Iteration in Python

Abram Hindle
abram.hindle@ualberta.ca
Department of Computing Science
University of Alberta
http://softwareprocess.es/

Warning

- This talk is not completely pythonic
- This talk is not completely idiomatic

Iteration

- Programming is useful because we can repeat tasks
- Iteration is one of the fundamental building blocks of most programming languages
- Iteration usually refers to repetition
  - In mathematics iterating a functions means applying it repeatedly
  - In programming iteration can refer to any method of repetition.

Concepts

- Block
  - A chunk of code
  - Usually the part that gets repeated
  - Usually follows scope rules
- Condition
  - An expression that evaluates to true or false
  - Commonly used to determine if a loop continues

Concepts

- Strict – Everything gets done
- Lazy – Something gets done when needed.
- In Order – Execute in Sequence
- Out of Order – Execute in any order
- Dependency – One value depends on another

While Loops

- Repeats a block until a condition is met.
- Loop invariant executes first
-

```
condition = True
while condition:
    """block"""
    condition = not condition
# condition is False

#how many lines ?
count = 0
while sys.stdin.readline() != '':
    count += 1

# maybe you're not sure how many
# iterations you need?
x = 100.0
while x > 1:
    x = x / 3
```

For Loop

- Iterate over an *Iterable* (collection or a range) in order
- Ranges let you loop a set number of times

```
sum = 0
for i in range(1,10):
    sum = sum + i
# sum is 45
sum = 0
for i in xrange(1,100000):
    sum = sum + i
# sum is 4 999 950 000
s = ""
for elm in ["a","b","c"]:
    s = s + elm
# s is abc
```

Iterable

- Object with a `next()` method
- raises `StopIteration`

```
class OnlyEvens(object):
    def __init__(self,s):
        self.sequence = s
        self.index = 0

    def __iter__(self):
        return self

    def next(self):
        if self.index >= len(self.sequence):
            raise StopIteration
        v = self.sequence[self.index]
        self.index += 2
        return v
```

```
oe = OnlyEvens(range(1,10))
for even in oe:
    print(even)
```

Recursion

- Arbitrary flow control
- Good for iterating datastructures like trees
- Watch out for stackoverflows!

```
def recsum(l,i=0):
    if (i < len(l)):
        return l[i] + recsum(l,i+1)
    else:
        return 0

recsum(range(1,10))
```

Order

- Did you notice something?
- Everything iterated in order.
- But what if order doesn't really matter?

Map

- In mathematics iterating a functions means applying it repeatedly
- A map function applies 1 function to all elements in a collection and produces a new collection of the results of that function
  - Usually this is in order
  - But you don't have to do it in order

Map Example

```
# add 1 to a list
v = [1,2,3]
u = map((lambda x: x+1), v)
# u is now [2,3,4], v is still [1,2,3]

def basename(path):
    return path.split("/")[-1]

v = ["/home","/","/usr/local"]
u = map(basename, v)
# u = ['home', '', 'local']

import urllib2
urls =
["http://cbc.ca","http://gc.ca","http://alberta.ca"]
def get_url(url):
    return urllib2.urlopen(url).read()

pages = map(get_url, urls)
```

Parallelism with Map

- If you think in “map” then you can parallelize with map
- Limit dependencies of a block in order to parallelize the computation!

```
# this is why you want blocks with
# few dependencies!
import multiprocessing as multi
def square(x):
    return x * x

p = multi.Pool( processes=8 )
u = p.map(square, range(1,1000000))
len(u)
#999999
#Network IO often parallelizes well
pages = p.map(get_url, urls)
```

Reduce

- Linear
- 1 at a time
- Collapse a collection in a single value
- Sum is a reduce

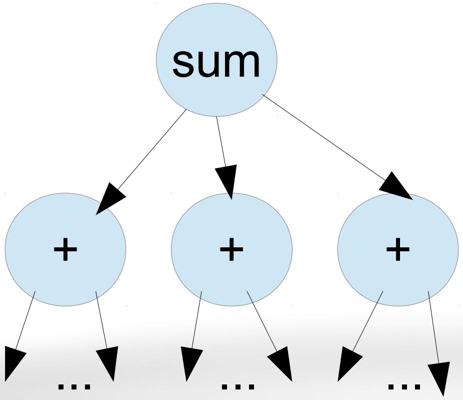
```python
import operator
l = range(1,1000000)
u = reduce(operator.add, map(square, l))
v = sum(map(square, l))
```

Reduce Can Be Parallel (Sometimes)

```
import operator
l = range(1,1000000)

# parallel map
def parallel_square(l):
    p = multi.Pool( processes=2 )
    return p.map(square, l)

# parallel reduce
def parallel_sum(l):
    p = multi.Pool( processes=2 )
    return sum(p.map(sum,
      [ l[0:len(l)/2], l[len(l)/2:len(l)] ]))

parallel_sum( parallel_square(l))
```

Trees, Communtativeness and Initialization

- Can your problem be modelled as a TREE?
- Problems with commutative or associative parts can often be modelled as a tree of computation.
- Different branches may be executed in Parallel.
- One can reduce dependencies by avoid initialization (e.g. sum = 0)



Conclusions

- Main forms of python iteration:
  - For / While / Iterable / Recursion / Map / Reduce
- Reducing dependencies in blocks allows iteration to be parallelized.
- Consider if order or strictness can be are actually needed?

- While – unknown number of iterations, unknown size
- For – iterate over a collection, iterate a fixed number of times
- Recursion – specialized iteration, often amenable to trees.
- Iterable – build OO collections that are compatible with For
- Map and reduce apply functions to collections