Python Object-Oriented Program with Libraries

Unit 1: Object-Oriented Programming

CHAPTER 4: ITERABLES, ITERATORS AND GENERATORS (ORDERING OF CONTAINERS)

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Understand what is iterator, iterable and generator
- A container is a data collection.
- A container is iterable if it can be accessed in a sequential format using iterators.
- An iterator is a pointer which can access each data element.

Iterables vs. Iterators vs. Generators

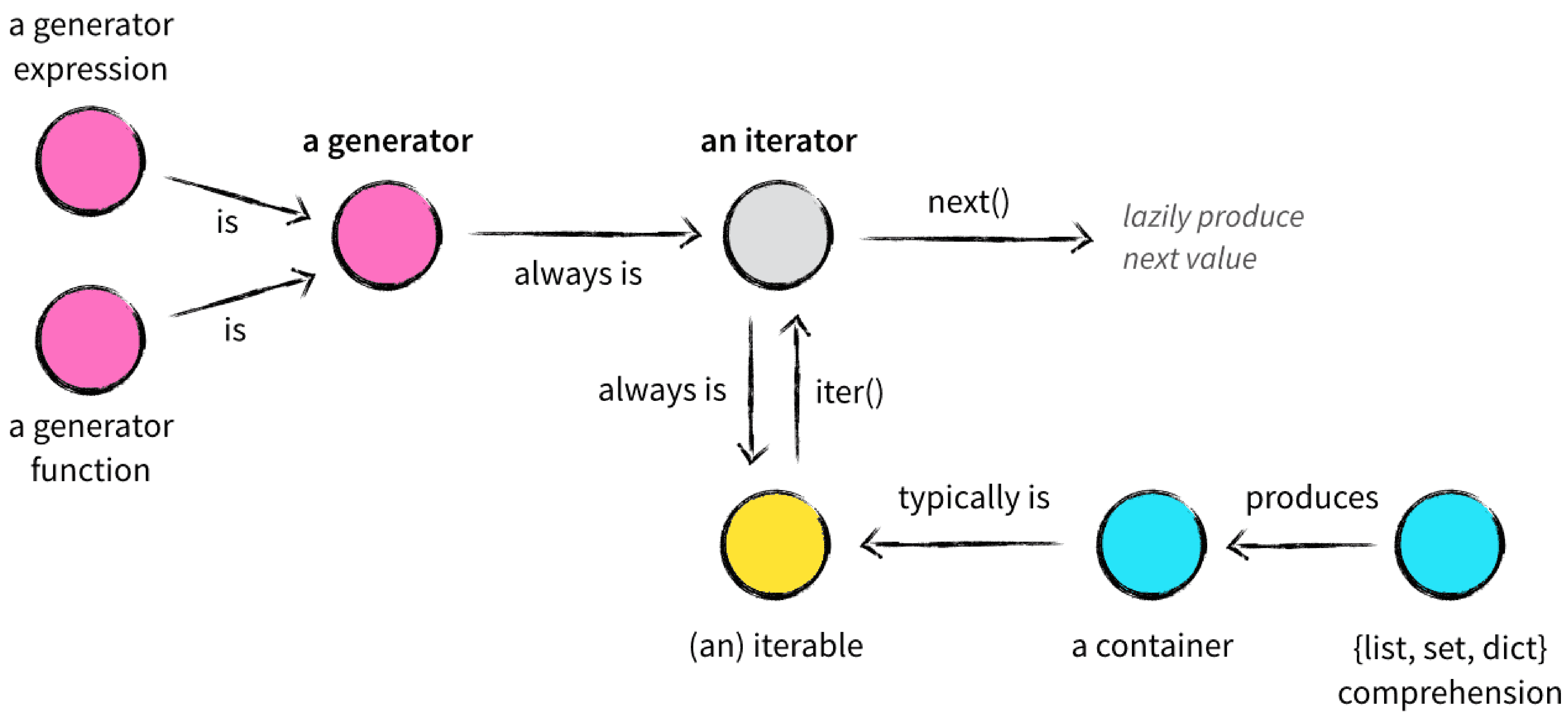
LECTURE 1

Iterables vs. Iterators vs. Generators

A little pocket reference on iterables, iterators and generators.

