

CSCI046 Lab: Scope and Memory Management

1 Instructions

For each problem below, complete the following steps:

1. Guess the output of the program.
2. Use vim to enter the program into the lambda server, and run the program. You should use the same file for each problem.
3. If the results of steps 1 and 2 are different, figure out why. You may find the website <https://pythontutor.com> helpful.

These problems are intentionally misleading, so don't just assume you know what the code will do. You will need to understand all of the concepts described in this document in order to successfully complete future programming homeworks, and your final exam will contain at least one problem similar to these.

2 Local and Global Variable Scope

By default, all variables are **global** and accessible anywhere inside or outside a function. A variable becomes **local** if it is defined within a function and the **global** keyword is not used.

Problem 1.

```
1 xs = [1,2,3]
2 def foo():
3     print('xs=',xs)
4 foo()
```

Problem 2.

```
1 xs = [1,2,3]
2 def foo():
3     xs = 'a'
4     print('xs=',xs)
5 foo()
```

Note: Modifying a global variable does not make the variable local. The only way to make a variable local is if it appears to the left of an equals sign *by itself* (i.e. appearing to the left of an equal sign does not matter if there is a slice). Calling a function that modifies a variable does not make a variable local.

Problem 3.

```
1 xs = [1,2,3]
2 def foo():
3     xs = 'a'
4 foo()
5 print('xs=',xs)
```

Problem 4.

```
1 xs = [1,2,3]
2 def foo():
3     xs[-1] = 'a'
4 foo()
5 print('xs=',xs)
```

Problem 5.

```
1 xs = [1,2,3]
2 def foo():
3     xs.pop()
4 foo()
5 print('xs=',xs)
```

Problem 6.

```
1 xs = [1,2,3]
2 def foo():
3     xs.append('a')
4 foo()
5 print('xs=',xs)
```

Problem 7.

```
1 xs = [1,2,3]
2 def foo():
3     xs.pop()
4 def bar():
5     global xs
6     xs = [4,5,6]
7     xs.append('a')
8 foo()
9 bar()
10 bar()
11 foo()
12 foo()
13 print('xs=',xs)
```

Problem 8.

```
1 xs = [1,2,3]
2 def foo():
3     xs = [4,5,6]
4     xs.append('a')
5 foo()
6 print('xs=',xs)
```

Problem 9.

```
1 xs = [1,2,3]
2 def foo():
3     xs = [4,5,6]
4     xs.append('a')
5     print('xs=',xs)
6 foo()
```

Problem 10.

```
1 xs = [1,2,3]
2 def foo():
3     global xs
4     xs = [4,5,6]
5     xs.append('a')
6 foo()
7 print('xs=',xs)
```

Problem 11.

```
1 xs = [1,2,3]
2 def foo():
3     xs.append('a')
4     xs = [4,5,6]
5 foo()
6 print('xs=',xs)
```

Problem 12.

```
1 xs = [1,2,3]
2 def foo():
3     global xs
4     xs.append('a')
5     xs = [4,5,6]
6 foo()
7 print('xs=',xs)
```

Note: It is fairly common to define functions within functions in python. These are called *local functions* because they are only accessible from within the outer function (just like local variables). These problems are designed to give you practice understanding how scope works in these local functions.

Problem 13.

```
1 x = 'a'
2 xs = [1,2,3]
3 def foo():
4     x = 'b'
5     xs = [1,2,3]
6     def bar():
7         x = 'c'
8         xs[0] = 'd'
9     bar()
10    print('x=', x)
11    print('xs=', xs)
12 foo()
```

Problem 14.

```
1 x = 'a'
2 xs = [1,2,3]
3 def foo():
4     global xs
5     x = 'b'
6     xs = [1,2,3]
7     def bar():
8         global x
9         x = 'c'
10        xs[0] = 'd'
11    bar()
12    print('x=', x)
13    print('xs=', xs)
14 foo()
```

3 Memory Management

Note: Assignment makes two variable names refer to the same object. Changing the contents of one variable actually changes the contents of the object, and therefore changes the contents of both variables. In order to create new, distinct, objects, you must copy the variable.

