

# Lecture 1: Language and NumPy basics

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn  
*NHH Norwegian School of Economics*

January 20, 2026

See GitHub repository for notebooks and data:

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

## Contents

<b>1</b>	<b>Language and NumPy basics</b>	<b>1</b>
1.1	Basic syntax	1
1.2	Built-in data types	3
1.3	NumPy arrays	10
1.4	Optional exercises	15
1.5	Solutions	17

## 1 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

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

```
[4]: 1
```

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

Hello world!

```
[5]: 1
```

### 1.1.1 Variable names

Python imposes some restrictions on variable names. The following summarizes what is permitted or not permitted when defining variables:

#### Valid variable names

- Must start with a letter (A–Z, a–z) or underscore `_` (e.g., `a`, `_count`, `arr2`)
- Remaining characters can be letters, digits or underscores
- Are case-sensitive (`var` != `Var`)
- Cannot be a Python keyword (words with special meaning which we encounter later)

#### Invalid variable names

- Start with a digit (e.g., `1var`)
- Contain spaces or punctuation (e.g., `my var`, `my-var`, `name$`)
- Are Python keywords (e.g., `def`, `class`, `True`, `None`)
- Contain operator characters or other symbols (e.g., `a+b`, `x%`)

## 1.2 Built-in data types

Python 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`)

### 1.2.1 Numerical data types

#### 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.

The most common way to create an integer or floating-point variable is to assign a literal value (1, 3.1415, ...) to a variable name:

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

[6]: int

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

[7]: float

Since Python uses dynamic typing, a variable can change type at any point:

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

[8]: int

An alternative way to create floating-point numbers is to use scientific notation which is particularly handy for very small and very large numbers. For example, to represent the number  $5 \times 10^8$ , we would type

```
[9]: # Define a floating-point number with value 5 * 10^8
x = 5e8
x
```

[9]: 500000000.0

Scientific notation can be combined with signs (+/-) and decimal digits as well:

```
[10]: # Define a floating-point number with value -4.1 * 10^(-3)
x = -4.1e-3
x
```

[10]: -0.0041

If you intend to do computations in floating-point, it is good programming practice to specify floating-point literals using a decimal point, even if the value is an integer. It makes a difference in a few cases (especially when using NumPy arrays, or Python extensions such as Numba or Cython):

```
[11]: x = 1.0          # instead of x = 1
type(x)
```

[11]: float

On the other hand, if you explicitly want to perform integer calculations with large integer numbers, you should *not* use the floating-point representation:

```
[12]: # Define large integer, store as floating point.
# Note that Python allows _ to be used as thousands separator
1_000_000_000_000_000_000.0 + 1.0
```

[12]: 1e+18

As you can see, the result of this computation is wrong due to the limited precision of floating-point numbers. Conversely, with integers, you get what you'd expect:

```
[13]: 1_000_000_000_000_000_000 + 1
```

[13]: 1000000000000000001

## Booleans

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:

```
[14]: x = True
x = False
type(x)
```

[14]: 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.

### Your turn.

Floating-point numbers cannot represent real numbers with arbitrary precision. This can lead to surprising results:

1. Define the floating-point number  $x$  with value  $1/3$ .
2. Add  $x$  three times ( $x + x + x$ ) and print the result.
3. Add  $x$  six times and print the result.
4. Rewrite the above expression as  $(x + x + x) + (x + x + x)$  and print the result.
5. Add the floating-point numbers  $1.0$  and  $10^{-15}$  and print the result. What happens if you add  $1.0$  and  $10^{-16}$  instead?

