

# 04 Lists

## 0.1 Data in memory

Everything we've done so far has had data (mostly) in a file. Just mostly because it wasn't quite true. First, there are the variables, such as the different temperatures, the names of the items and people in the auctions, and so on. But these were only individual values that referred to, say, the currently read line, the highest bid price, and so on.

In addition to these, we have also been familiar with dictionaries since last week. They stored more "mass" data; the keys could be the names of the persons and the corresponding values could be the total amount of money the person in question spent in the auction.

A dictionary can be very large, even thousands of keys and values. No, actually not that it wasn't quite true, it wasn't true at all. But it's true, let's say we never fully read the data from the file into memory and process it there. When solving exercises and homework, we read them from the file on the fly for each task. Moreover, dictionaries are only one - and perhaps not even the most common - data structure for storing data. Why would you want to read data, say a whole file into memory?

We can come up with a bunch of reasons.

- The first is, simply, that we don't want to read the file over and over again. It will also happen that the data in the file will be written in some nasty, complicated way that will take a lot of work and (computer) time to read. In this case, we certainly want to read them only once, process them accordingly, and then work with them in such a processed form.
- Another reason is that it is easier to manipulate the data in the memory, to fold it into a more suitable form. For a trivial example, let's take a file with names of auction participants. Let's say it looks like this.

Berta  
Cilka  
Ema  
Dani  
Ana  
Fanči  
Greta

We want to list them alphabetically.

This will (almost) do nothing but load them into memory, sort them alphabetically there (however that's done) and then print them out.

- Third: sequential access. (This one is a bit similar to the previous point.) In the tasks from the auction, we were repeatedly plagued by the fact that when reading a certain line, we also needed information about what was written in the previous one. We solved this by saving the data at the end of the loop (a typical statement was in the style `previous_temperature = temperature`). This goes as long as we only need one previous data.

When we need the previous five, it will be disgusting. In short: we want to load data here and there in memory. The only data structure we know that allows this is a dictionary.

Let us recall only two of its features.

- A dictionary consists of key-value pairs. For each key, we can find out (or set) its value.
- We cannot influence the order of elements in the dictionary. "Random" order is just the order of addition; we cannot change it later.

In the context of dictionaries, it makes no sense to really talk about order, that is, about how the elements follow each other (this element comes before this one and after that one). The for loop traverses the dictionary in some more or less random order. To put it another way: dictionary elements are not numbered: it makes no sense to talk about the "first", "second" and so on dictionary element. It can confuse us both. It will happen that we will have only some values and no corresponding keys. Let's say the highest daily temperatures in Ljubljana for each day in July 2023. Or the auction record. The numbers themselves. How to store this in a dictionary? It won't work. Except in some extra stupid way, let's say that the keys are just consecutive numbers of days or offered prices. But even in this case, it will annoy us that we are not masters of the order; temperatures, let's say we won't be able to sort them by size in order to list the three warmest days in July. Because - as we wrote in the second point above, dictionaries do not have this concept.

## 0.2 Lists

A list is a data structure that... Uh, this is the third time I've written that term, data structure, and italicized it again and again as if it were the first time I mentioned it. That's because I didn't explain it either first or second. At the third mention, it will be time. Wikipedia defines a data structure as "a collection of data values, the relationships among them, and the functions or operations that can be applied to the data".

- First, a data structure, obviously stores some data. We have nothing to philosophize about here.
- Furthermore, there is some connection between them. In the case of dictionaries, it is a key-value relationship.

For the list it will appear to have the concept of order. This element is the first, this is the second, this is the third. Some associations can be more technical and refer to how the data is arranged within the structure.

- Finally - and the most important - for a data structure are the operations that can be performed with this data and - especially if it were a subject of Algorithms and Data Structures - how efficient and fast these operations are.

We said for a dictionary that we can find out the corresponding value for each key very quickly (we said that Python dictionaries are so efficient that the time it takes to do this is independent of the number of elements stored in the dictionary!), just as quickly we can we also find out if a particular key exists; adding and deleting items is just as fast. It will be meaningless to talk about it in the context of lists, since they do not have keys. Next to each data structure we will say what it can contain and what it can (efficiently) do with these things. A list, then, is a data structure in which we can store any number of elements of any type (in Python; elsewhere, the rule is that all elements must be of the same type).

2

We can make, say, a list of names.

```
imena = ["Ana", "Berta", "Cilka", "Dani", "Ema", "Fanči"]
```

```
[1]: names = ["Ana", "Berta", "Cilka", "Dani", "Ema", "Fanči"]
```

We immediately see: dictionaries had curly brackets, lists angular. After that we separate them. Dictionaries have key-value pairs, lists just... elements. (We will not tell them either the key or the value. In terms of translation from English, it would be more correct to talk about things, because in English it is an item. And element is not exactly a Slovenian word either. :) But what is, is; that's how it settled down.) I can also put something else in the dictionary, for example numbers,

```
teze = [57, 66, 58, 52, 65, 68]
```

```
[2]: theses = [57, 66, 58, 52, 65, 68]
```

or a mix of strings and numbers (which I won't show because it's ugly). We can also make a list that contains several dictionaries, which would be a little unusual, but it will come in handy for this particular subject.

