PEP 234 – ITERATORS

Kevin Barry, Thomas Dressler, & Justin Spurgeon

Background

- In January of 2001, Yee and Rossum proposed a new iterator interface for objects in Python.
- The proposed implementation was an iterator object that allows programmers to loop over objects more easily and elegantly.

Background cont.

- The overarching goal of the proposal can be summarized as follows: performance enhancements for object iteration with respect to dictionaries, files, lists, and other objects implemented as collections and sequences.

Prior to Iterators

- Before iterators, iterating over lists, dictionaries, and collections each required very different syntaxes.
- Pre-iterator syntax typically relies on the use of while loops, in concert with the various means (list[i], dictionary.getValue(i), etc.) for obtaining member values.

- The while loop method requires having as much of the list/dictionary in memory as possible.

Syntactic Comparison between Pre-Iterator & Iterator Forms

Pre-Iterator Form:

```
instance = some_class(x,y)

while 1:

item = instance.f()

if not item:

break

do_something_with_item
```

Iterator Form:

```
for item iterating some_class(x,y).f:

DoSomethingWithItem
```

Prior to Iterators

- Iterators offer a very clean syntax compared to pre-iterator methods.
- Iterators offer similarly clean syntax for concatenating values, recursive generation, statistics (sum, min, max), and a wide variety of other functions

- Writing list comprehensions via iterators often uses a single line vs. several lines without iterators

Syntactic Comparison between Pre-Iterator & Iterator Forms, list comprehensions

Pre-Iterator Form:

```
instance = some_class(x,y)
S = []

while 1:

item = instance.f()

if not item:

break

if (item * 2 > 3):

S.append(item * 2)
```

Iterator Form:

```
instance = some_class(x,y)

S = [2 * x for x in instance if x ** 2 > 3]
```

C API Specification

- A new exception is defined, StopIteration
  - Can be used to signal the end of an iteration.
- StopIteration plays a large role in how the iterators work

tp_iternext

- The proposal called for a memory slot containing the next element of the collection.
- tp_iternext uses the PyIter_Next() method to obtain the next element.

tp_iternext

- Pyiter_Next() is a higher order function that takes an iterator object as it's argument.
- The next slide shows code that checks an iterator object for a Null value before iterating over a collection object.

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while (item = PyIter_Next(iterator)) {
    /* do something with item */
}
```

Checking for Null

- The significance of a Null return value for the PyIter_Next() method is simple:
  - No more items in the collection.
  - An error occurred.

Checking for Null

- This is much easier to interpret than the Null possibilities of the tp_iternext slot (which calls PyIter_Next()).

- Why?
  - The proposal also called for a new exception, “StopIteration,” to signal the exhaustion of a collection.

Checking for Null

- While the tp_iternext slot uses this exception (which returns Null), the function call to PyIter_Next() clears the exception.
  - The Null return value is subsequently easier to understand and work with.

The Proposal

- Difference between iterable and iterator
  - A container is said to be iterable if it has the __iter__ method defined
  - An iterator is an object that supports the iterator protocol:
    - It has an __iter__ method defined which returns itself
    - It has a next method defined which returns the next value every time the next method is invoked on it

Iterators

- Consider a list: A list is iterable, but a list is not its own iterator
- The iterator of a list is actually a listiterator object.
  - A listiterator is its own iterator

```
>>> a = [1, 2, 3, 4]
>>> # a list is iterable because it has the __iter__ method
>>> a.__iter__
<method-wrapper '__iter__' of list object at 0x014E5D78>
>>> # However a list does not have the next method, so it's not an iterator
>>> a.next
AttributeError: 'list' object has no attribute 'next'
>>> # a list is not its own iterator
>>> iter(a) is a
False
```

The Proposal

- Defined built-in function iter()
- Can be called one of two ways
  - iter(obj) calls PyObject_GetIter(obj)
  - iter(callable, sentinel)

The Proposal cont.

- iter(callable, sentinel)
  - Returns a special kind of iterator that calls the callable to produce a new value
    - Compares return value to sentinel
    - If it is equal to the sentinel, signals end of iteration
    - Otherwise return sentinel as next value from the iterator

The Proposal cont.

- Iterator objects returned have a next() method.
  - Method returns the next value in the iteration,
  - Or raises StopIteration to signal the end of the iteration.

The Proposal cont.

- An object can be iterated over with "for" if it implements __iter__() or __getitem__().
- An object can function as an iterator if it implements next().

Conclusion

- Six main benefits
  - 1. It provides an extensible iterator interface.
  - 2. It allows performance enhancements to list iteration.
  - 3. It allows big performance enhancements to dictionary iteration.
  - 4. It allows one to provide an interface for just iteration without pretending to provide random access to elements.
  - 5. It is backward-compatible with all existing user-defined classes and extension objects that emulate sequences and mappings
  - 6. It makes code iterating over non-sequence collections more concise and readable.