Recursion

David E. Culler

CS8 – Computational Structures in Data Science

http://inst.eecs.berkeley.edu/~cs88

Lecture 5

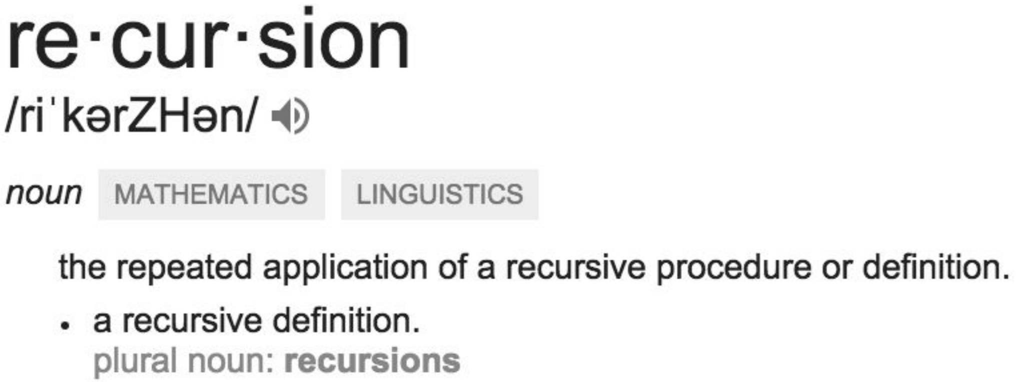
Sept 24, 2018

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
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions

Today: Recursion



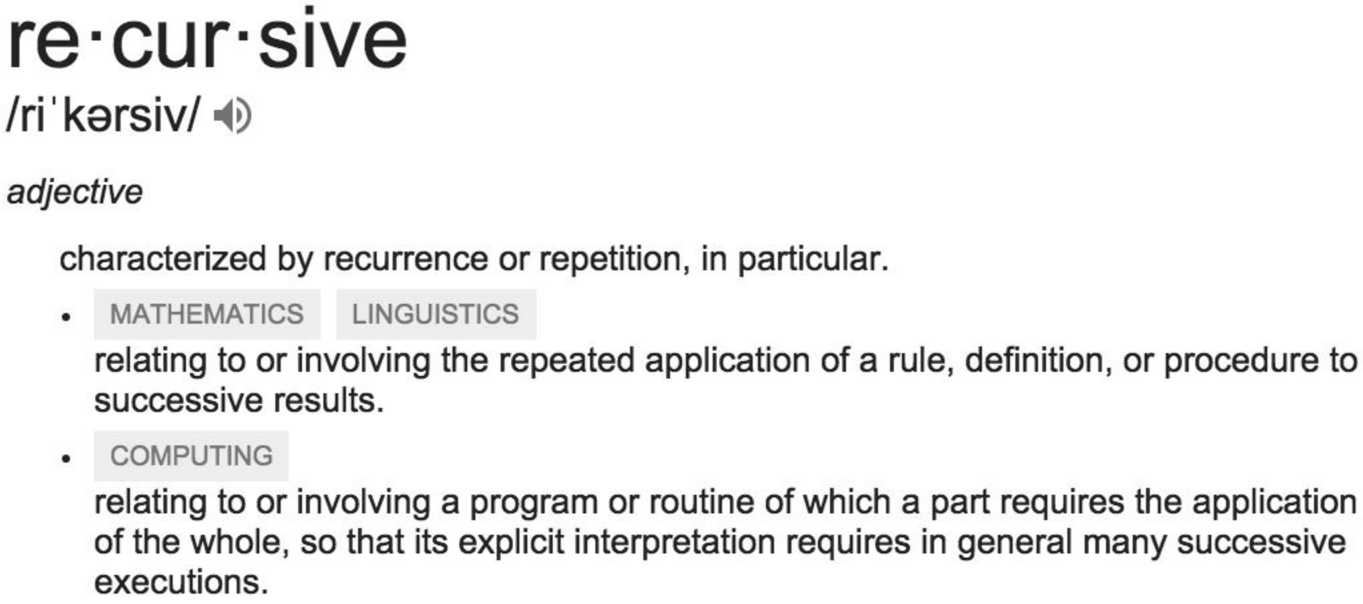
re·cur·sion

/riˈkərZHən/ 🔊

noun MATHEMATICS LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.
  plural noun: **recursions**



re·cur·sive

/riˈkərsiv/ 🔊

adjective

characterized by recurrence or repetition, in particular.

- MATHEMATICS LINGUISTICS
  relating to or involving the repeated application of a rule, definition, or procedure to successive results.
- COMPUTING
  relating to or involving a program or routine of which a part requires the application of the whole, so that its explicit interpretation requires in general many successive executions.

- **Recursive function calls itself, directly or indirectly**

Administrative Issues

- Midterm exam: wed Oct 3 6-8 pm
  - Room based on last digit of SID
  - 0-5 LeConte 1   (60%)
  - 6-9: VLSB 2040
  - Alternative and accommodations during 5-9 by request
- Labs are to help you learn the materials, so please make full use of them
- Materials will go through 10/1 Lecture
- Office hours start here after class and migrate down to BIDS in 190 Doe Library

Review: Higher Order Functions

- Functions that operate on functions
- A function

```
def odd(x):
    return x%2

>>> odd(3)
1
```

- A function that takes a function arg

```
def filter(fun, s):
    return [x for x in s if fun(x)]

>>> filter(odd, [0,1,2,3,4,5,6,7])
[1, 3, 5, 7]
```

Why is this not ‘odd’ ?

Review Higher Order Functions (cont)

- **A function that returns (makes) a function**

```
def leq_maker(c):
    def leq(val):
        return val <= c
    return leq
```

```
>>> leq_maker(3)
<function leq_maker.<locals>.leq at 0x1019d8c80>

>>> leq_maker(3)(4)
False

>>> filter(leq_maker(3), [0,1,2,3,4,5,6,7])
[0, 1, 2, 3]
>>>
```

One more example

- **What does this function do?**

```
def split_fun(p, s):
    ””” Returns <you fill this in>."""
    return [i for i in s if p(i)], [i for i in s if not p(i)]
```

```
>>> split_fun(leq_maker(3), [0,1,2,3,4,5,6])
([0, 1, 2, 3], [4, 5, 6])
```

Recall: Iteration

1. Initialize the “base” case of no iterations

2. Starting value

3. Ending value

```
def sum_of_squares(n):
    accum = 0
    for i in range(1,n+1):
        accum = accum + i*i
    return accum
```