### 1.2.2 Strings

The string (str) data type is used to store text, i.e., sequences of characters:

```
[15]: # 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:

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

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

```
[16]: '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.

```
[17]: # Approximate value of pi
pi = 3.1415

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

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

f-strings allow for a multitude of formatting instructions (see the [Format Specification Mini-Language](#) and the optional exercises at the end of this lecture). The most commonly used formatting rule is to specify the number of decimal digits to be printed:

```
[18]: # Print pi as floating point with only 2 decimal digits
f'Pi is approximately equal to {pi:.2f}'
```

```
[18]: 'Pi is approximately equal to 3.14'
```

### Your turn.

Continuing our experiments with floating-point numbers, perform the following tasks:

1. Define the floating-point number  $x$  with value  $1/3$ , and use an f-string to print it with 20 decimal digits.
2. Define the floating-point number  $x$  with value  $0.1$ , and use an f-string to print it with 20 decimal digits.

As you can see, problems not only arise if a real number has infinitely many decimal digits (like  $1/3$ ), but also if it cannot be exactly represented as a binary number (base-2).

### 1.2.3 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:

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

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

Parentheses are optional, but improve readability:

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

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

Maybe surprisingly, a tuple with a single element *cannot* be created as follows:

```
[21]: # Attempt to create a tuple with a single element
single = (1)
type(single)
```

```
[21]: int
```

As you can see, the variable `single` is in fact an integer with value 1, which happens because parentheses have the additional role of grouping expressions. To create the tuple instead, you need to explicitly add a comma:

```
[22]: # Create a tuple with a single element (note the ,)
single = (1, )
type(single)
```

```
[22]: tuple
```

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*.

```
[23]: items = ('foo', 1, 1.0)
first = items[0]      # variable first now contains 'foo'
first
```

```
[23]: 'foo'
```

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

```
[24]: second = items[1]          # second element
      second
```

[24]: 1

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

```
[25]: first, second, third = items

      # Print first element
      first
```

[25]: 'foo'

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

```
[26]: first, *rest = items

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

[26]: [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.)

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

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

## 1.2.4 Lists

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

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

[28]: ['foo', 1, 1.0]

Unlike with tuples, you can create lists with single elements without an additional comma:

```
[29]: # Create list with a single element
      single = [1]
      type(single)
```

[29]: list

Accessing list items works the same way as with tuples

```
[30]: lst[0]          # print first item
```

[30]: 'foo'

List items can be modified:

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

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

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

```
[32]: 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
```

```
[32]: ['abc', 'bar', 1, 2.0]
```

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

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

```
[33]: 4
```

### Your turn.

Perform the following tasks to practice working with lists:

1. Define a tuple containing a single value 'a'.
2. Convert the tuple to a list using the `list()` function.
3. Append the items 'b' and 'c' using the `append()` method.
4. Select the last element of the list. Determine the index of this last element using the `len()` function.

## 1.2.5 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

```
[34]: dct = {
      'institution': 'NHH',
      'course': 'Python course',
      'year': 2026
      }
      dct
```

```
[34]: {'institution': 'NHH', 'course': 'Python course', 'year': 2026}
```

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:

```
[35]: # Alternative way to define the same dictionary
      dct = dict(institution='NHH', course='Python course', year=2026)
      dct
```

```
[35]: {'institution': 'NHH', 'course': 'Python course', 'year': 2026}
```



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

```
[36]: dct['institution']
```

```
[36]: 'NHH'
```

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

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

```
[37]: {'institution': 'NHH',
      'course': 'Introduction to Python',
      'year': 2026,
      'city': 'Bergen'}
```

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

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

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

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

```
[39]: dict_values(['NHH', 'Introduction to Python', 2026, 'Bergen'])
```

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

```
[40]: 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:

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

```
[41]: 'Norway'
```

### 1.2.6 Overwriting built-ins

By now, we have encountered several built-in names present in the Python language or standard library. The most commonly used such built-ins are: `print`, `type`, `len`, `list`, `dict`, `tuple`, `int`, `float`, `str`, `bool`. Python does not stop you from defining your own variables with exactly these names, but this use is highly problematic as you then lose the ability to access the built-in variable.

To demonstrate, we create a dictionary and assign it to the variable `dict`. Once we have done so, we can no longer call the built-in `dict()` function:

```
[42]: dict = dict(a=1, b=2)  # dict is now a dictionary, not the built-in dict type
```

If we then attempt to use `dict()` in the same way, this is no longer possible:

```
[43]: d = dict(a=1, b=2)      # This triggers an error, dict is no longer the built-in
                               ↪ dictionary type
```

