6.009 Quiz 2: Practice Quiz B

Fall 2021

Name: **Answers**

Kerberos/Athena Username:

5 questions

1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- This quiz is closed-book, but you may use two 8.5 × 11 sheets of paper (both sides) as a reference.
- You may **NOT** use any electronic devices (including computers, calculators, phones, etc.).
- If you have questions, please **come to us at the front** to ask them.
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit. **Please do not write on the QR codes.**
- If you finish the exam more than 10 minutes before the end time, please quietly bring your exam to us at the front of the room. If you finish within 10 minutes of the end time, please remain seated so as not to disturb those who are still finishing their quizzes.
- You may not discuss the details of the quiz with anyone other than course staff until final quiz grades have been assigned and released.

1 ...And Beyond!

In this problem, we will create a data structure that is like a list but extends infinitely. We will achieve this via a class called `InfiniteList`, which is described below.

Initialization and Getting/Setting Values

At initialization time, `InfiniteList` should take a function mapping integer indices to values. To begin, the result of looking up any index in the resulting instance should be the result of calling that function on the given index.

However, it must also be possible to change the values in our `InfiniteList` instances. For example, setting `x[8] = 20` should mean that subsequent lookups of `x[8]` should produce `20` (rather than the result of calling the function given at initialization time). Trying to look up (or set) the value at a negative index should raise an `IndexError`.

You should implement the `__init__`, `__setitem__`, and `__getitem__` methods in the `InfiniteList` class to produce the behavior specified above.

For example, consider the following example transcripts from a Python REPL:

```
x = InfiniteList(lambda x: 0)
>>> x[20]
0
>>> x[200000]
0
>>> x[5000000000000000]
0
>>> x[-1]
IndexError
>>> x[7] = 8
>>> x[7]
8


>>> y = InfiniteList(lambda x: x**2)
>>> y[20]
400
>>> y[-20]
IndexError
>>> y[20] = 8
>>> y[20]
8
```

Combining/Modifying Infinite Lists

Now we'll add some additional functionality for combining and manipulating our infinite lists. Implement `__add__` and `__mul__` functions to support the following operations. (Recall that Python automatically calls these functions when using the + or * operators, respectively, when the first operand belongs to your class.)

- If `x` and `y` are `InfiniteList` instances, `z = x + y` should produce a new infinite list where `z[i]` is the sum of `x[i]` and `y[i]` for all (infinitely many!) possible values of `i`.
- If `x` is an `InfiniteList` instance and `c` is a number (`int` or `float`), then `z = x + c` should produce a new infinite list where `z[i]` is the sum of `x[i]` and `c` for all values of `i`.
- If `x` is an `InfiniteList` instance and `c` is a number (`int` or `float`), then `z = x * c` should produce a new infinite list where `z[i]` is the product of `x[i]` and `c` for all values of `i`.

In addition, implement `__iter__`. In this case, it should produce a generator that yields values from index 0 to infinity, one by one. Note that it is not possible to implement this as a list!

It is sufficient for your code to work on cases where, after we call one of these operators on an `InfiniteList`, we never again modify that `InfiniteList` with `__setitem__` (any behavior when we modify lists after adding them together is fine).

Your Task

On the following page(s), implement the `InfiniteList` class such that it behaves as described above.

```
class InfiniteList:
    def __init__(self, f):
        self.f = f
        self.vals = {}

    def __getitem__(self, i):
        if i < 0:
            raise IndexError
        elif i in self.vals:
            return self.vals[i]
        return self.f(i)

    def __setitem__(self, i, val):
        if i < 0:
            raise IndexError
        self.vals[i] = val

    def __iter__(self):
        x = 0
        while True:
            yield self[x]
            x += 1

    def __add__(self, other):
        if isinstance(other, (float, int)):
            f = lambda x: self[x] + other
        else:
            f = lambda x: self[x] + other[x]
        return InfiniteList(f)

    def __mul__(self, other):
        return InfiniteList(lambda x: self[x] * other)
```

2 Classes and Scoping

For each of the example programs below, what will be printed to the screen when it is run? None of the programs raise exceptions when run.

Program 1

```
n = 3

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        n = 1

bar = Bar()
print(bar.n)
```

Prints: 2

Program 2

```
n = 3

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        self.n = n

bar = Bar()
print(bar.n)
```

Prints: 3

Program 3

```
n = 3

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        self.n = n

class Baz(Bar):
    n = 5

baz = Baz()
print(baz.n)
```

Prints: 3

Program 4

```
n = 3

class Foo:
    n = 0

class Bar(Foo):
    n = 2
    def __init__(self):
        self.n = n

class Baz(Bar):
    n = 5

    def __init__(self):
        n = self.n + 2

baz = Baz()
print(baz.n)
```

Prints: 5

3 Latin Squares

A Latin square is an $n \times n$ grid of numbers in which each row and each column contains the numbers `1, 2, ..., n` exactly once. Below is an example of a 3×3 Latin square.

1	2	3
2	3	1
3	1	2

You are given a partially filled square with empty cells. Your task is to produce a Latin square by filling the empty cells, or determine that the task is impossible. We recommend that you perform a search over all possible values to place in the empty cells.

Suppose you were given the following partially filled 3×3 square:

1		
	3	

One way to fill in the empty squares with values of `1`, `2`, or `3` to produce a Latin square is shown above in the first figure.

