The all-powerful numpy.ndarray

The numpy ndarray is much different from the native Python data structures I've highlighted above. Firstly, it's actually an array. Python's lists are built on top of an array, but Python introduces a layer of seperation between the list and the array that negates the most of the power of arrays. The reason arrays are so good for fast processing comes from how the data is physically placed in memory. An array is a single large block of memory that has a starting point and a stride or step size. The stride is defined by the size of the thing to be stored in the array, and the size of the array is defined by the number of things in the array, and the size of each thing. This means that it is very easy to find each object in the array, since its location is defined simply by the stride and its index. Numpy ndarrays actually store arrays of the data itself, which restricts the arrays to storing one type of data. You can have an array of 32-bit integers, 64-bit floating point numbers, and all sorts of other numeric types, but never can two types be in the same array. Python however is perfectly fine with `['spam', 1, 'eggs', 'sausage', 'spam']`. This is because Python stores not the actual contents of the array, but an array of codes that tell the computer where the actual data is stored. This mixing of types and additional degree of seperation remove the ability of a modern processor to do vector operations using the intel math kernel library, which is at the heart of numpy, and the reason all of numpy's array computations are done in C or Fortran, then converted back into Python objects for you to use.

It is for this reason that the execution time of any data processing program you ever write should be mostly bound up in some line that includes numpy. If you write data processing programs like that, you with approach the processing speed of programs written in C and Fortran. You'll never beat such languages in pure execution time, but you can get close enough that the combined time to develop and run the program a few times is faster in Python.

Getting an array to work with

There are three ways to allocate numpy arrays, `np.empty`, `np.zeros`, and `np.ones`. All these take a tuple or integer that defines the shape of the array (integers return 1-d arrays). Allocating an array produces an array that is in some sense empty, though that isn't actually the case. `np.empty` is in some cases much faster than the others because it just gives you the values in memory as they were left when it was deallocated, probably by some other program running on your computer. `np.zeros` and `np.ones` both allocate the memory then overwrite it based on the data type you specify for the array, which means they must access every element in the array. If you are allocating an array that you are going to fill directly with data from another source use `np.empty`. If you're going to do something like add values to the array, you probably want to start with an array that is `0` everywhere. The other way to generate arrays on-the-fly that I use often for demonstration or testing is to use `np.array`, which takes a Python data structure like a list and just produces the `np.ndarray` version.

Indexing numpy ndarrays

Remember that Python uses row-major conventions, unlike science and math. This seems really annoying but there are ways to almost completely dodge the issue of indexing.

Basic indexing and slicing

Since we're in row-major, for a two-dimensional array `my_array[0]` will return the first row. The logical extension might be something like `my_array[0][0]` which is common in lower-level languages and that will work in Python, but it's ugly; `array[0, 0]` is preferred. Now isn't that much more clear? One other feature that Python's indexing supports (for all things that can be indexed) is negative integers. The last element in the structure is at index `-1`, and more negative indices move towards index `0`. Now actually what's going on here with the square brackets is much more interesting than just integer indices let on. You can put as many numbers in those square brackets as you have axes in the array. Numbers aren't the only valid way to index arrays though, you can also use slices. For example, `my_array[1:3]` will produce a new array that contains the same elements as `my_array[1]` and `my_array[2]` put together. So if `my_array` was two-dimensionsal, `my_array[1:3]` has the same number of columns but only two rows, the second and third. If the first or second number in a slice is omitted, it is assumed that the slice extends to the end of the array. Slices also have a third component, the stride. If not provided, it is assumed to be 1. It's also possible to provide only a stride. So `my_array[::2]` is every other item in the array, including the first; these are the items at even-numbered indices. For odd-numbered incides use `my_array[1::2]`. Negative numbers can be used in the stride too, `my_array[::-1]` is a neat way to get an array reversed along the first axis. This basic slicing has a lot of applications, but one that jumps out right away is getting a rectangular section of an array, like `my_array[:10,:10]`.

Indexing with arrays

Arrays can be indexed with arrays of integer or boolean type. To do this, just create your index array, maybe something like `index_array = np.array([3,1,2])` and put that in the square brackets for another array, like `my_array[index_array]`. This will get you an array that contains the elements of `my_array` that correspond to the integers in `index_array`, in the order they appear. This has the potential to be very powerful and also very dangerous. It can be very powerful because you can get the elements in any order you want, maybe such that the elements correspond to some sorting. It is dangerous because there is the potential to use the wrong index array and not produce any error. This is not an issue with the second way to index arrays with arrays.

Indexing arrays with boolean arrays is one of my favorite things in Python. I should briefly mention that all the basic math operations that work with integer and floating point variables in Python will do the same operation on numpy ndarrays, but element-wise. All basic math operations also work between ndarrays and constants. So if I want to get the entries of `array1` where it is greater than `array2`, I can do `array1[array1 > array2]`.

Don't underestimate the power of just indexing an array. Almost all of photometric data processing and a fair part of spectroscopy data processing is just indexing.

Excercise: Photometry data reduction

In the most general sense, we take 4 types of images: biases, darks, flats, and science images. A bias (or bias frame or bias image) is a zero second exposure that measures the what numbers the detector (in this case a CCD camera) records when no signal is present. A dark frame is an exposure taken with the camera's shutter closed; and thereby records only thermal electrons, which are produced at some approximately constant rate dependent on temperature and the properites of an individual pixel. A flat frame is an image taken of a uniformly illuminated object. The best way to do this is by taking images of the sky at sunrise and sunset, but sometimes it is done by taking images of a screen inside the telescope dome. Flats measure a combination of the pixel sensetivity variations and how well the detector is illuminated.

In general, the procedure is as follows: Combine bias frames and subtract the combined bias frames from everything. Combine dark frames, and subtract from flats and science images (possibly rescaling to match exposure time). Combine and normalize the flat field images, then divide the science images by the normalized flat. In the most optimal case, this simplifies to `reduced = (science - dark)/(flat/median(flat))`

The only other ingredient I haven't told you yet that you need to do all this is how to save fits files. Astropy to the rescue, `fits.writeto(filename, data=data, header=header)`. You don't need to supply a header, but I strongly advise that you take the header from the pre-reduction science image and save the reduced data with it. It's a good habit to get into.