MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

NATIONAL TECHNICAL UNIVERSITY

«KHARKIV POLYTECHNIC INSTITUTE»

Department of Software Engineering and Management Information Technologies

Report from lab № 3
discipline «Data processing Python»

Executed by:

Student of group KN.218g.e

Y. S. Borzova

Checked by:

Kizilov O. S

Kharkiv

2020

Theme: Numpy Package in Python

Main part:

Task 1. Count the product of nonzero elements on the diagonal of a rectangular matrix. For X = np.array ([[1, 0, 1], [2, 0, 2], [3, 0, 3], [4, 4, 4]]) answer 3.

Task 2. Given a matrix X and two vectors of the same length i and j. Build the vector np.array ([X [i [0], j [0]], X [i [1], j [1]],..., X [i [N-1], j [N-1]]]).

Task 3. Two vectors x and y are given. Check if they specify the same multiset. For x = np.array ([1, 2, 2, 4]), y = np.array ([4, 2, 1, 2]) the answer is True.

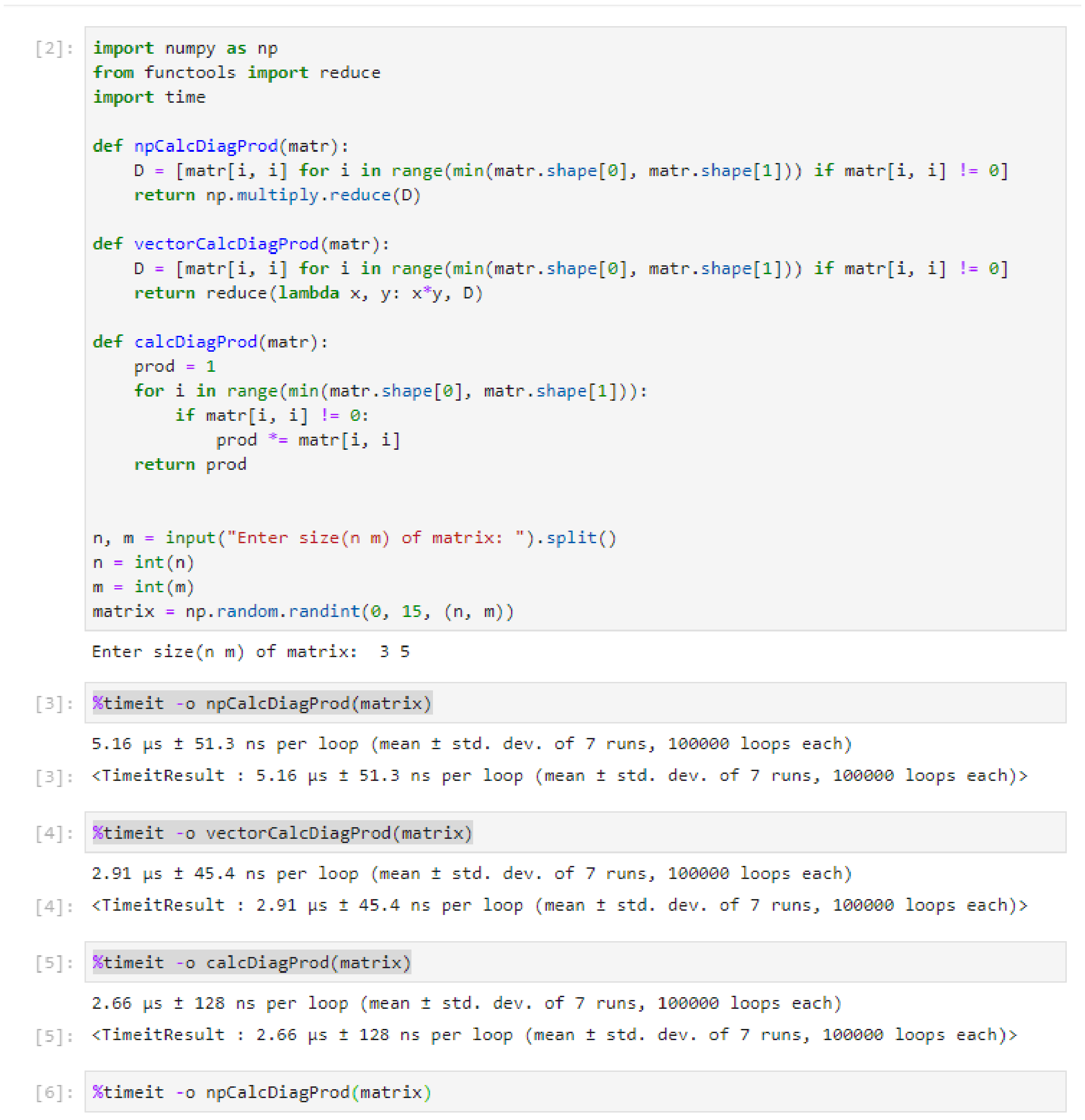
Task 4. Find the maximum element in the vector x among the elements before which it is zero. For x = np.array ([6, 2, 0, 3, 0, 0, 5, 7, 0]) answer 5.

Task 5. Given a three-dimensional array containing an image of size (height, width, numChannels), as well as a vector numChannels lengths. Add the channels of the image with the indicated weights, and return the result as a size matrix (height, width). You can read the real image using the scipy.misc.imread function (if image is not in png format, install the pillow package: conda install pillow). Convert the color image to shades of gray using the coefficients np.array ([0.299, 0.587, 0.114]).

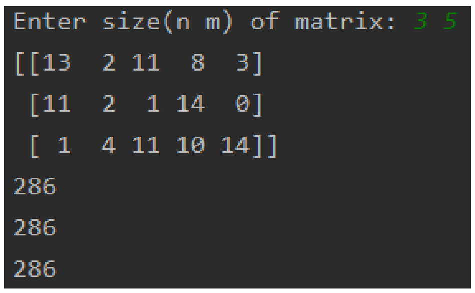
Task 6. Implement run-length encoding. Dan vector x. It is necessary to return a tuple of two
vectors of the same length. The first contains numbers, and the second - how many times they need to be repeated. Example: x = np.array ([2, 2, 2, 3, 3, 3, 5]). Answer: (np.array ([2, 3, 5]), np.array ([3, 3, 1])).

