Lecture 2: Language and NumPy basics

FIE463: Numerical methods in Macroeconomics and Finance using Python

Richard Foltyn
Norwegian School of Economics (NHH)

January 14, 2025

See GitHub repository for notebooks and data:

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

Contents

2 Language and NumPy basics

In this unit, we start exploring the Python language, covering the following topics:

1. Basic syntax
2. Built-in data types
3. NumPy arrays

2.1 Basic syntax

- Everything after a # character (until the end of the line) is a comment and will be ignored.
- Variables are created using the assignment operator =.
- Variable names are case sensitive.
- Whitespace characters matter (unlike in most languages)!

- Python uses indentation (usually 4 spaces) to group statements, for example loop bodies, functions, etc.
- You don't need to add a character to terminate a line, unlike in some languages.
- You can use the `print()` function to inspect almost any object.

[1]:
```python
# First example

# create a variable named 'text' that stores the string 'Hello, world!'
text = 'Hello, world!'

# print contents of 'text'
print(text)
```

```
Hello, world!
```

In Jupyter notebooks and interactive command-line environments, we can also display a value by simply writing the variable name.

[2]:
```python
text
```

[2]: `'Hello, world!'`

Alternatively, we don't even need to create a variable but can instead directly evaluate expressions and print the result:

[3]:
```python
2*3
```

[3]: `6`

This does not print anything in *proper* Python script files (ending in `.py`) that are run through the interpreter, though.

Calling `print()` is also useful if we want to display multiple expressions from a single notebook cell, as otherwise only the last value is shown:

[4]:
```python
text = 'Hello world!'
var = 1
text            # does NOT print contents of text
var             # prints only value of var
```

[4]: `1`

[5]:
```python
print(text)     # print text explicitly
var             # var is shown automatically
```

```
Hello world!
```

[5]: `1`

2.2 Built-in data types

Pythons is a dynamically-typed language:

- Unlike in C or Fortran, you don't need to declare a variable or its type.
- You can inspect a variable's type using the built-in `type()` function, but you rarely need to do this.

We now look at the most useful built-in data types:

Basic types

- integers (`int`)

- floating-point numbers (`float`)
- boolean (`bool`)
- strings (`str`)

Containers (or collections)

- tuples (`tuple`)
- lists (`list`)
- dictionaries (`dict`)

2.2.1 Integers and floats

Integers and floats (floating-point numbers) are the two main built-in data types to store numerical data (we ignore complex numbers in this course). Floating-point is the standard way to represent real numbers on computers since these cannot store real numbers with arbitrary precision.

[6]:
```python
# Integer variables
i = 1
type(i)
```

[6]:
```
int
```

[7]:
```python
# Floating-point variables
x = 1.0
type(x)
```

[7]:
```
float
```

[8]:
```python
# A name can reference any data type:
# Previously, x was a float, now it's an integer!
x = 1
type(x)
```

[8]:
```
int
```

It is good programming practice to specify floating-point literals using a decimal point. It makes a difference in a few cases (especially when using NumPy arrays, or Python extensions such as Numba or Cython):

[9]:
```python
x = 1.0          # instead of x = 1
```

A boolean (`bool`) is a special integer type that can only store two values, `True` and `False`. We create booleans by assigning one of these values to a variable:

[10]:
```python
x = True
x = False
type(x)
```

[10]:
```
bool
```

Boolean values are most frequently used for conditional execution, i.e., a block of code is run only when some variable is `True`. We study conditional execution in the next unit.

2.2.2 Strings

The string data type stores sequences of characters:

[11]:
```python
# Strings need to be surrounded by single (') or double (") quotes!
institution = 'Norwegian School of Economics'
institution = "Norwegian School of Economics"
```

Strings support various operations some of which we explore in the exercises at the end of this section. For example, we can use the addition operation `+` to concatenate strings:

[12]:
```python
# Define two strings
str1 = 'Python'
str2 = 'course'

# Concatenate strings using +
str1 + ' ' + str2
```

[12]:
```
'Python course'
```

An extremely useful variant of strings are the so-called *f-strings*. These allow us to dynamically insert a variable value into a string, a feature we'll use throughout this course.

