04. Lists and Tuples

October 1, 2025

1 4. Lists and Tuples

1.1 4.1. Lists

A list is a data structure used to store a collection of items. List elements are objects of any type (think of numbers, strings, booleans, or even other lists). Items are usually homogeneous (i.e. they have the same behaviour) so that the same operation can pe applied to all of them, but this is not a requirement of the type. Lists are one of the most commonly used data structures in Python due to their flexibility and ease of use.

1.1.1 4.1.1. Creating lists

The empty list Using a pair of square brackets to denote the empty list:

[1]:
```
my_list = []
type(my_list)
```

[1]:
```
list
```

Also, calling the `list` constructor without parameters will create an empty list:

[2]:
```
my_other_list = list()
my_list == my_other_list
```

[2]:
```
True
```

Enumerating elements between square brackets The most common way to create a list is by enclosing comma-separated elements within square brackets.

[3]:
```
my_list = [6, 2, 8, 1]
```

[4]:
```
my_list
```

[4]:
```
[6, 2, 8, 1]
```

[5]:
```
my_list = ["hello"]  # one-element list
my_list
```

[5]:
```
['hello']
```

Creating lists from other objects You can create a list by passing an iterable (such as a string, tuple, or another list) to the `list()` constructor.

```
[6]: my_list = list("hello")
     my_list
```

```
[6]: ['h', 'e', 'l', 'l', 'o']
```

```
[7]: my_list = list(range(10))
     my_list
```

```
[7]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1.1.2 Exercises 4.1.1

1. Create a list containing elements of different types (`int`, `float`, `bool`, `str`). Print the list and its type.
2. Create a list containing every second integer between 100 and 150.

1.1.3 4.1.2. Manipulating lists

Lists are sequences, i.e. ordered collections of items or elements. Strings are also sequences, so we'll see here some common operations which have a similar behavior for lists.

Membership testing (`in` and `not in` operators) `in` and `not in` operators are used to check whether a value is a member of the list or not.

```
[8]: words = ["good", "morning", "people"]
     "good" in words
```

```
[8]: True
```

```
[9]: "hello" not in words
```

```
[9]: True
```

Concatenation (`+` and `+=` operators) We can add two lists together using `+`. The result will be a new list with all elements from both lists:

```
[10]: first_list = [4, 7, 8]
      second_list = [2, 4]
      third_list = first_list + second_list
      print(third_list)
```

```
[4, 7, 8, 2, 4]
```

`+=` operator is the in-place operator we've already used for numbers or strings. It updates the list on the left with items from the list on the right:

[11]:
```
first_list += [1, 2, 3]
first_list
```

[11]:
```
[4, 7, 8, 1, 2, 3]
```

Multiplication (`*` and `*=` operators) Similar to its behaviour on strings, `*` does the concatenation of the list to itself multiple times:

[1]:
```
[True] * 3
```

[1]:
```
[True, True, True]
```

[13]:
```
names = ["Anna", "Mike"]
names *= 4
names
```

[13]:
```
['Anna', 'Mike', 'Anna', 'Mike', 'Anna', 'Mike', 'Anna', 'Mike']
```

1.1.4 Exercises 4.1.2.1

1. Given the following list:

   ```
   greetings = ["hello", "hi", "good morning", "good afternoon"]
   ```

   check, for each of the following objects, if they are in the list: `"hi"`, `"good"`, `["hello", "hi"]`

2. Add `"good evening"` and `"hey there"` (using in place list concatenation) to `greetings` and print the new value of the list.

3. Create a list with elements from `range(562, 873, 17)`. Is 800 in this list?

4. Multiply the list above by 3 and assign the result to a new variable. Print the new list.

Indexing and slicing (`[]` operator): For lists, the indexing operation means retrieving an item of the list from a certain index:

[14]:
```
my_list = [6, 2, 6, 7, 9, 2, 4, 3, 1]
my_list[0]
```

[14]:
```
6
```

[15]:
```
my_list[-1]
```

[15]:
```
1
```

The slicing operation is used to retrieve *slices* (lists containing only elements on certain indexes) of the initial list. The slicing notation is `list[start:end:step]`, where: * `start`: Optional. The starting index of the slice (inclusive). If not specified, the default value is 0. * `end`: Optional. The ending index of the slice (exclusive). If not specified, the default value is the length of the list. * `step`: Optional. The step size for selecting items. If not specified, the default value is 1.

[16]:
```
my_list[1:-1]  # all elements except first and last one
```

[16]:
```
[2, 6, 7, 9, 2, 4, 3]
```

[17]:
```
my_list[::2]  # every second element of the list
```

[17]:
```
[6, 6, 9, 4, 1]
```

Updating elements in a list Lists can be modified. You can assign a certain value to an existing index in order to change that item to the new value:

[18]:
```
my_list[1] = 1000
my_list
```

[18]:
```
[6, 1000, 6, 7, 9, 2, 4, 3, 1]
```

You can also change multiple elements at a time by changing an entire slice to a new list:

[19]:
```
my_list[2:7:2]
```

[19]:
```
[6, 9, 4]
```

[20]:
```
my_list[2:7:2] = [0, 0, 0]  # when specifying the step the two lists must have␣
  ↪the same length
my_list
```

[20]:
```
[6, 1000, 0, 7, 0, 2, 0, 3, 1]
```

[21]:
```
my_list[1:6]
```

[21]:
```
[1000, 0, 7, 0, 2]
```

[22]:
```
my_list[1:6] = [5, 5]  # when the step is not specified, the two lists can have␣
  ↪different dimensions
my_list
```

[22]:
```
[6, 5, 5, 0, 3, 1]
```

Deleting elements from a list Elements can also be deleted by index, using the keyword `del`:

[23]:
```
del my_list[3]
my_list
```

[23]:
```
[6, 5, 5, 3, 1]
```

[24]:
```
del my_list[1:3]
my_list
```

[24]:
```
[6, 3, 1]
```

1.1.5 Exercises 4.1.2.2

1. Create a list with at least 5 elements.
    1. Print the entire list.
    2. Print the reversed list, using slicing.
    3. Print the first element of the list using indexing.
    4. Print a slice of the list containing elements from index 2 to index 4.
    5. Modify the second element of the list using index assignment. Print the modified list.
    6. Modify a slice of the list to replace elements from index 1 to index 3 with new values. Print the modified list.
    7. Delete the last element of the list. Print the final modified list.

Using built-in functions on lists Python provides several built-in functions that work with lists to perform common operations. These functions are useful for manipulating and analyzing lists efficiently. Here are some of the built-in functions that work with lists:

- `len()`: Returns the number of elements in a list.

[25]:
```
my_list = [4, 1, 6, 3, 2]
len(my_list)
```

[25]:
```
5
```

- `min()`: Returns the minimum value in a list.

[26]:
```
min(my_list)
```

[26]:
```
1
```

- `max()`: Returns the maximum value in a list.

[27]:
```
max(my_list)
```

[27]:
```
6
```

- `sum()`: Returns the sum of all elements in a list.

[28]:
```
sum(my_list)
```

[28]:
```
16
```

1.1.6 Exercises 4.1.2.3

1. Given the following list:

    ```
    numbers = list(range(200, 150, -8))
    ```

    print the length, minimum value, maximum value and sum of all elements.

2. The built-in `sum` function can also be called on the result of the `range` function. Try it out!

Iterating on a list Lists are iterable, which means they can be used in conjunction with the `for` statement. Iterating on a list allows you to access each element in the list sequentially and perform operations on them.

[29]:
```python
words = ["sum", "of", "all", "elements"]
for word in words:
    print(word)
```

