CS 115 Lab 8: The Heap

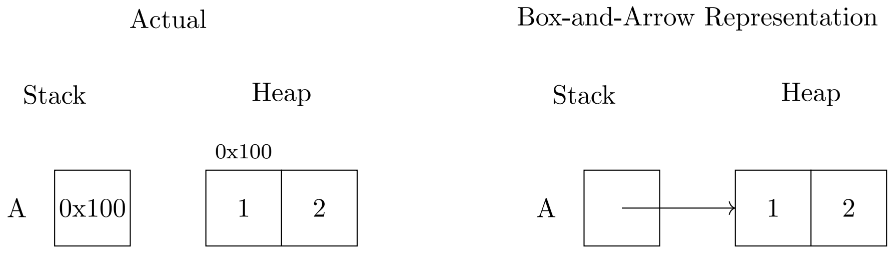
In class, we have now seen that the Python Memory Model is quite complex, involving the Heap, Stack, and several other things we did not even cover. On top of that, because larger values are stored as addresses/pointers, we can observe some interesting behaviour when it comes to making copies.

In this lab, you will practice your skills by analyzing some of these behaviours, both to figure out what is going on and how to use it to your advantage.

Task 1: A Box-and-Arrow Diagram

We begin by just reviewing how we represent objects like lists in memory. While the technical truth is that a list is stored on the heap with references to that list stored as an address, we typically denote this using a Box-and-Arrow Diagram, with an arrow representing one cell of memory storing the address of the object it points to.

For example, the following showcases the actual values and the Box-and-Arrow version of the memory layout for the code `A = [1,2]`[1].



Your job for this task is to draw the Box-and-Arrow Diagram for the memory state after the following snippet of code runs. This will be evaluated by the CA during the CA Check-In.

```
ex = [[3, 5], [9, 7]]
am = ex[:1]
am.append(ex[0])
```

Additionally, you should run the code afterwards. Inspect the values of the variables after the code runs. Does this match your predictions from the diagram? If not, find what the discrepancy is.

CA Check-In

Visit the CA, and show them your box-and-arrow diagram. Be prepared to explain your memory layout, and to be able to modify it.

The CA Check-In covers all the grades for Task 1 this time, so make sure that you do it before the lab period ends.

Task 2: Identifying Deep Copies

In lecture, we have seen several ways of making copies. Some we have seen don't actually copy a list, but instead only make a new reference to that same list. Others properly make a new list and copy the contents over to make a true copy. But even these methods sometimes fail to fully disentangle the copy, especially when it comes to multi-level lists.

[1]Note that the choice of 0x100 for the memory address is arbitrary. In actuality, the list would receive some memory address that cannot be predicted from looking at the code, which would be assigned dynamically during runtime. That is another reason to use Box-and-Arrow Diagrams: If we don't know what the number will be (just that it appears in multiple places), we should not write down an explicit number.

In this task, you will look at some methods of copying lists, and identify which of them properly deep copy. More specifically, for each of the following methods, classify that method into one of the following three classes:

- **Reference Duplication:** The new variable stores a pointer to the same list as the original. No copy was actually created.
- **Basic Copy:** A new list is created, that the new variable refers to. This new list stores pointers to the same list as the original list, however. The result is that operating on one list will not change the other list, but operating on a sublist of one list will have the change be visible from both.
- **Deep Copy:** A new list is created, and every sublist is also copied to a new list. The two copies are fully disentagled: You cannot edit the copy to change the original or vice versa.

The methods are listed in the following block of code. For each example, assume that `original` is a 2D array: A `list` of `lists`.

```python
method_1 = original
method_2 = original[:]
method_3 = list(original)

def id(x):
    return x

method_4 = id(original)
method_5 = list(map(id, original))
method_6 = list(map(lambda row: row, original))
method_7 = list(map(lambda row: id(row), original))
method_8 = list(map(lambda row: row[:], original))
```

To determine which method does what, we encourage to run the code. Examine the result using the debugger, use the `id` and `is` commands to inspect it, and experiment by trying to mutate the lists. Use whatever process will help you to figure out which method falls into which category.

