# 6 Profiling

**Key terms: time** `timeit cProfile resource`

**Reading: Marco Bonzanini's My Python Code is Slow? Tips for Profiling**[15]

**Exercise: Calculate the speedup from lazy evaluation in your Generators exercise.**

Timing or benchmarking is to measure the execution time of specific tasks. Because timings can be confounded by overhead before and after the job, all measurement errors are positive. In repeat tests, the fastest of the times yielded represents the run least encumbered by page faults and system jobs. As we shall see, better means of timing help to avoid counting these factors.

The shell command `time`, and `datetime` objects made from the `now()` method, and the results of `clock()`, all naively count clock time instead of system time. However, native techniques allow us to separate setup costs from the code under examination. `perf_counter()` returns a `float` number of seconds that is meaningful only relative to other measurements of same. `process_time()` is similar but only measures the costs of the current process, which is often what we want to know; see also `thread_time()`.

```
# Count the seconds occupied in operation():

import time
start = time.perf_counter()
operation()
elapsed = time.perf_counter() - start
```

We can also access fine-grained details on how time is spent in our programs. The modules `profile` or `cProfile` produce a report correlating elapsed time with the number of calls to each function. (The latter is a faster implementation of same in the C language.) Output shows for each function how many times it was called, and how much time was taken on various metrics. `tottime` is the overall time spent in the function excluding subcalls; `cumtime` includes time spent in subcalls. Often, the most relevant column for identifying an optimization target is `tottime`.

```
# Demonstrate the profiling of a trivial function:

import cProfile, time

def rest():
 time.sleep(2)

cProfile.run ( 'rest()' )
```

Instead of changing a program, invoking the profiler from the shell can also be convenient:

```
$ python3 -m cProfile -s tottime someprogram.py | head
```

Time is not the only resource of interest: another big one is memory usage. Check `resource.getrusage(resource.RUSAGE_` though unfortunately the unit of measurement differs by platform. Specific operating systems can also provide data on the size of running processes.

```
# Measure the memory taken by a running process.
# On Linux, the measurement is in kilobytes.

import resource
print(resource.getrusage(resource.RUSAGE_SELF).ru_maxrss)
```

Next, we will review testing mechanisms to improve reliability.

[15] `https://marcobonzanini.com/2015/01/05/my-python-code-is-slow-tips-for-profiling/`