Task 7. Two samples of objects are given - X and Y. Calculate the matrix of Euclidean distances between objects. Compare with scipy.spatial.distance.cdist function.

Task 8. Implement the function of calculating the logarithm of the density of a multidimensional normal distribution parameters: points X, size (N, D), mat. expectation m, vector of length D, covariance matrix C, size (D, D). It is allowed to use library functions to calculate the determinant of the matrix, as well as the inverse matrices, including the non-vectorized version. Compare with scipy.stats.multivariate_normal (m, C) .logpdf (X) both in terms of speed and accuracy of calculations.

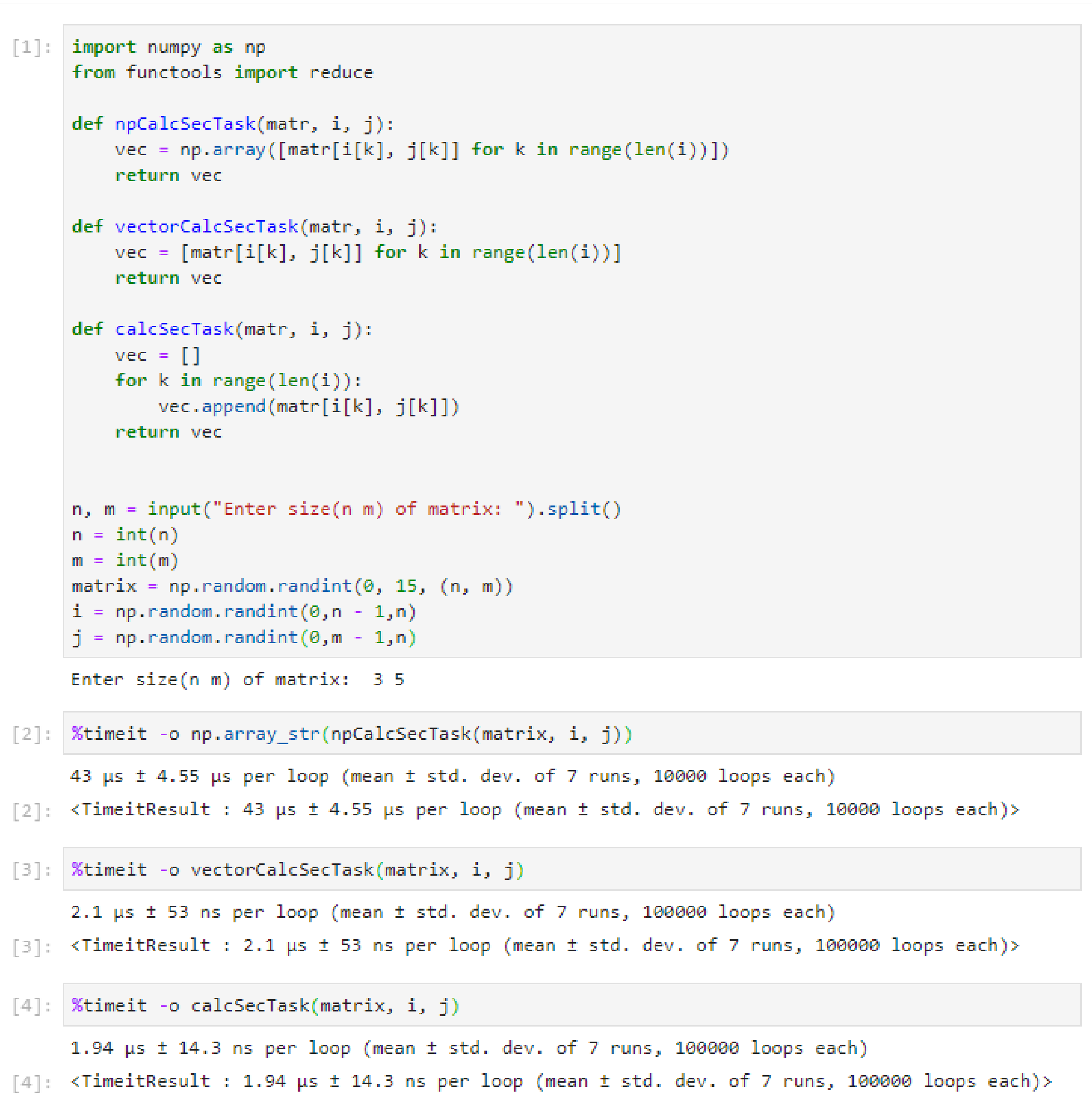


```
[2]: import numpy as np
     from functools import reduce
     import time

     def npCalcDiagProd(matr):
         D = [matr[i, i] for i in range(min(matr.shape[0], matr.shape[1])) if matr[i, i] != 0]
         return np.multiply.reduce(D)

     def vectorCalcDiagProd(matr):
         D = [matr[i, i] for i in range(min(matr.shape[0], matr.shape[1])) if matr[i, i] != 0]
         return reduce(lambda x, y: x*y, D)

     def calcDiagProd(matr):
         prod = 1
         for i in range(min(matr.shape[0], matr.shape[1])):
             if matr[i, i] != 0:
                 prod *= matr[i, i]
         return prod


     n, m = input("Enter size(n m) of matrix: ").split()
     n = int(n)
     m = int(m)
     matrix = np.random.randint(0, 15, (n, m))

     Enter size(n m) of matrix:  3 5

[3]: %timeit -o npCalcDiagProd(matrix)

     5.16 µs ± 51.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[3]: <TimeitResult : 5.16 µs ± 51.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[4]: %timeit -o vectorCalcDiagProd(matrix)

     2.91 µs ± 45.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[4]: <TimeitResult : 2.91 µs ± 45.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[5]: %timeit -o calcDiagProd(matrix)

     2.66 µs ± 128 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[5]: <TimeitResult : 2.66 µs ± 128 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[6]: %timeit -o npCalcDiagProd(matrix)
```



```
Enter size(n m) of matrix: 3 5
[[13  2 11  8  3]
 [11  2  1 14  0]
 [ 1  4 11 10 14]]
286
286
286
```

