Lab 8: Mutation

Due at 9:00pm on 10/25/2018.

Starter Files

Download lab08.zip (lab08.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the OK autograder.

Submission

When you are done, submit the lab by uploading the `lab08.py` file to okpy.org (https://okpy.org). You may submit more than once before the deadline; only the final submission will be graded.

Introduction to Mutation

A *mutable* data structure is any data structure that can be changed after it is created. You've already dealt with two mutable data structures: lists and dictionaries.

```
>>> x = [1, 2, 3]
>>> x[0] = 4
>>> x
[4, 2, 3]   # x changed!
>>> d = {'name': 'Rhonda', 'age': 21, 'major': 'Business'}
>>> d['major'] = 'Data Science'
>>> d
{'name': 'Rhonda', 'age': 21, 'major': 'Data Science'}  # Rhonda is an Data Science major now!
```

Unlike lists or dictionaries, tuples are *immutable* data structures. This means that once they are created, they can't be changed. For example, try this:

```
>>> x = (1, 2, 3)
>>> x[0] = 4
```

This will cause TypeError complaining that tuples don't "support item assignment." In other words, you can't change the elements in a tuple because tuples are immutable. Once a tuple is created, it can't be changed!

Question 1

What does Python print? Think about these before typing it into an interpreter!

```
>>> lst = [1, 2, 3, 4, 5, 6]
>>> lst[4] = 1
>>> lst
______

>>> lst[2:4] = [9, 8]
>>> lst
______

>>> lst[3] = ['hi', 'bye']
>>> lst
______
```

```
>>> lst[3:] = ['oski', 'bear']
>>> lst
______

>>> lst[1:3] = [2, 3, 4, 5, 6, 7, 8]
>>> lst
______
```

```
>>> lst == lst[:]
______

>>> lst is lst[:]
______

>>> a = lst[:]
>>> a[0] = 'oogly'
>>> lst
______
```

```
>>> lst = [1, 2, 3, 4]
>>> b = ['foo', 'bar']
>>> lst[0] = b
>>> lst
______

>>> b[1] = 'ply'
>>> lst
______

>>> b = ['farply', 'garply']
>>> lst
______

>>> lst[0] = lst
>>> lst
______

>>> lst[0][0][0][0][0]
______
```

Question 2: Map

Write a function that maps a function on the given list. Be sure to mutate the original list.

> This function should NOT return anything. This is to emphasize that this function should utilize mutability.

```
def map(fn, lst):
    """Maps fn onto lst using mutation.
    >>> original_list = [5, -1, 2, 0]
    >>> map(lambda x: x * x, original_list)
    >>> original_list
    [25, 1, 4, 0]
    """
    "*** YOUR CODE HERE ***"
```

Use OK to test your code:

```
python3 ok -q map --local
```

Question 3: reverse-todo

During the mini lecture, you saw a to-do list ADT that returns a function that adds items to the list. Say our priorities change and we want to reverse the order of the list!

Rewrite the constructor so that it also returns another function that reverses the todo list. Be sure to mutate the original list!

> This function should NOT return anything. This is to emphasize that this function should utilize mutability.

```
def todo():
    """Returns add and reverse, which add to and reverse the list
    >>> add, get_list, reverse = todo()
    >>> add("clean")
    >>> add("homework")
    >>> add("cook")
    >>> add("sleep")
    >>> get_list()
    ['clean', 'homework', 'cook', 'sleep']
    >>> reverse()
    >>> get_list()
    ['sleep', 'cook', 'homework', 'clean']
    >>> add("wake up")
    >>> get_list()
    ['sleep', 'cook', 'homework', 'clean', 'wake up']
    >>> reverse()
    >>> get_list()
    ['wake up', 'clean', 'homework', 'cook', 'sleep']
    """
    lst = []
    def get_list():
        return lst
    def add(item):
        lst.append(item)
    def reverse():
        "*** YOUR CODE HERE ***"

    return add, get_list, reverse
```

Use OK to test your code:

```
python3 ok -q todo --local
```

Mutation and ADT's

Question 4: mailbox

Mutation is crucial for ADTs. Recall the bank account example given in lecture. The bank account ADT can be deposited to and withdrawn from. When we deposit or withdraw, we want to change the state of the bank account so that the withdrawal or deposit will be reflected next time we access it.

Below, we can see the power of mutability which allows us to change the name and amount of money in `account_1`.

```
>>> account_1 = account("Jessica", 10)
>>> get_account_name(account_1)
Jessica
>>> change_account_name(account_1, "Ting")
>>> get_account_name(account_1)
Ting
>>> deposit(account_1, 20)
>>> get_account_savings(account_1)
30
>>> withdraw(account_1, 10)
>>> get_account_savings(account_1)
20
```

During the mini-lecture, we introduced the new concept of a constructor returning the very functions we want to run on the object! This kind of abstraction is very common in the programming world.

