National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester II, 2022/2023

Debugging Exercises IV

So far we have introduced basic debugging techniques using both IDLE's native debugger and an online tool called Python Tutor. In this session, we will focus on some common mistakes on Python list and sequences. It involves memory allocation, which can be easily visualized in Python Tutor.

To start,

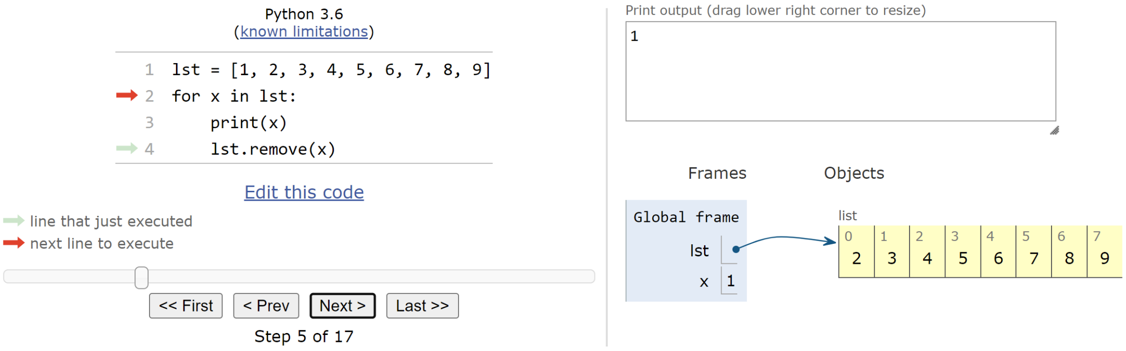
1. Open `http://pythontutor.com`.
2. Click on "Start writing and visualizing code now".
3. Make sure that the first dropdown menu is set to "Python 3.6".
4. Paste this code into the code editor (double-check the indentation after pasting):

   ```
   lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
   for x in lst:
       print(x)
       lst.remove(x)
   ```

5. Click on "Visualize Execution".
6. Click on "Last >>" to execute the code.

What is the output? Is it the same as what you expected? The code seems to print all elements in the list. Whenever an element is printed, it is removed from the list. But it turns out only `1, 3, 5, 7` and `9` are printed. Why is that so?

Let's go back to the beginning by clicking on the "<< First" button. Click on "Next >" until the program pauses at Step 5.



```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
for x in lst:
    print(x)
    lst.remove(x)
```

```
1
```

Global frame	
lst	•
x	1

0	1	2	3	4	5	6	7
2	3	4	5	6	7	8	9

We can see that after `lst.remove(x)` was executed, `lst` itself was changed. This indicates that List (as compared to Tuple) is a mutable data structure.

The problem comes from the line `for x in lst:`. In the process of execution, `x` will be assigned to the values starting from index `0` till the end of `lst` is reached. At index 0 (1st iteration), `x` has the value `1`. At index 1 (2nd iteration), `x` has the value `3` instead of `2`. This is because in the second iteration `lst` has become `[2, 3, 4, 5, 6, 7, 8, 9]`. Similarly, at index 2 (3rd iteration) `x` has the value `5` so on and so forth.

Note that in the end, `lst` is not empty: It has the value `[2, 4, 6, 8]` instead. This is a very common mistake in list processing, mutating a list while iterating through it may result in an unexpected output.

Let's see another example. Suppose we want to implement a function `insert(lst, idx, elem)` that takes in a list `lst` and modifies `lst` by inserting a new element `elem` at index `idx`. The function should return `None`.

Yang Shun has provided one implementation below:

```
def insert(lst, idx, elem):
    lst = lst[:idx] + [elem] + lst[idx:]
```