The following related concepts in Python are very confusing:

- a container
- an iterable
- an iterator
- a generator
- a generator expression
- a {list, set, dict} comprehension


a generator expression
a generator
an iterator
is
is
always is
next()
lazily produce next value
a generator function
always is
iter()
typically is
produces
(an) iterable
a container
{list, set, dict} comprehension

Containers

LECTURE 2

Containers

Containers are data structures holding elements, and that support membership tests. They are data structures that live in memory, and typically hold all their values in memory, too. In Python, some well known examples are:

- **list**, deque, …
- **set**, frozensets, …
- **dict**, defaultdict, OrderedDict, Counter, …
- **tuple**, namedtuple, …
- **str**

Containers are easy to grasp, because you can think of them as real life containers: a box, a cubboard, a house, a ship, etc.

Contains (Membership Check)

container1.py

- Technically, an object is a container when it can be asked whether it contains a certain element. You can perform such membership tests on lists, sets, or tuples alike: (Demo Program: container1.py)

```
>>> assert 1 in [1, 2, 3] # lists
>>> assert 4 not in [1, 2, 3]
>>> assert 1 in {1, 2, 3} # sets
>>> assert 4 not in {1, 2, 3}
>>> assert 1 in (1, 2, 3) # tuples
>>> assert 4 not in (1, 2, 3)
```

Dict membership will check the keys:

container2.py

```
>>> d = {1: 'foo', 2: 'bar', 3: 'qux’}
>>> assert 1 in d
>>> assert 4 not in d
>>> assert 'foo' not in d # 'foo' is not a _key_ in the dict
```

Finally you can ask a string if it "contains" a substring:

container3.py

- Dict membership will check the keys:

```
>>> s = 'foobar’
>>> assert 'b' in s
>>> assert 'x' not in s
>>> assert 'foo' in s # a string "contains" all its substrings
```

- The last example is a bit strange, but it shows how the container interface renders the object opaque. A string does not literally store copies of all of its substrings in memory, but you can certainly use it that way.

NOTE:

- Even though most containers provide a way to produce every element they contain, that ability does not make them a container but an iterable (we'll get there in a minute).
- Not all containers are necessarily iterable. An example of this is a Bloom filter. Probabilistic data structures like this can be asked whether they contain a *certain* element, but they are unable to return their individual elements.

Iterables

LECTURE 3

Iterables

- As said, most containers are also iterable. But many more things are iterable as well. Examples are open files, open sockets, etc. Where containers are typically finite, an iterable may just as well represent an infinite source of data.
- An iterable is any object, not necessarily a data structure, that can return an iterator (with the purpose of returning all of its elements). That sounds a bit awkward, but there is an important difference between an iterable and an iterator. Take a look at this example:

```
>>> x = [1, 2, 3]
>>> y = iter(x)
>>> z = iter(x)
>>> next(y)
1
>>> next(y)
2
>>> next(z)
1
>>> type(x)
<class 'list’>
>>> type(y)
<class 'list_iterator'>
```

