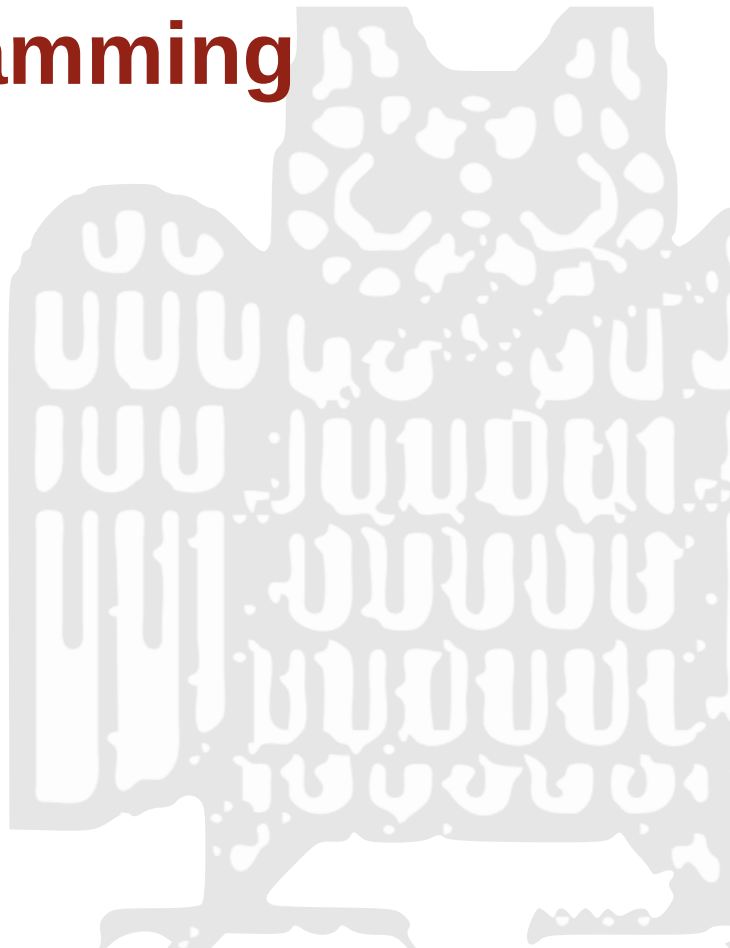
Introduction to Python Programming

16 – Iterators (Advanced Topic)

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

WS 2022/23



OOP

Many ways to look at OOP:

- Custom data structures & what you can do with the data …
- Modularity and large scale-software engineering
- Data encapsulation …
- Look at the world in terms of objects and their relationships
- Taxonomy and inheritance

OOP

A lot of things we want to model in our programmes involve a notion of iteration ≈ “stepping through a collection of things“

And we have seen this before …

Recap: Collection/Container Types

- Lists – [1, 2, 3]
- Tuples – (1, 2, 3)
- Sets – {1, 2, 3}
- Dictionaries – {1:'one', 2:'two', 3:'three'}
- Strings – '123'
- (and few more)

for loops & collections

```
for item in [1,2,3]:
    print(item)
for item in (1,2,3):
    print(item)
for item in {1,2,3}:
    print(item)
for key in {1:'one', 2:'two', 3:'three'}:
    print(key)
for ch in 'python':
    print(ch)
```

This is the “stepping through“, the iterating thing, I menioned earlier

Iterators

- But so far this works only for things (data structures, collection / container types) that are predefined in Python
- We would like to be able to do sth. like this also for special user-defined objects (user-defined classes/data types …)
- We do this in two steps:
  - First, we look at how these for loops are actually defined in Python under the hood (recall that lists, tuples, sets … are actually objects) …
  - Then, this will give us some ideas how we can do this for special user-defined objects/classes

for loops (simplified/“imperative“)

The following two loops are equivalent if **somelist** is a list (“**i**” is a fresh counter variable not used anywhere else)

```
for item in somelist:
    print(item)
```

```
i = 0
while i < len(somelist):
    item = somelist[i]
    print(item)
    i += 1
```

for loops (simplified/“imperative“)

The following two loops are equivalent if **somelist** is a list
(“**i**” is a fresh counter variable not used anywhere else)

```
for item in somelist:
    print(item)
```

```
i = 0
while i < len(somelist):
    item = somelist[i]
    print(item)
    i += 1
```

... but this is not (!) how **for** loops (over lists/collections) are implemented in Python … !

for loops (actually/under the hood using OO)

The following two loops **are** equivalent (for any collection/container object in Python)! ("**it**" is a fresh variable not used anywhere else)

```
for item in somecollection:
    print(item)
```

```
it = iter(somecollection)
while True:
    try:
        item = next(it)
    except StopIteration:
        break
    print(item)
```

for loops (actually/under the hood using OO)