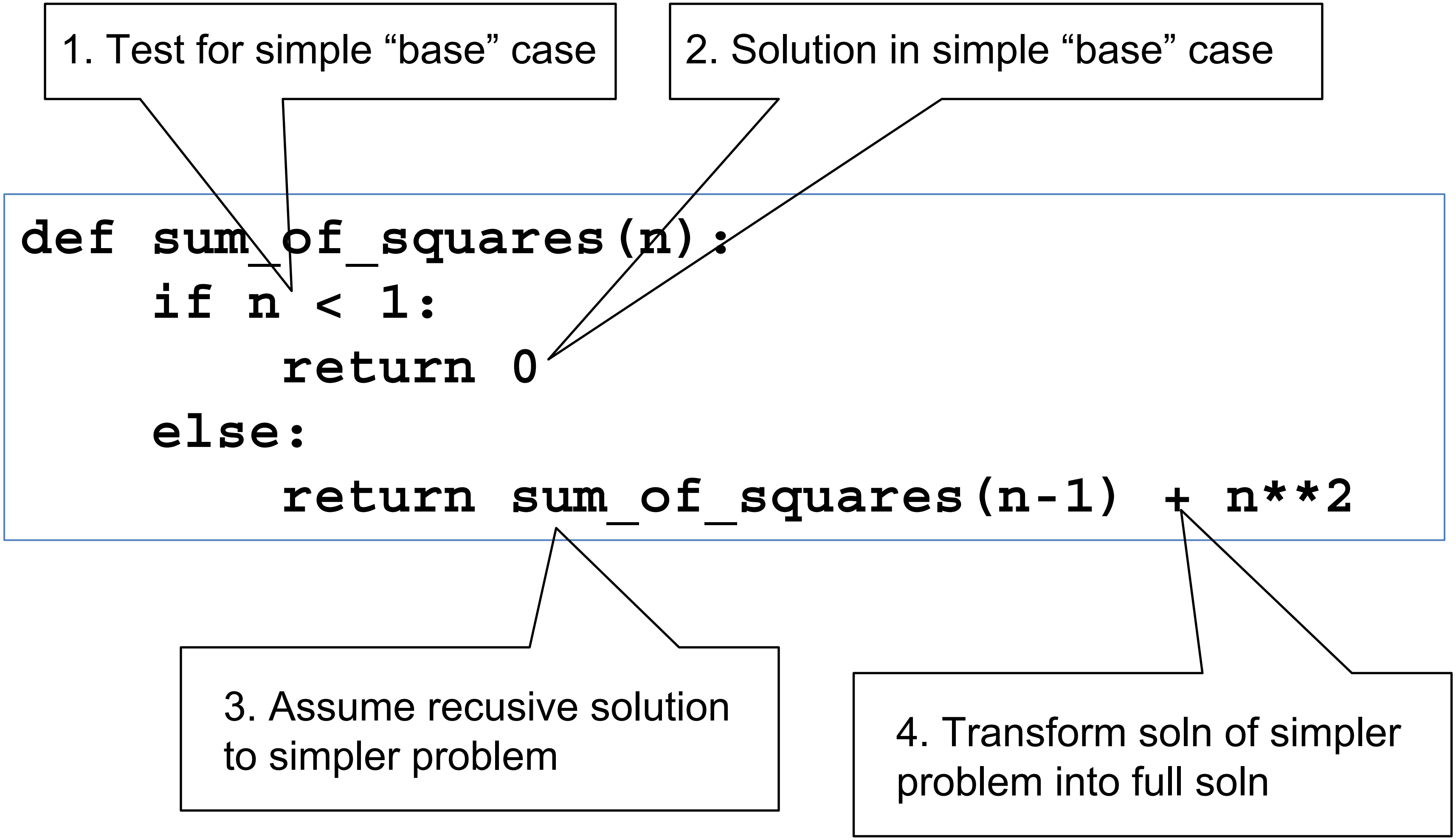
4. New loop variable value

- **Loops are a simple form of recursion – linear recursion**

Remember

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

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
```

Why does it work

```
sum_of_squares(3)

# sum_of_squares(3) => sum_of_squares(2) + 3**2
#                => sum_of_squares(1) + 2**2 + 3**2
#                => sum_of_squares(0) + 1**2 + 2**2 + 3**2
#                => 0 + 1**2 + 2**2 + 3**2 = 14
```

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

Questions

- In what order do we sum the squares ?
- How does this compare to iterative approach ?

```
def sum_of_squares(n):
    accum = 0
    for i in range(1,n+1):
        accum = accum + i*i
    return accum
```

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

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

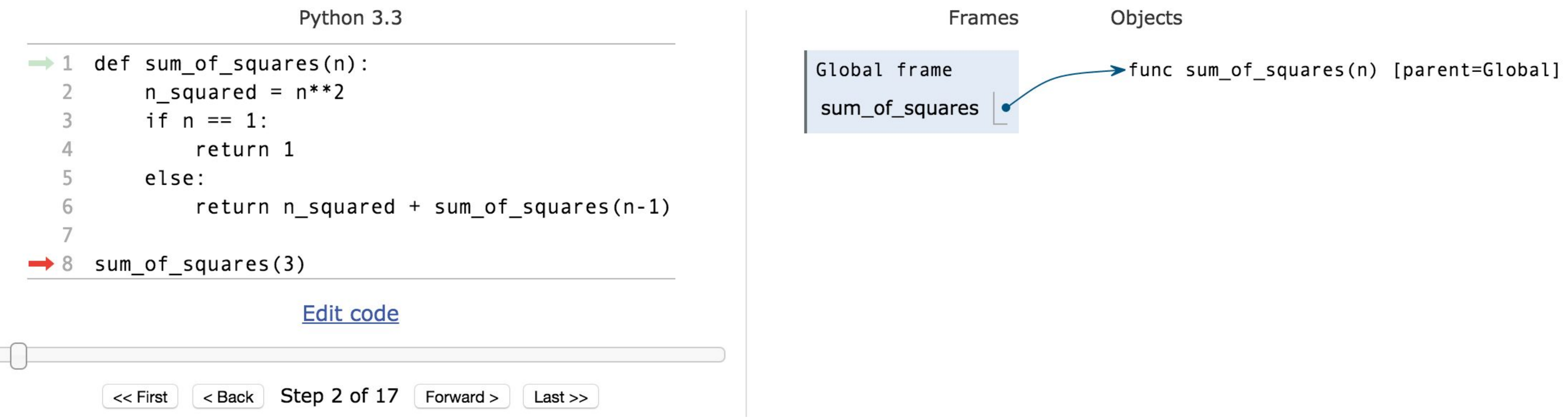
Local variables

