L9

June 23, 2017

1 Lecture 9: Vectorized Programming

CSCI 1360E: Foundations for Informatics and Analytics

1.1 Overview and Objectives

We've covered loops and lists, and how to use them to perform some basic arithmetic calculations. In this lecture, we'll see how we can use an external library to make these computations much easier and much faster.

Spoiler alert: if you've programmed in Matlab before, a lot of this will be familiar.

- Understand how to use `import` to add functionality beyond base Python
- Compare and contrast NumPy arrays to built-in Python lists
- Use NumPy arrays in place of explicit loops for basic arithmetic operations

1.2 Part 1: Importing modules

With all the data structures we've discussed so far--lists, sets, tuples, dictionaries, comprehensions, generators--it's hard to believe there's anything else. But oh *man*, is there a big huge world of Python extensions out there.

These extensions are known as *modules*. You've seen at least one in play in your assignments so far:

```
In [ ]: import numpy
```

Anytime you see a statement that starts with `import`, you'll recognize that the programmer is pulling in some sort of external functionality not previously available to Python by default. In this case, the `numpy` package provides a wide range of tools for numerical and scientific applications.

That's just one of **countless** examples...an infinite number that continues to nonetheless increase daily.

It's important to distinguish the hierarchy of packages that exist in the Python ecosystem.

At the first level, there are functions that are available to you by default. Python has a bunch of functionality that comes by default--no `import` required. Remember writing functions to compute the maximum and minimum of a list? Turns out, those already exist by default (sorry everyone):

```
In [1]: x = [3, 7, 2, 9, 4]
        print("Maximum: ", max(x))
        print("Minimum: ", min(x))
```

```
Maximum:  9
Minimum:  2
```

At the second level, there is functionality that comes with Python, but which must still be `import`-ed. Quite a bit of other functionality--still built-in to the default Python environment!--requires explicit `import` statements to unlock. Here are just a couple of examples:

```
In [ ]: import random   # For generating random numbers.
        import os       # For interacting with the filesystem of your computer.
        import re       # For regular expressions. Unrelated: https://xkcd.com/1171/
        import datetime # Helps immensely with determining the date and formatting it.
        import math     # Gives some basic math functions: trig, factorial, exponential, logarit
        import xml      # Abandon all hope, ye who enter.
```

Absolutely any Python installation will come with these. However, you still have to `import` them to access them in your program.

If you are so inclined, you can see the full Python default module index here: https://docs.python.org/3/py-modindex.html.

It's quite a bit! These are all available to you when using Python (don't even bother trying to memorize these; in looking over this list just now, I'm amazed at how many I didn't even know existed).

Once you've imported the module, you can access all its functions via the "dot-notation":

```
In [4]: import random
        random.randint(0, 1)
```

```
Out[4]: 0
```

Dot-notation works by

1. specifying `package_name` (in this case, `random`)
2. followed by a dot: `.`
3. followed by `function_name` (in this case, `randint`, which returns a random integer between two numbers)

As a small tidbit--you can treat imported packages almost like variables, in that you can name them whatever you like, using the `as` keyword in the import statement.

Instead of

```
In [6]: import random
        random.randint(0, 1)
```

```
Out[6]: 1
```

We can tweak it

```
In [7]: import random as r
        r.randint(0, 1)
```

```
Out[7]: 0
```

You can put whatever you want after the `as`, and anytime you call methods from that module, you'll use the name you gave it.

Which brings us to our third and final level of the hierarchy:

At the third level are packages you have to install manually, and then can be `import`-ed There's an ever-expanding universe of 3rd-party modules you can install and use. Anaconda comes prepackaged with quite a few (see the column "In Installer"), and the option to manually install quite a few more.

Again, **don't worry about trying to learn all these**. There are simply too many. You'll come across packages as you need them. For now, we're going to focus on one specific package that is central to most modern data science:

NumPy, short for **Num**erical **Py**thon.

1.3 Part 2: Introduction to NumPy

NumPy, or Numerical Python, is an incredible library of basic functions and data structures that provide a robust foundation for computational scientists.

Put another way: if you're using Python and doing any kind of math, you'll probably use NumPy.

At this point, NumPy is so deeply embedded in so many *other* 3rd-party modules related to scientific computing that even if you're not making *explicit* use of it, at least one of the other modules you're using probably is.