[13]:
```python
# Approximate value of pi
pi = 3.1415

# Use f-strings to embed the value of the variable version inside the string
s = f'Pi is approximately equal to {pi}'
s
```

[13]:
```
'Pi is approximately equal to 3.1415'
```

2.2.3 Tuples

Tuples represent an *ordered, immutable collection* of items which can have different data types. They are created whenever several items are separated by commas:

[14]:
```python
# A tuple containing a string, an integer and a float
items = 'foo', 1, 1.0
items
```

[14]:
```
('foo', 1, 1.0)
```

The parenthesis are optional, but improve readability:

[15]:
```python
items = ('foo', 1, 1.0)        # equivalent way to create a tuple
items
```

[15]:
```
('foo', 1, 1.0)
```

We use brackets `[]` to access an element in a tuple (or any other container object). Elements in tuples need to be accessed by their position or *index*.

[16]:
```python
first = items[0]            # variable first now contains 'foo'
first
```

[16]:
```
'foo'
```

Python indices are 0-based, so 0 references the *first* element, 1 the second element, etc.

[17]:
```python
second = items[1]           # second element
second
```

[17]:
```
1
```

Tuples and any other Python collections support automatic unpacking if we want to extract multiple (or all) values at once:

[18]:
```python
first, second, third = items

# Print first element
first
```

[18]:
```
'foo'
```

If we are not interested in extracting all items, we can collect any remaining items using a `*` as follows:

[19]:
```python
first, *rest  = items

# Rest contains a list of all remaining items
rest
```

[19]:
```
[1, 1.0]
```

Tuples are *immutable*, which means that the contents of a tuple cannot be changed. (Technically, the *references* to elements stored in the tuple cannot be changed.)

[20]:
```python
# This raises an error!
items = 'foo', 1, 1.0
items[0] = 123
```

```
TypeError: 'tuple' object does not support item assignment
```

2.2.4 Lists

Lists are like tuples, except that they can be modified (i.e., they are *mutable*). We create lists using brackets:

[21]:
```python
# Create list which contains a string, an integer and a float
lst = ['foo', 1, 1.0]
lst
```

[21]:
```
['foo', 1, 1.0]
```

Accessing list items works the same way as with tuples

[22]:
```python
lst[0]                  # print first item
```

[22]:
```
'foo'
```

Lists items can be modified:

[23]:
```python
lst[0] = 'bar'          # first element is now 'bar'
lst
```

[23]:
```
['bar', 1, 1.0]
```

Lists are full-fledged objects that support various operations, for example

[24]:
```python
lst.insert(0, 'abc')    # insert element at position 0
lst.append(2.0)         # append element at the end
del lst[3]              # delete the 4th element
lst
```

[24]: ['abc', 'bar', 1, 2.0]

The built-in function `len()` returns the number of elements in a list (and any other container object)

```
[25]: len(lst)
```

[25]: 4

2.2.5 Dictionaries

Dictionaries are container objects that map keys to values.

- Both keys and values can be (almost any) Python objects, even though we often use strings as keys.
- Dictionaries are created using curly braces: `{key1: value1, key2: value2, ...}`, or by using the `dict()` constructor `dict(key1=value1, key2=value2, ...)`.

For example, to create a dictionary with three items we write

```
[26]: dct = {
    'institution': 'NHH',
    'course': 'Python course',
    'year': 2025
}
dct
```

[26]: {'institution': 'NHH', 'course': 'Python course', 'year': 2025}

The alternative way to create dictionaries using the `dict()` constructor is less powerful and supports only keys that are strings. For most cases, this is sufficient:

```
[27]: # Alternative way to define the same dictionary
dct = dict(institution='NHH', course='Python course', year=2025)
dct
```

[27]: {'institution': 'NHH', 'course': 'Python course', 'year': 2025}

Specific values are accessed using the syntax `dict[key]`:

```
[28]: dct['institution']
```

[28]: 'NHH'

We can use the same syntax to either modify an existing key or add a new key-value pair:

```
[29]: dct['course'] = 'Introduction to Python'        # modify value of existing key
dct['city'] = 'Bergen'                          # add new key-value pair
dct
```

```
[29]: {'institution': 'NHH',
 'course': 'Introduction to Python',
 'year': 2025,
 'city': 'Bergen'}
```

Moreover, we can use the methods `keys()` and `values()` to get the collection of a dictionary's keys and values:

```
[30]: dct.keys()
```

[30]: dict_keys(['institution', 'course', 'year', 'city'])