```
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



Python 3.3

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

sum_of_squares(3)
```

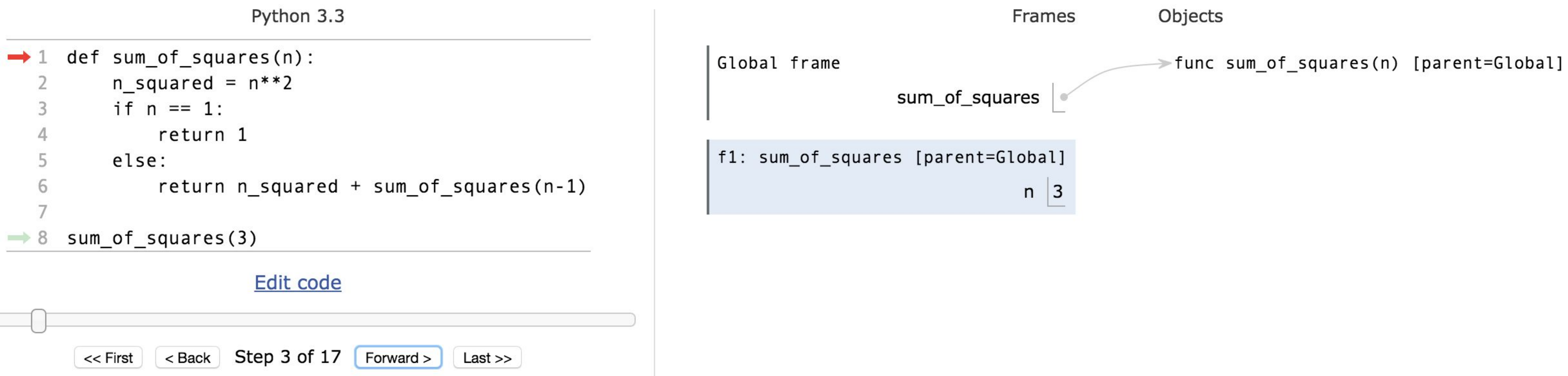
Frames

Global frame

sum_of_squares

Objects

func sum_of_squares(n) [parent=Global]



Python 3.3

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

sum_of_squares(3)
```

Frames

Global frame

sum_of_squares

f1: sum_of_squares [parent=Global]

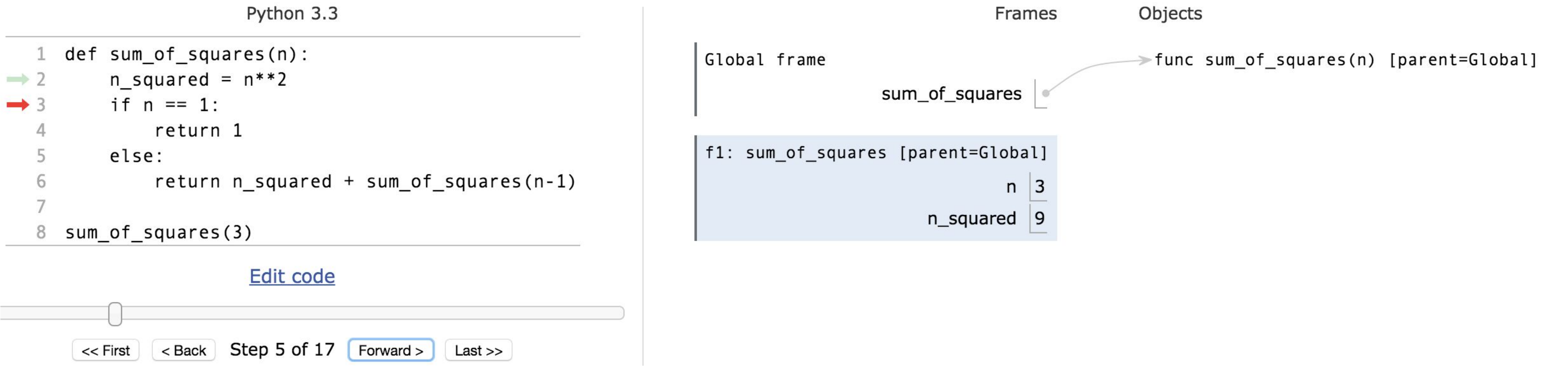
n 3

Objects

func sum_of_squares(n) [parent=Global]

pythontutor.com

Environments Example



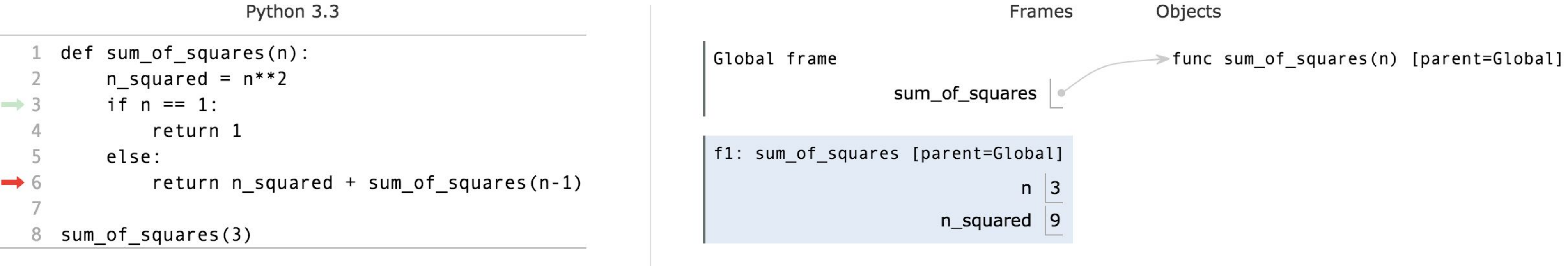
```
def sum_of_squares(n):
    n_squared = n**2
    if n == 1:
        return 1
    else:
        return n_squared + sum_of_squares(n-1)

sum_of_squares(3)
```

Step 5 of 17

```
Global frame
          sum_of_squares  ->  func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
                       n | 3
               n_squared | 9
```



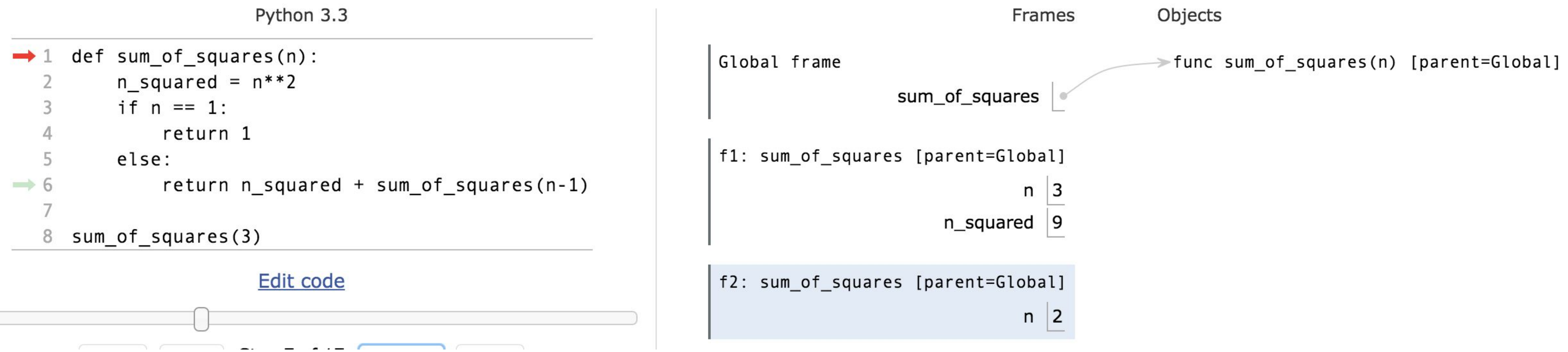
```
def sum_of_squares(n):
    n_squared = n**2
    if n == 1:
        return 1
    else:
        return n_squared + sum_of_squares(n-1)

sum_of_squares(3)
```

```
Global frame
          sum_of_squares  ->  func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
                       n | 3
               n_squared | 9
```

