Part 2 — Workshop 1

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
NHH Norwegian School of Economics

September 26, 2025

Exercise 1: Summing lists and arrays

In this exercise, we investigate another difference between built-in lists and NumPy arrays: performance. We do this by comparing the execution time of different implementations of the `sum()` function.

1. Create a list `lst` and a NumPy array `arr`, each of them containing the sequence of ten values `0, 1, 2, ..., 9`.

   Hint: You can use the list constructor `list()` and combine it with the `range()` function which returns an objecting representing a range of integers.

   Hint: You should create the NumPy array using `np.arange()`.

2. We want to compute the sum of integers contained in `lst` and `arr`. Use the built-in function `sum()` to sum elements of a list. For the NumPy array, use the NumPy function `np.sum()`.

3. You are interested in benchmarking which summing function is faster. Repeat the steps from above, but use the cell magic `%timeit` to time the execution of a statement as follows:

   ```
   %timeit statement
   ```

4. Recreate the list and array to contain 100 integers starting from 0, and rerun the benchmark.

5. Recreate the list and array to contain 10,000 integers starting from 0, and rerun the benchmark.

What do you conclude about the relative performance of built-in lists vs. NumPy arrays?

Solution.

Part 1: Creating a list and a NumPy array

[1]:
```python
# create list with 10 elements 0,1,...,9
lst = list(range(10))
```

[2]:
```python
import numpy as np
# create array with 10 elements 0,1,...,9
arr = np.arange(10)
```

Part 2: Computing the sum of elements

[3]:
```python
# sum the list using the built-in function sum()
sum(lst)
```

```
[3]: 45
```

```
[4]: # sum the NumPy array using NumPy's sum() function
     np.sum(arr)
```

```
[4]: np.int64(45)
```

Part 3: Benchmarking with 10 values

```
[5]: # benchmark summing list using built-in sum()
     %timeit sum(lst)
```

```
59.4 ns ± 0.196 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

```
[6]: # Benchmark summing array using NumPy's sum()
     %timeit np.sum(arr)
```

```
1.64 μs ± 27.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

As you can see, for a short list the built-in `sum()` was faster by a factor of about 25 (the exact difference varies depending on your hardware and platform).

Part 4: Benchmarking with 100 values

```
[7]: # Recreate list and array to contain 100 integers starting at 0
     N = 100
     lst = list(range(N))
     arr = np.arange(N)
```

```
[8]: # benchmark built-in sum() with 100 elements
     %timeit sum(lst)
```

```
316 ns ± 2.29 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

```
[9]: # benchmark NumPy's sum() with 100 elements
     %timeit np.sum(arr)
```

```
1.66 μs ± 29.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

For 100 elements, the built-in `sum()` is still faster, but only by a factor of 4 (again, the exact values depend on your platform). Note that the execution time for `np.sum()` remained almost unchanged, which suggest that the function call has a high fixed cost, but scales much better with the number of elements to be summed.

Part 5: Benchmarking with 10,000 values

```
[10]: # Recreate list and array to contain 10,000 integers starting at 0
      N = 10_000
      lst = list(range(N))
      arr = np.arange(N)
```

```
[11]: # benchmark built-in sum() with 10000 elements
      %timeit sum(lst)
```

```
54.5 μs ± 26.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[12]:
```python
# benchmark NumPy's sum() with 10000 elements
%timeit np.sum(arr)
```

```
2.4 µs ± 1.87 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Lastly, for 10,000 elements `np.sum()` is substantially faster by a factor about 20. You should conclude that for large arrays, you can expect much better performance from NumPy's functions, but this may not be true for small data sets.

Exercise 2: Maximizing quadratic utility

Assume that an individual derives utility from consuming c items according to the following utility function $u(\bullet)$:

$$u(c) = -A(c-B)^2 + C$$

where $A > 0$, $B > 0$ and C are parameters, and c is the consumption level.

In this exercise, you are asked to locate the consumption level which delivers the maximum utility.

