Week 14 - Lecture 27

István Albert
Huck Institutes for the Life Sciences
To INFINITY ... and BEYOND!
Infinite datasets

• Large dataset → the meaning of the word LARGE changes in time

• What is the limiting factor? The data processing or data representation?

• Infinite datasets → stream that gets processed
Iterators

• Iterators are objects that have a method called `next()` that returns the next element.

• We don’t usually need to call the `next()` method ourselves. It gets automatically called while iterating.
Get the iterator with the `iter()` function

```python
letters = "ABC"

# obtain the iterator from the string
it = iter(letters)
print(type(it))

# every iterator has next() method
print(it.next())
print(it.next())
```

Command Output:
```
<type 'iterator'>
A
B
```
Loops use the iterators

```python
# explicit loops, map, filter
# actually iterates behind the scenes

letters = "AB"
for c in letters:
    print c

# is equivalent to
for c in iter(letters):
    print c
```

```
A
B
A
B
```
Iterators can be moved forward

# we can advance the iterator ourselves

```
letters = "ABCDEFG"
it = iter(letters)

for c in it:
    if c == 'C':
        it.next()
        it.next()
    else:
        print c
```
File streams are iterators!

```python
# using gene-expression.txt from week 4

# streams are iterators
stream = file('gene-expression.txt')

print stream.next()
print stream.next()
```

```
#ID_REF = ID_REF

#CH1I_MEAN = Mean feature pixel intensity at wavelength 532 nm.; Type: integer; Scale:
```
Unwinding the iterator

```python
# iterators are for single use!

stream = file('gene-expression.txt')

# keeps calling the iterator
# and assembles everything into a list
rows1 = list(stream)
print len(rows1)

# the stream iterator has been completely used up
rows2 = list(stream)
print len(rows2)
```

```
37687
0
```
Two iterators over the same data

```python
# you may open two independent
# streams into the same file

stream1 = file('gene-expression.txt')
stream2 = file('gene-expression.txt')

rows1 = list(stream1)
print len(rows1)

rows2 = list(stream2)
print len(rows2)
```
Iterator recap

- Use minimal memory regardless of the data size
- Can be run forward only, can’t go back.
- Getting a value moves the iterator to next element
- Single use only at the end they run out

**BUT**

- You may usually run two iterators on the same underlying data structure
Infinite iterators

• Can be merged with finite data to produce certain types of behaviors ⇒ counting, cycling, repeating, chaining results

• Don’t unwind a an infinite generator ⇒ it has no end

• Many of the solutions are nicer and simpler with an iterator
from itertools import count

d = count(100)
print(type(d))
print(d.next())
print(d.next())
Counting elements

```python
from itertools import count

vals = count(0)
lett = "ABCD"

for elem in zip(vals, lett):
    print elem
```

Output:

```
(0, 'A')
(1, 'B')
(2, 'C')
(3, 'D')
```
Counting in a less optimal way
not a data structure anymore

```python
index = 0
lett = "ABCD"

for elem in lett:
    print(index, elem)
    index = index + 1
```
Careful with endless loops!

```python
from itertools import count

vals = count(0)

# the count is infinite,
# there is nothing to stop this loop
for elem in vals:
    print(elem)

# PS: DON'T DO IT, I KNOW YOU WANT TO!
```

Running `python -tt C:/cygwin/home/albert/sources/albert-web/ppt/week14/lecture27/dont.py` shows the output:

```
171689
171690
171691
171692
17169
```
from itertools import repeat

# if you don't pass the last parameter (5)
# the repeat will become infinite!
text = repeat('Yes!', 5)

for word in text:
    print(word)
from itertools import cycle

# cycle is infinite! so a limiter is required
status = cycle(["On", "Off"])
data = "ABCDE"

for elem in zip(data, status):
    print elem
“silent killer” – implicit infinite loop

```python
from itertools import count, cycle, repeat

# this only repeats twice
print zip(count(), repeat('HAHA', 2), cycle('ABC'))

# this is infinite
# print zip(count(), repeat('HAHA'), cycle('ABC'))
```

```
[(0, 'HAHA', 'A'), (1, 'HAHA', 'B')]
```
from itertools import count, cycle, repeat, chain

gimme_an_A = repeat('A', 2)