Environments Example



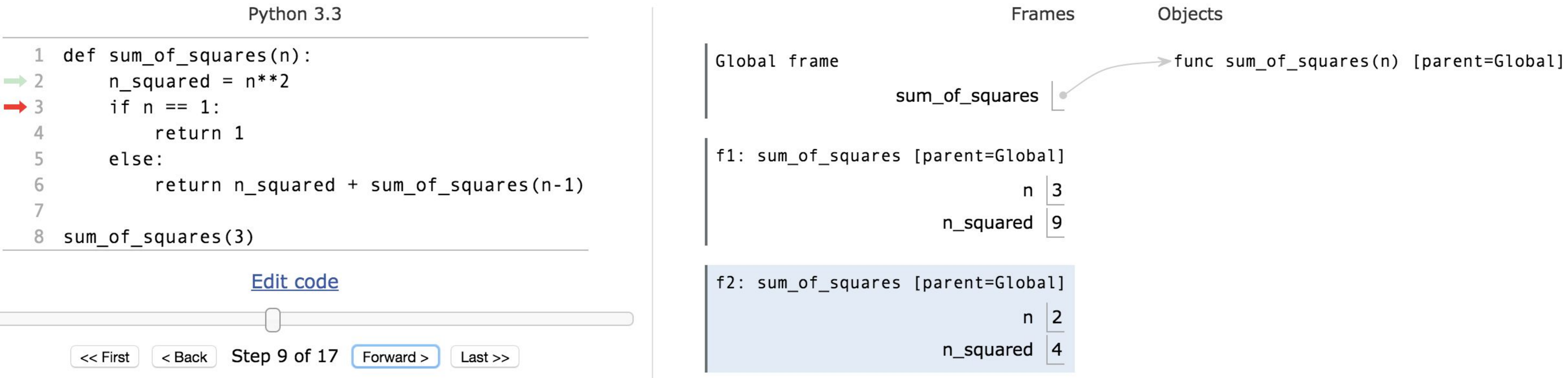
```
def sum_of_squares(n):
    n_squared = n**2
    if n == 1:
        return 1
    else:
        return n_squared + sum_of_squares(n-1)

sum_of_squares(3)
```

Global frame
sum_of_squares → func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
n 3
n_squared 9

f2: sum_of_squares [parent=Global]
n 2



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

sum_of_squares(3)
```

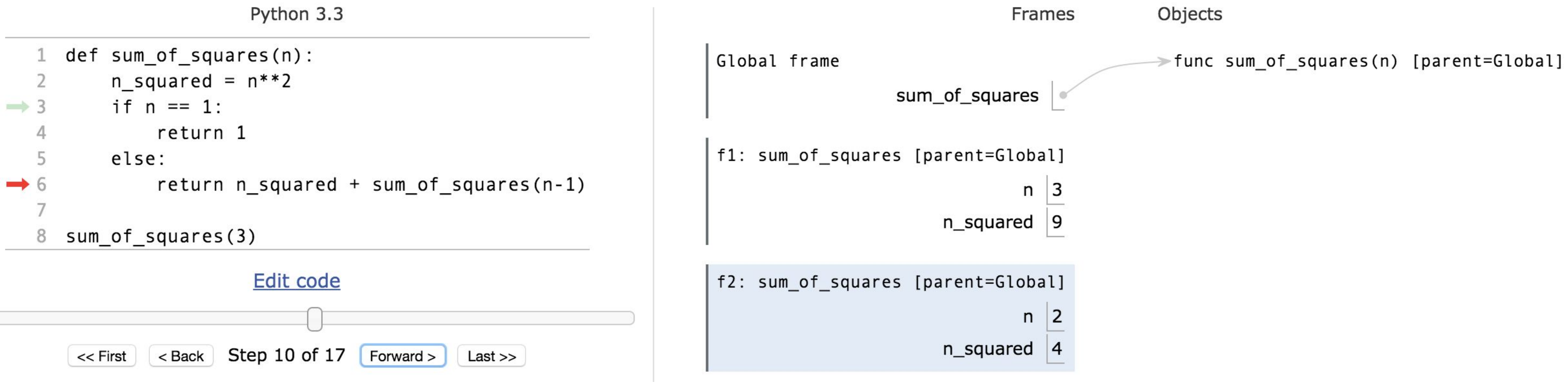
Step 9 of 17

Global frame
sum_of_squares → func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
n 3
n_squared 9

f2: sum_of_squares [parent=Global]
n 2
n_squared 4

Environments Example



Python 3.3

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

sum_of_squares(3)
```

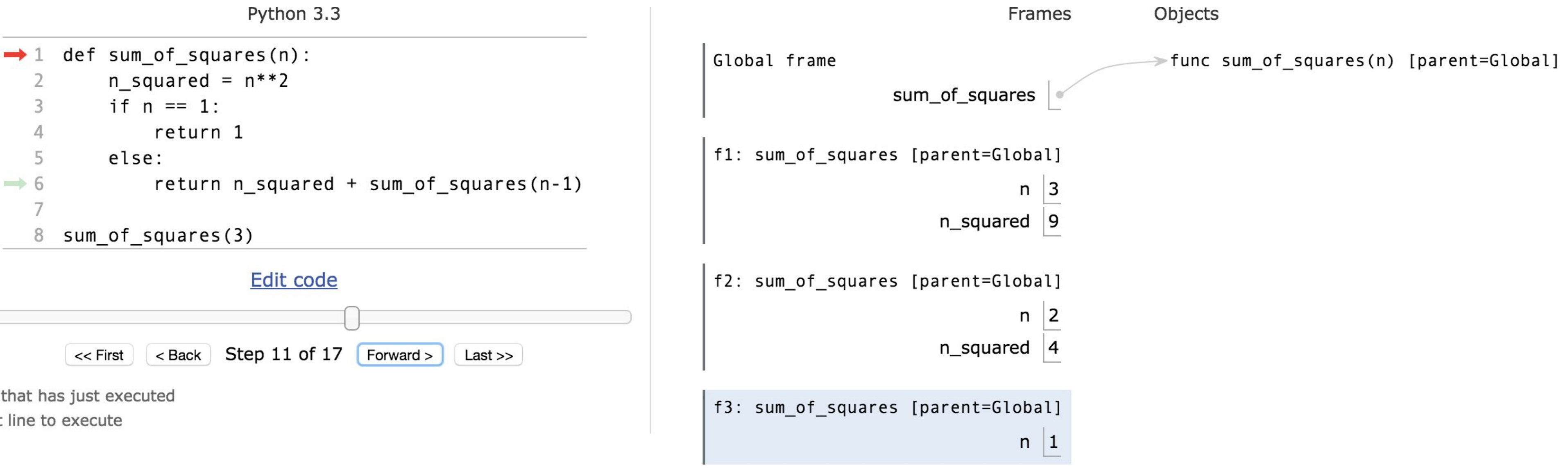
Global frame
sum_of_squares → func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]