Report question

1. For each method listed above, indicate whether it is a Reference Duplication, a Basic Copy, or a Deep Copy.
2. What process did you use to find your above answers? Write around 2 sentences explaining your approach.

Task 3: Copying in the Third Dimension

Consider a three dimensional array: A list of lists of lists of values. This can be used to represent images (like in homework 4), a 3-dimensional grid, or many other structures.

However, observe that as with any nested list, care has to be taken during copying, to avoid entangling the copy with the original.

In this task, you will implement the `copy_3d` function, which must deep copy a 3D array, such that changing the copy must not affect the original list. Use what you've learned in the previous part to both guide your development process and your testing to ensure that the function works as intended.

As an aside, Python provides a `deepcopy` function in its `copy` module. You are not allowed to use this for this lab. We want you to understand what goes into a deep copy, so we require that you do it yourself.

Task 4: Copy Verification

In this lab, you have now seen many ways of making copies, some of which successfully disentangle the copies from the original, and some of which do not copy the list sufficiently deeply.

But how could you tell? If your friend gives you code that they claims solves Task 3, how could you know if it works?

Well, you can write code that checks it! In this exercise, your job is to implement `check_copy_3d`. This takes in two 3D arrays `lll_1` and `lll_2` and verifies that one is a deep copy of the other. In other words, it checks that the two are equal, but that the two are unentangled: that they are not the same list, that their corresponding elements are not the same list in memory, and that this continues all the way down.

To test this, try different ways of making copies of lists, and check that your method works properly on them.

Bonus Task: ∞D Copies

Now, you have code that works for a 3D array. But what about a 4D array? Or a 5D one?

As a bonus task, we challenge you to write a `true_deepcopy` method, which works on lists of arbitrary depths. This part will not be graded, but we encourage you to try the challenge anyway.

Some interesting complications for you to consider:

- **Variable Nesting Depth:** What if the list does not have uniform depth? For example, consider `[1, [2, 3]]`. This cannot be assumed to a 1D or a 2D array. Consider making your code be able to handle such cases.
- **Variable Types:** Your code might be able to handle arbitrarily nested lists, but what about arbitrarily nested tuples? What about lists of tuples of lists? Can you make your code handle different types of nesting? You might find the fact that `type(x)` returns the type of `x`, which can then also be used as a casting function useful!
- **Circular Lists/Duplicate Pointers:** If you update a list via code like `x[0] = x` or `y[1][2] = y`, you can end up with a circular list: One where following several pointers can make you end up back in an earlier location. Any naive strategy for deep copying will likely end up in an infinite loop.

  But another potential issue is something like `x = [[1,1],0]` followed by `x[1]=x[0]`. Here, a deep copy should maintain the structure that the first and second elements are linked. And a naive strategy wouldn't work here either.

  Coming up with a strategy that manages to handle both of these scenarios requires some truly clever modifications, and we encourage you to think about how to accomplish this.

Submission Instructions

For this lab, you must submit the following files, with exactly the following names:

- `lab8.py`
- `lab8_report.txt`

Remember to put your name and the Stevens Honor Pledge on every file you submit! Failure to put the pledge can result in marks being deducted, with repeat penalties potentially getting stricter between assignments.

Submissions must be handled through Gradescope (accessible through Canvas). Grades will be released through Gradescope once the lab is graded.

Notes and Reminders

- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.
- We have provided a test file for you to test your solutions. By running this test file, you should be able to verify that your files are named properly, and that your code works correctly.
- Test frequently! Try out many different cases to verify that your code runs. Our provided test file may not be comprehensive, and so you should ensure to test your code on many other cases as well.

- Code that doesn't compile (throws a `SyntaxError` on being run) will receive a 0 for the assignment! Make sure you submit code that can be run, even if it does not pass all the test cases.
- **Don't forget to add your name and pledge!**