But we can also make an empty list

```
temperature = []
```

```
[11]: temperatures = []
```

and we add elements to it, say from a file.

```
for vrstica in open("december.txt"):
    temperature.append(int(vrstica))
```

```
[12]: for line in open("December.txt"):
    temperature.append(int(row))
```

Just like that, by the way, we learned about the first method of lists: append.

We pass whatever we want as an argument - in this case obviously some number, namely the temperature read from the file - and the append method adds this element to the end of the list.

This is how we get a list of predicted December temperatures in Radovljica.

```
temperature
[5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]
```

```
[9]: temperatures
```

```
[9]: [5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]
```

Tip: Resist the temptation and never call an empty list empty, even though it might feel that way at first.

```
prazen = []
for vrstica in open("december.txt"):
    prazen.append(int(vrstica))
print(prazen)
```

```
[10]: empty = []
for line in open("december.txt"):
    empty.append(int(row))
print(empty)
[5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]
```

It's obviously not empty, is it? Now we know the first part: what the list data structure can store. The second is what we can do with these things.

3

### 0.2.1 Walking through the list

First, something we already know: we can go through the elements of a list with a for loop.

```
for temp in temperature:  
    print(temp)
```

```
[14]: for temp and temperature:  
print(temp)
```

```
5  
4  
-1  
-6  
-8  
2  
5  
-6  
-8  
-12  
-15  
6  
7  
-20  
2  
3
```

So far we don't know how to do anything with files (and we won't know how to do much else in the future) other than to run a for loop through them. Since we can do the same with lists, we can do everything we've done with files so far with lists. With the advantage that the temperature lists will typically already be numbers (or whatever it takes) and not having to constantly call a nasty int or float. Well, obviously we skip the open. The list is already here, read, ready, and you just need to pray for it. So, for an exercise, let's find the smallest element of the list.

```
najm = 100000  
  
for temp in temperature:  
    if temp < najm:  
        najm = temp  
  
najm
```

```
-20
```

```
[17]: rent = 100000  
for temp and temperature:  
if temp < rent:  
rent = temp  
rental  
[17]: -20
```

We have already trained the for loop in such a way that there is nothing to repeat here. We already know everything.

### 0.2.2 Functions that walk for us

This is the fourth week of the course and in the first three we have calculated the largest element, the smallest element and the sum. There was no other way. Python does have max, min, and sum functions, and many others like them, but the files contained strings, and if you passed them to these functions, you would get a string that

4

is the last in the alphabet (max), the first in the alphabet (min), or the sum of all strings (sum + a little effort with additional arguments). Now that we have lists of numbers, all such functions will work as they should.

```
max(temperature)
```

7

```
min(teme)
```

-20

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

```
[27]:
```

```
[28]: min(topics)
```

```
[28]: -20
```

By the way, let's say that lists (like dictionaries and strings, but not files) have length. "Length" simply means "number of elements".

We find it with the function len, to which we give as an argument a dictionary, a string or any data structure that knows the idea of "length".)

```
print("Povprečna temperatura decembra bo", sum(temperature) / len(temperature))
```

```
[34]: print("The average temperature in December will be", sum(temperature) / len(temperature))
```

The average temperature in December will be -2,625

### 0.2.3 List Length

Incidentally, lists (like dictionaries and arrays, but not files) have length. "Length" simply means "number of elements". It is derived by the function len, given as an argument a dictionary, a string, or any other data structure that knows the idea of "length".)

```
len(temperature)
```

16

```
[35]: len(temperature)
```

```
[35]:
```

16

### 0.2.4 Content

We will often be interested in whether the dictionary contains such and such an element or not. A while ago we prepared a list of names, keeping silent that these were the names of the members of the Molj book club.

```
imena
```

```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči']
```

[46]: names

[46]: ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči']

We know the `in` operator from last week when it told us whether the dictionary contains a particular key or not. If we give it a list instead of a dictionary, it will tell us if it contains that element. So if we want to know if Cilka is in the club, the operator tells us that she is.

```
"Cilka" in imena
```

```
True
```

[47]:

"Cilka" and names

[47]:

True

For Greta, he tells us that it is not.

5

```
"Greta" in imena
```

```
False
```

[48]:

"Greta" and names

[48]:

False

## 0.2.5 List unpacking and skeleton from split closet

We have a list that we are absolutely, absolutely, unwaveringly certain contains two items.

```
s = [42, 13]
```

[33]: s = [42, 13]

Its elements can be assigned to two variables.

```
odgovor, nesreča = s
```

```
odgovor
```

```
42
```

```
nesreča
```

```
13
```

```
[36]: answer, accident = s
[37]: answer
[37]:
42
[38]: an accident
[38]:
13
```

This is not something completely new for us either. We have already taken the unpacking. We had a string that contained, say, the name of the place and the maximum and minimum temperature; so three pieces of data separated by commas. We broke it with `split` and adjusted it to three variables.

```
vrstica = "Ljubljana,19,15"
kraj, najvisja, najnizja = vrstica.split(",")
```

```
[40]: row = "Ljubljana,19,15"
place, highest, lowest = row.split(",")
```

I told you that, you understood that `split` returns three things and so we assign them to three variables. The authority argument (namely mine) won, you didn't ask anymore. Well, they should. Every function returns something - and exactly one thing! What kind of thing does `split` return? It actually returns a list.

```
vrstica.split(",")
['Ljubljana', '19', '15']
```

```
[41]: line.split(",")
[41]: ['Ljubljana', '19', '15']
```

So somehow we've been using the lists for the past hour, but I kept them quiet because it wasn't time for them yet. Now that it is, you also finally know what `split` does.

