Faster Python

Warwick RSE

13/12/2021

Generic Tricks

Redundant Work

- Sometimes obvious tricks just get missed
- Never do the same thing twice
- See pi and two_pi constants often
  - Tiny but it can help
- **Don’t recalculate things where you could keep them**
  - Unless they’re very large

Function Call Overhead

- Obvious, but calling a function many times costs
- **Use array functions wherever possible**
- (Sometimes) write functions to take both element or list/array/iterable
  - May not make sense to be single function
  - May want distinct name

Functions You Can't Skip

- Sometimes good design and readability indicate a function is ideal
- But performance disagrees
- Difficult stalemate to break
  - Reduce number of calls as much as possible
  - Break up and call only where essential
  - Wait until things work and Copy-Paste-Inline

Caveat

- General rule is DRY
  - Don’t Repeat Yourself
- Many advantages for maintainability and readability
  - Good for performance while developing since only one place to optimise
  - Not good for performance in final code
- Some of the stuff today involves doing this automatically
- **Copy and paste code judiciously to prevent overheads**

Data availability

- The CPU can only run at full speed if the data can be given to it as fast as it can consume it
- It is not possible to give a modern CPU data as fast as it can perform a single machine code instruction on it
  - Memory is slow (4 - 16 times slower)
- ***Want to do as much processing on one piece of data as possible before moving onto the next!***

Data availability

- Data is stored in memory essentially as a single long "strip" of data
- This data is not retrieved element by element but in chunks, the size of the chunk being processor dependent
- The chunks are stored in the processor **cache** that is much faster than main memory but also much smaller
- ***Want to access your data so that you use data that has already been loaded into cache rather than getting more from memory if possible***

Data availability

- Classic example is an array

- `array=np.zeros(100)`
  Behind the scenes this creates a block of 100 floating point numbers

- When you access an element the elements near it are also pulled into cache

- Get better performance if you go through in order than accessing elements at random

- ***Want to access 1D arrays in order***

Data availability

- What about 2D arrays?

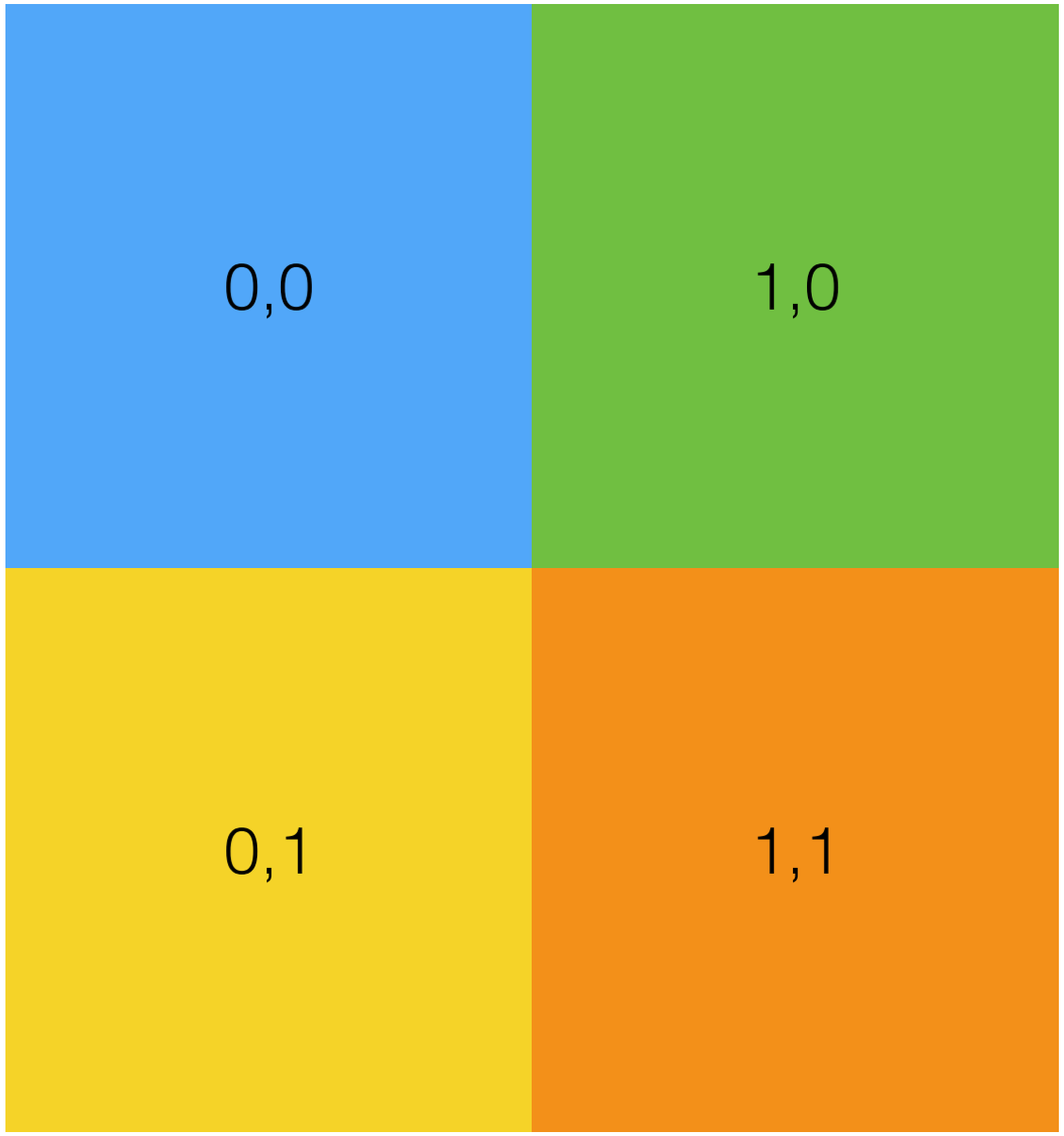
- `array=np.zeros([100,100])`
  Behind the scenes this creates a single block of 10000 floating point numbers. The 2D array only exits "theoretically"

