Homework 4: Functional Programming

Due February 15, 11:59 pm

Worth 15 points

Read this first. A few things to bring to your attention:

1. **Important:** If you have not already done so, please request a Flux Hadoop account. Instructions for doing this can be found on Canvas.
2. Start early! If you run into trouble installing things or importing packages, it's best to find those problems well in advance so we can help you.
3. **Make sure you back up your work!** I recommend, at a minimum, doing your work in a Dropbox folder or, better yet, using `git`.
4. **A note on grading:** overly complicated solutions or solutions that suggest an incomplete grasp of key concepts from lecture will not receive full credit.

Instructions on writing and submitting your homework.

Failure to follow these instructions will result in lost points. Your homework should be written in a jupyter notebook file. I have made a template available on Canvas, and on the course website at `http://www-personal.umich.edu/~klevin/teaching/Winter2019/STATS507/hw_template.ipynb`. You will submit, via Canvas, a `.zip` file called `yourUniqueName_hwX.zip`, where `X` is the homework number. So, if I were to hand in a file for homework 4, it would be called `klevin_hw4.zip`. Contact the instructor or your GSI if you have trouble creating such a file.

When I extract your compressed file, the result should be a directory, also called `yourUniqueName_hwX`. In that directory, at a minimum, should be a jupyter notebook file, called `yourUniqueName.hwX.ipynb`, where again `X` is the number of the current homework. Feel free to define supplementary functions in other Python scripts, but be sure to include them in your compressed directory if you use them. In short, I should be able to extract your archived file and run your notebook file on my own machine by opening it in jupyter and clicking, for example, `Cells->Run all`. Importantly, please ensure that none of the code in your submitted notebook file results in errors. Errors in your code cause problems for our auto-grader. Thus, even though we frequently ask you to check for errors in your functions, you should not include in your submission any examples of your functions actually raising those errors.

Please include all of your code for all problems in the homework in a single Python notebook unless instructed otherwise, and please include in your notebook file a list of any and all people with whom you discussed this homework assignment. Please also include an estimate of how many hours you spent on each of the sections of the assignment.

These instructions can also be found on the course web page at `http://www-personal.umich.edu/~klevin/teaching/Winter2019/STATS507/hw_instructions.html`. Please direct any questions to either the instructor or your GSI.

1 Iterators and Generators (4 points)

In this exercise, you'll get some practice working with iterators and generators. **Note:** in this problem, the word *enumerate* is meant in the sense of returning elements, not in the sense of the Python function `enumerate`. So, if I say that an iterator enumerates a sequence $a_0, a_1, a_2, \ldots$, I mean that these are the elements that it returns upon calls to the `__next__` method, *not* that it returns pairs (i, a_i) like the `enumerate` function.

1. Define a class `Fibo` of iterators that enumerate the Fibonacci numbers. For the purposes of this problem, the Fibonacci sequence begins $0, 1, 1, 2, 3, \ldots$, with the n-th Fibonacci number F_n given by the recursive formula $F_n = F_{n-1} + F_{n-2}$. Your solution should not make use of any function aside from addition (i.e., you should not need to use the function `fibo()` defined in lecture a few weeks ago). Your class should support, at a minimum, an initialization method, a `__iter__` method (so that we can get an iterator) and a `__next__` method. **Note:** there is an especially simple solution to this problem that can be expressed in just a few lines using tuple assignment.

2. We can generalize the Fibonacci sequence by following the same recursive procedure $F_n = F_{n-1} + F_{n-2}$, but using a different choice of initial two values for F_0 and F_1. For example, if we take $F_0 = 2$ and $F_1 = 1$, then we obtain the Lucas numbers, which are closely related to the Fibonacci numbers (`https://en.wikipedia.org/wiki/Lucas_number`). Define a class `GenFibo` of iterators that enumerate *generalized* Fibonacci numbers. Your class should inherit from the `Fibo` class defined in the previous subproblem. The initialization method for the `GenFibo` class should take two *optional* arguments that specify the values of F_0 and F_1, in that order, and their values should default so that `F = GenFibo()` results in an enumerator equivalent to the one that would have been created if you had called `F = Fibo()` (i.e., `GenFibo()` should produce an iterator over the Fibonacci numbers).

3. Define a generator `primes` that enumerates the prime numbers. Recall that a prime number is any integer $p > 1$ whose only divisors are p and 1. **Note:** you may use the function `is_prime` that we defined in class (or something similar to it), but such solutions will not receive full credit, as there is a more graceful solution that avoids declaring a separate function or method for directly checking primality. **Hint:** consider a pattern similar to the one seen in lecture using the `any` and/or `all` functions.