## 0.2.6 List Length

Incidentally, lists (like dictionaries and arrays, but not files) have length.

"Length" simply means "number of elements".

It is derived by the function `len`, given as an argument a dictionary, a string, or any other data structure that knows the idea of "length".)

```
6
```

```
len(temperature)
16
```

```
[35]: len(temperature)
[35]:
16
```

## 0.2.7 Example: counting masks

```
ana_je_rekla = """
danes sem am zjutraj like vstala pa ampak sem še napol spala pa like nisem vedla
a čem prec poklicat Julijo pa sem bla pač like ne bom še pa je pač nisem pa sem
↳pač
like še naprej kar ležala pa am čakal če me bo pač ona"
"""
```

[55]: ana\_je\_rekla = ""

Today I got up in the morning, but I was still half asleep, but I didn't know what to call Julia.

↳ just

I kept lying there, but I was waiting to see if she would take me" "" Let's count the number of times she managed to say am, like or pacha in one sentence.

```
: masila = ["like", "pač", "am"]

masil = 0
for beseda in ana_je_rekla.split():
    if beseda in masila:
        masil += 1

print(masil)
```

10

[56]: masila = ["like", "pach", "am"]

mass = 0

for word in ana\_je\_kla.split():

if word and masila:

masil += 1

print(oil)

10

ana\_je\_rekla.split() returns us a list of Ana's words (here it helps us a little that the sentence is written without punctuation marks, otherwise we should get rid of them. Everything is written in lowercase letters and so on. Most Slovenian also has its advantages.

```
ana_je_rekla.split()
```

```
['Danes',
 'sem',
 'am',
 'zjutraj',
 'like',
 'vstala',
 'pa',
 'ampak',
 'sem',
 'še',
 'napol',
 'spala',
 'pa',
 'like',
 'nisem',
 'vedla',
```



```
'a',  
'čem',  
'prec',  
'poklicat',  
'Julijo',  
'pa',  
'sem',  
'bla',  
'pač',  
'like',  
'ne',  
'bom',  
'še',  
'pa',  
'je',  
'pač',  
'nisem']
```

[50]: ana\_je\_kla.split()

[50]: ['Today',

'I am',

'am',

'morning',

'likes',

'risen',

'well',

'but',

'am'

'yet',

'half',

'slept',

'or',

'like',

'I'm not',

'knew',

7

'a',

'what',

'past',

'call',

'Julia',

'or',

'am',

'blah',

'just',

'like',

'no',

'I will',

'yet',

'or',

'is',

'pač',

'I didn't']

For each word from this list (for word and `ana_je_rekla.split()`), we check whether it is also in the list of masks that we defined at the top. If, then we add another mask. Maybe we are also interested in the proportion of masals in her speech? Then we need the number of words. In order not to split twice, we immediately call split, store the list in words and use it twice.

```
masila = ["like", "pač", "am"]

besede = ana_je_rekla.split()
masil = 0
for beseda in besede:
    if beseda in masila:
        masil += 1

print(masil / len(besede))
```

0.20408163265306123

```
[54]: masila = ["like", "pach", "am"]
```

```
words = ana_je_rekla.split()
masil = 0
for word in words:
    if word in masila:
        masil += 1
print(masil/len(words))
0.20408163265306123
```

Twenty percent, every fifth word, is a mask. The example is, of course, artificial. In actual speech, of course, he prayed more. Likes.

### 0.2.8 Lists of lists and unpacking in a for loop

Knowing that split actually returns lists, you might come up with the idea that the file “bikes.txt”, which contains the bike someone went on a trip with, the number of kilometers traveled and the total climb in meters,

```
Nakamura,16,24
Nakamura, 11.80
Nakamura,14,15
Cube, 50, 1888
```

8

... and so on ... read into a list that would look like this

```
rides = [["Nakamura", 16, 24], ["Nakamura", 11, 80], ["Nakamura", 14, 15], ["Cube", 50, 1888]]
```

So the list will contain lists. A list can contain anything, and last time I checked, the “anything” category covered lists as well. Before we continue, let's mention that the list can be extended to several lines.

```
driving = [
    ["Nakamura", 16, 24],
    ["Nakamura", 11, 80],
    ["Nakamura", 14, 15],
    ["Cube", 50, 1888]]
```

Then we warn: the idea is correct, but we have already warned against lists that contain different things. We will not like this; when we know a little more, the driving list will not be lists with three elements (a string and two numbers) but something else. But for now, let it be: we're going to make a list of lists. The naïve starts right away.

```
voznje = []
for vrstica in open("kolesa.txt"):
    voznje.append(vrstica.split(","))
```

```
[57]: drives = []
for line in open("bikes.txt"):
    driving.append(line.split(","))
```

After finishing the program, he is satisfied with its efficiency, because he immediately forwarded the result of the split - a list of data for an individual run - to append. We will not print it out, the list is too long, as it contains

```
len(voznje)
```

