Python for Scientific Computing

Lecture 4: Code Optimization

Esteban Meneses
emeneses@pitt.edu
Center for Simulation and Modeling (SaM)

September 23, 2013

What does *Code Optimization* mean?

- Algorithm Design
  - Implementation alternatives
  - Algorithmic alternatives
- Profiler
- Performance recommendations

Premature optimization is the root of all evil
Donald Knuth

Exercise 1

Write a Python function to compute the *n-th Fibonacci* number.

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

Example:

```
> fibonacci(10)
55
```

Recursive Fibonacci Function

```
def fibonacci(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

What is recursion?

- Mechanism to solve problems through recursive functions
- A recursive function calls itself (directly or indirectly) with a different set of parameters
- Matches the mathematical description of many processes
- Recursive solutions are usually concise
- Each recursive call requires a new context to be created
- Typical in functional programming languages (Lisp)

What is iteration?

- Mechanism to solve problems through repetition of operations (loops)
- Handles the dependence between values across iterations naturally
- Scientific algorithms tend to be iterative in nature
- No new context is necessary for each iteration
- Typical in imperative programming languages (Fortran, C)

Iterative Fibonacci function

```
def fibonacci_iter(n):
  fn_2 = 0
  fn_1 = 1
  for i in xrange(1,n):
    fn = fn_1 + fn_2
    fn_2 = fn_1
    fn_1 = fn
  return fn
```

Recursion vs Iteration

- Recursion is usually more elegant (clear, concise)
  - Functional solutions express *what* to compute
- Iteration is generally more efficient
  - Imperative solutions express *what* and *how* to compute
- Tradeoff between elegance and performance
- Recursion and iteration are theoretically equivalent

Python Profiler

- Provides a performance description of a program
- Magic functions in IPython
  - Special commands to modify execution or environment
  - Examples: %edit, %run, %env
  - Get timing information: %time, %timeit
  - Get profile information: %prun
- Line profiler
  - Install `line_profiler` and `kernprof` modules

Exercise 2

Write a Python function to sort a list of integers in increasing order using the *bubble sort* algorithm.

`The bubble sort algorithm sweeps a list of integers L swapping values L[i] and L[i+1] if L[i]>L[i+1]. It repeats this process until no elements are swapped.`

Example:

```
> L = [9,8,7,6,5,4,3,2,1]
> bubble_sort(L)
> L
> [1,2,3,4,5,6,7,8,9]
```

Bubble Sort Function

```
def bubble_sort(list):
    """ Sorts a list using bubble sort algorithm """
    change = True
    while change:
        change = False
        for j in xrange(len(list)-1):
            if list[j] > list[j+1]:
                list[j],list[j+1] = list[j+1],list[j]
                change = True
```

Line Profile of Bubble Sort

```
./kernprof.py -l -v profile.py
```

```
Wrote profile results to profile.py.lprof
Timer unit: 1e-06 s

File: profile.py
Function: bubble_sort at line 21
Total time: 2.38207 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    21                                           @profile
    22                                           def bubble_sort(list):
    23                                               """ Sorts a list using bubble sort algorithm """
    24         1            5      5.0      0.0      change = True
    25       969          870      0.9      0.0      while change:
    26       968          917      0.9      0.0          change = False
    27    968000       850571      0.9     35.7          for j in xrange(len(list)-1):
    28    967032       999759      1.0     42.0              if list[j] > list[j+1]:
    29    244228       302448      1.2     12.7                  list[j],list[j+1] = list[j+1],list[j]
    30    244228       227496      0.9      9.6                  change = True
```

Merge Sort Function

```
def merge_sort(list):
    """ Sorts a list using merge sort algorithm """
    if len(list) == 1:
        return
    middle = len(list)/2
    left = list[0:middle]
    right = list[middle:len(list)]
    merge_sort(left)
    merge_sort(right)
    merge(left, right, list)
```

Merge Function

```python
def merge(left, right, list):
    """ Marges left and right sublists into list """
    left_max = len(left)-1
    right_max = len(right)-1
    left_index = 0
    right_index = 0
    for i in range(len(list)):
        if left_index > left_max:
            list[i] = right[right_index]
            right_index += 1
            continue
        if right_index > right_max:
            list[i] = left[left_index]
            left_index += 1
            continue
        if left[left_index] < right[right_index]:
            list[i] = left[left_index]
            left_index += 1
        else:
            list[i] = right[right_index]
            right_index += 1
```

Algorithmic Complexity

- A measure of how many operations are performed per input value
- Described as a function of n, the input size
- Sorting algorithms:
  - Bubble sort: $O(n^2)$
  - Merge sort: $O(n \log(n))$

Iterators

- Efficient in the use of memory
- On-demand object creation
- Example: `range` vs `xrange`

```
for x in range(10)
  foo(x)
```

```
for x in xrange(10)
  foo(x)
```

Map Function

- Applies a function to each element of a list
- Works directly if each operation is independent
- Avoids overhead of `for` loop
- Example:

```
final_list = []
for x in list:
    final_list.append(foo(x))
```

```
final_list = map(foo, list)
```

- List comprehension is a syntactic sugar of map

```
final_list = [foo(x) for x in list]
```

Join Operation on Strings

- Faster than loops to accumulate results in a list
- Avoids overhead of `append` function
- Example:

```
s = ""
for x in list:
    s += foo(x)
```

```
list_foo = [foo(item) for item in list]
s = "".join(list_foo)
```

Local Variables

- Faster to access than global variables
- Safer, more modular code
- Example:

```
sum = 0
def foo(n):
    global sum
    for x in xrange(n+1):
        sum += x
```

```
sum = 0
def foo(n):
    sum = 0
    for x in xrange(n+1):
        sum += x
    return sum
sum += foo(n)
```

Exceptions

- Model abnormal behavior
- Disrupts the execution flow
- Use `raise` command to throw an exception
- Use `try` and `except` to handle an exception
- Example:

```python
def foo(n):
  if n < 0:
    raise Exception("Negative Value")
try:
  foo(n)
except Exception:
  foo(-n)
```

Exceptions (cont.)

- Faster than conditional statements
- Example:

```
repetitions = {}
for word in words:
  if word not in repetitions:
    repetitions[word] = 1
  else:
    repetitions[word] += 1
```

```
repetitions = {}
for word in words:
  try:
    repetitions[word] += 1
  except KeyError:
    repetitions[word] = 1
```

Concluding Remarks

- Get it right, get it faster
- Tradeoff between elegance and performance
- Different implementations: recursion $\rightarrow$ iteration
- Different algorithms, complexity: reduce number of operations
- Use profiler to detect performance bottlenecks
- Follow performance recommendations to avoid costly operations