Lecture 6: Recursion

CS 61A - Summer 2024
Raymond Tan

Review: Environment Diagrams

Example: Function Composition

[Python Tutor Link]

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

def square(x):
    return x * x

def triple(x):
    return 3 * x

def compose1(f, g):
    def h(x):
        return f(g(x))
    return h
```

```
>>> square(5)
25
>>> triple(5)
15
>>> squiple = compose1(square, triple)
>>> squiple(5)
225
>>> tripare = compose1(triple, square)
>>> tripare(5)
75
>>> squadder = compose1(square, make_adder(2))
>>> squadder(3)
25
>>> compose1(square, make_adder(2))(3)
25
```

Abstraction

Review: How do we talk about functions?

```python
def square(x):
    return mul(x, x)
```

A function's **domain** is the set of all possible inputs it can take

`square` can take in any single number for x

A function's **range** is the set of all possible outputs it can give

`square` returns a non-negative (real) number

A function's **behavior** is the relationship between inputs and outputs

`square` returns the square of **x**

Functional Abstraction

```python
def sum_of_squares(x, y):
    return square(x) + square(y)
```

What does sum_of_squares need to know about square?

square takes one argument.	Yes
square has the intrinsic name square.	No
square computes the square of a number.	Yes
square computes the square by calling mul.	No

Black box


```python
def square(x):
    return mul(x,x)
```

```python
def square(x):
    return pow(x,2)
```

```python
def square(x):
    return mul(x,x-1)+x
```

Recursive Functions

Standing in line analogy

- It’s lunchtime, and you’re hungry for some noodles, so you go to your favorite restaurant in Berkeley, Noodle Dynasty. As always, there is a line out the door, so you stand at the back of the line to enter yourself into the queue. Since you’re really hungry, you start to wonder how long it will take for you to get inside and order food. You would like to know how many people are in front of you in line so that you have an idea of the wait time.

Iterative Solution

- **Problem**: Count the number of people in front of you in line.
- Solution:
  - Ask a friend to go to the front of the line
  - Count each person in line, one-by-one
  - Then, ask your friend to come back and tell you the answer

Recursive Solution

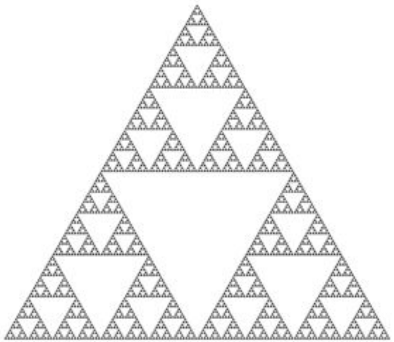
- **Problem**: Count the number of people in front of you in line.
- Solution:
    - You realize that the person at the front of the line clearly knows they're first
    - For every other person **not** at the front of the line:
        - Ask the person in front of them: “What’s your position number in line?”
        - This process repeats until we get to the front of the line
        - Once the person in front of you gets back to you, add 1 to that answer and tell the person behind you

General Structure of Recursive Functions

- **Base case(s)**: The simplest instance of the problem that can be solved without much work
    - If you're at the front of the line, you know how many people are in front of you (0)
- **Recursive call**: Making a call to the same function with a smaller input, getting you closer to the base case(s)
    - Ask the person in front of you, "What's your position in line?"
- **Recombination**: Using the result of the recursive call to solve the original problem
    - When the person in front of you tells you their answer, add one to it to get the answer to your original question

Recursive Functions

- **Definition**: A function is called recursive if the body of that function calls itself, either directly or indirectly
- Recursion is useful for solving problems with a naturally repeating structure - problems that are defined in terms of themselves.
- Recursive solutions require you to break an input into subproblems with “smaller” inputs



Example: Factorial

- A factorial of a number n is defined as:

$$n! = n \times (n-1) \times \cdots \times 1$$

    - 5! = 5 * 4 * 3 * 2 * 1
- Let’s write a Python function that will calculate n!

Factorial - Iterative

```python
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total
```

Factorial - Recursive

```python
def fact_recursive(n):
    if n == 0:
        return 1
    else:
        return n * fact_recursive(n - 1)
```

Recursion in Environment Diagrams

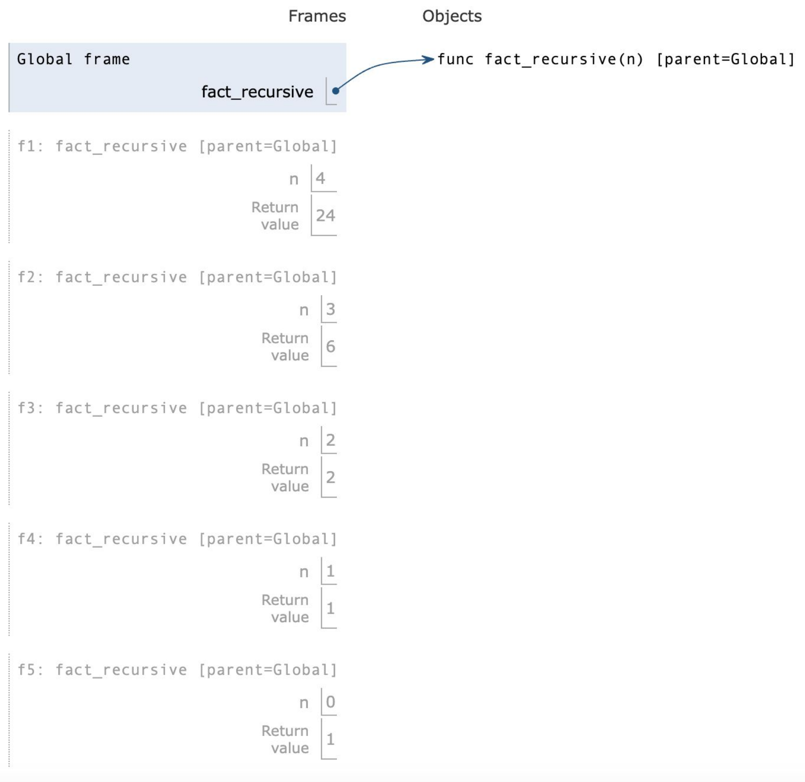
Python 3.6
([known limitations](#))