The following two loops **are** equivalent (for any collection/container object in Python)! ("**it**" is a fresh variable not used anywhere else)

```
for item in somecollection:
    print(item)
```

```
it = iter(somecollection)
while True:
    try:
        item = next(it)
    except StopIteration:
        break
    print(item)
```

this is a call to a method that turns the object into an iterable

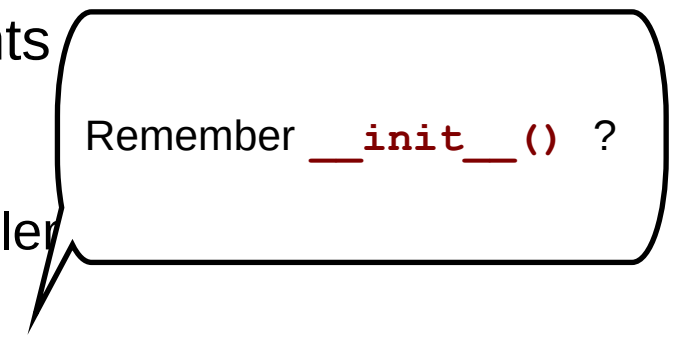
this is a call to a method that iterates

Iterators

- Iterators are objects that can be iterated over
- Iterators allow us to iterate over the elements of arbitrary collection types (lists, sets, …)
  - Think of iterators as pointers to the current element …
- `it = iter(⟨iterable⟩) # it = __iter__(⟨iterable⟩)`
  - creates an iterator for ⟨iterable⟩ ….
- `item = next(it)  # item = __next__(it)`
  - returns the next item
  - raises an exception (`StopIteration`) if there are no further items

Iterators

- Iterators are objects that can be iterated over
- Iterators allow us to iterate over the elements collection types (lists, sets, …)
  - Think of iterators as pointers to the current ele



- `it = iter(⟨iterable⟩)  # it = __iter__(⟨iterable⟩)`
  - creates an iterator for ⟨iterable⟩ ….
- `item = next(it)  # item = __next__(it)`
  - returns the next item
  - raises an exception (`StopIteration`) if there are no further items

Iterators

```
>>> it = iter([1,2,3])
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterators in Python

- **Iterators** are instances of classes (i.e. 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
- Lists, sets, tuples, dicts, …. are already programmed in Python to support `iter()` and `next()`
- You can make objects you define yourself interable by implementing `__iter__(self)` and `__next__(self)` as methods for the class from which the object is instantiated ☺ ☺ ☺

Duck Typing

- Iterators are an example of “duck typing”
- The idea is that it doesn’t matter what type the data is ...
- It just matters what one can do with the data, i.e., what kind of methods are implemented.

> *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*

iter(list) is like …

```
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]
```

for loops & collections

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

```
it = iter([1,2,3])
while True:
    try:
        item = next(it)
    except StopIteration:
        break
    print(item)
```

for loops & collections

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

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

Exercise

- What is the value of `result`?

```
# Version 1
result = 0
lis = [1,2,3]
for x in lis:
    for y in lis:
        result += x * y
print(result)
```

```
# Version 2
result = 0
it = iter([1,2,3])
for x in it:
    for y in it:
        result += x * y
print(result)
```

Exercise

- What happens here?

```
lis = [1,2,3]
for x in lis:
    lis.append(x)
    print(lis)
```

use “lis[:]” instead of “lis” here

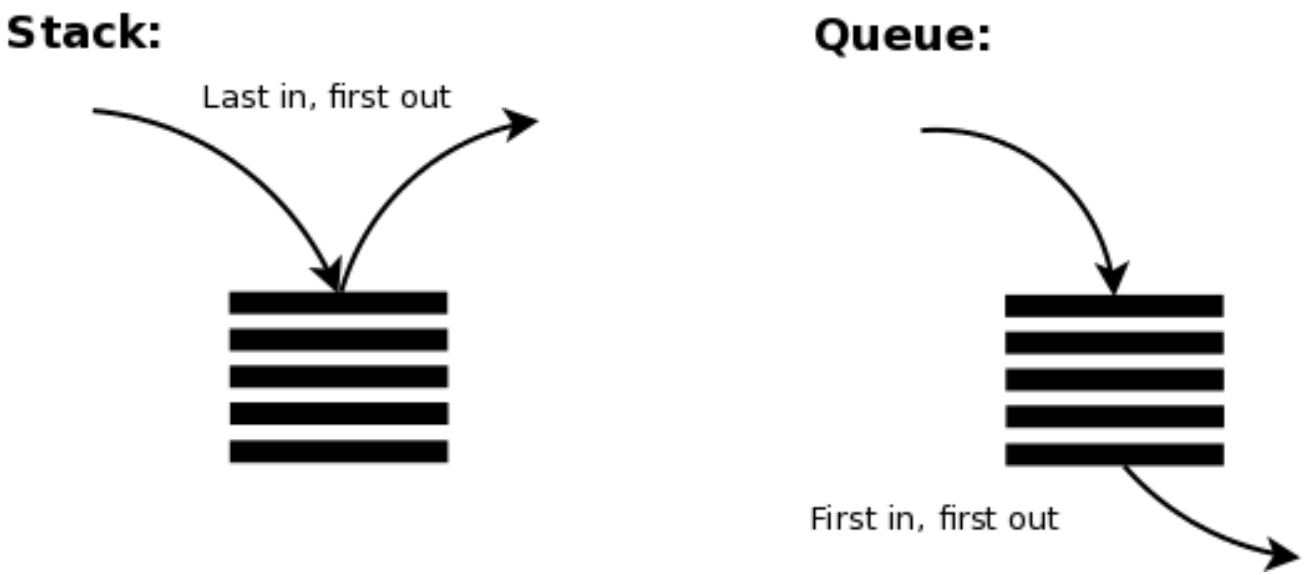
Iterators

- Iterators can often be used instead of lists (sets, …)
  - for x in iterator: ...
  - if x in iterator: ...
- Some limitations:
  - iterators have no length
    (⇒ len(iterator) does not work)
  - iterators cannot be copied (at least not easily)
  - no slicing etc.
- Why do we use them? User-defined objects, finite representation of infinite things … (all even numbers, all odd numbers …)

Iterator ⇒ Collection

- To turn an iterator into a collection, i.e. a list, set, tuple, …:
  - `list(iterator)`
  - `set(iterator)`
  - `tuple(iterator)`
  - …

Stacks



Source: https://gohighbrow.com/stacks-and-queues/

Another example: Stacks

