CS 61A Resources

Mutability and Nonlocal

Intro

This is a brief guide meant to serve as an extension to our discussion about mutability and nonlocal. Before continuing, be sure to have a good understanding of the difference between expressions and values and what a name is.

Names

Let's briefly review names. In Python, a name contains exactly one value. This value could be of any data type.

The easiest way to make a name-value binding is with an assignment statement. Let's assign `x` to the value 3:

```
>>> x = 3
>>> x
3
```

Assignment statements evaluate the right hand side of the `=` before assigning it to the name on the left. Here, `x` is also assigned to the value 3, since that's what `1 + 2` evaluates to. We do *not* store the expression that was used in the assignment statement. Therefore, names have no knowledge of where their value came from.

```
>>> x = 1 + 2
>>> x
3
```

We can change the value of a name by just overriding it with another assignment statement.

```
>>> x = 3
>>> x = 5
```

First, we assign `x` to `3`. Then, we change its assignment to the value `5`. `x` does not know that it used to be assigned to `3`. It only knows its new value.

Local names

Names that exist in frames are **local** to that frame. You are allowed to change and access its value within the same frame. Functions that are defined in the frame can also access these names.

Here, we define `bar` inside of `foo`, so when we call `foo`, it returns a `bar` function whose parent is f1. The `bar` frame will look for the name `x` in `foo`'s frame since it has no `x` defined in its own frame:

```
>>> def foo():
...     x = 4
...     def bar():
...         return x
...     return bar
>>> foo()()
4
```

Frames can do lookup in their parent's parent frame and so on all the way until the global frame

However, lookup is different from assignment. With lookup, if we cannot find the name in the current frame, we can look in parent frames as described above. However, with assignment, we are restricted to either making or changing a name in we local scope. In other words, if we cannot find the name in the current frame, we simple add a new binding to that frame.

```
>>> def foo():
...     x = 4
...     def bar():
...         x = 5
...         return x
...     return bar
>>> foo()()
5
```

Here, we assign `5` to `x` in `bar`'s frame which creates a new name in f2 instead of changing `x` from f1. The `x` in f1 still exists, but lookup for `x` in f2 will find `5` instead of `4`.

```
==================
| f1: foo [p = G] |
------------------
|            x: 4 |
==================

====================
| f2: bar [p = f1] |
--------------------
|             x: 5 |
|  return value: 5 |
====================
```

This makes for some funky behavior if you're not careful:

```
>>> def foo():
...     x = 4
...     def bar():
...         x = x + 1
...         return x
...     return bar
>>> foo()()
UnboundLocalError
```

Here, our intention might be to grab the value `4` from f1, add 1 to it, then store it in a new name `x` in f2. However, we end up getting an error because Python either sees a name as nonlocal or local, but not both. Specifically, when it sees the `x` on the left hand side of the assignment statement, it notes that `x` must be a local name, since we can't change bindings that aren't local. However, when it gets the right hand side of the assignment statement and tries to lookup the name `x` locally, it fails. That is why we get the `UnboundLocalError`; it sees no local binding for `x`.

Nonlocal

Now we introduce the **nonlocal** statement, which allows you to change an assignment in a parent or ancestor frame.

The following code will change `x`'s assignment in `foo`'s frame to 5 instead of creating a new name `x` in f2. This is because we tell Python explicitly that any references to `x` in this frame will be nonlocal, i.e. outside the scope of this frame.

```
>>> def foo():
...     x = 4
...     def bar():
...             nonlocal x
...         x = 5
...         return x
...     return bar

foo()()
```

```
==================
| f1: foo [p = G] |
------------------
|            x: 5 |
==================

====================
| f2: bar [p = f1] |
--------------------
|  return value: 5 |
====================
```

As with regular name lookup, `nonlocal` lookup can continue in parents' parent frames. There are two major caveats, however:

1. You cannot use the `nonlocal` keyword to change global bindings. You must use the `global` keyword for this.
2. You cannot use `nonlocal` to override bindings in the current frame.

