Working with NumPy

NumPy (Numerical Python, pronounced as "num-pie" or "num-pee")is a Python library that can be imported to perform basic numerical functions and matrix algebra. In principle, it provides support for large multi-dimensional arrays and mathematical functions that can act on these arrays.

Please read more at: https://numpy.org/doc/stable/user/index.html

Also here: https://www.w3schools.com/python/numpy/numpy_intro.asp

1. Arrays in NumPy -- Lists with more power

In [7]:
```
import numpy as np

a = [i for i in range(20)]  # creating a list in the usual sense
print (a)

b = np.array(a) # creates a numpy array of the list
b
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Out[7]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

In [9]:
```
a = [[1,2],[3,4]]
b = np.array(a)
```

In [14]:
```
# a[0,1]
b[0,1]
```

Out[14]:
```
2
```

In [2]:
```
print (a + a)  # creates a copy of the original list
b + b      # Performs vector or matrix addition
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 0, 1, 2, 3,
4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Out[2]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38])
```

In [3]:
```
print (a * 3)  # creates a copy of the original list 3 times
b * 3     # Element wise multiplication by an integer/scalar
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 0, 1, 2, 3,
4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 0, 1, 2, 3, 4, 5, 6, 7,
8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Out[3]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48,
       51, 54, 57])
```

In [4]:
```
b * 3.7     # Element wise multiplication by a float/scalar
```

Out[4]:
```
array([ 0. ,  3.7,  7.4, 11.1, 14.8, 18.5, 22.2, 25.9, 29.6, 33.3, 37. ,
       40.7, 44.4, 48.1, 51.8, 55.5, 59.2, 62.9, 66.6, 70.3])
```

In [8]:
```
print (type(a))
print (type(b))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

In [15]:
```python
a = np.array([1,2,3,'string'])
a * 3
```

Out[15]:
```
array([3, 6, 9])
```

In [16]:
```python
a = np.array([1,2,3,4])
a * 3
```

Out[16]:
```
array([ 3,  6,  9, 12])
```

How fast is np.array compared to list?

Let us square every element in a list or array.

In [17]:
```python
print ([i for i in range(10)])
print (np.arange(10))   # This is the array version of range() in lists
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0 1 2 3 4 5 6 7 8 9]
```

In [20]:
```python
%timeit -r1 [i**2 for i in range(1000)]
```

```
26.9 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 10,000 loops each)
```

In [21]:
```python
arr = np.arange(1000)
%timeit -r1 arr**2     # This squares every elemnt in the list and runs very fast
```

```
487 ns ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000,000 loops each)
```

Pre-defining an array for use in loops and matrix algebra

In [3]:
```python
σ_x = np.empty([2,2]) # All elements are empty with any random values.
σ_x[0,0] = 0
σ_x[0,1] = 1        # Type the elements individually or using a loop.
σ_x[1,0] = 1
σ_x[1,1] = 0

# print (σ_x)
σ_x
```

Out[3]:
```
array([[0., 1.],
       [1., 0.]])
```

In [4]:
```python
mat_0 = np.zeros([3,3])  # All elements are initialised as float 0.
mat_0
```

Out[4]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [5]:
```python
mat_1 = np.ones([2,2,2])  # Creates a 3D array (tensor), with all elements initiali
mat_1
```

Out[5]:
```
array([[[1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.]]])
```

In [6]:
```python
mat_identity = np.eye(4)  # Creates a 2D array with all diagonals as 1 (identity ma
mat_identity
```

Out[6]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [26]:
```python
Mat = np.empty([3,3])
for i in range(3):
    for j in range(3):
        Mat[i,j] = i + j

print (Mat)
```

```
[[0. 1. 2.]
 [1. 2. 3.]
 [2. 3. 4.]]
```

In [27]:
```python
Mat = np.empty([3,3])
i,j = 0,0

while i < 3:
    while j < 3:
        Mat[i,j] = i + j
        j += 1
    i += 1

print (Mat)
```

```
[[0. 1. 2.]
 [1. 2. 3.]
 [2. 3. 4.]]
```

NOTE 1: One can use np.ndarray to create an array. Here, all the elements are initialised with some random number. However, this is not recommended.

