Programming Principles in Python (CSCI 503/490)

Arrays

Dr. David Koop
CPU-Bound vs. I/O-Bound

CPU Processing
Compute Problem 1
Compute Problem 2

I/O Waiting
Request 1
Request 2
Request 3

CPU Processing

Time

[J. Anderson]
Threading

- Threading address the I/O waits by letting separate pieces of a program run at the same time.
- Threads run in the same process.
- Threads share the same memory (and global variables).
- Operating system schedules threads; it can manage when each thread runs, e.g. round-robin scheduling.
- When blocking for I/O, other threads can run.
Python Threading Speed

- If I/O bound, threads work great because time spent waiting can now be used by other threads
- Threads do not run simultaneously in standard Python, i.e. they cannot take advantage of multiple cores
- Use threads when code is I/O bound, otherwise no real speed-up plus some overhead for using threads
Python and the GIL

- Solution for reference counting (used for garbage collection)
- Could add locking to every value/data structure, but with multiple locks comes possible **deadlock**
- Python instead has a Global Interpreter Lock (GIL) that must be acquired to execute any Python code
- This effectively makes Python single-threaded (faster execution)
- Python requires threads to give up GIL after certain amount of time
- Python 3 improved allocation of GIL to threads by not allowing a single CPU-bound thread to hog it
Multiprocessing

• Multiple processes do not need to share the same memory, interact less
• Python makes the difference between processes and threads minimal in most cases
• Big win: can take advantage of multiple cores!
Multiprocessing using concurrent.futures

• import concurrent.futures
  import multiprocessing as mp
  import time

  def dummy(num):
    time.sleep(5)
    return num ** 2

  with concurrent.futures.ProcessPoolExecutor(max_workers=5,
      mp_context=mp.get_context('fork')) as executor:
    results = executor.map(dummy, range(10))

• mp.get_context('fork') changes from 'spawn' used by default in MacOS, works in notebook
asyncio

- Single event loop that controls when each task is run
- Tasks can be ready or waiting
- Tasks are **not interrupted** like they are with threading
  - Task controls when control goes back to the main event loop
  - Either waiting or complete
- Event loop keeps track of whether tasks are ready or waiting
  - Re-checks to see if new tasks are now ready
  - Picks the task that has been waiting the longest
- `async` and `await` keywords
- Requires support from libraries (e.g. `aiohttp`)
When to use threading, asyncio, or multiprocessing?

- If your code has a lot of I/O or Network usage:
  - If there is library support, use asyncio
  - Otherwise, multithreading is your best bet (lower overhead)
- If you have a GUI
  - Multithreading so your UI thread doesn't get locked up
- If your code is CPU bound:
  - You should use multiprocessing (if your machine has multiple cores)
## Concurrency Comparison

<table>
<thead>
<tr>
<th>Concurrency Type</th>
<th>Switching Decision</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-emptive multitasking (threading)</td>
<td>The operating system decides when to switch tasks external to Python.</td>
<td>1</td>
</tr>
<tr>
<td>Cooperative multitasking (asyncio)</td>
<td>The tasks decide when to give up control.</td>
<td>1</td>
</tr>
<tr>
<td>Multiprocessing (multiprocessing)</td>
<td>The processes all run at the same time on different processors.</td>
<td>Many</td>
</tr>
</tbody>
</table>

[J. Anderson]
Assignment 6

- Object-Oriented Programming & Exceptions
- Classes for an online market
- Use inheritance
- Due today
Test 2

• Thursday in class, 12:30-1:45pm
• Covers material from the beginning of course, emphasizing material since Test 1
• Similar Format to Test 1
Arrays

What is the difference between an array and a list (or a tuple)?
Arrays

• Usually a fixed size—lists are meant to change size
• Are mutable—tuples are not
• Store only one type of data—lists and tuples can store any combination
• Are faster to access and manipulate than lists or tuples
• Can be multidimensional:
  - Can have list of lists or tuple of tuples but no guarantee on shape
  - Multidimensional arrays are rectangles, cubes, etc.
Why NumPy?

• Fast **vectorized** array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
• Common array algorithms like sorting, unique, and set operations
• Efficient descriptive statistics and aggregating/summarizing data
• Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
• Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
• Group-wise data manipulations (aggregation, transformation, function application).

