NumPy Vectorization

This lesson teaches Numpy vectorization and explains it with a simple example using object-oriented, procedural and vectorized approach.

We'll cover the following

Vectorization, in simple words, means optimizing the algorithm so that it can run multiple operations from a single instruction. NumPy is all about vectorization. If you are familiar with Python, this is the main difficulty you'll face because you'll need to change your way of thinking and your new friends (among others) are named "vectors", "arrays", "views" or "ufuncs".

> **Note:** A custom magic command timeit is used in all codes. It's a tool for measuring the execution time of small code snippets.

Object Oriented Approach

Let's take a very simple example, a random walk. One possible object-oriented approach would be to define a `RandomWalker` class and write a walk method that would return the current position after each (random) step. It's nice, it's readable, but it is slow:

main.py

tools.py

```
from tools import timeit #get time it from tools.py(custom modu
import random
class RandomWalker:
    def __init__(self):
                self.position = 0
    def walk(self, n): # walk method
        self.position = 0
        for i in range(n):
            yield self.position
            self.position += 2*random.randint(0, 1) - 1
            #returns current position after each random step

walker = RandomWalker() # make instance of class walk
walk = [position for position in walker.walk(1000)]#call the wa

walker = RandomWalker()
timeit("[position for position in walker.walk(n=10000)]", globa
#calculates the  total loops and time per loop

```

Here loops are the total number of CPU cycles required during a random walk and the time in msec indicates time per cycle.

Procedural Approach

For such a simple problem, we can probably save the class definition and concentrate only on the walk method that computes successive positions after each random step. This new method saves some CPU cycles but not that much because of this function is pretty much the same as in the object-oriented approach and the few cycles we saved probably come from the inner Python object-oriented machinery.

main.py

tools.py

```
from tools import timeit #get timeit from tools.py (custom module)
import random
def random_walk(n):
    position = 0
    walk = [position]
    for i in range(n):
```

```
        position += 2*random.randint(0, 1)-1 #position takes up random values
        walk.append(position)# append position to walk
    return walk

walk = random_walk(1000) #call the function random_walk
timeit("random_walk(n=10000)", globals()) # calculates the total loops and time per loop
```

Here we can see that the time taken by the procedural approach is less than that of the object-oriented approach.

Vectorized Approach

For the vectorized approach, we can use Itertools or NumPy.

1. Itertools

Itertools is a python module that offers *a set of functions creating iterators for efficient looping*. If we observe that a random walk is an accumulation of steps, we can rewrite the function by first generating all the steps and accumulate them without any loop:

main.py

tools.py

```
from tools import timeit #get timeit function from tools.py(custom module)
from itertools import accumulate #get accumulate function from built accumulate module
import random
def random_walk_faster(n=1000):
  steps = random.choices([-1,+1], k=n)
  return [0]+list(accumulate(steps))#get the total number of steps

walk = random_walk_faster(1000)
timeit("random_walk_faster(n=10000)", globals())# calculates the total loops and time per lo
```

In fact, we've just *vectorized* our function. Instead of looping for picking sequential steps and add them to the current position, we first generated all the steps at once and used the accumulate function to compute all the positions. We got rid of the loop and this makes things faster.

2. Numpy

We gained 85% of computation-time compared to the previous version, not so bad. But the advantage of this new version is that it makes NumPy vectorization super simple. We just have to translate itertools call into NumPy ones:

main.py

tools.py

```
from tools import timeit #get timeit function from tools.py(custom module)
import numpy as np
def random_walk_fastest(n=1000):
    # No 's' in NumPy choice (Python offers choice & choices)
    steps = np.random.choice([-1,+1], n)
    return np.cumsum(steps) #return the cumulative sum of the steps along a given axis.

walk = random_walk_fastest(1000)
timeit("random_walk_fastest(n=1000)", globals())
#calculates the total loops and time per loop
```

Not too difficult, but we gained a factor 500x using NumPy.

Solve this quiz!

1

What's a good alternative in Numpy for the "accumulate" method from Itertools?

A) `Numpy.sum()`

B) `Numpy.cumsum()`

C) `Numpy.add()`

D) None of the above

COMPLETED 0%

1 of 2

This course is about vectorization, be it at the code or problem level. We'll see this difference is important before looking at custom vectorization.

In the next lesson, we'll learn about "Readability vs. Speed".