Let's apply this concept to a new ADT: a mailbox. The mail box's internal representation is a dictionary that holds key value pairs corresponding to a person, and a list of their mail. Below is an example of a TA mailbox.

```
{"Jessica":["receipt", "worksheets"], "Alex":["worksheets", "form", "bills"], "Andrew":["paycheck"
```

The `_get_mail(mailbox, name)` function, given a mailbox, should return the mail corresponding to the name argument given. The postman can also put in mail by using the `_deliver_mail(mailbox, name, mail)` function. You may have noticed that the function names are a bit strange: they each have an underscore in the beginning. We use this kind of syntax when a function is "hidden", meaning they are hidden to the outside. Remember, our goal is for the `mailbox` constructor to return all functions that the mailbox uses.

Please provide implementations for these functions. Make sure to mutate the dictionary! (You can assume mail will always be in a list.)

After doing that, implement the constructor so that it returns `get_mail` and `deliver_mail` such that when called, they will perform the function on the original mailbox. **Hint: use _get_mail and _deliver_mail**

```
def mailbox():
    """
    >>> get_mail, deliver_mail = mailbox()
    >>> get_mail("Amir")
    >>> deliver_mail("Amir", ["postcard"])
    >>> get_mail("Amir")
    ['postcard']
    >>> get_mail("Amir")
    >>> deliver_mail("Ting", ["paycheck", "ads"])
    >>> get_mail("Ting")
    ['paycheck', 'ads']
    >>> deliver_mail("Ting", ["bills"])
    >>> get_mail("Ting")
    ['bills']
    >>> deliver_mail("Alex", ["survey"])
    >>> get_mail("Alex")
    ['survey']
    >>> get_mail("Alex")
    >>> get_mail("John")
    >>> deliver_mail("John", ["postcard", "paycheck"])
    >>> deliver_mail("John", ["ads"])
    >>> get_mail("John")
    ['postcard', 'paycheck', 'ads']
    """
    "*** YOUR CODE HERE ***"

    return get_mail, deliver_mail

def _deliver_mail(mailbox, name, mail):
    "*** YOUR CODE HERE ***"


def _get_mail(mailbox, name):
    "*** YOUR CODE HERE ***"
```

Use OK to test your code:

```
python3 ok -q mailbox --local
```

Unexpected Behavior

Question 5: The Mutation Mystery

You just got a new job as the official code detective of UC Berkeley. You just got your first assignment on the job! There have been 3 reported cases of unusual list behavior from students who are writing python code for CS88. You're no python expert yet, but you've been studying up on the details of how python works. You're ready to investigate this mystery!

Case #1

The first case was brought to you by Alice Listmaker. Alice created two identical lists, `x` and `y`. She then wanted to modify only `y`. However, she was surprised to see that `x` had been modified too! Use the command below to see what happened.

```
python3 ok -q case1 -u --local
```

Hmm... what could have gone wrong here? Here are some clues that will help you figure this out:

1. This environment diagram (http://tinyurl.com/jzcyllh)
2. Slicing, such as `x[:]`, creates a new list.
3. The list function, `list(x)`, creates a new list.

You think you have found a fix using slicing and the list function. Use the command below to confirm that you have solved the mystery!

```
python3 ok -q case1_solved -u --local
```

Case #2

The second report of unusual list behavior came from Alice's cousin, Bob Listmaker. Unlike Alice, Bob knew about slicing and the list function. However, he was still a victim of mutation. Let's take a look at what happened to Bob's lists using the command below.

```
python3 ok -q case2 -u --local
```

Here are some clues as to what caused this error:

1. Slicing and the list function do create new lists, but do not deep copy the values.
2. This environment diagram (http://tinyurl.com/jasot7o)
3. This article on shallow vs. deep copy (http://www.python-course.eu/deep_copy.php)

You have figured out that Bob needs to perform a deep copy on his lists. You decide to write a `deep_copy` function for Bob to use. Fill in the body of the function in lab06.py. Do NOT use the copy module shown in the article, you need to write your own function! Test your code using the following command.

```
python3 ok -q deep_copy --local
```

Case #3

After solving those first two cases, you're ready to tackle the final challenge. This case was brought to you by Eve (she doesn't have a list-related last name, sorry). Eve knew that she was not properly deep copying her lists. However, she could not figure out why this only caused problems some of the time. Let's investigate further.

```
python3 ok -q case3 -u --local
```

Your final clues:

1. This environment diagram (http://tinyurl.com/zmrwvoj)
2. Lists contain pointers to other lists.
3. There are no pointers to strings or numbers, the values are stored directly.

Once you have solved this last case, write out an explanation for the unusual behavior in Eve'e code in `case3_explanation` in lab06.py.

Congratulations, you have solved the mutation mystery!

Additional unusual list behavior

1. Adding elements to a list using `+` creates a new list.
2. Adding elements to a list using `+=` does not create a new list.
3. Adding elements to a list using `.append()` does not create a new list.

Take a look at this environment diagram (http://tinyurl.com/hzb6vz8) to see an example of this. Pay close attention to the difference between using `+` and `+=` in the example with `y` and `z`.