

## 03-How the computer saves text

In reality, the computer only stores numbers. Both in memory (when we have a str) and on disk (in files) are just numbers. Even when a character travels over the internet, it travels as a number. All these numbers are only turned into characters when they are printed out. So it used to be necessary to agree what number would represent what character. In those days, there were no computers and nobody imagined the internet and all the many fonts and everything that would somehow have to be stored in memory, files and sent by connections. So there were several standards, not uniform, for different purposes. Of them all, the one that survived only: ASCII. This says that we represent capital A by 65, B by 66, C by 67 and so on. Little a is 97, b 98 .. and so on. Other characters have their own codes: a space is 32, a full stop 45, the bracket 40, etc. This is easy to see: the “ord” function accepts a character and returns the code for that character.

```
ord("A")
```

65

```
ord("a")
```

97

```
ord(" ")
```

32

```
ord(". ")
```

46

We will rarely need this feature. It won't go wrong (and in fact it might even be useful) if ... we'll see later.

However, here's a word about the reverse function: pass a code to the “chr”-function and it will return a character:

```
chr(65)
```

'A'

Problem 1: ASCII was developed for teleprinters (like typewriters but remote): what someone typed was transmitted over a wire or antenna to the other side of America, where it was the teleprinter that printed it out. Teleprinters also had functions such as “eject the sheet and take the next one” and so on. These functions also required their own number. If the teleprinter say, it received the number 12, it would take the next sheet of paper. These are called control code. We mention them only because..

Problem 2: ASCII is (was) a seven-bit standard, so there are only 128 digits. 32 numbers were taken up by control codes. The remaining 96 characters were sufficient for the English alphabet, punctuation and a few other odds and ends. The Chinese could not have done much with the full set of 96 characters. ASCII has undergone a bunch of extensions. Already in the seven-bit version, people in the Soviet homeland had corrected it to YUSCII, in which we sacrificed a few characters such as curly brackets and backslashes and replaced them with sharrows. Microsoft and IBM later each drew up their own standard, the so called code tables: these are schedules in which the “bottom part”, the

characters with codes up to 127, is the same as in the original ASCII, while the upper part – codes 128 to 256 – contains various other characters.

Websites usually contain a header which gives information about the code set used. If this is not present, the alphabets will show up right or wrong. And worse: all the text is written in the same code set, so it may not be possible for it to contain both the letter č and è. At the same time, the Chinese still can't be happy with this, because the extra 128 characters are a struggle for them. Until a few years ago, when the Unicode standard came into widespread use. It's made like this: each character is assigned one code. There are 232 codes, which is over and above enough even for the Chinese. Some characters can also be written in more than one way. To write 232 codes, you would need four bytes per character. But that's not ideal either – so the designers of the standard came up with different ways to “compress” the notation so that it requires less than four bytes per character. Of these, by far the most widely used is UTF-8, with UTF-16 occasionally seen.

So what's the difference between the previously full coded sets (CP1250 to CP125 and IBM's sets and ISO-8859) and a bunch of UTF's? Not a big one. The difference is that 2 old sets defined what codes each character had. After the new set, all characters have certain codes, there are only a couple of ways to write these codes. Of these, UTF-8 is predominant, at least in the West. So if we have some text – say in a file or on a web page – that we want to read, it will usually be written in UTF-8 or CP1250. If it is in English and contains only English characters, it is one and the same. If it is in Slovene, we need to know how it is written. If it is a more recent text, it's probably in UTF-8, if it's more than ten years old, or if it comes from a source that has anything to do with some amateur home web page builder, then CP-1250. It still occurs quite often in the vicinity of Windows. We can always try to read it as UTF-8 first. If that works, it's probably really UTF-8, otherwise we read it as CP1250.

How do we tell Python to read text? – The open method has a couple of arguments. The one we're interested in is called encoding, and it's usually given by name: instead of just writing `open("drazba.txt", "utf8")` (which wouldn't actually work because the encoding method is not the second argument, but is preceded by two others), we write `open("drazba.txt", encoding="utf8")`.

But what if we don't tell Python what we want the text to read as – what will it try to read as? This depends on the system settings – that's why Windows has a “System locale” setting. There it asks for “Current language for non-Unicode programs”; if you have English set, it will put the default language to CP1250. On Ubuntu and Mac, the default encoding is UTF-8.

