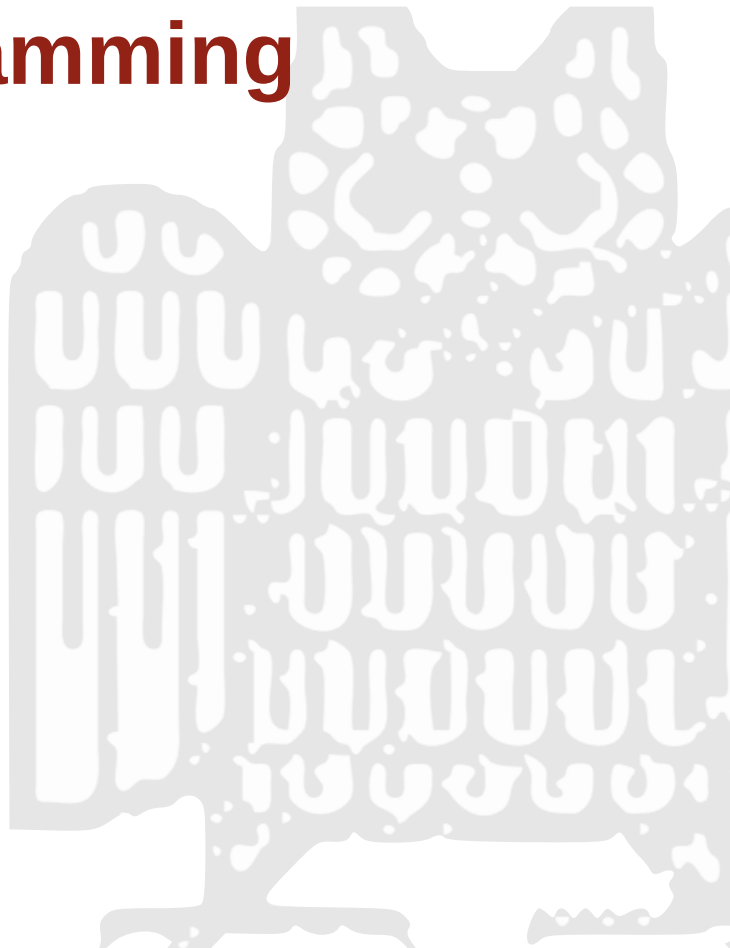
Introduction to Python Programming

18 – Generators (Advanced Topic)

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23



Iterators (Recap)

```
for item in ⟨iterable⟩:
    ⟨block⟩
```

```
it = iter(⟨iterable⟩)
while True:
    try:
        item = next(it)
    except StopIteration:
        break
    ⟨block⟩
```

Iterators (Recap)

- **Iterators** are instances of classes (objects) that implement the following two methods:
  - `__next__(self)` is called by `next()` and returns the next element or raises StopIteration
  - `__iter__(self)` is called by `iter()` and usually returns the iterator itself
- An object o is **iterable** if it supports `iter(o)`, i.e.
  - `__iter__(self)` is implemented and returns an iterator

Iterators (Recap)

```
class ListIterator:
    def __init__(self, lis):
        self.lis = lis
        self.index = -1
    def __iter__(self):
        return self
    def __next__(self):
        self.index += 1
        if self.index >= len(self.lis):
            raise StopIteration
        return self.lis[self.index]
```

Iterators (Recap)

- Iterators are good if we want to do iteration with custom objects we design and tailor-make for our purposes
- Iterators produce one value at a time, when prompted be next() implemented as __next__ in the iterator
- That is memory efficient: you can e.g. represent sth. infinite (like the sequence of all odd/even numbers) with finite means
- for loops elegantly iterate over iterators … (they do the prompting with next() under the hood)
- Also, sets e.g. cannot accessed via indices …

Generators

- A generator is like a simple way of producing an iterator
- A generator automatically implements __iter__ and __next__
- A generator is a function that produces (an iterator over) a sequence of results
- Instead of returning a value (list), a generator generates a series of values.
- Values are only produced when prompted … by next()
- You can use it to finitely represent sth. infinte e.g.
- Technically, a generator is a function that contains one or more **yield** statements (instead of a **return** statement)

The yield statement

- The yield statement is similar to the return statement
  - “yield something” returns a value
- However, there is a fundamental difference to the return statement (see next slides).

An Example

```
def myrange(n):
    print("count from 0 to", n)
    m = 0
    while m < n:
        print("next item is", m)
        return m
        m += 1
    print("done")
```

```
>>> myrange(3)
count from 0 to 3
next item is 0
>>>
```

An Example

```
def myrange(n):
    print("count from 0 to", n)
    m = 0
    while m < n:
        print("next item is", m)
        return m
        m += 1
    print("done")
```

```
>>> myrange(3)
count from 0 to 3
next item is 0
>>>
```

return returns value and terminates the function

An Example

```
def myrange(n):
    print("count from 0 to", n)
    m = 0
    while m < n:
        print("next item is", m)
        yield m
        m += 1
    print("done")
```

```
>>> it = myrange(3)
>>> next(it)
count from 0 to 3
next item is 0
0
>>> next(it)
next item is 1
1
>>> next(it)
next item is 2
2
>>> next(it)
done
Traceback [...]
StopIteration
```

An Example

```
def myrange(n):
    print("count from 0 to", n)
    m = 0
    while m < n:
        print("next item is", m)
        yield m
        m += 1
    print("done")
```

yield returns value and suspends the function, with all internal values remembered, ready for next prompt

```
>>> it = myrange(3)
>>> next(it)
count from 0 to 3
next item is 0
0
>>> next(it)
next item is 1
1
>>> next(it)
next item is 2
2
>>> next(it)
done
Traceback [...]
StopIteration
```

An Example

- Generators behave quite differently than normal functions
- Calling a generator function creates a generator object
  - but it does **NOT** start running the function
- Generator objects are iterators

```
>>> it = myrange(3)
>>> next(it)
count from 0 to 3
next item is 0
0
>>> next(it)
next item is 1
1
>>> next(it)
next item is 2
2
>>> next(it)
Traceback [...]
StopIteration
```

An Example

- The first call of next(it) starts execution of the function
- The yield statement returns a value and **suspends** the execution of the function (remembering all internal values).
- The next call of next(it) **continues** execution of the function.
- Iteration stops when the generator returns

```
>>> it = myrange(3)
>>> next(it)
count from 0 to 3
next item is 0
0
>>> next(it)
next item is 1
1
>>> next(it)
next item is 2
2
>>> next(it)
Traceback [...]
StopIteration
```

An Example

- Iteration stops when the generator returns
  - execution falls off the end (StopIteration), or
  - a return statement is executed
- Note: only “naked” return statements are permitted in generator functions
  - we cannot mix “yield” and “return something”

Generators vs. Iterators

- Generator functions are often easier to implement than an iterator.

Exercise

- Rewrite this code into a generator function

```
class ListIterator:
    def __init__(self, lis):
        self.lis = lis
        self.index = -1
    def __iter__(self):
        return self
    def __next__(self):
        self.index += 1
        if self.index >= len(self.lis):
            raise StopIteration
        return self.lis[self.index]
```

Exercise

- Corresponding generator function:

```
def ListIterator(lis):
    n = 0
    while n < len(lis):
        yield lis[n]
        n += 1
```

```
def ListIterator(lis):
    for item in lis:
        yield item
```

Another Example

- Suppose we want to implement a function that returns a list of all words in a file.

```
def words(filename):
    result = []
    with open(filename) as f:
        for line in f:
            for word in line.split():
                result.append(word)
    return result

for word in words("example.txt"):
    ...
```

Another Example

- Suppose we want to implement a function that returns a list of all words in a file.
- Drawback: may not work for very large file
  - the complete content of the file is stored in the results list
  - memory problems …

Another Example

- As a generator:

```
def words(filename):
    with open(filename) as f:
        for line in f:
            for word in line.split():
                yield word

for word in words("example.txt"):
    ...
```

Yet anpther Example

```
def duplicate(it):
    for item in it:
        yield item
        yield item
```

```
>>> for item in duplicate([1,2,3]): print(item)
1
1
2
2
3
3
```

Generators & Recursion

- Recursion = a function calls itself (directly or indirectly)
- Generators cannot (!) call themselves
  - or at least, this would not have the desired effect
- How can we implement a recursive generator function?

Generators & Recursion

```
def flatten(lis):
    result = []
    for item in lis:
        if isinstance(item, list):
            result.extend(flatten(item))
        else:
            result.append(item)
    return result
```

```
>>> flatten([1, [2, 3], 4, [[5], 6], [], 7])
[1, 2, 3, 4, 5, 6, 7]
```

Does not work …

```
def flatten(lis):
    for item in lis:
        if isinstance(item, list):
            yield flatten(item)
        else:
            yield item
```

```
>>> flatten([1, [2, 3], 4, [[5], 6], [], 7])
<generator object>
>>> list(flatten([1, [2, 3], 4, [[5], 6], [], 7]))
[1, <generator object>, 4, <generator object>, ...]
```

Generators & Recursion

```
def flatten(lis):
    for item in lis:
        if isinstance(item, list):
            for nested in flatten(item):
                yield nested
        else:
            yield item
```

Solution: wrap the recursion in a for loop

```
>>> flatten([1, [2, 3], 4, [[5], 6], [], 7])
<generator object flatten at 0xb7517fcc>
>>> list(flatten([1, [2, 3], 4, [[5], 6], [], 7]))
[1, 2, 3, 4, 5, 6, 7]
```

Exercises

- Write a generator function that takes a list of integers as input and returns all numbers incremented by 1.
- Write a generator function that “appends” two iterators.

```
>>> appendit(iter([1,2,3]), iter([4,5,6]))
<generator object>
>>> list(appendit(iter([1,2,3]), iter([4,5,6])))
[1,2,3,4,5,6]
```

- Write a generator function that takes merges two iterators over sorted lists:

```
>>> list(merge(iter([1, 3, 5]), iter([2, 4])))
[1, 2, 3, 4, 5]
```