CIS192 Python Programming

Graphical User Interfaces

Robert Rand

University of Pennsylvania

December 03, 2015

Outline

Time and Clock

- `time.time`
  - Returns the amount of time (in seconds) since the Epoch.
  - January 1, 1970 on UNIX and UNIX-based systems (eg. Linux, OSX)
  - January 1, 1601 on Windows
  - Higher accuracy on UNIX machines
- `time.clock`
  - Behaves differently on UNIX and Windows machines.
  - Shows processor time on UNIX machines (ignores time sleeping)
  - Shows time since first call on Windows.
  - Higher accuracy on Windows machines.

Timeit

- The `timeit` module times execution of bits of code
- Uses `time.clock` on Windows and `time.time` on everything else.
- It avoids some common traps for timing code
  - Setup code is separated out and not timed
  - Garbage collecting is turned off
  - Repeated trials suppress measurement noise
- Use `timeit` when you want to see which of 2 options is faster

Using Timeit

```
import timeit

t = timeit.Timer(stmt=stmt_code, setup=setup_code)
t.timeit(number=num_trials)
```

- `setup` is executed once before any `stmt`s
- `stmt` is executed `num_trials` times
- Returns time in seconds taken to execute
- The time does not include executing `setup`
- Copying the code to execute into a multi-line string could be useful
- A better idea is to import it:
  - `setup = 'from __main__ import func_to_time'`

Command Line and iPython Timeit

- Command Line
  - Use `python -m timeit ``[command]''`
  - Include setup code as first argument with `-s`
  - Chooses an appropriate number of iterations for you.
  - Good for small snippets of mostly native code.
- iPython
  - Allows you to type `%timeit [function]` in the iPython REPL
  - Has local scope: No need to import required functions.

Profile

- `profile` and `cProfile` are built-in profilers
- Profiling a program gives data on a particular execution
- Shows which functions the program spends time in
- Useful if a program is running slower than you expect/want
- Profiling can identify bottleneck functions
- Then you can target optimizations to those functions
- Since there is overhead to track which functions are being called:
    - Profiling can take longer than regular execution
    - The output should not be used to benchmark (use `timeit`)

profile vs cProfile

- `profile` and `cProfile`
  - have the same interface
  - cProfile is a faster C extension
- To profile a function call: `cProfile.run('function()')`
- Profile the whole program with

```
if __name__ == '__main__':
    cProfile.run('main()')
```

pstats to Format Output

- Nice printing of output with `pstats`

```
if __name__ == '__main__':
    cProfile.run('main()', 'restats')
    p = pstats.Stats('restats')
    p.sort_stats('cumulative').print_stats()
```

- Save the output to a file `'restats'`
- Parse that file with `pstats`
- Sort by a column of the output

Byte Code

- The CPython interpreter:
  - Generates byte code (.pyc)
  - Executes that byte code
- When a Python module is imported byte code is saved
- Byte code is put in the `__pycache__` directory
- By default a `.pyc` byte code file is used
- Running `python -O` uses an “optimized” `.pyo` file
  - Not much optimization actually happens
  - Ignores `assert` statements
- Benefits of pre-compilation
  - Skip the compilation step when invoking the `.py` file
  - If imported multiple times, it will only get compiled once
- Compiling to byte code will not make your program faster

Cython

- Cython is a optimizing static compiler for Python
- It is a superset of Python:
  - It *should* run all pure Python code correctly
  - Directly call C functions
  - Add C type declarations to Python variables
- Compiles through C instead of to byte code
  - Results in native machine code: shared object `.so`
- Have to jump through a few hoops to compile
  - create a `setup.py` file that invokes `cythonize`
  - create a stub `.py` file to import the original and call a function
- Faster Python code basically for free

Outline

1. Performance
   - Measurement
   - Compilation
2. Concurrency
   - Multi-Thread
   - Multi-Process
   - Worker Pools

Threading

- `threading` is the built-in threading library
- Create a thread:

```
from threading import Thread
args = (a1, a2, ...)
kwargs = {k1:v1, k2:v2, ...}
t = Thread(target=fun, args=args, kwargs=kwargs)
t.start()
```

- `t.start()`:
  - Creates a new thread in the current Python process
  - That thread then calls `fun(*args, **kwargs)`

Waiting on Threads

- When a thread is created it can execute in parallel
- Sometimes you need to be sure the Thread is done
- `t.join()` $\rightarrow$ Waits until thread t finishes
- If you create a bunch of threads to do a task
  - The task isn't finished until all of the threads finish
  - You should not return a partial result to the caller
  - `.join()` on all the workers before finishing

GIL

- CPython has a Global Interpreter Lock (GIL)
- This means that only one thread can execute at a time
- The exception is that threads release the GIL while doing I/O
- The reason is to make the implementation of CPython simple
  - Simple is better than complex
- Take away:
  - Multi-threaded Python code is not worth your time
  - unless your doing a lot of I/O

Multi-processing

- `multiprocessing` is the built-in mulitprocessing library
- Create a new process:

```
from multiprocessing import Process
as = (a1, a2, ...)
ks = {k1:v1, k2:v2, ...}
p = Process(target=fun, as=args, ks=kwargs)
p.start()
```

- `p.start()`:
  - Creates a new Python process
  - That process then calls `fun(*args, **kwargs)`
- You should wait on processes with `p.join()`

Differences from Threads

- Threads (In Python)
  - Threads share memory
  - Changing a variable in one thread can effects other threads
  - Threads are cheap to make
  - Threads basically need only a stack and Instruction Pointer
- Processes (In Python)
  - Processes do not share the same memory
  - Processes are expensive to create
  - A new process might copy all of the data of its parent
  - Each process gets its own GIL
  - Multiple processes actually run computations in parallel

Inter-Process Communication

- Since Processes don't share memory $\rightarrow$ need messages
- `from multiprocessing import Queue`

  ```
  result_queue = Queue()
  p = Process(target=func,
              args=(data, result_queue))
  p.start()
  ans = result_queue.get()
  p.join()
  ```

- If you try to `join` a process with a non-empty queue
  - The process won't terminate
  - You may deadlock

ProcessPoolExecutor

- Use a pool of worker processes instead of 1 process per task
  - Creating a process is expensive
  - Want to reuse the processes we already have
- `concurrent.futures` provides pools of workers
- `import concurrent.futures as cf`
- `cf.ProcessPoolExecutor`
  - Creates workers using `multiprocessing`
- `cf.ThreadPoolExecutor`
  - Creates workers using `threading`
- Map your workers to jobs

```
cpus = os.cpu_count()
with cf.ProcessPoolExecutor(cpus) as ex:
    results = ex.map(function, [data1, ...])
```

Concurrency is Complicated

- These are the basics for clearly separable tasks
- What to do if multiple threads want the same data?
  - Obstacles: Race Conditions, Starvation, Deadlock
  - Tools: Locks, Barriers, Semaphores, ...
- What if you want to run on multiple machines?
  - Distributed Computing

Thank You