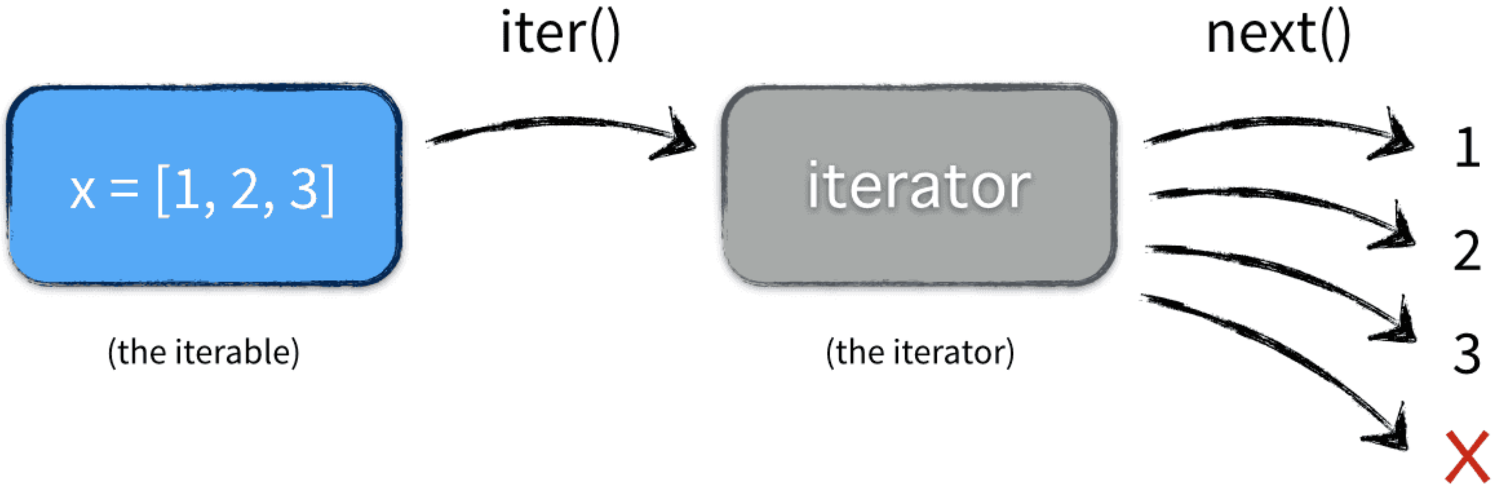
container4.py

Here, x is the iterable, while y and z are two individual instances of an iterator, producing values from the iterable x. Both y and z hold state, as you can see from the example. In this example, x is a data structure (a list), but that is not a requirement.

NOTE:
Often, for pragmatic reasons, iterable classes will implement both __iter__() and __next__() in the same class, and have __iter__() return self, which makes the class both an iterable and its own iterator. It is perfectly fine to return a different object as the iterator, though.

For-Each Loop

```
x = [1, 2, 3]
for elem in x:
    ...
```



Disassemble Python Code for Iterator

- When you disassemble this Python code, you can see the explicit call to GET_ITER, which is essentially like invoking iter(x). The FOR_ITER is an instruction that will do the equivalent of calling next() repeatedly to get every element, but this does not show from the byte code instructions because it's optimized for speed in the interpreter.

Disassemble Python Code for Iterator

disassemble.py

```
>>> import dis
>>> x = [1, 2, 3]
>>> dis.dis('for _ in x: pass’)
        1           0 SETUP_LOOP                14 (to 17)
                    3 LOAD_NAME                 0 (x)
                    6 GET_ITER
        >>          7 FOR_ITER                  6 (to 16)
                    10 STORE_NAME               1 (_)
                    13 JUMP_ABSOLUTE            7
        >>          16 POP_BLOCK
        >>          17 LOAD_CONST               0 (None)
                    20 RETURN_VALUE
```

Iterator

LECTURE 4

Iterators

- So, what is an iterator then? It's a stateful helper object that will produce the next value when you call **next()** on it. Any object that has a **__next__()** method is therefore an iterator. How it produces a value is irrelevant.
- So, an iterator is a value factory. Each time you ask it for "the next" value, it knows how to compute it because it holds internal state.

Iterators

- There are countless examples of iterators. All of the itertools functions return iterators. Some produce infinite sequences:

```
>>> from itertools import count
>>> counter = count(start=13)
>>> (counter)
13
>>> (counter)
14
```

Iterators

```python
from itertools import count
counter = count(start=13)
print((counter))
print(next(counter))
print(next(counter))
```

iterator2.py

```
count(13)
13
14
```

Cyclic Iterator

- Some produce infinite sequences from finite sequences:

iterator3.py

```
from itertools import cycle
colors = cycle(['red', 'white', 'blue'])
print(next(colors))
print(next(colors))
print(next(colors))
print(next(colors))
```

```
red
white
blue
red
```

Cyclic Iterator

- Some produce finite sequences from infinite sequences:

```
from itertools import islice, cycle
colors = cycle(['red', 'white', 'blue'])  # infinite
limited = islice(colors, 0, 7)            # finite
for x in limited:                         # so safe to use for-loop on
    print(x)
```

iterator4.py

```
red
white
blue
red
white
blue
red
```

Example for Iterable and Iterator

```
from itertools import islice
class fib:
    def __init__(self):
        self.prev = 0
        self.curr = 1
    def __iter__(self):
        return self
    def __next__(self):
        value = self.curr
        self.curr += self.prev
        self.prev = value
        return value

f = fib()
alist = list(islice(f, 0, 10))
print(alist)
```

iterator5.py

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Example for Iterable and Iterator

- Note that this class is both an iterable (because it sports an **`__iter__()`** method), and its own iterator (because it has a **`__next__()`** method).

- The state inside this iterator is fully kept inside the prev and curr instance variables, and are used for subsequent calls to the iterator. Every call to next() does two important things:
    1. Modify its state for the next next() call;
    2. Produce the result for the current call.