1. Define a function called `util()` which takes as arguments the consumption level `c` and three additional arguments `A`, `B`, and `C` which are the parameters of $u(\bullet)$ define above. The function should return the utility associated with the given consumption level `c`.
2. Write a function `find_max_cons()` which takes as arguments a sequence of candidate consumption levels and the three parameters `A`, `B`, and `C`, and returns the maximum utility as well as the consumption level at which utility is maximized. The function definition should look like this:

   ```python
   def find_max_cons(candidates, A, B, C):
       """
       Find the consumption level that maximizes utility from a
       list of candidates.

       Parameters
       ----------
       candidates : list or array-like
           List of candidate consumption levels to evaluate.
       A, B, C : float
           Parameters of the utility function.

       Returns
       -------
       u_max
           Maximized utility
       cons_max
           Consumption at which utility is maximized
       """
   ```

   Your algorithm should perform the following steps:

   1. Define the variable `u_max = -np.inf` (negative infinity) as the initial value.
   2. Loop through all candidate consumption levels, and compute the associated utility. If this utility is larger than the previous maximum value `u_max`, update `u_max` and store the associated consumption level `cons_max`.
   3. Return `u_max` and `cons_max` after the loop terminates.
3. Find the maximum:

   1. Create an array `cons` of 51 candidate consumption levels which are uniformly spaced on the interval $[0, 4]$.

      Hint: Use `np.linspace()` for this task.
   2. Use the parameters $A = 1$, $B = 2$, and $C = 10$.
   3. Use the function `find_max_cons()` to compute the maximum utility and the associated optimal consumption level, and print the results.
4. Repeat the exercise, but instead use vectorized operations from NumPy:
   1. Compute and store the utility levels for *all* elements in `cons` at once (simply apply the formula to the whole array).
   2. Locate the index of the maximum utility level using `np.argmax()`.
   3. Use the index returned by `np.argmax()` to retrieve the maximum utility and the corresponding consumption level, and print the results.

Solution.

Part 1: Define the utility function

[13]:
```python
def util(c, A=1.0, B=2.0, C=10.0):
    """
    Evaluate utility for a given level of consumption c.
    """
    u = - A * (c - B)**2.0 + C
    return u
```

Part 2: Define a function to locate the maximum

[14]:
```python
import numpy as np

def find_max_cons(candidates, A=1.0, B=2.0, C=10.0):
    """
    Find the consumption level that maximizes utility from a
    list of candidates.

    Parameters
    ----------
    candidates : list or array-like
        List of candidate consumption levels to evaluate.
    A, B, C : float
        Parameters of the utility function.

    Returns
    -------
    u_max
        Maximized utility
    cons_max
        Consumption at which utility is maximized
    """

    # Initialize max. utility level at the lowest possible value
    u_max = -np.inf

    # Consumption level at which utility is maximized
    cons_max = None
    for c in candidates:
        # Evaluate utility at candidate consumption level
        u = util(c, A, B, C)

        if u > u_max:
            # New maximum found, update utility and optimal consumption
            u_max = u
            cons_max = c

    return u_max, cons_max
```

Part 3: Find the maximum

```
[15]: # Candidate consumption levels
      cons = np.linspace(0.0, 4.0, 51)

      # Parameters
      A = 1.0
      B = 2.0
      C = 10.0
```

```
[16]: # Use function to find maximum
      u_max, cons_max = find_max_cons(cons, A, B, C)

      # Print maximum and maximizer
      print(f'Utility is maximized at c={cons_max} with u={u_max}')

      Utility is maximized at c=2.0 with u=10.0
```

Part 2: Using vectorization with NumPy

```
[ ]: # Evaluate all utilities at once using vectorized NumPy operations
     u = util(cons, A, B, C)

     # Print utility levels
     u

[ ]: array([ 6.    ,  6.3136,  6.6144,  6.9024,  7.1776,  7.44  ,  7.6896,
             7.9264,  8.1504,  8.3616,  8.56  ,  8.7456,  8.9184,  9.0784,
             9.2256,  9.36  ,  9.4816,  9.5904,  9.6864,  9.7696,  9.84  ,
             9.8976,  9.9424,  9.9744,  9.9936, 10.    ,  9.9936,  9.9744,
             9.9424,  9.8976,  9.84  ,  9.7696,  9.6864,  9.5904,  9.4816,
             9.36  ,  9.2256,  9.0784,  8.9184,  8.7456,  8.56  ,  8.3616,
             8.1504,  7.9264,  7.6896,  7.44  ,  7.1776,  6.9024,  6.6144,
             6.3136,  6.    ])
```

```
[18]: # Locate the index of the maximum
      imax = np.argmax(u)

      # Recover the utility and the consumption level at the maximum
      u_max = u[imax]
      cons_max = cons[imax]

      print(f'Utility is maximized at c={cons_max} with u={u_max}')

      Utility is maximized at c=2.0 with u=10.0
```