CSC148 winter 2014

stools, names, tracing

week 5

Danny Heap / Dustin Wehr
heap@cs.toronto.edu / dustin.wehr@utoronto.ca
BA4270 / SF4306D
http://www.cdf.toronto.edu/~heap/148/F13/

February 5, 2014

Outline

tracing... or not

prose to (recursive) code

memory model

name resolution (name lookup)

unit testing example

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 complex 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

getting that recursive insight for Tower of Hanoi

In order to implement a function that moves **n** cheeses from, say, stool 1 to stool 3, we'd first think of a name and parameters. We can start with **movecheeses(n, source, dest)**, meaning show the moves from source stool to destination stool for n cheeses.

getting that recursive insight for Tower of Hanoi

In order to implement a function that moves **n** cheeses from, say, stool 1 to stool 3, we'd first think of a name and parameters. We can start with **movecheeses(n, source, dest)**, meaning show the moves from source stool to destination stool for n cheeses.

getting that recursive insight for Tower of Hanoi

In order to implement a function that moves **n** cheeses from, say, stool 1 to stool 3, we'd first think of a name and parameters. We can start with **movecheeses(n, source, dest)**, meaning show the moves from source stool to destination stool for n cheeses.

stating that recursive insight:

The doodling on the previous slide breaks into a pattern.

- move all but the bottom cheese from source to intermediate stool (sounds recursive...)
- move the bottom cheese from the source to the destination stool (sounds like the 1-cheese move)
- move all but the bottom cheese from the intermediate to the destination stool (sounds recursive...)

The original problem repeats, except with different source, destination, and intermediate stools!

New name: **move_cheeses(n, source, intermediate, destination)**

write some code!

Fill in the three steps from the previous slide, using recursive calls to **movecheeses(...)** with different values for the number of cheeses, the source, destination, and intermediate stools, where appropriate.

```
def move_cheeses(n: int, source: int, intermediate: int,
                 destination: int) -> None:
    """Print moves to get n cheeses from source
       to destination, possibly using intermediate"""
    if n > 1: # fill this in!
        move_cheeses(      ?,      ?,      ?,      ?)
        move_cheeses(      ?,      ?,      ?,      ?)
        move_cheeses(      ?,      ?,      ?,      ?)
    else: # just 1 cheese --- leave this out for now!
```

complete that code!

Now, fill in what you do to move just one cheese — don't use any recursion! You will be returning a string that specifies you are moving from source to destination.

```
def move_cheeses(n: int, source: int, intermediate: int,
                 destination: int) -> None:
    """Print moves to get n cheeses from source
       to destination, possibly using intermediate"""
    if n > 1:
        move_cheeses(n - 1, source, destination, intermediate)
        move_cheeses(1, source, intermediate, destination)
        move_cheeses(n - 1, intermediate, source, destination)
    else: # just 1 cheese --- fill this in now!
        print( ???? )
```

complete that code!

Now, fill in what you do to move just one cheese — don't use any recursion! You will be returning a string that specifies you are moving from source to destination.

```
def move_cheeses(n: int, source: int, intermediate: int,
                 destination: int) -> None:
    """Print moves to get n cheeses from source
       to destination, possibly using intermediate"""
    if n > 1:
        move_cheeses(n - 1, source, destination, intermediate)
        move_cheeses(1, source, intermediate, destination)
        move_cheeses(n - 1, intermediate, source, destination)
    else:
        print(source, "-->", destination)
```

python gratification

Once you have your code entered into some Python environment, you should run it with a few small values of **n**. As usual, you can get more intuition about it by tracing examples, working from small to larger **n**

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** `x` contains a **value** `id(x)`
- Every **value** is a reference to the address of an object

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** `x` contains a **value** `id(x)`
- Every **value** is a reference to the address of an object
- Actually, the python docs consider the "address" part an implementation detail, *not* relevant for developers. Docs for `id()`:
  Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

searching for names

python looks, in order:

- innermost scope (function body usually) **local**
  (can also be a list/dictionary comprehension, or a lambda expression)
- scopes of enclosing functions **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

This might seem like a very python-specific thing, but in fact every programming language has some standard for name lookup, and they are all fairly similar.

methods

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

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

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

hunting for a method...

Start in the object's nearest subclass and work upwards (through the inheritance hierarchy), for example visualize method

write unit tests with good coverage for this function

```
def max_nested(L:list) -> 'int or None':
  """ REQ : L is a (possibly-nested) list of integers
  Returns the largest integer in L, or None if L has no
  integers in it.
  """
  if len(L) == 0:
    return None
  maxes_of_parts = []
  for x in L:
    if isinstance(x,int):
      maxes_of_parts.append(x)
    else:
      y = max_nested(x)
      if y is not None:
        maxes_of_parts.append(y)
  return max(maxes_of_parts)
```

review: choosing test cases

write unit tests with good coverage for this function

```
def max_nested(L:list) -> 'int or None':
  """ REQ : L is a (possibly-nested) list of integers
  Returns the largest integer in L, or None if L has no
  integers in it.
  """
  if len(L) == 0:
    return None
  maxes_of_parts = []
  for x in L:
    if isinstance(x,int):
      maxes_of_parts.append(x)
    else:
      y = max_nested(x)
      if y is not None:
        maxes_of_parts.append(y)
  return max(maxes_of_parts)
```

Did your unit tests find an error in the code?