100

```
[59]: len(driving)
[59]:
100
driving.
```

So you'll have to take my word for it that he didn't quite get lucky. split returns strings. He forgot to convert the second and third things, distance and height, to int. It's okay, it will have to be done slowly.

```
voznje = []
for vrstica in open("kolesa.txt"):
    kolo, razdalja, visina = vrstica.split(",")
    voznje.append([kolo, int(razdalja), int(visina)])
```

```
[62]: drives = []
for line in open("kolesa.txt"):
    wheel, distance, height = row.split(",")
    rides.append([wheel, int(distance), int(height)])
```

We will not write these 100 lines in the notes, but only in lectures and at home. We will check that we did it right by calculating how much he traveled with which bike. For practice and for repeating dictionaries.

```
razdalje = {}
for podatek in voznje:
```

```
    kolo, razdalja, visina = podatek

    if kolo not in razdalje:
        razdalje[kolo] = 0
    razdalje[kolo] += razdalja

print(razdalje)
```

```
[63]: distances = {}
for information and driving:
9
wheel, distance, height = data
if bike notes and distances:
distances[wheel] = 0
distances[wheel] += distance
print(distances)
{'Nakamura':
439, 'Cube':
3174, 'Canyon':
2766, 'Stevens':
607}
```

This is the first and last time we wrote such a thing. Namely

For data and driving:

```
wheel, distance, height = data
```

The second line is unnecessary. If we want to unpack a list element, we don't need to first put it in a variable (data) and then extract it. No, it's not just manipulation, =, but the for loop that unpacks the list. We can write - and from now on we will - just like this:

```
razdalje = {}

for kolo, razdalja, visina in voznje:
    if kolo not in razdalje:
        razdalje[kolo] = 0
    razdalje[kolo] += razdalja

print(razdalje)

{'Nakamura': 439, 'Cube': 3174, 'Canyon': 2766, 'Stevens': 607}
```

```
[64]: distances = {}
for bike, distance, height and rides:
if bike not and distances:
distances[bike] = 0
distances[bike] += distance
print(distances)
{'Nakamura':
439, 'Cube':
3174, 'Canyon':
2766, 'Stevens':
607}
```

What we wrote before works, but it's ugly and, as it will soon turn out, also causes other annoyances and long-term relationships. But it is also wrong; if you get help from friends who are more proficient in languages other than Python, they will show you something that will be even uglier and make your future programming even more awkward, time-consuming and cumbersome.

Therefore: if we run a for loop over a list (or something else) that contains elements that need to be unpacked, we unpack them already in the head of the loop.

### 0.3 The first friend of lists: the zip function

It will happen that the data we will work with will be in two lists. We indicated the situation above: we have a list of people's names and a list of their weights.

```
imena = ["Ana", "Berta", "Cilka", "Dani"]
teze = [57, 66, 58, 52]
```

[87]: names = ["Ana", "Berta", "Cilka", "Dani"] theses = [57, 66, 58, 52]

Now we have to write out the names and weights, in style

```
Ana: 57
Berta: 66
Cilka: 58
Dani: 52
```

```
Ema: 65
Fanči: 68
```

Ana:

57

Berta:

66

Cilka:

58

Dani:

52

10

Ema:

65

Fanči:

68

Maybe someone would come up with the idea to do this:

```
for ime in imena:
    for teza in teze:
        print(ime, teza)
```

```
Ana 57
Ana 66
Ana 58
Ana 52
Berta 57
Berta 66
Berta 58
Berta 52
Cilka 57
Cilka 66
Cilka 58
Cilka 52
Dani 57
Dani 66
Dani 58
Dani 52
```

[88]: for name and names:

for thesis and theses:

print(name, thesis)

Ana 57

Ana 66  
Ana 58  
Ana 52  
Berta 57  
Berta 66  
Berta 58  
Berta 52  
Objective 57  
Objective 66  
Objective 58  
Objective 52  
Days 57  
Days 66  
Days 58  
Days 52

The printout reveals that such a person would come up with a bad idea. What I would do is go through all the names and for each name through all the weights. This is clearly not it. Who else would come up with the idea to write

```
for ime in imena:  
    for teza in teze:  
        print(ime, teza)
```

```
Cell In[89], line 2  
    for teza in teze:  
    ^  
IndentationError: expected an indented block after 'for' statement on line 1
```

[89]: for name and names:  
for the thesis and theses:  
print(name, thesis)  
Cell In[89], line 2  
for thesis and thesis:

^ IndentationError: expected an indented block after 'for' statement on line 1

In the blind hope that the two loops will then run somehow... parallel. This idea is obviously even worse than the first. Previously, Python at least did something, but now it resists at the very beginning. (Actually, that might be better. A matter of taste. In any case, it's just as useless as before.) Let's clear up: we need one loop. Not two. So sure, for name, thesis and ...

