Solutions to CS 61A Challenge Problems: Mutable Data

Other worksheets and solutions at https://alextseng.net/teaching/cs61a/

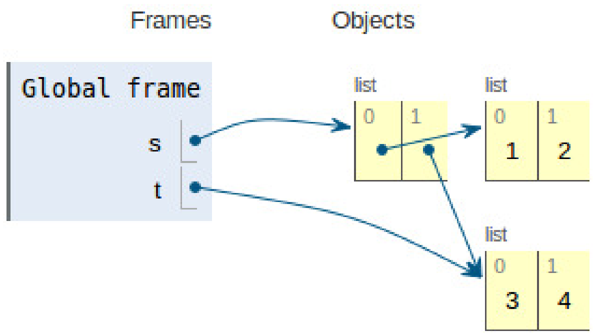
Alex Tseng

1 Environment Diagrams of Lists

Draw the environment diagrams of the following. Assume execution is all in the global scope.

(a)

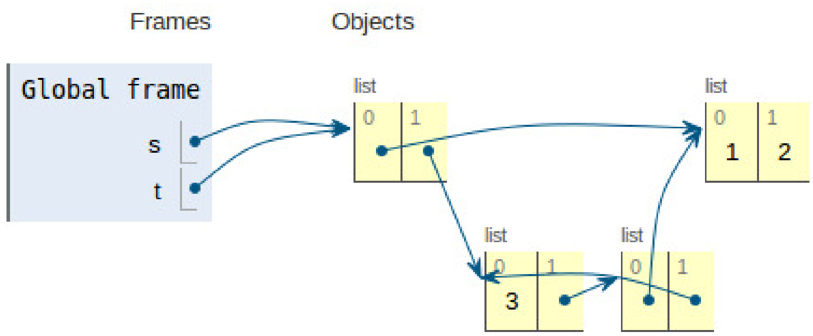
```
s = [[1, 2], [3, 4]]
t = s[1]
```



Do not be fooled by the complex structure of `s`. On the highest level, `s` is a 2-element list. It has 2 things inside of it. Those things just happen to be lists, too. When we set `t = s[1]`, we evaluate the right side of the assignment first, which is the second item in `s`. This is the list `[3, 4]`. Then we make `t` point to it.

(b)

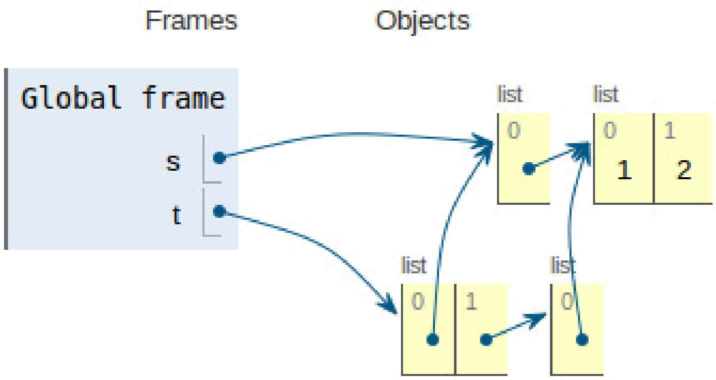
```
s = [[1, 2], [3, 4]]
t = s
t[1][1] = list(s)
```



Here, once we create `s`, we make `t` point to the same thing. After the second line is run, both `s` and `t` point to the list of 2 elements (both elements are lists). Now we run the third line. `t[1][1]` corresponds to the 4 in list that both `s` and `t` share. We set that to `list(s)`. `list` takes in an iterable value (like a tuple, a dictionary, another list, etc.), and will create a *new* list whose elements are *the same* as the input. This is why instead of the 4, we see it pointing to a new list whose elements are the same things that are in `s`.

(c)

```
s = [[1, 2]]
t = list([s, s])
t[1] = list(s)
```



When we set `t` to `list([s, s])`, we are creating a *new list* of length 2, with each element pointing to what `s` points to. Then we set the second element of this new list to another new list with only one item, that points to `[1, 2]`. Be careful with this one. The second line calls `list` on a list of `s`. The third line calls `list` on just `s`, which is already a list (a list of 1 element—another list!)

2 Linked Lists

(a) Create a linked list that includes a loop. That is, if we were to continuously call `rest` on the list, we would never reach `"empty"`.

```
x = [1, [2, [3, "empty"]]]
x[1][1][1] = x
```

(b) *Challenge* Write a function `has_loop(s)` that checks if `s` has a loop. Pseudocode is fine, but make sure you can translate it into native Python.

```
has_loop(s):
    tortoise = s
    hare = rest(s)
    while hare != "empty":
        if hare == tortoise:
            return True
        tortoise = rest(s)
        if rest(hare) == "empty":
            return False
        hare = rest(rest(hare))
    return False
```

Don't worry if you didn't get this problem. This is a very tricky one that is extremely prone to one-off errors. Just understand the concept of how it works. We set up a tortoise and a hare. The tortoise will traverse `s` slowly, one link at a time. The hare, on the other hand, will traverse `s` quicker, two nodes at a time. Notice that we call `rest(rest(hare))`. If the tortoise and the hare ever meet each other, then we know they must have encountered a cycle, because the hare is supposed to always be in front of the turtle.

3 List and Dictionary Comprehensions

(a) Using a single (possibly nested) list comprehension, compute the set of prime numbers from 0 to 99 (inclusive). Your list comprehension should return a list of lists, where the `i`th list is the list of prime numbers in `[i*10, (i*10)+9]`. The result should look something like:
`[[2, 3, 5, 7], [11, 13, 17, 19], ...]`
You may assume that there is a function `is_prime(x)` that returns `True` if `x` is prime and `False` otherwise.

```
[[x + (y * 10) for x in range(10) if is_prime(x + (y * 10))] for y in range(10)]
```

The nested list structure clues you in that this will probably be a nested list comprehension. The outer list comprehension in `y` simply iterates from 0 to 9. The inner list comprehension calculates the numbers `[i*10, (i*10)+9]`, and puts them in the list if they are prime.

(b) Use a single dictionary comprehension that maps each element of a list `items` to the number of times it appears in `items`, but only if it appears more than 2 times.
If `items` is: `["A", "A", "A", "B", "B", "C", "C", "C", "C", "D"]`,
then the result will be: `{"A": 3, "C": 4}`

```
{x:items.count(x) for x in items if items.count(x) > 2}
```

(c) Use a single list comprehension to compute the set of right triangles with *integer* side lengths no more than 30 (each side must be an integer ≤ 30). A triangle is defined by its three sides. Your list comprehension should return a list of tuples, each with the lengths of the three sides:
`[(3, 4, 5), (5, 12, 13), ...]`
Hint: all right triangles follow the Pythagorean theorem.

```
[(a, b, c) for a in range(1, 31) for b in range(a, 31) for c in range(b, 31) if a**2 + b**2
== c**2]
```

This is sort of a 3D list comprehension. We iterate through all `a` from 1 to 30 (inclusive), and then we iterate through all `b` from `a` to 30 (inclusive). The reason we start from `a` is because we don't want to include repetitions. If we had `b` start from 1 to 30, then we would end up with the same triangles repeated, but with a different ordering of edge lengths. Similarly, then we iterate through all `c` from `b` to 30. We include them in the list if and only if they form a Pythagorean triple. Notice that `a` is always smaller than `b`, and `b` is always smaller than `c`. This is another reason why we don't start from 1 every time. This way, we only need to check that $a^2 + b^2 = c^2$, and not $a^2 + c^2 = b^2$ or anything else.