

# 07- Array formatting

## 1. Array design

So far, we have been limited to a rather cumbersome printout. It's good enough to start with, but it soon becomes unwieldy because some things just can't be done (easily) with it. For a start, you have to negotiate with it not to go to a new line after each printout and not to add space between the things we print out.

If we had a dictionary that said how many kilometres someone had travelled on one of their bicycles,

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

and wanted to print it out, print would add spaces before the colons.

```
for kolo, razdalja in razdalje.items():  
    print(kolo, ":", razdalja)
```

```
Canyon : 2932  
Cube : 2939  
Nakamura : 488  
Stevens : 0
```

If you also wanted to write down in brackets what proportion of the journey was made on which bicycle, you would get something like this indigestible:

```
skupna = sum(razdalje.values())  
  
for kolo, razdalja in razdalje.items():  
    print(kolo, ":", razdalja, "(", razdalja / skupna * 100, "%")
```

```
Canyon : 2932 ( 46.10787859726372 %)  
Cube : 2939 ( 46.217958798553234 %)  
Nakamura : 488 ( 7.6741626041830475 %)  
Stevens : 0 ( 0.0 %)
```

Too many decimals, those awful spaces ... maybe some alignment would be nice, no?

The bad news: while we can tell print not to use extra spaces, it really can't be bothered with alignment and decimals.

Most of the text printed or otherwise displayed by programs contains fixed text with strings or numbers embedded in it. If the computer displays "Queen.py file not found", the embedded text is "queens.py", which is probably different each time, depending on which file is being (unsuccessfully) searched for. In "Today is 60 Fahrenheit, i.e. 16 degrees Celsius", the numbers 60 and 16 are inserted, depending on what time of year the program is run (and how much we have already managed to

the planet has warmed up), but the rest of the text is always the same.

Most computer languages allow you to construct strings by writing a "constant" string, marking the places where something needs to be inserted. Then we tell it in some way what to insert.

Popular compiled languages have mostly adopted the way C was invented, which is a bit more cumbersome, but newer languages, especially scripting languages, which are closer to the web, usually allow simpler and more powerful ways of assembling strings from patterns.

One of the principles of Python is that there is one and only one obvious way to do each thing. This is good because, as a consequence, all programmers write similar programs and therefore collaborate more easily. But it's in string formatting that this has been subverted: strings can be formatted in three different ways.

This is because certain ideas (not only in Python) have matured over the years, but you can't just discard (older) ways of doing things, because then older programs wouldn't work in newer versions of Python. A serious Python programmer needs to know all three modes, and the last, most modern one will do for us. In fact, I only use this one.

### 1.1. F-arrays

Suppose we have

```
fahr = 42
celz = (fahr - 32) * 5 / 9
```

Now we would like to make, in fact, a set like this:

```
"{fahr} Fahrenheitov je {celz} Celzijev"
'{fahr} Fahrenheitov je {celz} Celzijev'
```

except that we want Python to insert the values of the variables `fahr` and `celz` in the places marked `{fahr}` and `{celz}`.

This is what f-strings do. An f-string is like a normal string, except that we prefix the first quotation mark with the letter `f` (as a format). In such strings, the curly brackets have a special meaning: what is in them is (computed and) inserted.

```
f"{fahr} Fahrenheitov je {celz} Celzijev"
'42 Fahrenheitov je 5.555555555555555 Celzijev'
```

In a homework assignment - oh, how embarrassing that was! - we wrote

```
from random import randint

a = randint(2, 10)
b = randint(2, 10)

print(a, "krat", b)
c = input("Odgovor? ")
```

By first outputting and then requesting the answer, we avoided having to laboriously assemble the string when calling the input function:

```
a = randint(2, 10)
b = randint(2, 10)
c = input("Koliko je " + str(a) + "x" + str(b) + "?")
```

Now that we know the f-array method, we know a simpler way:

```
a = randint(2, 10)
b = randint(2, 10)
c = input(f"Koliko je {a}x{b}? ")
```

