**Section 13: Python Generators**

**10.03.**

**100. Generators with Python**

Generators let us to create functions that can send a value and later to resume

Instead of return we’ll use yield

The sequence is generated over time, not at once in order to not store all the data in memory

A generator supports an iteration protocol

It generates a value and then waits until is called again

An example of generator is range()

Yield isn’t the same as return because we can have other commands after yield

We can use next(generator) to get the next value instead of using a for loop

An iterable is not an iterator, but we can use iter(iterable) to generate an iterator

On the generated iterator we can apply next

When the end of the generated sequence is reached a StopIteration exception is thrown

This is automatically handled by the for loop

<https://www.programiz.com/python-programming/generator>

A generator different when compared to a normal function because:

It has \_\_iter\_\_() and \_\_next\_\_() methods automatically implemented

Local variables and their states are remembered between successive calls

There are generator expressions that are written as list comprehensions

The syntax difference is that we use () instead of [] = generator comprehension

In this way we create a memory-efficient method to generate elements

<https://stackoverflow.com/questions/1756096/understanding-generators-in-python>

A generator is created once, but the code isn’t run completely at once

The execution is triggered by next and it goes until the next yield statement

<https://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do>

Over a generator you can iterate only once

When you call the function, the written code in the body won’t run

The end of the generator is when the code inside the function won’t execute an yield statement

A return in a generator can be viewed as rising a StopIteration