Unit 2: Language and NumPy basics

Richard Foltyn

December 18, 2020

Contents

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.
- 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 that are run through the interpreter, though.

Calling `print()` is still 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
```

1.2 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`)

1.2.1 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 tutorial). Floating-point is the standard way to represent real numbers on computers since these cannot store real numbers with arbitrary precision.

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

[6]:
```
int
```

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

[7]:
```
float
```

[8]:
```
# 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, Numba or Cython):

[9]:
```
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]:
```
x = True
x = False
```

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.

1.2.2 Strings

The string data type stores sequences of characters:

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

1.2.3 Tuples

Tuples represent a collection of several items which can have different data types. They are created whenever several items are separated by commas. The parenthesis are optional:

```
(item1, item2, ...)
```

[12]:
```
# A tuple containing a string, an integer and a float
items = ('foo', 1, 1.0)
items
```

[12]:
```
('foo', 1, 1.0)
```

The parenthesis are optional, but improve readability:

[13]:
```
items = 'foo', 1, 1.0        # equivalent way to create a tuple
items
```

[13]:
```
('foo', 1, 1.0)
```

We use brackets `[]` to access an element in a tuple (or any other container object)

[14]:
```
first = items[0]             # first contains 'foo'
first
```

[14]:
```
'foo'
```

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

[15]:
```
second = items[1]           # second element
second
```

[15]:
```
1
```

Tuples are immutable, which means that the items stored in the tuple cannot be changed!

[16]:
```
# 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. We create lists using brackets,

```
[item1, item2, ...]
```

[17]:
```
# Create list contains a string, an integer and a float
lst = ['foo', 1, 1.0]
lst
```

[17]:
```
['foo', 1, 1.0]
```

Accessing list items works the same as with tuples

[18]:
```
lst[0]                  # print first item
```

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

Lists items can be modified:

[19]:
```
lst[0] = 'bar'          # first element is now 'bar'
lst
```

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

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

[20]:
```
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
```

[20]:
```
['abc', 'bar', 1, 2.0]
```

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

[21]:
```
len(lst)
```

[21]:
```
4
```

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 usually we use strings as keys.
- Dictionaries are created using curly braces: `{key1: value1, key2: value2, ...}`.
- Dictionaries are unordered, i.e. elements stored in dictionaries cannot be retrieved in any guaranteed order.

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

[22]:
```
dct = {'institution': 'University of Glasgow',
       'course': 'Python bootcamp',
       'year': 2020}
dct
```

[22]:
```
{'institution': 'University of Glasgow',
 'course': 'Python bootcamp',
 'year': 2020}
```

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

[23]:
```
dct['institution']
```

[23]:
```
'University of Glasgow'
```

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

[24]:
```
dct['course'] = 'Introduction to Python'        # modify value of existing key
dct['city'] = 'Glasgow'                         # add new key-value pair
dct
```

[24]:
```
{'institution': 'University of Glasgow',
 'course': 'Introduction to Python',
 'year': 2020,
 'city': 'Glasgow'}
```

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

[25]:
```
dct.keys()
```

[25]:
```
dict_keys(['institution', 'course', 'year', 'city'])
```

[26]:
```
dct.values()
```

[26]:
```
dict_values(['University of Glasgow', 'Introduction to Python', 2020,
'Glasgow'])
```

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

[27]:
```
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:

[28]:
```
dct.get('country', 'Scotland')      # return 'Scotland' if 'country' is
                                    # not a valid key
```

[28]:
```
'Scotland'
```

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:

[29]:
```
import numpy as np
```

1.3.1 Creating arrays

NumPy offers a multitude of functions to create arrays.

[30]:
```
# Create a 1-dimensional array with 10 elements, initialise values to 0
# We need to prefix the NumPy function zeros() with 'np'
arr = np.zeros(10)
arr
```

[30]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

[31]:
```
arr1 = np.ones(5) # vector of five ones
arr1
```

[31]:
```
array([1., 1., 1., 1., 1.])
```

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

[32]:
```
arr2 = np.arange(5)      # vector [0,1,2,3,4]
arr2
```

[32]:
```
array([0, 1, 2, 3, 4])
```

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

[33]:
```
start = 2
end = 10
step = 2
arr3 = np.arange(start, end, step)
arr3
```

[33]:
```
array([2, 4, 6, 8])
```

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

[34]:
```
# Create 11 elements, equally spaced on the interval [0.0, 1.0]
arr5 = np.linspace(0.0, 1.0, 11)
arr5
```

[34]:
```
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)`.