Mutability

Mutability refers to an object's ability to change its state after being created. For example, after you create a list, you are able to add things to it or remove things from it.

```
>>> a = [1, 2, 3, 4]
>>> b = a
>>> a.append(5)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
```

Notice that `b` 's value changes too, because we change `a` 's value, which `b` also points to. Here, we've used the list method `append` . There are several more list mutation methods that you should know, including `extend` , `pop` , `insert` , and `remove` . Note that although their effects are different, they all mutate the actual list and return `None` (with the exception of `pop`).

Mutation vs. Changing Bindings

It is important to recognize whether code is simply changing the assignment of a name from one value to another value, or is actually mutating the existing value to a different state.

Changing an assignment of a variable requires using an assignment statement to get the variable to point to a new value. This **does not** change, or *mutate*, any existing values.

Mutating a value actually changes an existing value, without affecting any bindings.

You can achieve similar but not the exact same effects with both techniques.

For example, let's say I want to define a function `bar` such that the following output is produced:

```
>>> def foo():
...     a = [1, 2, 3, 4]
...     def bar():
...         # fill this in
...     bar()
...     print(a)
>>> foo()
[1, 2, 3, 4, 5]
```

I don't care how it's done or whether or not new values are created, but the end result should be that `a` in frame 1 evaluates to `[1, 2, 3, 4, 5]`.

Here's one way:

```
def bar():
    nonlocal a
    a = a + [5]
```

Here, we use a nonlocal statement to tell Python to look for a in parent frames only. Then we assign a to a new list containing `[1, 2, 3, 4, 5]`. Why do I need to use nonlocal? Because I'm changing a variable that isn't in my current frame. If you were to draw this out, and I encourage that you do, you duplicate the list `[1, 2, 3, 4]`, add a 5 to the end to make `[1, 2, 3, 4, 5]`, erase a's pointer to the object `[1, 2, 3, 4]` and point it to the new list. Doing `a = [1, 2, 3, 4, 5]` has the same effect.

Can we do this without nonlocal? That is, can we achieve this effect without changing a's assignment, but instead by changing the actual list? That's where mutation comes in. Here's an example using the methods append, extend, and insert.

```
def bar():
    a.append(5)

def bar():
    a.extend([5])

def bar():
    a.insert(4, 5)
```

Here, we don't touch a's assignment. We simply add 5 after 4 to the list object. The list is mutated, or changed, from its original state, but it still has the same object ID as before. Because we don't change what a references, we don't need a nonlocal statement. If we draw this out, no new objects are created and no pointers are changed/erased.

As a final example of Python's quirkiness, take a look at this example:

```
def bar():
    nonlocal a
    a += [5] # equivalent to a.extend([5])
```

In this code, `+=` behaves much like `extend` instead of `a = a + [5]` as expected and actually mutates `a`. However, Python still sees the `=` as an assignment statement and requires the `nonlocal` statement. It is **not** recommended to do this as we are combining the use of `nonlocal` and mutation. Instead, use the previous example if you want to make a new list, or the following examples if you want to mutate.

Summary

The takeaway is that you need a `nonlocal` statement in order to use a nonlocal name on the left hand side of an assignment statement. You do not need `nonlocal` if you are simply mutating the value in a nonlocal frame. Take a look at all of these functions drawn out and take note of the objects being created and pointers being changed.

I recommend that you run the following block of code in Python tutor, replacing the `bar` function with the various examples above and some of your own attempts.

```
>>> def foo():
...     a = [1, 2, 3, 4]
...     b = a
...     def bar():
...         # fill this in
...     bar()
>>> foo()
[1, 2, 3, 4, 5]
```

Notice that we make an additional reference to the initial list here. Make sure you understand when `b` will change along with `a` (because of mutation) and when it still remains `[1, 2, 3, 4]`.