[31]:
```
dct.values()
```

[31]:
```
dict_values(['NHH', 'Introduction to Python', 2025, 'Bergen'])
```

When we try to retrieve a key that is not in the dictionary, this will produce an error:

[32]:
```
dct['country']
```

```
KeyError: 'country'
```

One way to get around this is to use the `get()` method which accepts a default value that will be returned whenever a key is not present:

[33]:
```
dct.get('country', 'Norway')        # return 'Norway' if 'country' is
                                    # not a valid key
```

[33]:
```
'Norway'
```

2.3 NumPy arrays

NumPy is a library that allows us to efficiently store and access (mainly) numerical data and apply numerical operations similar to those available in Matlab.

- NumPy is not part of the core Python project.
- Python itself has an array type, but there is really no reason to use it. Use NumPy!
- NumPy types and functions are not built-in, we must first import them to make them visible. We do this using the `import` statement.

The convention is to make NumPy functionality available using the `np` namespace:

[34]:
```
# Access functionality from NumPy using the 'np' short-hand
import numpy as np
```

2.3.1 Creating arrays

Creating arrays from other Python objects

Arrays can be created from other objects such as lists and tuples by calling `np.array()`

[35]:
```
# Create array from list [1,2,3]
arr = np.array([1, 2, 3])
arr
```

[35]:
```
array([1, 2, 3])
```

[36]:
```
# Create array from tuple
arr = np.array((1.0, 2.0, 3.0))
arr
```

[36]:
```
array([1., 2., 3.])
```

[37]:
```
# Create two-dimensional array from nested list
arr = np.array([[1, 2, 3], [4, 5, 6]])
arr
```

[37]: array([[1, 2, 3],
       [4, 5, 6]])

Array creation routines

Additionally, NumPy offers a multitude of functions to create new arrays from scratch.

[38]:
```
# Create a 1-dimensional array with 10 elements, initialize values to 0.
arr = np.zeros(10)
arr
```

[38]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

[39]:
```
arr1 = np.ones(5)       # vector of five ones
arr1
```

[39]: array([1., 1., 1., 1., 1.])

We can also create sequences of integers using the `np.arange()` function:

[40]:
```
arr2 = np.arange(5)     # vector [0,1,2,3,4]
arr2
```

[40]: array([0, 1, 2, 3, 4])

`np.arange()` accepts initial values and increments as optional arguments. The end value is *not* included.

[41]:
```
start = 2
end = 10
step = 2
arr3 = np.arange(start, end, step)
arr3
```

[41]: array([2, 4, 6, 8])

As in Matlab, there is a `np.linspace()` function that creates a vector of uniformly-spaced real values.

[42]:
```
# Create 11 elements, equally spaced on the interval [0.0, 1.0]
arr5 = np.linspace(0.0, 1.0, 11)
arr5
```

[42]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.])

We create arrays of higher dimension by specifying the desired shape. Shapes are specified as `tuple` arguments; for example, the shape of an $m \times n$ matrix is `(m,n)`.

[43]:
```
mat = np.ones((2,2))    # Create 2x2 matrix of ones
mat
```

[43]: array([[1., 1.],
       [1., 1.]])

2.3.2 Reshaping arrays

The `reshape()` method of an array object can be used to reshape it to some other (conformable) shape.

[44]:
```
# Create vector of 4 elements and reshape it to a 2x2 matrix
mat = np.arange(4).reshape((2,2))
mat
```

[44]:
```
array([[0, 1],
       [2, 3]])
```

[45]:
```
# reshape back to vector of 4 elements
vec = mat.reshape(4)
vec
```

[45]:
```
array([0, 1, 2, 3])
```

We use `-1` to let NumPy automatically compute the size of *one* remaining dimension.

[46]:
```
# with 2 dimensions, second dimension must have size 2
mat = np.arange(4).reshape((2, -1))
mat
```

[46]:
```
array([[0, 1],
       [2, 3]])
```

If we want to convert an arbitrary array to a vector, we can alternatively use the `flatten()` method.

[47]:
```
mat.flatten()
```

[47]:
```
array([0, 1, 2, 3])
```

Important: the reshaped array must have the same number of elements!