NOTE 2: The use of the **np.matrix** subclass is not recommended as it may get deprecated in the future. NumPy forums recommend only the use of **np.ndarray** class for all matrices.

Shape and dimension of an array

In [21]:
```python
mat_1.ndim # returns the dimension of the array
```

Out[21]:
```
3
```

In [22]:
```python
mat_1.shape # returns a tuple with the shape or all the dimensions of the array
```

Out[22]:
```
(2, 2, 2)
```

In [23]:
```python
num_list = np.arange(10) # creates a 1D array with shape 10
print (num_list)
num_list.shape
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Out[23]:
```
(10,)
```

In [24]:
```python
print(num_list.reshape(2,5)) # one can reshape it into a 2D array with shape (2,5)
num_list.reshape(2,5).shape
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
```

Out[24]:
```
(2, 5)
```

In [25]:
```python
print(num_list.reshape(4,2)) # cannot reshape it (4,2) as no. of elements must be s
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [25], in <cell line: 1>()
----> 1 print(num_list.reshape(4,2))

ValueError: cannot reshape array of size 10 into shape (4,2)
```

Accessing elements of an array

In [26]:
```
num_list = np.arange(16)
mat = num_list.reshape(4,4)
print (mat)  # Elements are arranged by rows and columns while reshaping
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

In [27]:
```
mat[0,2] # the containers for accessing element 2D arrays (row index, column index)
```

Out[27]:
```
2
```

In [28]:
```
tens = num_list.reshape(2,2,4)
print (tens)  # Elements are arranged by dim 1, dim 2,...,dim n while reshaping
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]]
```

In [29]:
```
tens[0,1,3]  # accesses 1st, 2nd, 4th element from the tensor
```

Out[29]:
```
7
```

In [30]:
```
tens[0,2,3]  # calls 1st, 3rd, 4th element from the tensor, but 3rd element does nc
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [30], in <cell line: 1>()
----> 1 tens[0,2,3]

IndexError: index 2 is out of bounds for axis 1 with size 2
```

Accessing slice of an array

In [78]:
```
num_list = np.arange(1,37)
print (num_list[:10]) # prints all element till 10
print (num_list[20:]) # prints all element after 20
print (num_list[10:20]) # prints all element after 10 and till 20
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36]
[11 12 13 14 15 16 17 18 19 20]
```

In [81]:
```
copy_array = num_list.reshape(4,3,3)   # so we now have 4, 3 x 3 matrices in the te
print (copy_array,'\n')

print (copy_array[1,:,:])  # Gives you the second 3 x 3 array
print ("***********",'\n')

print (copy_array[:,1,:])  # Gives you the cross 4 x 3 matrix
print ("***********",'\n')

print (copy_array[:,:,1],)  # Gives you the cross 4 x 3 matrix
print ("***********",'\n')

print (copy_array[0:2,1,:])  # Gives you the cross 2 x 3 matrix
print ("***********",'\n')

print (copy_array[:,1,0:2])  # Gives you the cross 4 x 2 matrix
```

```
[[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]

 [[10 11 12]
  [13 14 15]
  [16 17 18]]

 [[19 20 21]
  [22 23 24]
  [25 26 27]]

 [[28 29 30]
  [31 32 33]
  [34 35 36]]]

[[10 11 12]
 [13 14 15]
 [16 17 18]]
***********

[[ 4  5  6]
 [13 14 15]
 [22 23 24]
 [31 32 33]]
***********

[[ 2  5  8]
 [11 14 17]
 [20 23 26]
 [29 32 35]]
***********

[[ 4  5  6]
 [13 14 15]]
***********

[[ 4  5]
 [13 14]
 [22 23]
 [31 32]]
```