11

So the loop should get pairs, but from two lists at the same time. Deus ex machina is called zip. We give the function two lists, and it will pack them into a list of pairs... which we unpack again in for. (If anyone is worried that this packing and unpacking will be slow, or that it will in any way "burden" the computer, their worries are unnecessary. These things are well thought out and made to be fast. What we do here is a very basic and common thing.)

```
: for ime, teza in zip(imena, teze):  
    print(ime + ":", teza)
```

```
Ana: 57  
Berta: 66  
Cilka: 58  
Dani: 52
```

[90]: for name, thesis and zip(names, thesis):  
print(name + ":", thesis)

```
Ana:  
57  
Bertha:  
66  
Cilka:  
58  
Dani:  
52
```

The zip function can also receive more than one thing. If we give her three lists, she will make triplets.

```
imena = ["Ana", "Berta", "Cilka", "Dani"]  
teze = [57, 66, 58, 52]  
visine = [1.56, 1.75, 1.70, 1.68]  
  
for ime, teza, visina in zip(imena, teze, visine):  
    print(ime + ":", teza / visina ** 2)
```

```
Ana: 23.422090729783037  
Berta: 21.551020408163264  
Cilka: 20.06920415224914  
Dani: 18.42403628117914
```

[91]: names = ["Ana", "Berta", "Cilka", "Dani"] theses = [57, 66, 58, 52]  
heights = [1.56, 1.75, 1.70, 1.68]  
for name, thesis, height and zip(names, thesis, height): print(name + ":", thesis / height \*\* 2)

```
Ana:  
23.422090729783037  
Berta:  
21.551020408163264  
Cilka:  
20.06920415224914  
Dani:  
18.42403628117914
```

Or, better:

```
imena = ["Ana", "Berta", "Cilka", "Dani"]  
teze = [57, 66, 58, 52]  
visine = [1.56, 1.75, 1.70, 1.68]  
  
for ime, teza, visina in zip(imena, teze, visine):  
    print(ime + ":", round(teza / visina ** 2, 2))
```

```
Ana: 23.42  
Berta: 21.55  
Cilka: 20.07  
Dani: 18.42
```

```
[92]: names = ["Ana", "Berta", "Cilka", "Dani"] theses = [57, 66, 58, 52]
heights = [1.56, 1.75, 1.70, 1.68]
for name, thesis, height and zip(names, thesis, height): print(name + ":", round(thesis / height ** 2,
2))
Ana:
23.42
Berta:
21.55
Cilka:
20.07
Dani:
18.42
```

## 0.4 Another friend of lists: the enumerate function

So we have a list of December temperatures.

```
temperature
[5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]

[65]: temperatures
[65]: [5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]

12
```

and we long to write them out like this:

```
1. december: 5
2. december: 4
3. december: -1
```

and so on.

We will loop through the temperatures, but we also need to know the sequence number of the day. This could be done like this:

```
dan = 0

for temp in temperature:
    dan += 1
    print(dan, ". december:", temp)
```

```
1 . december: 5
2 . december: 4
3 . december: -1
4 . december: -6
5 . december: -8
6 . december: 2
7 . december: 5
8 . december: -6
9 . december: -8
10 . december: -12
11 . december: -15
12 . december: 6
13 . december: 7
14 . december: -20
15 . december: 2
16 . december: 3
```



The space before the full stop shines, but bear with me. When the time comes, we'll get rid of it thoroughly. But now it's time for something else: the situation when we need something from the dictionary and at the same time its "sequential number" is so common that Python offers us a special function for this purpose: `enumerate`. The `enumerate` function receives as an argument a list or any other thing through which it is possible to start the for loop, for example a file, a dictionary, a string or mysterious things that we do not yet know. `enumerate` turns such a list of elements into a list of (sequencenumber, element) pairs. We simply unpack the pair in the loop head. If this description was complicated: the example will be simple.

```
for dan, temp in enumerate(temperature):  
    print(dan, ". december:", temp)
```

```
0 . december: 5  
1 . december: 4  
  
2 . december: -1  
3 . december: -6  
4 . december: -8  
5 . december: 2  
6 . december: 5  
7 . december: -6  
8 . december: -8  
9 . december: -12  
10 . december: -15  
11 . december: 6  
12 . december: 7  
13 . december: -20  
14 . december: 2  
15 . december: 3
```

Great, isn't it? It saved us from having to count the days ourselves. A loop counts them, or, more precisely, enumerates them. Just replace `for temp and temperature` with `for day, temp and enumerate(temperature)`. As said above: `enumerate` makes sure that we get pairs of sequence numbers and elements instead of elements. Oh, what about December? Hmnoya, computer people count from 0. Most programming languages count from 0. Among the loners, unfortunately, there is also R, with which some of you - those who will encounter as much statistical data analysis as possible in your studies - will be bullied in other subjects. Counting from 1 sounds more natural, but in fact counting from 0 is more practical, as will be shown in this course as well. (At the same time, it also has historical and technical reasons in some connection between tables and pointers in C. Whatever tables and pointers are. And C.) If someone is wrong, we will not judge them; let's show him what to fix. It can just add 1 to the output.

```
for dan, temp in enumerate(temperature):  
    print(dan + 1, ". december:", temp)
```

```
1 . december: 5  
2 . december: 4  
3 . december: -1  
4 . december: -6  
5 . december: -8  
6 . december: 2  
7 . december: 5  
8 . december: -6  
9 . december: -8  
10 . december: -12  
11 . december: -15  
12 . december: 6  
13 . december: 7  
14 . december: -20  
15 . december: 2  
16 . december: 3
```

It is nicer and more correct to ask enumerate to count from 1. We do this with an additional argument start, which we specify by name.

```
for dan, temp in enumerate(temperature, start=1):  
    print(dan, ". december:", temp)
```

```
1 . december: 5  
2 . december: 4  
3 . december: -1  
4 . december: -6  
5 . december: -8  
6 . december: 2  
7 . december: 5  
8 . december: -6  
9 . december: -8  
10 . december: -12  
11 . december: -15  
12 . december: 6  
13 . december: 7  
14 . december: -20  
15 . december: 2  
16 . december: 3
```

Instead of 1, we can enter another number. However, we will probably always start with 1 (when not with 0, which will be more often than you imagine).

