Time and Clock

- \texttt{time.time}
  - Returns the amount of time (in seconds) since the Epoch.
  - January 1, 1970 on UNIX and UNIX-based systems (e.g. Linux, OSX)
  - January 1, 1601 on Windows
  - Higher accuracy on UNIX machines

- \texttt{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.
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

```python
import timeit

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

- **setup** is executed once before any **stmts**
- **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:
  ```python
  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.
Outline

1. Performance
   - Measurement

2. Concurrency
   - Multi-Thread
   - Multi-Process
   - Worker Pools

3. Distributed Computing
   - Shell Commands
   - Fabric
Threading

- **threading** is the built-in threading library

- **Create a thread:**

  ```python
  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()` → 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
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 multiprocessing library
- Create a new process:

```python
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 → need messages
- `from multiprocessing import Queue`
  ```python
  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

```python
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
Outline

1. Performance
   - Measurement

2. Concurrency
   - Multi-Thread
   - Multi-Process
   - Worker Pools

3. Distributed Computing
   - Shell Commands
   - Fabric
The `subprocess` module allows execution of shell commands

```
subprocess.call('ls')
```

The commands are run in a child process

Longer commands can be specified with a list of strings

```
call(['grep', '-ir', 'python', './'])
```

The I/O of the subprocesses can be set with kwargs

```
call('ls', stdin=f_handle, stdout=DEVNULL)
```
Alternatives to Call

subprocess.call
- Executes the command
- Waits until it exits
- Returns the exit code

subprocess.check_call
- Just like call but ...
- If the exit code is not one (Abnormal exit) raise a CalledProcessError

subprocess.check_output
- Just like check_call but ...
- returns the contents of stdout after the process finishes

subprocess.Popen
- Takes the same arguments as call (Except timeout)
- Doesn’t wait for the process to finish
- Have to check for output explicitly
- More flexible for more complicated tasks
Using `subprocess` you can pipe the output of one process to the input of another process.

The easiest way: Just use `'|'` in a subprocess command.

```python
subprocess.check_call(['ls', '|', 'wc', '-l'])
```

`subprocess.PIPE` lets you do this with more control.

**Must use** `subprocess.Popen` instead of `subprocess.call`

```python
subprocess.call(['ls', '|', 'wc', '-l'])
```

`subprocess.call` waits for the process to finish.

If the pipe fills up then the process will deadlock.
Installing Fabric

```
pip install fabric
```
What is it?

Fabric is “a Python library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.”

Fabric has two main features:

1. Easy use of the terminal from within a python program.
2. Allows multiple computers to easily communicate and coordinate tasks.
Invoking Fabric

- `$ fab func` will call the `func` method in `fabfile.py`
  - Optional arguments go after the colon.
  - eg. `$ fab hello:name=Bob` will call `hello` with the argument name set to “Bob”.

- If the file isn’t called `fabfile.py` use
  - `$ fab -f [filename] [function_name]`
There are multiple ways of specifying the remote host(s).

1. `fab -H system1,system2,...` specifies the system(s) upon call.
2. `env.hosts = system1,system2,...` in the fabric file.
3. The decorators `@hosts(system1,system2,...)` can be applied to individual functions.
4. Otherwise, fabric will ask for the system upon execution.
Commands

- **local(command)** runs a command on the local machine.
  - Set `capture = True` to return the output instead of sending it to stdout.

- **run(command)** runs a command on the remote machine.
  - Set `stdin = x` and `stdin = y` to pass stdin and stderr to the variables $x$ and $y$. 
Get and Put

- Both `get` and `put` take in a local_path and remote_path, relative to the current local and remote directories. (Absolute paths are also permitted.)
- `put` transfers a file from the local machine to the remote machine.
- `get` does the reverse.
- Use `with cd(dir)` to run all commands from a given directory.
For more on Fabric, check out:

http://docs.fabfile.org/en/1.11/tutorial.html
Thank You

Python!!!