Advanced Topics: Biopython
Day One - Iterators

Peter J. A. Cock
The James Hutton Institute, Invergowrie, Dundee, DD2 5DA, Scotland, UK

23rd – 25th January 2012,
Workshop on Genomics, Český Krumlov, Czech Republic
Talk Outline

1. What are iterators?
2. Usage
3. Creating an iterator
4. Exercises
5. Evening Class
What are iterators?

First I’ll show you some abstract definitions

Then I’ll describe them in terms of usage
What are iterators?

In computer programming, an iterator is an object that enables a programmer to traverse a container. Various types of iterators are often provided via a container’s interface. . . . An iterator is behaviorally similar to a database cursor.

http://en.wikipedia.org/wiki/Iterator
An object representing a stream of data. Repeated calls to the iterator’s `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. ...
What are Python iterators?

Iterators are objects (can be functions or methods) which give their values one by one (often in a for loop), e.g.

- Every line in a file
- Every entry in a list
- Every letter in a string
- Every prime number

BUT, you can only do this once.
Both Python iterators and Python sequences (lists, tuples, strings, etc) can be used in for loops.

- Sequences also support indexing (square brackets)
- Sequences have a known length
- Sequences are in memory, iterators usually not
- Iterators can be infinite
- Iterators can only be looped over once
Lines in a file - list

```python
with open("example.txt") as handle:
    lines = handle.readlines()

print len(lines)

for line in lines:
    if "Hello" in line:
        print line

total = 0
for line in lines:
    total += len(line)
print total
```
#File handles are iterators,
lines = open("example.txt")

#This fails,
print len(lines)

#You can get the count like this,
count = 0
for line in lines:
    count += 1
print count

#However, the iterator is now exhausted (empty).
#The handle is at the end of the file.
File handles are iterators,
lines = open("example.txt")

Can do everything in one pass though the file
count = 0
total = 0
for line in lines:
    count += 1
    if "Hello" in line:
        print line
    total += len(line)
print count
print total

lines.close()
Lines in a file

Loading a file as a list of strings is more flexible:

- You can loop over them multiple times
- You can access lines by indexing
- Might even be faster

However, there is a major downside:
- Using the list puts everything into memory!

Iterating over the file will let you work with large files
Creating an iterator

- Use `iter()` on an existing list, tuple, string, etc
- Use existing functions, e.g. `open`
- Create an iterator object
  (see `__iter__` and `next` methods)
- Create a generator function
- Write a generator expression (one line)

Usually iterators are defined in terms of other iterators - the module `itertools` can be very useful
http://docs.python.org/library/itertools.html
Selecting lines from a file

Returning to the earlier example, this for loop an iterator (a file handle) and finds just those lines with the word “Hello” in them:

```python
with open("example.txt") as handle:
    for line in handle:
        if "Hello" in line:
            print line
```

Very simple, but now let’s look at other ways to write it.
What are iterators?

Usage

Creating an iterator

Exercises

Evening Class

Lines from a file - function

This function takes an iterator (a file handle) and returns a list of matched lines:

```python
def wanted_lines(handle):
    wanted = []
    for line in handle:
        if "Hello" in line:
            wanted.append(line)
    return wanted

with open("example.txt") as handle:
    for line in wanted_lines(handle):
        print line
```

This solution could run out of memory if there are lots of matching lines!
This *generator function* takes an iterator (a file handle) and returns the matched lines one by one (using keyword *yield*):

```python
def wanted_lines(handle):
    for line in handle:
        if "Hello" in line:
            yield line

with open("example.txt") as handle:
    for line in wanted_lines(handle):
        print line
```

This specific *generator function* is acting like a filter.
I hope you’re familiar with *list comprehensions* in Python?

```python
with open("example.txt") as handle:
    wanted = [line for line in handle \n              if "Hello" in line]

#Variable wanted is a list
for line in wanted:
    print line
```

It is trivial to turn this into a *generator expression*
List comprehensions use square brackets, generator expressions use round brackets:

```python
with open("example.txt") as handle:
    wanted = (line for line in handle \ #Variable wanted is now a generator, not a list!
        if "Hello" in line)
    for line in wanted:
        print line

This was new in Python 2.4, see http://www.python.org/dev/peps/pep-0289/
```
Tip - range versus xrange

Not that for Python 2, the built in functions range and xrange return lists and iterators respectively:

```python
>>> range(4)
[0, 1, 2, 3]
>>> xrange(4)
xrange(4)
>>> for i in xrange(4):
...    print i
0
1
2
3
```

Python 3 moves to just having range, which returns an iterator
Complete this example using a generator function,

```python
def odd_filter(values):
    """Filter to return just odd integers."""
    for value in values:
        if ...:
            yield value

for i in odd_filter(xrange(20)):
    print i
```
Even/odd numbers

Complete this example using a generator expression,

odd_values = (value for value in xrange(20) if ...)

for i in odd_values:
    print i
Complete this example using a generator expression to select sequences of at least length 100.

```python
from Bio import SeqIO

records = SeqIO.parse("genes.fasta", "fasta")

long_records = (rec for rec in records if ...)

count = SeqIO.write(long_records, "long_genes.fasta", "fasta")

print "Saved %i long sequences" % count
```
Arithmetic progression

Complete this example using to give an infinite sequence of numbers, each time incremented by the step size given:

```python
def arithmetic_progression(start, step):
    """Returns start, start+step, start+2*step, ... ""
    yield start
    #...

for value in arithmetic_progression(1,2):
    print value
    if value > 100: break
```
Challenges

Read a FASTA file with many sequences using `SeqIO.parse` and:

- calculate the mean
- store the lengths in a dict
- ... and then draw a histogram (clue?)
- ... and then calculate the median (hard)

Tip:

```python
from Bio import SeqIO
for record in SeqIO.parse("example.fasta", "fasta"):
    print(len(record))
```