Figure 1 – Task 1



```
[1]: import numpy as np
     from functools import reduce

     def npCalcSecTask(matr, i, j):
         vec = np.array([matr[i[k], j[k]] for k in range(len(i))])
         return vec

     def vectorCalcSecTask(matr, i, j):
         vec = [matr[i[k], j[k]] for k in range(len(i))]
         return vec

     def calcSecTask(matr, i, j):
         vec = []
         for k in range(len(i)):
             vec.append(matr[i[k], j[k]])
         return vec


     n, m = input("Enter size(n m) of matrix: ").split()
     n = int(n)
     m = int(m)
     matrix = np.random.randint(0, 15, (n, m))
     i = np.random.randint(0,n - 1,n)
     j = np.random.randint(0,m - 1,n)
```

```
Enter size(n m) of matrix:  3 5
```

```
[2]: %timeit -o np.array_str(npCalcSecTask(matrix, i, j))
```

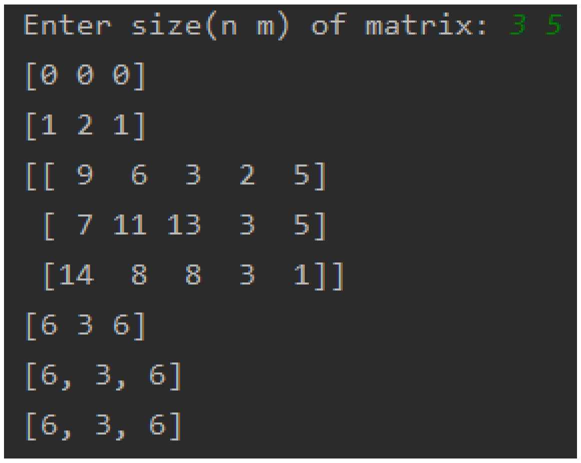
```
     43 µs ± 4.55 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
[2]: <TimeitResult : 43 µs ± 4.55 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)>
```

```
[3]: %timeit -o vectorCalcSecTask(matrix, i, j)
```

```
     2.1 µs ± 53 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[3]: <TimeitResult : 2.1 µs ± 53 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>
```

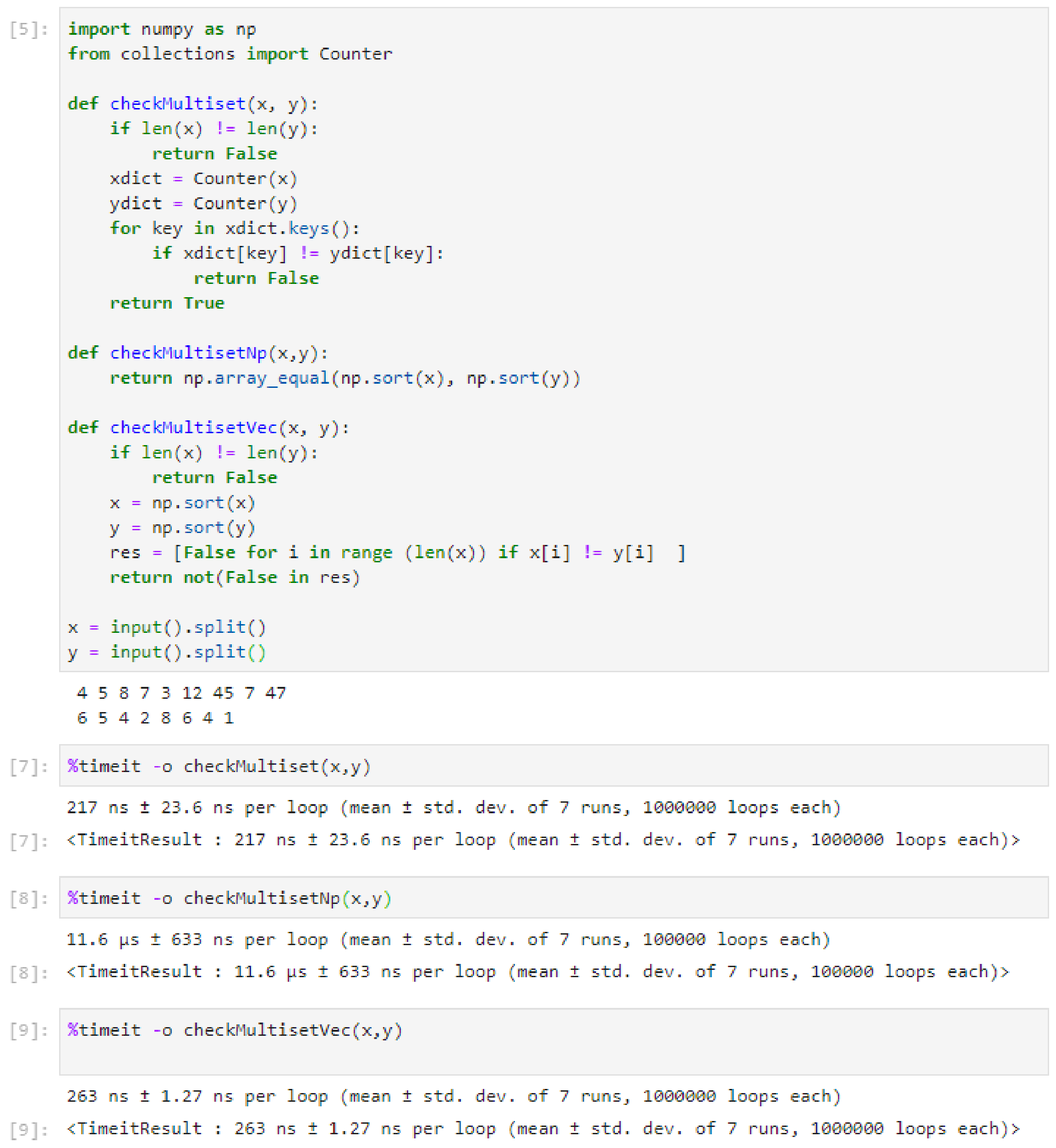
```
[4]: %timeit -o calcSecTask(matrix, i, j)
```

```
     1.94 µs ± 14.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[4]: <TimeitResult : 1.94 µs ± 14.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>
```

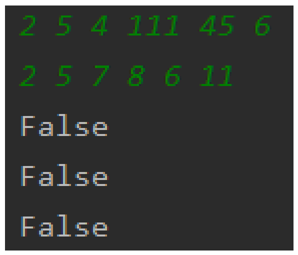


```
Enter size(n m) of matrix: 3 5
[0 0 0]
[1 2 1]
[[ 9  6  3  2  5]
 [ 7 11 13  3  5]
 [14  8  8  3  1]]
[6 3 6]
[6, 3, 6]
[6, 3, 6]
```