1.3.1 NumPy's core: the `ndarray`

NumPy, or Numerical Python, is an incredible library of basic functions and data structures that provide a robust foundation for computational scientists.

For those of you who attempted the bonus question on A2 dealing with the list-of-lists matrix, here's a recap of what that would look like:

```
In [1]: matrix = [[ 1, 2, 3],
                  [ 4, 5, 6],
                  [ 7, 8, 9] ]
        print(matrix)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Now, imagine using this in a data science context. Indexing would still work as you would expect, but looping through a matrix--say, to do matrix multiplication--would be laborious and highly inefficient.

We'll demonstrate this experimentally later, but suffice to say Python lists embody the drawbacks of using an *interpreted* language such as Python: they're easy to use, but oh so slow.

By contrast, in NumPy, we have the `ndarray` structure (short for "n-dimensional array") that is a highly optimized version of Python lists, perfect for fast and efficient computations. To make use of NumPy arrays, import NumPy (it's installed by default in Anaconda, and on JupyterHub):

```
In [7]: import numpy
```

Now just call the `array` method using our list from before!

```
In [5]: arr = numpy.array(matrix)
        print(arr)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

The variable `arr` is a NumPy array version of the previous list-of-lists!
To reference an element in the array, just use the same notation we did for lists:

```
In [27]: arr[0]

Out[27]: array([1, 2, 3])

In [30]: arr[2][2]

Out[30]: 9
```

You can also separate dimensions by commas, if you prefer:

```
In [31]: arr[2, 2]

Out[31]: 9
```

Either notation is fine. Just remember, with indexing matrices: the first index is the *row*, the second index is the *column*.

1.3.2 NumPy's submodules

NumPy has an impressive array of utility modules that come along with it, optimized to use its `ndarray` data structure. I highly encourage you to use them, even if you're not using NumPy arrays.

1: Basic mathematical routines

All the core functions you could want; for example, all the built-in Python math routines (trig, logs, exponents, etc) all have NumPy versions. (`numpy.sin`, `numpy.cos`, `numpy.log`, `numpy.exp`, `numpy.max`, `numpy.min`)

```
In [9]: a = numpy.array([45, 2, 59, -2, 70, 3, 6, 790])
        print("Minimum:", numpy.min(a))
        print("Cosine of 0th element: {:.2f}".format(numpy.cos(a[0])))

Minimum: -2
Cosine of 1st element: 0.53
```

2: Fourier transforms

If you do any signal processing using Fourier transforms (which we might, later!), NumPy has an entire sub-module full of tools for this type of analysis in `numpy.fft`

(these are beyond the scope of 1360, but if you do any kind of image analysis or analysis of time series data, you'll more than likely make use of FFTs)

3: Linear algebra

We'll definitely be using this submodule later in the course. This is most of your vector and matrix linear algebra operations, from vector norms (`numpy.linalg.norm`) to singular value decomposition (`numpy.linalg.svd`) to matrix determinants (`numpy.linalg.det`).

4: Random numbers

NumPy has a phenomenal random number library in `numpy.random`. In addition to generating uniform random numbers in a certain range, you can also sample from any known parametric distribution.

```
In [6]: print(numpy.random.randint(10))  # Random integer between 0 and 10
        print(numpy.random.randint(10))  # Another one!
        print(numpy.random.randint(10))  # Yet another one!
```

```
5
9
5
```

5: Testing You've probably noticed in your assignments a bizarre NumPy function in the autograder cells: `testing.assert_allclose`.

```
In [10]: a = numpy.array([1, 1])

         numpy.testing.assert_allclose(a, a)
```

This takes two values: an *expected* answer, and a *computed* answer. Put another way, it takes what I thought the correct answer was, and compares it to the answer provided by whatever function you write.

- Provided the two values are "close enough", it silently permits Python to keep running.
- If, however, the two values differ considerably, it actually *crashes* the program with an `AssertionError` (which you've probably seen!).

You can actually tweak how sensitive this definition of "close enough" is. It's a wonderful tool for testing your code!

1.4 Part 3: Vectorized Arithmetic

"Vectorized arithmetic" refers to how NumPy allows you to efficiently perform arithmetic operations on entire NumPy arrays at once, as you would with "regular" Python variables.

