

Introduction

Memory layout

Views and Copies

#### **NumPy Vectorization**

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



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:

```
from tools import timeit #get time it from tools.py(custom module)
main.py
                                   import random
                                  class RandomWalker:
                                    def __init__(self):
tools.py
                                             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
                              13 walker = RandomWalker() # make instance of class walk
                                  walk = [position for position in walker.walk(1000)]#call the walk function
                              16 walker = RandomWalker()
                                  timeit("[position for position in walker.walk(n=10000)]", globals())
    RUN
                                                                                       SAVE
                                                                                                              0
                                                                                                             ×
                                                                                                        1.751s
Output
 10 loops, best of 3: 31.8 msec 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.

```
from tools import timeit #get timeit from tools.py (custom module)
main.py
                                  import random
                                  def random_walk(n):
tools.py
                                      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
                              11 walk = random_walk(1000) #call the function random_walk
                              12 timeit("random_walk(n=10000)", globals()) # calculates the total loops and time
    RUN
                                                                                        SAVE
                                                                                                              0
                                                                                                             ×
Output
                                                                                                        1.579s
10 loops, best of 3: 29.9 msec 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

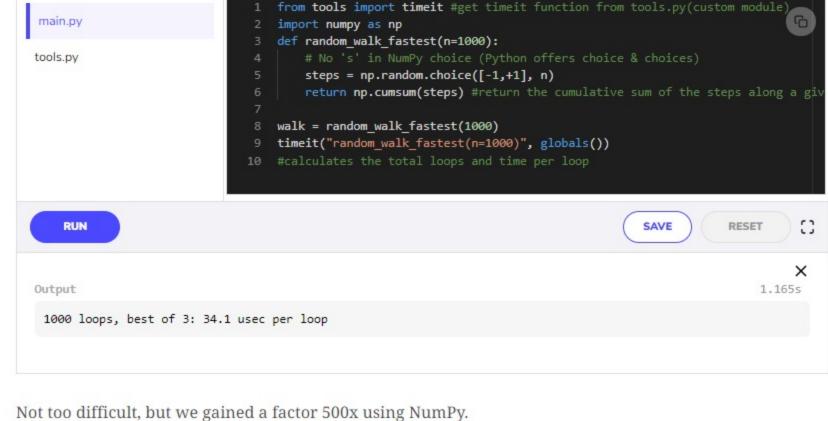
Itertools is a python module that offers a set of functions creating iterators for efficient looping. If we observe that a random walk is an accumulation of steps, we can rewrite the function by first generating all the steps and accumulate them without any loop:

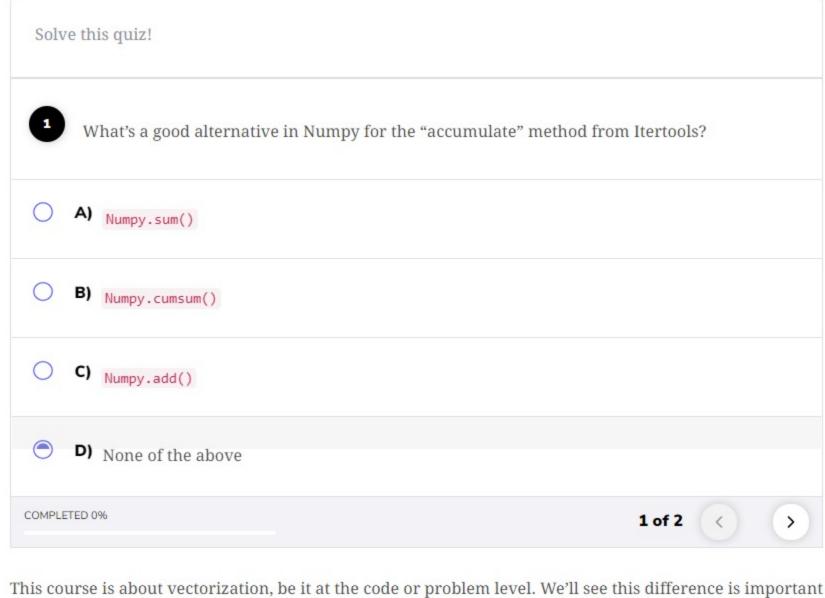
```
from tools import timeit #get timeit function from tools.py(custom module
main.py
                                   from itertools import accumulate #get accumulate function from built accumulate
                                   import random
tools.py
                                4 def random_walk_faster(n=1000):
                                     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 as
    RUN
                                                                                         SAVE
                                                                                                      RESET
                                                                                                               ×
Output
                                                                                                          1.6025
10 loops, best of 3: 6.71 msec per loop
```

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:





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

before looking at custom vectorization.

