Iterables and Iterators:

Round and Round the Mulberry Bush

Steve Holden
Department for International Trade
24 Feb, 2022

The Talk

- Slides and notebooks available from
  `https://github.com/steveholden/iteration.git`
  (update due after this talk – watch `#developers` for news)
- 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

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
- NOTE: This material is *not* relevant to **`while`** loops
  - They don't iterate *over* anything
  - Just loop until a condition is false

Identity to Remember

`o[index]`

is simply an alternative spelling for

`o.__getitem__(index)`

A Bit of History

- Python's original iteration required the object being iterated over to have a **`__getitem__`** method
- Suppose the interpreter were written in Python …
  - What would

```
for item in an_object:
    do_something_with(item)
```

look like

Interpreter "Internals" (but Python)

```
_index_var = 0
while True:
    try:
        item = an_object.__getitem__(_index_var)
    except IndexError:
        break
    do_something_with(item)
    _index_var += 1
```

This Mechanism Still Works

- Python regards backward compatibility as important!

```
class Stars():
    "Class with just __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 and save the result in a temporary
  - At the start of each iteration:
    - Produce the next value for the loop by calling the temporary's **`__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!

Internals of the Modern `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
    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
  - Do have an **`__iter__`** method …
  - … which returns 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']
for i in test_list:
    for j in test_list:
        do_something_with(f'{i} : {j}’)
```

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

Iterating Over Iterators (1)

```
test_list = ['Roberta', 'Tom']
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(f'{i} : {j}’)
outer loop
inner loop
--- Roberta : Roberta ---
inner loop
--- Roberta : Tom ---
outer loop
```

Iterating Over Iterators (2)

```
iterator_1 = iter(['Roberta', 'Tom'])
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 ---
```

More Identities to Remember

`iter(o)` is a spelling for `o.__iter__()`

`next(o)` is a spelling for `o.__next__()`

Internals Rewritten

- The interpreter's logic looks something like this

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

- See example in notebook

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
```

For Later Study

If you've ever wondered how generators

and generator expressions work, you now

have the necessary concepts to understand them!

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 a particular kind of iterator!

```
>>> def gen_func():
...     yield 42
...
>>> f = gen_func()
>>> type(f)
<type 'generator'>
```

How Generators Work

- Generators are iterators
- Generator functions are iterator *factories*
  - Each call returns a new iterator instance
- Calling the generator's **`__next__`** method runs the function body code until the next **`yield`**
- 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']
```

Generator Functions as `__iter__`

- **Remember:**

  Calling a generator function returns a generator …
  … which is also an iterator!

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

Thanks For Listening!

?

steve.holden@digital.trade.gov.uk

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