UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Recursion II

Announcements

- Maps due Friday 3/4

Learning Objectives

- Write a recursive function
- Understand the base case and a recursive case
- Apply Recursive Functions to lists

The Recursive Process

- Recursive solutions involve two major parts:
  - Base case(s), the problem is simple enough to be solved directly
  - Recursive case(s). A recursive case has three components:
    - Divide the problem into one or more simpler or smaller parts
    - Invoke the function (recursively) on each part, and
    - Combine the solutions of the parts into a solution for the problem.

Iteration vs Recursion: Sum Numbers

For loop:

```
def sum(n):
    s=0
    for i in range(0,n+1):
        s=s+i
    return s
```

Iteration vs Recursion: Sum Numbers

While loop:

```
def sum(n):
    s=0
    i=0
    while i<n:
        i=i+1
        s=s+i
    return s
```

Iteration vs Recursion: Sum Numbers

Recursion:

```
def sum(n):
    if n == 0:
        return 0
    return n+sum(n-1)
```

Iteration vs Recursion: Cheating!

Sometimes it's best to just use a formula! But that's not always the point. ☺

```
def sum(n):
    return (n * (n + 1)) / 2
```

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
```

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

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. ”Combine” the simpler part of the solution, with the recursive case

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
```

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

- Each call has its own “frame” of local variables
- Let’s see the environment diagrams

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

https://goo.gl/CiFaUJ

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 stops on the simple base case

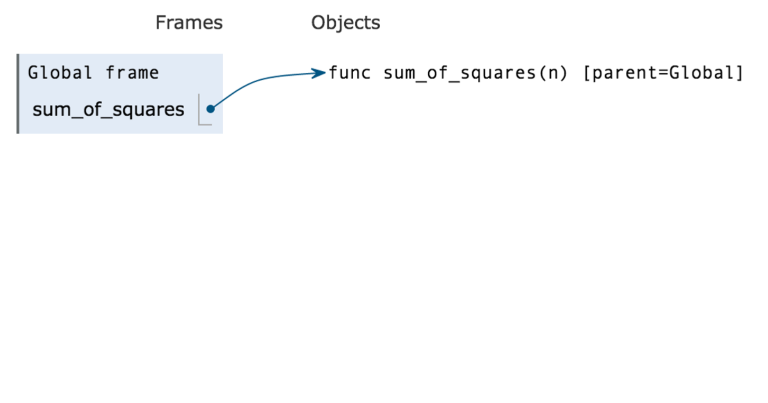
Environments Example

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)
```

Edit code

<< First | < Back | Step 2 of 17 | Forward > | Last >>

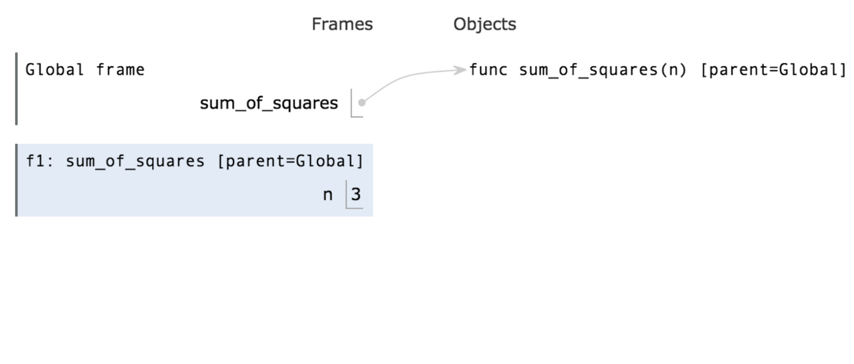


Python 3.3

```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)
```

Edit code

<< First | < Back | Step 3 of 17 | Forward > | Last >>



pythontutor.com

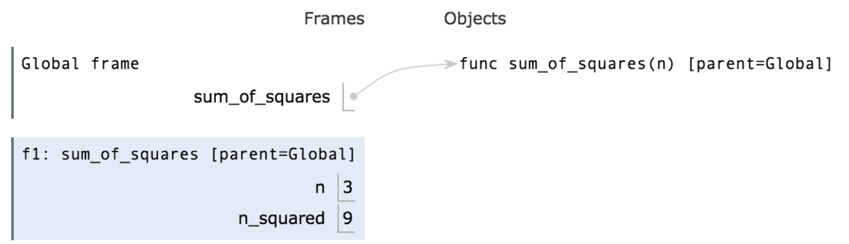
Environments Example

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)
```

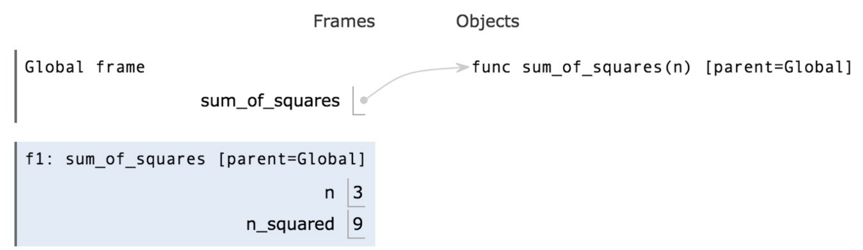
Edit code

<< First | < Back | Step 5 of 17 | Forward > | Last >>



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)
```

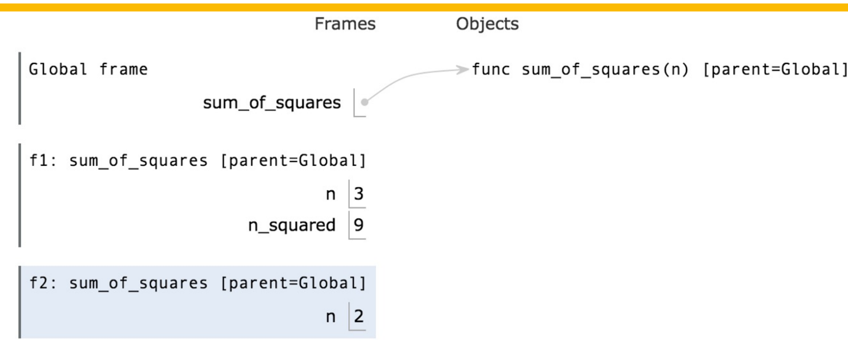


Environments Example

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)
```

Edit code

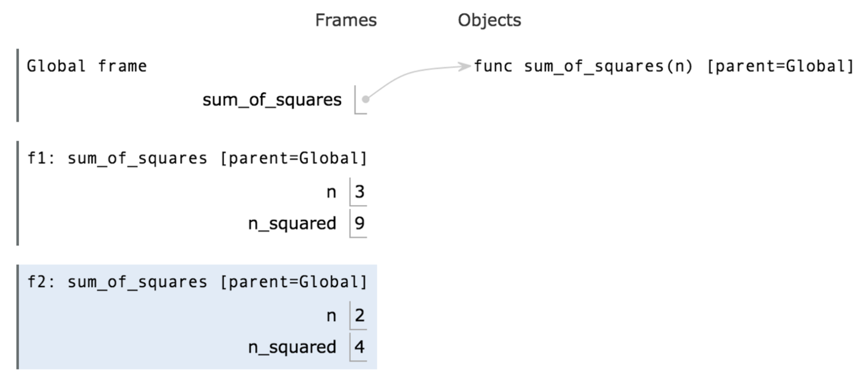


Python 3.3

```
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)
```

Edit code

<< First | < Back | Step 9 of 17 | Forward > | Last >>



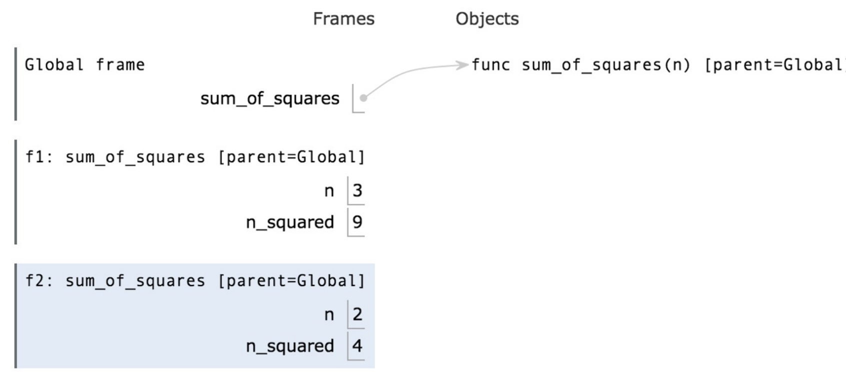
Environments Example

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)
```

Edit code

<< First < Back Step 10 of 17 Forward > Last >>



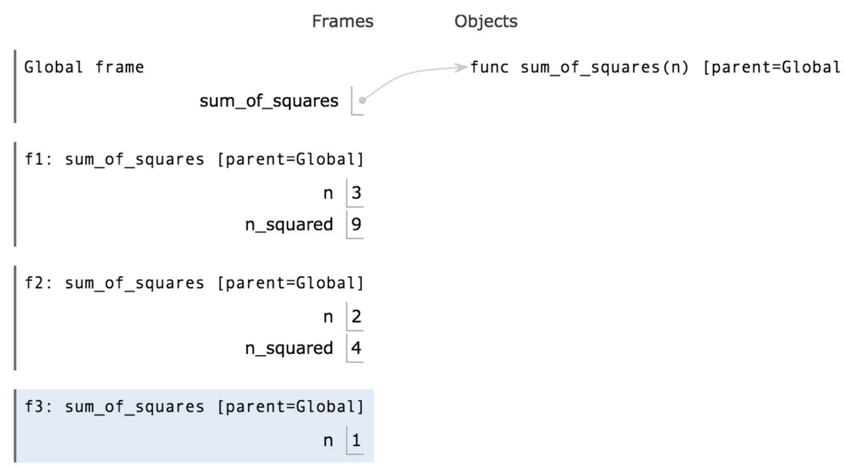
Python 3.3

```
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)
```

Edit code

<< First < Back Step 11 of 17 Forward > Last >>

that has just executed
: line to execute



Environments Example

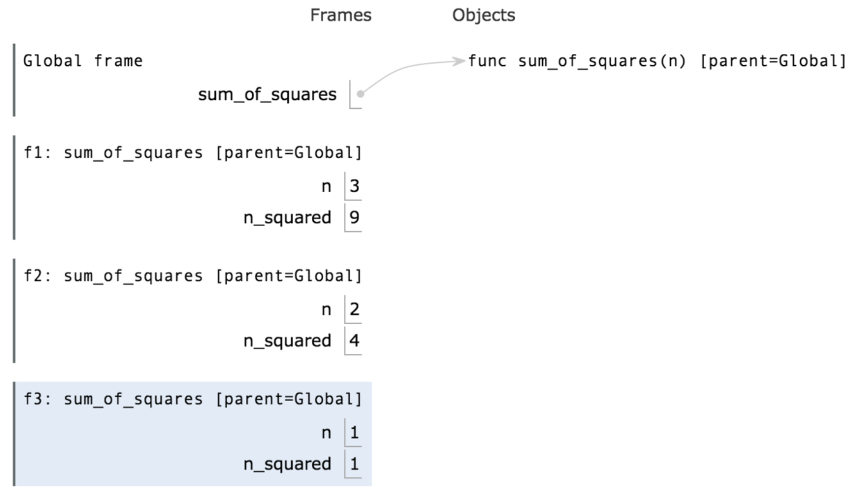
Python 3.3

```
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)
```

Edit code

<< First < Back Step 13 of 17 Forward > Last >>

that has just executed
t line to execute



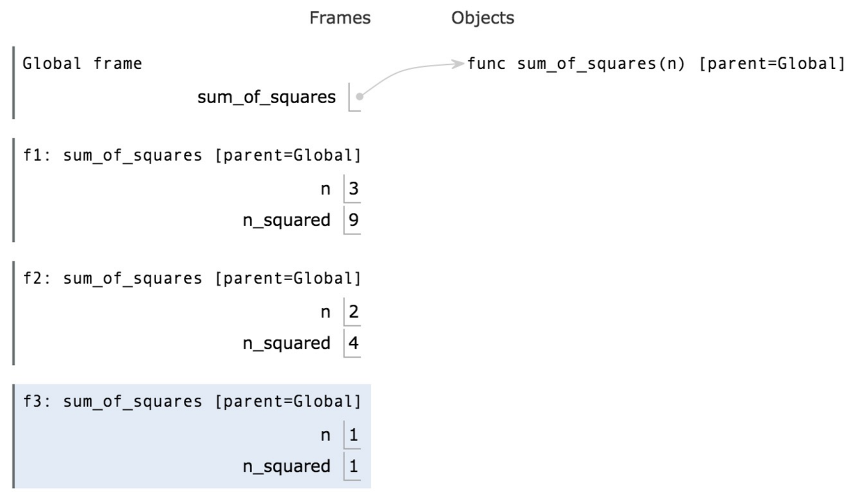
Python 3.3

```
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)
```

Edit code

<< First < Back Step 14 of 17 Forward > Last >>

that has just executed
t line to execute



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)
```

