CS 61A
Spring 2018

Nonlocals & Mutation

Discussion 4: February 21, 2018

1 Nonlocal

Until now, you've been able to access variables in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a variable in the parent frame outside the current frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal variable
        num = num + 1 # modifies num in the stepper frame
        return num
    return step
```

However, there are two important caveats with `nonlocal` variables:

- **Global variables** cannot be modified using the `nonlocal` keyword.
- **Variables in the current frame** cannot be overridden using the `nonlocal` keyword. This means we cannot have both a local and nonlocal variable with the same names in a single frame.

Questions

1.1 Draw the environment diagram for the following code.

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

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

1.2 (Fall 2016) Draw the environment diagram for the following code.

```
lamb = 'da'
def da(da):
    def lamb(lamb):
        nonlocal da
        def da(nk):
            da = nk + ['da']
            da.append(nk[0:2])
            return nk.pop()
    da(lamb)
    return da([[1], 2]) + 3

da(lambda da: da(lamb))
```

1.3 Write a function that takes in a value `x` and updates and prints the result based on input functions.

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

2 Mutable Lists

Let's imagine you order a mushroom and cheese pizza from La Val's, and that they represent your order as a list.

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, La Val's would have to build an entirely new list to add onions:

```
>>> pizza2 = pizza1 + ['onions'] # creates a new python list
>>> pizza2
['cheese', mushrooms', 'onions']
>>> pizza1 # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all La Val's had to do was add onions on top of `pizza1` instead of making an entirely new `pizza2`.

Python actually allows you to *mutate* some objects, includings lists and dictionaries. Mutability means that the object's contents can be changed. So instead of building a new `pizza2`, we can use `pizza1.append('onions')` to mutate `pizza1`.

```
>>> pizza1.append('onions')
>>> pizza1
['cheese', 'mushrooms', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created. We can use the familiar indexing operator to mutate a single element in a list. For instance `lst[4]='hello'` would change the fifth element in `lst` to be the string `'hello'`. In addition to the indexing operator, lists have many mutating methods. List *methods* are functions that are bound to a specific list. Some useful list methods are listed here:

1. `append(el)` adds `el` to the end of the list
2. `insert(i, el)` insert `el` at index `i` (does not replace element but adds a new one)
3. `remove(el)` removes the first occurrence of `el` in list, otherwise errors
4. `pop(i)` removes and returns the element at index `i`

Questions

2.1 What would Python display? It may be helpful to draw the box and pointers diagrams to the right in order to keep track of the state.

(a)
```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst1 == lst2      # compares each value
```

(b)
```
>>> lst1 is lst2      # compares references
```

(c)
```
>>> lst2 = lst1
>>> lst1.append(4)
>>> lst1
```

(d)
```
>>> lst2
```

(e)
```
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
```

(f)
```
>>> lst1
```

(g)
```
>>> lst2
```

(h)
```
>>> lst2 is lst1
```

2.2 Write a function that takes in a value `x`, a value `el`, and a list and adds as many `el`'s to the end of the list as there are `x`'s. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

2.3 Write a function that takes in a list and reverses it *in place*, i.e. mutate the given list itself, instead of returning a new list.

```
def reverse(lst):
    """ Reverses lst in place.
    >>> x = [3, 2, 4, 5, 1]
    >>> reverse(x)
    >>> x
    [1, 5, 4, 2, 3]
    """
```

3 Dictionaries

Dictionaries are data structures which map keys to values. Dictionaries in Python are unordered, unlike real-world dictionaries — in other words, key-value pairs are not arranged in the dictionary in any particular order. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary can be any *immutable* value, such as numbers, strings, and tuples.[1] Dictionaries themselves are mutable; we can add, remove, and change entries after creation. There is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed.

To access the value of `dictionary` at `key`, use the syntax `dictionary[key]`.

Element selection and reassignment work similarly to sequences, except the square brackets contain the key, not an index.

- To add `val` corresponding to `key` *or* to replace the current value of `key` with `val`:

  ```
  dictionary[key] = val
  ```

- To iterate over a dictionary's keys:

  ```
  for key in dictionary: #OR for key in dictionary.keys()
      do_stuff()
  ```

- To iterate over a dictionary's values:

  ```
  for value in dictionary.values():
      do_stuff()
  ```

- To iterate over a dictionary's keys and values:

[1] To be exact, keys must be *hashable*, which is out of scope for this course. This means that some mutable objects, such as classes, can be used as dictionary keys.

```
for key, value in dictionary.items():
    do_stuff()
```

- To remove an entry in a dictionary:

```
del dictionary[key]
```

- To get the value corresponding to `key` and remove the entry:

```
dictionary.pop(key)
```

Questions

3.1 What would Python display?

```
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'ditto': 25, 'mew': 151}

>>> 'mewtwo' in pokemon



>>> len(pokemon)



>>> pokemon['ditto'] = pokemon['jolteon']
>>> pokemon[('diglett', 'diglett', 'diglett')] = 51
>>> pokemon[25] = 'pikachu'
>>> pokemon




>>> pokemon['mewtwo'] = pokemon['mew'] * 2
>>> pokemon



>>> pokemon[['firetype', 'flying']] = 146
```

Note that the last example demonstrates that dictionaries cannot use other mutable data structures as keys. However, dictionaries can be arbitrarily deep, meaning the *values* of a dictionary can be themselves dictionaries.

3.2 Write a function that takes in a sequence `s` and a function `fn` and returns a dictionary.

The values of the dictionary are lists of elements from `s`. Each element `e` in a list should be constructed such that `fn(e)` is the same for all elements in that list. Finally, the key for each value should be `fn(e)`.

```
def group_by(s, fn):
    """
    >>> group_by([12, 23, 14, 45], lambda p: p // 10)
    {1: [12, 14], 2: [23], 4: [45]}
    >>> group_by(range(-3, 4), lambda x: x * x)
    {0: [0], 1: [-1, 1], 4: [-2, 2], 9: [-3, 3]}
    """
```

3.3 Write a function that takes in a deep dictionary `d` and replace all occurences of `x` *as a value (not a key)* with `y`.

Hint: You will need to combine iteration and recursion. Also, for a dictionary `z`, `type(z) == dict` will evaluate to `True`.

```
def replace_all_deep(d, x, y):
    """
    >>> d = {1: {2: 'x', 'x': 4}, 2: {4: 4, 5: 'x'}}
    >>> replace_all_deep(d, 'x', 'y')
    >>> d
    {1: {2: 'y', 'x': 4}, 2: {4: 4, 5: 'y'}}
    """
```