

# Lecture 7 – extra

## Why are we learning to write "oneliners"?! Isn't that nasty?!

This topic will leave the - wrong! - impression that we are learning to write functions in one line. But does this exhibitionism really belong in Programming 1?! Isn't it actually nasty?

It doesn't. And it can be nasty.

Some students are really keen on this sport, the challenge. "But is it possible to make an oneliner for this too?" is quite a common question. Yes. Everything is possible. But it is usually without connection. Then, once you know how, it's often not even a challenge anymore. It's like asking yourself if it's possible to walk to Paris. Yeah, I just don't know why I would do it.

Why then are we learning to "write oneliners"?

We don't. We learn to write generators, iterators, derived lists, sets and dictionaries. Why? It's short and concise, perfectly understandable and faster - if you use common sense. So definitely something useful to know. But if we force it when we don't need to, we will of course write code that is longer, incomprehensible and slow.

So what we are going to do today is another tool that is available to us, and it must be used, like all tools, at the right time.

The second purpose of this is to get used to a different way of thinking about programming. So far, we have looked at programs - especially the code that compiles a list, a dictionary, a set - *in a procedural way*. We've described *the steps of* building a list, a dictionary, a set -- each element is computed separately and *added* with `append`, `add` or whatever. Today's programs will be *declarative*: instead of telling you how to build a list, we'll just tell you what it should contain. The code won't even be that different, it will just be reversed a bit. What will have to be completely different is the way we think when we write and read.

Apart from these two reasons, writing functions in one line is of course also a fun sport - even when the result is not suitable for production code. :)

## "Declaratory record"

You already know what we are going to do. In mathematics, we sometimes describe a set by listing its elements.

$$S = \{1, 2, 3\}$$

Sometimes we just describe their properties (with some care [not to get Russell on us](#)). We would describe the set of multiples of all natural numbers whose square is less than 120 as follows:

$$S = \{2x; \forall x \in \mathbb{N} \wedge x^2 < 120\}.$$

## Derived lists

Let's start with what we have known for the longest: lists. So far, we have written them by listing their elements in square brackets,

```
k = [9, 16, 81, 25, 196]
```

We will continue to use parentheses, but instead of listing the elements inside them, we will write the expression that makes them up. Suppose I wanted to list the roots of the numbers in the list above (by some strange coincidence, the list above contains exactly the squares). I would do this:

```
from math import sqrt  
[sqrt(x) for x in k]
```

So: I wrote square brackets, as I always do when defining a list, but instead of listing the elements, I said what they should be. I said that the elements of my new list should be the *square root of x*, where *x* is the elements of the list *k*. What if I wanted to have squares of numbers instead? Same schmorn:

```
[x ** 2 for x in k]
```

The form of what we write is always [a for expression variable and something], where *the variable* is the name of some variable (e.g. *x*, the *expression* is some expression that (usually, but not necessarily) contains that variable (e.g. `sqrt(x)` or `x ** 2`), and the *something* is something over which a for loop can be passed, i.e. a list, a dictionary, a set, or something we don't know yet but will learn about today.

So when we construct such a *derived list*, we have to think:

- what do we want in this list? What term do I use to describe (calculate) an element?
- what (which list, range-a...) do I have to go through to get these elements.

Let's look at some more examples.

I have a list of names, say,

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

How would you calculate the average length of a name? I have a sum function that can add numbers in a list. So I just need a list of lengths (instead of a list of names). Let's look at the two "points" above. What do I want in the list? The lengths of the names. How do I get them? Obviously by some `len(name)`, where a *name* is just any name in the name list. In this sentence we have answered both questions -- what am I going to compute (`len(name)`) and what am I going to loop over (over names). So we know: we need

```
[len(ime) for ime in imena]
```

Now I just add this up and divide by the length of the list.

```
sum([len(ime) for ime in imena]) / len(imena)
```

The average length of the name is not particularly interesting. Let's calculate the average weight instead and, to avoid boredom, we will calculate it from these triples (weight, name, is\_female):

```
podatki = [  
    (74, "Anže", False),  
    (82, "Benjamin", False),  
    (58, "Cilka", True),  
    (66, "Dani", False),  
    (61, "Eva", True),  
    (84, "Franc", False)]
```

Here's how to do it: loop over the data, calculate the sum of the first elements and divide by the length of the list.