[35]:
```
mat = np.ones((2,2))     # Create 2x2 matrix of ones
mat
```

[35]:
```
array([[1., 1.],
       [1., 1.]])
```

Creating arrays from other Python objects

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

[36]:
```
# Create array from list [1,2,3]
arr = np.array([1,2,3])
arr
```

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

[37]:
```
# Create array from tuple
arr = np.array((1.0,2.0,3.0))
arr
```

[37]:
```
array([1., 2., 3.])
```

[38]:
```
# Create two-dimensional array from nested list
arr = np.array([[1,2,3],[4,5,6]])
arr
```

[38]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

1.3.2 Reshaping arrays

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

[39]:
```
mat = np.arange(4).reshape((2,2))
mat
```

[39]:
```
array([[0, 1],
       [2, 3]])
```

[40]:
```
# reshape back to vector of 4 elements
vec = mat.reshape(4)
vec
```

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

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

[41]:
```
# with 2 dimensions, second dimension must have size 2
mat = np.arange(4).reshape((2,-1))
mat
```

[41]:
```
array([[0, 1],
       [2, 3]])
```

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

[42]:
```
mat.flatten()
```

[42]:
```
array([0, 1, 2, 3])
```

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

[43]:
```
import numpy as np
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, 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.

[44]:
```
import numpy as np

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

[44]:
```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

[45]:
```
mat[0,1]    # returns element in row 1, column 2
```

[45]:
```
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:

[46]:
```
mat[0][1] == mat[0,1]       # don't do this!
```

[46]:
```
True
```

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:

[47]:
```
# Create a 3x2 matrix
mat = np.arange(6).reshape((2,3))
mat
```

[47]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[48]:
```
# Retrieve only the first and third columns:
mat[0:2,0:3:2]
```

[48]:
```
array([[0, 2],
       [3, 5]])
```

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 backwards: `vec[:-1]` retrieves the whole vector except for the last element.
  - If `step` is negative, the order of elements is reversed.

[49]:
```
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
```

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

You can reverse the order like this:

[50]:
```
vec[::-1]
```

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

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

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

[51]:
```
mat[1]      # includes all columns of row 2; same as mat[1,:]
```

[51]:
```
array([3, 4, 5])
```

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

[52]:
```
mat[:, 1]   # includes all rows of column 2
```

[52]:
```
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.

[53]:
```
# Apply start:stop:step indexing to tuple
tpl = (1,2,3)
tpl[:3:2]
```

[53]:
```
(1, 3)
```

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:

[54]:
```
import numpy as np

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

[54]:
```
dtype('float64')
```

[55]:
```
# Integer arguments return array of type np.int64
arr = np.arange(1,5,1)
arr.dtype
```

[55]:
```
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:

[56]:
```
# Create integer array
arr = np.ones(5, dtype=np.int64)
# Store floating-point in second element
arr[1] = 1.234
arr
```

[56]:
```
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 always 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 Exercises

1.4.1 Exercise 1: string operations

Experiment with operators applied to strings and integers.

1. Define two string variables and concatenate them using `+`
2. Define a string variable and multiply it by 2 using `*`. What happens?
3. Define two strings and compare whether they are equal using the `==` and `!=` relational operators.
4. Define a string. Use the operators `in` and `not in` to test whether a character is contained in the string.
5. Define two string variables and assign them the same value. Use the `is` operator to test whether these are identical objects.
6. Define a string variable and use the `+=` assignment operator to append another string.