Figure 2 – Task 2

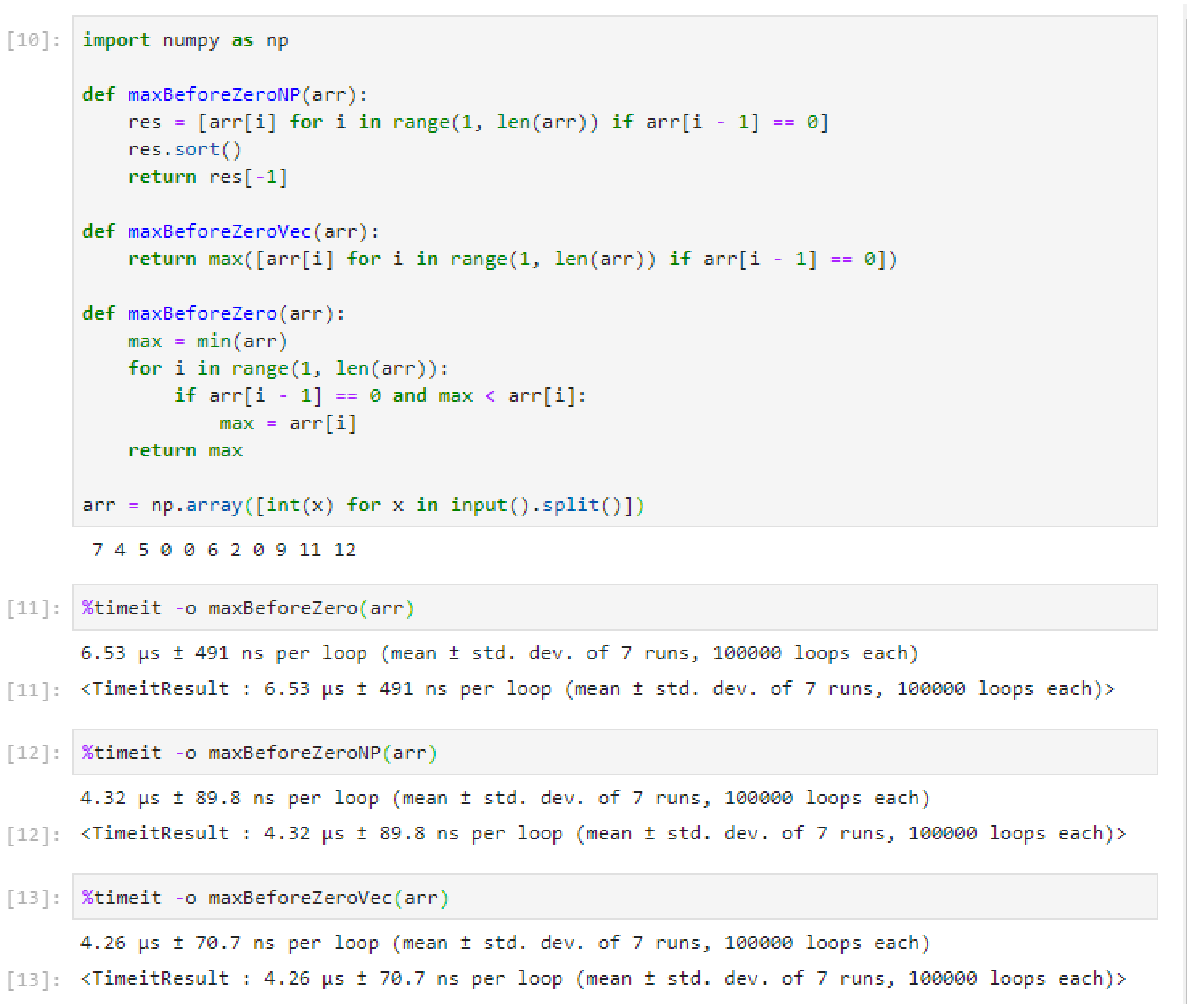


```
[5]: import numpy as np
     from collections import Counter

     def checkMultiset(x, y):
         if len(x) != len(y):
             return False
         xdict = Counter(x)
         ydict = Counter(y)
         for key in xdict.keys():
             if xdict[key] != ydict[key]:
                 return False
         return True

     def checkMultisetNp(x,y):
         return np.array_equal(np.sort(x), np.sort(y))

     def checkMultisetVec(x, y):
         if len(x) != len(y):
             return False
         x = np.sort(x)
         y = np.sort(y)
         res = [False for i in range (len(x)) if x[i] != y[i]  ]
         return not(False in res)

     x = input().split()
     y = input().split()

      4 5 8 7 3 12 45 7 47
      6 5 4 2 8 6 4 1

[7]: %timeit -o checkMultiset(x,y)

     217 ns ± 23.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
[7]: <TimeitResult : 217 ns ± 23.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)>

[8]: %timeit -o checkMultisetNp(x,y)

     11.6 µs ± 633 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[8]: <TimeitResult : 11.6 µs ± 633 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[9]: %timeit -o checkMultisetVec(x,y)

     263 ns ± 1.27 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
[9]: <TimeitResult : 263 ns ± 1.27 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)>
```

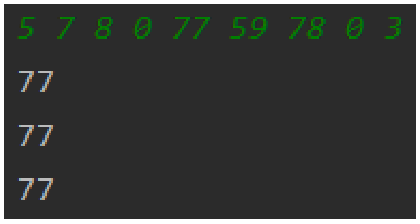


```
2 5 4 111 45 6
2 5 7 8 6 11
False
False
False
```