n	3
n_squared	9

f2: sum_of_squares [parent=Global]

n	2
n_squared	4



Python 3.3

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

sum_of_squares(3)
```

Global frame
sum_of_squares → func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]

n	3
n_squared	9

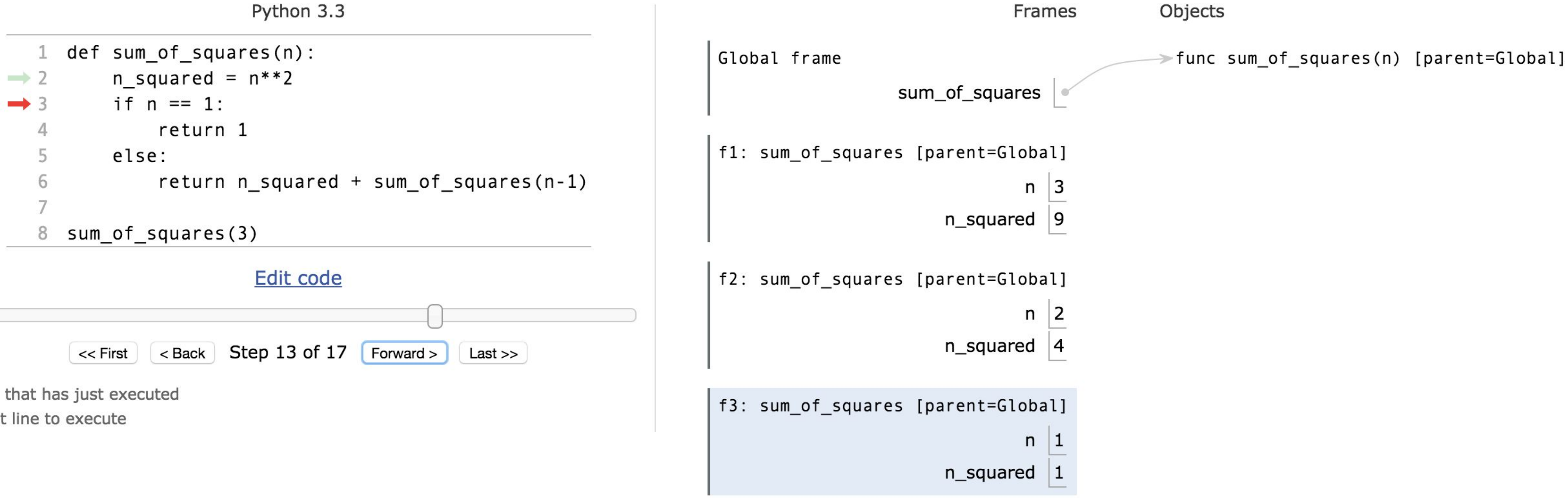
f2: sum_of_squares [parent=Global]

n	2
n_squared	4

f3: sum_of_squares [parent=Global]

n	1

Environments Example



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

sum_of_squares(3)
```

Step 13 of 17

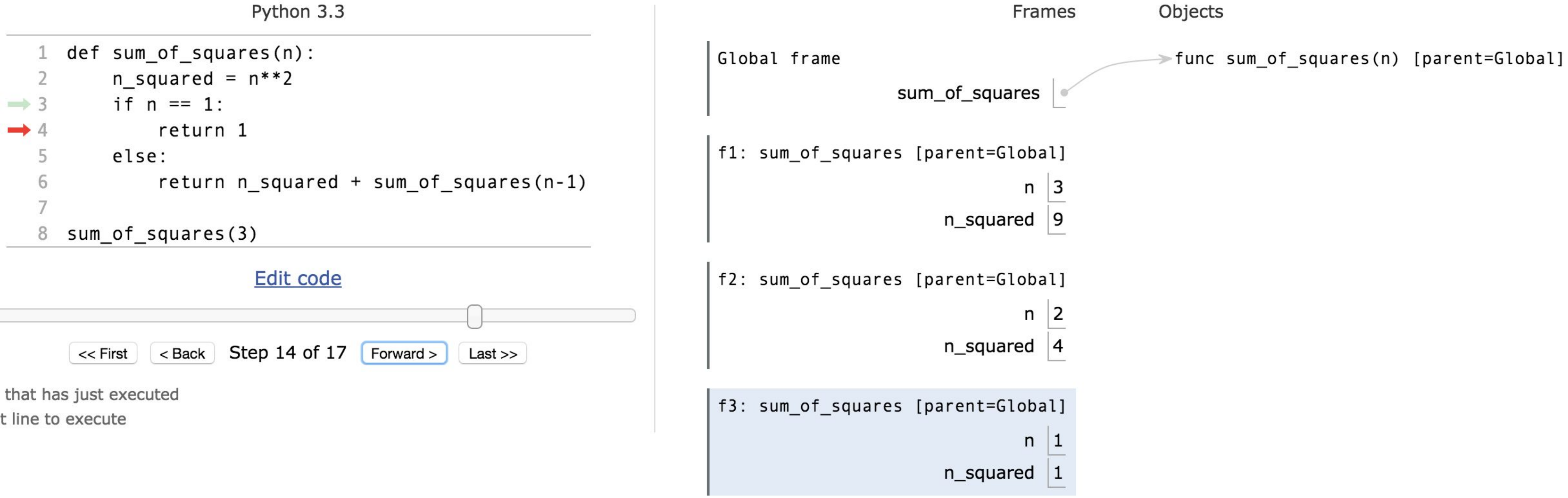
that has just executed
t line to execute

```
Global frame
      sum_of_squares            func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
                 n  3
         n_squared  9

f2: sum_of_squares [parent=Global]
                 n  2
         n_squared  4

f3: sum_of_squares [parent=Global]
                 n  1
         n_squared  1
```