2. Important NumPy functions

- **np.linspace(start,end,num_elements)**

https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

Also see *np.logspace( )*

In [67]:
```
np.linspace(0,10,11) # By default end point is included unlike np.arange();

import numpy as np
x_space = np.linspace(0,10,50) # can be used to create fractional steps; such as ar
x_space
```

```
Out[67]: array([ 0.        ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
        1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
        2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
        3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
        4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
        5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
        6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
        7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
        8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
        9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

- **Trigonometric and exponential functions**

```
In [68]: np.sin(x_space[:15]) # sin(x) of the first 15 elements from the above linspace

print(np.cos(x_space[:15]),'\n')

print(np.exp(x_space[:15]),'\n')

print(np.log(x_space[1:15]),'\n')  # Avoiding log_e(0)

print(np.log2(x_space[1:15]),'\n')  # log_2(x) (base 2)

print(np.pi) # Defining pi
```

```
[ 1.          0.97924752  0.91785141  0.81835992  0.68490244  0.52301811
  0.33942593  0.1417459  -0.0618173  -0.26281476 -0.45290412 -0.6241957
 -0.76958007 -0.88302305 -0.9598162 ]

[ 1.          1.22639826  1.5040527   1.84456762  2.26217453  2.77432691
  3.40242971  4.17273388  5.11743359  6.27601167  7.69688981  9.4394523
 11.57652791 14.19743372 17.41170806]

[-1.58923521 -0.89608802 -0.49062292 -0.20294084  0.02020271  0.20252426
  0.35667494  0.49020634  0.60798937  0.71334989  0.80866007  0.89567144
  0.97571415  1.04982212]

[-2.29278175 -1.29278175 -0.70781925 -0.29278175  0.02914635  0.29218075
  0.51457317  0.70721825  0.87714325  1.02914635  1.16664987  1.29218075
  1.40765797  1.51457317]

3.141592653589793
```

```
In [ ]: import matplo
```

- **Transpose of a matrix**

```
In [59]: num_list = np.arange(16)
mat = num_list[2:14].reshape(4,3)  # The slice takes 12 elements from num_list and
print (mat,'\n')
print (np.transpose(mat))
```

```
[[ 2  3  4]
 [ 5  6  7]
 [ 8  9 10]
 [11 12 13]]

[[ 2  5  8 11]
 [ 3  6  9 12]
 [ 4  7 10 13]]
```

- **Flatten, ravel**

```
In [60]: print (mat) # Is a 4 x 3 complex matrix
print ()
print (mat.flatten())  # Flattens it to a 1D array
```

```
print ()
print (np.ravel(mat)) # Unravels it to a 1D array; same as flatten
```

```
[[ 2  3  4]
 [ 5  6  7]
 [ 8  9 10]
 [11 12 13]]

[ 2  3  4  5  6  7  8  9 10 11 12 13]

[ 2  3  4  5  6  7  8  9 10 11 12 13]
```

- **Trace, diagonal, conjugate and floor of a matrix**

In [66]:

```
print ("The trace is =", np.trace(mat),'\n') # Trace of a matrix

print ("The diagonal elements are =",np.diagonal(mat),'\n') # Returns a 1D array of

comp_mat = np.random.rand(3,3) + 1j*np.random.rand(3,3)  # A random complex matrix
print (comp_mat,'\n')
print ("The conjugate of the matrix is =",np.conjugate(comp_mat))  # Conjugate of a

comp_mat = np.random.randint(5,size=[3,3]) + np.random.rand(3,3)
print (comp_mat,'\n')
print (np.floor(comp_mat))  # The floor of the scalar x is the largest integer i, s
```

```
The trace is = 18

The diagonal elements are = [ 2  6 10]

[[0.87434361+0.20231365j 0.36105939+0.84608249j 0.74445461+0.502479j  ]
 [0.0446541 +0.4684847j  0.91404551+0.48877103j 0.37580247+0.11571821j]
 [0.66915608+0.29394159j 0.28176568+0.15591016j 0.81578169+0.45494658j]]

