X = 'iNeuron'
>>> def func():
X = 'NI!'
>>> func()
>>> print(X)
```

The result of the code will be "iNeuron". In this code snippet, the variable `X` is defined outside the function `func()` with the value "iNeuron". Inside the function, a new local variable `X` is created and assigned the value "NI!". When the function is called, it assigns "NI!" to the local variable `X`, but this does not affect the global variable `X`. When the `print(X)` statement is executed outside the function, it refers to the global variable `X`, which still has the value "iNeuron", and hence, "iNeuron" is printed.

3) What does this code print, and why?

```
>>> X = 'iNeuron'
>>> def func():
X = 'NI'
print(X)
>>> func()
>>> print(X)
```

The code will print "NI" when the function `func()` is called. Inside the function, a local variable `X` is created and assigned the value "NI". When the `print(X)` statement is executed inside the function, it refers to the local variable `X` and prints "NI". However, when `print(X)` is called outside the function, it refers to a different scope where `X` is not defined, resulting in a NameError.

4) What output does this code produce? Why?

```
>>> X = 'iNeuron'
>>> def func():
global X
X = 'NI'
>>> func()
>>> print(X)
```

The output of the code will be "NI". Inside the function `func()`, the `global` keyword is used to declare that the variable `X` being assigned is referring to the global variable. Therefore, when the function is called and the assignment statement `X = 'NI'` is executed, it modifies the global variable `X` to have the value "NI". When `print(X)` is called outside the function, it prints the updated 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" when the function `func()` is called. Inside the function, a local variable `X` is created and assigned the value "NI". Then, a nested function `nested()` is defined. When `nested()` is called, it prints the value of `X` in its enclosing scope, which is the local variable `X` with the value "NI". Finally, when `func()` is called, it prints "NI" as the output. The variable `X` defined outside the function remains unaffected.

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()
```

The code will output "Spam" when the function `func()` is called. Inside the function, the `nonlocal` keyword is used to indicate that the variable `X` being assigned refers to the closest enclosing scope above the current function. The nested function `nested()` modifies the value of the nonlocal variable `X` to "Spam". When `nested()` is called within `func()`, it updates the nonlocal variable `X` to "Spam". Finally, when `print(X)` is executed within `func()`, it refers to the modified nonlocal variable `X` and prints "Spam".