Workshop 3: Functions and modules

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn
NHH Norwegian School of Economics

February 5, 2026

See GitHub repository for notebooks and data:

https://github.com/richardfoltyn/FIE463-V26

Exercise 1: Standard deviation of a sequence of numbers

The standard deviation σ characterizes the dispersion of a sequence of data $(x_1, x_2, \ldots, x_N)$ around its mean $\overline{x}$. It is computed as the square root of the variance σ^2, defined as

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} \left(x_i - \overline{x} \right)^2$$

where N is the number of elements (we ignore the degrees-of-freedom correction), and the mean $\overline{x}$ is defined as

$$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

The above formula for the variance can be rewritten as

$$\sigma^2 = \left(\frac{1}{N} \sum_{i=1}^{N} x_i^2 \right) - \overline{x}^2$$

This suggests the following algorithm to compute the standard deviation:

1. Compute the mean $\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$.
2. Compute the mean of squares $S = \frac{1}{N} \sum_{i=1}^{N} x_i^2$.
3. Compute the variance $\sigma^2 = S - \overline{x}^2$.
4. Compute the standard deviation $\sigma = \sqrt{\sigma^2}$.

In this exercise, you are asked to implement the above algorithm and compare your function with NumPy's implementation `np.std()`.

1. Create a module `my_stats.py` and add the function

   ```
   def my_std(x):
       """
       Compute and return the standard deviation of the sequence x.
       """
   ```

   which implements the above algorithm to compute the standard deviation of a given sequence `x` (this could be a tuple, list, array, etc.). Your implementation should *only use built-in functions* such as `len()`, `sum()`, and `sqrt()` from the `math` module.

2. Import this function into the Jupyter notebook. Using an array of 11 elements that are uniformly spaced on the interval $[0, 10]$, confirm that your function returns the same value as `np.std()`.
3. Benchmark your implementation against `np.std()` for three different arrays with 11, 101, and 10001 elements that are uniformly spaced on the interval $[0, 10]$.

   Hint: Use the cell magic `%timeit` to time the execution of a statement.

You should add the following cell magic so that the contents of `my_stat.py` are automatically reloaded whenever you change the file:

[1]:
```
%load_ext autoreload
%autoreload 2
```

Solution.

Part 1 — Implement my_std

You should implement `my_std()` in the separate file `my_stats.py`, but we will implement it directly in the notebook to keep the solution within one file.

[2]:
```
from math import sqrt

def my_std(x):
    """
    Compute standard deviation of x using the built-in functions sum()
    and len().

    Parameters
    ----------
    x: Sequence of numbers

    Returns
    -------
    sd : float
        Standard deviation of x.
    """

    # Number of observations
    N = len(x)

    # Compute mean
    mean = sum(x) / N

    # Compute the mean of squares
    S = sum(xi**2.0 for xi in x) / N

    # Compute variance
    var = S - mean**2.0

    # Compute standard deviation
    sd = sqrt(var)

    return sd
```

Part 2 — Compare with NumPy `std()`

[3]:
```
import numpy as np

# Create the test data
data = np.linspace(0.0, 10.0, 11)
```

[4]:
```
# Uncomment this to import my_std from the separate module
# from my_stats import my_std

# Call your own implementation
my_std(data)
```

[4]:
```
3.1622776601683795
```

[5]:
```
# Call NumPy's implementation
np.std(data)
```

[5]:
```
np.float64(3.1622776601683795)
```

As you can see, both implementations return the same value.

Part 3 — Benchmarks

We now compare the runtime for our own vs. NumPy's implementation for increasing sample sizes. For simplicity, we can directly embed the `%timeit` magic into the loop iterating over sample sizes. This is, strictly speaking, not valid Python syntax and only works in Jupyter notebooks. It should be avoided in real applications.

