RECURSION

Amir H. Chinaei

May 8, 2017, York University

ASSUMPTIONS

- familiarity with
  - simple Python programs
    - including function calls, and
    - list comprehensions
  - tracing programs
  - activation record (a.k.a. stack frame)
    - and its application in tracing function calls
  - PyCharm IDE

objectives

- introduce recursive functions/programs
- describe how recursive programs work
- compare recursion vs iteration

before we dive into it

- **recursion** occurs *when a thing is defined in terms of itself* (wikipedia)

- **recur** means *to come up again* (merriam-webster)

recursive examples

- **factorial** function

recursive case

$$\begin{cases} \textbf{factorial}(n) = n * \textbf{factorial}(n\textbf{-1}) \\ \textbf{factorial(0)} = 1 \end{cases}$$

base case

- **fibonacci** function

recursive cases

$$\begin{cases} \textbf{fibonacci}(n) = \textbf{fibonacci}(n\textbf{-1}) + \textbf{fibonacci}(n\textbf{-2}) \\ \textbf{fibonacci(1)} = 1 \\ \textbf{fibonacci(0)} = 1 \end{cases}$$

base cases

A recursive function has
at least one **base case** and at least one **recursive case**

another example

a recursive definition: **balanced_string**

- recursive cases:
  - (x) is balanced if x is a **balanced_string**
  - xy is balanced if x and y are **balanced_string**
- base case:
  - a string containing no parentheses is balanced

how about these functions?

- $f(n) = n^2 + n - 1$
- $f(n) = g(n-1) + 1, \quad g(n) = n/2$
- $f(n) = 5, f(n-1) = 4$
- $f(n) = n * (n-1) * (n-2) * \ldots * 2 * 1$
- $f(n) = f(\lfloor n/2 \rfloor) + 1, f(1) = 1$
- $f(n) = f(n-1)$

recursive programs

- solution defined in terms of solutions for smaller problems

```
def solve (n):
    . . .
    value = solve(n-1) + solve(n/2)
    . . .
```

- one or more base cases

```
    if n < 10:
        value = 1
```

- some base case is always reached eventually; otherwise, it’s an infinite recursion

general form of recursion

if (condition to detect a base case):

{do something without recursion}

else: (general case)

{do something that involves recursive call(s)}

★★★

devise it such that some base case is always reached eventually

recursive programs example 1

0! = 1 , n! = n*(n-1)!

```
def factorial(n):
    # pre: n ≥ 0
    # post: returns n!
    if n==0:
        return 1
    else:
        return n * factorial(n-1)
```

- structure of code typically parallels structure of definition

recursive programs example 2

fib(0) = 1, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)

```
def fib(n):
    # pre: n ≥ 0
    # post: returns the nth Fibonacci number
    if n<2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- structure of code typically parallels structure of definition

max_list() example 3

tracing factorial: intuitively

trace `fact(3)`

tracing max_list()

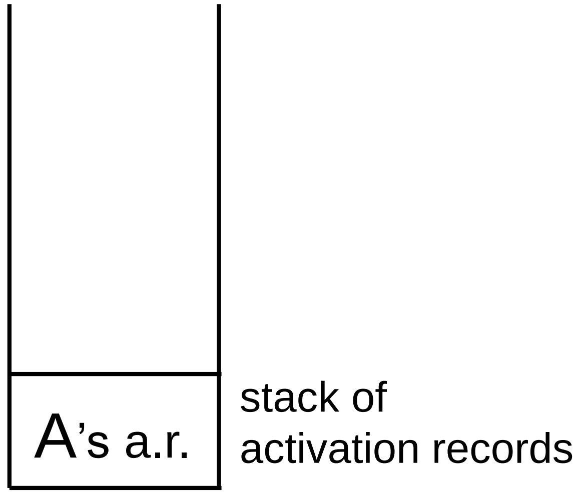
trace `max_list([4, 2, [[4, 7],5], 8])`

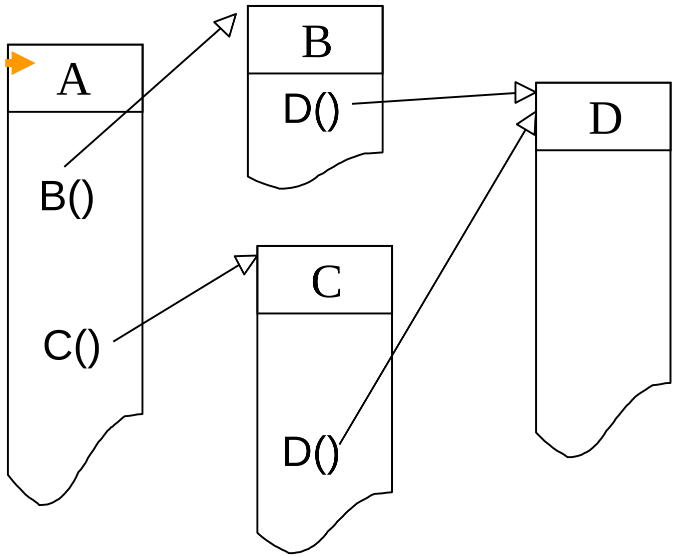
stacks and tracing calls review

- recall:
  - stack applications in tracing function calls
- activation record (a.k.a stack frame)
  - all information necessary for tracing a function call
  - such as *parameters*, *local variables*, *return address*, etc.
- when function called:
  - activation record is created, initialized, and pushed onto the stack
- when function finishes:
  - its activation record (that is on top of the stack) is popped from the stack

tracing calls review

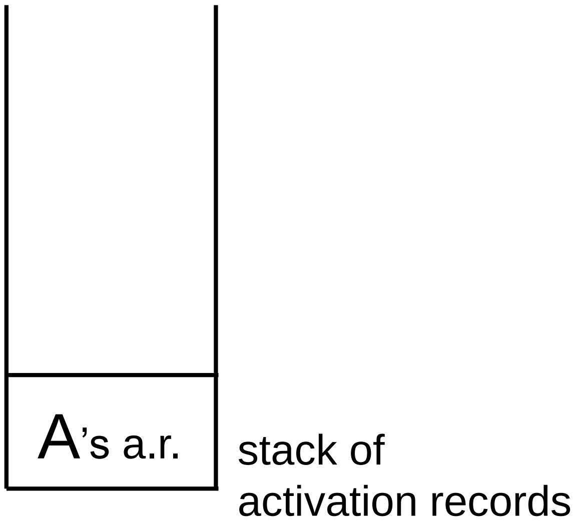
- recall: stack of activation records
  - when function called:
    - activation record created, initialized, and pushed onto the stack
  - when function finishes,
    - its activation record is popped

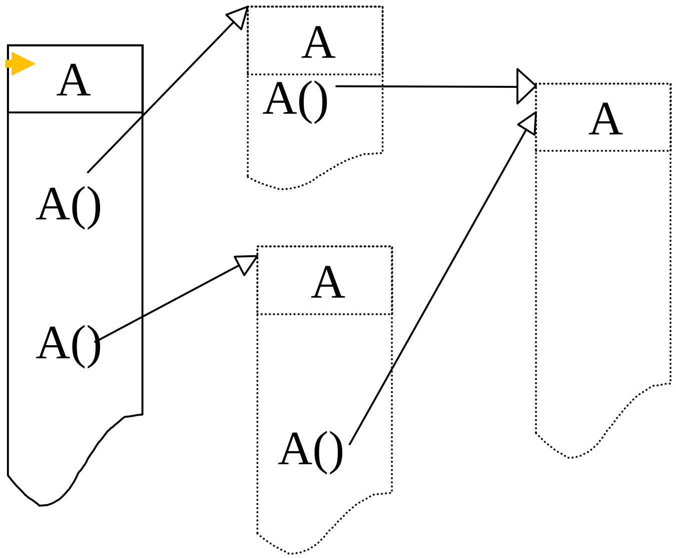




tracing recursive calls

- same mechanism for recursive programs





tracing factorial