(Incidentally, let's mention the two older ways of formatting strings that we mentioned at the beginning: in C-style, we would write `c = input("What is %sx%s? " % (a, b))`, and in the newer style `c = input("How many is {}x{}? ".format(a, b))`. What they both have in common is that they just mark the

places in the string where to insert, and then somewhere else, after the end of the string, we have to list the things we're inserting. With the newer style, we can also add variable names into the wrapped brackets by some weird trick, but it's nowhere near as convenient as the new style.)

```
datoteka = "kraljice.py"
f"Datoteke {datoteka} ne najdem"

'Datoteke kraljice.py ne najdem'
```

## 1.2 Expressions in f-arrays

It's not just the values of variables that we put into strings: we can write entire expressions in wrapped brackets.

```
fahr = 42

f"{fahr} Fahrenheitov je {(fahr - 32) * 5 / 9} Celzijev"

'42 Fahrenheitov je 5.555555555555555 Celzijev'
```

Now back on the bikes!

```
skupna = sum(razdalje.values())

for kolo, razdalja in razdalje.items():
    print(f"{kolo}: {razdalja} ({razdalja / skupna * 100} %)")
```

```
Canyon: 2932 (46.10787859726372 %)
Cube: 2939 (46.217958798553234 %)
Nakamura: 488 (7.6741626041830475 %)
Stevens: 0 (0.0 %)
```

Partial success. We got rid of spaces, but above all the printout has become clearer. Let's compare:

```
prej: print(kolo, ":", razdalja, "(", razdalja / skupna * 100, "%)")
zdaj: print(f"{kolo}: {razdalja} ({razdalja / skupna * 100} %)")
```

To make it even clearer, we can write

```
skupna = sum(razdalje.values())

for kolo, razdalja in razdalje.items():
    delez = razdalja / skupna * 100
    print(f"{kolo}: {razdalja} ({delez} %)")
```

```
Canyon: 2932 (46.10787859726372 %)
Cube: 2939 (46.217958798553234 %)
Nakamura: 488 (7.6741626041830475 %)
Stevens: 0 (0.0 %)
```

Now f-string is obviously a pattern into which we insert values.

What remains is the battle with decimals.

## 1.3 How we form things

If we want to tell how a number (or string, or whatever) should be written out, we add a colon to the expression, followed by a description of the format.

This is what we will most often need to format non-integer numbers. It goes like this:

```
f"{fahr:4.1f} Fahrenheitov je {celz:4.1f} Celzijev"
```

```
'42.0 Fahrenheitov je 5.6 Celzijev'
```

A description of the format is given here 4.1f. Here 4.1 means that we would like to have a four-place format, with one place reserved for the decimal point. For the letter f, imagine that it stands for float. If we leave too little space for a number, the printout will take up more space.

```
x = 1234.5678
f"Primer predolgega števila: {x:3.1f}"
```

```
'Primer predolgega števila: 1234.6'
```

We asked for one decimal place, and we got it, but the total is wider than four places, because the number is just too long.

Specifying the number of decimal places only makes sense for non-integer numbers. If we are inserting strings or integers, we can only specify the width:

```
podatki = [
    (74, "Anze", False),
    (82, "Benjamin", False),
    (48, "Cilka", True),
```

```
(66, "Dani", False),
(61, "Eva", True),
(101, "Franc", False),
]
```

```
for teza, ime, spol in podatki:
    print(f"{ime:10}{teza:6}")
```

```
Anze      74
Benjamin  82
Cilka     48
Dani      66
Eva       61
Franc    101
```

Here, we have not added the letter f after the number of digits, as they are not (non-integers). (There are other letters that could be added to tell Python what it's about, but we can safely forget them here.)

All names are written in ten characters; the missing space is filled with spaces. Numbers are written out in six digits, again with spaces filling the missing space.

The strings are aligned to the left - spaces are added after them. Most of the things we print (strings and numbers are not the only things that can be printed) are aligned like this. Numbers, on the other hand, are aligned to the right.

