Homework 2

Due Wednesday, Feb 19, 2025 at 8pm ET

The learning objectives of this homework are to:

- Practice working with higher-order functions
- Understand how to write and use Python decorators, which are higher-order functions
- Understand how to work with generators in Python

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/homework/hw2/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes code, test cases, and written solutions.

This assignment has several restrictions to ensure that you get the intended practice with the relevant abstractions and patterns. Please pay close attention to the requirements when writing your solutions. The autograder **will** enforce these restrictions.

- You may **not** import any modules in this assignment.
- You may **not** define any classes in this assignment.
- You may **not** use any loops or list/set/dictionary/generator comprehensions in `hof.py`.
- You are allowed to use loops and generator comprehensions in `generators.py`, but you may **not** use lists, sets, dictionaries, or any other built-in sequences in that file.

Some exercises have additional restrictions, as noted below.

Exercises

1. *Higher-order functions.* Define a function `make_accumulator` in Python that returns an accumulator function, which takes one numerical argument and returns the sum of all arguments ever passed to the accumulator. Do **not** define any classes for this problem.

```
def make_accumulator():
    """Return an accumulator function.

    The accumulator function takes a single numeric argument and
    accumulates that argument into a running total, then returns
    total.

    >>> acc = make_accumulator()
    >>> acc(15)
    15
    >>> acc(10)
    25
    >>> acc2 = make_accumulator()
    >>> acc2(7)
    7
    >>> acc3 = acc2
```

```
    >>> acc3(6)
    13
    >>> acc2(5)
    18
    >>> acc(4)
    29
    """
    # add your solution below
```

2. *Decorators and memoization.* Memoization is an optimization in recursive algorithms that have repeated computations, where the arguments and result of a function call are stored when a function is first called with that set of arguments. Then if the function is called again with the same arguments, the stored value is looked up and returned rather than being recomputed.

   For this problem, implement a `memoize` decorator in Python that takes in a function and returns a version that performs memoization. The decorator should work on functions that take in any number of non-keyword arguments. You may assume that all arguments are hashable.

   Hint: We recommend using a dictionary to store previously computed values, and variable argument lists to handle functions with any number of parameters.

```
def memoize(func):
    """Return a version of func that memoizes computation.

    Returns a function that computes the same result as func, but if a
    given set of arguments has already been seen before, returns the
    previously computed result instead of repeating the computation.
    Assumes func is a pure function (i.e. has no side affects), and
    that all arguments to func are hashable.

    >>> @memoize
    ... def sum_to_n(n):
    ...     return 1 if n == 1 else n + sum_to_n(n - 1)
    >>> try:
    ...     sum_to_n(300)
    ...     sum_to_n(600)
    ...     sum_to_n(900)
    ...     sum_to_n(1200)
    ... except RecursionError:
    ...     print('recursion limit exceeded')
    45150
    180300
    405450
    720600
    >>> @memoize
    ... def sum_k_to_n(k, n):
    ...     return k if n == k else n + sum_k_to_n(k, n - 1)
    >>> try:
    ...     sum_k_to_n(2, 300)
    ...     sum_k_to_n(2, 600)
    ...     sum_k_to_n(2, 900)
    ...     sum_k_to_n(2, 1200)
    ... except RecursionError:
    ...     print('recursion limit exceeded')
    45149
    180299
    405449
    720599
    """
    # add your code below
```

3. *Chain.* Recall that the `compose()` higher-order function takes in two single-parameter functions as arguments and returns the composition of the two functions. Thus, `compose(f, g)(x)` computes `f(g(x))`.

The following is a definition of `compose()` in Python:

```
def compose(f, g):
    return lambda x: f(g(x))
```

Composition can be generalized to function chains, so that `chain(f, g, h)(x)` computes `f(g(h(x)))`, `chain(f, g, h, k)(x)` computes `f(g(h(k(x))))`, and so on. Implement the variadic `chain()` function in Python.

```
def chain(*funcs):
    """Return a function that is the compositional chain of funcs.

    If funcs is empty, returns the identity function.

    >>> chain()(3)
    3
    >>> chain(lambda x: 3 * x)(3)
    9
    >>> chain(lambda x: x + 1, lambda x: 3 * x)(3)
    10
    >>> chain(lambda x: x // 2, lambda x: x + 1, lambda x: 3 * x)(3)
    5
    """
    # add your code below
```

Your solution **must** use recursion.