```
sum
of
all
elements
```

[30]:
```python
numbers = [78, 79, 32, 41, 23, 64]
for nr in numbers:
    if nr % 2 == 0:
        print(nr)
```

```
78
32
64
```

1.1.7 Exercises 4.1.2.4

1. Iterate on the `greetings` list created above and print, for each item, the length of the string and a message saying if the string contains `"good"` or not, e.g.

   ```python
   greetings = ["hello", "hi", "good morning", "good afternoon"]
   ```

   > "hello" has a length of 5 characters. "hello" does not contain "good". [...] "good morning" has a length of 12 characters. "good morning" contains "good".

2. Given the following list, print all elements that have two or more elements: `python list_of_lists = [[10, 20], [40], [30, 56, 25], [23, 14], [33], [], [40, 22, 47, 39]]`

1.2 4.2 Tuples

A tuple is a data structure used to store a collection of items. Tuple elements are objects of any type (numbers, strings, booleans, even other tuples or lists). Tuples are very similar to lists, but they are not modifiable. We will see in the tuple operations below, they only support reading operations, and no update/delete operations.

1.2.1 4.2.1. Creating tuples

The empty tuple Using a pair of parentheses to denote the empty tuple:

[31]:
```python
my_tuple = ()
type(my_tuple)
```

[31]:
```
tuple
```

Just like we did for lists, calling the `tuple` constructor without parameters will create an empty tuple:

```
[32]: my_other_tuple = tuple()
      my_other_tuple == my_tuple
```

```
[32]: True
```

The one element tuple In order to create a tuple with a single element, place the element inside the parentheses with a comma after it:

```
[33]: one_element_tuple = (2,)
      type(one_element_tuple)
```

```
[33]: tuple
```

Missing the trailing comma will lead the interpreter to believe that the parentheses are there to group the expression:

```
[34]: not_a_tuple = (2)
      type(not_a_tuple)
```

```
[34]: int
```

Enumerating elements between parentheses For tuples with multiples elements, you simply enclose comma-separated values between parentheses:

```
[35]: my_tuple = (5, 2, 3, 1, 5, 3)
      my_tuple
```

```
[35]: (5, 2, 3, 1, 5, 3)
```

Parentheses can be skipped Tuples can be created even without parentheses. This is often referred to as tuple packing or implicit tuple creation. Tuple packing occurs when multiple values are separated by commas, and Python automatically creates a tuple containing those values.