For example: let's say I want to square every element in a list of numbers. We've actually done this before:

```
In [11]: # Define our list:
         our_list = [5, 10, 15, 20, 25]
```

```
In [12]: # Write a loop to square each element:
         for i in range(len(our_list)):
             our_list[i] = our_list[i] ** 2  # Set each element to be its own square.
```

```
In [13]: print(our_list)
```

```
[25, 100, 225, 400, 625]
```

Sure, it works. But you might have a nagging feeling in the back of your head that there *has* to be an easier way...

With lists, unfortunately, there isn't one. However, with NumPy arrays, there is! And it's exactly as intuitive as you'd imagine!

```
In [15]: # Define our list:
         our_list = [5, 10, 15, 20, 25]

In [16]: # Convert it to a NumPy array (this is IMPORTANT)
         our_list = numpy.array(our_list)

In [17]: our_list = our_list ** 2  # Yep, we just squared the WHOLE ARRAY. And it works how you'
         print(our_list)
```

```
[ 25 100 225 400 625]
```

NumPy knows how to perform element-wise computations across an entire NumPy array. Whether you want to add a certain quantity to every element, subtract corresponding elements of two NumPy arrays, or square every element as we just did, it allows you to do these operations on all elements at once, **without** writing an explicit loop!

Here's another example: let's say you have a vector and you want to normalize it to be unit length; that involves dividing every element in the vector by a constant (the magnitude of the vector). With lists, you'd have to loop through them manually.

```
In [22]: vector = [4.0, 15.0, 6.0, 2.0]
         # To normalize this to unit length, we need to divide each element by the vector's magn
         # To learn it's magnitude, we need to loop through the whole vector.
         # So. We need two loops!
         magnitude = 0.0
         for element in vector:
             magnitude += element ** 2
         magnitude = (magnitude ** 0.5)  # square root
         print("Original magnitude:", magnitude)
```

```
Original magnitude: 16.76305461424021
```

```
In [23]: # Now that we have the magnitude, we need to loop through the list AGAIN,
         # dividing each element of the list by the magnitude.
         new_magnitude = 0.0
         for index in range(len(vector)):
             element = vector[index]
             vector[index] = element / magnitude
             new_magnitude += vector[index] ** 2
         new_magnitude = (new_magnitude ** 0.5)
         print("Normalized magnitude:", new_magnitude)
```

```
Normalized magnitude: 0.9999999999999999
```

Yeah, it's pretty complicated. The sad part, though, is that MOST of that complication comes from the loops you have to write!

So... let's see if vectorized computation can help!

```
In [24]: import numpy as np  # This tends to be the "standard" convention when importing NumPy.
         import numpy.linalg as nla

In [25]: vector = [4.0, 15.0, 6.0, 2.0]
         np_vector = np.array(vector)  # Convert to NumPy array.
         magnitude = nla.norm(np_vector)  # Computing the magnitude: a friggin' ONE-LINER!
         print("Original magnitude:", magnitude)

Original magnitude: 16.7630546142

In [26]: np_vector /= magnitude  # Vectorized division!!! No loop needed!
         new_magnitude = nla.norm(np_vector)
         print("Normalized magnitude:", new_magnitude)

Normalized magnitude: 1.0
```

No loops needed, far fewer lines of code, and a simple intuitive operation.

Operations involving arrays on both sides of the sign will also work (though the two arrays need to be the same length).

For example, adding two vectors together:

```
In [24]: x = np.array([1, 2, 3])
         y = np.array([4, 5, 6])
         z = x + y
         print(z)

[5 7 9]
```

Works exactly as you'd expect, but no [explicit] loop needed.

This becomes particularly compelling with matrix multiplication. Say you have two matrices, A and B:

```
In [25]: A = np.array([ [1, 2], [3, 4] ])
         B = np.array([ [5, 6], [7, 8] ])
```

If you recall from algebra, matrix multiplication $A \times B$ involves multipliying each *row* of A by each *column* of B. But rather than write that code yourself, Python (as of version 3.5) gives us a dedicated matrix multiplication operator: the `@` symbol!

```
In [26]: A @ B
```

```
Out[26]: array([[19, 22],
                [43, 50]])
```

In almost every case, vectorized operations are far more efficient than loops written in Python to do the same thing.