The conjugate of the matrix is = [[0.87434361-0.20231365j 0.36105939-0.84608249j 0.
74445461-0.502479j  ]
 [0.0446541 -0.4684847j  0.91404551-0.48877103j 0.37580247-0.11571821j]
 [0.66915608-0.29394159j 0.28176568-0.15591016j 0.81578169-0.45494658j]]
[[4.73084281 4.81037414 0.20540688]
 [2.878465   0.94704404 0.58708674]
 [1.23923798 2.68568617 1.68899478]]

[[4. 4. 0.]
 [2. 0. 0.]
 [1. 2. 1.]]
```

3. Using random numbers

Working with random integers

In [31]:

```
A = np.random.randint(5,size=(3, 4)) # Random array with shape (3,4) filled with in
print ('Matrix A is:\n', A)
```

```
Matrix A is:
 [[4 3 0 1]
 [4 3 4 2]
 [4 0 2 4]]
```

In [32]:

```
A = np.random.randint(1,[3,5,10]) # Random array with shape (1,3)
                                  # Integers with lower bound 1 and upperbound [3,5
print ('Matrix A is:\n', A)
```

```
Matrix A is:
 [1 4 1]
```

In [33]:

```
A = np.random.randint([1, 3, 5, 7], [[10], [20]]) # Random array with shape (2,4) w
```

```
print ('Matrix A is:\n', A)
```

```
Matrix A is:
 [[ 1  7  8  8]
 [17 13 14 12]]
```

Working with general random numbers (floats)

In [34]:
```
A = np.random.rand(3,2)  # Random array with shape (3,2) with uniform random number
print ('Matrix A is:\n', A)
```

```
Matrix A is:
 [[0.960704   0.31006085]
 [0.00543051 0.33408133]
 [0.26585241 0.53001972]]
```

In [35]:
```
A = np.random.randn(3,3)  # Random array of shape (3,3) from a Gaussian distributic
print ('Matrix A is:\n', A)
```

```
Matrix A is:
 [[-0.25045079 -2.08384439 -1.07010669]
 [-0.51716863  0.69883823 -1.46131635]
 [ 0.47136285 -1.33294674  1.46969341]]
```

In [97]:
```
Z = np.random.uniform(-1,1,size=(3,3)) # Random array of shape (3,3) from a uniform
print ('Matrix A is:\n', Z)
```

```
Matrix A is:
 [[-0.60287859 -0.18653867  0.04529753]
 [-0.11012506  0.55734506  0.70075088]
 [-0.85272057  0.3693696  -0.4454031 ]]
```

4. Arithmetic on arrays and matrices

Different ways to add and multiplying

In [36]:
```
A = np.random.randint(5,size=(3, 4)) # Random array with shape (3,4) filled with ir
B = np.random.randint(5,size=(3, 4))

print ('Matrix A is:\n', A)
print ()
print ('Matrix B is:\n', B)
print ()
print ('The sum is:\n', A + B)   # Both A and B must have same shape
```

```
Matrix A is:
 [[0 4 3 0]
 [3 4 3 4]
 [3 0 4 4]]

Matrix B is:
 [[3 1 1 2]
 [1 0 1 4]
 [4 3 4 0]]

The sum is:
 [[3 5 4 2]
 [4 4 4 8]
 [7 3 8 4]]
```

In [37]:
```
A = np.random.randint(5,size=(3, 4)) # Random array with shape (3,4) filled with ir
B = np.random.randint(5,size=(3, 4))

print ('Element-wise multiplication:\n', A * B)  # Element wise multiplication
print ()
print ('Element-wise exponentiation:\n',A ** 2) # Element wise exponentiation
```

```
Element-wise multiplication:
 [[ 0  6  0  2]
 [12  3  2  4]
 [ 2 12  0  4]]