[48]:
```
mat = np.arange(6).reshape((2,-1))
mat.reshape((2,2))      # Cannot reshape 6 into 4 elements!
```

```
ValueError: cannot reshape array of size 6 into shape (2,2)
```

2.3.3 Indexing

Single element indexing

To retrieve a single element, we specify the element's index on each axis ("axis" is the NumPy terminology for an array dimension).

- Remember that just like Python in general, NumPy arrays use 0-based indices.
- Unlike lists or tuples, NumPy arrays support multi-dimensional indexing.

[49]:
```
import numpy as np

mat = np.arange(6).reshape((3,2))
mat
```

[49]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

[50]:
```
mat[0,1]    # returns element in row 1, column 2
```

[50]:
```
1
```

It is important to pass multi-dimensional indices as a tuple within brackets, i.e., `[0,1]` in the above example. We could alternatively write `mat[0][1]`, which would give the same result:

[51]:
```
mat[0][1]        # don't do this!
```

[51]: 1

This is substantially less efficient, though, as it first creates a sub-dimensional array `mat[0]`, and then applies the second index to this array.

Index slices

There are numerous ways to retrieve a subset of elements from an array. The most common way is to specify a triplet of values `start:stop:step` called `slice` for some axis.

Indexing with slices can get quite intricate. Some basic rules:

- all tokens in `start:stop:step` are optional, with the obvious default values. We could therefore write `::` to include all indices, which is the same as `:`
- The end value is *not* included. Writing `vec[0:n]` does not include element with index n!
- Any of the elements of `start:stop:step` can be negative.
  - If `start` or `stop` are negative, elements are counted from the end of the array: `vec[:-1]` retrieves the whole vector except for the last element.
  - If `step` is negative, the order of elements is reversed.

[52]:
```
vec = np.arange(5)

# These are equivalent ways to return the WHOLE vector
vec[0:5:1]      # all three tokens present
vec[::]         # omit all tokens
vec[:]          # omit all tokens
vec[:5]         # end value only
vec[-5:]        # start value only, using negative index
```

[52]: array([0, 1, 2, 3, 4])

You can reverse the order like this:

[53]:
```
vec[::-1]
```

[53]: array([4, 3, 2, 1, 0])

With multi-dimensional arrays, the above rules apply for each dimension.

[54]:
```
# Create a 2x3 matrix
mat = np.arange(6).reshape((2,3))
mat
```

[54]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[55]:
```
# Retrieve only the first and third columns:
mat[0:2,0:3:2]
```

[55]:
```
array([[0, 2],
       [3, 5]])
```

We can omit indices for higher-order dimensions if all elements should be included.

[56]:
```
mat[1]      # includes all columns of row 2; same as mat[1,:]
```

[56]: array([3, 4, 5])

We cannot omit the indices for *leading* axes, though! If an entire leading axis is to be included, we specify this using `:`

[57]:
```
mat[:, 1]   # includes all rows of column 2
```

[57]:
```
array([1, 4])
```

Indexing lists and tuples

The basic indexing rules we have covered so far also apply to the built-in `tuple` and `list` types. However, `list` and `tuple` do not support advanced indexing available for NumPy arrays which we study in later units.

[58]:
```
# Apply start:stop:step indexing to tuple
tpl = (1,2,3)
tpl[:3:2]
```

[58]:
```
(1, 3)
```

2.3.4 Numerical data types (advanced)

We can explicitly specify the numerical data type when creating NumPy arrays.

So far we haven't done so, and then NumPy does the following:

- Functions such as `zeros()` and `ones()` default to using `np.float64`, a 64-bit floating-point data type (this is also called *double precision*)
- Other functions such as `arange()` and `array()` inspect the input data and return a corresponding array.
- Most array creation routines accept a `dtype` argument which allows you to explicitly set the data type.

Examples:

[59]:
```
import numpy as np

# Floating-point arguments return array of type np.float64
arr = np.arange(1.0, 5.0, 1.0)
arr.dtype
```

[59]:
```
dtype('float64')
```

[60]:
```
# Integer arguments return array of type np.int64
arr = np.arange(1,5,1)
arr.dtype
```

[60]:
```
dtype('int64')
```

Often we don't care about the data type too much, but keep in mind that

- Floating-point has limited precision, even for integers if these are larger than (approximately) 10^{16}
- Integer values cannot represent fractional numbers and (often) have a more limited range.

