Iteration in Python

The for loop

```
>>> numbers = [ 0, 1, 2, 3 ]
>>> for n in numbers:
...     print(n)
```

Oddly enough, some interesting stuff is going on under the hood.

Iteration in Python

Lists have a magic method called `__iter__`.

```
>>> numbers.__iter__
<method-wrapper '__iter__' of list object at 0x10130e4c8>
>>> numbers.__iter__()
<list_iterator object at 0x1013180f0>
```

Iterators

An **iterator** is an object that produces the values in a sequence, one at a time.

```
>>> it = numbers.__iter__()
>>> for n in it:
...     print(n)
...
0
1
2
3
```

It's like a moving pointer into the collection.

Iterators work once

But you can only use an iterator once.

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> names_it = names.__iter__()
>>> for name in names_it:
...     print(name)
Tom
Shelly
Garth
>>> for name in names_it:
...     print(name)
>>> print("No output the second time!")
No output the second time!
```

Different iterators

But you can create multiple iterators. Each one has its own identity.

```
>>> names_it1 = names.__iter__()
>>> names_it2 = names.__iter__()
>>> for name in names_it1: print(name)
...
Tom
Shelly
Garth
>>> for name in names_it2: print(name)
...
Tom
Shelly
Garth
```

Different Iterators

They each have their own identity:

```
>>> id(names_it1)
4314993272
>>> id(names_it2)
4314993384
```

Magically Called Methods

Essentially, the `for` loop calls the `__iter__` method of whatever it's looping over, to get an iterator.

```
for name in names:
    print(name)
# just like:
names_iter = names.__iter__()
for name in names_iter:
    print(name)
```

If an object has an `__iter__` method, we say it is *iterable*.

What if...

What if you aren’t working with a simple sequence of numbers or strings, but something more complex?

What if you are calculating or reading or otherwise obtaining the sequence as you go along?

Square Processing

Let's say you need to do something with square numbers.

```python
MAX = 5
squares = []
for n in range(MAX):
    squares.append(n**2)
for square in squares:
    do_something_with(square)
```

This works. But...

Maximum MAX

What if MAX is not 5, but 10,000,000? Or 10,000,000,000? Or more?

What if you aren’t doing arithmetic to get each element, but making a truly expensive calculation? Or making an API call? Or reading from a database?

Now your program has to wait… to create and populate a huge list… before the FIRST loop can start.

Lazily Looping

The solution is to create an iterator to start with, which lazily computes each value just as it’s needed. Then each cycle through the loop happens just in time.

The Iterator Protocol

Here’s how you do it in Python:

```python
class Squares:
    def __init__(self, max_base):
        self.max_base = max_base
        self.current_base = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_base == self.max_base:
            raise StopIteration
        value = self.current_base ** 2
        self.current_base += 1
        return value

for square in Squares(5):
    print(square)
```

There’s got to be a better way

Good news. There’s a better way.

It’s called the **generator**.

You’re going to love it!

Yield for Awesomeness

A generator looks just like a regular function, except it uses the `yield` keyword instead of `return`.

```
>>> def gen_squares(max_base):
...     for n in range(max_base):
...         yield n * n
...
>>> for square in gen_squares(5):
...     print(square)
...
0
1
4
9
16
```

Generator Functions & Objects

The function with `yield` is called a **generator function**.

The object it returns is called a **generator object**.

```
>>> def gen_squares(max_base):
...     for n in range(max_base):
...         yield n ** 2
...
>>> squares = gen_squares(5)
>>> type(squares)
<class 'generator'>
```

Pop quiz

Create a Python file called `gensquares.py`. Copy this into it and run it:

```python
def gen_squares(max_base):
    for n in range(max_base):
        yield n ** 2

for square in gen_squares(5):
    print(square)
```

Give a thumbs up when you're done.

The next() thing

```
>>> squares = gen_squares(5)
>>> next(squares)
0
>>> next(squares)
1
>>> next(squares)
4
```

Stop the iteration

```
>>> # Keep going...
... next(squares)
9
>>> next(squares)
16
>>> next(squares)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Multiple Yields

You can have more than one yield statement.

```
>>> def myitems():
...     for n in range(3):
...         yield n*n
...     yield "All done"
...
>>> for item in myitems():
...     print(item)
...
0
1
4
All done
```

Lab: Generators

Lab file: `generators/generators.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `generators/generators_extra.py`

Dictionary Views & Iterators

Here's a Python 3 dictionary:

```
>>> calories = {
...     "apple": 95,
...     "slice of bacon": 43,
...     "cheddar cheese": 113,
...     "ice cream": 15, # You wish!
... }
>>> items = calories.items()
>>> type(items)
<class 'dict_items'>
>>> hasattr(items, '__next__')
False
>>> hasattr(items, '__iter__')
True
```

What is returned by .items()?

A *dictionary view* object.

Quacks like a dictionary view if it supports three things:

- `len(view)` returns the number of items
- `iter(view)` returns an iterator over the key-value pairs
- `key in view` returns True if `key` is in the dictionary, else False.

Iterable Views

A view is iterable, so you can use it in a for loop:

```
>>> for food, count in calories.items():
...     print("{:.<20s} {:<d} cal".format(food, count))
...
ice cream........... 15 cal
slice of bacon...... 43 cal
apple............... 95 cal
cheddar cheese...... 113 cal
```

Dynamically updates

A view dynamically updates, even if the source dictionary changes:

```
>>> items = calories.items()
>>> len(items)
4
>>> calories['orange'] = 50
>>> len(items)
5
>>> ('orange', 50) in items
True
```

Other methods

There are two other methods on dictionaries, called `.keys()` and `.values()`. They also return views.

```
>>> foods = calories.keys()
>>> counts = calories.values()
>>> 'yogurt' in foods
False
>>> 100 in counts
False
>>> calories['yogurt'] = 100
>>> 'yogurt' in foods
True
>>> 100 in counts
True
```

Benefits

Views improve over regular iterators:

- Providing a more dynamic view of the dictionary's contents
- Let you pass dict contents to caller and know it won't be modified
- Of course, more scalable/performant than a list of (key, value) pairs

What about Python 2?

All the above was for Python 3. Here's how it works in 2:

- `calories.items()` returns a list of `(key, value)` tuples.
  - So if it has 100,000 entries...

- `iteritems()`: returns an iterator over the key-value tuples
- `viewitems()`: which returned a view

`iteritems` is basically obsoleted by `viewitems`, but most people don't realize this yet.

Obsolete methods

In Python 3, what used to be called `viewitems()` was renamed `items()`, and the old `items()` and `iteritems()` went away.

If you still need an actual list in Python 3, you can just say `list(calories.items())`

Future-proofing "next"

```python
def gen_up_to(limit):
    n = 0
    while n <= limit:
        yield n
        n += 1

it = gen_up_to(10)

# Works in Python 3 only
it.__next__()
# Works in Python 2 only
it.next()
# Works in Python 2, 3, 4, ...
next(it)
```