- Mapping from the 2D array to the 1D behind the scenes is arbitrary

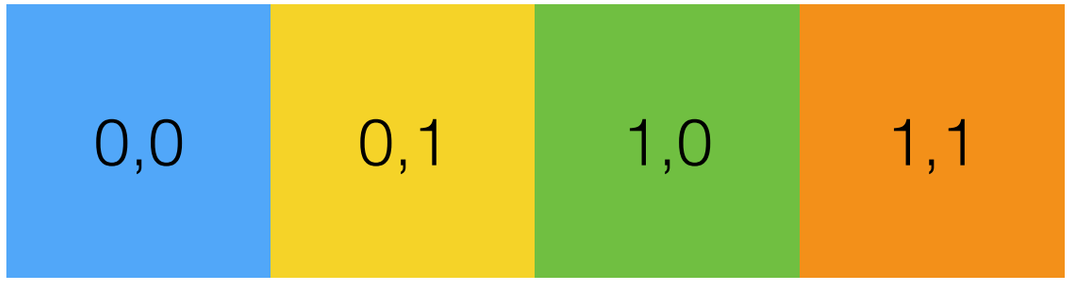
  - First index fastest - column major order (Fortran Order)

  - Last index fastest - row major order (C Order)

Data availability



Column Major
Fortran/Matlab



Row Major
C & Almost everything else

Data availability

- What about Python?
- Python itself is C like (Row Major)
- Numpy and SciPy can do both but Row Major is the default
- Some libraries that are designed to work with Fortran might return arrays in Column Major order
- Can check using

```
np.zeros([100,100]).flags['F_CONTIGUOUS']
np.zeros([100,100]).flags['C_CONTIGUOUS']
```

Data availability

- Want to access your multidimensional arrays in the order that they are laid out in memory

- So you need to check the ordering of your array and then access it so that you are going through the underlying array in order.

- ```
  array=np.zeros([100,100])
  for i in range(99):
      for j in range(99):
          array[i][j] += 10 #C order
          array[j][i] += 10 #Fortran order
  ```