[6]:
```
# Sample sizes to benchmark
N = [11, 101, 10001]

# Benchmark our own implementation for various sample sizes
for n in N:
    # Create test data of given size
    data = np.linspace(0.0, 10.0, n)
    print(f'Running own implementation for N={n}')
    # Time the execution
    %timeit my_std(data)
```

```
Running own implementation for N=11

3.12 µs ± 7.37 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Running own implementation for N=101

19.1 µs ± 172 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Running own implementation for N=10001

1.77 ms ± 13.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[7]:
```
# Benchmark NumPy implementation for various sample sizes
for n in N:
    # Create test data of given size
    data = np.linspace(0.0, 10.0, n)
    print(f'Running NumPy implementation for N={n}')
    # Time the execution
    %timeit np.std(data)
```

```
Running NumPy implementation for N=11

9.26 µs ± 890 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Running NumPy implementation for N=101
```

```
8.69 µs ± 86.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Running NumPy implementation for N=10001

16.2 µs ± 535 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

As you can see, our implementation is fast for small arrays but substantially slower for arrays of 10,000 elements (by a factor of around 100, depending on hardware and software).

Exercise 2: Locating maximum values

In this exercise, you are asked to write a function that returns the position of the largest element from a given sequence (list, tuple, array, etc.).

1. Write a function `my_argmax()` that takes as an argument a sequence and returns the (first) index where the maximum value is located. Only use built-in functionality in your implementation (no NumPy).
2. Create an array with 101 values constructed using the sine function,

   ```
   arr = np.sin(np.linspace(0.0, np.pi, 101))
   ```

   and use it to test your function.
3. Compare the result returned by your function to NumPy's implementation `np.argmax()`.

Solution.

Part 1 — Implement `my_argmax()`

[8]:
```
def my_argmax(x):
    """
    Return the location of the (first) maximum element.

    Parameters
    ----------
    x : array-like
        A list or array of numerical values.

    Returns
    -------
    int
        The index of the first occurrence of the maximum value in x.
    """

    # Initially, the maximum element must be the first one.
    imax = 0
    xmax = x[0]

    # Iterate through REMAINING elements to see if there is a larger one
    for i, xi in enumerate(x[1:]):
        if xi > xmax:
            # Update the location of the maximum if a larger value is found
            imax = i
            xmax = xi

    return imax
```

Part 2 — Test implementation

[9]:
```
import numpy as np

# Construct test array
arr = np.sin(np.linspace(0.0, np.pi, 101))
```

[10]:
```
# Find maximum location and print the result
i = my_argmax(arr)
print(f'The maximum is located at index {i} with value = {arr[i]}')
```

```
The maximum is located at index 49 with value = 0.9995065603657316
```

Part 3 — Compare with NumPy `argmax()`

[11]:
```
# Find location using NumPy's argmax()
j = np.argmax(arr)
print(f'The maximum is located at index {j} with value = {arr[j]}')
```

```
The maximum is located at index 50 with value = 1.0
```

Exercise 3: Two-period consumption-savings problem

This exercise asks you to find the utility-maximizing consumption levels using grid search, an algorithm that evaluates all possible alternatives from a given set (the "grid") to locate the maximum.

Consider the following standard consumption-savings problem over two periods with lifetime utility $U(c_1, c_2)$ given by

$$\begin{aligned} \max_{c_1,\, c_2} \quad & U(c_1, c_2) = u(c_1) + \beta u(c_2) \\ \text{s.t.} \quad & c_1 + \frac{c_2}{1+r} = w \\ & c_1 \geq 0,\ c_2 \geq 0 \end{aligned}$$

where β is the discount factor, r is the interest rate, w is initial wealth, and (c_1, c_2) is the optimal consumption allocation to be determined. The second line is the budget constraint which ensures that the chosen consumption bundle (c_1, c_2) is feasible. The per-period CRRA utility function $u(c)$ is given by

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

where γ is the coefficient of relative risk aversion (RRA) and $\log(\bullet)$ denotes the natural logarithm.