We can change this by adding <, > or ^ after the colon to tell it to be left, right or centre aligned. The reverse of the above would be achieved by

```
for teza, ime, spol in podatki:
    print(f"{ime:>10}{teza:<6}")
```

```
Anze74
Benjamin82
Cilka48
Dani66
Eva61
Franc101
```

This is not very good. Let's add a space.

```
for teza, ime, spol in podatki:
    print(f"{ime:>10} {teza:<6}")
```

```
Anze 74
Benjamin 82
Cilka 48
Dani 66
Eva 61
Franc 101
```

This is easier to read, but of course the first printout is still the most beautiful.

The alignment character can be preceded by a symbol to be used for padding - unless, of course, we are happy with a space. Let's align names to the left, numbers to the right, and fill the empty space with dots instead of spaces.

```
for teza, ime, spol in podatki:
    print(f"{ime:.<10}{teza:.>6}")
```

```
Anze...74
Benjamin...82
Cilka...48
Dani...66
Eva...61
Franc...101
```

Any other character can be used instead of dots. When filling space with anything other than spaces like this, we need to add alignment characters (<, > or ^), even when we use them to choose the alignment we would like by default (right for numbers, left for everything else).

We won't go into all the details. There are many more things that can be specified. For numbers, we can ask for the sign to always be displayed, i.e. + when the number is positive. For numbers, we can specify that they should be printed in binary or hexadecimal... Anyone who needs more than that should Google it.

### 1.3.1 Designing the wheels

Back to bicycles. Most of the output is fine, but we would like to output the fractions with fewer decimal places - say one. It is enough to add `:.1f`.

```
: skupna = sum(razdalje.values())

for kolo, razdalja in razdalje.items():
    print(f"{kolo}: {razdalja} ({razdalja / skupna * 100:0.1f} %)")
```

```
Canyon: 2932 (46.1 %)
Cube: 2939 (46.2 %)
Nakamura: 488 (7.7 %)
Stevens: 0 (0.0 %)
```

The whole story of the bicycles is a bit longer. The information is taken from the file.

```

voznje = {"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(",")
    voznje[kolo] += 1
    razdalje[kolo] += int(razdalja)
    visine[kolo] += int(visina)

```

In the dictionaries of ride, distance and height, we have the number of rides and the total distance and height travelled by each bicycle. What if we could display this in a table?

```

sk_voz = sum(voznje.values())
sk_raz = sum(razdalje.values())
sk_vis = sum(visine.values())

for kolo in voznje:
    voz, raz, vis = voznje[kolo], razdalje[kolo], visine[kolo]
    print(f"{kolo:12}{voz:5} ({voz / sk_voz:>5.1%}) {raz:10} ({raz / sk_raz:>5.1%}) {vis:10} ({vis / sk_vis:>5.1%}) ")

```

Canyon	26 (26.0%)	2766 (39.6%)	26392 (27.0%)
Cube	43 (43.0%)	3174 (45.4%)	66705 (68.3%)
Nakamura	22 (22.0%)	439 ( 6.3%)	1119 ( 1.1%)
Stevens	9 ( 9.0%)	607 ( 8.7%)	3395 ( 3.5%)

The new thing is the formatting of the percentages: we don't multiply by 100 and we don't use f (say, above. .1f), but %. Specifically, we wrote {carriage / sk\_carriage:>5.1%}. We right-aligned the quotient carriage / sk\_carriage, outputting it to five places with one decimal place, as a percentage. This means that Python will multiply the value by 100 and add the % sign.

Incidentally, notice that we have given short names to the variables voz, raz and vis, and similar, also short, names to the variables total number of trips, length and distance. Even with such short names, the line is already quite long.

