

# 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  $\gamma$  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  $\gamma$ .
  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:

```
from math import log
```

*Hint:* To perform exponentiation, use the `**` operator (see the [list of operator](#)).

3. Store the utility in a dictionary, using the values of  $\gamma$  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.

---

*Solution.*

### Part (1)

```
[1]: # Define list of gammas that should be considered
      gammas = [0.5, 1.0, 2.0]
```

```
[2]: # Import log function from math module
      from math import log
```

```

# Create empty dictionary to store gamma & utility values
utils = {}

# Consumption level at which to evaluate utility
cons = 2.0

for gamma in gammas:
    if gamma == 1.0:
        # Handle log case
        u = log(cons)
    else:
        # Handle general CRRA case
        u = cons**(1.0-gamma) / (1.0 - gamma)

    # Store resulting utility level in dictionary
    utils[gamma] = u

# Print utility levels
utils

```

```
[2]: {0.5: 2.8284271247461903, 1.0: 0.6931471805599453, 2.0: -0.5}
```

## Part (2)

It is possible to compress the entire loop into a single list comprehension. We need to use a conditional expression within the list comprehension to correctly handle the two CRRA cases.

```

[3]: # Compute utilities in a single list comprehension
utils = {
    gamma: log(cons) if gamma == 1.0 else cons**(1.0-gamma) / (1.0-gamma)
    for gamma in gammas
}

```

```

[4]: # Print utility levels
utils

```

```
[4]: {0.5: 2.8284271247461903, 1.0: 0.6931471805599453, 2.0: -0.5}
```

## 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 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)

```
[5]: # Import NumPy
import numpy as np

# Candidate consumption levels
cons = np.linspace(0.0, 4.0, 51)

[6]: # Parameters
A = 1.0
B = 2.0
C = 10.0

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

# Consumption level at which utility is maximized
cons_max = None

# Evaluate utility for each candidate consumption level, update maximum
for c in cons:
    u = - A * (c - B)**2.0 + C
    if u > u_max:
        # New maximum found, update values
        u_max = u
        cons_max = 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)

```
[8]: # Evaluate all utilities at once using vectorized NumPy operations
util = -A * (cons - B)**2.0 + C

# Print utility levels
util
```

```
[8]: 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.      ])
```

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

# Recover the utility and the consumption level at the maximum
u_max = util[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

---

## 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  $\infty$  (`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.

---

**Solution.**

### Part (1)

```
[10]: import numpy as np

# Create uniformly spaced data
data = np.linspace(0, 10, 1001)

# Set every second element to NaN
data[1::2] = np.nan
```

```
# Print first 10 elements to illustrate pattern
data[:10]
```

```
[10]: array([0. , nan, 0.02, nan, 0.04, nan, 0.06, nan, 0.08, nan])
```

```
[11]: # Check that "standard" summation returns NaN
      np.sum(data)
```

```
[11]: np.float64(nan)
```

## Part (2)

To implement a loop summing all finite elements, we proceed as follows:

```
[12]: # Initialize sum to zero
      s = 0.0

      # Loop through elements and only add them if they are finite
      for x in data:
          if np.isfinite(x):
              # Add finite element
              s += x

      print(f'The sum of finite values is {s}')
```

The sum of finite values is 2505.0000000000005

## Part (3)

To benchmark the loop, we copy the code from above but remove the `print()` statement as we are not interested in benchmarking that part. Note that we need to use the `%%timeit` cell magic with *two* leading `%` to benchmark the entire cell, not just the first line.

```
[13]: %%timeit
      # Initialize sum to zero
      s = 0.0

      # Loop through elements and only add them if they are finite
      for x in data:
          if np.isfinite(x):
              # Add finite element
              s += x
```

664  $\mu$ s  $\pm$  1.66  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 1,000 loops each)

We can compare this to NumPy's `np.nansum()` function:

```
[14]: %timeit np.nansum(data)
```

8.11  $\mu$ s  $\pm$  55.9 ns per loop (mean  $\pm$  std. dev. of 7 runs, 100,000 loops each)

As you can see, `np.nansum()` is approximately 80 times faster (the exact value depends on your hardware and software).

---

## Exercise 4: Approximating the sum of a geometric series

Let  $\alpha \in (-1, 1)$ . The sum of the geometric series  $(1, \alpha, \alpha^2, \dots)$  is given by

$$\sigma = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}$$

In this exercise, you are asked to approximate this sum using the first  $N$  values of the sequence, i.e.,

$$\sigma \approx s_N = \sum_{i=0}^N \alpha^i$$

where  $N$  is chosen to be sufficiently large.

1. Assume that  $\alpha = 0.9$ . Write a while loop to approximate the sum  $\sigma$  by computing  $s_N$  for an increasing  $N$ . Terminate the computation as soon as an additional increment  $\alpha^N$  is smaller than  $10^{-10}$ . Compare your result to the exact value  $\sigma$ .
2. Now assume that  $\alpha = -0.9$ . Adapt your previous solution so that it terminates when the *absolute value* of the increment is less than  $10^{-10}$ . Compare your result to the exact value  $\sigma$ .

*Hint:* Use the built-in function `abs()` to compute the absolute value.

---

### Solution.

#### Part (1)

```
[15]: alpha = 0.9

# True sum
sigma = 1 / (1 - alpha)

# Termination tolerance
tol = 1e-10

# Variable to store the running sum
s_N = 0.0
# Iteration counter
N = 0

while True:
    # Increment to be added to sum
    increment = alpha**N

    # Check whether increment satisfies termination criterion
    if increment < 1e-10:
        break

    # Increment sum and loop counter
    s_N += increment
    N += 1

print(f'Approximation: {s_N:.10f}')
print(f'Exact value: {sigma:.10f}')
print(f'Difference: {sigma-s_N:.8e}')
print(f'Number of iterations: {N}')
```

```
Approximation: 9.9999999990
Exact value: 10.0000000000
Difference: 9.53034984e-10
Number of iterations: 219
```

## Part (2)

```
[16]: alpha = -0.9

# True sum
sigma = 1 / (1 - alpha)

# Variable to store the running sum
s_N = 0.0
# Iteration counter
N = 0

while True:
    # Increment to be added to sum
    increment = alpha**N

    # Check whether increment satisfies termination criterion
    if abs(increment) < 1e-10:
        break

    # Increment sum and loop counter
    s_N += increment
    N += 1

print(f'Approximation: {s_N:.10f}')
print(f'Exact value:    {sigma:.10f}')
print(f'Difference: {sigma-s_N:.8e}')
print(f'Number of iterations: {N}')
```

```
Approximation: 0.5263157895
Exact value:   0.5263157895
Difference: -5.01603203e-11
Number of iterations: 219
```

---