Lab 2: Python, Scheme, and Value/Reference Semantics

Learning objectives:

- Understand the difference between reference semantics and value semantics
- Review writing recursive code
- Gain experience writing Scheme code

Most of the code you write in this course will be in Python or Scheme, both of which have reference semantics. This lab will help you adjust to that. This lab also gives you practice in writing recursive code as well as reading a language specification, both of which will be necessary for projects 2 through 5.

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab02/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Value and reference semantics.* Consider two fragments of code in C++ and Python:

C++	Python

`class Foo { ... };`
 	
`int main() {`
`  Foo x = Foo();`
`  Foo y = x;`
`  Foo &z = y;`
`  x = Foo();`
`}`
	`class Foo:`
	`  ...`
	
	`x = Foo()`
	`y = x`
	`z = y`
	`x = Foo()`

   For each program, draw a memory diagram that represents the state of the program just before it terminates.

2. *Reference semantics and swap.* Alyssa P. Hacker is trying to write a Python function to swap the contents of two lists. Here is what she has tried so far:

```
def swap_contents(list1, list2):
    """Modify list1 to contain the contents of list2 and vice versa.

    >>> list1 = [1, 2, 3]
    >>> list2 = ['hello', 'world']
    >>> swap_contents(list1, list2)
    >>> list1
    ['hello', 'world']
    >>> list2
    [1, 2, 3]
    """
    tmp = list1
    list1 = list2
    list2 = tmp
```

   a) Explain why this code does not swap the contents of the two lists.

   b) Modify the code so that it does swap the contents. Starter code for this problem can be found in `swap_contents.py`. To run the doctests on your implementation:

```
$ python3 -m doctest swap_contents.py
```

3. *Recursion.* Implement the `flatten()` function in Python. Given a nested list structure, it produces a new flattened list that contains all elements from the original structure. Use `isinstance(item, list)` to determine if `item` is a list.

```
def flatten(item):
    """Produce a flattened version of the given structure.

    If item is a list, returns a new, flat list containing all
    the items contained within list or any of its elements. If
    item is not a list, returns a list containing item.

    >>> flatten(3)
    [3]
    >>> items = [1, 2, 3]
    >>> flattened = flatten(items)
    >>> flattened
    [1, 2, 3]
    >>> flattened is items  # verify flattened is a new list
    False
    >>> flatten([[], [1, 2], [[3, [4, 5]], 6], 7])
    [1, 2, 3, 4, 5, 6, 7]
    """
    # your code here
```

Starter code for this problem can be found in `flatten.py`. To run the doctests on your implementation:

```
$ python3 -m doctest flatten.py
```

4. *List manipulation.* Implement the following procedures in `lists.scm`. Simple tests for each procedure are included in the `lists.scm` starter file. To run the tests:

```
$ plt-r5rs lists.scm
```

You may only use the following special forms and primitive procedures: `define` at global scope, `let`, `let*`, `lambda`, `if`, `cond`, `and`, `or`, `quote`, `null?`, `list?`, `not`, `cons`, `car`, `cdr`, `list`, and `equal?`. Your functions do not have to be tail recursive.

a) Implement the `list-append` procedure, which takes two lists as parameters and returns a new list that is the second list appended to the first list.

```
(list-append '(1 2) '(2 3))  ; returns (1 2 2 3)
(list-append '() '(2 3))     ; returns (2 3)
```

b) Now implement the `deep-reverse` procedure, which takes a list as a parameter and returns a new list that is the reverse of the input list. If the input list contains a list, that list must be reversed as well. You may use your implementation of `list-append`, if necessary.

```
(deep-reverse '(1 2 3 4 5))    ; returns (5 4 3 2 1)
(deep-reverse '((2 4) (1 3)))  ; returns ((3 1) (4 2))
```

c) Implement the `contains` procedure, which takes a list and a value and returns true if the value is found in the list and false otherwise.

```
(contains '(1 2 4) 4)  ; returns #t
(contains '(1 2 4) 3)  ; returns #f
```