unpacking

Key idea

unpacking is a way to extract values from a data container into separate variables.

In [1]:
```
a, b, c = [1, 2, 3]
print(a)
print(b)
print(c)
```

```
1
2
3
```

If the number of variables on left side of `=` is not the number of elements of the container, an error is raised:

In [2]:
```
a, b = [1, 2, 3]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 1
----> 1 a, b = [1, 2, 3]

ValueError: too many values to unpack (expected 2)
```

One can discard specific elements using `*`.

In [3]:
```
var1, var2, var3, *unwanted, var4 = "abcdefg"
print(var1, var2, var3, var4)
```

```
a b c g
```

Yet, in this case, there must be at most one unknown variable (one `*`)

In [4]:
```
a, *unwanted, c, *unwanted, g = "abcdefg"
```

```
  Cell In[4], line 1
    a, *unwanted, c, *unwanted, g = "abcdefg"
    ^
SyntaxError: multiple starred expressions in assignment
```

Use cases

Unpacking can be used in the following situations:

- `for` loops
- permutationw with **no intermediate values**
- arguments passed to a function (see hereafter)

`for` loops:

In [5]:
```
for a, b, *_ in [(1, 2, 30),
                 (4, 5, 60, 42)]:
    print(a, b)
```

```
1 2
4 5
```

Permutations:

In [6]:
```
a = 5
b = 6
a, b = b, a
print(a, b)
```

```
6 5
```

Function signature

Simple

A function signature presents the name and expected order of every argument of this function.

In a function call, these arguments are of two types:

- positional: their role is defined by the place they take in the arguments order
- named (*keyword arguments*, i.e. `kwargs`)

In [7]:

```
def f(a, b, c):
    print(f"`a`: {a}        `b`: {b}        `c`: {c}")

f(1, 2, 3)        # all positional
f(1, 2, c=5)      # some positional, some named
f(c=5, a=1, b=2)  # all named, order does not matter
```

```
`a`: 1        `b`: 2        `c`: 3
`a`: 1        `b`: 2        `c`: 5
`a`: 1        `b`: 2        `c`: 5
```

Named arguments are always placed **after** positional arguments.

In [8]:
```
f(1, b=2, 3)
```

```
  Cell In[8], line 1
    f(1, b=2, 3)
                ^
SyntaxError: positional argument follows keyword argument
```

An argument cannot be specified both as positional and named:

In [9]:
```
f(1, a=1, b=2, c=5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 f(1, a=1, b=2, c=5)

TypeError: f() got multiple values for argument 'a'
```

Default value

An argument can be absent from a function call if the function signature defines for this argument a default value. If this argument is given, default value is not taken into account.

In [10]:

```
def f2(a, b, c=3):
    print(f"`a`: {a}        `b`: {b}        `c`: {c}")

f2(1, 2)
f2(1, 2, 5)
```

```
 `a`: 1        `b`: 2        `c`: 3
 `a`: 1        `b`: 2        `c`: 5
```

It is common to assign a `None` value to optional arguments. Then the body of the function must contain a special treatment for this argument.

In [11]:
```
def f3(a, b, c=None):
    if c is None:
        c = 0
    return a + b + c

print(f3(1, 2))
print(f3(1, 2, 5))
```

```
3
8
```

Beware: default argument is defined only once (when the function is defined): if it **mutable, it will be modified from a call to another**

In [12]:
```
def f4(a, c=[0]):
    c.append(a)
    print(c)

f4(1)
f4(2)
f4(3)
```

```
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
```

Undeterminated positional arguments: `*args`

A function can take an undeterminated number of positional arguments with the syntax `*args`.

During a function call, all positional arguments undescribed by the function signature are gathered into a `tuple` and passed to the function using the name `args`.

In [13]:
```
def f5(a, b, *args):
    print(a, b, args)

f5(1, 2, 3, 4, 5)
f5(1, 2)
```

```
1 2 (3, 4, 5)
1 2 ()
```

Note that word "`args`" is only a convention.

In [14]:
```
def f6(a, b, *other_values):
    print(a, b, other_values)
```

Every argument following `*args` must be **named**.

In [15]:
```
def f7(a, *args, b):
    print(a, b, args)

f7(1, 3, 4, 5, b=2)
# f7(1, 3, 4, 5, 2)
```

```
1 2 (3, 4, 5)
```

That explains why only `**kwargs` comes after `*args`.

Undeterminated keyword arguments: `**kwargs`

Similar to `*args`, `**kwargs` contains named arguments that are not defined by the function signature. Beware that the keys of `kwargs` are of type `str` (and the values are the passed variables).

In [16]:
```
def f8(a, b, **kwargs):
    print(kwargs)
    print(a, "  ", b, end="   ")
    print(kwargs.get("c", 0), end="   ")    # if 'c' does not exist as a dict key, ta
    print(kwargs.get("d", 0))
```

In [17]:
```
f8(1, 2)
```

```
{}
1    2   0   0
```

In [18]:
```
f8(1, 2, c=3, d=5)
```

```
{'c': 3, 'd': 5}
1    2   3   5
```

In [19]:
```
f8(1, 2, d=5)
```

```
{'d': 5}
1    2   0   5
```

One can also provide named argument using a **dictionary unpacking**..

In the example below, unpacking is used to deconstruct the dictionary into a group of named arguments:

- some are explicitely named
- other go to the `kwarg` variable

In [20]:

```
f8(1, 2)
f8(1, **{"b": 9, "c": 3, "d": 5})
f8(1, 2, **{"d": 5})
```

```
{}
1    2   0   0
{'c': 3, 'd': 5}
1    9   3   5
{'d': 5}
1    2   0   5
```

Advanced examples

In [21]:

```
def custom_sum(a, b, *args, **kwargs):
    weight = kwargs.get("weight", 1)                    # get the value of 'weight' if
    print(a + b + sum([arg * weight for arg in args]))

custom_sum(1, 2)
custom_sum(1, 2, 3, 4)
custom_sum(1, 2, 3, 4, weight=2)
custom_sum(1, 2, weight=2)
```

```
3
10
17
3
```

Note: a force of `**kwargs` (and `*args`) is that it can be easily passed from a function call to another.

In [22]:
```
def advanced_function(a, b, *args, **kwargs):
    if kwargs.get("advanced") > 10:
        kwargs["weight"] = 1
    else:
        kwargs["weight"] = 0
    custom_sum(a, b, *args, **kwargs)

advanced_function(1, 2, 3, 4, advanced=11, useless_kwarg=1000)
advanced_function(1, 2, 3, 4, advanced=9, useless_kwarg=1000)
advanced_function(*[1, 2, 3, 4], advanced=9, useless_kwarg=1000)
```

```
10
3
3
```