

# Iterators in Python

## Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

©Hans de Nivelles, 2025

## Iterators

**iterators** (also called **generators**) are central to Python, so you will have to understand them well.

They are used in `for` loops, and they can be used to construct containers like `list`, `queue`, `set` or `dict`.

Every container (including string) can be implicitly converted into an iterator.

Generators use a new programming language concept, that you are likely not familiar with.

## Iterators

You have already seen for:

```
for i in obj :  
    (do something with i)
```

It is possible to write a `for` loop with everything that can be converted into an iterator. This is the case for all containers, and to `range`.

Iterators are an essential part of Python. If you want to be a good Python programmer, you have to understand how they work.

They are implemented by means of an unusual language construction called **coroutine**.

## Functions vs. Coroutines

Iterators in Python are usually implemented by **coroutines**.

A coroutine is similar to a function (subroutine), but with the following difference:

When it ‘returns’ a value, it remembers its state. Next time it is called, it continues computation at the point where it stopped last time, so that it can possibly ‘return’ many values.

Computation continues until the coroutines falls over the end, or encounters a **return** statement.

## Fibers

Co-routines are a restricted form of **fibers**. (Outside of CS, fibers are used for producing ropes, reminding of threads.)

A group of processes are called **fibers** if:

1. At each moment, exactly one of them is running. The others are suspended.
2. The currently active fiber decides by itself when it passes control (yields) to one of the other fibers.

When it passes control to another fiber, it can also decide to terminate. After that, the fiber is removed from the group.

The difference with threads is that threads run in parallel in unpredictable ways. The OS decides.

It seems that fibers did not deliver, and they are not used any more.

## Coroutines

Coroutines are a restricted form of fibers. A coroutine can:

1. Create a new child coroutine.
2. Pass control to a child coroutine that it created, and pass a message.
3. Return control to the coroutine by which it was created (its parent), and pass a message. This is the yield operation.
4. Destroy itself. This will automatically destroy all of its children.

In addition, a coroutine consists of a single function. It can not return control to its creator from a subfunction (coroutines are **stackless**).

## Yielding 1,2,3

The following iterator yields 1, 2, 3:

```
def it123( ) :  
    yield 1  
    yield 2  
    yield 3
```

```
for nr in it123( ) :  
    print( nr )
```

Prints 1, 2, 3.

## Yielding 1,2,3

In order to better understand what is going on exactly, insert print statements:

```
def it123( ) :  
    print( 'a' )  
    yield 1  
    print( 'b' )  
    yield 2  
    print( 'c' )  
    yield 3  
    print( 'd' )  
  
for nr in it123( ) :  
    print( nr )
```

## Distinguishing Functions from Iterators

Functions and iterators are both defined with `def`. If the code contains a `yield` somewhere, it is an iterator. Otherwise, it is a function.

Iterators can also be constructed from classes, but we didn't cover classes yet.

## Fibonacci

Iterators can be finite or infinite. Below is an infinite iterator that yields the Fibonacci numbers:

```
def fib( ) :  
    f0,f1 = 0,1  
    while True:  
        yield f1  
        f0,f1 = f1, f0 + f1  
        # This is simultaneous assignment, using  
        # tuples.
```

To use this iterator, type :

```
for i in fib( ) :  
    print(i)    # needs to be stopped with ctrl-c
```

## Stopping a Generator

A generator can stop in two ways:

1. By reaching the end:

```
i = 0
while i < 10 :
    yield i
    i = i + 1
```

2. By a return statement:

```
i = 0
while True :
    if i >= 10 :
        return i
    yield i
    i = i + 1
```

## Range

Range is an iterator. It has three versions:

- `range(n)` : Integers from 0 up to (not including)  $n$ .
- `range(m,n)` : Integers from  $m$  up to (not including)  $n$ .
- `range(m,n,s)` : Integers  $m, m + s, m + 2s, m + 3s, \dots$  up to (not including)  $n$ .