Element-wise exponentiation:
 [[ 0  4  0  1]
 [ 9  1  1  1]
 [ 1 16  0  4]]
```

In [38]:

```
A = np.random.randint(5,size=(3, 4)) # Random array with shape (3,4) filled with in
B = np.random.randint(5,size=(3, 4))

print ('Matrix multiplication:\n') #
print ()
A @ B  # Matrix multiplication; does not work due to mismatch of shape between row
```

```
Matrix multiplication:
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [38], in <cell line: 6>()
      4 print ('Matrix multiplication:\n') #
      5 print ()
----> 6 A @ B

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gu
func signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 4)
```

In [39]:

```
A = np.random.randint(2,size=(4, 4))
B = np.random.randint(2,size=(4, 4))
print ('Matrix A is:\n', A)
print ()
print ('Matrix B is:\n', B)
print ()
print ('Matrix multiplication:\n', A @ B)  # Matrix multiplication
```

```
Matrix A is:
 [[0 1 0 1]
 [0 1 0 0]
 [0 1 0 0]
 [1 0 1 1]]

Matrix B is:
 [[1 1 1 1]
 [1 0 0 0]
 [1 1 1 1]
 [0 1 1 1]]

Matrix multiplication:
 [[1 1 1 1]
 [1 0 0 0]
 [1 0 0 0]
 [2 3 3 3]]
```

In [40]:

```
A = np.random.randint(2,size=(4, 4))
B = np.random.randint(2,size=(4, 1))
print ('Matrix A is:\n', A)
print ()
print ('Matrix B is:\n', B)
print ()
print ('Matrix multiplication:\n', A @ B)  # Matrix multiplication
```

```
Matrix A is:
 [[0 0 0 1]
 [0 1 1 1]
 [1 1 0 0]
 [1 0 0 0]]

Matrix B is:
 [[0]
 [0]
 [1]
 [0]]

Matrix multiplication:
 [[0]
 [1]
 [0]
 [0]]
```

In [41]:
```
A = np.random.randint(3,size=(4))
B = np.random.randint(3,size=(4))
print ('Matrix A is:\n', A)
print ()
print ('Matrix B is:\n', B)
print ()
print ('Inner product:\n', np.dot(A,B))  # Inner product of two 1D arrays.
                                          # For 2D this gives matrix multiplication,
```

```
Matrix A is:
 [0 0 2 0]

Matrix B is:
 [2 0 1 2]

Inner product:
 2
```

In [17]:
```
a = np.array([1+2j,3+4j])
b = np.array([5+6j,7+8j])
print (np.vdot(a,a))
print (np.vdot(b,b))
print (np.vdot(a,b))
```

