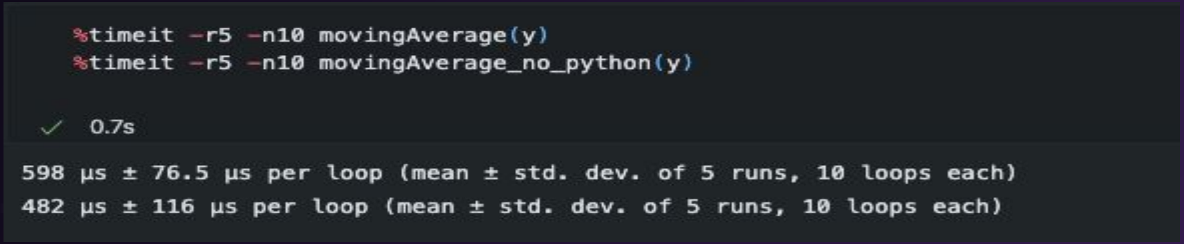
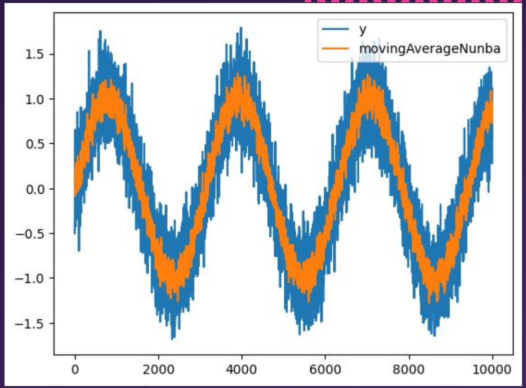


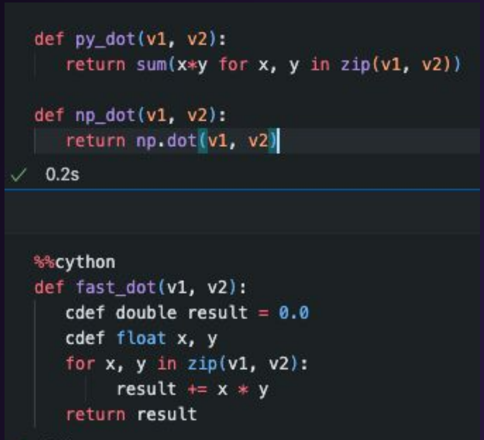
```
%timeit -r5 -n10 movingAverage(y)
%timeit -r5 -n10 movingAverage_no_python(y)

✓ 0.7s

598 µs ± 76.5 µs per loop (mean ± std. dev. of 5 runs, 10 loops each)
482 µs ± 116 µs per loop (mean ± std. dev. of 5 runs, 10 loops each)
```

Aicha cheridi : 23138371

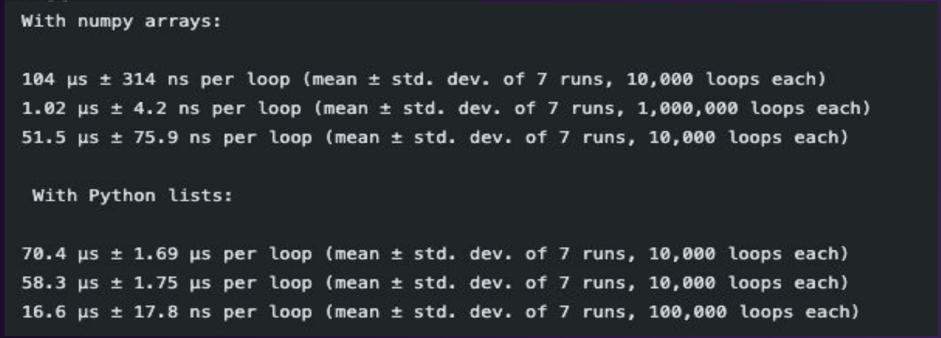




```
def py_dot(v1, v2):
    return sum(x*y for x, y in zip(v1, v2))

def np_dot(v1, v2):
    return np.dot(v1, v2)
✓ 0.2s

%%cython
def fast_dot(v1, v2):
    cdef double result = 0.0
    cdef float x, y
    for x, y in zip(v1, v2):
        result += x * y
    return result
```



```
With numpy arrays:

104 µs ± 314 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
1.02 µs ± 4.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
51.5 µs ± 75.9 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

 With Python lists:

70.4 µs ± 1.69 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
58.3 µs ± 1.75 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
16.6 µs ± 17.8 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

- Answer to question A: njit indicated the use of numba nopython mode which baypasses Python interprete whereas jit decorator uses python interpreter.
- Answer to question B: because the first time it compiles the code thus take more time
- Answer to question C:
  a. using Numpy arrays the fastest is **np_dot** because it used vectorized and the input vectors are as well numpy arrays
  b. Using python lists : the fastest is **fast_dot** because cython compiles the code that was improved with some C language declarations as well the input are lists and not vectorized (thus numpy is not optimized)
- Answer to question D: No it does not affect the measurements becaus the number of iterations does not affect this libraries performance