

CSC108: Mutability and Functions

Daniel Zingaro
University of Toronto
daniel.zingaro@utoronto.ca

November 2012

1 Changing or Returning

Here are two functions that seem to be similar, but have important differences. It's important that you understand the differences between these functions because these differences affect the code you will write to implement the functions.

Function A:

```
def insert_after(L, n1, n2):  
    '''(list of int, int, int) -> NoneType  
    Insert n2 after each occurrence of n1 in L.  
    '''
```

Function B:

```
def insert_after2(L, n1, n2):  
    '''(list of int, int, int) -> list of int  
    Return a new list consisting of all elements from L,  
    plus a copy of n2 after each occurrence of n1.  
  
>>> insert_after2([3, 4, 5], 3, 10)  
[3, 10, 4, 5]  
'''
```

Both of these functions have something to do with inserting `n2` after each occurrence of `n1`.

The first difference you should note is in the type contracts. The parameters for both functions are the same, but the types of the return values are different. `insert_after` does not return anything (indicated by `NoneType`) and `insert_after2` does return something (a list of int). So if `insert_after` tries to return something, or `insert_after2` does not have a return statement, the function would be incorrect.

The fact that one function returns something but the other does not is also reflected in the descriptions in the docstrings. Read the descriptions again for both functions. The description for `insert_after` refers to inserting elements directly in `L`. The description for

`insert_after2`, on the other hand, does not talk about modifying `L` but instead says that the function should return a new list. So `insert_after` is supposed to modify `L` directly and not return anything; `insert_after2` is supposed to leave `L` alone and create and return a new list.

Here is some code for both functions.

```
def insert_after(L, n1, n2):
    '''(list of int, int, int) -> NoneType
    Insert n2 after each occurrence of n1 in L.
    '''
    i = 0
    while i < len(L):
        if L[i] == n1:
            L.insert(i+1, n2)
            i += 1
        i += 1

def insert_after2(L, n1, n2):
    '''(list of int, int, int) -> list of int
    Return a new list consisting of all elements from L,
    plus a copy of n2 after each occurrence of n1.

    >>> insert_after2([3, 4, 5], 3, 10)
    [3, 10, 4, 5]
    '''
    new_L = []
    for element in L:
        new_L.append(element)
        if element == n1:
            new_L.append(n2)
    return new_L
```

Note how `insert_after` uses `insert` to directly modify `L`, whereas `insert_after2` never modifies `L`.

Another difference in the docstrings of these functions is that `insert_after` does not include an example, but `insert_after2` does include an example. The reason is that docstring examples give the **return value** for the function when it is called with the given parameters. `insert_after` does not return anything, so it would be incorrect to say:

```
>>> insert_after([3, 4, 5], 3, 10)
[3, 10, 4, 5]
```

It is incorrect because `[3, 10, 4, 5]` is not the return value from `insert_after`.

2 Using the Functions

Let's play around in the shell to see the difference between `insert_after` and `insert_after2`.

First, `insert_after` (I have saved both functions in file `mutable.py`):

```
>>> from mutable import *
>>> lst = [1, 2, 3, 4, 5]
>>> insert_after(lst, 3, 10)
>>> lst
[1, 2, 3, 10, 4, 5]
>>> lst2 = insert_after(lst, 3, 10)
>>> lst2
>>> print(lst2)
None
>>> lst
[1, 2, 3, 10, 10, 4, 5]
```

Make sure you understand why `lst2` is `None`. `insert_after` returns `None` since there is no return statement. It is therefore incorrect to do:

```
lst = insert_after(lst, ...)
```

Now for `insert_after2`:

```
>>> lst = [1, 2, 3, 4, 5]
>>> insert_after2(lst, 3, 10)
[1, 2, 3, 10, 4, 5]
>>> lst
[1, 2, 3, 4, 5]
>>> insert_after2(lst, 3, 10)
[1, 2, 3, 10, 4, 5]
>>> lst
[1, 2, 3, 4, 5]
>>> lst2 = insert_after2(lst, 3, 10)
>>> lst2
[1, 2, 3, 10, 4, 5]
>>> lst
[1, 2, 3, 4, 5]
```

Here, if we don't assign the return value of `insert_after2` to a variable, that return value is lost. This is because `insert_after2` creates a new list rather than modifies the existing list.

3 Exercise

Write the body for the following function. Note that the word `object` in the docstring just means that `v` can be any object (i.e. of any type). So `v` could be an integer or a string or a list or anything else. The type of `v` doesn't change your code at all: you just set every key's value to `v`.

```
def make_all_values_equal(d, v):  
    '''(dict, object) -> NoneType  
    Modify d so that each of its values equals v.  
    '''
```