FINAL REVIEW 11

COMPUTER SCIENCE 61A

December 4, 2015

1 Rain, Rain, Go Away

1. For each row below, fill in the blanks in the output displayed by the interactive Python interpreter when the expression is evaluated. Expressions are evaluated in order, and expressions may affect later expressions.

```
>>> cats = [1, 2]
>>> dogs = [cats, cats.append(23), list(cats)]
>>> cats
```

Solution:
```
[1, 2, 23]
```

```
>>> dogs[1] = list(dogs)
>>> dogs[1]
```

Solution:
```
[[1, 2, 23], None, [1, 2, 23]]
```

```
>>> dogs[0].append(2)
>>> cats
```

Solution:
```
[1, 2, 23, 2]
```

```
>>> dogs[2].extend([list(cats).pop(0), 3])
>>> dogs[3]
```

Solution:
```
Index Error
```

```
>>> dogs
```

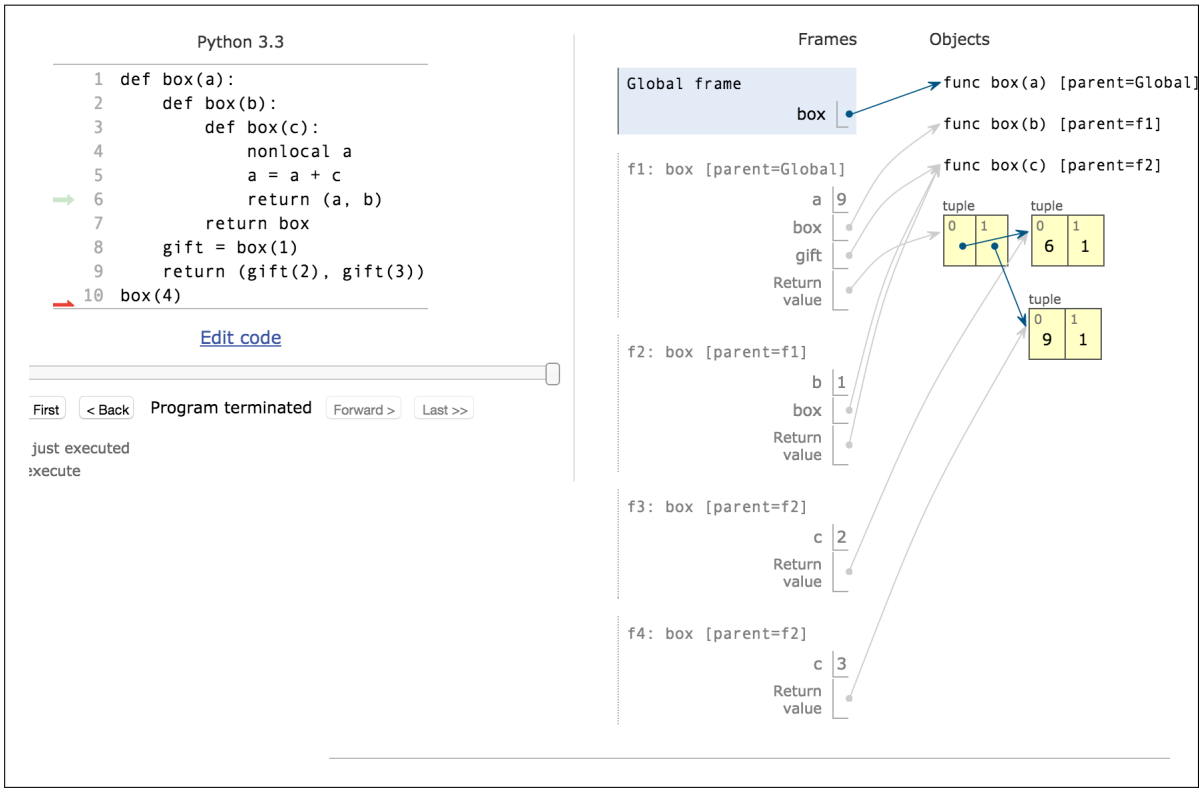
Solution:
```
[[1, 2, 23, 2], [[1, 2, 23, 2], None, [1, 2, 23, 1, 3]],
   [1, 2, 23, 1, 3]]
```

2 Gift in a Box

1. (Fall 2012) Draw the environment diagram.

```
def box(a):
    def box(b):
        def box(c):
            nonlocal a
            a = a + c
            return (a, b)
        return box
    gift = box(1)
    return (gift(2), gift(3))
box(4)
```

