8 Profiling

Key terms: `timeit cProfile resource`

Reading: Marco Bonzanini's My Python Code is Slow? Tips for Profiling[20]

Exercise: Use the `profile` module to examine how time is spent by your own and other programs from the "Time" assignment.

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()`.

```
>>> 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 matrix report correlating elapsed time with the number of calls to each function. (The former is a slower pure-Python implementation of the latter.) 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`. For advanced use, instantiate an object of class `cProfile.Profile` and use the `pstats` module to handle the resulting data.

```
import cProfile
import time, random

def rest():
 time.sleep(random.random())

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

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

```
$ 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_SELF).ru_maxrss`. More detailed analyses of individual objects can be acquired using several PyPI modules.

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

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