Scientific programming

In [1]:
```
import numpy as np
```

Generalized Fibonacci numbers

In [2]:
```
def fibonacci(nterm: int, order=2):
    """Calculates the terms of a generalized Fibonacci series.

    Args:
        nterm: Number of terms to calculate.
        order: Number of previous elements to sum up in order to get the
            Optional, default: 2

    Returns:
        Generalized fibonacci terms as array
    """
    fib = np.empty((nterm,), dtype=int)
    fib[0 : min(order, nterm)] = 1
    for iterm in range(order, nterm):
        fib[iterm] = np.sum(fib[iterm - order : iterm])
    return fib
```

In [3]:
```
fibonacci(10)
```

Out[3]:
```
array([ 1,  1,  2,  3,  5,  8, 13, 21, 34, 55])
```

In [4]:
```
fibonacci(10, 3)
```

Out[4]:
```
array([  1,   1,   1,   3,   5,   9,  17,  31,  57, 105])
```

In [5]:
```
fibonacci(10, order=3)
```

Out[5]:
```
array([  1,   1,   1,   3,   5,   9,  17,  31,  57, 105])
```

In [6]:
```
args = [10, 3]
fibonacci(*args)
```

Out[6]:
```
array([  1,   1,   1,   3,   5,   9,  17,  31,  57, 105])
```

In [ ]:
```

```

Optimizied generalized Fibonacci numbers

In [7]:
```
def fibonacci_optimized(nterm: int, order=2):
    """Calculates the terms of a generalized Fibonacci series with optimi

    Args:
        nterm: Number of terms to calculate.
        order: Number of previous elements to sum up in order to get the
            Optional, default: 2

    Returns:
        Generalized fibonacci series terms as array.
    """
    fib = np.empty((nterm,), dtype=int)
    fib[0 : min(order, nterm)] = 1
    if nterm <= order:
        return fib
    fib[order] = order
    for iterm in range(order + 1, nterm):
        fib[iterm] = 2 * fib[iterm - 1] - fib[iterm - order - 1]
    return fib
```

In [8]:
```
fibonacci_optimized(10, 3)
```

Out[8]:
```
array([  1,   1,   1,   3,   5,   9,  17,  31,  57, 105])
```

Matrix product

In [9]:
```
def matrix_product(mtx1, mtx2):
    """Calculates the product of two matrices

    Note: Only for exercise purposes, use the @-operator or np.dot() in p

    Args:
        mtx1: (M, N) shaped real array
        mtx2: (N, M') shaped real array

    Returns:
        Matrix product as (M, M') shaped array.
    """
    m1, n1 = mtx1.shape
    m2, n2 = mtx2.shape
    result = np.empty((m1, n2), dtype=float)
    for ii in range(m1):
        for jj in range(n2):
            result[ii, jj] = np.sum(mtx1[ii,:] * mtx2[:,jj])
    return result
```

In [10]:
```
m1 = np.array([[1, 2, 3], [4, 5, 6]])
m2 = np.array([[1, 2], [3, 4], [5, 6]])
matrix_product(m1, m2)
```

Out[10]:
```
array([[22., 28.],
       [49., 64.]])
```

In [11]:
```
m1 @ m2
```

Out[11]:
```
array([[22, 28],
       [49, 64]])
```

Matrix determinant

In [12]:
```
def determinant(mtx):
    """Calculates the deteminant of a square matrix.

    Note: Only for exercise purposes, use np.linalg.det() in production

    Args:
        mtx: Square numpy array

    Returns:
        Determinant of the matrix.
    """
    nn = mtx.shape[0]
    if nn == 1:
        return mtx[0, 0]
    det = 0.0
    submtx = np.empty(((nn - 1), (nn - 1)), dtype=float)
    for ii in range(nn):
        submtx[:, 0:ii] = mtx[1:, 0:ii]
        submtx[:, ii:] = mtx[1:, ii + 1:]
        det += (-1)**ii * mtx[0, ii] * determinant(submtx)
    return det
```

In [13]:
```
mtx = np.array([[1.0, 2.0], [3.0, 4.0]])
```

In [14]:
```
determinant(mtx)
```

Out[14]: -2.0

In [15]:
```
np.linalg.det(mtx)
```

Out[15]: -2.0000000000000004

In [16]:
```
mtx2 = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
```

In [17]:
```
determinant(mtx2)
```

Out[17]: 0.0

In [18]:
```
np.linalg.det(mtx2)
```

Out[18]: 0.0

In [19]:
```
mtx3 = np.array([[1.0, 2.0, 9.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]])
```

In [20]:
```
determinant(mtx3)
```

Out[20]: -18.0

In [21]:
```
np.linalg.det(mtx3)
```

Out[21]: -18.000000000000014

In [ ]: