**Content**

[1. Nadpis 1 2](#_Toc14110498)

1. Random questions
   1. Yield, generators, iterables in Python

**Iterables**

When you create a list, you can read its items one by one, and it’s called iteration. Everything you can use “for… in…” on is an iterable: lists, strings, files…

These iterables are handy because you can read them as much as you wish, but you store all the values in memory and it’s not always what you want when you have a lot of values.

**Generators**

Generators are iterators, but you can only iterate over them once. It’s because they do not store all the values in memory, they generate the values on the fly:

>>> mygenerator = (x\*x for x in range(3))

>>> for i in mygenerator:

... print(i)

0

1

It is just the same except you used () instead of []. BUT, you can not perform for i in mygenerator a second time since generators can only be used once: they calculate 0, then forget about it and calculate 1, and end calculating 4, one by one.

**Yield**

Yield is a keyword that is used like return, except the function will return a generator.

>>> def createGenerator():

... mylist = range(3)

... for i in mylist:

... yield i\*i

...

>>> mygenerator = createGenerator() # create a generator

>>> print(mygenerator) # mygenerator is an object!

<generator object createGenerator at 0xb7555c34>

>>> for i in mygenerator:

... print(i)

0

1

4

The first time the for calls the generator object created from your function, it will run the code in your function from the beginning until it hits yield, then it’ll return the first value of the loop. Then, each other call will run the loop you have written in the function one more time, and return the next value, until there is no value to return.

The generator is considered empty once the function runs but does not hit yield anymore. It can be because the loop had come to an end, or because you do not satisfy a “if/else” anymore.

* 1. TCP vs UDP
  2. OSI/ISO vs TCP/IP
  3. Testing
     1. Integration tests
  4. Maven vs Gradle