Review Questions

What Would Python Print?
Tuples, Lists, Dictionaries

```
>>> a = (1, 2, 3, 4)
>>> a[::-1]
_________
>>> a = a[:0:-1]
>>> a
_________
>>> b = [1, 2, 3, 4]
>>> b[3] = a[1:]
>>> b
_________
>>> b[3][0] = a[:-2]
_________
```

What Would Python Print?
Tuples, Lists, Dictionaries

```
>>> a = (1, 2, 3, 4)
>>> a[::-1]
(4, 3, 2, 1)
>>> a = a[:0:-1]
>>> a
(4, 3, 2)
>>> b = [1, 2, 3, 4]
>>> b[3] = a[1:]
>>> b
[1, 2, 3, (3, 2)]
>>> b[3][0] = a[:-2]
TypeError: 'tuple' object does not support item assignment
```

Coding Practice
Recursion

Write a function `deep_map(f, lst)` which applies a one-argument function onto every element in the given list. If an element is itself a list, then you should recursively apply the function onto each of its elements. You should NOT return anything—instead, mutate the original list (and any nested lists).

```
def deep_map(f, lst):
    """
    >>> lst = [1, 2, [3, 4, [5], 6], 7, [], 8]
    >>> deep_map(lambda x: x * x, lst)
    >>> lst
    [1, 4, [9, 16, [25], 36], 49, [], 64]
    """
```

Coding Practice
Recursion

```
def deep_map(f, lst):
    if lst:
        last = lst.pop()
        if type(last) is list:
            deep_map(f, last)
        else:
            last = f(last)
        deep_map(f, lst)
        lst.append(last)
```

Coding Practice
Nonlocal

Write a function that returns a function that returns the last thing it received (the first time it's called, it returns '...')

```
>>> slowpoke = make_delayed_repeater()
>>> slowpoke("hi")
...
>>> slowpoke("hello?")
hi
>>> slowpoke("stop repeating what I'm saying")
hello?
```

Coding Practice
Nonlocal

```
def make_delayed_repeater():
    last = '...'
    def delayed_repeater(phrase):
        nonlocal last
        last, to_return = phrase, last
        return to_return
    return delayed_repeater
```

Coding Practice
Equality vs. Identity

```
>>> l1, l2 = list(range(5)), list(range(5))
>>> l1 == l2

_____
>>> l1 is l2

_____
>>> l2 = l1
>>> l1 is l2

_____
>>> d1, d2 = {1: 3, 5: 7}, {5: 7, 1: 3}
>>> d1 == d2

_____
>>> d1 is d2

_____
```

Coding Practice
Equality vs. Identity

```
>>> l1, l2 = list(range(5)), list(range(5))
>>> l1 == l2
True
>>> l1 is l2
False
>>> l2 = l1
>>> l1 is l2
True
>>> d1, d2 = {1: 3, 5: 7}, {5: 7, 1: 3}
>>> d1 == d2
True
>>> d1 is d2
False
```

What Would Python Print?
OOP

```
class Foo(object):
    baz = 0
    bar = 'something'
    def __init__(self):
        self.bar = 'anything'
        self.__qux = self.baz
        Foo.baz += 1

    @property
    def foo(self):
        return self.__qux
```

```
>>> a = Foo()
>>> a.bar

_____
>>> a.__qux

_____
>>> a.foo

_____
>>> Foo.baz

_____
>>> b = Foo()
>>> b.foo

_____
```

What Would Python Print?
OOP

```
class Foo(object):
    baz = 0
    bar = 'something'
    def __init__(self):
        self.bar = 'anything'
        self.__qux = self.baz
        Foo.baz += 1

    @property
    def foo(self):
        return self.__qux
```

```
>>> a = Foo()
>>> a.bar
'anything'
>>> a.__qux
AttributeError
>>> a.foo
0
>>> Foo.baz
1
>>> b = Foo()
>>> b.foo
1
```

Coding Practice
Trees

Given a binary tree (with left and right), implement a function sum_tree, which adds up all the items (assumed to be numbers) in the tree.

```
def sum_tree(tree):
    """ Your Code Here """
```