Edit code

<< First   < Back   Step 15 of 17   Forward >   Last >>

e that has just executed
xt line to execute

Frames   Objects

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

Python 3.3

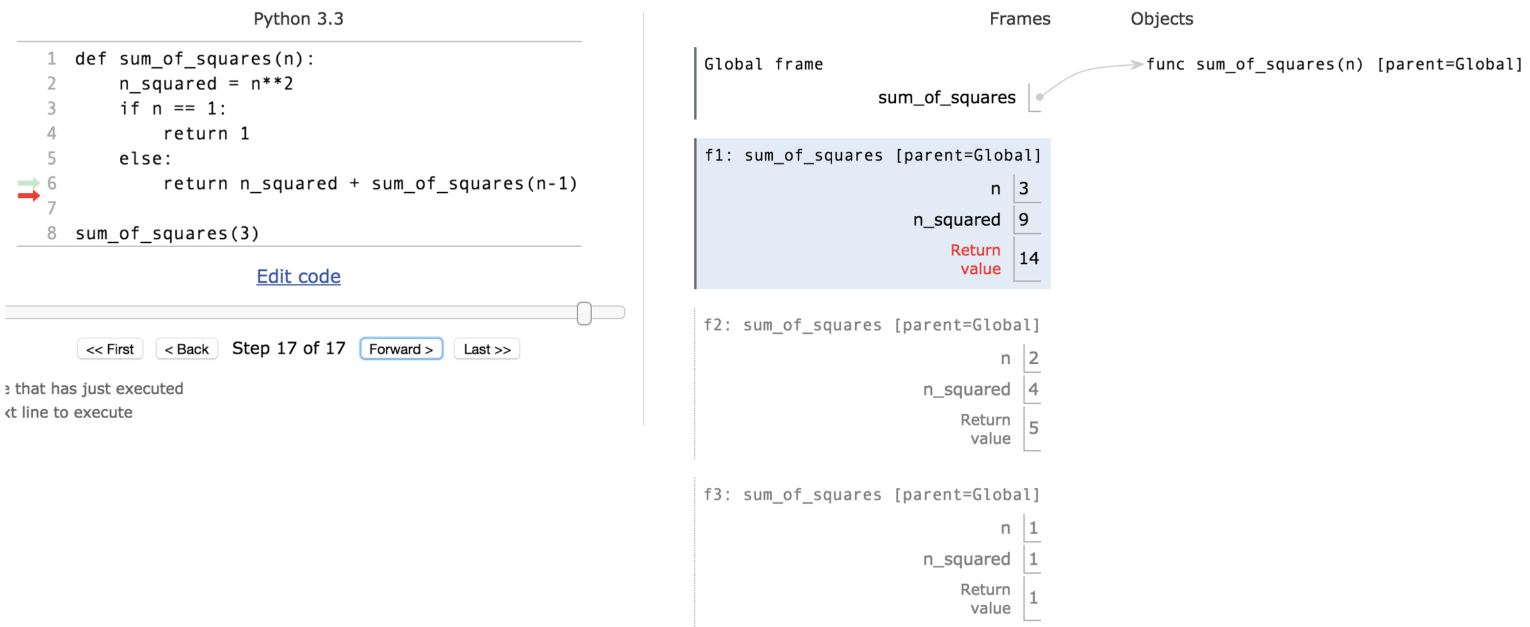
```python
def sum_of_squares(n):
    n_squared = n**2
    if n == 1:
        return 1
    else:
        return n_squared + sum_of_squares(n-1)

sum_of_squares(3)
```

Edit code

<< First | < Back | Step 16 of 17 | Forward > | Last >>

: that has just executed
t line to execute

Environments Example



Recursion Visualizer

- A new tool, similar to PythonTutor which shows just the recursive calls.
- View Recursion

Recursion With Lists

- Goal: Find the smallest item in a list, recursively.
- Consider: How do we break this task into smaller parts? What is the "smallest list"?
  - We care about the size of the list itself, not the values.

```
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:]

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

    else:
           Recursive Case
```

`min_r`

- Works because we can eventually call min() with just two numbers
- Each recursive call shrinks the list by 1 element.
- Python Tutor Link (with first and rest functions)
- Python Tutor (no first/rest defined)
  - This is just shorter and reduces the number of frames, but the same recursive calls
- Sadly recursionvisualizer.com doesn't work on this example ☹

Recursion With Strings, and Other Iterables

- Consider the lists example. It's basically the same thing. ☺
- Recursive case: Split up the item into a small "first" item, and the "rest"

```
def reverse(s):
    """
    >>> reverse('hello')
    'olleh'
    >>> reverse(reverse('hello'))
    'hello'
    """
    if not s:
        return ''
    return reverse(rest(s)) + first(s)
    # return reverse(s[1:]) + s[0]
```

Why Recursion?

- “After Abstraction, Recursion is probably the 2^{nd} biggest idea in this course”
- “It’s tremendously useful when the problem is self-similar”
- “It’s no more powerful than iteration, but often leads to more concise & better code”
- “It’s more ‘mathematical’”
- “It embodies the beauty and joy of computing”