- *Access multidimensional arrays in the order of the underlying storage in memory*
  ```

Data availability

- Loading data from disk is very slow
  - ***If possible load your data once and hold it in memory***
- Getting data from remote servers (internet sources, database servers) is even worse
  - Metadata, request overheads, network latency
  - **Request as little data as you can but request it all at once**

Data availability

- Example - SQL Database
  - `SELECT * FROM data;`
    - Gets all of the data from the server and returns it to you. Bad if you don't need it all
  - `SELECT product_id FROM data WHERE product_name='toy';`
    `SELECT product_id FROM data WHERE product_name='game';`
    - A lot of the time in remote access is waiting for the data to start transferring. Two commands slows everything down

Data availability

- Example - SQL Database
  - `SELECT product_id FROM data WHERE product_name = ‘toy’ OR product_name = ‘game’;`
  - Much better. Get only the data you need in a single command
- No way around it though internet access is much slower than local storage

Data availability

- Same idea if you are getting data from a web API
  - Make as few request as possible for as little data as possible
- Same with “smart” remote filesystems
- Pretty much anything where data isn’t sat on your computer

Asynchronous Data

- If you need to request your data in multiple chunks you'll want to look at getting the data asynchronously
- Tell network layer to get data from server (asyncio and aiohttp in Python for web API data, see documentation for databases etc.)
- Work on data you already have
- Once that data has been processed wait for network transfer to finish then repeat the cycle
- Best case scenario you are entirely hiding the time to transfer the data if it takes longer to process than to fetch
- Still the best you can do if that isn't the case

Algorithm Tricks

Algorithm performance

- Switching algorithm has the possibility of the biggest improvement in performance
- Sometimes different algorithm for the same task
  - Quick Sort vs Bubble Sort
- Sometimes different model
  - Fluid description vs particle description

Algorithm performance

- Always look if there are better algorithms before trying programming improvements
- Quite often there aren't though
- Also be careful of using a clever algorithm if there is a trusted one in your field
  - People have to trust your results

Algorithm tricks

- Early termination
  - Stop your algorithm as soon as you know it has succeeded/failed
- Divide and Conquer
  - Algorithms that can split the work up into pieces and then combine the results can be much faster
  - If you can keep splitting then you can get **much** faster
  - Also scope for parallelism (see later)

Python Tricks

Loops

- Python loops are slow
  - Python while loops are very slow since they use Python code to test conditions etc.
    - **Never use *while* in place of *for* loops**
  - Python range loops are quite slow
  - Python “for in” loops are the fastest that you are going to get
- List comprehensions and Dict comprehensions are faster than loops

Containers

- Python tuples are fairly fast but rather limited
- Python lists are not fast but they are the fastest native Python container after tuples
- dicts are very slow, both to store from and to iterate over

Libraries

Numpy/Scipy

- Almost certainly familiar with the Numpy numerical library
- Numpy arrays are substantially faster than standard Python containers
- Numpy intrinsic operations on whole arrays are implemented in C/C++/Fortran
  - Orders of magnitude faster than Python functions
- Scipy extends this to scientific tools
  - e.g. Lomb-Scargle Periodogram

Numpy/Scipy

- They are some of the best supported and best performing libraries in the entire Python ecosystem
- Always look in these two before trying to use other libraries
  - Except domain specific libraries like Astropy
- Performance will be much, much faster than anything you can write in Python

Hoisting

- Always pull as much as possible out of loops
  - Be aware of what things unpack to loops
- Some libraries that implement containers (notably numpy) have options that allow you to “offload” loops to their backend
  - c.f. m1.dot(m2) is 6000 times faster than implementing matrix multiply in Python

Numpy Vectorize

- Numpy can take a function that operates on a single element of an array and turn it into a function that operates on all elements of an array

- Called "vectorize"

  - ```
    vector_function = np.vectorize(scalar_function)
    array_out = vector_function(array_in)
    ```

- Performance is sometimes better than using a loop but much slower than a Numpy intrinsic array function

- Much more convenient than having loops everywhere
    ```

Numexpr

- Library that can execute numerical expressions faster than Numpy
- https://github.com/pydata/numexpr
  - Available through pip
- Downside - it doesn't execute Python code, it executes it's own expressions very fast
- When you want to know the answer to a simply expressed mathematical problem it can be much faster than Numpy

Other libraries

- Libraries speed development of your code quite a lot
  - Much more variable on performance
- Libraries written in Python themselves can be slower than your own implementation
  - They have to work for general case

Other libraries

- Be careful using libraries of unknown provenance
  - Might be very slow for your problem
  - Might have bugs
  - Might be abandoned later and have to be replaced - maintenance load
- Be careful of using libraries to do things that you don't understand
  - Can easily be subtleties that you miss
  - Hard to test that you are getting the answer right

Other libraries

- Be careful using libraries to solve small/trivial problems
  - Especially when the library is meant to do a much harder job
    - Will tend to be slow if implemented in Python
  - Leftpad problem

10

Outputting Data

- Python IO is a fairly generic buffered IO system
- Hard to get really good performance
  - Performance not much changed by how you choose to write your data
- Good compression libraries available for large data files
  - Tend to slow down code due to compression more than they save on size reduction

Outputting Data

- Main choice in outputting data is ASCII (human readable) vs binary (not human readable)

- ASCII produces much larger files and attendant slower output (2-3x), but is slightly less ambiguous since it is human readable

  - Still need more information that just a list of numbers to read file

- On balance I'd always be tempted to write binary data rather than ASCII

- `file=open('filename.dat','wb')`

Outputting Data

- Once you are writing data you pretty much have to hand the output system a bytearray object

- Most numerical and string variables, lists and arrays can be converted to bytearrays by simple conversion

- `out = bytearray(np.random.rand(100,100))`
  `file.write(out)`

Outputting Data

- More complicated data types (classes, dicts etc.) must be serialised in some way
  - Python provides a standard mechanism for doing this **pickle**
- `pickle.dump(item, filename)`
  `bytes = pickle.dumps(item)`
- Do not use pickle for large items in Python2. The files produced are very large

Outputting Data

- Be careful with pickle for long term storage
- Not a standard format, Python only
- NOT SECURE
  - Never trust a Python pickle from someone else
  - Can easily contain malicious code
- For long term storage or portability consider a standard file format

Standard Formats

- Text
  - XML
  - JSON
  - YAML
- Binary
  - HDF5
  - NetCDF
- Domain specific formats are also common

Reading data

- Important trick for reading data
- **DON’T OPEN FILE FOR READING AND WRITING UNLESS YOU INTEND TO WRITE**
- Slows buffering so much that it can be 10x slower