Profiling and Type Checking

DS3500: Advanced Programming with Data

Purpose

- Use decorators to enhance and/or construct classes that support profiling and type checking
- Practice using type hints and annotations
- Practice using decorators to overlay new functionality on top of existing methods and functions.

Overview

In our last assignment, we used a *priority queue* to determine how orders should be fulfilled. We told you that a priority queue, like a regular queue, has two operations: *enqueue* and *dequeue*, but that the priority queue always dequeues the highest priority item from the queue. Whether this involves maintaining items in priority order with each enqueue operation, or having the priority queue search and find the highest-priority item as part of the dequeue operation was left as an implementation detail for you.

Part A: Profiling

For part A, use the Profiler class developed in lecture (and posted to Canvas) to profile the performance of your Priority code. Create a function or a method called:

process_orders(n)

that generates n random orders, adds all n orders to your priority queue, then simply dequeues all n orders (in priority order of course!). Profile your enqueue and dequeue methods over a range of n. I recommend trying n from 10,000 to 100,000 in steps of 10,000. Plot your total runtime and average runtime per call as a function of n for both the enqueue and dequeue methods. *Explain the resulting performance curves*. How does runtime increase as a function of n?

Part B: Type Checking

Imagine you were implementing production code that demands great reliability. You want to check the data types of parameters submitted to many functions and methods in your code base and have your code raise standard exceptions whenever bad parameters are passed to the function. But you most definitely do not want to have to pollute each function with lots of error checking. Wouldn't it be nice if we could simply tag the function with a decorator and have it do the type checking for us while leveraging the power of type hints and leaving the original function code unchanged?

Following the same pattern as Profiler, implement a TypeChecker class that includes a check method used to decorate functions. What will adding *@TypeChecker.check* as a decorator to a function do? It will take the type-hinted parameters of a function (any made-up function you like – an example test is provided) and compare them one by one to the actual parameters submitted to that function. Recall that the type-hints are stored in a dictionary called ***function.__annotations__***. If your type checker detects any sort of

inconsistency, a special built-in exception *InvalidParameterTypeException* will be raised. The decorator method should also automatically check to ensure that the returned object is also of the correct type. If the returned object's type does not correspond to the type-hinted return type of the function or method, an *InvalidReturnTypeException* exception should be thrown.

For both exceptions raised (InvalidParameterTypeException and InvalidReturnTypeException), include an informative error message identifying the function called, the invalid parameter name, and the expected and actual parameter (or return) type.

Your type checker does *not* need to support types defined in the typing module such as List, Dict, and Tuple, but it *should support basic types:* int, float, bool, list, dict, tuple, and str. See if you can also get your type checker to work on user-defined classes in a class hierarchy.

What to submit:

- Your performance profiling from Part A
- Your type checker class: TypeChecker.py
- Test code and outputs