CS 61A Structure and Interpretation of Computer Programs
Spring 2017

Mock Midterm 2

INSTRUCTIONS

- You have 1 hour to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except one 8.5” × 11” cheat sheet of your own creation.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Last name	
First name	
Student ID number	
Instructional account (`cs61a-_`)	
BearFacts email (`_@berkeley.edu`)	
TA	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

1. (10 points) The Ugly Duckling

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. If an error occurs, write "Error". If a function is displayed write `Function`, if a generator is displayed write `Generator`.

The first box has been filled in for you. Assume that you have started `python3` and executed the following statements:

```
all_ducks = []
class Duck:
    ducks = 0
    def __init__(self, name):
        self.name = name
        all_ducks.append(self)
        Duck.ducks += 1
    def __iter__(self):
        while True:
            yield all_ducks[0]
            first = all_ducks.pop(0)
            all_ducks.append(first)
    def __str__(self):
        return "String Duck"
    def __repr__(self):
        return "Duck(" + self.name + ")"
def clean(self):
    all_ducks = []
    return self
```

```
class Duckling:
    mother_duck = Duck
    def __init__(self, name):
        Duck.__init__(self, name)
        ducks = 0
    def __repr__(self):
        return "Duckling(" + \
            self.name + ")"

class Swan(Duckling):
    def __init__(self, name="autumn"):
        Duckling.__init__(self, name)
        self.mother_duck = \
            self.mother_duck("Swan")
    def __iter__(self):
        Duckling.next=Duck.__iter__(self)
        while True:
            yield next(Duckling.next)
```

```
drake = Duckling("drake")
helen = drake.mother_duck("helen")
iter1 = iter(Duck("temp"))
```

`print("hi")` hi	`Duckling.next` Generator

`next(iter1)`
Duckling(drake)
	`Duck.all_ducks[0]`
	Error

`next(iter1)`
Duck(helen)
	`all_ducks`
	[Duck(temp), Duckling(drake), Duckling(autumn), Duck(Swan), Duck(helen)]

`Duck.ducks`
3
	`next(iter1)`
	Duckling(drake)

`different = Swan()`
`different.mother_duck.ducks`
5
	`all_ducks[0]`
	Duckling(drake)

`new_iter = iter(different)`
`next(new_iter)`
Duck(helen)
	`clean(different.mother_duck)`
	Duck(Swan)

`next(iter1)`
Duck(temp)
	`all_ducks`
	[Duckling(drake), Duckling(autumn), Duck(Swan), Duck(helen), Duck(temp)]

2. (10 points) Lost in Links

Assume linked lists are defined as follows:

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Given a linked list and an interval [i, j], reverse all elements contained between the i and j indices, inclusive. Assume the linked list is 1-indexed (first element is denoted as 1), i < j, and the linked list has stricly more than j elements. Mutate the linked list.

```
def indexReverse(lnk, i, j):
    """
    >>> ll = Link(1, Link(2, Link(3)))
    >>> indexReverse(ll, 1, 2)
    >>> ll
    Link(2, Link(1, Link(3)))
    >>> ll2 = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> indexReverse(ll2, 2, 4)
    >>> ll2
    Link(1, Link(4, Link(3, Link(2, Link(5)))))
    """

    dummy = lnk

    for _ in range(i-2) :

        dummy = dummy.rest

    reverse = Link.empty

    cur = dummy.rest

    for _ in range(j - i + 1) :

        next  = cur.rest

        cur.rest = reverse

        reverse = cur

        cur = next

    dummy.rest.rest = cur

    dummy.rest = reveres
```

3. (10 points) The Forbidden Forest (of Binary Trees)
Recall the implementation of a `Tree`:

```
class Tree:
    def __init__(self, label, branches=[]):
        for c in branches:
            assert isinstance(c, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

A Binary Search Tree is a tree where the left subtree contains only nodes with keys less than the node's keys, and the right subtree contains only nodes with keys greater than the node's keys. Both left and right subtree must also be Binary Search Trees.

```
class BST:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        self.entry = entry
        self.left, self.right = left, right
```

Write a function `is_binary` that takes in a Tree, `tree`, and returns True if `tree` is a Binary Search Tree and False otherwise.

```
def is_binary(tree):
    if is_leaf(tree):
        return True
    left = tree.left
    right = tree.right
    if tree.label <= left.label or tree.label >= right.label:
        return False
    return is_binary(left) and is_binary(right)
```

Now write a function `insert` that takes in a Binary Search Tree `tree` and value `n` and inserts `n` into the tree. It mutates `tree` and the return value is also a Binary Search Tree.

```
def insert(tree, n):
    if tree is BST.empty:
        return BST(n)
    to_change = tree.left if n < tree.entry else tree.right
    if n < tree.entry:
        tree.left = insert(to_change, n)
    else:
        tree.right = insert(to_change, n)
```

What is the runtime of `is_binary` if there are n nodes in `tree`? What is the runtime of `insert`? $\theta(n)$ $\theta(logn)$

4. (10 points) Perfect Engine!

You are in an apocalyptic society and have been charged with making an n-gen, or a generator that computes all of the n-perfect numbers. However, in this apocalyptic society, **built-in AND user-defined Python multiplication is forbidden** in any form!

You have a blueprint for building a few n-gins from a natural number generator:

A 2-gen:

```
1 2 3 4 5 6 7 8 9 ...
1   4   9  16   25 ...
```

A 3-gen:

```
1 2 3 4 5 6 7 8 9 ...
1 3   7 12 19 27   ...
1     8     27   ...
```

Hint: Here is how `yield from` works. When used inside an iterable `yield from` will issue each element from another iterable as though it was issued from the first iterable. The following code is equivalent:

```
def generator1():
    for item in generator2():
        yield item
    # do more things in this generator
```

```
def generator1():
    yield from generator2()
    # more things on this generator
```

Now its your job to build the perfect n-gen and power society out of the apocalypse! Good luck!

```
def nats():
    """
    A generator that yields
    all natural numbers.
    Might be helpful!
    """
    curr = 0
    while True:
        curr += 1
        yield curr

def create_skip(n, gen):
    if n == 1:

        yield from gen

    curr, skip = 0 , 1

    for elem in gen :

        if skip == n:

            skip = 1
        else:
            curr = curr + elem

            skip = skip + 1

            yield curr
```

```
def perfect_ngen(n):
    """
    >>> two_gen = perfect_ngen(2)
    >>> next(two_gen)
    1
    >>> next(two_gen)
    4
    >>> next(two_gen)
    9
    >>> three_gen = perfect_ngen(3)
    >>> next(three_gen)
    1
    >>> next(three_gen)
    8
    >>> next(three_gen)
    27
    """
    gen = create_skip(n , nats() )

    while n > 1 :

        n = n - 1

        gen = create_skip(n , gen )

    return gen
```