For this question, we would like a function `solve_latin_square` that adheres to the specification below:

The function should take a single input (`grid`), a list-of-lists representation of a partially filled square. `grid[r][c]` gives the value of the cell in row `r` and column `c` (`grid[0][0]` gives the value in the top left corner). **A value of `-1` indicates that the value is missing**

If a solution exists, your program should return a list-of-lists, corresponding to `grid` with the empty cells filled in properly. Multiple solutions may exist, but you only need to return one. It is fine to solve this problem by mutating the given `grid`, or by creating a new one.

If no solution exists, your program should return `False`.

Examples:

- Using the example above, one valid solution to `solve_latin_square([[1, -1, -1], [-1, 3, -1], [-1, -1, -1]])` is `[[1, 2, 3], [2, 3, 1], [3, 1, 2]]`
- One valid solution to `solve_latin_square([[-1, -1], [-1, -1]])` is `[[1, 2], [2, 1]]`.

The skeleton on the facing page (which also defines several helper functions) is missing some pieces. By filling in the necessary code in the appropriate places, complete the definition of the function such that it behaves according to the specification above.

```
def solve_latin_square(grid):
    n = len(grid)
    all_numbers = set(range(1, n+1))

    def vals_okay(values):
        return values.count(-1) > 0 or set(values) == all_numbers

    def check_square():
        if not all(vals_okay(row) for row in grid):
            return False
        if not all(vals_okay([grid[r][c] for r in range(n)]) for c in range(n)):
            return False
        return True

    for r in range(n):
        for c in range(n):
            if grid[r][c] == -1:  # this is an empty cell
                # all valid choices for this spot
                choices = set(range(1,n+1)) - set(grid[r]) - {grid[rr][c] for rr in range(n)}

                for choice in choices:
                    grid[r][c] = choice
                    if check_square():
                        result = solve_latin_square(grid)
                        if result != False:
                            return result

                # after the for loop (looping over choice), what should we do/return (if anything)?
                grid[r][c] = -1
                return False

    # back outside the main loop (looping over r), what should we do/return (if anything)?
    return grid
```

4 Environments

For each of the programs below (two programs total), what will be printed to the screen? Additionally, how many *total* frames (**not including the global frame or the built-ins**) are created when the program runs? Your count should include all frames that were created during the program's run (even if they were garbage collected), but it should not include the global frame or the frame containing the Python built-ins.

Program a.

```
def foo(x):
    if x <= 0:
        return x
    else:
        return bar(x)

def bar(x):
    if x > 2:
        return foo(x-2) + 2
    else:
        return foo(x-1) + 3

print(foo(10))
```

How many frames? 13

What is printed?

14

Program b.

```
def partial(func, arg1):
    return lambda *args: func(arg1, *args)

def foo(a, b, c, x):
    return a*x**2 + b*x + c

def bar(x, y, z):
    return partial(partial(partial(foo, x), y), z)

f = bar(2, 3, 4)
print(f(10))
```

How many frames? 8

What is printed?

```
234
```

5 `k_mins`

In the space on the facing page, write a generator `k_mins` that takes a sequence `seq` of numbers and non-negative integer `k` as arguments, and yields each element (in order beginning from the start of the sequence) if **that element** is the minimum value within a neighborhood of $\pm k$ elements around that element. For example, for `seq = [1, 4, 5, 3, 4, 2]`, the `k = 1` elements (in order) are `1` (since 1 is less than or equal to 4 on the right), then `3` (since 3 is less than or equal to 5 on the left and 4 on the right), and finally `2`. However, for a larger `k = 2` neighborhood for the same sequence, the values yielded will be only 1 and `2`, since now the window around `3` in the sequence has values 4, 5 in its neighborhood to the left and 4, 2 in the neighborhood to the right, and thus the element 3 is no longer the minimum in the the neighborhood around that 3.

Examples:

```
>>> list(k_mins([1, 4, 5, 3, 4, 2], 1))
[1, 3, 2]
>>> list(k_mins([1, 4, 5, 3, 4, 2], 2))
[1, 2]
>>> list(k_mins([7, 6, 3, 4, 5], 2))
[3]
>>> list(k_mins([5, 4, 3, 4, 5], 0))
[5, 4, 3, 4, 5]
>>> list(k_mins([8, 9], 5))
[8]
>>> list(k_mins([], 2))
[]
>>> list(k_mins([7, 4, 5, 6, 1, 5, 6, 4, 7, 8], 2))
[4, 1, 4]
```

While all correct solutions will receive substantial credit, for full credit, your solution should be able to handle even large inputs (both in terms of k and in terms of the list of numbers) relatively efficiently.

```
def k_mins(seq, k):
    for ix, num in enumerate(seq):
        if all(num <= neighbor for neighbor in seq[max(0, ix-k):ix+1+k]):
            yield num


# or the following, which can be more efficient when k is large by avoiding the
# expensive slicing operation:
def k_mins(seq, k):
    def neighbors(ix):
        for i in range(max(0, ix-k), min(len(seq), ix+1+k)):
            yield seq[i]

    for ix, num in enumerate(seq):
        if all(num <= neighbor for neighbor in neighbors(ix)):
            yield num
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)