Generators

LECTURE 5

Generators

- Finally, we've arrived at our destination! The generators are my absolute favorite Python language feature. **A generator is a special kind of iterator**—the elegant kind.
- A generator allows you to write iterators much like the Fibonacci sequence iterator example above, but in an elegant succinct syntax that avoids writing classes with **__iter__()** and **__next__()** methods.
- Let's be explicit:
  - Any generator also is an iterator (not vice versa!);
  - Any generator, therefore, is a factory that lazily produces values.

Central idea: a lazy factory

From the outside, the iterator is like a lazy factory that is idle until you ask it for a value, which is when it starts to buzz and produce a single value, after which it turns idle again.

Generating Function

generator1.py

```
from itertools import islice
def fib():
    prev, curr = 0, 1
    while True:
        yield curr
        prev, curr = curr, prev + curr

f = fib()
alist = list(islice(f, 0, 10))
print(alist)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Generating Function

- Wow, isn't that elegant? Notice the magic keyword that's responsible for the beauty:

  `yield`

- Let's break down what happened here: first of all, take note that fib is defined as a normal Python function, nothing special. Notice, however, that there's no return keyword inside the function body. The return value of the function will be a generator (read: an iterator, a factory, a stateful helper object).

Generating Function

- Now when `f=fib()` is called, the generator (the factory) is instantiated and returned. No code will be executed at this point: the generator starts in an idle state initially. To be explicit: the line `prev, curr = 0, 1` is not executed yet.
- Then, this generator instance is wrapped in an `islice()`. This is itself also an iterator, so idle initially. Nothing happens, still.
- Then, this iterator is wrapped in a `list`(), which will consume all of its arguments and build a list from it. To do so, it will start calling `next()` on the `islice()` instance, which in turn will start calling `next()` on our f instance.

Generating Function

- But one step at a time. On the first invocation, the code will finally run a bit: `prev, curr = 0, 1` gets executed, the while True loop is entered, and then it encounters the yield curr statement. It will produce the value that's currently in the curr variable and become idle again.
- This value is passed to the `islice()` wrapper, which will produce it (because it's not past the 10th value yet), and list can add the value 1 to the list now.

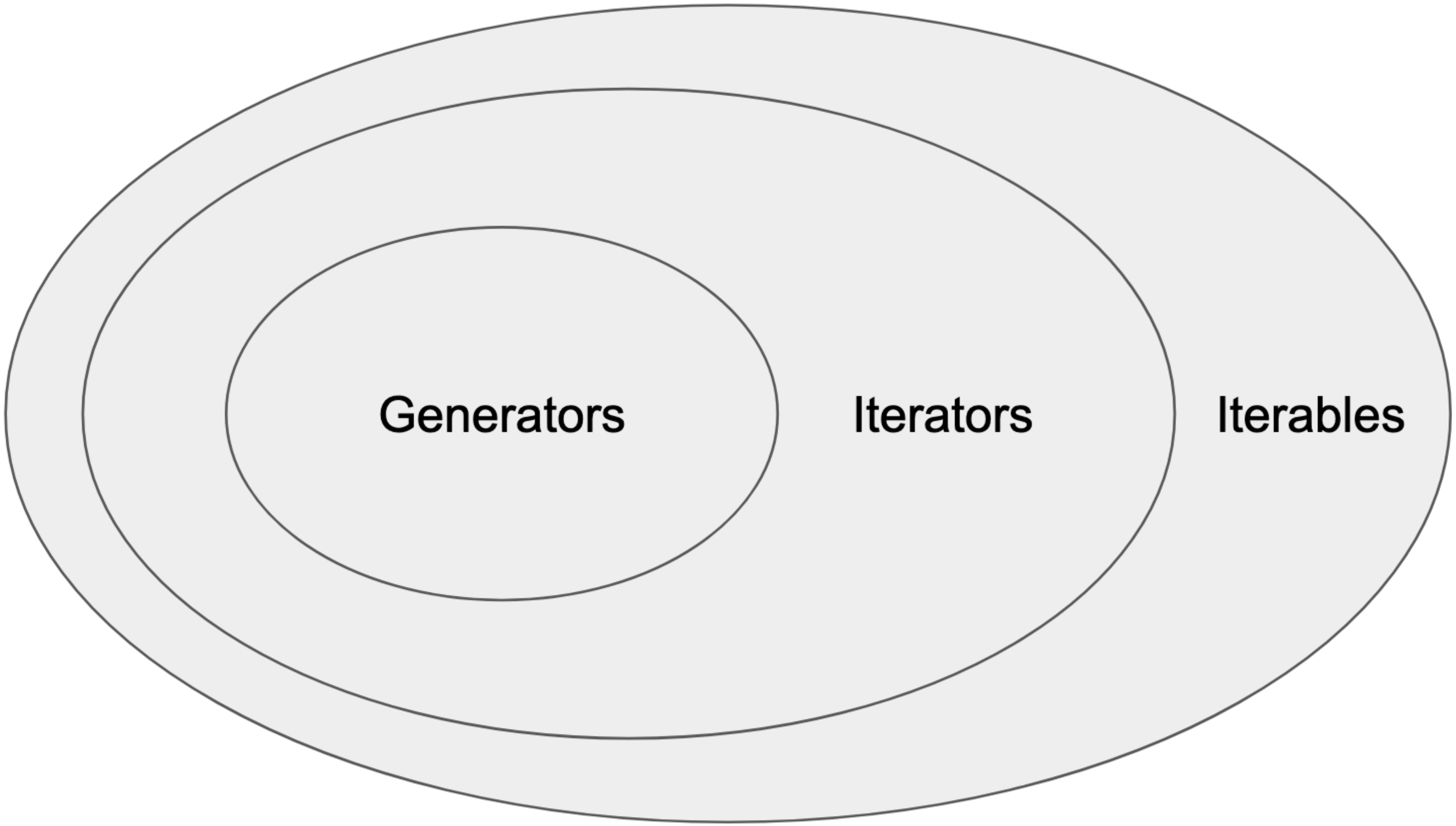
Generating Function

- Then, it asks `islice()` for the next value, which will ask f for the next value, which will "unpause" f from its previous state, resuming with the statement `prev, curr = curr, prev + curr`. Then it re-enters the next iteration of the while loop, and hits the `yield curr` statement, returning the next value of `curr`.

- This happens until the output list is 10 elements long and when `list()` asks `islice()` for the 11th value, `islice()` will raise a **`StopIteration`** exception, indicating that the end has been reached, and list will return the result: a list of 10 items, containing the first 10 Fibonacci numbers. Notice that the generator doesn't receive the 11th `next()` call. In fact, it will not be used again, and will be garbage collected later.

Types of Generators

LECTURE 6


Generators
Iterators
Iterables

Types of Generators

- There are two types of generators in Python: generator functions and generator expressions. A generator function is any function in which the keyword yield appears in its body. We just saw an example of that. The appearance of the keyword yield is enough to make the function a generator function.
- The other type of generators are the generator equivalent of a list comprehension. Its syntax is really elegant for a limited use case.

Python Generators

A Quick Guide for Beginners

