Homework 4

MPCS 51042 - Python Programming - Fall 2021

Deadline: November 9 at 5:00 pm CDT

Write each problem's solutions in a separate source file: `problem1.py` and `problem2.py`. Upload the 2 source files to Gradescope before the deadline.

- The source files, functions, and classes **must** be named as specified.
- You may define additional functions, if your solution requires them.
- You **must** comment each function definition with the following info. Place each function's comments immediately below its `def` statement.
  - The purpose of the function
  - The meaning of the function's parameters, if any
  - The meaning of the function's return value, if any
- We will not be providing a set of automated tests for Homework 4. You are responsible for thoroughly testing your own code.

Problem 1

Write a generator function called `prime_generator` that yeilds successive prime numbers, starting at 2. Your generator should cache previously-discovered primes and use them to save time when discovering subsequent primes.

Sample use:

```
>>> gen = prime_generator()
>>> for i in range(10):
...     print(next(gen))
...
2
3
5
7
11
13
17
19
23
29
```

Problem 2

Memoization is an optimization technique that speeds up function calls by caching the results of a function on a given set of inputs. The standard library `functools` module contains a function wrapper (a function that encloses another function) called lru_cache, which performs memoization on any function that it wraps. That is, it stores function results for the N most recent calls. This is called a least-recently used (LRU) cache. For this problem, you will write your own version of the `lru_cache` wrapper.

Problem 2 specifications:

Write a function wrapper with the following definition:

```
lru_cache(func, maxsize=128):
    # code here
```

The returned function object should:

- Call `func` with arbitrary positional arguments. You can assume that `func` will only be called with positional arguments (not keyword arguments).
- Maintain an LRU cache called `cache` in its closure.
  - It should store up to `maxsize` groups of arguments and corresponding results (the default value is 128).
  - You can implement the cache with your choice of data structure. One good choice is an OrderedDict (documentation here).
- Maintain a `NamedTuple` named `CacheInfo` in its closure (`NamedTuple` documentation here). It should have the following attributes:
  - `hits`: The number of calls to the function where the result was previously calculated and can be returned from the cache.
  - `misses`: The number of calls to the function where the result was not previously calculated.
  - `maxsize`: The maximum number of entries that the cache can store.
  - `currsize`: The number of entries currently stored in the cache.
- Have a function attribute `cache_info()` in its closure that returns `CacheInfo`.

Sample session:

```
import operator

>>> inputs = ["a", "b", "c", "d", "a", "e", "d", "f", "g", "d", "h"]
>>> cached_upper = lru_cache(str.upper, maxsize=32)
>>> [cached_upper(x) for x in inputs]
['A', 'B', 'C', 'D', 'A', 'E', 'D', 'F', 'G', 'D', 'H']
>>> cached_upper.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)

>>> cached_add = lru_cache(operator.add, maxsize=32)
>>> inputs = [1, 2, 3, 4, 2, 2, 5, 6, 4, 1]
>>> [cached_add(x, 1) for x in inputs]
[2, 3, 4, 5, 3, 3, 6, 7, 5, 2]
>>> cached_add.cache_info()
```

```
CacheInfo(hits=4, misses=6, maxsize=32, currsize=6)

>>> cached_mul = lru_cache(operator.mul, maxsize=4)
>>> inputs = [1, 2, 3, 4, 2, 2, 5, 6, 4, 1, 2, 1, 5]
>>> [cached_mul(x, 2) for x in inputs]
[2, 4, 6, 8, 4, 4, 10, 12, 8, 2, 4, 2, 10]
>>> cached_mul.cache_info()
CacheInfo(hits=4, misses=9, maxsize=4, currsize=4)
```