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.

Solution.

Part 1 — String conversion example

[1]:
```python
# Define the string variable
s = '1'
```

[2]:
```python
# Convert to integer
int(s)
```

[2]:
```
1
```

[3]:
```python
# Convert to float
float(s)
```

[3]:
```
1.0
```

[4]:
```python
# Convert to boolean: this returns True because any non-empty string is considered True
bool(s)
```

[4]:
```
True
```

The behavior of `float()` and `bool()` is identical if we define `s='1.0'` instead, but the `int()` function fails because the string contains a decimal point even though it can still be interpreted as an integer value of 1.

[5]:
```python
# Define a new string variable with a decimal point
s = '1.0'

# Attempt to convert to integer: this fails because the string contains a decimal point
int(s)
```

```
ValueError: invalid literal for int() with base 10: '1.0'
```

[6]:
```python
# Convert to float - works as before
float(s)
```

[6]:
```
1.0
```

[7]:
```python
# Convert to boolean: works as before
bool(s)
```

[7]:
```
True
```

Part 2 — Boolean string conversion

[8]:
```python
# Define string variables
s1 = 'True'
s2 = 'False'
s3 = ''
```

[9]:
```python
# Convert 'True' to boolean
bool(s1)
```

[9]:
```
True
```

[10]:
```python
# Convert 'False' to boolean
bool(s2)
```

```
[10]: True
```

```
[11]: # Convert '' to boolean
      bool(s3)
```

```
[11]: False
```

Part 3 — Float conversion examples

```
[12]: # Define floating-point number
      x = 0.9
```

```
[13]: # Convert to integer: The fractional part is truncated, NOT rounded!
      int(x)
```

```
[13]: 0
```

```
[14]: # Convert to boolean: Any numeric value other than 0 is interpreted as True
      bool(x)
```

```
[14]: True
```

```
[15]: # Convert to string
      str(x)
```

```
[15]: '0.9'
```

Part 4 — Integer to boolean

```
[16]: # Define integer variables
      i1 = 0
      i2 = 2
```

```
[17]: # Convert 0 to boolean: the numeric value 0 is interpreted as False
      bool(i1)
```

```
[17]: False
```

```
[18]: # Convert 2 to boolean: any non-zero numeric value is interpreted as True
      bool(i2)
```

```
[18]: True
```

Part 5 — Boolean to integer

```
[19]: # Define boolean variables
      b1 = True
      b2 = False
```

```
[20]: # Convert True to integer
      int(b1)
```

```
[20]: 1
```

```
[21]: # Convert False to integer
      int(b2)
```

```
[21]: 0
```

Part 6 — NumPy array conversions

```
[22]: # Import numpy
      import numpy as np

      # Create array of floating-point numbers
      arr = np.array([0.0, 0.5, 1.0])
      arr
```

```
[22]: array([0. , 0.5, 1. ])
```

```
[23]: # Convert to integer array: note that values are again truncated, NOT rounded.
      arr.astype(int)
```

```
[23]: array([0, 0, 1])
```

```
[24]: # Convert to boolean array
      arr.astype(bool)
```

```
[24]: array([False,  True,  True])
```

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.

Solution.

Part 1 — Strip whitespace

[25]:
```
# Define the string
s = '  NHH Norwegian School of Economics  '

# strip leading & trailing whitespace
s = s.strip()
s
```

[25]:
```
'NHH Norwegian School of Economics'
```

Part 2 — Count letter H

[26]:
```
# Count the number of H in the string
s.count('H')
```

[26]:
```
2
```

Part 3 — Case-insensitive count

[27]:
```
# Count number of H, ignore case:
# We first convert the string to lowercase, then look for the number of h
s.lower().count('h')
```

[27]:
```
3
```

Part 4 — Reverse string

[28]:
```
# Reverse the string
s[::-1]
```

[28]:
```
'scimonocE fo loohcS naigewroN HHN'
```

Recall that the slice `start:stop:step` can be used to index sequences *and* strings. We use the default values for `start` and `stop` (so they can be omitted), but set the step to `-1` to reverse the order.

Part 5 — Every second character

[29]:
```
s2 = s[::2]
s2
```

[29]:
```
'NHNreinSho fEoois'
```

Part 6 — Select last character

[30]:
```
# Select last element using -1
s2[-1]
```

[30]:
```
's'
```

```
[31]: # Select last element using len()
      s2[len(s2) - 1]
```

```
[31]: 's'
```

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?

Solution.

Part 1 — Create list and array

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

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

Part 2 — Sum using functions

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

```
[34]: 45
```

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

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

Part 3 — Benchmark with %timeit

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

```
65.3 ns ± 0.236 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

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

```
1.6 µs ± 20.7 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 — Benchmark 100 elements

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

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

```
397 ns ± 10 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

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

```
1.63 µs ± 32.5 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 suggests that the function call has a high fixed cost but scales much better with the number of elements to be summed.

Part 5 — Benchmark 10000 elements

```
[41]: # Recreate list and array to contain 10,000 integers starting at 0
      N = 10000
      lst = list(range(N))
      arr = np.arange(N)
```

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

```
54.3 µs ± 246 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

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

```
2.4 µs ± 12.3 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 of 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 datasets.