```
def gen_func():
  ...
  while <cond>:
    ...
    yield num

next(gen_func())
```

```
gen_expr = (a**(1/2)
for a in alist)
```

```
pipeline
gen_fn1()
   => gen_fn2()
      => gen_fn3()
```

```
for item in
gen_func(args):
   print(item)
```

www.techbeamers.com

Comprehension Generators

generator2.py

```
numbers = [1, 2, 3, 4, 5, 6]
a = (x*x for x in numbers)
b = [x*x for x in numbers]      # comprehensive list
c = {x*x for x in numbers}      # comprehensive set
d = {x:x*x for x in numbers}    # comprehensive dict
print(a)
print(b)
print(c)
print(d)

<generator object <genexpr> at 0x000001BE7126F048>
[1, 4, 9, 16, 25, 36]
{1, 4, 36, 9, 16, 25}
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Generator expressions

- A comprehension-based expression that results in an iterator object
  - Does not result in a container of values
  - Must be surrounded by parentheses unless it is the sole argument of a function
  - May be returned as the result of a function

```
numbers = (random() for _ in range(42))
sum(numbers)
```

```
sum(random() for _ in range(42))
```

Generator Expression

note: this is not a tuple comprehension

generator3.py

```
# Generator Expression
numbers = [1, 2, 3, 4, 5, 6]
a = (x*x for x in numbers)
print(next(a))
print(next(a))
print(next(a))
print(next(a))
```

```
1
4
9
16
```