```
1.  def f(n):
2.      # pre: n≥0
3.      # post: returns n!
4.      if (n==0): return 1
5.      else: return n * f(n-1)
```

```
1. def main():
       . . .
8.     print(f(3))
       . . .
```

6

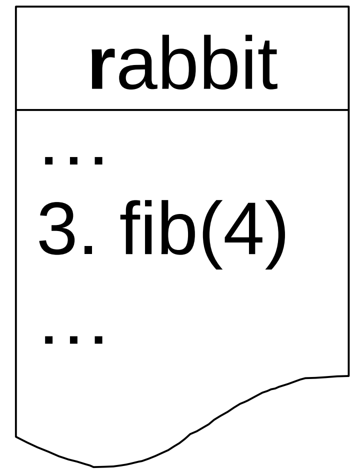
5,f,0	Return 1
5,f,1	Return 1
5,f,2	Return 2
8,m,3	Return 6

line#	func.	n

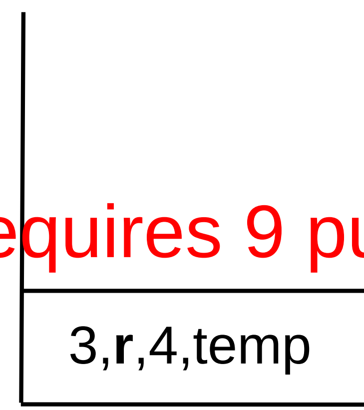
activation record

tracing fibonacci

```
1. def fib(n):
2.    # pre: n ≥ 0
3.    # post: returns the
4.    # nth Fibonacci number
4.    if (n < 2): return 1
5.    else: return fib(n-1) +
6.                 fib(n-2)
```



hint: requires 9 pushes



stack of
activation records

line#	func.	n	temp

activation record

why 9?

- using rewriting (aka call tree, intuitively)

fib(4)

fib(3)+ fib(2)

fib(2)+fib(1)   fib(1)+fib(0)

fib(1)+fib(0)

recursive vs iterative

- recursive functions impose a repeated task (i.e. loop)
- the loop is implicit and Python takes care of it
- this comes at a price: time & memory
- the price may be negligible in many cases

- after all, no recursive function is more efficient than its iterative equivalent

recursive vs iterative cont'ed

- every recursive function can be written iteratively (by explicit loops)
  - may require stacks too
- yet, when the nature of a problem is recursive, writing it iteratively can be
  - time consuming, and
  - less readable
- so, recursion is a very powerful technique for problems that are naturally recursive

more examples

- balanced strings
- more functions on nested lists
- merge sort
- quick sort
- traversing trees

- in general,
  - properties of recursive definitions/structures

summary

- recursion is

  fun(fun(fun(…fun(this is the base)…)))
- recursion is powerful

- questions on today’s lecture?

- don’t forget to
  - trace `fib(4)`
  - trace `max_list([4,[[4,7],9],8])`
  - implement this lecture’s examples
- prior to start working on this week’s lab