The output already looks quite nominal; it will look even more so if we add column names.

```

sk_voz = sum(voznje.values())
sk_raz = sum(razdalje.values())
sk_vis = sum(visine.values())

print()
print(f"{'Kolo':>6}{{'Število voženj':>19}{{'Razdalja':>19}{{'Višina':>19}}}")
print("-" * (6 + 3 * 19))
for kolo in voznje:
    voz, raz, vis = voznje[kolo], razdalje[kolo], visine[kolo]
    print(f"{kolo:12}{voz:5} ({voz / sk_voz:>5.1%}) {raz:10} ({raz / sk_raz:>5.1%}) {vis:10} ({vis / sk_vis:>5.1%}) ")
print("-" * (6 + 3 * 19))

```

Kolo	Število voženj	Razdalja	Višina
Canyon	26 (26.0%)	2766 (39.6%)	26392 (27.0%)
Cube	43 (43.0%)	3174 (45.4%)	66705 (68.3%)
Nakamura	22 (22.0%)	439 ( 6.3%)	1119 ( 1.1%)
Stevens	9 ( 9.0%)	607 ( 8.7%)	3395 ( 3.5%)

It's so beautiful, it's a bit kitsch.

## 07b More on reading files

We have been reading files for a long time, since the second week. What I failed to tell you the other day is that files also have methods. Simply because we haven't come across methods before and there was no need to have them right next to files. Now we can. There are three or four of them, they are uncomplicated and, well, they are not necessarily very useful either. That is why I have kept them from you so far.

We've always read files with a for loop (until we learned about `csv.reader` and `csv.DictReader`, and we'll get to that).

```
for vrstica in open("kolesa-z-glavo.txt"):
    print(vrstica)
```

kolo,razdalja,višina,lastnik,leto nakupa

Cube,5031,159,Janez,2017

Stevens,3819,1284,Ana,2012

Focus,3823,1921,Benjamin,2019

The file has, as written, methods.

```
f = open("kolesa-z-glavo.txt")
```

One is `read()`; this reads the whole file into a string.

```
f.read()
```

Python

```
'kolo,razdalja,višina,lastnik,leto nakupa\nCube,5031,159,Janez,2017\nStevens,3819,1284,Ana,2012\nFocus,3823,1921,Benjamin,2019'
```

You only have to do it once. The file is now "exhausted".

```
f.read()
```

..

If we want to do anything else with it, we'll have to reopen it. (There is a way to "roll it back", but we won't do that.)

```
f = open("kolesa-z-glavo.txt")
```

The second is `readlines()`; this reads all the lines of the file into a list of strings.

```
f = open("kolesa-z-glavo.txt")
f.readlines()
```

```
['kolo,razdalja,višina,lastnik,leto nakupa\n',
 'Cube,5031,159,Janez,2017\n',
 'Stevens,3819,1284,Ana,2012\n',
 'Focus,3823,1921,Benjamin,2019']
```

The third is `readline()`. This reads one line.

```
f = open("kolesa-z-glavo.txt")
f.readline()
```

```
'kolo,razdalja,višina,lastnik,leto nakupa\n'
```

One after the other.

```
f.readline()
```

```
'Cube,5031,159,Janez,2017\n'
```

```
f.readline()
```

```
'Stevens,3819,1284,Ana,2012\n'
```

What is read is read and is not returned.

Of these three, you will most often need the last (`readline()`), and sometimes the first (`read()`). If you are interested in using `readlines()`, you will almost certainly be doing yourself a favour by looping over the file.

We'll use `readline()` when the file doesn't have a "homogeneous" format: different lines have different meanings, say, and it's practical for us to read them one by one.

"What is read is read," I wrote with this example in mind:

```
f = open("kolesa-z-glavo.txt")

stolpci = f.readline().split(",")
for vrstica in f:
    print(vrstica)
```

```
Cube,5031,159,Janez,2017
```

```
Stevens,3819,1284,Ana,2012
```

```
Focus,3823,1921,Benjamin,2019
```

We read the first line in the `column`; maybe it comes in handy, maybe we just wanted to get rid of it. The rest of the lines are read with `for`. The first one won't read to us because it's already read.