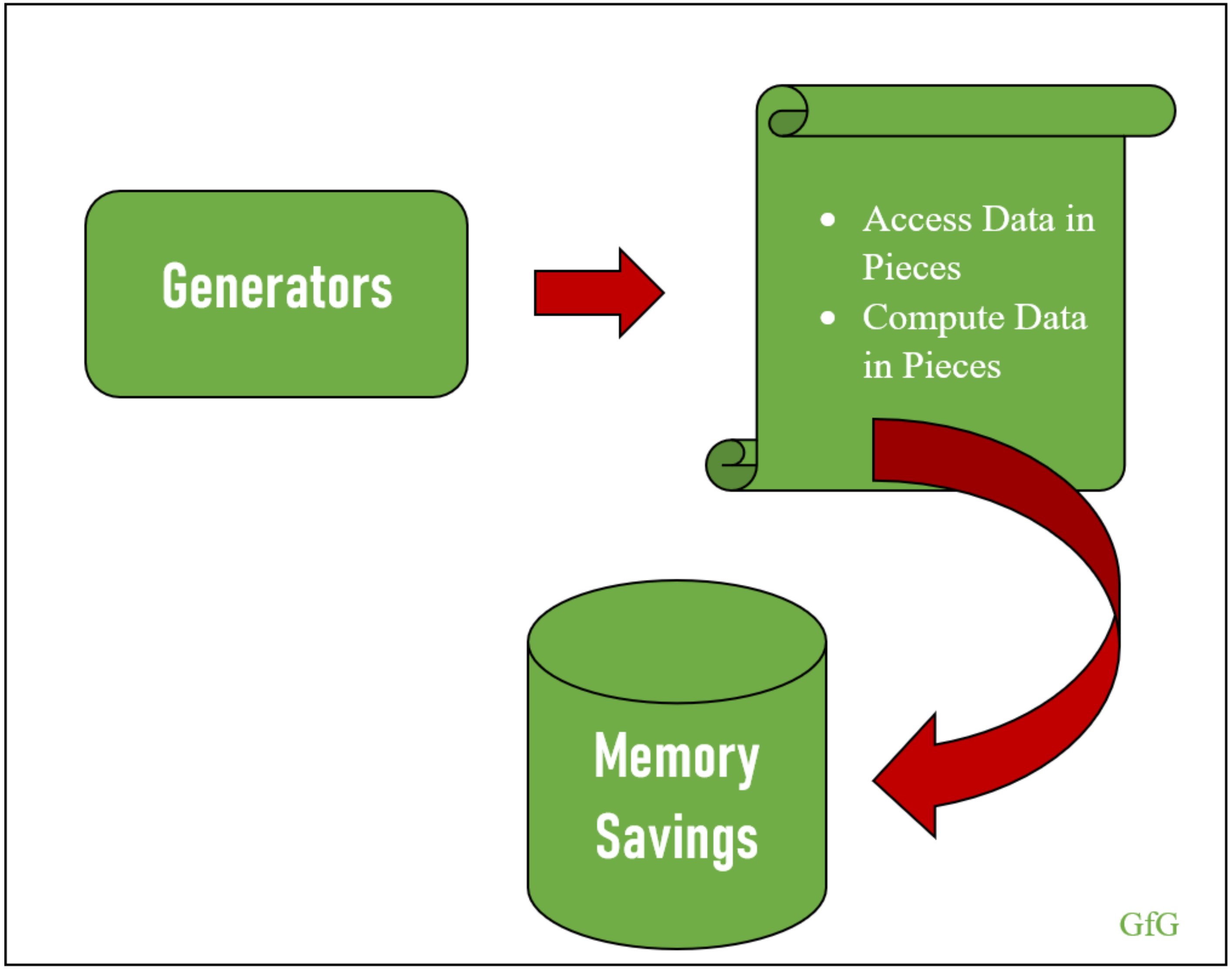
Generator Expression

note: this is not a tuple comprehension

- Note that, because we read the first value from lazy_squares with next(), it's state is now at the "second" item, so when we consume it entirely by calling list(), that will only return the partial list of squares. (This is just to show the lazy behaviour.) This is as much a generator (and thus, an iterator) as the other examples above.

Summary

LECTURE 7


Generators
Access Data in Pieces
Compute Data in Pieces
Memory Savings
GfG

Summary

- Generators are an incredible powerful programming construct. They allow you to write streaming code with fewer intermediate variables and data structures. Besides that, they are more memory and CPU efficient. Finally, they tend to require fewer lines of code, too.

Generating a list

- Tip to get started with generators: find places in your code where you do the following:

```
def something():
    result = []
    for ... in ...:
        result.append(x)
    return result
```

Generating a list

- Tip to get started with generators: find places in your code where you do the following:

```
def iter_something():
    for ... in ...:
        yield x
# def something():
# Only if you really need a list structure
# return list(iter_something())
```