## 02. Control characters in strings

A string can span over several lines if enclosed in triple quotes.

```
s = """Niz, ki je
dolga vec vrstic.
Konkretno, tri!"""

print(s)
```

```
Niz, ki je
dolga vec vrstic.
Konkretno, tri!
```

This is one set, not three. So obviously it must also contain some end-of-line indicator. Because strings are just sequences of numbers, so there is a number that indicates the end of the line.

Specifically, it is the number 10. Control characters also have names (and abbreviations): a character with the code 10 is called a line feed or LF.

```
: ime = "Benjamin"
ime
```

```
: 'Benjamin'
```

```
: print(ime)
```

```
Benjamin
```

The print function prints the string. Without print, we get the representation of the string as it would be typed into the program with quotes. Here is a multi-line string with `\n`. “`\n`” is counted as a character:

```
s = "Tole gre pa\nv dve vrstici"
print(s)
```

```
Tole gre pa
v dve vrstici
```

That is because whenever we write `\n` in a string, Python will not understand it as a backslash but as a newline character. Of all the other similar characters we usually use only one: `\t` – which stands for tab.

Sequences with `\n` and `\t` are called escape sequences. In principle, everything else that starts with an `\` is an escape sequence, but Python is smart enough that when `\` is followed by a nonsense character, it keeps `\`. The backslash therefore has a special role in strings: it is used to the escape sequence.

But what if we want a backslash to be just a backslash? In this case we make two: `\\` is the escape sequence representing (one) backslash.

```
dir = "c:\\documents and settings\\nina\\telovadba"
print(dir)
```

```
c:\documents and settings\nina\telovadba
```

Important: When we write directory names, we always write double backward slashes to get single ones. Or, better: we write just normal slashes, single.

```
dir = "c:\documents and settings\nina\telovadba"
print(dir)
```

Double slashes can be avoided by using “r”. We just put an “r” before the quotation mark and the backslash will just be a backslash.

```
dir = r"c:\documents and settings\nina\telovadba"
print(dir)
```

```
c:\documents and settings\nina\telovadba
```

In this case, we cannot put the escape sequence \n in the string, since \ and n will just be \ and n. Among all the other escape sequences, let us mention just two: \" and \' and double quotes.

## 03b Array methods

### 0.1 Problems with arrays

Having learned about how arrays are stored and what else they contain besides the visible characters, let's run to the weather in Radovljica, the auction data and the wheel selection.

```
for vrstica in open("temperature.txt"):
    print(vrstica)
```

```
24
18
15
16
18
```

Do we see blank lines? Which wouldn't be there if we were outputting numbers, not strings?

```
for vrstica in open("temperature.txt"):
    temp_c = int(vrstica)
    print(temp_c)
```

```
24
18
15
16
18
```

This happens because every line read from the file also contains an end-of-line character. This is, for example, looks like the last one.

```
vrstica
```

```
'18\n'
```

This is then a problem in the exercises, where we have to count the number of times an owner of four bicycles has ridden one of them, say a Cube.

```

: cube = 0
  for vrstica in open("../vaje/kolesa.txt"):
      if vrstica == "Cube":
          cube += 1

cube

: 0

```

If we look at the last line, we can see that he also rode with Cube, but of course the line has `\n` at the end, so comparing `line == "Cube"` returns False.

```
vrstica
```

```
'Cube\n'
```

```
vrstica == "Cube\n"
```

```
True
```

Of course, we can write:

```

cube = 0
for vrstica in open("../vaje/kolesa.txt"):
    if vrstica == "Cube\n":
        cube += 1

cube

46

```

And we get the right result, but it gets on our nerves. Is there any way, to get rid of this embarrassing `\n`? Even worse: what if he is inconsistent in his spelling and sometimes writes “Cube”, sometimes “cube” and sometimes “CUBE”? Is there some form of collation that ignores the difference between upper and lower case? Is there any simple way to replace all the double n’s in a string with single n’s? Where do we go? To the functions for working with strings. Something more general, actually.