Problem 15.

```
1 xs = [1,2,3]
2 ys = xs
3 ys.append('a')
4 print('xs=',xs)
5 print('ys=',ys)
```

Problem 16.

```
1 import copy
2 xs = [1,2,3]
3 ys = copy.copy(xs)
4 ys.append('a')
5 print('xs=',xs)
6 print('ys=',ys)
```

Note: When lists contain non-container objects like integers, `copy` and `deepcopy` behave exactly the same way. When lists contain containers, then `copy` and `deep copy` behave differently.

Problem 17.

```
1 import copy
2 xs = [[1,2,3],[4,5,6]]
3 ys = copy.copy(xs)
4 ys[0][0] = 'a'
5 xs[1][1] = 'b'
6 print('xs=',xs)
7 print('ys=',ys)
```

Problem 18. Use the same code as above, but change `copy.copy` to `copy.deepcopy`.

HINT: Use vim's find/replace functionality with the `:s` command.

See https://vim.fandom.com/wiki/Search_and_replace for details.

Problem 19.

```
1 import copy
2 xs = [[1,2,3],[4,5,6]]
3 ys = copy.copy(xs)
4 ys.append('a')
5 ys[0].append('b')
6 print('xs=',xs)
7 print('ys=',ys)
```

Problem 20. Use the same code as above, but change `copy.copy` to `copy.deepcopy`.

HINT: Use vim's find/replace functionality with the `:s` command.

See https://vim.fandom.com/wiki/Search_and_replace for details.

Note: Variables that are parameters to a function are always local variables. But, if a default parameter is used, these objects are only created once and the same copy is used in all subsequent calls that require a default parameter.

Problem 21.

```
1 xs = [1,2,3]
2 def foo(xs=[]):
3     print('xs=',xs)
4     xs.append(len(xs)+1)
5 foo()
6 foo()
7 foo()
8 foo([])
9 foo()
10 foo()
```

Note: Unfortunately, default parameters behave differently for container objects and non-container objects.

Problem 22.

```
1 n = 7
2 def foo(n=0):
3     print('n=',n)
4     n+=1
5 foo()
6 foo()
7 foo()
8 foo(1)
9 foo()
10 foo()
```

Note: The following problems illustrate different ways that you might try to reverse a list in python. At first glance, they all look the same. Due to memory management issues, however, they are not all correct. Understanding memory management is important when tracking down these subtle bugs, and you are guaranteed to run into these situations in later assignments in this course.

Note: Writing a function that reverses a list is one of the classic interview questions for python programming jobs. Interviewers frequently say that they are shocked by the number of interviewees who fail these questions because they don't understand memory management. Pay special attention to these problems so that you do not fail future interview questions and can get a great job.

Problem 23.

```
1 def reverse_list(xs):
2     ys = xs
3     for i in range(len(ys)):
4         ys[i] = xs[-i-1]
5     return ys
6 xs = [1,2,3]
7 ys = reverse_list(xs)
8 print('xs=',xs)
9 print('ys=',ys)
```

Problem 24.

```
1 import copy
2 def reverse_list(xs):
3     ys = copy.copy(xs)
4     for i in range(len(ys)):
5         xs[i] = ys[-i-1]
6     return ys
7 xs = [1,2,3]
8 ys = reverse_list(xs)
9 print('xs=',xs)
10 print('ys=',ys)
```

Problem 25.

```
1 import copy
2 def reverse_list(xs):
3     ys = copy.copy(xs)
4     for i in range(len(ys)):
5         ys[i] = xs[-i-1]
6     return ys
7 xs = [1,2,3]
8 ys = reverse_list(xs)
9 print('xs=',xs)
10 print('ys=',ys)
```

Problem 26.

```
1 def reverse_list():
2     ys = xs
3     for i in range(len(ys)):
4         ys[i] = xs[-i-1]
5     return ys
6 xs = [1,2,3]
7 ys = reverse_list()
8 print('xs=',xs)
9 print('ys=',ys)
```

Note: Because reversing a list is a common task, python provides two inbuilt methods to accomplish it. The `reverse` function does not create a copy of a list, but instead changes the list in place. The `reversed` function makes a copy of the list, reverses the copy, and leaves the original list unmodified. Both functions are widely used in python code.