# chain concatenate iterators
for row in chain(gimme_an_A, "XY", range(3)):
    print(row)
Homework 1/2

• Solve the problems on the next pages with iterators

• Provide either the code or the output

• If it is output then you should try to determine what the output is without running the code!
Problem 1

```python
from itertools import count, cycle, repeat, chain

numbers = count()
values = range(1, 100, 25)

for row in zip(numbers, values):
    print(row)
```
Problem 2

```python
from itertools import count, cycle, repeat, chain

numbers = count()
values = range(1, 100, 25)

for row in zip(numbers, numbers, values):
    print(row)
```
Problem 3

```python
from itertools import count, cycle, repeat, chain

for message in zip(questio, answer):
    print message
```

Command Output:

```
('Are you ready?', 'Yes')
('Are you ready?', 'No')
('Are you ready?', 'Maybe')
('Are you ready?', 'Yes')
```
```python
index = count(100)
data = range(4)
color = cycle([ 'Red', 'Green', 'Blue' ])

for row in zip(index, data, color, color):
    print row
```
Problem 5

```python
from itertools import count, cycle, repeat, chain

a = repeat('A', 1)
b = repeat('B', 2)
c = repeat('C', 1)

abc = chain(a, b, c)

for row in zip(count(100), abc):
    print(row)
```
Problem 6

```python
from itertools import count, cycle, repeat, chain

a = repeat('A', 1)
b = repeat('B', 2)
c = chain(a, b)
d = cycle(c)

for row in zip("abcde", d):
    print(row)
```
Not a homework but for fun which iterator stops the loop?

```python
from itertools import *

Jack = repeat('Jack')
And = repeat('and')
Jill = repeat('Jill')
Went = cycle(('fetch', 'fell'))
Up = cycle(('water', 'down'))
The = chain(repeat('uphill', 2), repeat('ouch', 1))
Hill = repeat('!')

for row in zip(Jack, And, Jill, Went, Up, The, Hill):
    print row
```
Week 14 - Lecture 28

István Albert

Huck Institutes for the Life Sciences
More on iterators

Some of these labels are not Apple related products but Python iterator tools
Compute the sum of N numbers

```python
import time

# set a smaller value first and see
# how much memory it uses
# the list below will contain 50
# million elements
N = 5*10**7
data = range(N)

print "Type", type(data)
print "Sum", sum(data)
print "Size", len(data)

time.sleep(15)
```

Uses almost 1 Gb
Same result with xrange

```python
import time

# set a smaller value first and see
# how much memory it uses
# the list below will contain 50
# million elements
N = 5*10**7
data = xrange(N)

print "Type", type(data)
print "Sum", sum(data)
print "Size", len(data)

time.sleep(15)
```

Memory use is not noticeable
Naming conventions for iterator tools

- Usually they are labeled with i* → imap, ifilter, izip etc

- Imported from the itertools module

- One exception: `xrange` instead of `range` it was introduced before the iterator formalism was established – plus it has some extra attributes (like length)
imap and ifilter
like a gearbox produce one value at each turn of the “crank” next()
izip vs zip

```python
from itertools import *

def square(x):
    return x * x

data = xrange(10)
res1 = map(square, data)
print(res1)
res2 = imap(square, data)
print(res2.next())
print(res2.next())
print(list(res2))
```

Output:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
0
1
[4, 9, 16, 25, 36, 49, 64, 81]
```
zipping large lists

```python
import time
from itertools import *

counter = count()
counter = islice(counter, 10**7)

start = time.time()
data = zip(counter, repeat('X'))
end1 = time.time()
print 'Creation %4.2f sec' % (end1 - start)

for elem in data:
    elem = None # doing nothing
end2 = time.time()
print 'Iteration %4.2f sec' % (end2 - end1)