This might lead to surprising consequences:

[61]:
```
# Create integer array
arr = np.ones(5, dtype=np.int64)
# Store floating-point in second element
arr[1] = 1.234
arr
```

```
[61]: array([1, 1, 1, 1, 1])
```

The array is unchanged because it's impossible to represent 1.234 as an integer value!

The take-away is to explicitly write floating-point literal values and specify a floating-point `dtype` argument when we want data to be interpreted as floating-point values. For example, always write 1.0 instead of 1, unless you *really* want an integer!

2.4 Optional exercises

Exercise 1: string operations

Experiment with operators applied to strings and integers:

1. Define two string variables using the values `'Hello'` and `'World'`, and concatenate them using `+`. Modify your solution to add a space.
2. Define a string variable `'NHH'` and multiply it by 2 using `*`. What happens?
3. Define a string variable `'Hello'` and use the `+=` assignment operator to append another string `' World'`.

   The `+=` operator is one of several operators in Python that combine assignment with another operation, such as addition. In this particular case, these statements are equivalent:

   ```
   a += b
   a = a + b
   ```

Exercise 2: string formatting with f-strings

We frequently want to create strings that incorporate integer and floating-point data, possibly formatted in a particular way.

Python offers quite powerful formatting capabilities which can become so complex that they are called the *Format Specification Mini-Language* (see the docs). In this exercise, we explore a small but useful subset of formatting instructions.

A format specification is a string that contains one or several `{}`, for example:

```
[62]: version = 3.13
      f'The current version of Python is {version}'
```

```
[62]: 'The current version of Python is 3.13'
```

What if we want to format the float `3.13` in a particular way? We can augment the `{}` to achieve that goal. For example, if the data to be formatted is of type integer, we can specify

- `{:wd}` where `w` denotes the total field width and `d` indicates that the data type is an integer.

  To print an integer into a field that is 3 characters wide, we would thus write `{:3d}`.

For floats we have additional options:

- `{:w.df}` specifies that a float should be formatted using a field width `w` and `d` decimal digits.

  To print a float into a field of 10 characters using 5 decimal digits, we would thus specify `{:10.5f}`
- `{:w.de}` is similar, but instead uses scientific notation with exponents.

  This is particularly useful for very large or very small numbers.
- `{:w.dg}`, where `g` stands for *general* format, is a superset of `f` and `e` formatting. Either fixed or exponential notation is used depending on a number's magnitude.

In all these cases the field width `w` is optional and can be omitted. Python then uses as many characters as are required.

Now what we have introduced the formatting language, you are asked to perform the following exercises:

1. Modify the above f-string so that only the first decimal digit of the Python version is printed.
2. Modify the above f-string, but truncate the Python version to *not* include any decimal digits. Does this work with the integer formatting specification `'{:d}'`?
3. Print π using a precision of 10 decimal digits. *Hint:* the value of π is available as

   ```
   from math import pi
   ```

4. Print e^{10}, computed as `exp(10.0)`, using exponential notation and three decimal digits. *Hint:* To use the exponential function, you need to import it using

   ```
   from math import exp
   ```

Exercise 3: operations on tuples and lists

Create two lists `a` and `b` with the values `1,2,3` and `'a','b','c'`, respectively. Perform the following tasks and examine their results:

1. Concatenate the two lists using `+`.
2. Multiply the list `a` by the integer 2.
3. Append the elements `['x', 'y', 'z']` to `b` using the `+=` operator. Alternatively, do this using the list method `extend()`. Is the list `b` modified in place?
4. Append the integer `10` to `b` using the `+=` operator.
5. Duplicate the list `a` using the `*=` operator. Is the list `a` modified in place?

Repeat steps 1-5 using *tuples* instead of lists.

Finally, create a list and a tuple and try to add them using `+`. Does this work?

