Iterables and Iterators: Going Loopy with Python

Steve Holden
BMLL Technologies
PyData London 2016

Me

- 40+ years interest in object-oriented programming
- 20+ years Python experience
  - Wrote & taught Python classes professionally
  - Also consulted with all sorts of clients
  - Started PyCon in the USA
  - Former director and chair of Python Software Foundation
  - Generally just *really* enthusiastic about Python
- CTO of BMLL Technologies, London

The Talk

- Code in slides for discussion
- Executable Jupyter Notebook for demonstration
  - Think of it as a multimedia presentation
- Please feel free to ask questions
  - Happy to enter *ad hoc* code to answer them
- There'll be a short Q&A session at the end too
- Leaving before lunch– buttonhole me now!

Iteration in Python

- Iteration is generally an enumerative technique
  - "Do this on every one of these"
- Generally, anything that uses the **`for`** keyword
  - **`for`** statements
  - List, dict and set comprehensions
- This material is *not* relevant to **`while`** loops

A Bit of History

- How Python *originally* did loops (pseudocode):

```
_private_var = 0
while True:
    try:
        i = test_list.__getitem__(_private_var)
    except IndexError:
        break
    do_something_with(i)
    _private_var += 1
```

This Mechanism Still Works

- Python is great at backward compatibility!

```
class Stars():
    "Class with only __init__ and __getitem__."
    def __init__(self, N):
        self.N = N
    def __getitem__(self, index):
        print("Getting item:", index)
        if index > self.N:
            raise IndexError
        return "*" * index
```

This Method Has Limitations

- Items must be numerically indexable
  - *i.e.* needs a **`__getitem__`** method that takes numerical arguments
- Indices must run from 0 through N-1
- ∴ cannot be used with unordered containers
  - *e.g.* sets, dicts

So a New Mechanism Was Born

- If the subject of a **`for`** has a **`__iter__`** method:
  - Call that method in a temporary and save the result
  - At the start of each iteration:
    - Produce the next value for the loop by calling the temporary's **`__next__`** method
    - [Python 2 – call its **`next`** method]
  - If the call raises a **`StopIteration`** exception:
    - Terminate the loop
- In the absence of the a **`__iter__`** method:
  - Use the original **`__getitem__`**-based mechanism

These objects are *Iterables*

- An object with `__iter__` method
  - But no `__next__`
- What Python types are iterables?
  - Lists
  - Tuples
  - Dicts
  - Sets
- In short, anything you might want to iterate over!

Pseudo-Code of a `for` Loop

- The interpreter's logic looks something like this

```
_ = test_list.__iter__()  # creates an iterator
while True:
    try:
        i = _.__next__()  # Gets next iterator value
                          # Python 2: _.next()
    except StopIteration: # iterator is exhausted
        break
    do_something_with(i)
```

- See example in notebook

Iterables *vs.* Iterators

- Iterables (lists, tuples, sets, dicts, etc.)
  - Do *not* have a **`__next__`** method
  - Have an **`__iter__`** method
  - Whose result is an *iterator*
- Iterators (built-in types, usually created by factory)
  - Have both a **`__iter__`** *and* a **`__next__`** method
  - The **`__iter__`** method returns self
    - So iterators *can* be iterated over
    - With some interesting differences in behaviour

Iterating Over Iterables

```
test_list = ['Roberta', 'Tom, 'Alice']
for i in test_list:
    for j in test_list:
        do_something_with(i + ":" + j)
```

```
--- Roberta:Roberta ---
--- Roberta:Tom ---
--- Roberta:Alice ---
--- Tom:Roberta ---
--- Tom:Tom ---
--- Tom:Alice ---
--- Alice:Roberta ---
--- Alice:Tom ---
--- Alice:Alice ---
```

Iterating Over Iterators (1)

```
iterator_1 = iter(test_list)
iterator_2 = iter(test_list)
for i in iterator_1:
    print("outer loop")
    for j in iterator_2:
        print("inner loop")
        do_something_with(i + ":" + j)
```

```
outer loop
inner loop
--- Roberta:Roberta ---
inner loop
--- Roberta:Tom ---
inner loop
--- Roberta:Alice ---
outer loop
outer loop
```

Iterating Over Iterators (2)

```
iterator_1 = iter(test_list)
for i in iterator_1:
    print("outer loop")
    for j in iterator_1:
        print("inner loop")
        do_something_with(i + ":" + j)
```

```
outer loop
inner loop
--- Roberta:Tom ---
inner loop
--- Roberta:Alice ---
```

Identities to Remember

```
iter(o) == o.__iter__()
```

```
next(o) == o.__next__()
```

in Python 2:

```
next(o) == o.next()
```

A Homebrew Iterator (1)

```
class MyIterator:
    """An iterator to produce each
    character of a string N times."""

    def __init__(self, s, N):
        self.s = s
        self.N =  N
        self.pos = self.count = 0
    def __iter__(self):
        return self
```

A Homebrew Iterator (2)

```
def __next__(self):
    if self.pos == len(self.s):
        raise StopIteration
    result = self.s[self.pos]
    self.count += 1
    if self.count == self.N:
        self.pos += 1
        self.count = 0
    return result
```

Generators: Easier Iterables

- Function bodies with **`yield`** in them are special
- They are called *generator functions*
- Calling them does not execute the code in the body
- Instead, it returns an iterator!

```
>>> def gen_func():
...     yield
...
>>> f = gen_func()
>>> type(f)
<type 'generator'>
```

How Generators Work

- Generators are iterators
- Generator functions are iterator *factories*
- Calling the generator's **`__next__`** method runs the code in the function body
- Each yield becomes the return value of the **`__next__`** call

A Simple Generator Function

```
def rangedown(n):
    for i in reversed(range(n)):
        yield I
generator = rangedown(4)
for x in generator:
    print(x)

3
2
1
0
```

Generator Expressions

```
genexp = (i*2 for i in range(4))
type(rangedown), type(generator), type(genexp)

(function, generator, generator)

for o in genexp:
    print(o)

0
2
4
6
```

The Basic Iterable

```
class MIString(str):
    def __new__(cls, value, N):
        return str.__new__(cls, value)
    def __init__(self, value, N):
        self.N = N
    def __iter__(self):
        return MyIterator(self, self.N)

[s for s in MIString("xyz", 3)]

['x', 'x', 'x', 'y', 'y', 'y', 'z', 'z', 'z']
```

Using a Generator as `__iter__`

```
def __iter__(self):
        for c in str(self):
            for i in range(self.N):
                yield c
```

- Why isn't the second line as follows?

```
for c in str(self):
```

Thanks For Listening!

?

steveholden@bmlltech.com

https://github.com/steveholden/iteration.git