4. *Generators.* This problem will give you practice in working with generators that represent infinite sequences.

   - You may **not** use any built-in sequences (e.g. lists, tuples, sets, dicts) for this problem.

   a) Implement a `scale()` generator that, given an iterator of numbers, scales them by a constant to produce a new iterator. For example, given a generator `naturals()` for the natural numbers, `scale(naturals(), 2)` produces an iterator of the even natural numbers.

   ```
   def scale(items, factor):
       """Produce an iterator of the elements in items scaled by factor.

       Consumes the elements from items.

       >>> def naturals():
       ...     num = 0
       ...     while True:
       ...         yield num
       ...         num += 1
       >>> values = scale(naturals(), 3)
       >>> [next(values) for i in range(5)]
       [0, 3, 6, 9, 12]
       """
       # add your code below
   ```

   b) Implement a `merge()` generator that, given two infinite iterators whose elements are in strictly increasing order, produces a new iterator that contains the items from both input iterators, in increasing order and without duplicates.

   ```
   def merge(items1, items2):
       """Produce an ordered iterator that is the merge of the inputs.

       The resulting iterator contains the elements in increasing order
       from items1 and items2, without duplicates. Requires each of
       items1 and items2 to be infinite iterators in strictly increasing
       order. Consumes the elements from items1 and items2.

       The input iterators should only be advanced when necessary, after
   ```

```
    yielding or discarding the previous item produced by the iterator.

    >>> def naturals():
    ...     num = 0
    ...     while True:
    ...         yield num
    ...         num += 1
    >>> values = merge(naturals(), naturals())
    >>> [next(values) for i in range(5)]
    [0, 1, 2, 3, 4]
    >>> values2 = merge(scale(naturals(), 2), scale(naturals(), 3))
    >>> [next(values2) for i in range(10)]
    [0, 2, 3, 4, 6, 8, 9, 10, 12, 14]
    >>> class Wrapper:  # used to test for advancing too early
    ...     def __init__(self, iterator):
    ...         self._iterator = iterator
    ...         self._last = None
    ...     def __next__(self):
    ...         self._last = next(self._iterator)
    ...         return self._last
    ...     def __iter__(self):
    ...         return self
    ...     def last(self):
    ...         return self._last
    >>> it2 = Wrapper(scale(naturals(), 2))
    >>> it3 = Wrapper(scale(naturals(), 3))
    >>> values3 = merge(it2, it3)
    >>> next(values3), it2.last(), it3.last()
    (0, 0, 0)
    >>> next(values3), it2.last(), it3.last()
    (2, 2, 3)
    >>> next(values3), it2.last(), it3.last()
    (3, 4, 3)
    >>> next(values3), it2.last(), it3.last()
    (4, 4, 6)
    >>> next(values3), it2.last(), it3.last()
    (6, 6, 6)
    """
    # add your code below
```

c) A famous problem, first raised by R. Hamming, is to enumerate, in ascending order with no repetitions, all positive integers with no prime factors other than 2, 3, or 5. One obvious way to do this is to simply test each integer in turn to see whether it has any factors other than 2, 3, and 5. But this is very inefficient, since, as the integers get larger, fewer and fewer of them fit the requirement. As an alternative, we can build an iterator of such numbers using a generator. Let us call the required iterator of numbers `s` and notice the following facts about it.

- `s` begins with 1.
- The elements of `scale(s, 2)` are also elements of `s`.
- The same is true for `scale(s, 3)` and `scale(s, 5)`.
- These are all of the elements of `s`.

All that is left is to combine the elements from these sources, which can be done with the `merge()` generator above. Fill in the `make_s()` generator that produces the required iterator `s`.

You **must** implement the algorithm described above. You may **not** test the factors of any number – you may **not** use any division or mod operators.

```
def make_s():
    """Produce iterator of ints that only have factors in {2, 3, 5}.

    >>> values = make_s()
    >>> [next(values) for i in range(18)]
```

```
    [1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30]
    """
    # add your code below
```

Submission

Place your solutions to the first three questions in the provided `hof.py` file, and your solution to the last question in `generators.py`. Submit `hof.py` and `generators.py` to the autograder before the deadline. Be sure to register your partnership on the autograder if you are working with a partner.