Figure 3 – Task 3

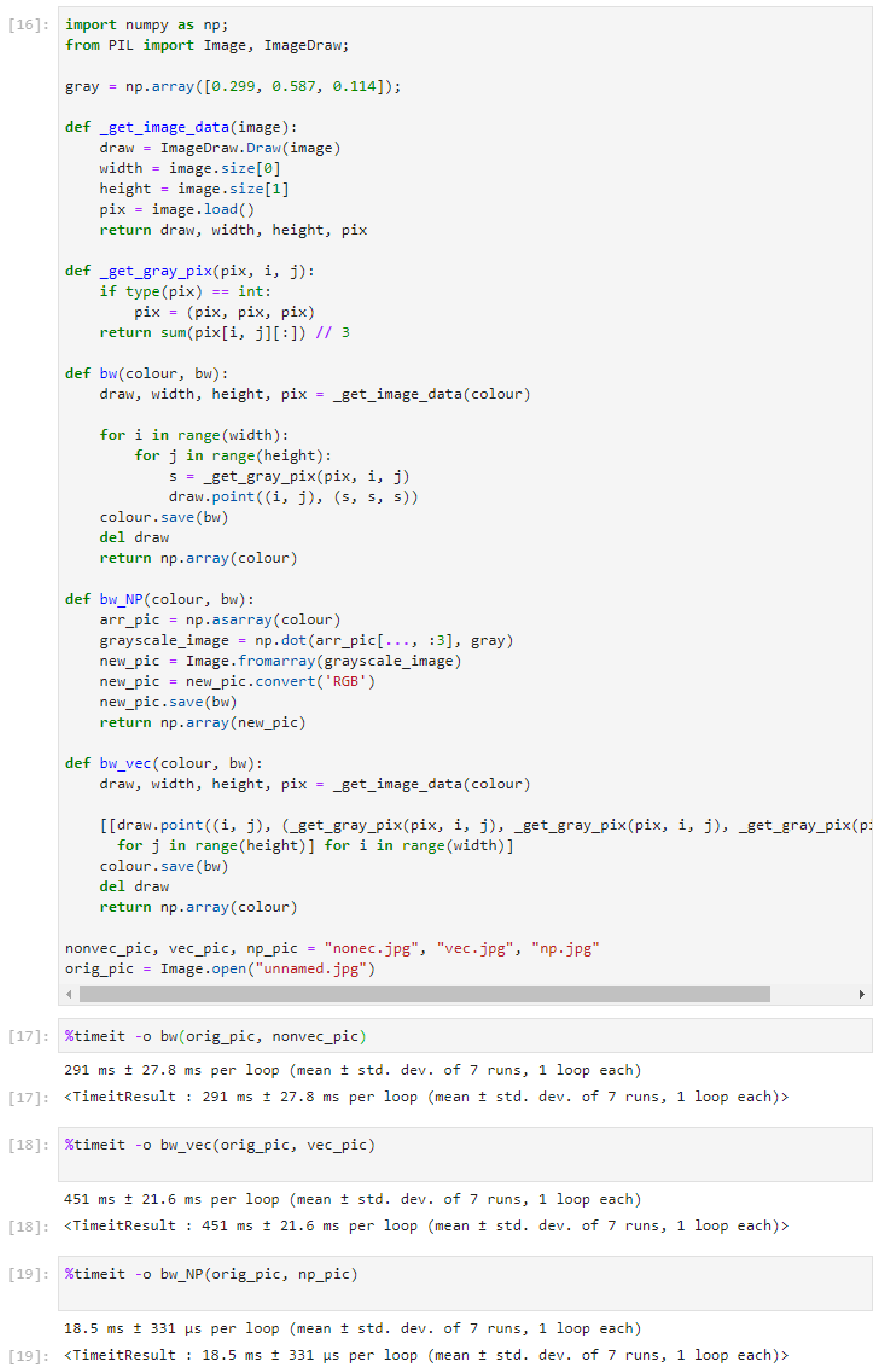


```
[10]: import numpy as np

      def maxBeforeZeroNP(arr):
          res = [arr[i] for i in range(1, len(arr)) if arr[i - 1] == 0]
          res.sort()
          return res[-1]

      def maxBeforeZeroVec(arr):
          return max([arr[i] for i in range(1, len(arr)) if arr[i - 1] == 0])

      def maxBeforeZero(arr):
          max = min(arr)
          for i in range(1, len(arr)):
              if arr[i - 1] == 0 and max < arr[i]:
                  max = arr[i]
          return max

      arr = np.array([int(x) for x in input().split()])

       7 4 5 0 0 6 2 0 9 11 12

[11]: %timeit -o maxBeforeZero(arr)

      6.53 µs ± 491 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

[11]: <TimeitResult : 6.53 µs ± 491 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[12]: %timeit -o maxBeforeZeroNP(arr)

      4.32 µs ± 89.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

[12]: <TimeitResult : 4.32 µs ± 89.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[13]: %timeit -o maxBeforeZeroVec(arr)

      4.26 µs ± 70.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

[13]: <TimeitResult : 4.26 µs ± 70.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>
```



```
5 7 8 0 77 59 78 0 3
77
77
77
```

