CIS 41B
Advanced Python Programming

Timing

De Anza College
Instructor: Clare Nguyen

Code Timing

- Typically in the first version of an application, the goal is to get it to work and meet all requirements.
- In later versions it can become important to analyze the code to see if we can optimize it to enable it to run faster.
- By utilizing timing functions available in Python, we can find bottlenecks, the parts in the code that are slow, in order to improve their speed.
- Timing code is also useful when we have a choice between 2 implementations that seem to do the same work on the surface, but we want to know if one version is faster than the other.
- An important point about trying to write code that runs faster:

  “Premature optimization is the root of all evil” – Donald Knuth

  Especially for Python, which does quite a bit of work ‘under the hood’ that we may not fully be aware of, it’s always good to time the code first before making any big change in the name of speed.

time

- Simplest way to measure some Python code

```
import time
start = time.time()
    # code to test
print(time.time() – start)
```

- Returns epoch time in seconds
- Time can be converted to day / time format
- Resolution: 1e-06 (microseconds)

- More precise way to measure some Python code

```
import time
start = time.perf_counter()
    # code to test
print(time.perf_counter() – start)
```

- Returns time in seconds
- Includes process sleep time
- Resolution: 1e-09 (nanoseconds) (but limited by the system)

```
import time
start = time. process_time()
    # code to test
print(time. process_time()– start)
```

- Returns time in seconds
- Does not include process sleep time
- Resolution: 1e-09 (but limited by the system)

- Other time measurements are discussed in the time module documentation.

timeit

- Measuring the code one time may not be enough since the process running the code could be suspended by the OS if the system happens to be busy during the measurement.
- We can take multiple measurements of the same code by running it multiple times and measuring each time

```
import timeit
total = timeit.timeit(stmt=‘aFunction’, number=N)
```

  - stmt is a string that contains the code we want to measure. If stmt is in multiple lines, use triple quotes.
  - number is the number of times we want to run stmt
  - total is the total number of seconds to run stmt N times

cProfile

- cProfile returns statistics on the code as it times the code.
- To profile a function:

```
import cProfile
cProfile.run(‘a_function()’)
```

- To profile a block of code

```
import cProfile
pr = cProfile.Profile()
pr.enable()
# Python code
pr.disable()
pr.print_stats()
```

- The statistics are:
  - ncalls: how many times the function/method was called
  - tottime: the total time in seconds excluding the time of other functions/methods
  - percall: average time to execute function (per call)
  - cumtime: the total or cumulative time in seconds, including the times of other functions it calls
  - percall: similar to the previous percall, but includes network delays, sleep

Line Profiler

- cProfile is used to find the function in a module that could be the bottleneck. After the function has been identified, we can use a line profiler.
- A line profiler shows the timing for each line of code. The commonly used line_profiler module is not part of the Python core modules and must be downloaded and installed separately.
- To start the line_profiler:

```
import line_profiler
profile = line_profiler.LineProfiler()
```

- To profile a block of code:

```
@profile
def aPythonFunction() :
    code
profile.print_stats()
```

- The resulting statistics are:
  - Total time: total time for the function to run
  - Hits: how often the line runs in the function
  - Time: total time for the line to run Hits times
  - Per Hit: Time / Hits or average time per run
  - %Time: percent of total time that the line takes to run

Timing Strategies (1)

- Python is a ‘higher’ level language compared to compiled languages such as C++ or Java because it is an interpreted language.
- Therefore, for the same task, Python code is generally shorter than other compiled languages. This means Python can help us develop code quickly so we can implement and test theories or develop proof of concepts.
- But because a one liner in Python can involve multiple steps ‘under the hood’, it’s good to keep in mind the basic strategies for writing more efficient code.
- On the next slide are common strategies that were covered in class that can help Python run more efficiently.

Timing Strategies (2)

These should not be blindly applied, but strategically used, they could help Python run more efficiently.

- Use list comprehension instead of a loop to build a new iterable.
- Use built-in functions, which are optimized and thoroughly tested. Here is a list of built-in functions.
- When working with very large size iterables, consider using a generator.
- Remember the in operator.
- Remember the advantage of a set: unique values, union, intersection.
- Use multiple assignments, especially when swapping data.
- Don't use global variables. They are harder to trace and slower to access than local variables, besides the fact that their value can change by accident.
- Remember the join method for strings.
- Test for error cases and back out early, and consider using try except.
- Use a decorator for caching previous data or memoization.
- Remember the key argument for sorted.
- And when it's not clear cut, time or profile the code first before changing it.

Congratulations!

You’ve reached the end of the material for CIS 41B.