# this to allow us to investigate
#the memory footprint
time.sleep(10)
```

Execution time is surprisingly bad!

Memory footprint
Optimized zip – garbage collector off

this is an internal feature that leads to a **pathological** behavior in this piece of code

but rather than turning gc off use the iterator as shown in the next slide
iZipping large lists

```python
import time
from itertools import *

counter = count()
counter = islice(counter, 10**7)

start = time.time()
data = izip(counter, repeat('X'))
end1 = time.time()
print 'Creation %4.2f sec' % (end1 - start)

for elem in data:
    elem = None  # doing nothing
end2 = time.time()
print 'Iteration %4.2f sec' % (end2 - end1)

# this to allow us to investigate
#the memory footprint
time.sleep(10)

python -u /cygwin/home/albert/sources/albert-web/ppt/week14/lecture28.zip.py 2>&1 returned 0

Creation 0.00 sec
Iteration 1.98 sec
```

Memory footprint

Compare to previous slide
Slicing iterators

```python
from itertools import count, islice

N = 10**2
data1 = range(N)

# default method of slicing
print(data1[10:15])

# this is infinite iterator
# that we will slice to limit
counter = count()
result = islice(counter, 10, 15)

# unwind the iterator
print(list(result))
```

```
[10, 11, 12, 13, 14]
[10, 11, 12, 13, 14]
```
Slicing a stream/reader

```python
import csv
from itertools import *

stream = file('saccharomyces_cerevisiae.gff')
reader = csv.reader(stream, delimiter='\t')

# limit the reader from 100 to 110
reader = islice(reader, 100, 110)

# this is the 100th row
print reader.next()

# this is the 101th row
print reader.next()
```
Equivalence

map and filter are iterators that are unwound

map(something) \rightarrow list(imap(something))

filter(something) \rightarrow list(ifilter(something))
Reading our gff file
there is sequence at the end – we need to remove that
Slice to that index

```python
import csv
from itertools import *

stream = file('saccharomyces_cerevisiae.gff')
reader = csv.reader(stream, delimiter='\t')
reader = islice(reader, 19, 16423)

print reader.next()
```

```
['chrI', 'SGD', 'chromosome', '1', '230208', '.', '.','ID=chrI;dbxref=NCBI:NC_001133']
```
There is a better way
takewhile, dropwhile

```python
import csv
from itertools import *

def condition(row):
    return row[0] != '#FASTA'

stream = file('saccharomyces_cerevisiae.gff')
reader = csv.reader(stream, delimiter='\t')
reader = takewhile(condition, reader)

# exhaust the iterator to check last element
for row in reader:
    pass

# print last element
print row
```

Output:

```
['##
```
import csv
from itertools import *

def condition(row):
    return row[0] != '##FASTA'

def datacheck(row):
    # in gff each data row must have 9 fields
    return len(row) == 9

stream = file('saccharomyces_cerevisiae.gff')
reader = csv.reader(stream, delimiter='\t')
reader = takewhile(condition, reader)
reader = ifilter(datacheck, reader)

print reader.next()
Filter for the genes

```python
return row[0] != '#FASTA'

def datacheck(row):
    # in gff each datarow must have 9 fields
    return len(row) == 9

def getgenes(row):
    return row[2] == 'gene'

stream = file('saccharomyces_cerevisiae.gff')
reader = csv.reader(stream, delimiter='t')
reader = takewhile(condition, reader)
reader = ifilter(datacheck, reader)
reader = ifilter(getgenes, reader)

print reader.next()
```
Homework 2/2

• In the gff format the fourth and fifth columns contain the start and end coordinates for the genes. Use imap to extract these.

• Generate a histograms of the gene length distribution for all genes. Choose enough bins to show a finer structure.

• Create a histogram that only plots the lengths that are smaller than 2000bp