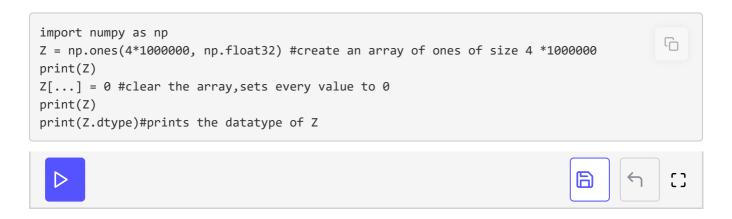
Introduction

In this lesson, we'll learn how to maximize the speed using NumPy!

This chapter explains the basic anatomy of NumPy arrays, especially regarding the memory layout, view, copy and the data type. They are critical notions to understand if you want your computation to benefit from NumPy philosophy.

Let's consider a simple example where we want to clear all the values from an array which has the data type <code>np.float32</code>. How does one write it to **maximize speed**? The below syntax is rather obvious (at least for those familiar with NumPy) but the above question asks to find the fastest operation.



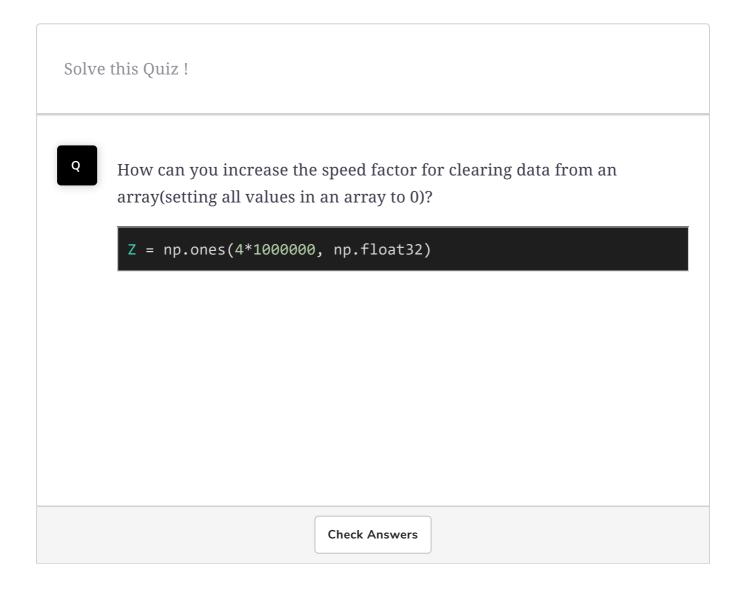
If you look more closely at both the <a href="https://dtpe.com/dtp

main.py		C
tools.py		
	timeit from tools.py(custom module) at32) #create an array of size 4*10000000 np.float32	
<pre>print("np.float16:")</pre>		

```
#time required to view array as np.float16
timeit("Z.view(np.float16)[...] = 0", globals())
print("np.int16:")
#time required to view array as np.int16
timeit("Z.view(np.int16)[...] = 0", globals())
print("np.int32:")
#time required to view array as np.int32
timeit("Z.view(np.int32)[...] = 0", globals())
print("np.float32:")
#time required to view array as np.float32
timeit("Z.view(np.float32)[...] = 0", globals())
print("np.int64:")
#time required to view array as np.int64
timeit("Z.view(np.int64)[...] = 0", globals())
print("np.float64:")
#time required to view array as np.float64
timeit("Z.view(np.float64)[...] = 0", globals())
print("np.complex128:")
#time required to view array as np.complex128
timeit("Z.view(np.complex128)[...] = 0", globals())
print("np.int8:")
#time required to view array as np.int8
timeit("Z.view(np.int8)[...] = 0", globals())
print("np.float16:")
#time required to view array as np.float16
timeit("Z.view(np.float16)[...] = 0", globals())
print("np.int16:")
#time required to view array as np.int16
timeit("Z.view(np.int16)[...] = 0", globals())
print("np.int32:")
#time required to view array as np.int32
timeit("Z.view(np.int32)[...] = 0", globals())
print("np.float32:")
#time required to view array as np.float32
timeit("Z.view(np.float32)[...] = 0", globals())
print("np.int64:")
#time required to view array as np.int64
timeit("Z.view(np.int64)[...] = 0", globals())
print("np.float64:")
#time required to view array as np.float64
timeit("Z.view(np.float64)[...] = 0", globals())
print("np.complex128:")
#time required to view array as np.complex128
timeit("Z.view(np.complex128)[...] = 0", globals())
print("np.int8:")
#time required to view array as np.int8
```



Here timeit is a custom function used. Interestingly enough, the obvious way of clearing all the values is not the fastest. The total number of CPU cycle to execute each above instruction are 100 but the two instruction take less time per loop. By casting the array into a larger data type such as <code>np.float64</code>, we gained a **25% speed factor**. But, by viewing the array as a byte array (<code>np.int8</code>), we gained a **50% factor**. The reason for such speedup is to be found in the internal NumPy machinery and the compiler optimization.



This simple example illustrates the philosophy of NumPy. Let's move on to the next lesson to learn memory layouts.