## 02.Array methods

So far, we’ve only learned a handful of Python functions – just enough to learn how to call them and to do a bit of programming – we know about `print` and `input`. These are both generally useful functions, especially the first. (The second will quickly sink from memory, because it’s not really that useful.) Similarly `open`. – We know about a few functions for working with numbers, such as `sqrt`, `sin` and `log`. These are really tucked away in a little box called `math`, and to use them we had to conjure them up with `from math import *`. – We know about `int` and `float`. These are not really functions, but types. But all types in Python are also functions in some way. Anyway `int` and `float` are obviously very general things. Python 3.12 actually only has 71 functions (of which some 20 are types). Moreover, out of these 71 functions, that were listed briefly, there are only 20 functions that we will actually use on a somewhat regular basis. The rest are more exotic than not. The rest, the useful functions – and Python has thousands of them – are boxed up. What boxes? Like this. Take a string, say:

```
vrstica = "NNakamura\n"
```

Like numbers, we drew the strings on the board as boxes, containing a piece of information. In addition, these boxes also contain functions to work with this data. As mentioned in the “Bikes” exercise, we can get rid of the extra `\n` at the end of a string by writing `line = line.strip()`. This `strip` is a function that is located “inside” the string `line`. And to get to what’s “inside”, we write `line`, then a

period (which in all Python contexts – there are a few more to come – means “inside”), and then the name of what’s “inside”, i.e. strip. Since it’s a function, we call it. And since we don’t have any special arguments for it (now), we leave the brackets empty.

```
vrstica
'NNakamura\n'

vrstica.strip()
'NNakamura'
```

We call such functions inside boxes methods, or method in English. Historical. In the context of thinking at the time, describing what happens when you call methods, it made sense, but it's not usually used today.)

### 0.2.1 strip

strip is a string method. All strings have a strip (each its own, another? whatever.) and its job is to return a string with no whitespace at the beginning and at the end. The whitespace is spaces, the newline character and tabs.

```
s = "    niz s    preveč presledki. \n Res.    \n\n "
s.strip()

'niz s    preveč presledki. \n Res.'
```

We see: it has removed all the spaces and new lines at the beginning and the end, and left the ones in between alone. There are lstrip and rstrip versions, which only remove the spaces at the beginning and end. You will remember which is which, because you know English. We can also give an argument to the strip method if we want to remove something other than spaces. We will do this rarely, but let us show anyway:

```
".....čemu služijo te pike?!..."strip(".")
'čemu služijo te pike?!'
```

“..... what are these periods for?!..."strip(".")

By the way, we have seen that we can also call methods on string literals ("literally" specified strings), not only on strings that are already named, like the above s. So we will get rid of the \n at the end of the line with the wheel.

```
: vrstica.strip()
: 'NNakamura'
```

No string method changes the string, but at most returns a new string. Strings are immutable (/unchangeable). The string to which the row refers is still the same as it was.

```
vrstica
'NNakamura\n'
```

The strip method just returned a new string, and if we want the string referenced by the line to no longer contain \n, we need to assign the value returned by line.strip() to the line name.

```
vrstica = vrsrica.strip()
vrstica
'NNakamura'
```

### 0.3 Upper and lower case letters

Our next stumbling block was the comparison, which ignores the difference between upper and lower case letters. That is, one where the strings "canyon", "Canyon" and "CANYON" are identical. I must admit that I have never used a function designed for such a comparison, but I had a hunch where to find it and quickly found it there. For those who might need it, it is called `locale.strcoll`. How does an old programmer like me not know about it? Simple: I solve this problem by taking the simplest shortcut. `locale.strcoll` should only be used for alphabetical ordering of strings, and for equality I simply convert both strings to lower (or upper) case.

```
vrstica.lower()
'nnakamura'

: vrsrica.upper()
: 'NNAKAMURA'
```

Or even:

```
vrstica.capitalize()
'Nnakamura'
```

Again: none of these functions modify the string, they all just return a new string. If I want a line to contain a string written in all lowercase letters, I need:

```
vrstica = vrsrica.lower()
vrstica
'nnakamura'
```

For the sake of completeness, let's talk about one more method: `casefold`. This is similar to `lower`, but makes some other conversions, for example replacing ß with ss, since "Straße" is the same as "strasse" for those who know German. In addition, for example, it makes sure that all the syllables are spelled in the same way (to spell them can be a single character or a combination of c/s/z with a semicolon).

```
"Straße".casefold()
'strasse'
```

### 0.4 Replacement of subsets

The `replace` method also has an awkward name, because it does not modify a string, but returns a string in which a character or substring is replaced by another.

```
vrstica.replace("a", "e")
```

```
'nnekemure'
```

```
vrstica.replace("mu", "mjav")
```

```
'nnakamjavra'
```

```
vrstica.replace("am", "")
```

```
'nnakura'
```

What we need is to replace two nn's with one:

```
vrstica.replace("nn", "n")
```

```
'nakamura'
```

It does the same for Canyon:

```
"canyon".replace("nn", "n")
```

```
'canyon'
```

```
"canyonn".replace("nn", "n")
```

```
'canyon'
```

```
"cannyonn".replace("nn", "n")
```

```
'canyon'
```

## 0.5 All together

As each of these methods returns a new string, they can be "chained".

```
vrstica = "NNakamura\n"
```

```
vrstica.strip().lower().replace("nn", "n")
```

```
'nakamura'
```

And now we can finally count those bikes:



```

cube = stevens = nakamura = canyon = 0
for vrstica in open("../vaje/kolesa.txt"):
    vrstica = vrstica.strip().lower().replace("\n", " ")
    if vrstica == "cube":
        cube += 1
    if vrstica == "stevens":
        stevens += 1
    if vrstica == "nakamura":
        nakamura += 1
    if vrstica == "canyon":
        canyon += 1

print("Cube:", cube)
print("Stevens:", stevens)
print("Nakamura:", nakamura)
print("Canyon:", canyon)

```

Cube: 46  
Stevens: 1

Nakamura: 23  
Canyon: 30

## 0.6 Start and end

Arrays have a couple of dozen more methods, but most of them are either not yet understood or not very generally applicable. We'll list just two more here, and then devote a longer section to the star of today's lecture, the split method.

Simple but often useful methods are startswith and endswith. They are passed a string and return True or False.

```
ime = "Benjamin"
```

```
ime.startswith("Ben")
```

True

```
ime.startswith("B")
```

True

```
ime.startswith("")
```

True

```
ime.startswith("tisto zgoraj je malo hecno")
```

False

```
ime.endswith("min")
```

True

```
ime.endswith("max")
```

False

## 1 Breaking the string

Yep. In reality, there is only a rather simplified version of the cycling statistics file in the tutorial directory. In reality, it looks like this:

```
Cube,51,1216
Canyon,104,444
Cube,74,1508
Cube,79,936
Nakamura,17,95
(in tako naprej)
```

Each line contains three pieces of information, separated by commas: first the name of the bike the unidentified cyclist had for the trip on a given day, then the distance travelled, then the height metres. How to read this? Let's start with just one line.

```
vrstica = "Cube,51,1216"
```

The so far silent method of stacking is split. This breaks strings into multiple strings; the argument is a separator - in our case a comma.

```
: vrstica.split(",")
: ['Cube', '51', '1216']
```

The method returned three sets. What those square brackets mean is not today's topic. They will not distract us. Since we have three sets, we will adapt them to the three variables.

```
kolo, razdalja, visina = vrstica.split(",")
```

Understood? To the right of the equation we have three things, so to the left of it we need three variables. Their values are as they should be.

```
kolo
```

```
'Cube'
```

```
razdalja
```

```
'51'
```

```
visina
```

```
'1216'
```

Let's find out how far the man has travelled and the total height of his trips.

```
skupna_razdalja = 0
skupna_visina = 0
for vrstica in open("kolesa.txt"):
    kolo, razdalja, visina = vrstica.split(",")
    skupna_razdalja += int(razdalja)
    skupna_visina += int(visina)

print("Skupna razdalja:", skupna_razdalja)
print("Skupna višina:", skupna_visina)
```

```
Skupna razdalja: 6359
Skupna višina: 98076
```

I think that is quite remarkable. We can already read quite complex files. There is something else to be said about the split. We often get comma-separated data files from Excel. But sometimes the data is separated by tabs or even just spaces - that is to say, white space, what you strip off. In this case, we call split without arguments.

```
"Cube 45 1024".split()
```

```
['Cube', '45', '1024']
```

This is similar to, but not exactly the same as, using a space as an argument.

```
"Cube 45 1024".split(" ")
```

```
['Cube', '45', '1024']
```

The difference is in how the split handles consecutive spaces in the string.

```
"Cube 45 1024".split()
```

```
['Cube', '45', '1024']
```

```
"Cube 45 1024".split(" ")
```

```
['Cube', '', '', '45', '', '', '', '', '1024']
```

In the first case, it treated consecutive spaces as a single space, but in the second case, when we made the argument, it treated five consecutive spaces as five punctuation marks - with empty strings in between. This is not what we usually want. So, when you're too busy to call `split(" ")`, take the plunge and call `split()`.

