`list`

`list` are the most common data containers. They are:

- **mutable**: one can change their content by adding new elements or changing existing elements
- **indexable**: one can access the content of a list using an index, starting from **0**

All the operations done on a list are done **in place**. That means no other list is returned but the current instance is modified.

Appending content

1 at a time at the end of the list

In [1]:

```
var = [34, 23]
var.append(1)
var.append(2)
var.append("b")
print(var)
```

```
[34, 23, 1, 2, 'b']
```

1 at a time by position

In [2]:
```
var.insert(3, "new")
print(var)
```

```
[34, 23, 1, 'new', 2, 'b']
```

Already stored in a list

In [3]:
```
var1 = [1, 2]
var2 = [3, 4]
var1.extend(var2)
print(var1)
```

```
[1, 2, 3, 4]
```

Modifying content

In [4]:
```
var[0] = 3
print(var)
```

```
[3, 23, 1, 'new', 2, 'b']
```

One can specify a **negative index**: -1 is the latest value of the list, -2 is the second to last, etc...

In [5]:
```
var = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(var[-1])
print(var[-3])
```

```
10
8
```

Note than you can go from negative to positive index using this simple trick:

In [6]:

```
positive_index  = 3
negative_index  = -(len(var)-positive_index)
positive_index2 =  len(var)+negative_index
print(var[positive_index])
print(var[negative_index])
print(var[positive_index2])
```

```
3
3
3
```

The `index` method returns the index (positive) of the first occurence of an element. A lower and upper index can also be specified to look for a value at a particular location.

In [7]:

```
var.append(1)
print(var)
print(f"Index of first '1' value starting from index 0: {var.index(1)}")
print(f"Index of first '1' value starting from index 0: {var.index(1, 4, len(var))}")
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
Index of first '1' value starting from index 0: 1
Index of first '1' value starting from index 0: 11
```

Deleting content

At the end of the list

In [8]:
```
last = var.pop()
print(f"Last value: {last}")
print(var)
```

```
Last value: 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Positionnal

In [9]:
```
var.remove(1)
print(var)
```

```
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sorting

With homogeneous data types

Beware that sorting the variable is done in place.

In [10]:

```
var = [1, -6.7, 189]
var.sort(reverse=True)
print(var)
```

```
[189, 1, -6.7]
```

Using a custom sorting key

A custom sorting key is a function that returns, for all elements of the list to sort, a value that can be compared to the other.

In [11]:

```
var = [56, "98", -102, "102.45"]
var.sort(key=lambda key:float(key))
print(var)
```

```
[-102, 56, '98', '102.45']
```

Advanced

Sorting is done internally by a call to methods `__lt__` and `__gt__` of an instance. One can redefine these methods to sort elements on some relevant properties.

Below is an example of a custom object who internal value (used for the sorting process) depends on the order of creation of the instance (see `COUNTER`). The `__lt__` method is redefined using this value.

In [12]:

```python
class Custom():
    COUNTER = 10
    def __init__(self):
        self.value = Custom.COUNTER
        Custom.COUNTER -= 1

    def __lt__(self, other):
        if not isinstance(other, Custom):
            raise NotImplementedError()
        return self.value < other.value

    def __repr__(self):
        return f"Custom ({self.value})"

var1 = Custom()
var2 = Custom()
var3 = Custom()
var = [var1, var2, var3]
print(var)
var.sort()
print(var)
```

```
[Custom (10), Custom (9), Custom (8)]
[Custom (8), Custom (9), Custom (10)]
```

Another way to sort data containers is the `sorted` function. Differently from method `.sort()`, it returns a new sorted list.

In [13]:
```
var = (3, 7, 2, 9, -4)
sorted(var)
```

Out[13]:
```
[-4, 2, 3, 7, 9]
```

Concatenation

Two lists can be concatenated using `+`.

In [14]:
```
var1 = [1, 2]
var2 = [3, 4]

print(var1)
print(var2)
print(var1 + var2)
```

```
[1, 2]
[3, 4]
[1, 2, 3, 4]
```

Conclusion

`list` are commonly used in Python. Yet they are neither always fitted to all use cases, nor the the most powerful solution.

`set`

`set` are data containers that can store a given value at most one time. They are not **mutable** and not **indexable**. Thus, **there is no guarantee for the insertion order to be preserved.**

In [15]:

```python
var = {1, 2, 3}
var.add(4)
print(var)
var.add(3)
print(var)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
```

One can perform unions, intersections and differences of `set` instances.

In [16]:

```python
var1 = {1, 2, 3}
var2 = {2, 3, 4}
print(f"{'Union':<15}: {var1&var2}")
print(f"{'Intersection':<15}: {var1|var2}")
print(f"{'Difference 1':<15}: {var1-var2}")
print(f"{'Difference 2':<15}: {var2-var1}")
```

```
Union          : {2, 3}
Intersection   : {1, 2, 3, 4}
Difference 1   : {1}
Difference 2   : {4}
```

Above, we used operators `&`, `|`, `-`: these are shortcuts for the dedicated methods. These methods can be showed using `dir`.

In [17]:
```
attrs = dir(var1)
[attr for attr in attrs if not attr.startswith("__")]
```

Out[17]:
```
['add',
 'clear',
 'copy',
 'difference',
 'difference_update',
 'discard',
 'intersection',
 'intersection_update',
 'isdisjoint',
 'issubset',
 'issuperset',
 'pop',
 'remove',
 'symmetric_difference',
 'symmetric_difference_update',
 'union',
 'update']
```

`tuple`

`tuple` are similar to `list`, yet they are **immutable** (not **mutable**). Whenever it's possible `tuple` must be prefered over `list`.

