Iteration in Python

Iterator Protocol

Any object in Python can be an iterator. It just needs to define proper `__iter__` and `__next__` methods.

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

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

We call this the *iterator protocol*.

GenObjs are Iterators

But not vice versa.

```
def gen_squares(max_root):
    for x in range(max_root):
        yield x ** 2
```

```
>>> gen_obj = gen_squares(5)
>>> generator_type = type(gen_obj)
>>> generator_type
<class 'generator'>

>>> hasattr(gen_obj, '__iter__')
True
>>> hasattr(gen_obj, '__next__')
True

>>> iterator = Squares(5)
>>> isinstance(iterator, generator_type)
False
```

Iterators

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

A list is not an iterator. But you can get an iterator over the list by using the built-in `iter()` function.

```
>>> numbers = [ 0, 1, 2, 3 ]
>>> # iter() calls numbers.__iter__()
... it = iter(numbers)
>>> for n in it: print(n)
0
1
2
3
# second time: no output
... for n in it: print(n)
>>>
```

You can only use it once.

Iterable (vs. "iterator")

An object is said to be **iterable** if it provides a way to spawn new iterators, by passing it to `iter()`.

Normally you never need to do this. `for` loops do it automatically.

```
>>> # A good working definition: an object is iterable
... # if you can pass it to more than one for loop.
... names = [ "Joe", "Tina", "Abdul"]
>>> for name in names: print(name)
Joe
Tina
Abdul
>>> for name in names: print(name)
Joe
Tina
Abdul
```

GenObjs are iterators (not iterable)

```
def gen_squares(max_root):
    for n in range(max_root):
        yield n * n
```

```
>>> def gen_squares(max_root):
...     for n in range(max_root):
...         yield n * n
>>> squares = gen_squares(2)
>>> for square in squares: print(square)
0
1
>>> for square in squares: print(square)
... # no output
>>>
```

Dictionary Views & Iteration

Here's a Python 3 dictionary:

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

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
- `view` is iterable
- `(key, value) in view` returns `True` if that pair is in the dictionary; else, `False`.

Iterable Views

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

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

What if you need list?

What if you need an actual list?

- Of key-value pairs?
- Or of the keys?
- Or of the values?

Then you just past the view to `list()`:

```
>>> list(calories.items())
[('apple', 95), ('slice of bacon', 43), ('cheddar cheese', 113), ('ice
cream', 15)]
>>> list(calories.keys())
['apple', 'slice of bacon', 'cheddar cheese', 'ice cream']
>>> list(calories.values())
[95, 43, 113, 15]
```

This works for any iterator or iterable.