Workshop 2: Control flow and list comprehensions

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn
NHH Norwegian School of Economics

January 23, 2025

See GitHub repository for notebooks and data:

`https://github.com/richardfoltyn/FIE463-V25`

Exercise 1: CRRA utility function

The CRRA utility function (constant relative risk aversion) is the most widely used utility function in macroeconomics and finance. It is defined as

$$u(c) = \begin{cases} \frac{c^{1-\gamma}}{1-\gamma} & \text{if } \gamma \neq 1 \\ \log(c) & \text{else} \end{cases}$$

where c is consumption and γ is the (constant) risk-aversion parameter, and $\log(\bullet)$ denotes the natural logarithm.

1. You want to evaluate the utility at $c = 2$ for various levels of γ.
    1. Define a list `gammas` with the values 0.5, 1, and 2.
    2. Loop over all elements in `gammas` and evaluate the corresponding utility. Use an `if` statement to correctly handle the two cases from the above formula.

       Hint: Import the `log` function from the `math` module to evaluate the natural logarithm.

       Hint: To perform exponentiation, use the `**` operator (see the list of operator).
    3. Store the utility in a dictionary, using the values of γ as keys, and print the result.
2. Can you solve the exercise using a single list comprehension to create the result dictionary?

   Hint: You will need to use a conditional expression we covered in the lecture.

Exercise 2: Maximizing quadratic utility

Consider the following quadratic utility function

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

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. Find the maximum using a loop:

    1. Create an array `cons` of 51 candidate consumption levels which are uniformly spaced on the interval $[0, 4]$.
    2. Use the parameters $A = 1$, $B = 2$, and $C = 10$.
    3. 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`.
    4. Print `u_max` and `cons_max` after the loop terminates.
2. 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 formulate 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.

Exercise 3: Summing finite values

In this exercise, we explore how to ignore non-finite array elements when computing sums, i.e., elements which are either NaN ("Not a number", represented by `np.nan`), $-\infty$ (`-np.inf`) or ∞ (`np.inf`). Such situations arise if data for some observations is missing and is then frequently encoded as `np.nan`.

1. Create an array of 1001 elements which are uniformly spaced on the interval $[0, 10]$. Set every second element to the value `np.nan`.

   Hint: You can select and overwrite every second element using `start:stop:step` array indexing.

   Using `np.sum()`, verify that the sum of this array is NaN.
2. Write a loop that computes the sum of finite elements in this array. Check that an array element is finite using the function `np.isfinite()` and ignore non-finite elements.

   Print the resulting sum of finite elements.
3. Since this use case is quite common, NumPy implements the function `np.nansum()` which performs exactly this task for you.

   Verify that `np.nansum()` gives the same result and benchmark it against your loop-based implementation.

   Hint: You'll need to use the `%%timeit` cell magic (with two %) if you want to benchmark all code contained in a cell.