## 03c Dictionaries

### 0.1 Dictionaries

Take, again, the cycling data, the ones with four columns. So far, we have only found out the total distance travelled and the height, but we have ignored the bicycle information. We could calculate, say, how far he travelled on the Canyon.

```
vozenj_canyon = 0
razdalja_canyon = 0
visina_canyon = 0

for vrstica in open("kolesa.txt"):
    kolo, razdalja, visina = vrstica.split(",")
    if kolo == "Canyon":
        vozenj_canyon += 1
        razdalja_canyon += int(razdalja)
        visina_canyon += int(visina)

print("Voženj:", vozenj_canyon)
print("Razdalja:", razdalja_canyon)
print("Višina:", visina_canyon)
```

Voženj: 28

Razdalja: 2932

Višina: 25630

Voženj = rides; razdalja = distance; visina = elevation/height

Now the same for Cube, Nakamura and Stevens. If we put these in the same programme we will have a foursome after we put these in the same program, we will have four times three variables. But it could be worse: there are five of us in the family and we have eight bikes. How nice it would be if we could have a single variable of rides, with the "compartments" Canyon, Cube, Nakamura and Stevens. And, of course, the distance and altitude variables, each with four compartments. Obviously, this exists, otherwise I wouldn't want it, would I? Such variables - or, more precisely, such a type of variable is called a dictionary - dictionary in English, and the name of the Python data type is dict. Let's start with something simpler: a dictionary with hill heights. This is what it would look like:

```
hribi = {"Triglav": 2864, "Storžič": 2132, "Grintovec": 2558}
```

The dictionary of hills has compartments "Triglav", "Storžič" and "Grintovec", and the values stored in these compartments are 2864, 2132 and 2558. As we can see, the dictionary is compiled by writing pairs of the compartment name (which is conventionally called the key) and the associated values (the value) between the wrapped brackets, separated by colons. Python can, of course, print out the entire contents of the dictionary.

```
hribi
{'Triglav': 2864, 'Storžič': 2132, 'Grintavec': 2558}
```

To access the individual values in the dictionary, put "compartment" in square brackets.

```
hribi["Triglav"]
2864
hribi["Storžič"]
2132
```

If a key does not exist, Python makes this clear.

```
hribi["Stol"]
-----
KeyError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 hribi["Stol"]

KeyError: 'Stol'
```

The compartments behave like variables. Triglav can be assigned a different height.

```
hribi["Triglav"] = 2800
hribi
{'Triglav': 2800, 'Storžič': 2132, 'Grintavec': 2558}
```

We can also add the (now missing) 64 metres.

```
hribi["Triglav"] += 64
hribi
{'Triglav': 2992, 'Storžič': 2132, 'Grintavec': 2558}
```

In short, like any other variable.

Armed with this new knowledge, let's calculate the cycling statistics.

```
vozenj = {"Canyon": 0, "Cube": 0, "Nakamura": 0, "Stevens": 0}
razdalje = {"Canyon": 0, "Cube": 0, "Nakamura": 0, "Stevens": 0}
visine = {"Canyon": 0, "Cube": 0, "Nakamura": 0, "Stevens": 0}

for vrstica in open("kolesa.txt"):
    kolo, razdalja, visina = vrstica.split(",")
    vozenj[kolo] += 1
    razdalje[kolo] += int(razdalja)
    visine[kolo] += int(visina)
```

And now? Well, what we need is in dictionaries.

```
vozenj
```

```
{'Canyon': 28, 'Cube': 43, 'Nakamura': 29, 'Stevens': 0}
```

```
razdalje
```

```
{'Canyon': 2932, 'Cube': 2939, 'Nakamura': 488, 'Stevens': 0}
```

```
visine
```

```
{'Canyon': 25630, 'Cube': 70947, 'Nakamura': 1499, 'Stevens': 0}
```

