CSC148 fall 2013

names, tracing, abstraction recursion

week 12

Danny Heap

heap@cs.toronto.edu

BA4270 (behind elevators)

http://www.cdf.toronto.edu/~heap/148/F13/

416-978-5899

November 28, 2013

Next week:

Mon - review, including some exercise, lab, assignment solutions.

Wed optional overflow.

A2 - remark on MarkUs

!

Computer Science
UNIVERSITY OF TORONTO

Outline

memory model

tracing... or not

consequences of recursion

how much detail for developers?

Enough detail to predict results and efficiency of our code — more detail than end users, less than compiler/interpreter designers. In Python:

- Every **name** contains a **value**
- Every **value** is a reference to the address of an object

searching for names

python looks, in order:

- innermost scope (function body, usually) **local**
- enclosing scopes **nonlocal**
- **global** (current module or `__main__`)
- built-in names
- see scopes and namespaces

intense example

Try running python docs namespace example to check your comfort level

methods

The first parameter, **conventionally** called self, is a reference to the instance:

```
>>> class Foo:
...     def f(self):
...         return "Hi world!"
...
>>> f1 = Foo()
```

Now **Foo.f(f1)** means **f1.f()**

hunting for a method...

Start in the nearest subclass and work upwards, for example visualize method

don't trace too far!

```
def rec_max(L):
    """
    Return the maximum number in possibly nested list of numbers.

    >>> rec_max([17, 21, 0])
    21
    >>> rec_max([17, [21, 24], 0])
    24
    >>> rec_max([17, [21, 24], [18, 37, 16], 0])
    37
    """
    return max([rec_max(x) if isinstance(x, list) else x for x in L])
```

Recommended:

- trace the simplest (non-recursive) case
- trace the next-most complext case, **plug in known results**
- same as previous step...

TMI tracing

In contrast to the step-by-step, plus abstraction (previous slide), you **could** trace this in the visualizer

fibonacci

0 1 1 2 3 5 8

gestation period

$fib(0) = 0$
$fib(1) = 1$
$fib(n) = fib(n-1) + fib(n-2)$ for $n > 1$

immortal

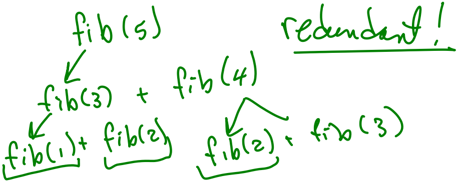
This sequence arises in applied rabbit breeding and depth of balanced BSTs. See vi hart for details.

code writing efficiency

The code is almost a direct translation of the algorithm. But, initially, there is a performance problem:

```
def fibonacci(n: int) -> int:
    """
    nth fibonacci number, where fibonacci(0) is 0,
    fibonacci(1) is 1,
    and fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) if n > 1

    >>> fibonacci(5)
    5
    >>> fibonacci(6)
    8
    """
    return n if n < 2 else fibonacci(n - 1) + fibonacci(n - 2)
```



avoiding redundant calls...

memoization

If fibonacci is called on **exactly** the same input, the result should be the same:

```
def fibonacci_mem(n: int) -> int:
    """memoized fibonacci"""
    cached = {}
    def fib_rec(n: int) -> int:
        if not n in cached:
            if n < 2:
                cached[n] = n
            else:
                cached[n] = fib_rec(n - 1) + fib_rec(n - 2)
        return cached[n]
    return fib_rec(n)
```

has been previously computed?

exactly fib from previous

automatic memoization

Indeed, memoization can be automated:

```
def memoize(f: 'function') -> 'function':
    """Return memoized version of f"""
    table = {}
    def g(x):
        if not x in table:
            table[x] = f(x)
        else:
            pass
        return table[x]
    return g
```

top down

Dynamic Programming

bottom up

memo fibonacci approach

quicksort revisited

EC tail call optimization

The efficiency of our quicksort example depended on the input list not being sorted:

```
import random
L = list(range(1000))
random.shuffle(L)
def quick(L: list) -> list:
    """Produce list with same elements as L in ascending order"""
    return (quick([x for x in L[1:] if x < L[0]]) +
            [L[0]] +
            quick([x for x in L[1:] if x >= L[0]])
            if len(L) > 1 else L)
```

if sorted, this empty

efficiency

[4, 3, 2, 1, 1, 2, 3, 4, 5]

randomize quicksort

You can tinker with sys.setrecursionlimit to overcome python's incomplete support for recursion, or randomize the algorithm:

```
def quick2(L: list) -> list:
    """Produce list with same elements as L in ascending order"""
    if len(L) < 2:
        return L
    else:
        p = random.randint(0, len(L) - 1)
        return (quick2([x for x in L[:p] + L[p+1:] if x < L[p]]) +
                [L[p]] + quick2([x for x in L[:p] + L[p+1:]
                if x >= L[p]]))
```

TA roster

Tuesday: Xin, Orion, Amirali

Thursday: Sam, Aida, Edy, Anton

Friday: Zhaowei, Raymond, Patricia