In [18]:
```
var = (1, 2, 3)
print(var)
```

```
(1, 2, 3)
```

Reminder, `tuple`s are immutable:

In [19]:
```
var[2] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[19], line 1
----> 1 var[2] = 5

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

Important: what defines a `tuple` is not the parenthesis `(` ... `)` but **the comma** `,` .

In [20]:

```python
var = (1)
print(type(var))
var = 1,
print(type(var))
var = 1, 2, 3
print(type(var))
```

```
<class 'int'>
<class 'tuple'>
<class 'tuple'>
```

`dict`

Dictionaries makes it possile to store a **value** with a unique **key** as identifier.

In [21]:

```
var = {"one": 1, "two": 2, "three": 3}
var
```

Out[21]:

```
{'one': 1, 'two': 2, 'three': 3}
```

Keys and values are retrieved using dedicated methods. These methods return something similar to a list (but different):

In [22]:

```
keys = var.keys()
print(keys)
values = var.values()
print(values)
```

```
dict_keys(['one', 'two', 'three'])
dict_values([1, 2, 3])
```

A call to `items` extract couples of (key, value).

In [23]:

```
items = var.items()
print(items)
```

```
dict_items([('one', 1), ('two', 2), ('three', 3)])
```

One can read and modify a dictionary using any key.

In [24]:
```
var["one"] = "first"
print(var)
print(var["two"])
```

```
{'one': 'first', 'two': 2, 'three': 3}
2
```

Retriving an element using brackets `[]` is internally done by a call to the `get` method. One can call explicitely the `get` method with a default value in case the key does not exist.

In [25]:
```
print(var.get("three", "Unknown!"))
print(var.get("four", "Unknown!"))
```

```
3
Unknown!
```

Other methods are presented using `dir(dict)`.

In [26]:
```
attrs = dir(dict)
[attr for attr in attrs if not attr.startswith("__")]
```

Out[26]:
```
['clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

Recall that help is available on any Python object using `help`:

In [27]:
```
help(dict.popitem)
```

```
Help on method_descriptor:

popitem(self, /) unbound builtins.dict method
    Remove and return a (key, value) pair as a 2-tuple.

    Pairs are returned in LIFO (last-in, first-out) order.
    Raises KeyError if the dict is empty.
```

Notes

As for values of a `set`, keys of a dictionary must be **hashable**. Hence, a list cannot be a dictionary key.

In practice, most immutable objects are hashable.

In [28]:
```
var = {[1,2]: 5}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 1
----> 1 var = {[1,2]: 5}

TypeError: unhashable type: 'list'
```

`deque`

A `deque` is kind of `list` that is optimized for fast values appending at both ends (beginning and end of the container). The speed up is observed maily for very large containers.

`deque` thus have the following methods `appendleft`, `popleft` et `extendleft`.

In [29]:

```
from collections import deque
d = deque([1, 2, 3 ,4])
d.popleft()
print(d)
d.appendleft(0)
print(d)
d.extendleft([-5, 63])
print(d)
```

```
deque([2, 3, 4])
deque([0, 2, 3, 4])
deque([63, -5, 0, 2, 3, 4])
```

About **slicing**

slicing is a way to extract part of an indexable object (ex: `list`, `tuple`, `str`).

Slincing is done using brackets with a specific notation: `start:end+1:step` (last index is excluded). `step` is not mandatory (if none, it is assumed `step=1`) but if `step` is given then the two other must be given too.

Hereafter, a use case using a `list`.

In [30]:

```
var = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(var[:5])     # the first 5 values
print(var[5:])     # values from index 5 to the end
print(var[1:5])    # values from index 1 to index 4
print(var[1:5:2])  # values from index 1 to index 4, every 2 values
```

```
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9, 10]
[1, 2, 3, 4]
[1, 3]
```

Some negative indexers are also possible here, as long as `start` references an element locarted before `end`.

In [31]:
```
print(var[-3:])    # the last three values
print(var[-6:8:2])  # values from index -6 to index 8, every 2 values
```

```
[8, 9, 10]
[5, 7]
```

This small trick reverse the container:

In [32]:
```
var = var[::-1]
print(var)
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

slicing returns a **copy** of its argument, even when the argument is mutable.

In [33]:

```
var = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
var2 = var[:5]
print(f"var: {var}\t\t var2: {var2}")
var[1] = 1000                                   # modification
print(f"var: {var} \t var2: {var2}")
```

```
var: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]          var2: [0, 1, 2, 3, 4]
var: [0, 1000, 2, 3, 4, 5, 6, 7, 8, 9, 10]       var2: [0, 1, 2, 3, 4]
```

Booleans

Some values are interpreted as booleans in a **boolean context**, for instance an `if`.

Example: an empty data container has a value of `False` (`True` if it contains at least one element)

In [34]:

```
for var in ([], 0, None):
    if var:
        print(f"{var!s:<5} is interpreted as True")
    else:
        print(f"{var!s:<5} is interpreted as False")
```

```
[]    is interpreted as False
0     is interpreted as False
None  is interpreted as False
```

This is not the case outside of these contexts, unless using **type casting** toward a **bool** type.

In [35]:

```
print([] is False)          # False, since a list is not a bool, even empty
print(bool([]) is False)  # True
```

```
False
True
```