[W. McKinney, Python for Data Analysis]
import numpy as np
Creating arrays

- `data1 = [6, 7, 8, 0, 1]`
  `arr1 = np.array(data1)`
- `data2 = [[1.5, 2, 3, 4], [5, 6, 7, 8]]`
  `arr2 = np.array(data2)`
- `data3 = np.array([6, "abc", 3.57])` # !!! check !!!
- Can check the type of an array in `dtype` property
- Types:
  - `arr1.dtype # dtype('int64')`
  - `arr3.dtype # dtype('<U21'), unicode plus # chars`
Types

- "But I thought Python wasn't stingy about types..."
- numpy aims for speed
- Able to do array arithmetic
- int16, int32, int64, float32, float64, bool, object
- Can specify type explicitly
  - arr1_float = np.array(data1, dtype='float64')
- astype method allows you to convert between different types of arrays:
  - arr = np.array([1, 2, 3, 4, 5])
  - arr.dtype
  - float_arr = arr.astype(np.float64)
**numpy data types (dtypes)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Type code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8, uint8</td>
<td>i1, u1</td>
<td>Signed and unsigned 8-bit (1 byte) integer types</td>
</tr>
<tr>
<td>int16, uint16</td>
<td>i2, u2</td>
<td>Signed and unsigned 16-bit integer types</td>
</tr>
<tr>
<td>int32, uint32</td>
<td>i4, u4</td>
<td>Signed and unsigned 32-bit integer types</td>
</tr>
<tr>
<td>int64, uint64</td>
<td>i8, u8</td>
<td>Signed and unsigned 64-bit integer types</td>
</tr>
<tr>
<td>float16</td>
<td>f2</td>
<td>Half-precision floating point</td>
</tr>
<tr>
<td>float32</td>
<td>f4 or f</td>
<td>Standard single-precision floating point; compatible with C float</td>
</tr>
<tr>
<td>float64</td>
<td>f8 or d</td>
<td>Standard double-precision floating point; compatible with C double and Python float object</td>
</tr>
<tr>
<td>float128</td>
<td>f16 or g</td>
<td>Extended-precision floating point</td>
</tr>
<tr>
<td>complex64,</td>
<td>c8, c16,</td>
<td>Complex numbers represented by two 32, 64, or 128 floats, respectively</td>
</tr>
<tr>
<td>complex128,</td>
<td>c32</td>
<td></td>
</tr>
<tr>
<td>complex256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>?</td>
<td>Boolean type storing True and False values</td>
</tr>
<tr>
<td>object</td>
<td>0</td>
<td>Python object type; a value can be any Python object</td>
</tr>
<tr>
<td>string_</td>
<td>S</td>
<td>Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'</td>
</tr>
<tr>
<td>unicode_</td>
<td>U</td>
<td>Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')</td>
</tr>
</tbody>
</table>
Array Shape

- Our normal way of checking the size of a collection is... `len`
- How does this work for arrays?
  - `arr1 = np.array([1,2,3,6,9])`
    ```
    len(arr1) # 5
    ```
  - `arr2 = np.array([[1.5,2,3,4],[5,6,7,8]])`
    ```
    len(arr2) # 2
    ```
- All dimension lengths → shape: `arr2.shape # (2,4)`
- Number of dimensions: `arr2.ndim # 2`
- Can also reshape an array:
  - `arr2.reshape(4,2)`
  - `arr2.reshape(-1,2) # what happens here?`
Speed Benefits

• Compare random number generation in pure Python versus numpy

• Python:
  - import random
  - %timeit rolls_list = [random.randrange(1,7) for i in range(0, 60_000)]

• With NumPy:
  - %timeit rolls_array = np.random.randint(1, 7, 60_000)

• Significant speedup (80x+)
Array Programming

• Lists:
  - c = []
    for aa, bb in zip(a, b):
      c.append(aa + bb)

• How to improve this?
Array Programming

• Lists:
  - `c = []`
    ```python
    for aa, bb in zip(a, b):
        c.append(aa + bb)
    ```
  - `c = [aa + bb for aa, bb in zip(a, b)]`

• NumPy arrays:
  - `c = a + b`

• More functional-style than imperative
• **Internal iteration** instead of external
Operations

- \(a = \text{np.array([1,2,3])}\)
  \(b = \text{np.array([6,4,3])}\)

- (Array, Array) Operations (Element-wise)
  - Addition, Subtraction, Multiplication
  - \(a + b \ # \ array([7, 6, 6])\)

- (Scalar, Array) Operations (Broadcasting):
  - Addition, Subtraction, Multiplication, Division, Exponentiation
  - \(a ** 2 \ # \ array([1, 4, 9])\)
  - \(b + 3 \ # \ array([9, 7, 6])\)