But what if you want to spell it out nicely? Say, the data for each wheel? When we say "for each", it reminds us of ... the for loop. So far, we've been running the for loop through files. What happens if we try to run it over the dictionary?

```
for bogvekajboto in vozenj:
    print(bogvekajboto)
```

```
Canyon
Cube
Nakamura
Stevens
```

The loop **for** gives the keys to the dictionary. Right, then we know how to get to the key.

```
for kolo in vozenj:
    print(kolo, "-", vozenj[kolo], "vozenj")
```

```
Canyon - 28 vozenj
Cube - 43 vozenj
Nakamura - 29 vozenj
Stevens - 0 vozenj
```

Since all three dictionaries have the same keys anyway, we can also print out the total height and distance travelled in passing.

```
for kolo in vozenj:
    print(kolo, "-", vozenj[kolo], "vozenj", razdalje[kolo], "km", "\n",
          "\n", visine[kolo], "m.")
```

```
Canyon - 26 vozenj, 2766 km, 26392 m.
Cube - 43 vozenj, 3174 km, 66705 m.
Nakamura - 22 vozenj, 439 km, 1119 m.
Stevens - 9 vozenj, 607 km, 3395 m.
```

## 0.2 Adding to the dictionary

The program starts with:

```
vozenj = {"Canyon": 0, "Cube": 0, "Nakamura": 0, "Stevens": 0}
razdalje = {"Canyon": 0, "Cube": 0, "Nakamura": 0, "Stevens": 0}
visine = {"Canyon": 0, "Cube": 0, "Nakamura": 0, "Stevens": 0}
```

There seems to be some room for improvement. First: do we really need to type all these wheel names and zeros? Couldn't we do without? Second: for this to work, we need to know in advance which bike brands will appear in the file. What if this is not predefined? How would we program in that case?

We will have to learn three things:

1. create an empty dictionary,
2. check whether the dictionary has a specific compartment and
3. if it doesn't, add a new compartment.

All three of these will turn out to be quite simple. How to build an empty dictionary is easy to guess. We just don't write anything between the brackets.

```
vozenj = {}
```

```
vozenj
```

```
{}
```

How do I check whether a dictionary contains a certain key? So far, we have used the word **in** and **for** in the loop when we write, for example, for wheel and commute. You can also use it without if, in a boolean expression, as an operator. The operator `a in c` returns True if `c` contains `a`.

We still have at hand:

```
razdalje
```

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

The dictionary contains "Canyon" but not "Focus". The operator agrees with us.

```
"Canyon" in razdalje
```

```
True
```

```
"Focus" in razdalje
```

```
False
```

Great, we can check if the compartment exists. Tick under point 2.

Now to add the compartment.

```
razdalje["Focus"] = 0
```

```
razdalje
```

```
{'Canyon': 2766, 'Cube': 3174, 'Nakamura': 439, 'Stevens': 607, 'Focus': 0}
```

```
hribi["Šmarna gora"] = 667
```

```
hribi
```

```
{'Triglav': 2992, 'Storžič': 2132, 'Grintavec': 2558, 'Šmarna gora': 667}
```

And we have a better programme for cycling statistics.

```
vozenj = {}
razdalje = {}
visine = {}

for vrstica in open("kolesa.txt"):
    kolo, razdalja, visina = vrstica.split(",")
    if not kolo in vozenj:
        vozenj[kolo] = razdalje[kolo] = visine[kolo] = 0
    vozenj[kolo] += 1
    razdalje[kolo] += int(razdalja)
    visine[kolo] += int(visina)

for kolo in vozenj:
    print(kolo, "-", vozenj[kolo], "vozenj,", razdalje[kolo], "km,",
          visine[kolo], "m.")
```

Nakamura - 22 vozenj, 439 km, 1119 m.

Cube - 43 vozenj, 3174 km, 66705 m.

Canyon - 26 vozenj, 2766 km, 26392 m.

Stevens - 9 vozenj, 607 km, 3395 m.

As all three dictionaries either contain or do not contain a bicycle, we only check whether the bicycle is already in the dictionary of rides. If it's not there, it's not in the others, so we add it to all of them.

Since not bike and ride read strangely, there is a **not** in operator in addition to the **in** operator.

```
"Scott" not in vozenj
```

```
True
```

### 0.3 Dictionaries - in general

The keys of the above dictionary ("compartment names") were strings and the values were numbers, int. Dictionaries are not limited to these types.

