Lecture 6: Recursion

June 28, 2022
Laryn Qi

Announcements

Abstraction

Describing Functions

```
def square(x):
    """Return the square of X"""
    ...
```

A function's **domain** is the set of all inputs it can possibly take as arguments.

`x` is a number

A function's **range** is the set of all outputs it can possibly return.

`square` returns a non-negative number

A function's **behavior** is the relationship it creates between input and output.

`square` returns the square of `x`

Functional Abstraction

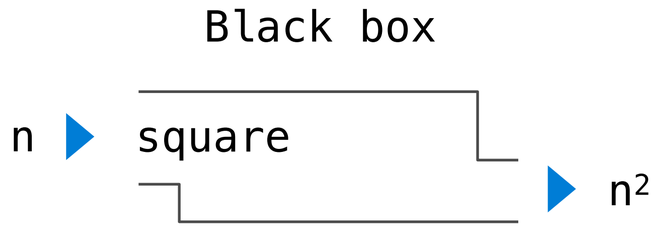
```
def sum_of_squares(x, y):
    return square(x) + square(y)
```

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**

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

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

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

Recursive Functions

Analogy for Recursion

It's 12:30 PM. You just finished listening very intently to CS 61A Lecture (you are extra happy with yourself today because you mustered the confidence to ask all the questions you were wondering and even collaborated with the people around you when Laryn asked). Immediately, you begin sprinting to Crossroads Dining Hall to beat the afternoon rush. As you arrive, you see a massive line with more people than you can count...

You can't step out of line because you'd lose your spot.

Problem: What is your position in line?

Analogy for Recursion

Iterative Solution:

1. Ask a friend to go to the front of line.
2. Count each person in line one-by-one.
3. Then, come back and tell the answer.

Analogy for Recursion

***Recursive* Solution:**

- You realize that the person at the front of the line clearly knows they're first
- For any other person, you ask the person in front of you, "what's your position in line?"
  - The following person repeats the process until the question gets to the front
  - Once the person in front of you responds, you add one to their answer to get your answer

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 you're first.

Recursive call: making a call to the same function with a smaller input

- 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

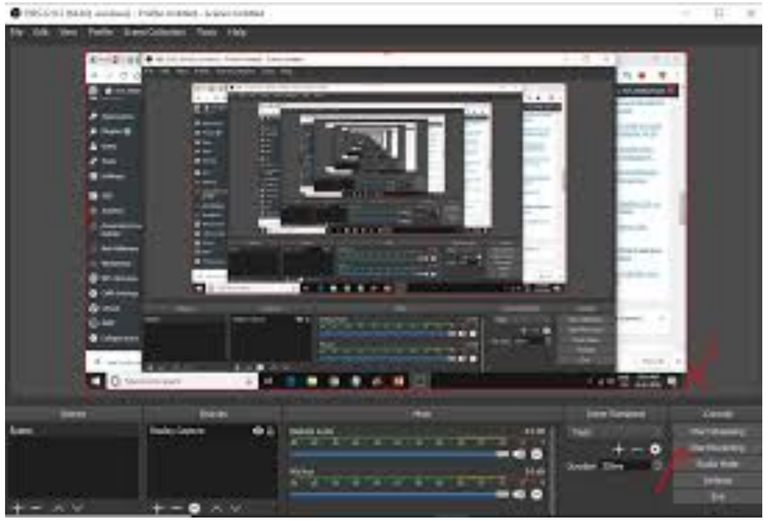
- When the person in front of you tells you their answer, **add one to it to get the answer to your original question.**