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

sum_of_squares(3)
```

Step 14 of 17

that has just executed
t line to execute

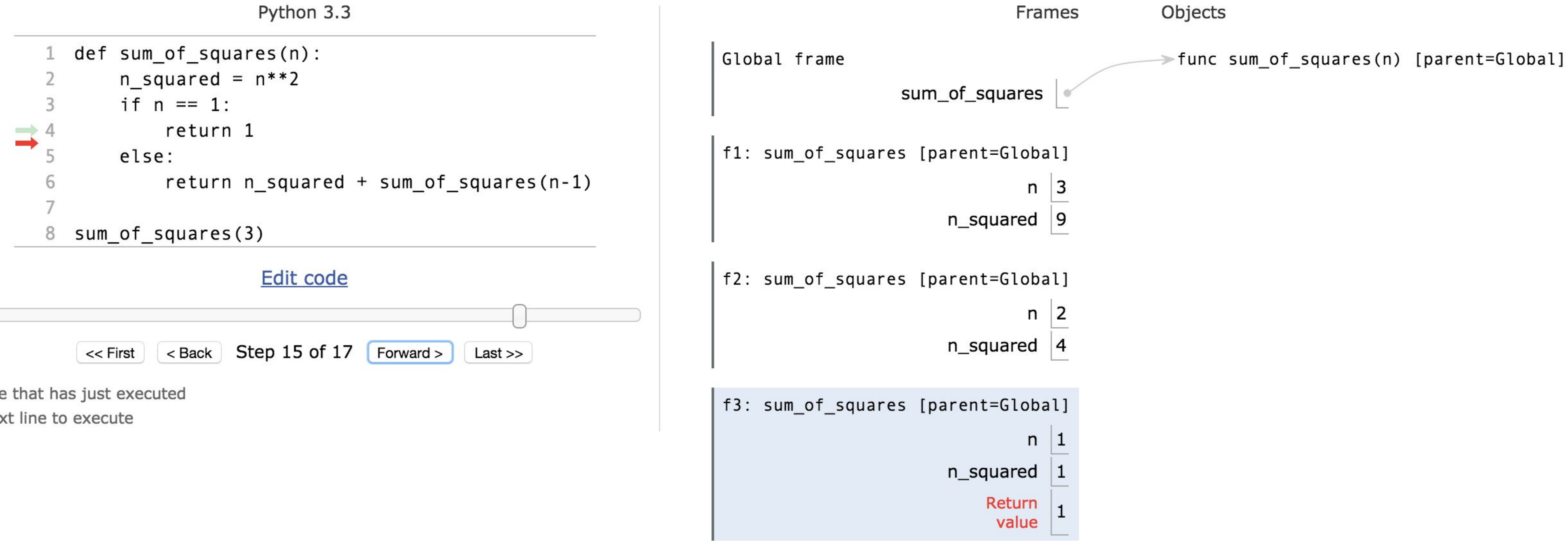
```
Global frame
      sum_of_squares            func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
                 n  3
         n_squared  9

f2: sum_of_squares [parent=Global]
                 n  2
         n_squared  4

f3: sum_of_squares [parent=Global]
                 n  1
         n_squared  1
```

Environments Example



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

sum_of_squares(3)
```

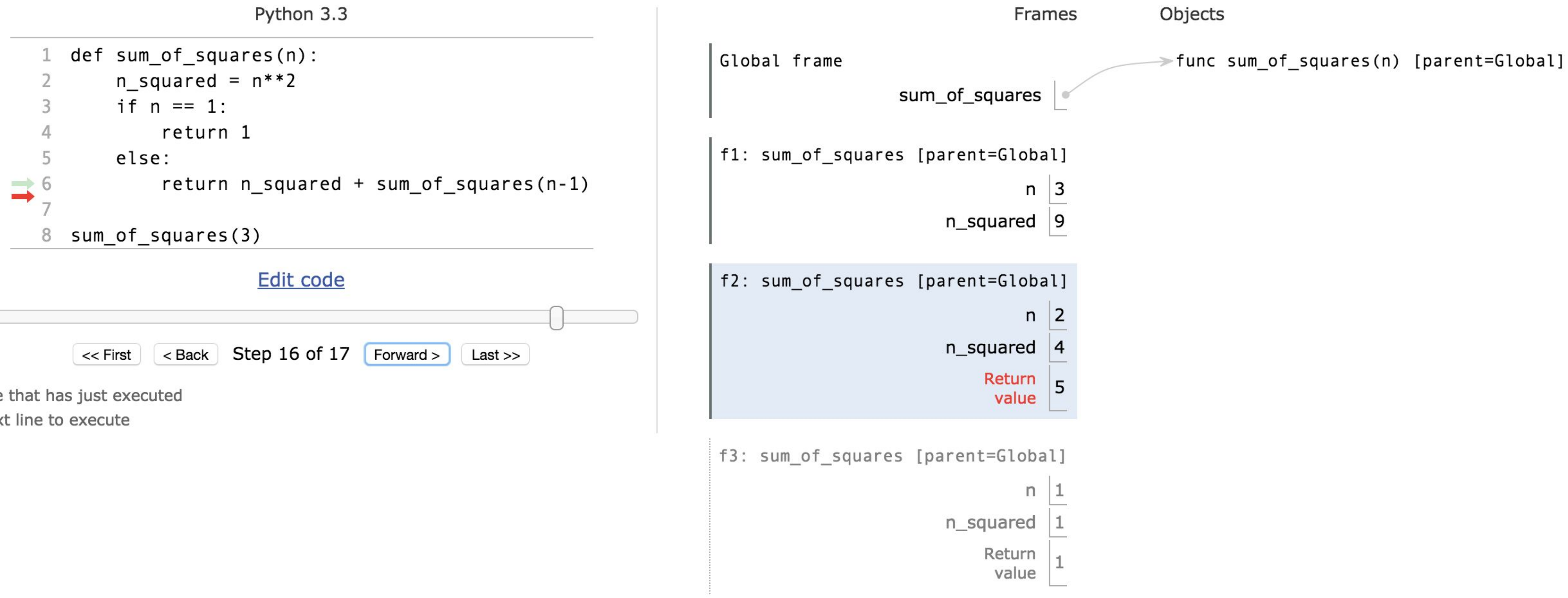
```
Global frame
        sum_of_squares   →  func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
        n          3
        n_squared  9

f2: sum_of_squares [parent=Global]
        n          2
        n_squared  4

f3: sum_of_squares [parent=Global]
        n          1
        n_squared  1
        Return
        value      1
```