## 0.5 Indexing

We got to the elements of the dictionary by specifying the key in square brackets, and we got the corresponding value. Lists do not have keys, but they do have an order. Since we can talk about the first, second, and so on elements in this way, we can take them by their sequential numbers.

```
temperature
```

```
[5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]
```

```
[18]: temperatures
```

```
[18]: [5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]
```

The fourth temperature is obtained simply by writing 4 in the brackets.

```
temperature[4]
```

```
-8
```

```
[19]: temperatures[4]
```

```
[19]: -8
```

Yes, it's true, the fourth temperature in the list is .. oops, -6?! -8 is the fifth! Yes, again. We start counting at 0. The initial (to avoid the word "first") element of the dictionary has index 0.

```
temperature[0]
```

```
5
```

Recall that the temperature list has sixteen elements.

```
len(temperature)
```

```
16
```

```
[29]: len(temperatures)
[29]:
16
```

The last temperature is therefore the sixteenth.

```
temperature[16]

-----
IndexError                                Traceback (most recent call last)
Cell In[22], line 1
----> 1 temperature[16]

IndexError: list index out of range
```

```
[22]: temperatures[16]
```

-----IndexError Traceback (most recent call last)

```
Cell In[22], line 1
----> 1 temperature[16]
IndexError: list index out of range
```

I did this for three reasons. :) The first is to think: if the first element has index 0, the sixteenth element has index 15.

```
temperature[15]
3
```

```
[23]: temperatures[15]
[23]:
```

Hold on, the last number is 15. The second is to see what happens if we try to use an index that is too large. The error description should be enough, index out of range, but it's okay if you see it and if we say: when you see this error, check the indexes. The third lesson learned from the example above is that indexing from the end can go wrong, and Python has a handy trick for that: we index from the right with negative indices. The last element has an index of -1, the penultimate element has an index of -2, and so on.

```
temperature
[5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]

temperature[-1]
3

temperature[-2]
2
```

```
[24]: temperatures
[24]: [5, 4, -1, -6, -8, 2, 5, -6, -8, -12, -15, 6, 7, -20, 2, 3]
[25]: temperatures[-1]
[25]:
3
[26]: temperatures[-2]
[26]:
2
```

### 0.5.1 Example: total value of products sold at auction

Having learned how to access elements by their index, we deal with another ugliness from the past, the auction skeleton: how to get to the previous element? For now, let's use what we know (although we'll soon know more and be able to do simpler things): enumerate. Let's remember: the first record of the auction looked like a list of offered prices, and "price" -1 meant that the product was sold for the price from the previous line. Now the prices can be in the list.

```
: cene = []
  for vrstica in open("../domace-naloga/02-drazba/drazba.txt"):
      cene.append(int(vrstica))

cene
```

```
: [11,
   17,
   24,
   30,
   -1,
   13,
   27,
   33,
   -1,
   12,
   27,
   34,
   40,
   -1,
   9,
   -1,
   8,
   20,
   30,
   31,
   -1]
```

```
[73]: prices = []
for line in open("../home-tasks/02-auction/auction.txt"):
    prices.append(int(row))
prices
[73]: [11,
17,
24,
30,
-1,
13,
27,
33,
-1,
12,
27,
34,
40,
-1,
9,
-1,
8,
20,
```

```
30,  
31,  
-1]
```

The above piece of code probably doesn't work on your computer. Great: This means you'll learn how to fix the (relative) path to the auction.txt file. In my case, it is one directory higher, then in the homework subdirectory and within that in 02-auction. In your area... search. Now we can go through the numbered price list. When -1 is encountered, the value of the previous element is added to the sum.

```
vsota = 0  
for i, cena in enumerate(cene):  
    if cena == -1:  
        vsota += cene[i - 1]
```

```
vsota
```

```
143
```

```
[74]: sum = 0  
for i, price and enumerate(prices):  
if price == -1:  
sum += prices[i - 1]
```

```
17
```

```
Sum
```

```
[74]:  
143
```

Great for power. If we want to make a little joke, we count from -1, but we will not need to subtract -1. Namely, i will contain the index of the previous element. :)

```
vsota = 0  
for i, cena in enumerate(cene, start=-1):  
    if cena == -1:  
        vsota += cene[i]  
  
vsota
```

```
143
```

```
[75]: sum = 0  
for i, price and enumerate(price, start=-1):  
if price == -1:  
sum += prices[i]  
sum  
[75]:  
143
```

## 0.5.2 Slices

It often happens that we don't need the whole list, but only a part. Let's say the first five elements. Or the last five. Or all items from fifty-two to fifty-seven. By number, from 52 to 57. Let me clear this up first: how many elements are there from 52nd to 57th? This is usually counted on the fingers; we will need one palm and the first finger of the other. Six.