2.5 Solutions

Solution for exercise 1

[63]:
```
# 1. string concatenation using addition
str1 = 'Hello'
str2 = 'World'

# Concatenate two strings using +
str1 + str2
```

[63]:
```
'HelloWorld'
```

Note that this does not insert a space inbetween, so we have to do this manually:

[64]:
```
str1 + ' ' + str2
```

[64]:
```
'Hello World'
```

[65]:
```
# 2. string multiplication by integers
str1 = 'NHH'
# Repeat string using multiplication!
str1 * 2
```

[65]: 'NHHNHH'

[66]:
```
# 3. Append using +=
str1 = 'Hello'
str1 += ' World'       # Append ' World' to value in str1, assign result to str1
str1
```

[66]: 'Hello World'

Solution for exercise 2

[67]:
```
# 1. Print Python version with only one decimal digit
version = 3.13
f'The current version of Python is {version:.1f}'
```

[67]: 'The current version of Python is 3.1'

[68]:
```
# 2. Truncate all decimal digits
# To do this, we use floating-point formatting with 0 decimal digits.
f'The current version of Python is {version:.0f}'
```

[68]: 'The current version of Python is 3'

Note that this does not work with the integer formatting specification because that one does not accept any float-valued variables:

[69]:
```
f'The current version of Python is {version:d}'
```

```
ValueError: Unknown format code 'd' for object of type 'float'
```

[70]:
```
# 3. Print pi using 10 decimal digits
from math import pi
f'The first 10 digits of pi: {pi:.10f}'
```

[70]: 'The first 10 digits of pi: 3.1415926536'

[71]:
```
# 4. Print exp(10.0) using three decimal digits and exponential notation
from math import exp
f'exp(10.0) = {exp(10.0):.3e}'
```

[71]: 'exp(10.0) = 2.203e+04'

Solution for exercise 3

List operators

[72]:
```
# Create lists
a = [1, 2, 3]
b = ['a', 'b', 'c']
```

[73]:
```
# 1. Adding two lists concatenates the second list to the first
# and returns a new list object
a + b
```

[73]: [1, 2, 3, 'a', 'b', 'c']

[74]:
```
# 2. multiplication of list and integer duplicates the list!
# (as opposed to multiplying each element by 2)
a * 2
```

[74]:
```
[1, 2, 3, 1, 2, 3]
```

[75]:
```
# 3. Extending a list in place using +=
# This does not return a new list but instead operates directly on b.
b += ['x', 'y', 'z']
b
```

[75]:
```
['a', 'b', 'c', 'x', 'y', 'z']
```

This is the same as using the `extend()` list method:

[76]:
```
# Recreate original list b
b = ['a', 'b', 'c']
b.extend(['x', 'y', 'z'])
b
```

[76]:
```
['a', 'b', 'c', 'x', 'y', 'z']
```

[77]:
```
# 4. Append the integer 10. Note that we cannot directly append
# the integer as such, this produces an error:
b += 10
```

```
TypeError: 'int' object is not iterable
```

Instead, we have to embed the integer in a list if we want to use `+=`, or alternatively, we can use the `append()` method.

[78]:
```
# Append single integer, wrap it in a list first
b += [10]

 # Alternatively, use append()
 # b.append(10)
```

[79]:
```
# 5. Replicating list in place using *=
a *= 2
a
```

[79]:
```
[1, 2, 3, 1, 2, 3]
```

Tuple operators

[80]:
```
# Create tuples
a = 1, 2, 3
b = 'a', 'b', 'c'
```

[81]:
```
# 1. Adding two tuples concatenates the second tuple to the first
# and returns a new tuple object
a + b
```

[81]:
```
(1, 2, 3, 'a', 'b', 'c')
```

[82]:
```
# 2. multiplication of tuple and integer replicates the tuple!
a * 2
```

[82]: (1, 2, 3, 1, 2, 3)

[83]:
```python
# 3. Extending tuple in place
b += ('x', 'y', 'z')
b
```

[83]: ('a', 'b', 'c', 'x', 'y', 'z')

It might be surprising that this works since a `tuple` is an immutable collection. However, what happens is that the original `tuple` is discarded and the reference a now points to a newly created `tuple`.

When appending a single item to a `tuple`, we need to embed it in a `tuple` just as we did for the list earlier.

[84]:
```python
# Append integer 10 to tuple
b += (5, )
```

Similarly, if we replicate a `tuple` with `*=` "in place" this actually returns a new tuple:

[85]:
```python
# 5. Replicate tuple in place using *=
a *= 2
a
```

[85]: (1, 2, 3, 1, 2, 3)

Tuple and list operators

We cannot mix tuples and lists as operands!

[86]:
```python
# Create list
a = [1, 2, 3]

# Create tuple
b = 'a', 'b', 'c'

# Cannot concatenate list and tuple!
a + b
```

```
TypeError: can only concatenate list (not "tuple") to list
```