Speed considerations in Python

Python has a well-earned reputation for being slower than other programming languages used for science or data processing in general. So far as execution time is concerned, Python is never faster than compiled C or Fortran, the languages often used for heavy numerical work. Python is used because it's faster to write, and for most work in science, we spend many hours writing and much fewer hours running programs. Now that doesn't mean that we just have to deal with Python's inherent slowdown, there are plenty of ways to mitigate it.

Looping

A lot of idioms in other languages don't apply in Python. Consider `examples/iteration.py`. You incur a ~2.5x slowdown just by trying to use Python's `for each` loop like a classical `for` loop. The speed of the loop itself should never be a limiting factor in actual situations, but it's a good example of the problems that can arise from *how* you write Python code. Just because it produces the right output doesn't mean there is no room to improve.

Numpy

The first-order speed improvement is to lean on `numpy` as often as possible and avoid looping. If you need to do some operation on an array think of how you can do it by avoiding loops. It was almost always be faster without a loop. Sometimes you'll only want to do an operation on some elements of an array. Figure out how to identify those elements, build a boolean array or array of indices then do your operation on the subset of elements by indexing with an array.

Greedy evaluation

Python uses greedy evaluation, as opposed to lazy evaluation. That means that these two examples are exactly as costly in terms of CPU time:

```
y = m*x + b

y = m*x
y += b
```

That is; there is no reason to cram a long equation onto a single line. Break up long expressions by using intermediate variables, or by using Python's implicit line completion. New lines that occur within grouping symbols `(), [], {}` don't actually end a line. For example:

```
y = (m * x
     +b)
```

numexpr

I mentioned before that there is no speed increase by cramming an expression into a single line, but with numexpr, there is. `numexpr.evaluate`, takes a single string that's a numerical expression written out. It evaluates the expression and returns the result. Numexpr provides substantial speed increases as the number of array operations in the expression increase. The one caveat is that numexpr does not support all operations that numpy can handle, and can impose a speed penalty on medium-sized arrays. Medium here means ~10,000 elements. Keep in mind a small image is 1,000,000 elements.

numba

Disclaimer

Before I get too far into the numba hype, I need to be clear that numba is a work-in-progress. In most real-world cases you can't just add it to your code, and in some cases it will require substantial rewriting, if adding it is even possible at all. Numba is designed for relatively simple

situations where you just want to do some operations on an array, without calling functions in numpy and scipy or other libraries, though support for numpy functions is growing (some work, most don't).

Compiled vs Interpreted

The secret to all the big speed increases is to run code that isn't Python, and numba is the best example of this. Numba is a jit, a just-in-time compiler. Python is really just a specification for what the text we type does; how that is done doesn't matter. The most common implementation is CPython which is interpreted, but for real speed you need to use a compiler. A compiler can analyze large blocks of code and determine how best to do what the code specifies; this is how numba speeds up Python.

The jit

Numba is pretty much just two functions, `numba.jit` and `numba.vectorize`. For now we'll just worry about `numba.jit`. A classic compiler analyzes code then builds a file that's written in machine code, a language that your computer understands directly. A jit waits for you to try to run the marked code, then tries to compile. The benefit is that while a classic compiler requires you to state types for everything, a jit just uses the particular variables you tried to run the code with to identify types. It then produces machine code, and runs that machine code next time the marked function is called with the same variable types. To use the numba jit, just put `@numba.jit` directly above a function definition. The `@` indicates a decorator, which is special kind of function that takes a function and returns a modified version of it. `numba.jit` is a function that takes another function and returns a compiled version.

nopython

`numba.jit` has an option, `nopython`. Normally `numba.jit` will try to infer the types of everything in the decorated function, and if it can't figure out what they should all be it falls back to using Python objects. That's unhelpful in terms of speed, so setting `nopython=True` will raise an exception with a slightly helpful traceback if the jit can't figure out how to efficiently compile the function. In general, if the code doesn't work with `nopython=True`, there's no reason to use numba.