Solution: http://goo.gl/myL8Qq


Python 3.3
1 def box(a):
2 def box(b):
3 def box(c):
4 nonlocal a
5 a = a + c
6 return (a, b)
7 return box
8 gift = box(1)
9 return (gift(2), gift(3))
10 box(4)
Edit code
First
< Back
Program terminated
Forward >
Last >>
just executed
execute
Frames
Objects
Global frame
box
func box(a) [parent=Global]
func box(b) [parent=f1]
func box(c) [parent=f2]
f1: box [parent=Global]
a 9
box
gift
Return value
tuple
0 1
6 1
tuple
0 1
9 1
f2: box [parent=f1]
b 1
box
Return value
f3: box [parent=f2]
c 2
Return value
f4: box [parent=f2]
c 3
Return value

3 The Gift & The Recurse

1. The **quicksort** sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the **pivot** element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

   First, implement the `quicksort_list` function. Choose the first element of the list as the pivot. You may assume that all elements are distinct.

```
def quicksort_list(lst):
    """
    >>> quicksort_list([3, 1, 4])
    [1, 3, 4]
    """

    if _____:

        _____

    pivot = lst[0]

    less = _____

    greater = _____

    return _____
```

Solution:

```
def quicksort_list(lst):
    if len(lst) <= 1:
        return lst
    pivot = lst[0]
    less = [e for e in lst[1:] if e < pivot]
    greater = [e for e in lst[1:] if e > pivot]
    return list_quicksort(less) + [pivot] +
           list_quicksort(greater)
```

2. We can also use quicksort to sort linked lists! Implement the `quicksort_link` function, without constructing additional `Link` instances.

   You can assume that the `extend_links` function is already defined. It takes two linked lists and mutates the first so that it points to the second.

```
>>> l1, l2 = Link(1, Link(2)), Link(3, Link(4))
>>> l3 = extend_links(l1, l2)
>>> l3
Link(1, Link(2, Link(3, Link(4))))
>>> l1 is l3
True
```

```
def quicksort_link(link):
    """
    >>> s = Link(3, Link(1, Link(4)))
    >>> quicksort_link(s)
    Link(1, Link(3, Link(4)))
    """

    if ____________________________________________________:

        return link

    pivot, _______ = ____________________________________________

    less, greater = ____________________________________________

    while link is not Link.empty:

        curr, rest = link, link.rest

        if ____________________________________________________:

            ____________________________________________________

        else:

            ____________________________________________________

        link = ____________________________________________

    less = ____________________________________________

    greater = ____________________________________________

    ____________________________________________________

    return ____________________________________________
```

Solution:

```
def quicksort_link(link):
    if link is Link.empty or link.rest is Link.empty:
```

```
        return link
    pivot, link = link, link.rest
    less, greater = Link.empty, Link.empty
    while link is not Link.empty:
        curr, rest = link, link.rest
        if curr.first < pivot.first:
            less, curr.rest = curr, less
        else:
            greater, curr.rest = curr, greater
        link = rest
    less = quicksort_link(less)
    greater = quicksort_link(greater)
    pivot.rest = greater
    return extend_links(less, pivot)
```

4 Can You Take Me Higher?

1. (Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns the number of ways of reaching y from x by repeatedly incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9.

```
def inc(x):
    return x + 1

def double(x):
    return x * 2

def paths(x, y):
    """Return the number of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5) # inc(inc(3))
    1
    >>> paths(3, 6) # double(3), inc(inc(inc(3)))
    2
    >>> paths(3, 9) # E.g. inc(double(inc(3)))
    3
    >>> paths(3, 3) # No calls is a valid path
    1
    """
    if x > y:

        return _____

    elif x == y:

        return _____

    else:

        return _____
```

Solution:

```
def paths(x, y):
    if x > y:
```

```
        return 0
    elif x == y:
        return 1
    else:
        return paths(inc(x), y) + paths(double(x), y)
```

2. (Fall 2013) Fill in the blanks in the implementation of `pathfinder`, a higher-order function that takes an increasing function `f` and a positive integer `y`. It returns a function that takes a positive integer `x` and returns whether it is possible to reach `y` by applying `f` to `x` zero or more times. For example, 8 can be reached from 2 by applying `double` twice. A function `f` is *increasing* if $f(x) > x$ for all positive integers `x`.

```
def pathfinder(f, y):
    """
    >>> f = pathfinder(double, 8)
    >>> {k: f(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: False, 4: True, 5: False}
    >>> g = pathfinder(inc, 3)
    >>> {k: g(k) for k in (1, 2, 3, 4, 5)}
    {1: True, 2: True, 3: True, 4: False, 5: False}
    """
    def find_from(x):

        while _____________________________________________:

            ___________________________________________________

        return _____________________________________________

    _____________________________________________________________
```