```
(30+0j)
(174+0j)
(70-8j)
```

For more options see https://numpy.org/doc/stable/reference/generated/numpy.dot.html

5. Tasks for today

Solve the following problems.

1. Consider the code for the Fibonacci series done in the last class using **recursive functions** and the **for...in** loop.

   Check the time taken in both the methods by importing the **time** module (see below). See, how this changes if N in the series is increased from 10 to 100, in steps of 10.

2. Create two random 2D arrays/matrices A and B of size 3×6, that take random real numbers between [0,10). Reshape them to 1D vectors and add them. Now reshape them back to 3×6 and compare the sum with the direct sum A + B.

3. Consider the set of equations:

$$15x + 3y + 5z = x'$$

$$-5x + 2z = y'$$

$$8x + 13y + 5z = z'$$

If the above equation can be written as $A.\mathbf{x} = \mathbf{x}'$, write down A.

Define a function, **def sol(x)**, that gives the output $\mathbf{x}'$ for any input $\mathbf{x}$. Solve for $\mathbf{x} = [1, 0, 0]$.

4. Using NumPy, show that Pauli matrices are indeed unitary and Hermitian. Do not use in-built check functions. Also, check if sum of two Pauli matrices is unitary and Hermitian.

5. Check the **np.kron(A,B)** function. This creates a tensor (Kronecker) product of two matrices.

   https://en.wikipedia.org/wiki/Kronecker_product

   Use NumPy matrix calculations to verify the **np.kron(A,B)** command.

6. i. Create a random complex 3×3 matrix A, using **NumPy**. Both real and imaginary numbers range from (-1 to 1).

   ii. Write a function that checks whether A is Hermitian or not. (Do not use in-built function). *Hint: You can use* 10^{-10} *as 0 iff needed.*

   iii. If A is not Hermitian, create a Hermitian matrix B by only performing basic arithmetic operations on A.

   iv. Check if B is Hermitian.

7. Write an array that takes 100 values between 0 to 4π. *Hint: Use* **np.pi** *for* π.

   Use numpy to calculate the sine and cosine of the above values.

   Plot the sine and cosine using matplotlib.

8. Write a code to create an array for a tensor $\mathcal{M}$, with elements $m_{ijk} = i + j + k$, where $i, j, k \in [0, 1, 2]$, using a **for...in** or **while** loop. What is the dimension and shape of $\mathcal{M}$?

   Find the matrix which corresponds to a) $i = 1$, b) $j = 1$, and c) $k = 1$.

EXTRA NOTES -- How fast can you run?

Here we consider tests for how fast can a block of code run in Python. This allows you to check which of the several loops you can implement will give you a faster output.

This allows you to optimize your code at a very elementary level.

timeit command

In [1]:
```
%timeit [i for i in range(10000)] #-rx runs the timer x times, and -ny means run cc
                                  # If you do not specify, then r is 7 and y
```

```
158 µs ± 2.85 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Let us compare **while** with **for...in** loops

In [2]:
```
%%timeit -r1 -n100 # -rx and -ny mean same as before. Double %% means we check time

i = 0
```

```
while i < 10000:
    i += 1
```

297 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)

In [3]:
```
%%timeit -r1 -n100 # Again, -rx mean run x times, and -ny means y loops for each r.
                    # If you do not specify, then r is 7 and y is 1000000 by defaul
k = 0
for i in range(10000):
    k += 1
```

636 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)

Let us compare a recursive function with a loop

In [4]:
```
def factorial_rec(n): # A recursive funstion for factorial
    if n == 1:
        return 1
    else:
        return n*factorial_rec(n-1)

def factorial_loop(n): # A loop function for factorial
    prod = 1
    for i in range(n):
        prod *= i+1
    return prod
```

In [5]:
```
%%timeit
factorial_loop(5)
```

253 ns ± 10.5 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)

In [6]:
```
%%timeit
factorial_rec(5)
```

288 ns ± 8.84 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)

Importing the time module

Simple loops are usually faster than recursive functions

In [7]:
```
import time

start = time.time() # gives the wall-time (total time taken by I/O and processor)
# start = time.process_time() # gives the process-time (total time taken by the pro

# Run the main block inside start and end

factorial_loop(1500)

end = time.time()
# end = time.process_time()
print ("Time taken to run the block in seconds (default):",end-start)

start = time.time()
# start = time.process_time()
factorial_rec(1500)

end = time.time()
# end = time.process_time()

print ("Time taken to run the block in seconds (default):",end-start)
```

Time taken to run the block in seconds (default): 0.00033974647521972656
Time taken to run the block in seconds (default): 0.0006968975067138672

EXTRA NOTES -- Be careful while copying in Python?