Coding Practice
Trees

```
def sum_tree(tree):
    if tree is None:
        return 0
    else:
        left = sum_tree(tree.left)
        right = sum_tree(tree.right)
        return tree.entry + left + right
```

Coding Practice
Trees

Implement a function same_shape, which takes two binary trees and checks if they have the same shape (not if they have the same items).

```
def same_shape(tree1, tree2):
   """ Your Code Here """
```

Coding Practice
Trees

```
def same_shape(tree1, tree2):
    if tree1 is None and tree2 is None:
        return True
    elif tree1 is None or tree2 is None:
        return False
    left = same_shape(tree1.left, tree2.left)
    right = same_shape(tree1.right, tree2.right)
    return left and right
```

Orders of Growth

```
def foo(n):
  if n <= 1000:
      return n
  for i in range(n):
      print(i)
  for i in range(n*n):
      print(i)
```
Θ(?)

```
def bar(n):
  if n < 3:
      return n
  return bar(n // 3)
```
Θ(?)

```
def blip(n):
  for i in range(n//2):
      bar(n)
```
Θ(?)

```
def zeta(n):
    if n <= 1:
        return 1
    return zeta(n-1) + \
               zeta(n-2)
```
Θ(?)

Orders of Growth

```
def foo(n):
  if n <= 1000:
      return n
  for i in range(n):
      print(i)
  for i in range(n*n):
      print(i)
```
Θ(n^2)

```
def bar(n):
  if n < 3:
      return n
  return bar(n // 3)
```
Θ(log n)

```
def blip(n):
  for i in range(n//2):
      bar(n)
```
Θ(n*log n)

```
def zeta(n):
    if n <= 1:
        return 1
    return zeta(n-1) + \
               zeta(n-2)
```
Θ(2^n)

Coding Practice
Scheme

Write a function append that takes in a list a value and returns a list with that value appended.

```
(define (append lst v)
        'yourcodehere)
```

Coding Practice
Scheme

```
(define (append lst v)
        (cond ((null? lst)
         (list v))
        (else (cons (car lst) (append (cdr lst) v))))
```

Coding Practice
Scheme

Implement the insert function in Scheme, which inserts item at index, if index is within the bounds of the list, or at the end of the list otherwise.

```
(define (insert lst item index)
        'yourcodehere)
```

Coding Practice
Scheme

```
(define (insert lst item index)
        (cond ((null? lst)
               (list item))
              ((= index 0)
               (cons item lst))
              (else (cons (car lst)
               (insert (cdr lst) item (- index 1))))))
```

Coding Practice
Extras

Write a function **find_path** that takes in a dictionary, **friends** mapping every person to the list of their friends, and returns whether it is possible to move from the person **start** to the person **finish** by following friend relationships.

```
def find_path(friends, start, finish):
  """
  >>> allfriends = {"Steven" : ["Eric"],
                    "Eric": ["Mark", "Jeffrey", "Brian"],
                    "Albert" : ["Robert", "Andrew", "Leonard"]}
  >>> find_path(allfriends, "Eric", "Robert")
  True
  >>> find_path(allfriends, "Steven", "Robert")
  False
  """
```

Coding Practice
Extras

```
def find_path(friends, start, finish):
  def find_path2(visited, start):
      if start == finish:
            return True
      if start in friends:
            for vertex in friends[start]:
                  if vertex not in visited:
                        visited.append(vertex)
                        if find_path2(visited, vertex):
                              return True
      return False
  return find_path2([], start)
```

Coding Practice
Extras

Implement a function flatten that takes in a scheme list and removes any nested lists, replacing them with their elements. (Does not have to work for lists nested in nested lists)

```
STK> (define a (list 1 (list 2 3 4) 5 6 (list 7 8)))
STK> a
(1 (2 3 4) 5 6 (7 8))
STK> (flatten a)
(1 2 3 4 5 6 7 8)

(define (flatten lst)
        'yourcodehere)
```

Coding Practice
Extras

```
(define (flatten lst)
    (cond ((null? lst) lst)
          ((list? (car lst))(append (flatten (car lst))
  (flatten (cdr lst))))
          (else (cons (car lst) (flatten (cdr lst))))))
```