This, classically, happens when we go on vacation from July 17 to 22 and we would like to know how many days it will last. We somehow know that one day more than the difference, but we better count just in case, right? :) Python is a programming language, and in programming languages we prefer not to allow mistakes that come from someone getting lost.

Things should be quite simple (after mastering them; the time needed to master them should of course be as short as possible, but what is more important is that after knowing, we make as few mistakes as possible). So, Python is simple: if we ask for all elements from 52 to 57, it will return five elements. If we want everything from 8 to 12, there will be four. Which four? Exactly: eighth, ninth, tenth and eleventh. And the twelfth? No. This would be the fifth. So, Python has a simple rule: the lower bound counts, the upper bound doesn't. Once you get used to it, it seems like the only logical thing to do. And you can see that it also works perfectly. How do we request such a sublist? So that we have something to work with, let's extend the list of names a little.

```
imena = ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta', 'Helga', 'Iva']
```

```
[78]: names = ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči', 'Greta', 'Helga', 'Iva']
```

The names from the fifth to the eighth are obtained by

```
imena[5:8]
```

```
['Fanči', 'Greta', 'Helga']
```

```
[79]: names[5:8]
```

```
18
```

```
[79]: ['Fanči', 'Greta', 'Helga']
```

```
imena[5]
```

```
'Fanči'
```

```
[80]: names[5]
```

```
[80]:
```

```
'Fanči,
```

```
imena[7]
```

```
'Helga'
```

```
[81]: names[7]
```

```
[81]:
```

```
'Helga'
```

From the fifth to the eighth therefore means without the eighth, so that there are three. Let's take the first five names.

```
imena[0:5]
```

```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

```
[82]: names[0:5]
```

```
[82]: ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

Since we often need the first so many and so many, we can just leave the lower limit out.

```
imena[:5]
```

```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

```
[83]: names[:5]
```

```
[83]: ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema']
```

We can also omit the upper limit.

```
imena[5:]
```

```
['Fanči', 'Greta', 'Helga', 'Iva']
```

```
[84]: names[5:]
```

```
[84]: ['Fanči', 'Greta', 'Helga', 'Iva']
```

This is quite easy to read and understand:

- :5 means the first five, i.e. the first five (zero, first, second, third and fourth, but without the fifth).
- 5: means without the first five, namely everything from the fifth onwards (without zero, first, second, third and fourth). Counting from 0 fits perfectly with the rule that the last index is excluded. The celebrity isn't over yet. What would that be?

```
imena[-3:]
```

```
['Greta', 'Helga', 'Iva']
```

```
[85]: names[-3:]
```

```
[85]: ['Greta', 'Helga', 'Iva']
```

We gave the lower bound and omitted the upper bound. However, the lower limit was negative, that is, from the right. So we get minus the third, minus the second and minus the first element. Which happens to be exactly three. How about this?

19

```
imena[:-3]
```

```
['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči']
```

```
[86]: names[:-3]
```

```
[86]: ['Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči']
```

These are all from the beginning (we omitted the lower limit, 0) to minus the third - but without it. So -3, -2 and -1 are missing. Exactly three. Let's complete what we wrote earlier:

- :5 means the first five,
- 5: means without the first five,
- -5: means the last five,

- :-5 means no last five.

You won't remember this. You don't have to. Just remember that it exists and try to reconstruct it from the rules again and again. It will eventually get into your blood - if you use.

### 0.5.3 Slices and Zip and previous item

Now we know how to calculate the sum of sold items even better than before. Do we understand this?

```
vsota = 0
for cena, naslednji in zip(cene, cene[1:]):
    if naslednji == -1:
        vsota += cena

vsota
```

143

```
[94]: sum = 0
for price, next and zip(prices, prices[1:]):
    if next == -1:
        sum += price
sum
[94]:
143
```

What's going on? We basically have two lists, prices and prices without the first one.

```
print(cene)
print(cene[1:])
```

```
[11, 17, 24, 30, -1, 13, 27, 33, -1, 12, 27, 34, 40, -1, 9, -1, 8, 20, 30, 31,
-1]
[17, 24, 30, -1, 13, 27, 33, -1, 12, 27, 34, 40, -1, 9, -1, 8, 20, 30, 31, -1]
```

```
[95]: print(prices)
print(prices[1:])
[11, 17, 24, 30, -1, 13, 27, 33, -1, 12, 27, 34, 40, -1, 9, -1, 8, 20, 30, 31, -1]
[17, 24, 30, -1, 13, 27, 33, -1, 12, 27, 34, 40, -1, 9, -1, 8, 20, 30, 31, -1]
```

zip burn them:

11 and 17, 17 and 24, 24 and 30, 30 and -1 ... Since the second list is offset from the first by only one, the pairs returned by zip will simply be consecutive elements.

We unpack them into the price and the next.

If next equals -1, the price is the last price offered for "this" item.



# 04. Tuple

The lists are in a seemingly close boat with the tuple data type. The name is coined from the English words quintuple,

In Slovene we have quintuple, sextuple, septuple, octuple .. in short: **Tuple**.

## 1.1 How to make them

At first glance, the difference is that a **Tuple** has round brackets instead of a list's round brackets.