   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
   ```

1.4.2 Exercise 2: string formatting

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:

```
s = 'The current version of Python is {}'
```

The token `{}` will be replaced with data converted to a string when we apply the `format()` method:

```
s = 'The current version of Python is {}'.format(3.9)
```

The string `s` now contains the value

```
'The current version of Python is 3.9'
```

What if we want to format the float `3.9` 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 numbers magnitude.

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

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

1. Define two strings and concatenate them using the `format()` function. Add a space between them.
2. Use the above example format string, but truncate the Python version to its major version number. Do you get the expected result?
3. Print π using a precision of 10 decimal digits. *Hint:* the value of π is available as `from math import pi`
4. Print `exp(10.0)` using exponential notation and three decimal digits. *Hint:* To use the exponential function, you need to import it usding `from math import exp`

1.4.3 Exercise 3: string formatting with f-strings

Since Python 3.6 there is an additional, more convenient way to format strings, the so-called *formatted string literals* or *f-strings* (official documentation). Instead of calling the `format()` method as in the previous example, one can instead define string which contain expressions that will be evaluated at runtime.

The simplest example is to print the value of a variable using default formatting:

[57]:
```
name = 'Python'
s = f'{name} programming is fun!'
print(s)
```

```
Python programming is fun!
```

Note that the string needs to be prefixed by an `f` to indicate that it contains expressions which need to be evaluated. These expressions are again wrapped in braces. Within braces, a syntax similar to the one shown in the previous exercise can be used to specify detailed formatting instructions. For example, you can specify the number of decimal digits as follows:

[58]:
```
value = 1.2345
s = f'Value with 2 significant digits: {value:.2f}'
print(s)
```

```
Value with 2 significant digits: 1.23
```

Note that in the above examples, the variables `name` and `value` need to be known in the scope where the f-string is being defined. Otherwise, you'll get the following error:

[59]:
```
# cannot create f-string using unknown names
s = f'{unknown} is not defined'
```

```
NameError: name 'unknown' is not defined
```

Now that you have seen the basic usage, repeat Exercise 2 using f-strings instead of the `format()` method!

1.4.4 Exercise 4: operations on tuples and lists

Perform the following tasks and examine their results:

1. Create two lists and add them using +
2. Multiply a list by the integer 2
3. Create a list `list1` and inspect the result of `list1 += ['x', 'y', 'z']`
4. Create a list `list1` and inspect the result of `list1 *= 2`

Repeat steps 1-4 using tuples instead of lists.

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

1.5 Solutions

1.5.1 Solution for exercise 1

[60]:
```
# 1. string concatenation using addition
str1 = 'abc'
str2 = 'xyz'

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

[60]:
```
'abcxyz'
```

[61]:
```
# 2. string multiplication by integers
str1 = 'abc'
# Repeat string using multiplication!
str1 * 2
```

[61]:
```
'abcabc'
```

[62]:
```
# 3. Test for string equality
str1 = 'abc'
str2 = 'xyz'
str1 == str2
```

[62]: False

[63]:
```
# 3. Test for string inequality
str1 = 'abc'
str2 = 'xyz'
str1 != str2
```

[63]: True

[64]:
```
# 4. Test whether individual character is included in string
str1 = 'abc'
'b' in str1
```

[64]: True

[65]:
```
# 4. Test whether individual character is NOT included in string
str1 = 'abc'
'x' not in str1
```

[65]: True

The last two examples illustrate that in Python strings will be interpreted as collections (of characters), just like lists or tuples, if the context requires it. We can therefore apply the `in` operator to test for membership.

[66]:
```
# 5. Test for identity
str1 = 'abc'
str2 = 'abc'
str1 is str2
```

[66]: True

This result should be surprising and is somewhat specific to Python. We would not expect two objects that were created completely independently from each other to be *identical*, ie point to the same memory. Python, however, caches string literals for reasons of efficiency, so it actually does keep only *one* copy of `'abc'` around, irrespective of how many variables containing `abc` are created.

[67]:
```
# 6. Append using +=
str1 = 'abc'
str1 += 'xyx'        # Append 'xyz' to value in str1, assign result to str1
str1
```

[67]: 'abcxyx'

1.5.2 Solution for exercise 2

[68]:
```
# 1. String concatenation
str1 = 'abc'
str2 = 'xyz'

# format specification to concatenate two string with a space inbetween
fmt = '{} {}'
fmt.format(str1, str2)
```

[68]: 'abc xyz'

[69]:
```
# 2. Truncate Python version to major version number
# To do this, we specify 0 decimal digits!
fmt = 'The current major version of Python is {:.0f}'
fmt.format(3.9)      # Now this didn't work out as intended :)
```

[69]:
```
'The current major version of Python is 4'
```

This does not work as intended because formatting with zero decimal digits rounds the floating-point number, instead of just truncating the decimal part. We could instead convert the `float` to `int` before applying formatting, since the function `int()` will truncate the fractional part:

[70]:
```
version = 3.9
fmt = 'The current major version of Python is {:d}'
print(fmt.format(int(version)))
```

```
The current major version of Python is 3
```

[71]:
```
# 3. Print pi using 10 decimal digits
from math import pi
fmt = 'The first 10 digits of pi: {:.10f}'
fmt.format(pi)
```

[71]:
```
'The first 10 digits of pi: 3.1415926536'
```

[72]:
```
# 4. Print exp(10.0) using three decimal digits and exponential notation
from math import exp
fmt = 'exp(10.0) = {:.3e}'
fmt.format(exp(10.0))
```

[72]:
```
'exp(10.0) = 2.203e+04'
```

1.5.3 Solution for exercise 3

We now repeat exercise 2 using f-strings instead of the `format()` method.

[73]:
```
# 1. String concatenation
str1 = 'abc'
str2 = 'xyz'

# format specification to concatenate two string with a space inbetween
s = f'{str1} {str1}'
print(s)
```

```
abc abc
```

[74]:
```
# 2. Truncate Python version to major version number
# To do this, we specify 0 decimal digits!
version = 3.9
s = f'The current major version of Python is {version:.0f}'
print(s)        # does not work as intended!
```

```
The current major version of Python is 4
```

This does not work as intended because formatting with zero decimal digits rounds the floating-point number, instead of just truncating the decimal part. See the previous exercise for one possible solution.

[75]:
```
# 3. Print pi using 10 decimal digits
from math import pi
s = f'The first 10 digits of pi: {pi:.10f}'
print(s)
```

```
The first 10 digits of pi: 3.1415926536
```

[76]:
```
# 4. Print exp(10.0) using three decimal digits and exponential notation
from math import exp
s = f'exp(10.0) = {exp(10.0):.3e}'
print(s)
```

```
exp(10.0) = 2.203e+04
```

The last example illustrates that f-string expressions can also be function calls, not just variable names!

1.5.4 Solution for exercise 4

List operators

[77]:
```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

# 1. Adding two lists concatenates the second list to the first
# and returns a new list object
list1 + list2
```

[77]:
```
[1, 2, 3, 'a', 'b', 'c']
```

[78]:
```
# 2. multiplication of list and integer replicates the list!
list1 * 2
```

[78]:
```
[1, 2, 3, 1, 2, 3]
```

[79]:
```
# 3. Extending list in place using +=
list1 += ['x', 'y', 'z']
list1
```

[79]:
```
[1, 2, 3, 'x', 'y', 'z']
```

Note that we cannot append an element to the list that is not a list:

[80]:
```
list1 += 10
```

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

Instead, we need to wrap a singular element to create a list like this:

[81]:
```
list1 += [10]
list1
```

[81]:
```
[1, 2, 3, 'x', 'y', 'z', 10]
```

[82]:
```
# 4. Replicating list in place using *=
list1 *= 2
list1
```

[82]:
```
[1, 2, 3, 'x', 'y', 'z', 10, 1, 2, 3, 'x', 'y', 'z', 10]
```

Tuple operators

[83]:
```
tpl1 = 1, 2, 3
tpl2 = 'a', 'b', 'c'

# 1. Adding two tuples concatenates the second tuple to the first
# and returns a new tuple object
tpl1 + tpl2
```

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

[84]:
```
# 2. multiplication of tuple and integer replicates the tuple!
tpl1 * 2
```

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

[85]:
```
# 3. Extending tuple in place
tpl1 += ('x', 'y', 'z')
tpl1
```

[85]:
```
(1, 2, 3, 'x', 'y', 'z')
```

It might be surprising that this works since a `tuple` is an immutable object. However, what happens is that the original `tuple` is discarded and the reference `tpl1` now points to a newly created `tuple`.

The same happens when we replicate a `tuple` with `*=`:

[86]:
```
# 4. Replicate tuple in place using *=
tpl1 *= 2
tpl1
```

[86]:
```
(1, 2, 3, 'x', 'y', 'z', 1, 2, 3, 'x', 'y', 'z')
```

Tuple and list operators

We cannot mix tuples and lists as operands!

[87]:
```
lst = [1, 2, 3]
tpl = 'a', 'b', 'c'

# Cannot concatenate list and tuple!
lst + tpl
```

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