

JOHNS HOPKINS

WHITING SCHOOL  
of ENGINEERING

# Johns Hopkins Engineering

## **Module 9: Object Oriented Programming II**

### **Python Pointer: Generators and Yield**

EN605.206: Introduction to Programming in Python

# Generator Functions (Generators)

A **generator** is a function that behaves like an iterator. Each element is "generated" by a **lazy iterator** and control is yielded back to the caller. Generators are of particular benefit over functions when you're dealing with a potentially large, or even infinite, amount of data. A generator is created using the `yield` keyword.



Do not attempt!

```
main.py
1 def sequence(N):
2     numbers = []
3
4     for i in range(N):
5         numbers.append(i)
6
7     return numbers
8
9 N = 40000000
10
11 for i in sequence(N):
12     print(i)
```

Function: not enough memory to store and return 40 million int's

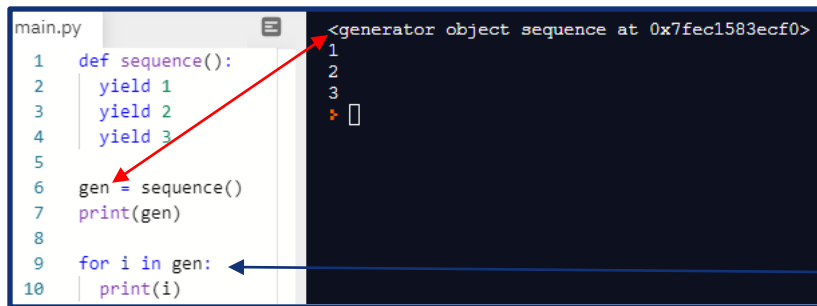
```
main.py
1 def sequence(N):
2     for i in range(N):
3         yield i
4
5 N = 40000000
6
7 for i in sequence(N):
8     print(i)
```



Generator: "generate" one value at-a-time, passing control back to the call each time

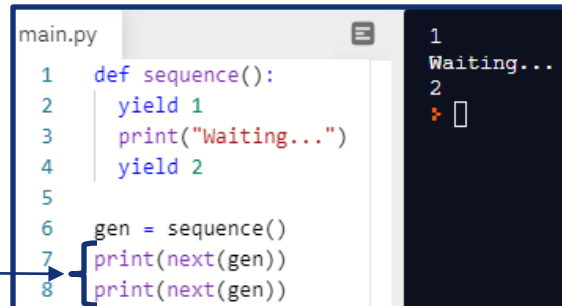
# yield

To create a generator, we use the `yield` keyword instead of `return`. In a function we compute some result in its entirety then `return` that result, as well as execution control, back to the caller. In a generator, however, we "generate" (compute) the next value only and `yield` it to the caller. So we give control flow back, but only until the caller wants to retrieve the next item in the generator.



```
main.py
1 def sequence():
2     yield 1
3     yield 2
4     yield 3
5
6 gen = sequence()
7 print(gen)
8
9 for i in gen:
10    print(i)
```

```
<generator object sequence at 0x7fec1583ecf0>
1
2
3
4 []
```



```
main.py
1 def sequence():
2     yield 1
3     print("Waiting...")
4     yield 2
5
6 gen = sequence()
7 print(next(gen))
8 print(next(gen))
```

```
1
Waiting...
2
3 []
```

# Generator Function Example

The diagram illustrates the state of a generator function and its usage in Python. It shows a code editor window with the following code:

```
main.py
1  def count_lines(file):
2      f = open(file)
3      lines = f.read().split("\n")
4      return lines
5
6  def count_lines_gen(file):
7      for line in open(file):
8          yield line
9
10     i = 0
11     for line in count_lines("file.txt"):
12         i += 1
13     print(i)
14
15     generator = count_lines_gen("file.txt")
16
17     i = 0
18     for line in generator:
19         i += 1
20     print(i)
21
22     i = 0
23     for line in generator:
24         i += 1
25     print(i)
```

Annotations and arrows point to specific lines of code:

- State of iterator is saved, function is paused:** Points to line 8 (`yield line`).
- Generator object:** Points to line 15 (`generator = count_lines_gen("file.txt")`).
- Cannot iterate more than once:** Points to line 23 (`for line in generator:`).

# Functions vs. Generators

Here are some of the key differences between a **generator** and a **function**.

Functions	Generators
Uses the <code>return</code> statement	Uses the <code>yield</code> statement
Begins executing when called	Returns a generator object
Control returned to caller when finished	Function pauses after <code>yield</code> , can be resumed
Stack space is deallocated after return	Stack space maintained after <code>yield</code>
Result is an iterable that can be iterated multiple times	Generator object can only be iterated over once