Advanced Topics: Biopython

Day Three

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 Optimization and Profiling

2 Profiling in Python

3 Testing

Optimization and Profiling

- Once you’ve got some working code, you may want or need it to run faster
- With experience you may be able to guess what is slow
- If in doubt, profile it (measure it)

Premature optimization

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”

Donald Knuth
(author of ‘The Art of Computer Programming’, etc)

Optimization Costs

- Optimization is *not* free
- Efforts to speed code up may be wasted
- You risk breaking things (unit tests help)
- Beware of trading speed for higher memory

If you double the speed, will it actually save you any time?

Only optimize if you need to

1. Get it right
2. Test really is right
3. Profile if slow
4. Optimize
5. Repeat from 2

Profiling in Python

- Try the `profile` module
  `http://docs.python.org/library/profile.html`
- Can be run 'live'
- Can be run with profiling logged to a file

Example: Counting FASTQ with SeqIO.parse

```
from Bio import SeqIO

def count_seqio(fastq_filename):
    count = 0
    for record in SeqIO.parse(fastq_filename, "fastq"):
        count += 1
    print "%i records in %s" % (count, fastq_filename)
    return count

import profile
profile.run("count_seqio('SRR014849.fastq')",
            sort="cumulative")
```

Example: Counting FASTQ with SeqIO.parse

```
94696 records in /Users/pjcock/repositories/examples/SRR014849.fastq
         2557126 function calls (2462430 primitive calls) in 24.585 CPU seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   24.585   24.585 profile:0(count_seqio(filename))
        1    0.000    0.000   24.578   24.578 <string>:1(<module>)
        1    0.595    0.595   24.578   24.578 count_fastq.py:6(count_seqio)
    94697    0.863    0.000   23.983    0.000 __init__.py:446(parse)
    94697    4.803    0.000   23.115    0.000 QualityIO.py:945(FastqPhredIterator)
    94697    4.753    0.000    9.037    0.000 QualityIO.py:788(FastqGeneralIterator)
    94696    2.734    0.000    6.247    0.000 SeqRecord.py:147(__init__)
473482/378786    2.168    0.000    2.839    0.000 :0(len)
   378785    1.803    0.000    1.803    0.000 :0(readline)
   378786    1.558    0.000    1.558    0.000 :0(isinstance)
   378784    1.430    0.000    1.430    0.000 :0(rstrip)
    94696    0.736    0.000    1.099    0.000 Seq.py:71(__init__)
    94696    0.670    0.000    0.986    0.000 Seq.py:200(__len__)
    94696    0.727    0.000    0.727    0.000 :0(min)
    94696    0.702    0.000    0.702    0.000 :0(max)
    94696    0.531    0.000    0.531    0.000 SeqRecord.py:67(__init__)
    94696    0.498    0.000    0.498    0.000 :0(split)
        1    0.007    0.007    0.007    0.007 :0(setprofile)
...
```

Example: Counting FASTQ with SeqIO.parse

- Note these times are slower than with no profiling
- Overall time for `SeqIO.parse` was 23.983 seconds
- Of this 9.037 seconds was for `FastqGeneralIterator`
- Big overhead creating `SeqRecord` objects, 6.247 seconds
- We don't need the `SeqRecord` objects here...

See also `http://news.open-bio.org/news/2009/09/biopython-fast-fastq/`

Example: Counting FASTQ with FastqGeneralIterator

```
from Bio.SeqIO.QualityIO import FastqGeneralIterator

def count_low(fastq_filename):
    count = 0
    with open(fastq_filename) as handle:
        for values in FastqGeneralIterator(handle):
            count += 1
    print "%i records in %s" % (count, fastq_filename)
    return count

import profile
profile.run("count_seqio('SRR014849.fastq')",
            sort="cumulative")
```

Example: Counting FASTQ with FastqGeneralIterator

```
94696 records in /Users/pjcock/repositories/examples/SRR014849.fastq
         1136359 function calls in 9.415 CPU seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    9.415    9.415 profile:0(count_low(filename))
        1    0.000    0.000    9.415    9.415 <string>:1(<module>)
        1    0.392    0.392    9.415    9.415 count_fastq.py:21(count_low)
    94697    4.789    0.000    9.024    0.000 QualityIO.py:788(FastqGeneralIterator)
   378785    1.728    0.000    1.728    0.000 :0(readline)
   378784    1.451    0.000    1.451    0.000 :0(rstrip)
   284087    1.055    0.000    1.055    0.000 :0(len)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(open)
        1    0.000    0.000    0.000    0.000 :0(__enter__)
        0    0.000             0.000          profile:0(profiler)
```

Testing

- Write some tests *before* trying to optimise!

Only optimize if you need to

1. Get it right
2. Test really is right
3. Profile if slow
4. Optimize
5. Repeat from 2

Fast and wrong is *NOT* useful!