```
TypeError: 'dict' object is not callable
```

At this point, it is easiest to just restart the Jupyter kernel (press Restart in the toolbar) to restore the original version.

## 1.3 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:

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

### 1.3.1 Creating arrays

#### Creating arrays from other Python objects

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

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

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

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

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

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

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

#### Array creation routines

Additionally, NumPy offers a multitude of functions to create new arrays from scratch. The most important are:

- `np.zeros()` creates an array of a given shape and initializes it to zeros.
- `np.ones()` creates an array of a given shape and initializes it to ones.

- `np.arange(start, stop, step)` creates an array with evenly spaced elements over the range  $[start, stop)$ .
  - start and step can be omitted and then default to `start=0` and `step=1`.
  - Note that the number `stop` is never included in the resulting array!
- `np.linspace(start, stop, num)` returns a vector of `num` elements which are evenly spaced over the interval  $[start, stop]$ .

There are many more array creation functions for more exotic use cases, see the NumPy [documentation](#) for details.

*Examples:*

We create arrays of zeros or ones as follows:

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

```
[48]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

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

```
[49]: array([1., 1., 1., 1., 1.])
```

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

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

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

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

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

```
[51]: array([2, 4, 6, 8])
```

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

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

```
[52]: 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)`.

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

```
[53]: array([[1., 1.],
           [1., 1.]])
```

### Your turn.

Lists and NumPy arrays behave differently in potentially unexpected ways. Perform the following tasks and inspect the result for both list and array arguments.

1. Create two variables, a list and a NumPy array, both containing the elements [1, 2, 3].
2. Multiply the list and the array by 2.
3. Add [4] to both the list and the array.
4. Add 4 to both the list and the array. Does this work?

### 1.3.2 Reshaping arrays

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

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

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

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

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

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

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

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

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

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

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

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

```
[58]: 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)
```

### 1.3.3 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.

```
[59]: import numpy as np

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

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

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

```
[60]: np.int64(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:

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

```
[61]: np.int64(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.

```
[62]: 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
```

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

You can reverse the order like this:

```
[63]: vec[::-1]
```

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

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

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

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

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

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

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

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

```
[66]: 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 :

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

```
[67]: 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.

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

```
[68]: (1, 3)
```

### Your turn.

Perform the following tasks to practice working with NumPy arrays:

1. Create a NumPy array containing the sequence from 5 to 100 (inclusive).
2. Select and print every 5<sup>th</sup> element from the array.
3. Select the last element in two different ways without hardcoding the index.  
*Hint:* The function `len()` also works for NumPy arrays.

## 1.3.4 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:

```
[69]: import numpy as np

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

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

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

```
[70]: 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:

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

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

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

The takeaway 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!

## 1.4 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:

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

```
[72]: '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 that 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 } '?  

```
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?

## 1.5 Solutions

### Solution for exercise 1

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

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

```
[73]: 'HelloWorld'
```

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

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

```
[74]: 'Hello World'
```

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

```
[75]: 'NHHNHH'
```

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

```
[76]: 'Hello World'
```

### Solution for exercise 2

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

```
[77]: 'The current version of Python is 3.1'
```

```
[78]: # 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}'
```

```
[78]: '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:

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

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

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

```
[80]: 'The first 10 digits of pi: 3.1415926536'
```

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

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

### Solution for exercise 3

#### List operators

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

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

```
[83]: [1, 2, 3, 'a', 'b', 'c']
```

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

```
[84]: [1, 2, 3, 1, 2, 3]
```

```
[85]: # 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
```

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

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

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

```
b
```

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

```
[87]: # 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.

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

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

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

```
[89]: [1, 2, 3, 1, 2, 3]
```

## Tuple operators

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

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

```
[91]: (1, 2, 3, 'a', 'b', 'c')
```

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

```
[92]: (1, 2, 3, 1, 2, 3)
```

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

```
[93]: ('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.

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

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

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

```
[95]: (1, 2, 3, 1, 2, 3)
```

### Tuple and list operators

We cannot mix tuples and lists as operands!

```
[96]: # 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
```