First, let's see how we would have done it the old-fashioned way. Please, not like this:

```
vsota = 0  
for i in range(len(podatki)):  
    vsota += podatki[i][0]  
  
print(vsota / len(podatki))
```

This is better:

```
vsota = 0  
for podatek in podatki:  
    vsota += podatek[0]  
  
print(vsota / len(podatki))
```

If you want your professor to have a lasting good memory of you, this is what you will do:

```
vsota = 0  
for teza, ime, zenska in podatki:  
    vsota += teza  
  
print(vsota / len(podatki))
```

Now let's do as we have learned today. We can do

```
vsota = sum([podatek[0] for podatek in podatki])  
print(vsota / len(podatki))
```

or, better, the latest version

```
vsota = sum([teza for teza, ime, zenska in podatki])
print(vsota / len(podatki))
```

## Derived lists and conditions

When deriving lists, you can add a condition to select the elements to be added.

How would you make a list of squares of all odd numbers up to 100? A list of squares of all numbers up to 100 is trivial, all we need to do is write:

```
[x ** 2 for x in range(10)]
```

If you want to pick only odd numbers, you can add a condition to the list derivation.

```
[x ** 2 for x in range(10) if x % 2 == 1]
```

This is not really necessary, we could simply say

```
s = [x ** 2 for x in range(1, 10, 2)]
```

What if you want a list of all numbers up to 30 that are not divisible by 7 and do not contain the digit 7?

```
[x for x in range(30) if x % 7 != 0 and "7" not in str(x)]
```

Out[16]:

```
[1,
2,
3,
4,
5,
6,
8,
9,
10,
11,
12,
13,
15,
16,
18,
19,
20,
22,
23,
24,
25,
26,
29]
```

The first part of the condition ensures that the number is not divisible by 7 (the remainder after division by 7 must be different from 0). The second part ensures that the number does not contain 7: simply convert the number to a string and check that it does not contain 7.

How would you calculate the sum of the weights of the men in the above list of data? We add a condition to the loop. To start with, let's just make a list of the names of all the men.

```
[ime for teza, ime, zenska in podatki if not zenska]
```

Out[17]:

```
['Anže', 'Benjamin', 'Dani', 'Franc']
```

In the definition of our list, we add an if at the end, after the for, and a condition that must be satisfied by the element that we are interested in.

We also pick up the weights in the same way.

```
[teza for teza, ime, zenska in podatki if not zenska]
```

```
Out[18]:  
[74, 82, 66, 84]
```

Add the elements of this list with sum, and we have.

```
sum([teza for teza, ime, zenska in podatki if not zenska])
```

```
Out[19]:  
306
```

Now you know why I'm so tired of not using for i in range(len(s)): because with it, instead of a nice, short, transparent [thesis for thesis, name, woman and data if not woman], we get [data[i][0] for i in range(len(data)) if not data[i][2]]. Of course, you can also work the other way if you like. We live in freedom; no law forbids masochism.

Let us make a list of all the factors of a given number n. Take, say, n = 60. The list of divisors is then the list of all x, where x comes from the interval  $1 \leq x \leq n$ , for which the remainder after dividing n by x is 0 (i.e. x divides n).

```
def delitelji(n):  
    return [x for x in range(1, n + 1) if n % x == 0]
```

Now we can quickly see if a given number is perfect: perfect numbers are (remember?) numbers that are equal to the sum of their divisors (excluding themselves).

```
def popolno(n):  
    return n == sum(delitelji(n)) - n
```

Even more remarkable are prime numbers, numbers without divisors. That is, those that are divisible only by 1 and by themselves.

```
def prastevilo(n):  
    return delitelji(n) == [1, n]
```

If the sharers didn't have the function yet - no big deal. You would still find out whether a number is a prime number in a single line. Moreover, we could only go from 2 to n (without n); so we would leave out 1 and n and declare a number prime if it is divisible only by 1 and by itself.

```
def prastevilo(n):  
    return [x for x in range(2, n) if n % x == 0] == []
```

Of course, it is simpler to change the bounds in the ranges so that they do not include 1 and n.

```
def prastevilo(n):  
    return [x for x in range(2, n) if n % x == 0] == []
```