```
imena = ("Ana", "Berta", "Cilka", "Dani", "Ema", "Fanči")
```

You can also make a blank.

```
prazna = ()
```

Or a Tuple with one element.

```
ena = ("Ana", )
```

Don't miss the comma. ("Ana", ) is a single-element tuple, and ("Ana") is just the string "Ana", which for some reason we put in brackets. Since things are allowed to be put in unnecessary parentheses, the things in the parentheses are just things, not tuples. To say that it is a tuple, we need to add a comma.

Another strange thing: tuples don't need parentheses!

```
imena = "Ana", "Berta", "Cilka", "Dani", "Ema", "Fanči"
```

the names are now tuples/tuple, exactly as they were before. But this is so strange that even Python, when asked for the contents of a name variable, responds by writing it in brackets.

```
imena
('Ana', 'Berta', 'Cilka', 'Dani', 'Ema', 'Fanči')
```

For a length 0, of course, you need brackets. It is annoying that for a target of length 1 we do not need brackets.

```
ena = "Ana",
```

```
ena je zdaj terka,
```

```
ena
('Ana',)
```

and that's only because we - perhaps quite by accident - put a comma at the end of the line.

So it is all a bother with the braceless tuples? Why is this even allowed, then? Because there are situations where it looks better. Then - and only then - do we write them without. There is an example near the end of this chapter, and another at the end of the chapter on functions.

## 1.2 What do we do with them - and what don't we do with them?

We can do many things with tuples that we can do with lists.

```
for ime in imena:  
    print(ime)
```

```
Ana  
Berta  
Cilka  
Dani  
Ema  
Fanči
```

```
imena[2]
```

```
'Cilka'
```

```
imena[-3:]
```

```
('Dani', 'Ema', 'Fanči')
```

But we cannot change them.

```
imena.append("Greta")
```

```
-----  
AttributeError                                Traceback (most recent call last)  
Cell In[7], line 1  
----> 1 imena.append("Greta")  
  
AttributeError: 'tuple' object has no attribute 'append'
```

```
del imena[2]
```

```
-----  
TypeError                                Traceback (most recent call last)
```

```
: imena[2] = "Cecilija"
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[9], line 1  
----> 1 imena[2] = "Cecilija"  
  
TypeError: 'tuple' object does not support item assignment
```

No and no. It's not.

Tuples are an interesting thing: once you make a tuple, it will always stay the way it was when you made it. when it was born. Actually, that's not unexpected: int, float, bool, str have the same property. So, apart from lists and dictionaries, everything we know.

## 1.3 So why do they exist at all?

Why, indeed, would anyone want a data type whose values can never be changed?

### 1.3.1 Because they can't be changed

Sometimes we need exactly that: something that is certain not to change. Remember dictionary keys?

We said then that keys can only be things that are immutable/unchangeable. These are, more or less, numbers and strings (and True, False and None, which are not very common dictionary keys, and, say, functions and modules, which are even rarer). Sometimes someone might be tempted to use a list as

a key. This is not the case: lists can obviously be modified, so they cannot be keys. But if you throw the same values into a Tuple, it works.

```
d = {}
s = [1, 2, 3]

d[s] = 0

-----
TypeError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 d[s] = 0

TypeError: unhashable type: 'list'

t = tuple(s)
t
```

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

```
d[t] = 0
```

```
d
```

```
{(1, 2, 3): 0}
```

### 1.3.2 Because they are so simple

Second, Python uses Tuples a lot internally. For example, when you call a function with multiple arguments, the function actually (but unnoticeably to the programmer) gets them packed into a Tuple. A lot of things inside Python are actually tuples. Lists would be less suited to this role, because they can change, because they need to be able to extend and truncate... They are simply too complex, they allow too many things, they are too hard to control. However, this, the second reason, is the one that interests us least in this subject.

It will also happen to us that we will do a quick tuple at some point, because it will be so practical. Suppose we have two variables, x and y.

```
x = 6
y = 3
```

It may be necessary for x to be smaller and y to be larger in order for the programme to continue. So, if necessary, swap them.

```
x, y = y, x
```

Understood? On the right side is the tuple (just without brackets - this is one of the times we dare to spell it that way, so we do). On the left, we unpack that tuple. So basically  $x, y = (y, x)$ , but with less clutter.

But what if we don't know whether x is actually smaller than y and we're not sure they should be swapped? The beginner writes:

```
if x > y:
    x, y = y, x
```

And Python's old cat:

```
: x, y = min(x, y), max(x, y)
```

### 1.3.3 To keep "records"

Suppose you want to store data from one line of a file about, for example, an auction. So you would like to record the name of the item, the bidder and the bid price. You could use a list,

```
ponudba = ["slika", "Ana", 30]
```

There's nothing wrong with that, but it's not a typical job for lists. Lists are something that changes, we add items to it, we delete items from it. We typically store things like that in tuples.

```
ponudba = ("slika", "Ana", 30)
```

The difference is not even big, in practice it will be relatively indifferent. At least for you. A real Python programmer, however, sees this as, if nothing else, an optional rule, and shudders a little at the first one in the list. I recommend that you get used to such an order yourself.