The fourth method of note is `close`.



```
f = open("kolesa-z-glavo.txt")
f.readline()
```

```
'kolo,razdalja,višina,lastnik,leto nakupa\n'
```

```
f.readline()
```

```
'Cube,5031,159,Janez,2017\n'
```

```
f.close()
```

```
f.readline()
```

```
-----
ValueError                                Traceback (most recent call last)
Cell In[19], <a href='vscode-notebook-cell:?execution_count=19&line=1'>line 1</a>
----> <a href='vscode-notebook-cell:?execution_count=19&line=1'>1</a> f.readline()

ValueError: I/O operation on closed file.
```

Once the file is closed, it can no longer be read :)

Why would anyone close a file? Keep it open! First of all: a program on MS Windows can only have 512 files open at the same time. I don't know about other operating systems. There used to be a limit on the number of files that could be opened at the same time in all programs together ... OK, do we care? Not really.

Python closes files by itself.

```
f = open("kolesa-z-glavo.txt")
f.readline()

f = open("kolesa.txt")
```

Since `f` now refers to another file and we can't read from the first one anyway, Python closes it.

Even more common is the situation when a file is opened inside a function. When the function is over, the file can no longer be accessed and Python closes it. In short: you won't call `close` very often.

## ## Writing files

After learning how to format strings, we printed out a very nice table.

```

voznje = {"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(",")
    voznje[kolo] += 1
    razdalje[kolo] += int(razdalja)
    visine[kolo] += int(visina)

sk_voz = sum(voznje.values())
sk_raz = sum(razdalje.values())
sk_vis = sum(visine.values())

print()
print(f'{"Kolo":6}{ "Število voženj":>19}{ "Razdalja":>19}{ "Višina":>19}')
print("-" * (6 + 3 * 19))
for kolo in voznje:
    voz, raz, vis = voznje[kolo], razdalje[kolo], visine[kolo]
    print(f'{"kolo:12}{voz:5} ({{voz / sk_voz:>5.1%}}) {raz:10} ({{raz / sk_raz:>5.1%}}) {vis:10} ({{vis / sk_vis:>5.1%}})')
    print("-" * (6 + 3 * 19))

```

Kolo	Število voženj	Razdalja	Višina
Canyon	26 (26.0%)	2766 (39.6%)	26392 (27.0%)
Cube	43 (43.0%)	3174 (45.4%)	66705 (68.3%)
Nakamura	22 (22.0%)	439 (6.3%)	1119 (1.1%)
Stevens	9 (9.0%)	607 (8.7%)	3395 (3.5%)

It's so beautiful that you could just write it in a file. The file should be called table-wheels.txt.

Let's try it.

```

f = open("tabelica-kolesa.txt")

-----
FileNotFoundError                                Traceback (most recent call last)
Cell In[23], <a href='vscode-notebook-cell:?execution_count=23&line=1'>line 1</a>
----> <a href='vscode-notebook-cell:?execution_count=23&line=1'>1</a> f = open("tabelica-kolesa.txt")

File ~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:286, in _modified_open(file, *args, **kwargs)
   <a href='~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:279'>279</a> if file in {0, 1, 2}:
   <a href='~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:280'>280</a>     raise ValueError(
   <a href='~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:281'>281</a>         f'IPython won't let you open fd={file} by default "
   <a href='~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:282'>282</a>         "as it is likely to crash IPython. If you know what you are doing, "
   <a href='~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:283'>283</a>         "you can use builtins' open."
   <a href='~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:284'>284</a>     )
--> <a href='~/opt/miniconda3/envs/prog/lib/python3.11/site-packages/IPython/core/interactiveshell.py:286'>286</a> return io_open(file, *args, **kwargs)

FileNotFoundError: [Errno 2] No such file or directory: 'tabelica-kolesa.txt'

```

Of course not. There is no file; there is nothing to open. Of course not: we would like to make a file.

