Discussion 4:
Nonlocal and Mutation

Tammy Nguyen (tammynguyen@berkeley.edu)

February 22, 2018

Assignments

Homeworks

HW 5 due next Thursday

Projects

Maps due tonight

Lab check-offs

An opportunity to speak one-on-one with a staff member about the previous week's material.

Goal: Make sure you are caught up in the class and understand concepts enough to move on.

Not meant to be an evaluation. You won't lose points if you ask questions or are unsure about something!

If you struggle during a check-off, that's an indication that you should seek out extra help for those concepts (CSM, tutoring, OH, etc.).

Extra resources

Today’s discussion is loaded with new concepts

Consult the following resources as supplements to today’s discussion:

- Appendix slides (at the end of this presentation)
- Nonlocal/mutability guide on my website: tmmydngyn.com/cs61a/guides/mutability

Agenda

I. Variable scope
II. Mutable lists
III. Dictionaries

Variable scope

Name lookup

x is found in local frame:

```
def foo():
    x = 10
    def bar(x):
        return x
    return bar

foo()(3)
```

x is found in parent frame:

```
def foo():
    x = 10
    def bar(y):
        return x + y
    return bar

foo()(3)
```

***Takeaway:** use binding in current frame if it exists and look in parent frames if it doesn't*

Assignment statements

Assigning a new variable in bar:

```python
def foo():
    x = 10
    def bar():
        x = 13
        return x
    return bar

foo()()
```

Reassigning an existing name in bar:

```python
def foo():
    x = 10
    def bar(x):
        x = 13
        return x
    return bar

foo()(3)
```

Takeaway: *rebind name in current frame if it exists and create new binding if it doesn’t; parent frames are uninvolved*

Nonlocal

By default,

- you *can* **access** variables in parent frames.
- you *cannot* **modify** variables in parent frames.

`nonlocal` statements allow you to modify a name in a parent frame instead of creating a new binding in the current frame.

```python
def foo():
    x = 10
    def bar():
        nonlocal x
        x = 13
    bar()
    return x

foo()
```

This `nonlocal` *statement tells Python: "Don't create a new local variable x; modify the one in the parent frame instead!"*

New name lookup rules

Introducing nonlocal variables means we need new rules to do name lookups.

The only difference between lookin up local and nonlocal names is where to start looking.

Name is **local**.

1) Look for the name in the **current frame** first.
2) If it's not found, look in parent frames.

Name is **nonlocal**.

1) Look for the name in the **parent frame** of the current frame first.
2) If it's not found, continue looking at parent frames.

Environment diagram practice

```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()
```

Nonlocal name lookup rules:

1) Look for the name in the current frame's parent frame first.
2) If it's not found, continue looking at parent frames.

Nonlocal variable assignment rules:

1) Use *nonlocal name lookup rules* to find the binding.
2) Replace the old binding of the name with the new value.

Side effects

A function has a **side effect** if it has a lasting effect in the program not involving its return value, such as printing something.

A function that modifies nonlocal variables has a side effect since it makes a change to the environment that will persist even when its frame is closed.

Code writing practice

Problem 2.3 - `memory`

What are the input, output, and side effect(s) of this function?

Note that `memory` is a higher order function that returns another function. What are the input, output, and side effect(s) of the returned function?

```python
def memory(n):
    """
    >>> f = memory(10)
    >>> f(lambda x: x * 2)
    20
    >>> f(lambda x: x - 7)
    13
    >>> f(lambda x: x > 5)
    True
    """
    # Fill this in
```

Code writing practice

`memory(n)`

- **Input:** one number, n
- **Output:** a one-argument function (see below)
- **Side-effects:** none

return value of `memory(n)`

- **Input:** a one-argument function
- **Output:** none
- **Side-effects:** updates n with the result of calling the input function on n, prints the result

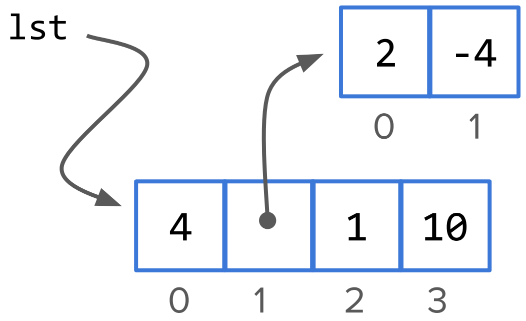
```
def memory(n):
    def update(fn):
        # Fill this in
    return update
```

Mutable Lists

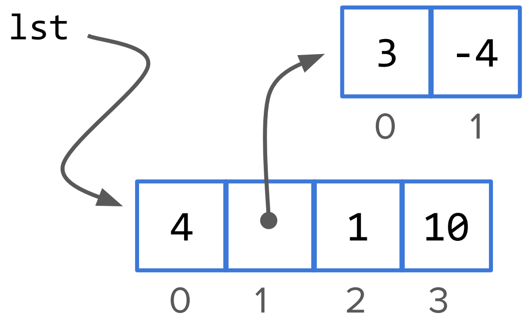
Mutation

A list is a **mutable** object, meaning that we can modify its value!