`range` is easy to implement. The hard part is that there are three versions of it with the same name. We will later see how to deal with that.

## Use of Iterators

We have seen how to build iterators from co-routines. Now we explain how to use them:

Suppose that `obj` is **iterable**, which means that it can be transformed into an iterator (a container or co-routine):

1. Call `it = iter( obj )`. Now `it` is an iterator.
2. Call `next( it )` to get elements from the iterator. When `it` has no values left, it raises a `StopIteration`.

The mechanism for coroutines is bizarre. It either returns or falls over the end. The interpreter will raise `StopIteration`.

The coroutine cannot raise the `StopIteration` by itself. If you try, the interpreter will change `StopIteration` into `RuntimeError`.

Before Python 3.7, iterator code was allowed to raise `StopIteration` by itself. A lot of code had to be rewritten.

## Zip

It is sometimes useful to **zip** two iterators together. Every time when it is called, `zip( it1, it2 )` takes one object from `it1` and one object from `it2`, and yields them as a tuple.

The iterator stops as soon as either of `it1`, `it2` stops.

`zip` is defined in Python. If you want to implement it by yourself, it is tricky:

We are using two iterators `it1, it2`, while at the same we are an iterator by ourself.

`it1` and `it2` will raise `StopIteration` when they are done, which we have to catch, after which we must return.

Before Python 3.7, one could simply let the `StopIteration` 'fly through'.

## Zip (2)

```
def my_zip( iterable1, iterable2 ) :  
    it1 = iter( iterable1 )  
    it2 = iter( iterable2 )  
  
    while True:  
        try :  
            yield next( it1 ), next( it2 )  
        except StopIteration :  
            return
```

An example of its use:

```
for i in my_zip( range(1,4), range(20) ) :  
    print(i)
```

## File Reading

It is possible to transform an open file into an iterator (which will generate the lines in the file):

```
f = open( "test.py", "r" )
for s in f.readlines( ) :
    print( s )
f.close( )
```

If you write `for s in f.read( )` instead, the complete file is read into memory (as a string), after which the string is iterated.

## Making an Iterator from a User

```
def ask_user(s) :  
    while True :  
        x = input( f"{s} : " )  
        yield x  
  
def getnumber( ) :  
    for s in iter( ask_user( "please type a number  
                            between 1 and 100" )) :  
        if s. isdigit( ) :  
            x = int(s)  
            if x >= 1 and x <= 100 :  
                return x
```

## How for works

`for i in obj` first constructs an iterator from `obj` by calling `it = iter( obj )`.

After that, it keeps on calling `i = next( it )` until a `StopIteration` is raised.

The implementation is on the next slide:

## Implementation of for

```
for i in iterable :  
    (body of the for loop)
```

is implemented as follows:

```
it = iter( iterable )  
while True:  
    try:  
        i = next( it )  
    except StopIteration:  
        break # Out of the while.  
  
    (body of the for loop)
```

## Generator Expressions

Generator expressions are expressions that modify iterators.

The simplest form is:

`expr(containing var) for var in iterator`. It enumerates the value of the expression for every value generated by the iterator.

For example:

```
def iter( ) :  
    yield 1  
    yield 2  
    yield 3
```

```
for x in ( m * m for m in iter( ) ) :  
    print(x)
```

## Generator Expressions (2)

The `for`-s in generator expressions can be nested, the inner one runs faster:

```
( (m,n) for m in iter( ) for n in iter( ))
```

The expressions of later `for`-s may use the variables generated by the `for`-s before it:

```
lst = [ [1,2], [3,4], [4,5] ]  
for s in ( n for m in lst for n in m ) :  
    print(s)
```

Once more: Generator objects do not contain their elements, they generate them when asked for it.

## Generator Expressions (3)

Remember that iterators can be used for constructing lists, tuple, dictionaries, sets:

```
l = list( (m,n) for m in range(4) for n in range(5) )
```

```
lst = [ [1,2], [3,4], [4,5] ]
```

```
s = set( n for m in lst for n in m )
```