Notes on using arrays in Python

In Python, you've been using three-component NumPy arrays for a while to represent vectors. However, NumPy arrays are much more general than that.

A NumPy array can have any number of dimensions: you may make 3-component arrays like you have been, but you can also make a 50×3 array (which you might do in order to hold 50 vectors, for the positions of your vibrating string elements), a 50×50 array to hold a large matrix, or even a monstrosity like a $1920 \times 1080 \times 3 \times 600$ array (to hold the red/green/blue components of 1920×1080 pixels for each of 600 frames of a movie).

For this project, you will need to create some arrays that are $N + 1 \times 3$ for the positions and velocities of all of the masses that comprise the string.

0.1 Creating an array

Before you use an array, you will need to tell Python its dimensions. (This is rather like declaring variables in C.) You may do this with the `zeroes` function to create an array of your choice of size and shape:

`position = zeros((50, 3))` will create an array that is 50×3 and store it as `position`.

Once you've done this, you may address any element with syntax like `position[30,2]`. If x, y, and z are the 0, 1, and 2 elements of your vectors, then `position[30,2]` means "the z-component of the 30th position".

0.2 Manipulating arrays, the traditional way

Arrays and `for` loops are great friends. Suppose you wanted to create a multiplication table. Try the following code:

```
from numpy import *

timestable = zeros((13,13))

for i in range(13):
    for j in range(13):
        timestable[i,j]=i*j

print (timestable)
```

Notice how multiple indices are separated by a comma and that `print` does sensible things when you ask it to print arrays.

You can thus use `for` loops to iterate over your arrays.

0.3 Array slicing

In NumPy, you have already seen code that does mathematics with arrays. Consider the following code to simulate planetary orbits:

```
from numpy import *

G = 4*pi**2
pos = array([1.0,0.0,0.0])
vel = array([0.0,2*pi,0.0])
dt = 1e-2

while True:
    pos += vel * dt/2
    vel += -pos*G/(linalg.norm(pos) ** 3) * dt
    pos += vel * dt/2
    print ("C 1 1 0\nc3 0 0 0 0.1\nC 0.3 0.3 1\nct3 0 %e %e %e\nF\n" % (pos[0],pos[1],pos[2]))
```

In the leapfrog update, note that writing `pos += vel * dt/2` treats the entire `pos` array (which we are using to represent a Cartesian vector) as one object, updating all three components at once. The rule is that *if you omit an index, Python will loop over it.* The above code is equivalent to the following:

```
from numpy import *

G = 4*pi**2
pos = array([1.0,0.0,0.0])
vel = array([0.0,2*pi,0.0])
dt = 1e-2

while True:
    for i in range(3):
        pos[i] += vel[i] * dt/2
    for i in range(3):
        vel[i] += -pos[i]*G/(linalg.norm(pos) ** 3) * dt
    for i in range(3):
        pos[i] += vel[i] * dt/2
print ("C 1 1 0\nc3 0 0 0 0.1\nC 0.3 0.3 1\nct3 0 %e %e %e\nF\n" % (pos[0],pos[1],pos[2]))
```

However, if you have two-dimensional arrays, things can get a little bit more complex. Suppose that you are simulating a vibrating string with 100 points and have generated two 100×3 arrays called `position` and `velocity` to hold the position and velocity of the masses. The first index tells you which point you're talking about and runs from 0 to 99; the second index tells you which component (x, y, or z), and runs from 0 to 2. You can address those arrays as the following:

- `position[50,1]` – means the y-component of mass number 50, a single number
- `position[50]` – means the vector position of mass number 50, a three-component object
- `position` – means the entire array, a 100×3 object

Replacing an index with a colon tells NumPy to iterate over that index. So for instance:

- `position[:,1]` – means the y-component of all of the masses, a 50-component object

So, for instance, all of the following codes do the same thing:

First:

```
position = position + velocity      // update every component of every point at once
```

Second:

```
for i in range(100):                          // iterate over every point...
    position[i] = position[i] + velocity[i] // ... and update all three components of its position
```

Third:

```
for i in range(100):                                    // iterate over every point...
    for j in range(3):                                  // ... and over Cartesian directions
        position[i,j] = position[i,j] + velocity[i,j] // ... and update that component of that point
```

Fourth:

```
for i in range(3):                                  // iterate over every Cartesian direction...
    position[:,i] = position[:,i] + velocity[:,i] // ... and update that direction for all
                                                    // 100 points at once
```

This behavior lets you avoid writing out explicit `for` loops when writing NumPy code. For our class, if there is ever anything you'd like to do (for instance: "add this vector to all the vectors in this array", "add 2 to all of the y-components of the vectors in this list", etc.), please ask; we'll be happy to show you one-on-one how to do whatever you want.