Figure 4 – Task 4



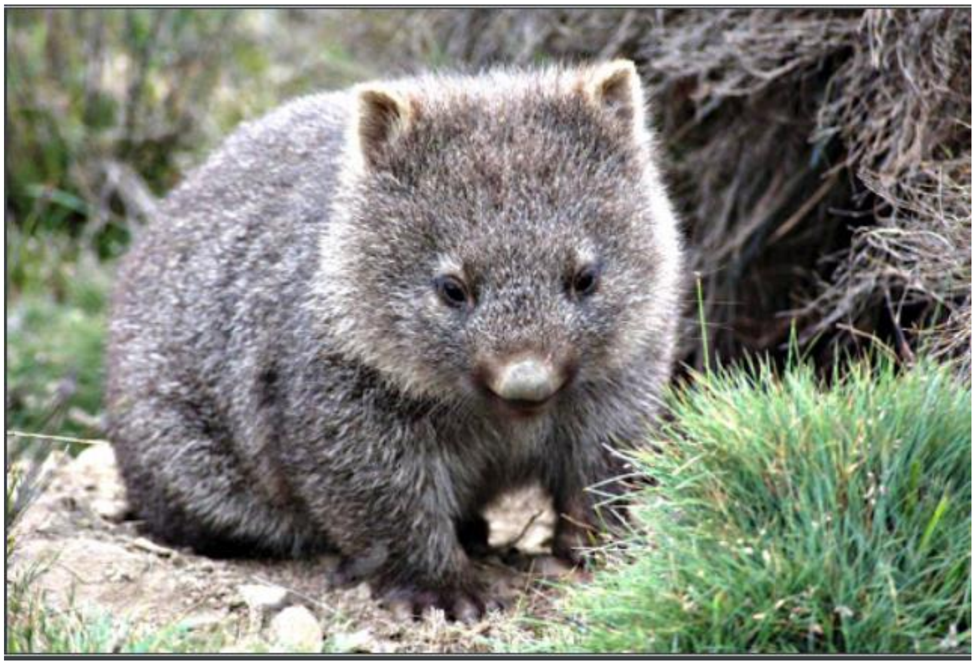
```
[16]: import numpy as np;
      from PIL import Image, ImageDraw;

      gray = np.array([0.299, 0.587, 0.114]);

      def _get_image_data(image):
          draw = ImageDraw.Draw(image)
          width = image.size[0]
          height = image.size[1]
          pix = image.load()
          return draw, width, height, pix

      def _get_gray_pix(pix, i, j):
          if type(pix) == int:
              pix = (pix, pix, pix)
          return sum(pix[i, j][:]) // 3

      def bw(colour, bw):
          draw, width, height, pix = _get_image_data(colour)

          for i in range(width):
              for j in range(height):
                  s = _get_gray_pix(pix, i, j)
                  draw.point((i, j), (s, s, s))
          colour.save(bw)
          del draw
          return np.array(colour)

      def bw_NP(colour, bw):
          arr_pic = np.asarray(colour)
          grayscale_image = np.dot(arr_pic[..., :3], gray)
          new_pic = Image.fromarray(grayscale_image)
          new_pic = new_pic.convert('RGB')
          new_pic.save(bw)
          return np.array(new_pic)

      def bw_vec(colour, bw):
          draw, width, height, pix = _get_image_data(colour)

          [[draw.point((i, j), (_get_gray_pix(pix, i, j), _get_gray_pix(pix, i, j), _get_gray_pix(p
            for j in range(height)] for i in range(width)]
          colour.save(bw)
          del draw
          return np.array(colour)

      nonvec_pic, vec_pic, np_pic = "nonec.jpg", "vec.jpg", "np.jpg"
      orig_pic = Image.open("unnamed.jpg")
```

```
[17]: %timeit -o bw(orig_pic, nonvec_pic)

      291 ms ± 27.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
[17]: <TimeitResult : 291 ms ± 27.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)>
```

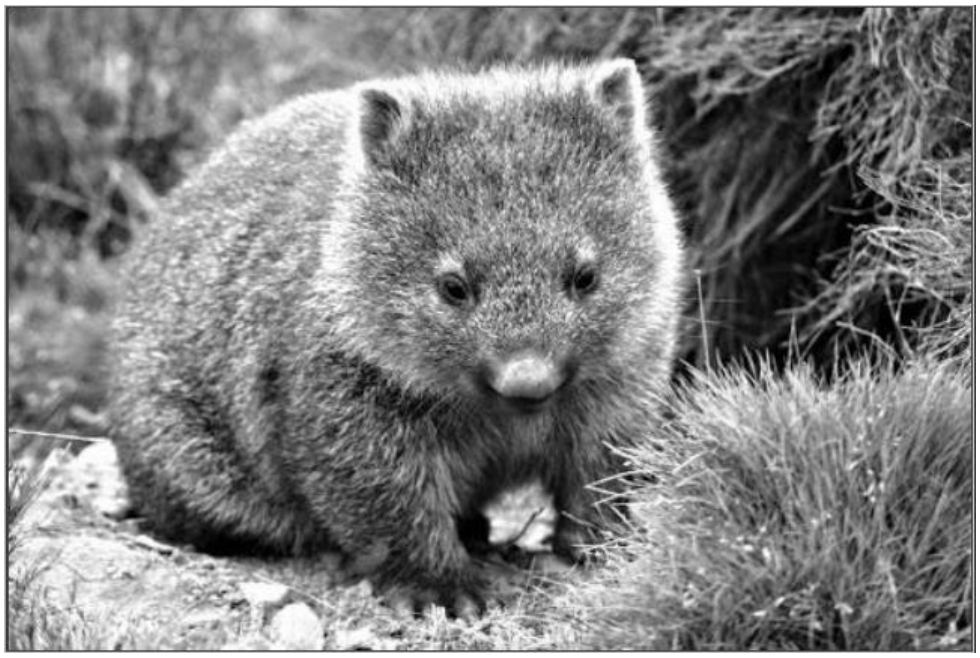
```
[18]: %timeit -o bw_vec(orig_pic, vec_pic)

      451 ms ± 21.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
[18]: <TimeitResult : 451 ms ± 21.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)>
```

```
[19]: %timeit -o bw_NP(orig_pic, np_pic)

      18.5 ms ± 331 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
[19]: <TimeitResult : 18.5 ms ± 331 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)>
```