1. Write a function `util(c, gamma)` which evaluates the per-period utility $u(c)$ for a given consumption level c and the parameter γ. Make sure to take into account the log case!

   Hint: You can use the `np.log()` function from NumPy to compute the natural logarithm.

2. Write a function `util_life(c_1, c_2, beta, gamma)` which uses `util()` from above to compute the lifetime utility $U(c_1, c_2)$ for given consumption levels (c_1, c_2) and parameters.

3. Assume that $r = 0.04$, $\beta = 0.96$, $\gamma = 1$, and $w = 1$.

   - Create a candidate array (grid) of period-1 consumption levels with 100 grid points that are uniformly spaced on the interval $[\epsilon, w - \epsilon]$ where $\epsilon = 10^{-5}$.

     Note that we enforce a minimum consumption level ϵ, as zero consumption yields $-\infty$ utility for the given preferences, which can never be optimal.

- Compute the implied array of period-2 consumption levels from the budget constraint.
- Given these candidate consumption levels, use the function `util_life()` you wrote earlier to evaluate lifetime utility for each bundle of consumption levels (c_1, c_2).

4. Use the function `np.argmax()` to locate the index at which lifetime utility is maximized. Print the maximizing consumption levels (c_1, c_2) as well as the associated maximized utility level.

Solution.

Part 1 — Implement util function

[12]:
```
import numpy as np

def util(c, gamma):
    """
    Return per-period utility for a given consumption level c.
    """
    if gamma == 1:
        # Utility for log preferences
        u = np.log(c)
    else:
        # Utility for general CRRA preferences
        u = c**(1.0 - gamma) / (1.0 - gamma)

    return u
```

Part 2 — Compute lifetime utility

[13]:
```
def util_life(c1, c2, beta, gamma):
    """
    Return lifetime utility for given consumption levels.
    """

    # Utility in period 1
    u1 = util(c1, gamma)

    # Utility in period 2
    u2 = util(c2, gamma)

    # Lifetime utility
    U = u1 + beta * u2

    return U
```

Part 3 — Set parameters and evaluate utility

[14]:
```
# Parameters
r = 0.04
beta = 0.96
gamma = 1.0

# Initial wealth
wealth = 1.0
```

We can now create the candidate grid for period-1 consumption. Period-2 consumption then follows from the budget constraint. The candidate grid is created on the interval $[\epsilon, w - \epsilon]$ for a small value of epsilon. The reason for this is twofold:

1. With CRRA preferences, zero consumption yields $-\infty$ utility, which can never be optimal.
2. Moreover, trying to evaluate `np.log(0)` generates warnings which we wish to avoid.

We avoid these complications by creating a grid of candidate consumption levels on the interval $[\epsilon, w - \epsilon]$ instead of $[0, w]$.

[15]:
```
# Grid size
N = 100

# Minimum consumption level
epsilon = 1.0e-5

# Candidate grid for period-1 consumption
c1_grid = np.linspace(epsilon, wealth - epsilon, N)

# Candidate grid for period-2 consumption (from budget constraint)
c2_grid = (1+r) * (wealth - c1_grid)
```

With the consumption grids at hand, we can evaluate the lifetime utility for each alternative.

[16]:
```
# Evaluate lifetime utility for each (c1, c2)
u_grid = util_life(c1_grid, c2_grid, beta, gamma)
```

Part 4 — Find optimal consumption

[17]:
```
# Find maximum
imax = np.argmax(u_grid)

# Recover the maximizing consumption levels and utility
c1_max = c1_grid[imax]
c2_max = c2_grid[imax]
u_max = u_grid[imax]

# Report consumption levels and utility at maximum
print(f'Utility is maximized at c1={c1_max:.3f}, c2={c2_max:.3f} with u={u_max:.5e}')
```

```
Utility is maximized at c1=0.515, c2=0.504 with u=-1.32060e+00
```