Environments Example



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

sum_of_squares(3)
```

e that has just executed

xt line to execute

Global frame

sum_of_squares → func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]

n	3
n_squared	9

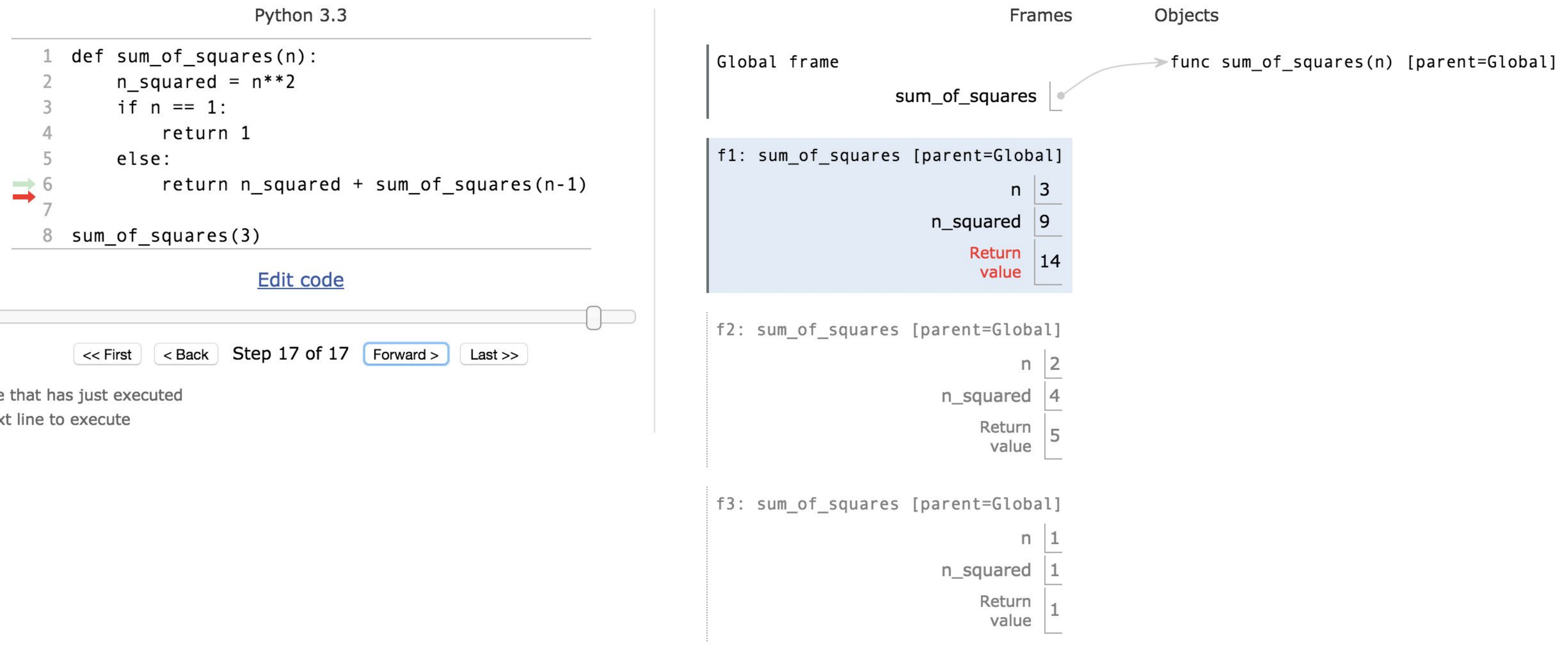
f2: sum_of_squares [parent=Global]

n	2
n_squared	4
Return value	5

f3: sum_of_squares [parent=Global]

n	1
n_squared	1
Return value	1

Environments Example



Python 3.3

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

sum_of_squares(3)
```

```
Global frame
        sum_of_squares          func sum_of_squares(n) [parent=Global]

f1: sum_of_squares [parent=Global]
        n               3
        n_squared       9
        Return value    14

f2: sum_of_squares [parent=Global]
        n               2
        n_squared       4
        Return value    5

f3: sum_of_squares [parent=Global]
        n               1
        n_squared       1
        Return value    1
```

Another Example

indexing an element of a sequence

```
def first(s):
    """Return the first element in a sequence."""
    return s[0]
def rest(s):
    """Return all elements in a sequence after the first"""
    return s[1:]
```

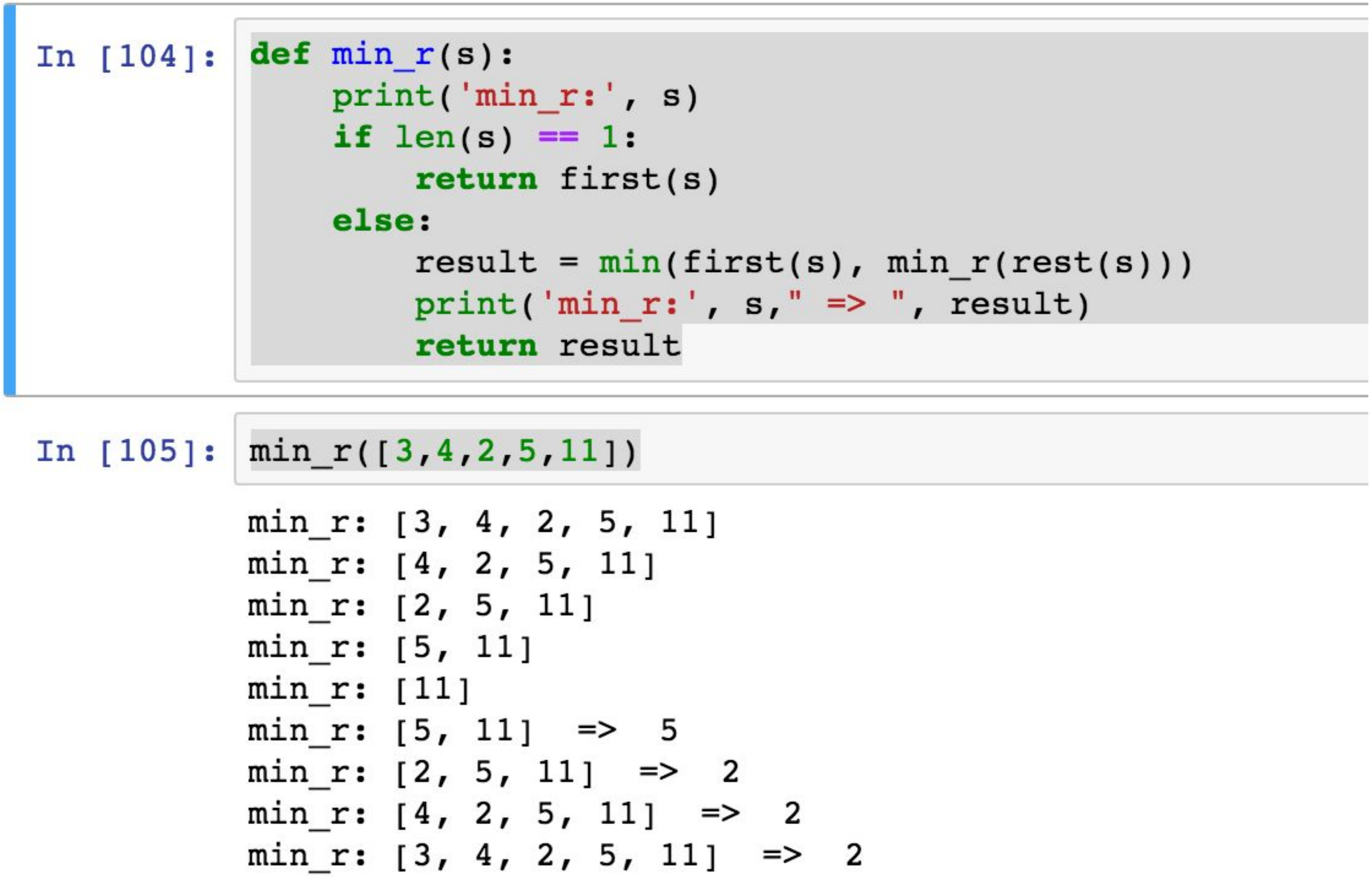
Slicing a sequence of elements

```
def min_r(s):
    """Return minimum value in a sequence."""
    if      Base Case

    else:
             Recursive Case
```