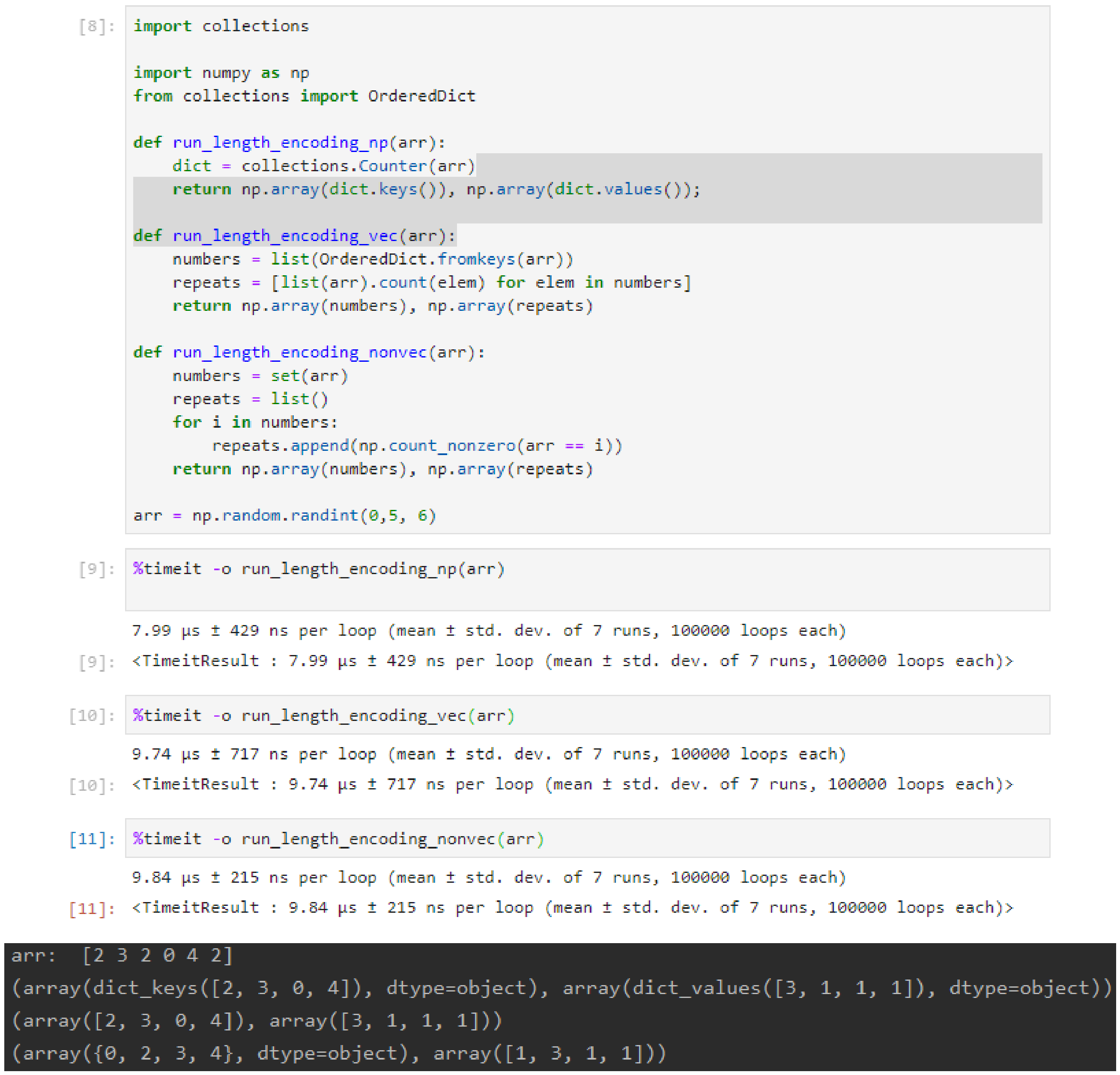
Figure 5 – Task 5



Before



After



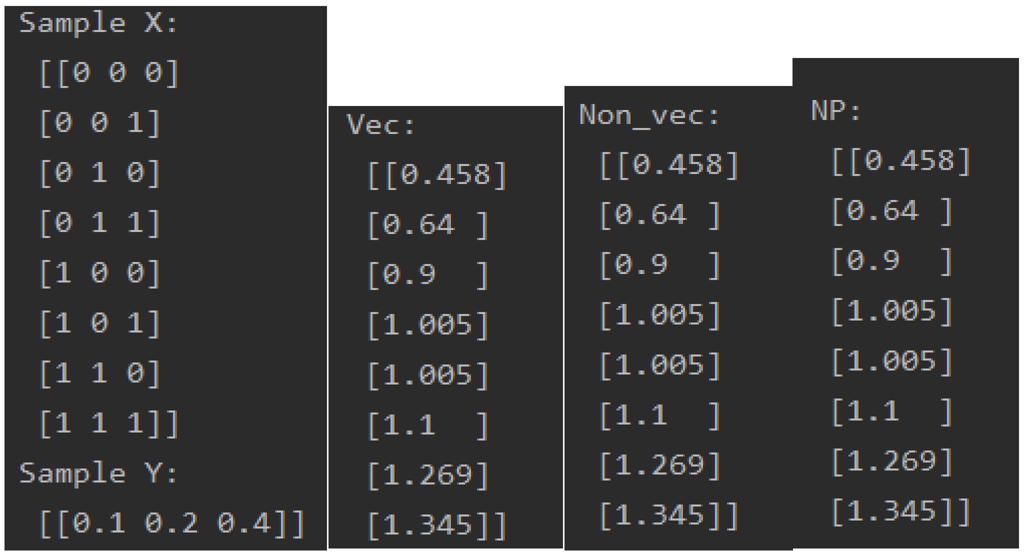
```
[8]: import collections

     import numpy as np
     from collections import OrderedDict

     def run_length_encoding_np(arr):
         dict = collections.Counter(arr)
         return np.array(dict.keys()), np.array(dict.values());

     def run_length_encoding_vec(arr):
         numbers = list(OrderedDict.fromkeys(arr))
         repeats = [list(arr).count(elem) for elem in numbers]
         return np.array(numbers), np.array(repeats)

     def run_length_encoding_nonvec(arr):
         numbers = set(arr)
         repeats = list()
         for i in numbers:
             repeats.append(np.count_nonzero(arr == i))
         return np.array(numbers), np.array(repeats)

     arr = np.random.randint(0,5, 6)
```

```
[9]: %timeit -o run_length_encoding_np(arr)

     7.99 µs ± 429 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[9]: <TimeitResult : 7.99 µs ± 429 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[10]: %timeit -o run_length_encoding_vec(arr)

     9.74 µs ± 717 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[10]: <TimeitResult : 9.74 µs ± 717 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>

[11]: %timeit -o run_length_encoding_nonvec(arr)

     9.84 µs ± 215 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
[11]: <TimeitResult : 9.84 µs ± 215 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)>
```

```
arr:  [2 3 2 0 4 2]
(array(dict_keys([2, 3, 0, 4]), dtype=object), array(dict_values([3, 1, 1, 1]), dtype=object))
(array([2, 3, 0, 4]), array([3, 1, 1, 1]))
(array({0, 2, 3, 4}, dtype=object), array([1, 3, 1, 1]))
```