Better still, we should remember that empty lists are untrue.

```
def prastevilo(n):  
    return not [x for x in range(2, n) if n % x == 0]
```

For an encore, let's add something we won't explain.

```
def prastevilo(n):  
    return all(n % x for x in range(2, n))
```

This function - in any form - can be used to quickly list all primes between 2 and 100:

```
[n for n in range(2, 101) if prastevilo(n)]
```

Out[27]:

```
[2,  
3,  
5,  
7,  
11,  
13,  
17,  
19,  
23,  
29,  
31,  
37,  
41,  
43,  
47,  
53,  
59,  
61,  
67,  
71,  
73,  
79,  
83,  
  
89,  
97]
```

Of course, we can do without a function; we just write what the function does into the condition.

```
[n for n in range(2, 101) if not [x for x in range(2, n) if n % x == 0]]
```

Out[28]:

```
[2,  
3,  
5,  
7,  
11,  
13,  
17,  
19,  
23,  
29,  
31,  
37,  
41,  
43,  
47,  
53,  
59,  
61,  
67,  
71,  
73,  
79,  
83,  
89,  
97]
```

We prefer not to write such things, as they quickly become obscure.

## General pattern

In general: any piece of code that looks like this

```
r = []  
for e in s:  
    if pogoj(e):  
        r.append(izraz)
```

lahko prepišemo v

```
r = [izraz for e in s if pogoj(e)]
```

## Multiplication

Sets are constructed exactly like lists, except that instead of square brackets, we use curly brackets.

This gives all divisors 60

```
{i for i in range(1, 61) if 60 % i == 0}
```

Out[29]:

```
{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60}
```

## Dictionaries

Same thing. If we want to make a dictionary that contains numbers up to 10 as keys and their squares as values, we write

```
{i: i ** 2 for i in range(11)}
```

Out[30]:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

## Some useful features

These things really come to life when used in combination with a few features. Some of them we already know, some of them will only make sense today.

### enumerate

So far, I've been annoyed that we use it when we need both the index of an element and its value. Now it will be even more useful, as it will significantly improve the clarity of derived lists, sets and dictionaries.

Now that we are just as clever, we also know how to write enumerates if they did not already exist.

```
def enumerate(s):  
    i = 0  
    for x in s:  
        yield i, x  
        i += 1  
  
list(enumerate("Ana"))
```

```
Out[31]:  
[(0, 'A'), (1, 'n'), (2, 'a')]
```

### zip

We already know about zip: we give it some lists and it returns a new list - a list of targets with the elements of the lists we sent as arguments.

Well, as of today we know: it actually returns a pair generator, not a list of pairs.

```
zip("Benjamin", "Margareta")
```

```
Out[32]:  
<zip at 0x7ff2830931e0>
```

Ignore those 0x10193c308(or whatever is displayed) - the printout said we made a zipand not a list. Of course we can "generate" a list from it.

```
list(zip("Benjamin", "Margareta"))
```

```
Out[33]:  
[('B', 'M'),  
 ('e', 'a'),  
 ('n', 'r'),  
 ('j', 'g'),  
 ('a', 'a'),  
 ('m', 'r'),  
 ('i', 'e'),  
 ('n', 't')]
```

Let's write a function that tells how many letters match the given two words. For example, MELODY and EMPLOYER match in six letters.



MELODY

EMPLOYER

EMPLOYEE

To make our work easier, let us first remember that `True` behaves as if it were 1 and `False` as if it were 0.

```
True + 7
```

```
Out[34]:  
8
```

Take, for example, the words `MELODY` and `EMPLOYER`.

```
b1 = "MELODIJA"  
b2 = "DELODAJALEC"
```

Combine them into a list of letter pairs (but only add the leaf in the printout to see what we've actually sown).

```
list(zip(b1, b2))
```

```
Out[36]:  
[('M', 'D'),  
 ('E', 'E'),  
 ('L', 'L'),  
 ('O', 'O'),  
 ('D', 'D'),  
 ('I', 'A'),  
 ('J', 'J'),  
 ('A', 'A')]
```

Let's loop over the list of letter pairs and create a new list containing `True` if the letters are the same and `False` if they are different.

```
[c == d for c, d in zip(b1, b2)]
```

```
Out[37]:  
[False, True, True, True, True, False, True, True]
```