Values can be anything: not only int and float, but also strings, booleans, and all the various types we will learn about. A dictionary can even contain other dictionaries (uh, even itself!) as a key.

Keys are a bit more limited. Keys must be immutable/unchangeable. For our purposes and our knowledge so far: a key can be anything but a dictionary. The "forbidden" key types will soon be joined by the list and the set, but you will not find any other forbidden key types in this subject

For this object, keys will almost always be strings, sometimes int's, and almost never anything else. The same dictionary can have different types of keys. And different types of values.

```
{"Ana": 129415, 4: True, 3.14: {}}
```

```
{'Ana': 129415, 4: True, 3.14: {}}
```

Usually all the keys of a dictionary and all the values of a dictionary will be of the same type.

## 0.4 Dictionaries also have methods

Every single thing in Python has methods. A set of, as we said, dozens. Dictionaries have a few less, but we won't list them all, because we wouldn't remember them anyway. We'll see what we need as we go along.

### 0.4.1 Values

Let's take the dictionary of visines (height). What is the total height travelled by the cyclist?

```
visine
```

```
{'Nakamura': 1119, 'Cube': 66705, 'Canyon': 26392, 'Stevens': 3395}
```

```
1119 + 66705 + 26392 + 3395
```

```
97611
```

No, I didn't think so. We won't be adding them up manually. We'll let the software do it.

```
: skupna = 0
  for kolo in visine:
      skupna += visine[kolo]

  skupna
```

```
: 97611
```

On reflection, we are not really interested in keys here. We are only interested in values. Here we would be better served if the for loop went through the values, not the keys.

No problem. Dictionaries have a values method that returns values.

```
for visina in visine.values():
    print(visina)
```

```
1119
66705
26392
3395
```

And of course we can add it up.

```
skupna = 0
for visina in visine.values():
    skupna += visina

skupna
```

```
97611
```

Sooner or later, you will discover that there is a function **sum** that can add up whatever you give it as an argument. If you pass it visine.values(), it will add up all the values in the dictionary.

```
sum(visine.values())
```

```
97611
```



Therefore:

```
print("Število voženj:", sum(vozenj.values()))
print("Skupna razdalja:", sum(razdalje.values()))
print("Skupna višina:", sum(visine.values()))
```

```
Število voženj: 100
Skupna razdalja: 6986
```

```
Skupna višina: 97611
```

#### 0.4.2 Default values

Dictionaries have a method **setdefault(key, value)** which sets the given key to the given value; if the key already exists, it does nothing. In either case, it returns the value that belongs to that key - either the existing (and hence unchanged) value, or the new value just set.

```
vozenj
{'Nakamura': 22, 'Cube': 43, 'Canyon': 26, 'Stevens': 9}

vozenj.setdefault("Scott", 0)

0

vozenj
{'Nakamura': 22, 'Cube': 43, 'Canyon': 26, 'Stevens': 9, 'Scott': 0}

vozenj.setdefault("Nakamura", 0)

22

vozenj
{'Nakamura': 22, 'Cube': 43, 'Canyon': 26, 'Stevens': 9, 'Scott': 0}
```

This simplifies our calculations:

```
vozenj = {}
razdalje = {}
visine = {}

for vrstica in open("kolesa.txt"):
    kolo, razdalja, visina = vrstica.split(",")
    vozenj.setdefault(kolo, 0)
    razdalje.setdefault(kolo, 0)
    visine.setdefault(kolo, 0)
    vozenj[kolo] += 1
    razdalje[kolo] += int(razdalja)
    visine[kolo] += int(visina)

for kolo in vozenj:
    print(kolo, "-", vozenj[kolo], "vozenj,", razdalje[kolo], "km,",
    visine[kolo], "m.")
```

```
Nakamura - 22 vozenj, 439 km, 1119 m.
Cube - 43 vozenj, 3174 km, 66705 m.
```

```
Canyon - 26 vozenj, 2766 km, 26392 m.
Stevens - 9 vozenj, 607 km, 3395 m.
```

An if sentence and a bit of tampering...

```
if kolo not in vozenj:
    vozenj[kolo] = razdalje[kolo] = visine[kolo] = 0
```