Copying an array - the notion of assignment, shallow copy and deep copy

In [13]:
```
simple_list = list([1,2,10,34,'man','cat',[9,1,2],['women','baby','dog']])
print (simple_list)

dup_list = simple_list # This is an assignment '='
dup_list[4] = 'human' # Let us change the 4th element in the duplicate copy

print (dup_list)
print (simple_list,'\n')  # WOW!! the 4th element in the original has also changed
```

```
[1, 2, 10, 34, 'man', 'cat', [9, 1, 2], ['women', 'baby', 'dog']]
[1, 2, 10, 34, 'human', 'cat', [9, 1, 2], ['women', 'baby', 'dog']]
[1, 2, 10, 34, 'human', 'cat', [9, 1, 2], ['women', 'baby', 'dog']]
```

In [14]:
```
print (id(dup_list))   # id gives you an integer associated with an object -- typic
print (id(simple_list)) # both lists have the same id if you are doing an assignmen
```

```
4436593920
4436593920
```

In [20]:
```
sliced_list = simple_list[2:] # Creating a slice of the original list
sliced_list[2] = 'second man'

print (sliced_list,'\n')
print (simple_list,'\n')

print (id(slice_list),'\n')  # So the slice now is a different list in some sense,
                             # THIS IS NO LONGER TRUE WHILE WORKING WITH NUMPY ARRAY

sliced_list[5][1] = 'teen'

print (sliced_list)
print (dup_list)
print (simple_list,'\n')  # So changes in first index is independent -- but second
                          # This is called a shallow copy
```

```
[10, 34, 'second man', 'cat', [9, 1, 2], ['women', 'teen', 'dog']]

[1, 2, 10, 34, 'man', 'cat', [9, 1, 2], ['women', 'teen', 'dog']]

4438223168

[10, 34, 'second man', 'cat', [9, 1, 2], ['women', 'teen', 'dog']]
[1, 2, 10, 34, 'human', 'cat', [9, 1, 2], ['women', 'teen', 'dog']]
[1, 2, 10, 34, 'man', 'cat', [9, 1, 2], ['women', 'teen', 'dog']]
```

Using the copy module for deep copy

In [24]:
```
import copy as cp

simple_list = list([1,2,10,34,'man','cat',[9,1,2],['women','baby','dog']])  # Let u

dup_list = cp.copy(simple_list) # This slice will again create a shallow copy
dup_list_deep = cp.deepcopy(simple_list) # This slice will create a deep copy

dup_list[2] = 102
dup_list[6][1] = 'teen'
dup_list_deep[6][2] = 'cat'

print (simple_list)
print (dup_list)
print (dup_list_deep) # So a deep copy creates an independent copy at all levels of
```

```
[1, 2, 10, 34, 'man', 'cat', [9, 'teen', 2], ['women', 'baby', 'dog']]
[1, 2, 102, 34, 'man', 'cat', [9, 'teen', 2], ['women', 'baby', 'dog']]
[1, 2, 10, 34, 'man', 'cat', [9, 1, 'cat'], ['women', 'baby', 'dog']]
```

In [25]:
```
simple_list = list([1,2,10,34,'man','cat',[9,1,2],['women','baby','dog']])  # Let u

slice_copy = cp.copy(simple_list[2:]) # This slice will again create a shallow copy
slice_deepcopy = cp.deepcopy(simple_list[2:]) # This slice will create a deep copy

slice_copy[2] = 102
slice_copy[5][1] = 'teen'
slice_deepcopy[5][1] = 'adult'

print (simple_list)
print (slice_copy)
print (slice_deepcopy) # So a deep copy creates an independent copy at all levels c
```

```
[1, 2, 10, 34, 'man', 'cat', [9, 1, 2], ['women', 'teen', 'dog']]
[10, 34, 102, 'cat', [9, 1, 2], ['women', 'teen', 'dog']]
[10, 34, 'man', 'cat', [9, 1, 2], ['women', 'adult', 'dog']]
```