```
class Stack:
    def __init__(self):
        self.data = None
    def push(self, elt):
        self.data = (elt, self.data)
    def pop(self):
        if self.data:
            (elt, self.data) = self.data
        else:
            raise IndexError
        return elt
    def __iter__(self):
        return StackIterator(self)
```

Another example: Stacks

```
class Stack:
    def __init__(self):
        self.data = None
    def push(self, elt):
        self.data = (elt, self.data)
    def pop(self):
        if self.data:
            (elt, self.data) = self.data
        else:
            raise IndexError
        return elt
    def __iter__(self):
        return StackIterator(self)
```

None
(a,None
(b,(a,None)
(c, (b,(a,None))

Another example: Stacks

```
class Stack:
    def __init__(self):
        self.data = None
    def push(self, elt):
        self.data = (elt, self.data)
    def pop(self):
        if self.data:
            (elt, self.data) = self.data
        else:
            raise IndexError
        return elt
    def __iter__(self):
        return StackIterator(self)
```

Non
(a,None
(b,(a,None)
(c,(b,(a,None))

(b,(a,None)
(a,None

Another example: Stacks

```
class StackIterator:
    def __init__(self, stack):
        self.data = stack.data
    def __iter__(self):
        return self
    def __next__(self):
        if self.data == None:
            raise StopIteration
        elt = self.data[0]
        self.data = self.data[1]
        return elt
```

None
(a,None)
(b,(a,None))
(c, (b,(a,None)))

Another example: Stacks

```
>>> s = Stack()
>>> s.push(1)
>>> s.push(2)
>>> s.push(3)
>>> for item in s:
...     print(item)
...
3
2
1
>>> list(s)
[3, 2, 1]
```

Some built-in iterators

- `enumerate(iterable)`
  - Returns an iterator that yields pairs containing a count and a value yielded by the iterable argument.
  - For instance:

```
for (num, item) in enumerate(["a", "b", "c"]):
    print(num, item)
```

Some built-in iterators

- `zip(it1, it2, ...)`
  - Return an iterator that yields tuples where the i-th element of each tuple comes from the i-th iterable argument.
  - For instance:

```
for pair in zip([9, 1, 42], ["a", "b", "c"]):
    print(pair)
```

Some built-in iterators

- `open(filename)`
  - File objects are iterators (over the lines in the file)

```
1  def grep(filename, word):
2      '''Returns True if filename contains word'''
3      f = open(filename)
4      while True:
5          line = f.readline()
6          if line == '': # no input => stop
7              break
8          if word in line:
9              return True
10     f.close()
11     return False
```

Some built-in iterators

- `open(filename)`
  - File objects are iterators (over the lines in the file)

```
1 def grep(filename, word):
2     with open(filename) as f:
3         while True:
4             line = f.readline()
5             if line == '':
6                 break
7             if word in line:
8                 return True
9     return False
```

Some built-in iterators

- `open(filename)`
  - File objects are iterators (over the lines in the file)

```
1 def grep(filename, word):
2     with open(filename) as f:
3         for line in f:
4             if word in line:
5                 return True
6     return False
```

Exercise #1

- Reimplement the builtin iterator enumerate(it).

```
class MyEnumerate:
    ...

for (i, ch) in MyEnumerate("Python"):
    print(i, ch)

# 0 P
# 1 y
# 2 t
# 3 h
# 4 o
# 5 n
```

Exercise #2

- Reimplement the following function so that it also works with iterators:

```
def avg(seq):
    return sum(seq) / len(seq)

print(avg([1,2,3,4])) # prints 2.5
print(avg(iter([1,2,3,4]))) # error
```

Exercise #3

- Implement an iterator that iterates over a file by paragraph:

```
class ByParagraph:
    ...

with open("example.txt") as f:
    for par in ByParagraph(f):
        print('BEGIN PAR' + par + 'END PAR')
```