Solution:

```
def pathfinder(f, y):
    def find_from(x):
        while x < y:
            x = f(x)
        return x == y
    return find_from
```

3. Write a generator function that yields functions that are repeated applications of a one-argument function `f`. The first function yielded should apply `f` 0 times (the identity function), the second function yielded should apply `f` once, etc.

```
def repeated(f):
    """
    >>> [g(1) for _, g in
    ...  zip(range(5), repeated(double))]
    [1, 2, 4, 8, 16]
```

```
    """

    g = ____________________________________

    while True:

        ____________________________________________

        ____________________________________________
```

Solution:

```
def repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = (lambda g: lambda x: f(g(x)))(g)
```

4. Ben Bitdiddle proposes the following alternate solution. Does it work?

```
def ben_repeated(f):
    g = lambda x: x
    while True:
        yield g
        g = lambda x: f(g(x))
```

5 Slim Shady

1. Implement `widest_level`, which takes a `Tree` instance and returns the elements at the depth with the most elements.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]

    while ____________________________________________________:

        ____________________________________________________

        __________ = sum(_____________________________________, [])

    return max(levels, key=_____________________________________)
```

Solution:

```
def widest_level(t):
    levels = []
    x = [t]
    while x:
        levels.append([t.entry for t in x])
        x = sum([t.branches for t in x], [])
    return max(levels, key=len)
```

6 Scheming With a Broken Heart

1. Consider the following Scheme tree data abstraction.

```
(define (make-tree entry children) (cons entry children))
(define (entry tree) (car tree))
(define (children tree) (cdr tree))
(define tree 'below-example)
;                 5
;        +--------+--------+
;        |        |        |
;        6        7        2
;     +--+--+     |     +--+--+
;     |     |     |     |     |
;     9     8     1     6     4
;                       |
;                       |
;                       3
```

Write a procedure `tree-sums` that takes a tree of numbers (like the one above) and outputs a list of sums from following each possible path from root to leaf.

Hint: You may find the `flatten` procedure helpful.

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

(define (tree-sums tree)

  (if _____________________________________________________________

      _____________________________________________________________

      (map (lambda (x) _____________________________________________)

      _____________________________________________________________)))
```

```
scm> (flatten '(0 (1) ((2)) (3 ((4)))))
(0 1 2 3 4)
scm> (tree-sums tree)
(20 19 13 16 11)
```

Solution:

```
(define (tree-sums tree)
  (if (null? (children tree))
      (list (entry tree))
      (map (lambda (x) (+ x (entry tree)))
           (flatten (map tree-sums (children tree))))))
```

7 Stream On

1. Implement the `append-stream` procedure, which takes in two streams and returns a stream with the two streams concatenated. (Note that if the first stream is infinite, the result will not contain any elements from the second stream.)

```
(define (append-stream s1 s2)
```

Solution:

```
(cond
  ((null? s1) s2)
  (else (cons-stream (car s1)
          (append-stream (cdr-stream s1) s2)))))
```

2. Now implement `subset-stream`, which takes in a normal Scheme list and returns a stream with every possible subset of that Scheme list.

```
(define (subset-stream lst)
```

Solution:

```
(cond
  ((null? lst) (cons-stream nil nil))
  (else (append-stream
          (map-stream (lambda (x) (cons (car lst) x))
                      (subset-stream (cdr lst)))
          (subset-stream (cdr lst))))))
```

8 Turning Tables

1. You're trying to re-organize your music library! The table `tracks` below contains song titles and the corresponding album. Create another table `tracklist` with two columns: the album and a comma-separated list of all songs from that album.

```
create table tracks as
  select "Human" as title, "The Definition" as album union
  select "Simple and Sweet", "The Definition"        union
  select "Paper Planes", "Translations Through Speakers";

create table tracklist as
  with

    songs(album, total) as (

      _____________________________________________________________

    ),

    _____________________________________________________ as (

      _____________________________________________________________

      _____________________________________________________________

      _____________________________________________________________

    )

    select _____________________________________________________

      where _____________________________________________________;
```

Solution:

```
create table tracklist as
  with
    songs(album, total) as (
      select album, count(*) from tracks group by album
    ),
```

```
    listing(album, songs_so_far, last_song,
       num_songs_so_far) as (
      select album, title, title, 1 from tracks union
      select l.album, songs_so_far || ', ' || title, title,
          num_songs_so_far + 1
       from tracks as t, listing as l
       where l.album = t.album and title > last_song
    )
    select l.album, l.songs_so_far
      from songs as s, listing as l
      where s.album = l.album and total = num_songs_so_far;
```

```
sqlite3> select * from tracklist order by album;
The Definition|Human, Simple and Sweet
Translations Through Speakers|Paper Planes
```