Now let's count how many `True`'s there are. How? If `False` is the same as 0 and `True` is the same as 1, then we simply calculate the sum of the elements of the list.

```
sum(c == d for c, d in zip(b1, b2))
```

```
Out[38]:  
6
```

So if you wanted to write a function that tells you how many letters the two words match, you would say

```
def ujemanj(b1, b2):  
    return sum(c==d for c, d in zip(b1, b2))
```

Let's write a function that calculates the Euclidean distance between two points whose coordinates are represented by a list. Let's have

```
a = [2, 1, -3]  
b = [3, 5, 4]
```

Sometimes we would solve the problem like this:

```
def euclid(a, b):  
    s = 0  
    for x, y in zip(a, b):  
        s += (x - y) ** 2  
    return sqrt(s)
```

Now we know how to do it much, much more simply. Can we make a list of pairs? We can.

```
[(x, y) for x, y in zip(a, b)]
```

```
Out[42]:  
[(2, 3), (1, 5), (-3, 4)]
```

Can we make a list of pair differences? We can.

```
[x - y for x, y in zip(a, b)]
```

```
Out[43]:  
[-1, -4, -7]
```

Actually, we need a list of squares of the differences.

```
[(x - y) ** 2 for x, y in zip(a, b)]
```

And now we just need to add it all up.

```
sum((x - y) ** 2 for x, y in zip(a, b))
```

```
Out[44]:  
66
```

The function that calculates the Euclidean distance is

```
def euclid(a, b):  
    return sqrt(sum((x - y) ** 2 for x, y in zip(a, b)))
```

Once you get used to it, these things are eminently readable. The program is an almost verbatim natural language description: the Euclidean distance is the square root of the sum of the squares of the differences of the elements taken "in parallel" from the two lists.

## All and any

The functions `all` and `any` look trivial, but in fact they are very useful. The first, `all`, receives a generator (or, of course, a list, a set, a string, a dictionary, etc.) and returns `True` if it generates true things (say, `True` itself).

A number `n` is prime if all remainders after division by `i` (for each `i` from 2 to `n`) are different from 0. With `all`, this translates almost literally into Python.

```
def prastevilo(n):  
    return all(n % i != 0 for i in range(2, n))
```

The `any` function returns `True` if what is passed as an argument contains at least some true value. Let's show how it works with inverse prime numbers: a number is composite if the remainder after division by some other number is 0.

```
def sestavljeno(n):  
    return any(n % i == 0 for i in range(2, n))
```

Incidentally, we notice that `all` and `any` are linked by de Morgan's rule, which we should remember from mathematics. A number is prime if it is not composite,

```
def prastevilo(n):  
    return not sestavljeno(n)
```

If, instead of calling `compound`, we insert the function code `compound`, we get

```
def prastevilo(n):  
    return not any(n % i == 0 for i in range(2, n))
```

If we just reverse the condition, we see that

```
not any(n % i == 0 for i in range(2, n))
```

same as

```
all(not n % i == 0 for i in range(2, n))
```

## itertools.count

There is even more sugar in the `itertools` module.

The `count` function of `itertools` is like a range, which can be given at most a lower bound or not yet a lower bound. `count()` generates all numbers from 0 to infinity, and `count(n)` generates all numbers from `n` onwards.

I use it most often with `zip`. `zip(count(), s)` is the same as `enumerate(s)`. That's not very useful - obviously `enumerate(s)` is simpler and clearer than `zip(count(), s)`. What happens is that you zip over three lists at once, and you need indexes. This can be written as

```
for i, (x, y, z) in enumerate(zip(s, t, u)):
```

or we can write

```
for i, x, y, z in zip(count(), s, t, u)
```

And here's where it will come in handy, if you remember it

### **itertools.chain**

If we have several generators (or lists, dictionaries, etc.) and we want to loop through them all, one by one, we build them with chain.

```
from itertools import chain

for c in chain("Ana", "Berta", "Cilka"):
    print(c)
```

A  
n  
a  
B  
e  
r  
t  
a  
C  
i  
l  
k  
a

Of course, this is the same as for c and "Anna" + "Berta" + "Cilka";, but the beauty is that we haven't added the strings, it's a chain. This is nice especially when we're not chaining strings, but something that can't be added.

### **Itertools**

Anyone who liked this will also enjoy the other [goodies from the itertools module](#).