```python
def fact_recursive(n):
    if n == 0:
        return 1
    else:
        return n * fact_recursive(n - 1)

print(fact_recursive(4))
```

[Edit this code](#)

- The same function, `fact_recursive`, is called multiple times
- Different frames keep track of the different arguments in each call
- What n evaluates to depends upon the current environment
- Each call to `fact_recursive` solves a simpler problem than the last: a smaller `n`



Verifying Recursive Functions

Recursive Leap of Faith

Is `fact_recursive` implemented correctly?

1. Verify the base case
2. Treat `fact_recursive` as a **functional abstraction**
3. Assume `fact_recursive(n-1)` is correct
4. Verify that `fact_recursive(n)` is correct

Don't trace the function call all the way to the base case

```python
def fact_recursive(n):
    if n == 0:
        return 1
    else:
        return n * fact_recursive(n - 1)
```

Arms-Length Recursion

- Arms-length recursion occurs when we "reach" into the next level of recursion doing work that should be done by the next recursive call(s)
- Violates The Recursive Leap of Faith
- Is redundant, complicates code, and makes a recursive function more difficult to verify

Arms-Length Recursion: fact_recursive

```python
def fact_recursive(n):
    if n == 0 or n == 1:
        return 1
    elif n == 2:
        return 2 * 1
    elif n == 3:
        return 3 * 2 * 1
    elif n == 4:
        return 4 * 3 * 2 * 1
    elif n == 5:
        return 5 * fact_recursive(4)
    else:
        return n * fact_recursive(n - 1)
```

- This implementation of fact_recursive is correct
- However, it is redundant
    - We have explicit cases that the combination of the recursive case and base case would already handle
- Simplify repeated code with recursive calls

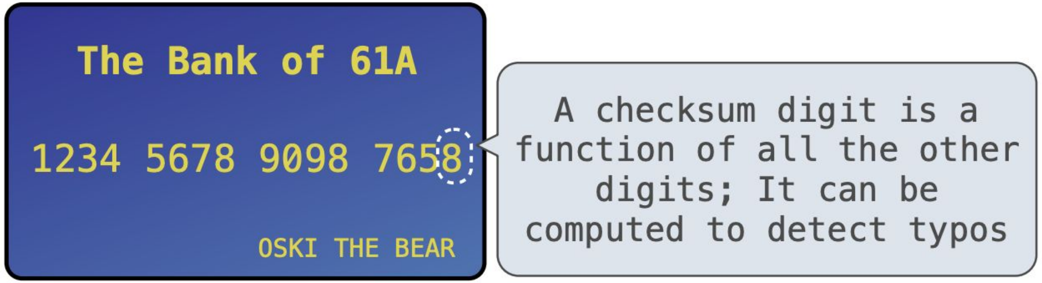
Break

More Examples

Digit Sums

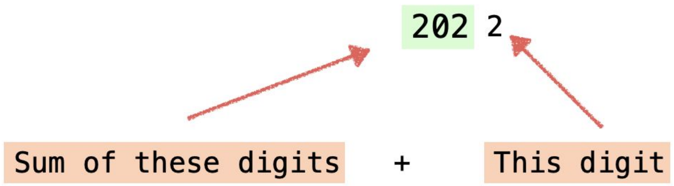
$$2+0+2+2 = 6$$

- We want a function that sums all the individual digits of a number
- If a number a is divisible by 9, then sum_digits(a) is also divisible by 9
- Useful for typo detection!



The Problem Within the Problem

- The sum of the digits of 6 is 6.
- Likewise for any one-digit (non-negative) number (i.e., < 10).
- The sum of the digits of 2022 is



That is, we can break the problem of summing the digits of 2022 into a smaller instance of the same problem, plus some extra stuff.

Converting Iteration to Recursion

- More formulaic: **Iteration is a special case of recursion**
- Idea: The state of an iteration can be passed as arguments

```
def sum_digits_iter(n):
    digit_sum = 0
    while n > 0:
        n, last = split(n)
        digit_sum = digit_sum + last
    return digit_sum
```

Updates via assignment become...

```
def sum_digits_rec(n, digit_sum):
    if n == 0:
        return digit_sum
    else:
        n, last = split(n)
        return sum_digits_rec(n, digit_sum + last)
```

...arguments to a recursive call

Example: largest_drop

Return the largest drop of N where the digits of N are read both left-to-right and right-to-left. That is, return the largest drop in value going from a digit in N to an adjacent digit in N

Summary

- Recursive functions are functions that call themselves
- Creating recursive solutions consist of 3 steps:
    - Base case
    - Recursive case
    - Recombination
- Treat recursive function calls as abstractions
    - Take the recursive leap of faith
    - Avoid arms-length recursion
- Iteration is a special form of recursion