The same thing holds for NumPy arrays, albeit with some subtle differences

One can use the **np.copy** function

In [19]:
```
import numpy as np

simple_array = np.array([1,2,10,34,'man','cat',[9,1,2],['women','baby','dog']],dtyp
                # create an array from the list; we use dtype = object
                # this ensures different objects are listed in an array including a

print (simple_array)

slice_array = simple_array[3:]

print (slice_array,'\n')

slice_array[1] = 'human'
print (slice_array)
print (simple_array,'\n')  # IMPORTANT --- slice here is an assignment and not a sh

slice_array_copy = cp.copy(simple_array[3:])  # You can also use np.copy (in NumPy)
slice_array_copy[1] = 'monkey'
print (slice_array_copy)
print (simple_array,'\n') # IMPORTANT --- slice here is an assignment and not a sha

slice_array_copy[4][0] = 'queen'
print (slice_array_copy)
print (simple_array,'\n')  # IMPORTANT --- this is indeed a shallow copy


slice_array_deepcopy = cp.deepcopy(simple_array[3:])  # Now we have a deep copy
slice_array_deepcopy[4][1] = 'rascal'
print (slice_array_deepcopy)
print (simple_array,'\n')

slice_array_deepcopy = np.copy(simple_array[3:])  # np.copy is a shallow copy, but
slice_array_deepcopy[4][1] = 'naughty'
print (slice_array_deepcopy)
print (simple_array,'\n')
```

```
[1 2 10 34 'man' 'cat' list([9, 1, 2]) list(['women', 'baby', 'dog'])]
[34 'man' 'cat' list([9, 1, 2]) list(['women', 'baby', 'dog'])]

[34 'human' 'cat' list([9, 1, 2]) list(['women', 'baby', 'dog'])]
[1 2 10 34 'human' 'cat' list([9, 1, 2]) list(['women', 'baby', 'dog'])]

[34 'monkey' 'cat' list([9, 1, 2]) list(['women', 'baby', 'dog'])]
[1 2 10 34 'human' 'cat' list([9, 1, 2]) list(['women', 'baby', 'dog'])]

[34 'monkey' 'cat' list([9, 1, 2]) list(['queen', 'baby', 'dog'])]
[1 2 10 34 'human' 'cat' list([9, 1, 2]) list(['queen', 'baby', 'dog'])]

[34 'human' 'cat' list([9, 1, 2]) list(['queen', 'rascal', 'dog'])]
[1 2 10 34 'human' 'cat' list([9, 1, 2]) list(['queen', 'baby', 'dog'])]

[34 'human' 'cat' list([9, 1, 2]) list(['queen', 'naughty', 'dog'])]
[1 2 10 34 'human' 'cat' list([9, 1, 2]) list(['queen', 'naughty', 'dog'])]
```

In [52]:

```
num_list = np.arange(1,37)
dup_array = num_list.reshape(4,3,3) # so we now have an assignment 4, 3 x 3 matrice
copy_array = np.copy(num_list.reshape(4,3,3)) # so we now have a copy of 4, 3 x 3 m
                                              # for a N-dim array, a shallow copy

dup_array[0,0,0] = 1002
copy_array[0,0,1] = 2000

print (dup_array,'\n')
print (copy_array,'\n')
print (num_list) # Original list is changed by assignment but not by copy
```

```
[[[1002    2    3]
  [   4    5    6]
  [   7    8    9]]

 [[  10   11   12]
  [  13   14   15]
  [  16   17   18]]

 [[  19   20   21]
  [  22   23   24]
  [  25   26   27]]

 [[  28   29   30]
  [  31   32   33]
  [  34   35   36]]]

[[[   1 2000    3]
  [   4    5    6]
  [   7    8    9]]

 [[  10   11   12]
  [  13   14   15]
  [  16   17   18]]

 [[  19   20   21]
  [  22   23   24]
  [  25   26   27]]

 [[  28   29   30]
  [  31   32   33]
  [  34   35   36]]]

[1002    2    3    4    5    6    7    8    9   10   11   12   13   14
   15   16   17   18   19   20   21   22   23   24   25   26   27   28
   29   30   31   32   33   34   35   36]
```

In [55]:

```
print (id(copy_array))
print (id(dup_array))
print (id(num_list)) # IDs are all different here due to reshape
```

```
4544507088
4544506320
4487899984
```

In [ ]: