C M S I 3 8 6

Quiz 1 Answers

1. Here is the direct approach, clear as can be, though not super-cool:

```
/*
 * mapIterate(f, [a0, a1, a2, ...]) returns [a0, f(a1), f(f(a2)), ...]
 */
var mapIterate = function (f, a) {
    var result = [];
    for (var i = 0, n = a.length; i < n; i += 1) {
        var x = a[i];
        for (var j = 0; j < i; j += 1) {
            x = f(x);
        }
        result.push(x);
    }
    return result;
}
```

   Another possibility is this recursive one-liner, where I even gave the function a short name:

```
function mi(f, a) {return a.length === 0 ? [] : [a[0]].concat(mi(f, a.slice(1).map(f)));}
```

2. Simple and direct:

```
def map_iterate(a, f):
  result = []
  for i, x in enumerate(a):
      for _ in range(i):
          x = f(x)
      result.append(x)
  return result
```

   Here's the dense one-liner:

```
def mi(a, f): return [a[0]] + mi([f(x) for x in a[1:]], f) if a else []
```

3. The usual approaches do not work in Python 2. You cannot use a class because there is no way to hide the state completely. Closures with the shared integer as the local variable in the wrapping function don't work because those variables cannot be written to, only read. So you have to be really clever to get this! Here you go:

```
def make_the_two_functions():
    state = {'x': 0}
    def a(): state['x'] -= 10; return state['x']
    def b(): state['x'] = 3 * abs(state['x']); return state['x']
    return (a, b)

f, g = make_the_two_functions()
```

There is a more direct way to do this in Python 3; we'll cover this in class some day.

4. I would just make an array with the functions, hiding the state in a local variable in an enclosing function.

```
var myFunctionPair = (function () {
    var x = 0;
    return [
        function () {return x -= 10;},
        function () {return x = 3 * Math.abs(x);}
    ];
}());

// The two functions are myFunctionPair[0] and myFunctionPair[1]
```

If you don't like an array, you can assign to global variables inside a function:

```
var f, g;
(function () {
    var x = 0;
    f = function () {return x -= 10;}
    g = function () {return x = 3 * Math.abs(x);}
}());
```

You know, you actually don't *have* to declare `f` and `g` at the top-level, but automatically bringing them into existence with assignments inside a function is evil.

5. Well if we need the one-liner, we should use `match`. It's not efficient space-wise, because it builds up a whole array, which is stupidly wasteful. But the code looks cool.

```
function countOfAsciiVowels(s) {
    return (s.match(/[aeiou]/ig) || []).length;
}
```

Okay, well, not that cool because JavaScript's `match` is a WAT! It actually returns `null` if there are no matches, instead of an empty array. But we compensated for that.

6. If you want to use the regular expression technique from the last problem, this is how you do it in Python:

```
import re

def count_of_ascii_vowels(s):
    return len(re.findall('[aeiou]', s, re.IGNORECASE))
```

But any one-liner is okay. How about this one?

```
def count_of_ascii_vowels(s):
    return len([c for c in s if c in 'aeiouAEIOU'])
```

There are other ways, too.

7. After this Python 2 script is run, *b* is [0,1,2,3,4,5,6,7,8,9]. Here's why. First, we create the variable *a*, which becomes a list of 10 elements, each with the value `lambda: i` (a function of no arguments returning the value of *i*). After we make *a*, the variable *i* has the value 9. Next we build up a list step by step as follows:
   - i = 0, a[0]() = (lambda: i)() = 0, so the first element is 0.
   - i = 1, a[1]() = (lambda: i)() = 1, so the second element is 1.

- i = 2, a[2]() = (lambda: i)() = 2, so the third element is 2.
- ...
- i = 9, a[9]() = (lambda: i)() = 9, so the last element is 9.

Now as a followup question, see if you can determine how the output would have been different if we had changed the second line of the script to:

```
b = [a[k] for k in range(10)]
```

Cool, eh?

8. After this JavaScript script is run, *b* is [10,10,10,10,10,10,10,10,10,10]. Here's why. First, we create the variable *a*, which becomes a list of 10 elements, each with the value `function(){return i}` (a function of no arguments returning the value of *i*). After we make *a*, the variable *i* has the value 10. Next we build up a list step by step as follows:
   - j = 0, a[0]() = function(){return i}() = 10, so the first element is 10.
   - j = 1, a[1]() = function(){return i}() = 10, so the second element is 10.
   - j = 2, a[2]() = function(){return i}() = 10, so the third element is 10.
   - ...
   - j = 9, a[9]() = function(){return i}() = 10, so the last element is 10.