Lab 1.1 Stopwatch and Tasks

In this lab you will write a Stopwatch class that measures elapsed time between a start and stop event. Then you will write some tasks and use the Stopwatch to see how long they take to complete.

Stopwatch

Write a Stopwatch class having this behavior. The implementation (private attributes) is your choice.

constructor	Constructor does not accept any parameters. When you create a stopwatch it is in the stopped state.
`running`	A read-only property that returns `True` if the stopwatch is running, `False` otherwise.
`start()`	Start the stopwatch if it is not running. Has **no effect** if called when the stopwatch is running. After calling stop(), if you call start() again it resets the timer (that is, it does not add on the previous elapsed time).
`stop()`	Stop the stopwatch if it is running. If the stopwatch is already stopped, then it has **no effect**.
`elapsed`	A read-only property that returns the elapsed time, in **seconds**. There are 2 cases: - if stopwatch is running, it is the elapsed time between start time and the current time - if stopwatch is stopped, it is the elapsed time between calls to start and stop

How to compute time and elapsed time?

Python has several functions in the **`time`** module that you can use to compute elapsed time. As described in PEP564 "Clocks Resolution in Python".their precision and accuracy vary. All the functions are more accurate under Linux than Windows; another good reason to switch your OS to Linux.

time.time_ns()	Returns the current time in nanoseconds. The precision is around 100ns.
time.perf_counter()	Returns an arbitrary time (in seconds with decimals) that can be used for computing short time intervals. The result is slightly more precise than time_ns, but not much.
time.time()	Returns the current time in seconds with decimals. The precision is about 200ns on Linux, but only about 1 millisec on Windows. **You should use one of the other functions** for more accurate timing.

Example:

```
timer = Stopwatch()
timer.start()
who = input("Type your name fast! Then press ENTER: ")
timer.stop()
print("You took", timer.elapsed, "seconds")
```

Tasks

Write the following tasks, and use the Stopwatch to compute how long they take to complete. Each task should accept a parameter that determines the upper bound or size. For example, "Add Integers from 1 to 1 million" task the size is the upper limit for the values to add.

1. Write the tasks in a file named **tasks.py**, not in stopwatch.py.
2. When executed, each task should:
   (a) describe itself before running,
   (b) use the stopwatch to compute elapsed time,
   (c) print the result after you stop the stopwatch (why does "after" matter?),
   (d) print the elapsed time in **milliseconds with decimal**.
   Example for Task 1:
   ```
   Summing integers from 1 to 1,000,000
   Finished
   The sum is 500,000,500,000
   Elapsed time: 62.632298 millisec.
   ```
3. Write the best code that you can.

Task 1: Add integers from 1 to 1 million and print the sum.

Task 2: Add Money objects from 1 Baht to 1 million Baht and print the sum.

Task 3: Add the first 1,000,000 terms of the harmonic series using floating point values, and print the sum

1 + 1/2 + 1/3 + 1/4 ... + 1/1,000,000

Task 4: Add the first 10,000 terms of the harmonic series using `fractions.Fraction` objects. Print the sum as a float. Do this by calling float(fraction). If you print the result as a Fraction it will be very long.

```
from fractions import Fraction
Fraction(1) + Fraction(1,2) + Fraction(1,3) + ... + Fraction(1, 10,000)
```

FYI: The harmonic series 1 + 1/2 + 1/3 + ... never converges to a finite value. The sum is infinite.

Document & Explain

1. In your repository README.md file, record the times used for each task.
2. Write an explanation of *why* some tasks are slower than others. In particular:
   - Why does it take longer to add Money objects than integers (which are also objects in Python)?
   - Why is it slower to add Fraction objects than to perform division (1/n) and then addition using floats?

Push Your Code to Github

Push your work for this part to Github. Use the commit message "Finish part 1."
Be sure to commit and push all files, including tasks.py.

Lab 1.2 Apply OO Principles

After you finish part 1 this will be described in class.
The principles we will apply are:

- *Don't Repeat Yourself*. Avoid duplicate code and duplicate logic.
- *Separate the part that varies from the part that remains the same*. *Then encapsulate them.*

What to Submit

Push your code to Github.