4. This one is good practice for coding interview questions. The *Ulam numbers* are a sequence $u_1, u_2, u_3, \ldots$ of positive integers, defined in the following way: $u_1 = 1$, and $u_2 = 2$. For all $n > 2$, u_n is the smallest integer that is expressible as a sum of two *distinct* terms from earlier in the sequence in *exactly one way*. See the Examples section of the Wikipedia page for an illustration: `https://en.wikipedia.org/wiki/Ulam_number`. Define a generator `ulam` that enumerates the Ulam numbers. **Hint:** it will be helpful to try and break this problem into smaller, simpler subproblems. In particular, you may find it helpful to write a function that takes a list of integers `t` and one additional integer `u`, and determines whether or not `u` is expressible as a sum of two distinct elements of `t` in exactly one way.

2 List Comprehensions and Generator Expressions (4 points)

In this exercise you'll write a few simple list comprehensions and generator expressions. Again in this problem I use the term *enumerate* to mean that a list comprehension or generator expression returns certain elements, rather than in the sense of the Python function `enumerate`.

1. Write a list comprehension that enumerates the sequence $2^n - 1$ for $n = 1, 2, 3, \ldots, 20$. For ease of grading, please assign this list comprehension to a variable called `pow2minus1`.

2. The *Lazy Caterer's sequence* is a sequence of numbers that counts, for each $n = 0, 1, 2, \ldots$, the largest number of pieces that can be cut from a disk with at most n cuts (`https://en.wikipedia.org/wiki/Lazy_caterer's_sequence`). The n-th number in this sequence is given by $p_n = (n^2 + n + 2)/2$, where $n = 0, 1, 2, \ldots$. Write a generator expression that enumerates the Lazy Caterer's sequence. For ease of grading, please assign this generator expression to a variable called `caterer`. **Hint:** you may find it useful to define a generator that enumerates the non-negative integers.

3. Write a generator expression that enumerates the tetrahedral numbers. The n-th tetrahedral number ($n = 1, 2, \ldots$) is given by $T_n = \binom{n+2}{3}$, where $\binom{x}{y}$ is the binomial coefficient
$$\binom{x}{y} = \frac{x!}{y!(x-y)!}.$$
For ease of grading, please assign this generator expression to a variable called `tetra`. **Hint:** you may find it useful to define a generator that enumerates the positive integers.

3 Map, Filter and Reduce (3 points)

In this exercise, you'll learn a bit about map, filter and reduce operations. We will revisit these operations in a few weeks when we discuss MapReduce and related frameworks in distributed computing. In this problem, I expect that you will use only the functions `map`, `filter` and functions from the `functools` and `itertools` modules, along with the `range` function (and similar list-related functions) and a sprinkling of lambda expressions.

1. Write a one-line expression that computes the sum of the first 10 even square numbers (starting from 4). For ease of grading, please assign the output of this expression to a variable called `sum_of_even_squares`.

2. Write a one-line expression that computes the product of the first 13 primes. You may use the `primes` generator that you defined above. For ease of grading, please assign the output of this expression to a variable called `product_of_primes`.

3. Write a one-line expression that computes the sum of the squares of the first 31 primes. You may use the `primes` generator that you defined above. For ease of grading, please assign the output of this expression to a variable called `squared_primes`.

4. Write a one-line expression that computes a list of the first twenty harmonic numbers. Recall that the n-th harmonic number is given by $H_n = \sum_{k=1}^{n} 1/k$. For ease of grading, please assign the output of this expression to a variable called `harmonics`.

5. Write a one-line expression that computes the geometric mean of the first 12 tetrahedral numbers. You may use the generator that you wrote in the previous problem. Recall that the geometric mean of a collection of n numbers $a_1, a_2, \ldots, a_n$ is given by $(\prod_{i=1}^{n} a_i)^{1/n}$. For ease of grading, please assign the output of this expression to a variable called `tetra_geom`.

4 Fun with Polynomials (4 points)

In this exercise you'll get a bit of experience writing higher-order functions. You may ignore error checking in this problem.

1. Write a function `make_poly` that takes a list of numbers (ints and/or floats) `coeffs` as its only argument and returns a function `p`. The list `coeffs` encodes the coefficients of a polynomial, $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$, with a_i given by `coeffs[i]`. The function `p` should take a single number (int or float) `x` as its argument, and return the value of the polynomial p evaluated at `x`.

2. Write a function `eval_poly` that takes two lists of numbers (ints and/or floats), `coeffs` and `args`. `coeffs` encodes the coefficients of polynomial p, and your function should return the list of numbers (ints and/or floats) representing the result of evaluating the polynomial p on each of the elements in `args`, in order. You should be able to express the solution to this problem in a single line (not including the function definition header, of course). Your function should make use of `make_poly` from the previous part to receive full credit.