```
[36]: my_tuple = 1, 2, 3
      my_tuple
```

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

```
[37]: my_tuple = 1,
      my_tuple
```

```
[37]: (1,)
```

While tuple packing without parentheses can be convenient in some cases, using parentheses can improve readability and make the code more explicit.

Tuple unpacking Tuple unpacking in Python allows you to assign the elements of a tuple to individual variables. It's a convenient way to extract values from tuples and work with them separately.

[38]:
```
my_tuple = (1, 2, 3)
x, y, z = my_tuple
print(x)
print(y)
print(z)
```

```
1
2
3
```

Tuple unpacking provides a concise and readable way to work with tuples and is a common idiom in Python programming. A common use case for tuple unpacking is variable interchange:

[39]:
```
a = 1
b = 2
print("Before interchange:", a, b)
a, b = b, a
print("After interchange:", a, b)
```

```
Before interchange: 1 2
After interchange: 2 1
```

Tuple unpacking can also be used while iterating. For example, iterating on a list containing tuples of fixed size can be done by using a single variable:

[40]:
```
list_of_tuples = [("hello", 3), ("hi", 6), ("bye", 2)]
for tup in list_of_tuples:
    print(tup)
```

```
('hello', 3)
('hi', 6)
('bye', 2)
```

Or it can be done by unpacking the inner tuples into separate variables, allowing more flexibility in working with them inside the `for` block:

[41]:
```
for word, nr in list_of_tuples:
    print(word)
    print(nr, end="\n\n")
```

```
hello
3

hi
6

bye
```

```
2
```

Creating tuples from other objects You can create a tuple by passing an iterable (such as a string, list, range) to the `tuple()` constructor.

[42]:
```
tuple([1, 2, 3])
```

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

[43]:
```
tuple(range(5))
```

[43]:
```
(0, 1, 2, 3, 4)
```

[44]:
```
tuple("hello")
```

[44]:
```
('h', 'e', 'l', 'l', 'o')
```

1.2.2 Exercises 4.2.1

1. Given the following variables:

   ```
   name = "Jane"
   age = 20
   is_student = True
   ```

   create a `person` tuple with their values. Print the tuple and its type.

2. Given the following data structure: `python friends = [ ("Jane", 20, ["reading", "hiking", "biking"]), ("Mike", 17, ["hiking", "fishing"]), ("Anna", 25, []), ("Sam", 40, ["playing guitar"]), ("Dan", 34, ["painting", "reading"]), ]` containing name, age and hobbies:

   - first, iterate on `friends` list, print every item and its type;
   - then, iterate on `friends` list using tuple unpacking for higher readability, and print each item in the tuples and their respective types;
   - iterate a third time on `friends` list, use tuple unpacking, and print the names of people who are over 18 and have at least two hobbies.

1.2.3 4.2.2 Manipulating tuples

Operations we've already seen for lists also work for tuples, except those that change the object (item and slice assignment and deletion). Let's see some examples below.

Membership testing (`in` and `not in` operators)

[45]:
```
my_tuple = ("hello", "world")
"world" in my_tuple
```

[45]:
```
True
```

```
[46]: "hello" not in my_tuple
```

```
[46]: False
```

Concatenation and multiplication

```
[47]: my_tuple + ("!", )
```

```
[47]: ('hello', 'world', '!')
```

```
[48]: my_tuple * 2
```

```
[48]: ('hello', 'world', 'hello', 'world')
```

Indexing and slicing

```
[49]: my_tuple[0]
```

```
[49]: 'hello'
```

```
[50]: my_tuple[1:]
```

```
[50]: ('world',)
```

Using built-in functions on tuples The same built-in functions presented above also work for tuples.

```
[51]: my_tuple = (4, 6, 1, 4)
      len(my_tuple)
```

```
[51]: 4
```

```
[52]: min(my_tuple)
```

```
[52]: 1
```

```
[53]: max(my_tuple)
```

```
[53]: 6
```

```
[54]: sum(my_tuple)
```

```
[54]: 15
```

Iterating on a tuple Tuples are iterable, which means they can be used in conjunction with the `for` statement.

```
[55]: my_tuple = (4, 6, 1, 4)
      for item in my_tuple:
          print(item)
```

```
4
6
1
4
```

1.2.4 Exercises 4.2.3

1. Given the following variables:

   ```
   x = 10
   first_tuple = x + 2, x + 1, x
   second_tuple = tuple(range(5))
   third_tuple = (2, 3, 4)
   ```

   create a new tuple `final_tuple` containing:

   - all elements in `first_tuple`, two times
   - elements from index 1 (inclusive) up to index 4 (exclusive) from `second_tuple`
   - element on index 1 from `third_tuple`

   Do not hardcode values, the code should still work if we change the initial tuples. Use tuple concatenation, multiplication, indexing and slicing.

2. Given the following tuple:

   ```
   movies = (
       ("Titanic", 7.9),
       ("Star Wars", 8.6),
       ("The Lord of the Rings", 9),
       ("Harry Potter", 7.6),
       ("Transformers", 7),
   )
   ```

   representing movie titles and ratings:

   - iterate on it, printing the names of the movies with a rating higher than 8 (consider these to be recommended movies);
   - write a message saying `"I have recommended x ouf of y movies"`, where `x` and `y` should be the actual counts of recommended and total movies.