Figure 6 – Task 6

```
[12]: import numpy as np
      from scipy.spatial import distance

      def euclidean_distance_nonvec(x, y):
          matrix = np.zeros((len(x), len(y)))
          for i in range(len(x)):
              for j in range(len(y)):
                  matrix[i, j] = np.linalg.norm(x[i] - y[j])
          return matrix

      def euclidean_distance_np(x, y):
          return distance.cdist(x, y, 'euclidean')

      def euclidean_distance_vec(x, y):
          matrix = np.array(
              [[np.linalg.norm(a_elem - b_elem)
                for b_elem in y] for a_elem in x])
          return matrix

      x = np.array([[0, 0, 0],
                    [0, 0, 1],
                    [0, 1, 0],
                    [0, 1, 1],
                    [1, 0, 0],
                    [1, 0, 1],
                    [1, 1, 0],
                    [1, 1, 1]])
      y = np.array([[0.1, 0.2, 0.4]])

[13]: %timeit -o euclidean_distance_vec(x, y).round(3)

      72.4 µs ± 2.27 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

[13]: <TimeitResult : 72.4 µs ± 2.27 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)>

[14]: %timeit -o euclidean_distance_nonvec(x, y).round(3)

      65.1 µs ± 3.33 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

[14]: <TimeitResult : 65.1 µs ± 3.33 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)>

[15]: %timeit -o euclidean_distance_np(x, y).round(3)

      19 µs ± 1.81 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)

[15]: <TimeitResult : 19 µs ± 1.81 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)>
```



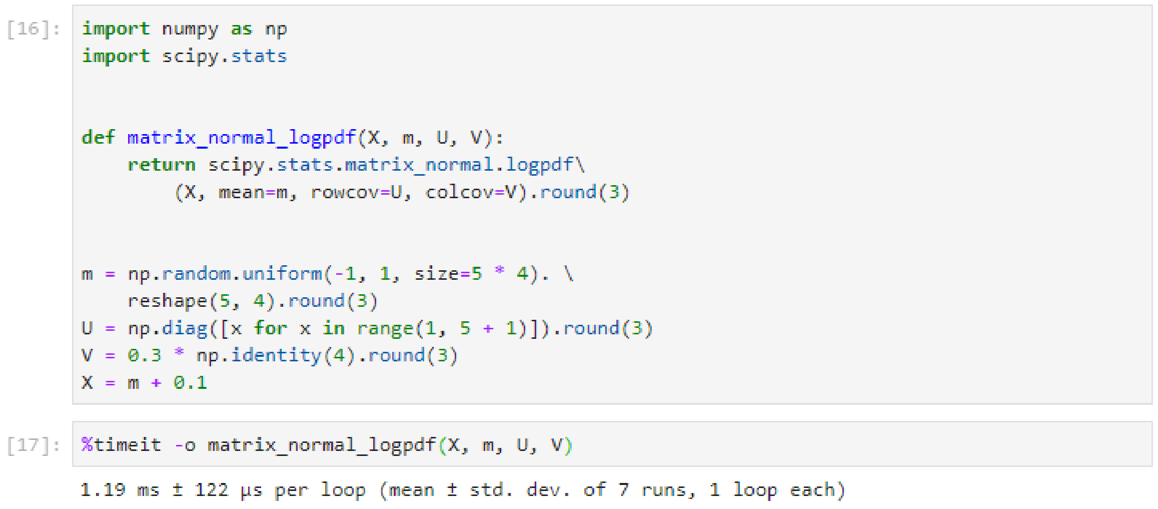
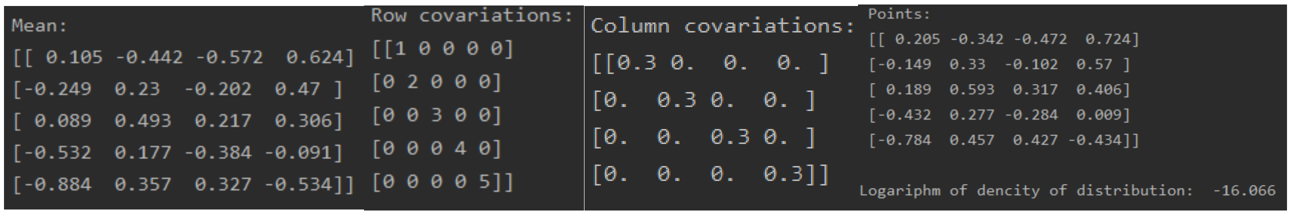
```
Sample X:
 [[0 0 0]
 [0 0 1]
 [0 1 0]
 [0 1 1]
 [1 0 0]
 [1 0 1]
 [1 1 0]
 [1 1 1]]
Sample Y:
 [[0.1 0.2 0.4]]
```

```
Vec:
 [[0.458]
 [0.64 ]
 [0.9  ]
 [1.005]
 [1.005]
 [1.1  ]
 [1.269]
 [1.345]]
```

```
Non_vec:
 [[0.458]
 [0.64 ]
 [0.9  ]
 [1.005]
 [1.005]
 [1.1  ]
 [1.269]
 [1.345]]
```

```
NP:
 [[0.458]
 [0.64 ]
 [0.9  ]
 [1.005]
 [1.005]
 [1.1  ]
 [1.269]
 [1.345]]
```

Figure 7 – Task 7


```
[16]: import numpy as np
      import scipy.stats


      def matrix_normal_logpdf(X, m, U, V):
          return scipy.stats.matrix_normal.logpdf\
              (X, mean=m, rowcov=U, colcov=V).round(3)


      m = np.random.uniform(-1, 1, size=5 * 4). \
          reshape(5, 4).round(3)
      U = np.diag([x for x in range(1, 5 + 1)]).round(3)
      V = 0.3 * np.identity(4).round(3)
      X = m + 0.1

[17]: %timeit -o matrix_normal_logpdf(X, m, U, V)

      1.19 ms ± 122 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
Mean:
[[ 0.105 -0.442 -0.572  0.624]
 [-0.249  0.23  -0.202  0.47 ]
 [ 0.089  0.493  0.217  0.306]
 [-0.532  0.177 -0.384 -0.091]
 [-0.884  0.357  0.327 -0.534]]

Row covariations:
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]

Column covariations:
[[0.3 0.  0.  0. ]
 [0.  0.3 0.  0. ]
 [0.  0.  0.3 0. ]
 [0.  0.  0.  0.3]]

Points:
[[ 0.205 -0.342 -0.472  0.724]
 [-0.149  0.33  -0.102  0.57 ]
 [ 0.189  0.593  0.317  0.406]
 [-0.432  0.277 -0.284  0.009]
 [-0.784  0.457  0.427 -0.434]]

Logariphm of dencity of distribution:  -16.066
```
Figure 8 – Task 8

Conclusions: during this laboratory training we have worked with different methods from NumPy package. During some tests we received interesting results. In most cases NumPy works faster than other implementations of the same task. This proves that NumPy is optimal thig to use in programming.