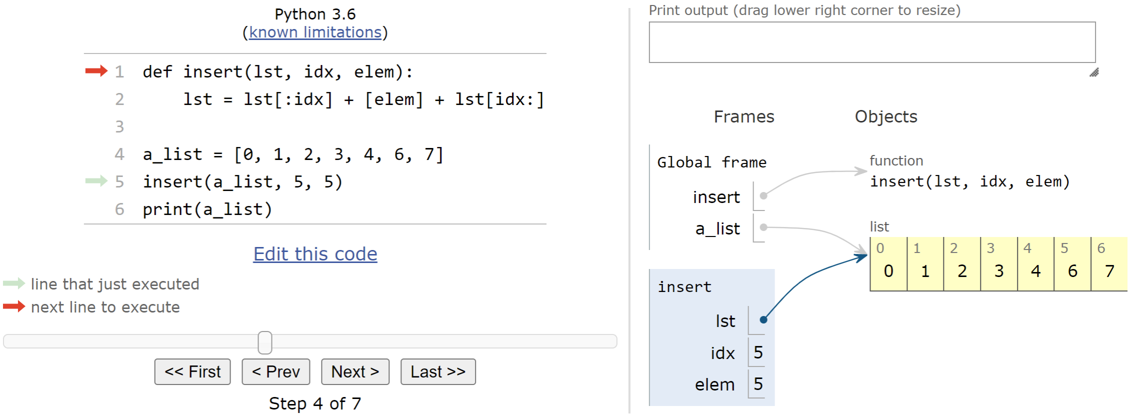
Is Yang Shun's implementation right? Can you come up with test code to check whether his implementation is right or wrong?

```
a_list = [0, 1, 2, 3, 4, 6, 7]
insert(a_list, 5, 5)
print(a_list)
```

The testing code above tries to insert `5` at index `5`. If the original list that was passed into the function `insert` was properly modified, the printout should be `[0, 1, 2, 3, 4, 5, 6, 7]`. But the actual printout is `[0, 1, 2, 3, 4, 6, 7]`!

Paste both the function and testing code into the Python Tutor editor and start visualizing. We can see two boxes in the Frames column. The boxes mark different scopes. Function `insert` and List `a_list` are defined globally. The variables produced during execution inside the function `insert` belongs to another scope.

At Step 4, we can see from the right side that the parameter `lst` points to the same object as `a_list`. That means whatever changes we made to `lst` inside function `insert` will also have an effect on `a_list`.

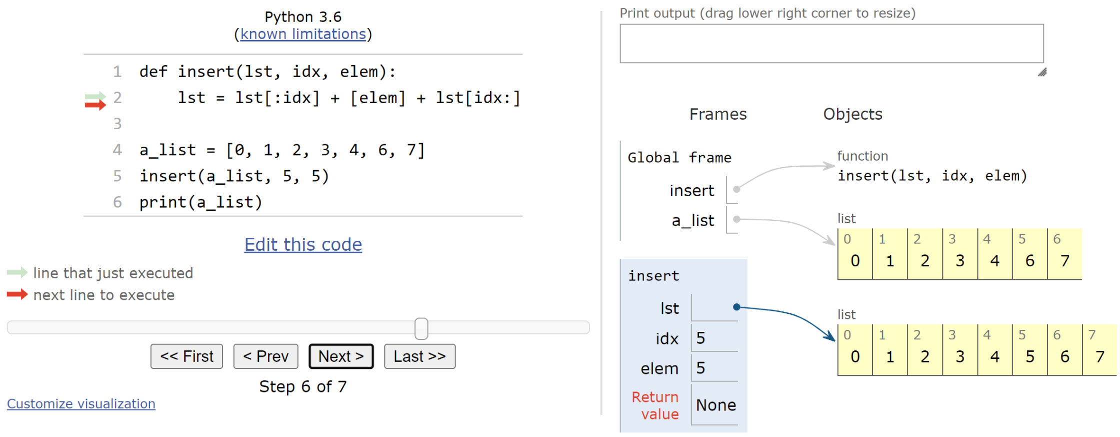


Python 3.6
(known limitations)

```
def insert(lst, idx, elem):
    lst = lst[:idx] + [elem] + lst[idx:]

a_list = [0, 1, 2, 3, 4, 6, 7]
insert(a_list, 5, 5)
print(a_list)
```

Let’s continue to Step 6. We find that something unexpected has happened. `lst` now has the value we want. But it no longer points to the original `a_list`. It is pointing to a new list object which will be released when function `insert` finishes execution. `a_list` wasn’t modified at all.



Python 3.6
(known limitations)

```
def insert(lst, idx, elem):
    lst = lst[:idx] + [elem] + lst[idx:]

a_list = [0, 1, 2, 3, 4, 6, 7]
insert(a_list, 5, 5)
print(a_list)
```

Edit this code

The reason here is that list slicing (i.e., `lst[start:end]`) creates a **new copy** of the original list instead of modifying it. Now think about how to fix this program and submit your answer in Question 1 of the `Debugging Exercises IV` Training. There are also 3 more questions for you to apply your new debugging skills!