Workshop 1: Language and NumPy basics

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn
NHH Norwegian School of Economics

January 22, 2026

See GitHub repository for notebooks and data:

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

Exercise 1: Type conversions

In the lecture, we discussed the basic built-in data types: integers, floating-point numbers, booleans, and strings. Python allows us to convert one type to another using the following functions:

- `int()` converts its argument to an integer.
- `float()` converts its argument to a floating-point number.
- `bool()` converts its argument to a boolean.
- `str()` converts its argument to a string.

These conversions mostly work in an intuitive fashion, with some exceptions. Perform the following tasks to see how they work in detail:

1. Define a string variable `s` with the value `'1'`. Convert this variable to an integer, a float, and a boolean. Do you get the same behavior if you define the string `s` to be `'1.0'` instead?
2. Define the string variables `s1`, `s2`, and `s3` with values `'True'`, `'False'`, and `''` (empty string), respectively. Convert each of these to a boolean. Can you guess the conversion rule?
3. Define a floating-point variable `x` with the value `0.9`. Convert this variable to an integer, a boolean, and a string.
4. Define the integer variables `i1` and `i2` with values `0` and `2`, respectively. Convert each of these variables to a boolean.
5. Define the boolean variables `b1` and `b2` with values `True` and `False`, respectively. Convert each of them to an integer.
6. NumPy arrays cannot be converted using `int()`, `float()`, etc. Instead, we have to use the method `astype()` and pass the desired data type (e.g., `int`, `float`, `bool`) as an argument.

   Create a NumPy array called `arr` with elements `[0.0, 0.5, 1.0]` and convert it to an integer and a boolean type.

Exercise 2: Working with strings

Strings in Python are full-fledged objects, i.e., they contain both the character data as well as additional functionality implemented via functions or so-called *methods*. The official documentation provides a comprehensive list of these methods. For our purposes, the most important are:

- `str.lower()` and `str.upper()` convert the string to lower or upper case, respectively.
- `str.strip()` removes any leading or trailing whitespace characters from a string.
- `str.count()` returns the number of occurrences of a substring within a string.
- `str.startswith()` and `str.endswith()` check whether a string starts or ends with a given substring.

Important: These methods need to be applied to a particular string variable, not the class `str` itself. For example, if you have a string variable `s`, you use `s.lower()`, etc.

Moreover, strings are also sequences and as such support indexing in the same way as lists or tuples, so for example `'NHH'[1]` returns the 2nd character `'H'`.

In this exercise, you are asked to apply a few of these concepts. Create a string variable with the value

```
s = '  NHH Norwegian School of Economics  '
```

and perform the following tasks:

1. Strip the surrounding spaces from the string using `strip()`.
2. Count the number of `'H'` in the string.
3. Modify your code so that it is case-insensitive, i.e., both instances of `'h'` and `'H'` are counted.
4. Reverse the string, i.e., the last character should come first, and so on.
5. Create a new string which contains every 2nd letter from the original.
6. Select the last character from this new string using at least two different methods.

Exercise 3: Summing lists and arrays

In this exercise, we investigate an additional difference between built-in lists and NumPy arrays: performance. You are asked to investigate performance differences for 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 object 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?