We need to give the `open` function one more argument - the way we want to open the file. This can be `r` (read), `w` (write), or `a` (append to an existing file, as `\*append\*`). (We could add another `t` to say that it is a text file, not a binary file, `b`. We are not interested in that for this object.)

```

● f = open("tabelica-kolesa.txt", "w")

```

As `f` is a file, it has the usual file methods.

```

f.readline()

```

```

-----
UnsupportedOperation                                Traceback (most recent call last)
Cell In[25], <a href='vscode-notebook-cell:?execution_count=25&line=1'>line 1</a>
----> <a href='vscode-notebook-cell:?execution_count=25&line=1'>1</a> f.readline()

UnsupportedOperation: not readable

```

Of course not. We write to this file. Nothing that starts with `read` will work. Instead, `write` works for it (which we didn't even show before, when we were still reading, because - you guessed it - it wouldn't work, just like reading doesn't work here).

```
f.write(f'{"Kolo":6}{{"Število voženj":>19}{{"Razdalja":>19}{{"Višina":>19}}\n')
f.write("-" * (6 + 3 * 19) + "\n")
for kolo in voznje:
    voz, raz, vis = voznje[kolo], razdalje[kolo], visine[kolo]
    f.write(f'{"kolo:12}{voz:5} ({voz / sk_voz:>5.1%}) {raz:10} ({raz / sk_raz:>5.1%}) {vis:10} ({vis / sk_vis:>5.1%})\n')
f.write("-" * (6 + 3 * 19) + "\n")
```

64

Everything is the same as before, but

- Replace `print` with `f.write`,
- add `\n` to the end of each line.

The `print` went to the new line itself, the `write` will not.

There are a couple of differences between `print` and `write`:

- `write` does not go to a new line by itself,
- `print` accepts multiple arguments, and they can be of any type, from strings to numbers to lists to sets to anything. `write` accepts a single argument and it must be a string.
- `print` does not return a result (it returns `None`), `write` returns the number of bytes written in this call. Why? I have no idea. But it's the `64` that's printed at the end: it's  $6 + 3 * 19 = 63$  minuses and one `\n`.

If we now open the `tables-wheels.txt` file, we see that it is ... completely empty. So where did all those `write`s go? Including the last 64?

The computer doesn't write everything to the files all the time, it only writes it as it accumulates. Otherwise it would have to access the disk continuously, line by line; even in the era of SSDs this would be slow. And here ... it just hasn't built up yet. Right, but now we're done. How to tell Python it's done? Simple: close the file. :)

```
f.close()
```

(There is another way: `f.flush()` writes what has accumulated so far, but does not close the file.)

Does this mean that when we write, we will always call `close()` to write the contents of the file? No. I don't usually call it. But that's why we talked about when Python closes a file when we were reading. If you write it inside a function, at the end of the function, when the local variable (say `f`) disappears, the file itself will be closed.

## ## Opening files correctly

I have never given this lecture before, but maybe it is a good time to start. For the last ten years or so, Python files should be opened like this:

```
with open("tabela-kolesa.txt", "w") as f:
    f.write(f'{"Kolo":6}{{"Število voženj":>19}{{"Razdalja":>19}{{"Višina":>19}}\n')
    f.write("-" * (6 + 3 * 19) + "\n")
    for kolo in voznje:
        voz, raz, vis = voznje[kolo], razdalje[kolo], visine[kolo]
        f.write(f'{"kolo:12}{voz:5} ({voz / sk_voz:>5.1%}) {raz:10} ({raz / sk_raz:>5.1%}) {vis:10} ({vis / sk_vis:>5.1%})\n')
    f.write("-" * (6 + 3 * 19) + "\n")
```

The ``with`` statement works with things that exist within a block (the official name for this is "context"), and when the execution of the block finishes, they want to be notified because they have some "finishing work" to do. A file is already such a thing: it is opened at the beginning of a block, and when the block is over, it is notified and "closes" itself. For ``time f``, we assign the variable ``f`` to the result of the ``open`` function.

The same applies to reading: even ``csv.reader`` should be officially called this way:

```
...
```

```
with open("wheels.txt") as f:
```

```
    for lines in csv.reader(f):
```

```
        ... and so on
```

```
..
```

Think what you want. Understand if you want. You can open files with the normal ``open`` call and not close them (as we have been doing since the beginning of the semester), or you can close them with ``close``. Ignore what it says in this section. Almost certainly nothing will happen to you. Or you can use ``with``.