But don't take my word for it--let's test it!

```
In [27]: # This function does matrix-matrix multiplication using explicit loops.
         # There's nothing wrong with this! It'll just be a lot slower than NumPy...
         def multiply_loops(A, B):
             C = np.zeros((A.shape[0], B.shape[1]))
             for i in range(A.shape[1]):
                 for j in range(B.shape[0]):
                     C[i, j] = A[i, j] * B[j, i]
             return C

In [28]: # And now a function that uses NumPy's matrix-matrix multiplication operator.
         def multiply_vector(A, B):
             return A @ B
```

We've got our functions. Now, using a handle timer tool, we can run them both on some sample data and see how fast they go!

```
In [29]: # Here's our sample data: two randomly-generated, 100x100 matrices.
         X = np.random.random((100, 100))
         Y = np.random.random((100, 100))

In [30]: # First, using the explicit loops:
         %timeit multiply_loops(X, Y)

4.23 ms ± 107 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

It took about 4.23 milliseconds (that's $4.23 * 10^{-3}$ seconds) to perform 1 matrix-matrix multiplication. Certainly not objectively slow! But let's see how the NumPy version does...

```
In [31]: # Now, the NumPy multiplication:
         %timeit multiply_vector(X, Y)

46.6 µs ± 346 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

46.6 *micro*seconds per multiplication--that's $46.4 * 10^{-6}$ seconds, or *two orders of magnitude faster*!

tl;dr: if you're writing loops in conjunction with NumPy arrays, see if there's any way to use vectorized operations instead.

1.4.1 In summary

- NumPy arrays have all the abilities of lists (indexing, mutability, slicing) plus a whole lot of additional benefits, such as *vectorized computations*.
- About the only limitation of NumPy arrays relative to Python lists is constructing them: with lists, you can build them through generators or the `append()` method, but you can't do this with NumPy arrays. Therefore, if you're building an array from scratch, the best option would be to build the list and then pass that to `numpy.array()` to convert it. Adjusting the length of the NumPy array *after* it's constructed is more difficult.
- The Python ecosystem is *huge*. There is some functionality that comes with Python by default, and some of this default functionality is available immediately; the other default functionality is accessible using `import` statements. There is even more functionality from 3rd-party vendors, but it needs to be installed before it can be `imported`. NumPy falls in this lattermost category.
- Vectorized operations are always, always preferred to loops. They're easier to write, easier to understand, and in almost all cases, much more efficient.

1.5 Review Questions

Some questions to discuss and consider:

1: NumPy arrays have an attribute called `.shape` that will return the dimensions of the array in the form of a tuple. If the array is just a vector, the tuple will only have 1 element: the length of the array. If the array is a matrix, the tuple will have 2 elements: the number of rows and the number of columns. What will the `shape` tuple be for the following array: `tensor = np.array([ [ [1, 2], [3, 4] ], [ [5, 6], [7, 8] ], [ [9, 10], [11, 12] ] ])`

2: Vectorized computations may seem almost like magic, and indeed they are, but at the end of the day there has to be a loop *somewhere* that performs the operations. Given what we've discussed about interpreted languages, compiled languages, and in particular how the delineations between the two are blurring, what would your best educated guess be (ideally without Google's help) as to where these loops actually happen that implemented the vectorized computations?

3: Does the answer to the above question change your perspective on whether Python is a compiled or an interpreted language?

4: Using your knowledge of slicing from a few lectures ago, and your knowledge from this lecture that NumPy arrays also support slicing, let's take an example of selecting a sub-range of rows from a two-dimensional matrix. Write the notation you would use for slicing out / selecting all the rows *except* for the first one, while retaining all the columns (hint: by just using `:` as your slicing operator, with no numbers, this means "everything").

1.6 Course Administrivia

- **How is A3 going?** It's due Saturday night!
- **How is A4 going?** It's due next Monday. One of the questions makes use of the concepts taught here!
- **In previous experience, this and next Monday's lectures are easily the most difficult--as lots of things are happening behind the scenes with vectorized computations--but also the most important**. So...

- **If you're having trouble with the material, please ask!** Post first in the `#questions` channel, because there's no way someone else doesn't have the same question. I'm also more than happy to meet 1-on-1, either in person on campus, or remotely via Slack calls.

1.7 Additional Resources

1. Grus, Joel. *Data Science from Scratch*. 2015. ISBN-13: 978-1491901427
2. McKinney, Wes. *Python for Data Analysis*. 2012. ISBN-13: 860-1400898857
3. NumPy Quickstart Tutorial: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html