COMP10001 Foundations of Computing
Iterators

Semester 2, 2016
Chris Leckie

October 2, 2016

Lecture Agenda

- Last lecture:
  - Images
- This lecture:
  - Iterators

for i in iterable

You are all very familiar with iterables

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

But what is going on under the hood?

What is Python doing for you?

- Imagine you could not use `for ... in ...`
- How would you do this?

```
for element in [1, 2, 3]:
    print(element)
```

- Something like...

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

What do you need to keep track of?

```
iterable = [1,2,3]
current_index = 0
while current_index < len(iterable):
    print(iterable[current_index])
    current_index += 1
```

- The iterable
- The current index in the iterable
- The end of the iterable

This is exactly what an Iterator keeps track of for you.

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

object	something you can manipulate
traverse	walk through/across
container	an object representing a collection of other objects (eg list, set, etc)

Iterators

- Definition: an iterator is an object that keeps track of the traversal of a container

object	something you can manipulate
traverse	walk through/across
container	an object representing a collection of other objects (eg list, set, etc)

- Definition: an iterable object will return an iterator object when you pass it to the built-in Python function `iter()`.
(This happens automatically with `for..in..:`)

Iterator Objects

- Iterators have a `__next__` method that will return the next thing in the iteration, and update their state/memory of where they are up to. You can access it with the built-in function `next()`
- Iterators raises a `StopIteration` exception when the container is empty

```
iterator = iter([1,2,3])
try:
    while True:
        print(next(iterator))
except StopIteration:
    pass
```

Why?!?!?

So I am telling you that instead of writing

```
for element in [1, 2, 3]: print(element)
```

you can write

```
iterator = iter([1,2,3])
try:
    while True: print(next(iterator))
except StopIteration:
    pass
```

Why?

- So you will know how to use iterators in other ways.

Worksheet 12: CSV

Recall in Worksheet 12 the `csv.reader()` was an iterator.

```
import csv
visitors = open("vic_visitors.csv")

data = csv.reader(visitors)  # an iterator

header = next(data)

for line in data:    # Python auto. expands
    print(line)      # this to try:..next()..

visitors.close()
```

So good programmers know how to use iterators.

Examples of Iterators

- What is the container in each case?
  - file objects (returned by `open()` or `urlopen()`)
  - `csv.reader()`
  - `Image.getdata()` from PIL (Pillow)
  - `range()`
  - generators

Iterators vs. Sequences

Iterators

- no random access
- "remembers" last item seen
- no `len()`
- can be infinite
- traverse exactly once (forwards)

Sequences

- supports random access
- doesn't track last item
- has `len()`
- must be finite
- "traverse" multiple times (fwd/rev/mix)

Why Use Iterators?

- Main reason: memory efficiency
- Process only one item at a time
- Avoid generating/copying a whole collection of items
- Access each item in turn

Iterable vs. Iterator

- *Iterable* describes something that could conceptually be accessed element-by-element
- *Iterator* is an actual interface (or an object implementing the interface) allowing element-by-element access to its contents
- We use an iterator to access an iterable object

The `itertools` Module

- Implements a number of iterator "building blocks"
- Inspired by other programming languages (APL, Haskell, SML)
- Standardises a set of fast, memory efficient tools
- Each tool can be used alone or in combination
- Forms an "iterator algebra"

product: Cross-product of Sequences

```
import random
from itertools import product
def get_deck(shuffle=False):
    suits = 'CDHS'
    values = '234567890JQKA'
    deck = product(values, suits)
    deck = [''.join(c) for c in deck]
    if shuffle:
        random.shuffle(deck)
    return (deck)
```

cycle: Repeating Items Indefinitely

```
from itertools import cycle
def deal(players=4):
    deck = get_deck(shuffle=True)
    hands = [[] for i in range(players)]
    players = cycle(hands)
    for card in deck:
        player = next(players)
        player.append(card)
    return(hands)
```

Example Uses

```
>>> deck = get_deck()
>>> len(deck)
52
>>> deck[:7]
['3C', '3S', '3D', '3H', '4C', '4S', '4D']
>>> hands = deal()
>>> hand1 = sorted(hands[0])
>>> hand1
['2C', '3H', '3S', '4C', '4S', '5S', '6D',
 '7D', '8D', 'AC', 'AD', 'AH', 'AS']
```

groupby: Group Items by Some Criterion

```
>>> from itertools import groupby
>>> def first(x): return(x[0])
>>> for rank, group in groupby(hand, first):
...     print("{} {}".format(rank, list(group)))
2 ['2C']
3 ['3H', '3S']
4 ['4C', '4S']
5 ['5S']
6 ['6D']
7 ['7D']
8 ['8D']
A ['AC','AD','AH', 'AS']
```

combinations: *n* Choose *k*

```
>>> from itertools import combinations
>>> aces = ['AC','AD','AH', 'AS']
>>> combinations(aces,2)
<itertools.combinations object at 0x1794998>
>>> list(combinations(aces,2))
[('AC', 'AD'), ('AC', 'AH'), ('AC', 'AS'),
 ('AD', 'AH'), ('AD', 'AS'), ('AH', 'AS')]
```

Lecture Summary

- What is an iterator?
- Why is it useful?
- Differences between sequences and iterators
- The `itertools` module