

# 05 Functions

## 0.1 Functions

You have noticed - and it gets on your nerves a little, I would say - that we keep repeating the same things.

Let's say: you have to keep looking for the key of the list that has the highest value. In other words, we have a dictionary whose keys are objects and their price values, and the task wants us to find the most expensive object.

```
cene = {
    'slika': 45,
    'pozlačen dežnik': 29,
    'Meldrumove vaze': 78,
    'skodelice': 83,
    'kip': 107,
    'čajnik': 15,
    'srebrn jedilni servis': 63,
    'perzijska preproga': 21}
```

And then, time after time, we kill ourselves with:

```
max_k = max_v = None
for k, v in cene.items():
    if max_v is None or v > max_v:
        max_v = v
        max_k = k

print(max_k)
```

kip

I apologise for the short variable names, but I really have a lot of this.

I have good news and two bad news. The good news: there is a function `argmax` which is given a dictionary and returns the key that belongs to the largest value.

The first bad news: the good news is not true. We don't give the `argmax` function a dictionary, we give it a list of pairs. The function compares the other elements of the pairs, finds the largest one and returns the corresponding first element. But this bad news is not so fatal: we just pass `d.items()` to the function (if `d` is our dictionary).

This will give `[('painting', 45), ('gilded umbrella', 29), ('Meldrum vases', 78),`

`('cups', 83), ('statue', 107), ('teapot', 15), ('silver cutlery', 63),`

`('Persian carpet', 21)]` and returned `'statue'`, since this is the first element to which the largest second element belongs. Second bad news: the good news is not true at all. There is no such feature. Fake news.

I always like to quote the unforgettable high school mathematics professor Františ Oblak: "What is not there is there." So let's do a function like this.

Let's look at cell 2, where we calculated our famous maximum. It says everything this function should do. Of course, we want it to look up the maximum in any dictionary, not necessarily in prices. In addition, we will agree that what it looks for the maximum in will not be a dictionary but a list of pairs. So we will have something like this:

```

max_k = max_v = None

for k, v in s:
    if max_v is None or v > max_v:
        max_v = v
        max_k = k

```

This is just a rewrite of the above cell, except I replaced `prices.items()` with `s`. This `s` will be what the function took as an argument.

What is this saying `s` will be what the function got as an argument?! Well, anyway, if we call `argmax(prices.items())`, the function `s` will contain `prices.items()`.

Obviously, there are (at least) two more things to do. Three, actually. Four. (This is becoming like the Spanish Inquisition)!

1. These lines need to be "grouped" somehow, telling Python that this will be the code of a function.
2. The function needs to be given a name (`argmax`).
3. It needs to be told that it will require one argument.
4. We need to say under what name we will see (access...) the data given as argument inside the function.

All this is done practically in one line.

```

def argmax(s):
    max_k = max_v = None

    for k, v in s:
        if max_v is None or v > max_v:
            max_v = v
            max_k = k

```

1. The word `def` is used to say that the following is the definition of a ganz new function. (Today's young people don't even know proper Slovene anymore, so they need to be reminded that there are other, more Slovene words than totally.)
2. Follow the name of the function, i.e. `argmax`.
3. Then come the parentheses and as many names as the number of arguments the function requires. Put commas between the names (if there are more than one). If the function doesn't like arguments, we write brackets anyway, even if empty brackets look silly. There are all sorts of other complications, such as arguments that can be omitted and have a default value, or functions that take any number of arguments, functions that require you to name the arguments.

A beginner's course in programming, especially for students who are probably not going to be computer professionals, does not need all this.

4. What the function receives as argument(s) is seen under the names given in brackets. Our function has given one name, `s`, so it expects one thing as an argument, and within the function we will refer to that thing by the name `s`. What name that thing is known by outside is neither of interest to us, nor (unless we are really diligent and know Python well) can we find out.

And rightly so.

Let's see if it works!

```
print(argmax(cene.items()))
```

None

function returns? A function is populated by a bunch of variables - not only max\_v and max\_k but also k and v. Python cannot know which of them should be the result of the function. If any; there are also functions (say print) that return nothing. Yes, Python obviously imagined that this function would return nothing, but that's why it laughed at us with this None. So let's add

5. We need to say what the function returns.

and complete the function.

```
def argmax(s):  
    max_k = max_v = None  
  
    for k, v in s:  
        if max_v is None or v > max_v:  
            max_v = v  
            max_k = k  
    return max_k
```

In jo preskusiti.

```
print(argmax(cene.items()))
```

kip

What about the cups and the statue?

Some of them were probably worried before, some of them understand now, and some of them will be worried again. The task was not very well defined. What exactly was the function supposed to return? A string? Multiple strings? That is, what type of objects should they be?

The text of the task somehow implied that it would be multiple strings, but did not make it clear how these multiple strings were to be stored. Naively, we put the return in a loop and expected it to execute multiple times ... but Python will already do something with these multiple results. Well, it won't.

Three lessons.

1. When Python encounters a return, the execution of the function ends. The function returns what it returns and that's it. return has thus broken the loop.
2. Just as when we compiled a dictionary we decided to think first about what its keys and its values would be, so when we compile a function we need to think about what (for example: what type) its arguments and its result will be. Here we are not.
3. A function can also return other things, say lists.

The third lesson has not yet been learned, so let's just quickly fix the function.

```
def dragoceni(s):
    drag1 = []
    for k, v in s:
        if v > 70:
            drag1.append(k)
    return drag1

predmet1 = dragoceni(cene.items())

print(predmet1)

['Meldrumove vase', 'skodelice', 'kip']
```

Let's write another similar function with a similar doctrine: the `exist_precious` function, which returns True if a valuable object exists and False if it does not.

```
def obstajajo_dragoceni(s):
    drag1 = []
    for k, v in s:
        if v > 70:
            return True
    return False
```

Understood? This function has two returns. As we have seen, only one can happen - the one we encountered first. For each object, we check whether it is valuable. If it is, we return True. But when do we return False? If we have never returned True. That is, if there was no valuable object.

## 1.2 Functions calling each other

In the programs we have written so far, we have called various Python functions. We can call them in our own functions too. In the examples above, we didn't do much of that, we just called `append`, but probably nobody would raise an eyebrow if we also called `lazy` or `print` or whatever. But what about our functions? We can call our own functions in our functions. Yes, of course we can. We build larger programs as stacks of our own functions; we combine simpler functions in ever more complex ones, levels and levels high. There won't be anything very complicated about this subject, but we can see a simple example already here: the function `precious` is more general than the function `exist_precious`. The former returns a list of valuable objects, the latter just tells whether valuable objects exist. The second function could simply call the first and check whether it returns an empty list.

```
def obstajajo_dragoceni(s):
    # Ne programirajte tako!!!
    if dragoceni(s) != []:
        return True
    else:
        return False
```

Today's examples are rich indeed. Each shows a couple of things. This one at least two. Uh, three. The Inquisition.

1. You'll make programming easier if you think about what you already have and use it. What's more; sometimes you'll write two similar functions, and if you're clever, you'll extract what they have in common into a third function. So you will be able to shorten the two functions by calling what is common.

2. Caution: the function `exist_precious` has become slower! Previously, the for loop ran to the first precious object and ended there, broken by `return`. Now `exist_precious` does not contain any loop, but it does start a loop in `precious`, which always runs to the end, and it piles the objects into a list,

which is then discarded, because we only want to know if it is empty or not. Does that hurt us? It depends on the situation. If we analyse some short files like this, for our own needs, no problem. If the data were huge and if

we would call a function like `exist_valuable` many times, we would prefer to leave it as it was.

3. That's not how to program!

Let's continue with the third point. `precious(s) != []` is a Boolean expression whose result is already True or False. The function would more correctly (not in the sense of returning the correct result, but in the sense of, uh, making sense) read

```
def obstajajo_dragoceni(s):  
    return dragoceni(s) != []
```

Lesson 4: Functions can be very short. I have written many functions in my life that are only one line long. It's just that this line is often longer and more complicated than the one here.

### 1.3 Multiple arguments

The above functions do not take inflation into account. Today 80 is a lot, but tomorrow you won't be able to go to Hofer or Lidl without leaving at least that much. (For the archives: I'm writing this on 7 November 2023. We'll see how it ages.)

Let's write better functions `valuable` and `exist_valuable`: in addition to the list, let's give them a "valuable" limit. If we call `valuable(prices.items(), 70)` we will get the same result as now, if `valuable(prices.items(), 100)` we will just get a statue.

```
def dragoceni(s, meja):  
    drag1 = []  
    for k, v in s:  
        if v > meja:  
            drag1.append(k)  
    return drag1  
  
def obstajajo_dragoceni(s, meja):  
    return dragoceni(s, meja) != []
```

### 1.4 Arguments with default values

By the way, let's learn this: we often write functions that "hardly need" an argument. Suppose we were to call a function like `precious` in a hundred places in the program, and the limit would almost always be 70, and only in a few cases something else. In this case, we would set the `bounds` argument to the default value 70. If we omit it, it would have a value of 70.

This is easy to do.

```
: def dragoceni(s, meja=70):  
    drag1 = []  
    for k, v in s:  
        if v > meja:  
            drag1.append(k)  
    return drag1  
  
def obstajajo_dragoceni(s, meja=70):  
    return dragoceni(s, meja) != []
```

```
dragoceni(cene.items(), 100)
```

```
['kip']
```

```
dragoceni(cene.items())
```

```
['Meldrumove vase', 'skodelice', 'kip']
```

## 1.5 More results ... and no results

We already know what happens if a function doesn't have a return statement (or if it does, but it was never invoked during the execution of the function because it was, say, hidden inside an if whose condition was never true). In this case, the function returns None.

Functions always return exactly one thing. Not zero, not two, not five. One. That's bad. But not too much, because this rule can be easily flouted. First things first: if we return nothing, the function returns None. This returns (exactly) one thing, but it doesn't hurt us. Let it. In the same way, we can bend the rule that a function can return only one thing and return several.

```
def argminmax(s):
    min_k = min_v = None
    max_k = max_v = None

    for k, v in s:
        if min_v == None or v < min_v:
            min_v = v
            min_k = k
        if max_v == None or v > max_v:
            max_v = v
            max_k = k
    return min_k, max_k
```

```
najm, najv = argminmax(cene.items())

print(najm)
print(najv)
```

```
čajnik
kip
```

Do we understand what is happening?

min\_k, max\_k is in fact a target.

```
argminmax(cene.items())

('čajnik', 'kip')
```

When we were introduced to tuples, we mentioned that we are allowed to write them without brackets, but we rarely do so. Here is an example of when we always write tuples without parentheses. (This is again an unwritten rule: we could also add parentheses). We pretend (imagine) that return returned two things (even though it is really one, namely a terq with two elements).

When we call the function, we immediately unpack the result. And there we have it: functions that seem to return multiple values. This particular one, two. So the rule that functions always return exactly one thing can be easily flouted. They also seemingly return none or more. That there is in fact only one, nobody will notice.

## 05 while-loop

## 0.1 "The other loop"

The for loop can go through file lines, string characters, dictionary keys, list elements. And more. The characteristic of the for loop is that it "goes". It requires a sequence, and it will iterate over a block of code, assigning the next value (or values) in the sequence to a variable (or several) before each iteration.

There are situations where you want to repeat something, but not in a way that goes beyond "something". At least not through a sequence.

## 0.2 What Collatz assumed

Take any number and do this with it: if it is even, divide it by 2, if it is odd, multiply it by 3 and add 1. Repeat this until you get 1. For example, take 12. Since it is even, divide it by 2 and you get 6. 6 is even, so divide it by 2 to get 3. 3 is odd, so multiply it by 3 and add 1 - the result is 10. 10 is even, so divide it by 2 to get 5... The whole sequence is 12, 6, 3, 10, 5, 16, 8, 4, 2, 1. When we get to 1, we stop. Mathematicians, who always like to do useful things, beat their heads with the question of whether a theorem always runs out or whether it can ever even loop so that the sequence repeats and repeats indefinitely. Lothar Collatz, in particular, has assumed since 1937 that the sequence always runs out. As he says

biography, his conjecture ended in 1990, and mathematicians continue to make the conjecture. One of the most fearless mathematicians of the 20th century, couch surfer Erdos, said on the subject that mathematics is not yet ripe for such questions. Our goals will be: we will write a program for which the user will enter a number and the program will print out the sequence as above.

Let's put it in Slovene first:

```
stevilo = kar vpiše uporabnik
dokler stevilo != 1:
    če je stevilo sodo:
        deli stevilo z 2
    sicer:
        pomnoži stevilo s 3 in prištej 1
```

We can almost translate this into Python (it will actually be literal), the only thing missing is the line "as long as the number is even". This sounds suspiciously like "if the number is even", but instead of "if" we have "until". Let's try it.

```
n = int(input("Vnesi število: "))
while n != 1:
    print(n)
    if n % 2 == 0:
        n //= 2
    else:
        n = n * 3 + 1
print(1)
```

```
Vnesi število: 42
42
21
64
32
16
8
4
2
n 1
```

Please note that we have used integer division. If we had used ordinary division,  $n /= 2$ ,  $n$  would have turned into an infinite number (float) and the printout would have had awkward decimal places (each number would have been followed by a .0).

We need to make sure that we print out both the first and the last number, so we need another print outside the loop: after the loop, we print out the last member of the sequence, 1. Instead of `print(1)`, we could also write `print(n)` - we know that  $n$  is 1 anyway, otherwise the program wouldn't get to where it has got to, namely beyond the loop, which runs as long as  $n$  is different from 1.

The while loop is therefore similar to an if statement in its own way. While if statements are executed if the condition is true, while statements within the while loop are repeated as long as the condition is true.