Specifically, we can add, remove, or change the elements of a list.



```
lst[1][0] = 3
```



List mutation methods

`append(el)`: adds `el` to the end of the list

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.append(6)
>>> lst
[1, 2, 3, 4, 5, 6]
```

`insert(i, el)`: inserts `el` at index `i` and shifts the rest of the elements over

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.insert(2, 2.5)
>>> lst
[1, 2, 2.5, 3, 4, 5]
```

remove(el): removes the first occurence of `el` from the list

```
>>> lst = [1, 2, 3, 4, 2, 5]
>>> lst.remove(2)
>>> lst
[1, 3, 4, 2, 5]
```

pop(i): removes and returns the element at index `i`

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.pop(3)
4
>>> lst
[1, 2, 3, 5]
```

List mutation methods summary

These functions all *mutate* the original list. They *do not* create or return new lists.

- **`append(el)`:** adds `el` to the end of the list
- **`insert(i, el)`:** inserts `el` at index `i` and shifts the rest of the elements over
- **remove(el):** removes the first occurence of `el` from the list
- **pop(i):** removes and returns the element at index `i`

Use *dot notation* to call these methods.

```
lst.append(3)
```

Try it out! Problem 2.2 - `add_this_many`

Dictionaries

Dictionaries

Dictionaries map keys to values.

Keys must be unique and immutable.

Values can be any data type and can be repeated.

Like lists, dictionaries are *mutable*!

Mutating dictionaries

Add a binding

```
>>> d = {1: 2, 3: 4}
>>> d[5] = 6
>>> d
{1: 2, 3: 4, 5:6}
```

Change a binding

```
>>> d = {1: 2, 3: 4}
>>> d[3] = 5
>>> d
{1: 2, 3: 5}
```

Remove a binding

```
>>> d = {1: 2, 3: 4}
>>> del d[1]
>>> d
{3: 4}
```

Remove a binding and **return** its value

```
>>> d = {1: 2, 3: 4}
>>> d.pop(3)
4
>>> d
{1: 2}
```

Try it out! Problem 3.2 - `group_by`, **Problem 3.3** - `replace_all_deep`

Appendix

Name lookup

Recall rules for name lookup:

1) Look for the name in the current frame. If it's there, use the value bound to it.
2) If the name isn't there, look in the current frame's *parent frame*.
3) Follow the chain of parent frames until the name is found.
4) If you reach the Global frame without finding the name, a `NameError` is raised.

In short, by default ***you are able to access variables in parent frames*** (unless that name exists in the current frame).

Assignment statements

Recall rules for assignment statements:

1) Evaluate the expression on the RHS of the `=` sign.
2) Look for the name given on the LHS in the current frame.
    - If the name is there, remove the current binding.
    - If the name is not there, add it.
3) Bind the value to the name.

Note the difference from name lookup: if the name doesn’t exist, do ***not*** look in parent frames. Simply add the name to the current frame!

In short, by default ***you are not able to modify variables in parent frames***.

New assignment statement rules

The first step is still to evaluate the expression on the RHS of the `=` sign.

The next steps depend on whether or not the given name is local or nonlocal.

Name is **local**. Use regular rules:

1) Look for the name in the current frame. If the name is not there, add it.
2) Bind the value to the name, removing any old binding if necessary.

Name is **nonlocal**. Use special rules:

1) Use *nonlocal* name lookup rules to find the binding. If the name isn't found before the global frame is reached, a SyntaxError is raised.
2) Replace the old binding of the name with the new value.

Nonlocal errors

Global variables cannot be modified by using nonlocal.

```
x = 10
def foo():
    nonlocal x
    x = 20
    return x

foo()
```

This code errors because Python cannot find any nonlocal variables named x.

A name can only be either local or nonlocal, not both.

```
def foo():
    x = 10
    def bar(x):
        nonlocal x
        x = 20
        return x
    return bar

foo()(30)
```

This code errors because x is already a local variable (a parameter) in bar's frame.