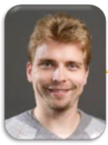


UC Berkeley EECS
Adj. Ass. Prof.
Dr. Gerald Friedland

Computational Structures in Data Science

Lecture 6: Environment Diagrams, Recursion Review, Midterm Review

Computational Concepts Toolbox

- Data type: values, literals, operations,
  - e.g., int, float, string
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
  - indexing
- Data structures
- Tuple assignment
- Call Expressions
- Function Definition Statement
- Conditional Statement
- Iteration:
  - data-driven (list comprehension)
  - control-driven (for statement)
  - while statement
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Recursion
- Environment Diagrams

Recursion Key concepts – by example

1. Test for simple "base" case

2. Solution in simple "base" case

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```

3. Assume recusive solution to simpler problem

4. Transform soln of simpler problem into full soln

In words

- The sum of no numbers is zero
- The sum of 1^2 through n^2 is the
  - sum of 1^2 through $(n-1)^2$
  - plus n^2

```
def sum_of_squares(n):
    if n < 1:
        return 0
    else:
        return sum_of_squares(n-1) + n**2
```


How does it work?

- Each recursive call gets its own local variables
  - Just like any other function call
- Computes its result (possibly using additional calls)
  - Just like any other function call
- Returns its result and returns control to its caller
  - Just like any other function call
- The function that is called happens to be itself
  - Called on a simpler problem
  - Eventually bottoms out on the simple base case
- Reason about correctness "by induction"
  - Solve a base case
  - Assuming a solution to a smaller problem, extend it

Local variables

```
def sum_of_squares(n):
    n_squared = n**2
    if n < 1:
        return 0
    else:
        return n_squared + sum_of_squares(n-1)
```

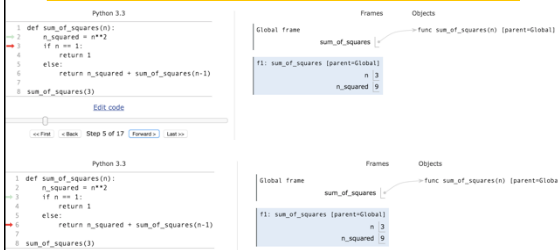
- Each call has its own "frame" of local variables
- What about globals?
- Let's see the environment diagrams

https://goo.gl/CiFaUJ

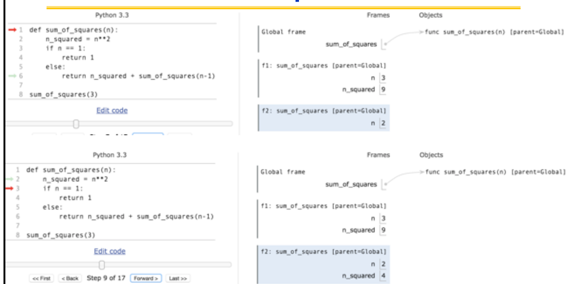
Environments Example

pythontutor.com

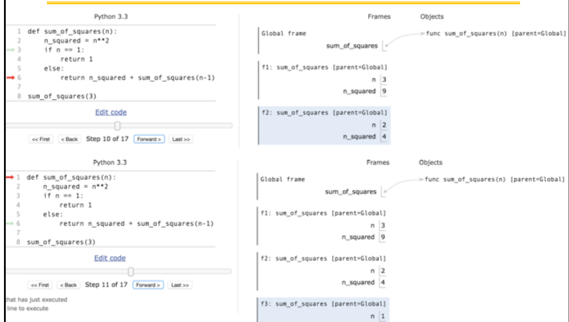
Environments Example



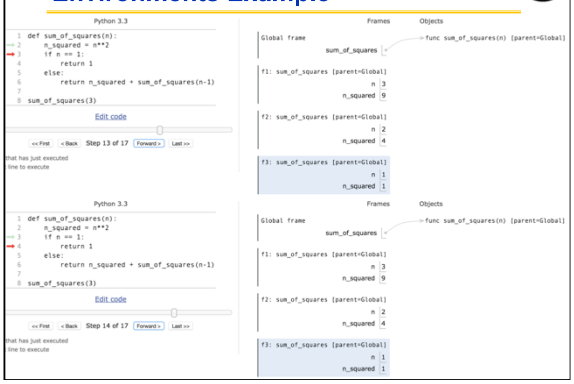
Environments Example



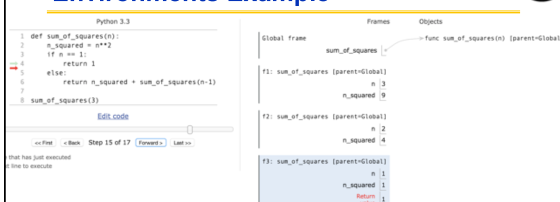
Environments Example



Environments Example



Environments Example



Environments Example

Environments Example

How much ???