... has been replaced by three lines of `setdefault`. Well, if we had a single dictionary, it's:

```
d.setdefault(x, 0)
```

...which is certainly shorter and simpler than:

```
if x not in d:
    d[x] = 0
```

### 0.4.3 Get

I must tell you about the **get** method simply because it is better to learn about such things from your elders than from your peers. The **get** method returns the value that belongs to the given key.

```
visine
{'Nakamura': 1119, 'Cube': 66705, 'Canyon': 26392, 'Stevens': 3395}
visine.get("Cube")
66705
Ne počnite tega. V Pythonu se reče
visine["Cube"]
66705
```

I will be sad to see such use of **get** in your programs. It will occur because you will be "helped" to solve problems by people you know who program in Java or similarly cumbersome languages, who, because of certain restrictions, do not need to access the contents of dictionaries with square brackets.

Why does Python even have **get** if you're not supposed to use it?

The **get** method can be given a default value to return if the key does not exist.

```
visine.get("Cube", 0)
66705
visine.get("Focus", 0)
0
```

### 0.4.4 items

We have seen that when the **for** loop is passed over the dictionary, it goes over its keys. If we run it through what returns values, it will go through values. Dictionaries also have an `items` method that returns key-value pairs. Let's go back a bit. Let's say we have a two-word string, "Anna Benjamin", and we split it with `split`. We get two words, so we need to store them in two variables.

```
ona, on = "Ana Benjamin".split()
ona
'Ana'
on
'Benjamin'
```

If the `items` method returns a pair, the `for` loop will also need two variables to store it.

```
for kolo, visina in visine.items():
    print(kolo, " - ", visina)
```

```
Nakamura - 1119
Cube - 66705
Canyon - 26392
Stevens - 3395
```

Now, let's write a programme right now to tell which bike has climbed the most.

```
naj_visina = 0

for kolo, visina in visine.items():
    if visina > naj_visina:
        naj_visina = visina
        naj_kolo = kolo

print("Največ se je vzpenjal", naj_kolo, "in sicer", naj_visina, "metrov.")
```

```
Največ se je vzpenjal Cube in sicer 66705 metrov.
```

## Python Tutorial for Beginners 5: Dictionaries - Working with Key-Value Pairs

- Starting a dictionary: student = {}
- Put in keys and values: student = {"name": "John", "age": 25, "courses": ["Math", "Programming"]}
- We want to access a value, we use the name-key: print(student["name"])
- The values in our dictionary can be anything (integer, string, etc.)
- What happens if we access a key that we don't have → error
- We want a value from an existing key: print(student.get("key")) or print(student.get("name"))
- How to add a new value to our dictionary: student["phone"] = "555-5555"
  - If the key already exists, it will update the value
  - If the key does not exist yet, it will create the key and add the value automatically
- We can also update values like that: student.update({"name": "Jane", "age": 26})
  - This takes a dictionary as an argument
  - The other values from before are still in the dictionary
- Delete a key: del student["age"]
- Delete a key or value:
  - age = student.pop("age")
  - → age key is now deleted
- If we wanna see how many keys we have in a dictionary: print(len(student))
- If we wanna see just the keys/values: print(student.keys()) or print(student.values())
- If we wanna see the keys and the values: print(student.items())
  - Then we see key and value pairs
- If we wanna loop through all keys and values of a dict:
  - For key, value in student.items():
    - Print(key, value)

[https://www.w3schools.com/python/exercise.asp?x=xrcise\\_dictionaries\\_add3](https://www.w3schools.com/python/exercise.asp?x=xrcise_dictionaries_add3)

Add an item to a dictionary:

```
x = {'type' : 'fruit', 'name' : 'apple'}  
x.update({'color' : 'green'})
```

Remove an item from the dictionary: dict.pop()

Use the `pop` method to remove "model" from the `car` dictionary.

```
car = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
car.pop("model")
```

Loops and Dictionaries:

What is a correct syntax for looping through the values of this dictionary:

```
x = {'type' : 'fruit', 'name' : 'apple'}
```



```
for y in x.values():  
    print(y)
```

What is a correct syntax for looping through the keys AND values of this dictionary:

```
x = {'type' : 'fruit', 'name' : 'apple'}
```



```
for y, z in x.items():  
    print(y, z)
```