Computational Structures in Data Science

Recursion II

UC Berkeley | Computer Science 88 | Michael Ball | https://cs88.org
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:

```python
def sum(n):
    s=0
    for i in range(0,n+1):
        s=s+i
    return s
```
Iteration vs Recursion: Sum Numbers

While loop:

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

```python
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. 😊

```python
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 $n^2$

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

```python
def sum_of_squares(n):
    accum = 0
    for i in range(1, n+1):
        accum = accum + i*i
    return accum
```
Recursion Key concepts – by example

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

1. Test for simple “base” case
2. Solution in simple “base” case
3. Assume recursive 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 $n^2$

```python
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?

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

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

```python
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

```python
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
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

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

Frames

```
Global frame
```

Objects

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

Frames

```
Global frame
```

Objects

```
f1: sum_of_squares [parent=Global]
n 3
```
Environments Example

```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)
```

Frames

- Global frame

Objects

- `func sum_of_squares(n) [parent=Global]`
- `n 3`
- `n_squared 9`
Environments Example

Python 3.3

```python
1 def sum_of_squares(n):
2     n_squared = n**2
3     if n == 1:
4         return 1
5     else:
6         return n_squared + sum_of_squares(n-1)
7     sum_of_squares(3)
```

Frames and 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`

Edit code

UC Berkeley | Computer Science 88 | Michael Ball | http://cs88.org
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)
```

Frames

Global frame

```
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
```

that has just executed

*: line to execute*
Environments Example

```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)
```

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

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

Frames  | `f2: sum_of_squares [parent=Global]`
---|---
| `n` | 2 |
| `n_squared` | 4 |

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

that has just executed the line to execute

that has just executed the line to execute
Environments Example

```python
def sum_of_squares(n):
    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
  - `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)
```

Frames | Objects
-------|--------
Global Frame | `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

```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)
```

Frames

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

- `sum_of_squares`
  - `n` 3
  - `n_squared` 9
  - Return value 14

- `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
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.
    ```python
    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:]
    ```

    ```python
    def min_r(s):
        'Return minimum value in a sequence.'
        if len(s) == 1:
            return first(s)
        else:
            return min(first(s), min_r(rest(s)))
    ```
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”

```python
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 2nd 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”