- Time is required to compute **`sum_of_squares(n)`**?
  - Recursively?
  - Iteratively ?
- Space is required to compute **`sum_of_squares(n)`**?
  - Recursively?
  - Iteratively ?

Linear proportional to n *cn* for some c

- Count the frames…
- Recursive is linear, iterative is constant!

Tail Recursion

- All the work happens on the way down the recursion
- On the way back up, just return

```
def sum_up_squares(i, n, accum):
    """Sum the squares from i to n in incr. order"""
    if i > n:
        Base Case
    else:
        Tail Recursive Case

>>> sum_up_squares(1,3,0)
14
```


Tree Recursion

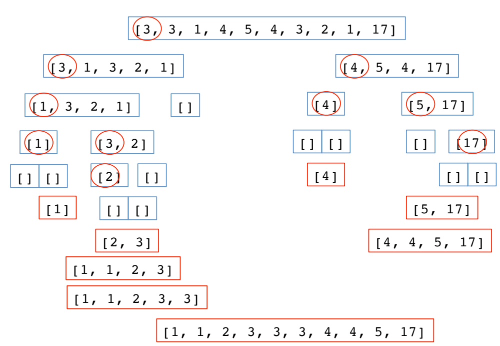
- Break the problem into multiple smaller sub-problems, and Solve them recursively

```
def split(x, s):
    return [i for i in s if i <= x], [i for i in s if i > x]

def qsort(s):
    """Sort a sequence - split it by the first element,
    sort both parts and put them back together."""
    if not s:
        return []
    else:
        pivot = first(s)
        lessor, more = split(pivot, rest(s))
        return qsort(lessor) + [pivot] + qsort(more)

>>> qsort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
```


QuickSort Example



Tree Recursion with HOF

```
def qsort(s):
    """Sort a sequence - split it by the first element,
    sort both parts and put them back together."""

    if not s:
        return []
    else:
        pivot = first(s)
        lessor, more = split_fun(leq_maker(pivot), rest(s))
        return qsort(lessor) + [pivot] + qsort(more)

>>> qsort([3,3,1,4,5,4,3,2,1,17])
[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]
```


On Computer Science Exams

In computer science exams, we try to assess the student's understanding of concepts and his or her ability to practically apply these.

- In CS, we do not:
  - require extensive memorization (e.g. we allow cheat sheet)
  - require a lot of reading
  - require essay writing skills

In CS, we do:
- require the ability to translate a given textual problem into programming code
- require you to be able to read other people's code
- value solutions that are almost right over no solution
- accept solutions we did not think about if they work
- prioritize math (logic) and science (experiment) over opinion or authority

How to prepare for a CS exam

- Explain the content of the computational concepts toolbox to somebody else
  - Describe the concept
  - What is an example of using it?
  - When does it not work? Corner cases?
  - Why does it exist?
- Practice programming:
  - Play around with the examples from lecture, lab, homework
  - Think about your own similar examples
- In the exam:
  - Make sure you understand the question: What is the given input? What is the required output?
  - Think of easy cases first (e.g. n=1?).
  - What is the iteration/recursion doing (e.g. i=i+1)?
  - What are corner cases that need explicit handling (e.g. division by zero, negative numbers, empty list)?

Computational Concepts Toolbox

- **Data type: values, literals, operations,**
  - **e.g., int, float, string**
- **Expressions, Call expression**
- **Variables**
- **Assignment Statement**
- **Sequences: tuple, list**
  - **indexing**
- **Data structures**
- **Tuple assignment**
- **Call Expressions**
- **Function Definition Statement**
- **Conditional Statement**
- **Iteration:**
  - **data-driven (list comprehension)**
  - **control-driven (for statement)**
  - **while statement**
- **Higher Order Functions**
  - **Functions as Values**
  - **Functions with functions as argument**
  - **Assignment of function values**
- **Recursion**
- **Environment Diagrams**

Answers for the Wandering Mind

The computer choses a random element *x* of the list generated by `range(0,n)`. What is the smallest amount of iteration/recursion steps the best algorithm needs to guess *x*?

$\log_2 n$

How would the algorithm look like?

Guess the binary digits of x starting with the highest significant digit. This is, ask questions of the form
"smaller than 2^{n-1}?" (yes => 0…),
"smaller than 2^{n-2}?" (no => 0 1…),
"smaller than $2^{n-2}+2^{n-3}$?", …

This method is also called: binary search

Quantum physics: Allow less than $\log_2 n$ guesses.