Problem 27.

```
1 xs = [1,2,3]
2 ys = xs.reverse()
3 print('xs=',xs)
4 print('ys=',ys)
```

Problem 28.

```
1 xs = [1,2,3]
2 ys = list(reversed(xs))
3 print('xs=',xs)
4 print('ys=',ys)
```

Note: Sorting is another common task, and python contains two functions for sorting lists: `sort` and `sorted`. It is a python convention that any function in the past tense returns a copy of the list without modifying the list, and any function written in the imperative modifies the list in place.

Problem 29.

```
1 xs = [2,3,1]
2 ys = xs.sort()
3 print('xs=',xs)
4 print('ys=',ys)
```

Problem 30.

```
1 xs = [2,3,1]
2 ys = list(sorted(xs))
3 print('xs=',xs)
4 print('ys=',ys)
```

Note: Internally, the `reversed` and `sorted` functions perform shallow copies instead of deep copies.

Problem 31.

```
1 xs = [[1,2,3],[4,5,6]]
2 ys = list(reversed(xs))
3 ys.append('a')
4 ys[0].append('b')
5 print('xs=',xs)
6 print('ys=',ys)
```

Problem 32.

```
1 import copy
2 xs = [[1,2,3],[4,5,6]]
3 ys = list(reversed(copy.deepcopy(xs)))
4 ys.append('a')
5 ys[0].append('b')
6 print('xs=',xs)
7 print('ys=',ys)
```

Note: There are many ways in python to make shallow copies of containers besides using `copy.copy`. You will see people use all of them in real python code, although the most “pythonic” way to do it is to use the `copy.copy` function since it is the most readable. The only way to make deep copies is with the `copy.deepcopy` function.

Problem 33.

```
1 xs = [1,2,3]
2 ys = xs[:]
3 ys.append('a')
4 print('xs=',xs)
5 print('ys=',ys)
```

Problem 34.

```
1 xs = [1,2,3]
2 ys = list(xs)
3 ys.append('a')
4 print('xs=',xs)
5 print('ys=',ys)
```

4 Memory and Loops

Modifying a container that you are looping over is dangerous; strange behavior can occur depending on how you modify the container. We call these errors “silent errors” because they do not generate error messages.

Silent errors are one of the most insidious sources of programming bugs because they are so difficult to detect. Unfortunately, python is notorious for these silent errors due to its weak type system. Other languages (like C++, Java, and Haskell) have much stronger type systems that prevent certain classes of silent errors from ever occurring.

If you want to modify a container inside a for loop, you should probably instead loop over a copy of the container to prevent unintended behavior.

Problem 35.

```
1 xs = [1,2,3,4,5]
2 for x in xs:
3     print('x=',x)
4     xs.pop()
```

Problem 36.

```
1 xs = [1,2,3,4,5]
2 for x in xs:
3     print('x=',x)
4     del xs[0]
```

Problem 37.

```
1 xs = [1,2,3,4,5]
2 for x in list(xs):
3     print('x=',x)
4     xs.pop()
```

Note: Not all containers allow modification in the middle of a loop; the following code tries to modify a `deque` and generates an error. (A `deque` is a container that supports all of the same operations as python’s `list` type, but it has different runtime properties, which we’ll talk about next week.) These error messages are good, and follow the so-called “fail loudly” principle. The inconsistency that some containers fail loudly and some fail silently is one of python’s major flaws IMNSHO.

Problem 38.

```
1 from collections import deque
2 xs = deque([1,2,3,4,5])
3 for x in xs:
4     print('x=',x)
5     del xs[0]
```
