Example 142. Python Let us implement Euler's method to redo and extend Example 141.

```
>>> def euler(f, x0, y0, xmax, n):
        h = (xmax - x0) / n
        ypoints = [y0]
        for i in range(n):
            y0 = y0 + f(x0,y0)*h
            x0 = x0 + h
            ypoints.append(y0)
        return ypoints

>>> def f(x, y):
        return (2*x-3*y)**2 + 2/3
```

If we choose the number of steps `n` to be 3 and `xmax` to be 2 (because we want $x_n = 2$), then the following matches exactly our computation in Example 141 (with 3 steps):

```
>>> euler(f, 1, 1/3, 2, 3)

    [0.3333333333333333, 0.8888888888888888, 1.1111111111111112, 1.3333333333333335]
```

As expected, increasing the number of steps provides better approximations to the exact solution $y(x) = \frac{2}{3}x - \frac{1}{3(3x-2)}$ with $y(2) = \frac{5}{4} = 1.25$.

```
>>> euler(f, 1, 1/3, 2, 10)

    [0.3333333333333333, 0.5, 0.6156666666666667, 0.7129142333333334, 0.8008567296803291,
    0.8833184418108875, 0.9622382358968495, 1.0387196700012582, 1.1134429074281287,
    1.1868524913418415, 1.259252430333012]

>>> euler(f, 1, 1/3, 2, 100)[-1]

    1.2508710385019504
```

If `ypoints` is a list, then its elements can be accessed as `ypoints[0]`, `ypoints[1]`, ... Moreover, we can access the last element as `ypoints[-1]`. For instance, above, we used `euler(f, 1, 1/3, 2, 100)[-1]` to get the last element of the 101 approximations $y_0, y_1, ..., y_{100}$. That last element is the approximation of $y(2) = 1.25$.

The following convincingly illustrates that the error in Euler's method is $O(h)$.

```
>>> [euler(f, 1, 1/3, 2, 10**n)[-1] - 1.25 for n in range(7)]

    [0.7499999999999998, 0.00925243033301193, 0.0008710385019503608, 8.668868812233832e-05,
    8.664791898871371e-06, 8.664409443248644e-07, 8.66125893228542e-08]
```

However, note that our computer had to work pretty hard to get the final approximation, because that entailed computing 10^6 values. We clearly need a higher order method in order to compute to higher accuracy.