- **Recursion over sequence length, rather than number magnitude**

Visualize its behavior (print)



```
In [104]: def min_r(s):
              print('min_r:', s)
              if len(s) == 1:
                  return first(s)
              else:
                  result = min(first(s), min_r(rest(s)))
                  print('min_r:', s," => ", result)
                  return result
```

```
In [105]: min_r([3,4,2,5,11])
```

```
min_r: [3, 4, 2, 5, 11]
min_r: [4, 2, 5, 11]
min_r: [2, 5, 11]
min_r: [5, 11]
min_r: [11]
min_r: [5, 11]  =>  5
min_r: [2, 5, 11]  =>  2
min_r: [4, 2, 5, 11]  =>  2
min_r: [3, 4, 2, 5, 11]  =>  2
```

- **What about sum?**
- **Don’t confuse print with return value**

Recursion with Higher Order Fun

```
def map(f, s):
    if  Base Case

    else:
        Recursive Case

def square(x):
    return x**2

>>> map(square, [2,4,6])
[4, 16, 36]
```

- **Divide and conquer**

Trust …

- The recursive “leap of faith” works as long as we hit the base case eventually

How much ???

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

Linear
proportional to n
cn for some c

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

- Count the frames…
- Recursive is linear, iterative is constant !
- And what about the order of evaluation ?

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
```

Using HOF to preserve interface

```
def sum_of_squares(n):
    def sum_upper(i, accum):
        if i > n:
            return accum
        else:
            return sum_upper(i+1, accum + i*i)

    return sum_upper(1,0)
```

- What are the globals and locals in a call to `sum_upper`?
  - Try python tutor
- Lexical (static) nesting of function def within def - vs
- Dynamic nesting of function call within call

Tree Recursion

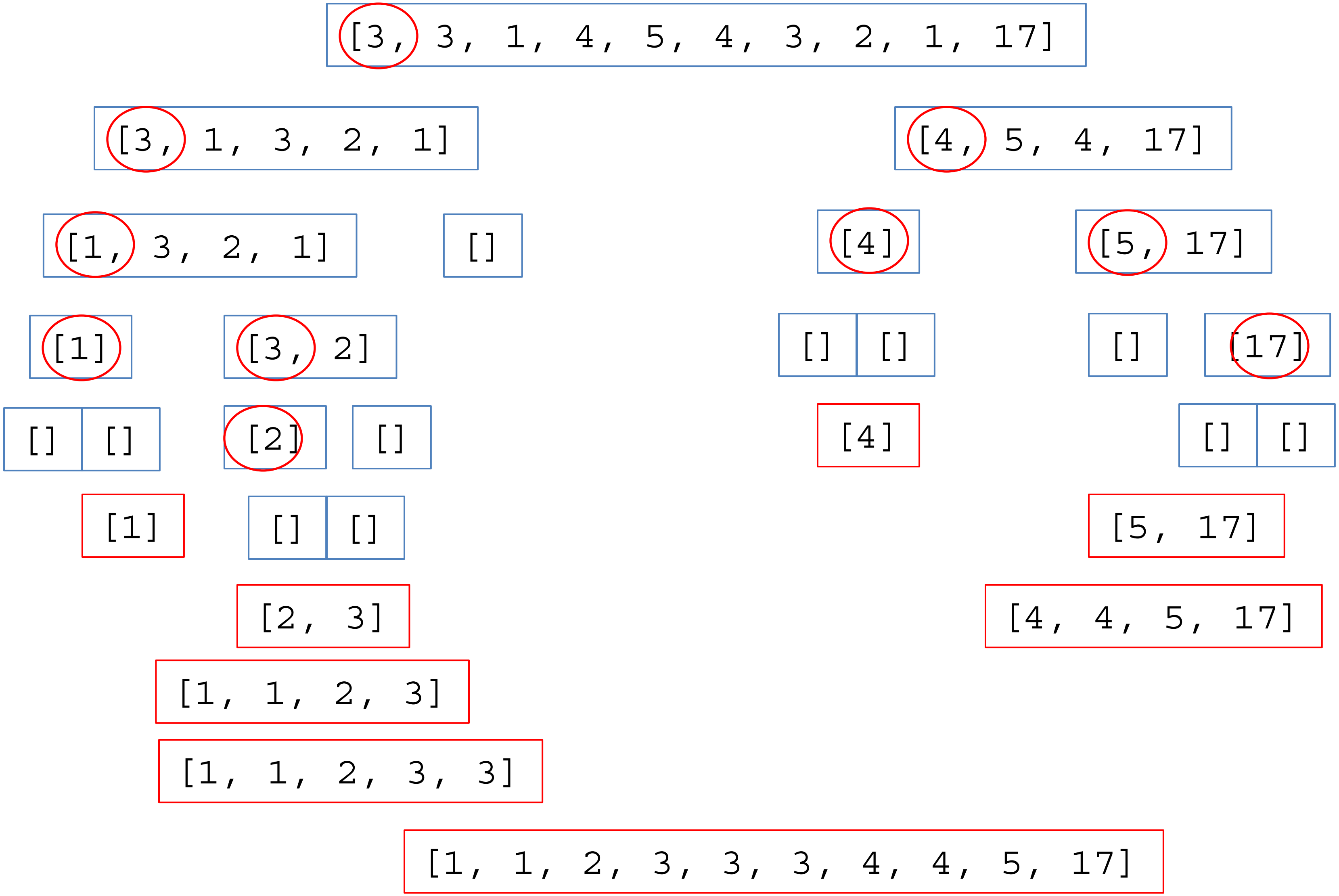
- **Break the problem into multiple smaller sub-problems, and Solve them recursively**

```
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



[3, 3, 1, 4, 5, 4, 3, 2, 1, 17]

[3, 1, 3, 2, 1] [4, 5, 4, 17]

[1, 3, 2, 1] [] [4] [5, 17]

[1] [3, 2] [] [] [] [17]

[] [] [2] [] [4] [] []

[1] [] [] [5, 17]

[2, 3] [4, 4, 5, 17]

[1, 1, 2, 3]

[1, 1, 2, 3, 3]

[1, 1, 2, 3, 3, 3, 4, 4, 5, 17]

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]
```

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