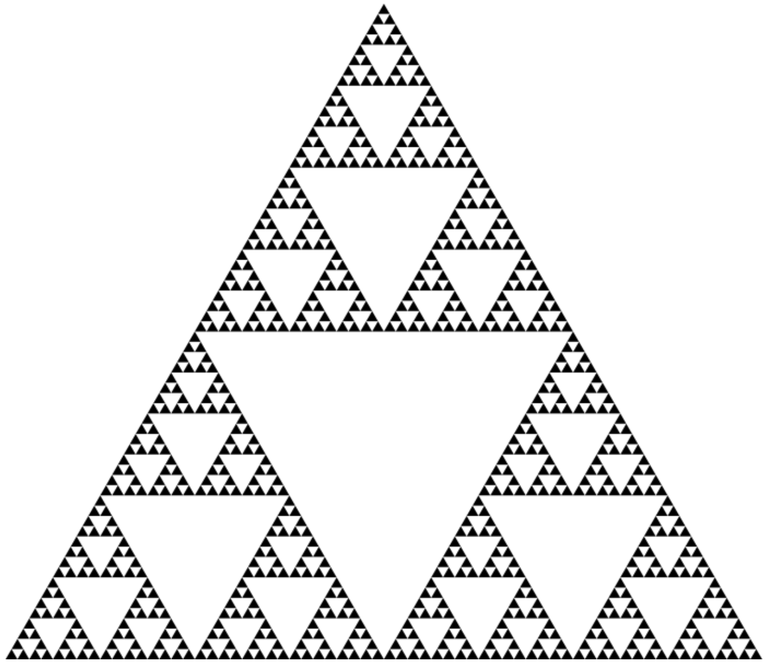
Recursive Functions

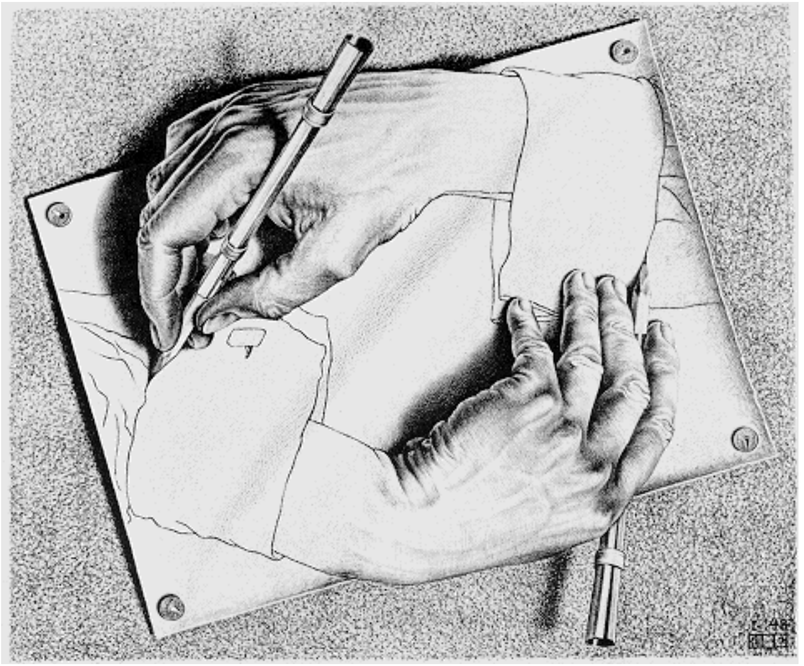
Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly.

Recursion is useful for solving problems with a naturally repeating structure – problems that are defined in terms of themselves.

Recursive solutions require you to find patterns of smaller problems and to define the smaller problem possible.







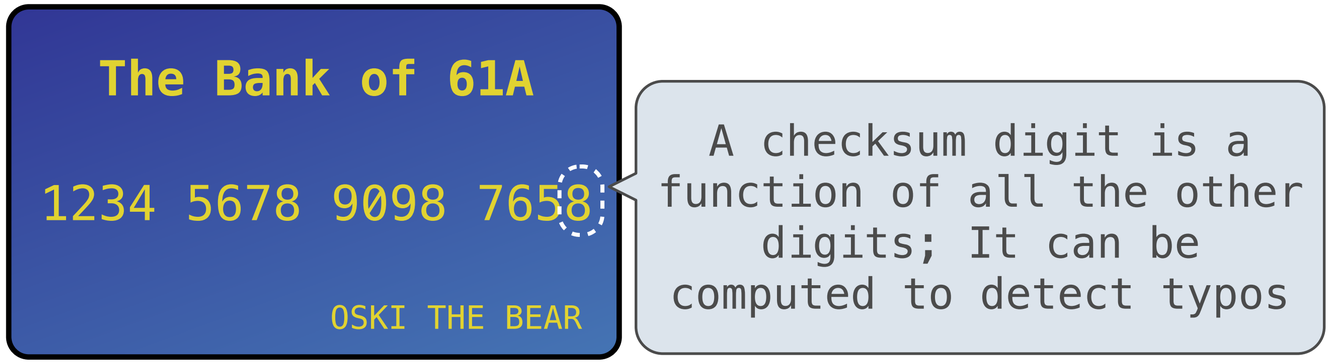
Drawing Hands, by M. C. Escher (lithograph, 1948)

Example: Sum Digits

Digit Sums

2+0+2+2 = 6

- If a number a is divisible by 9, then `sum_digits(a)` is also divisible by 9
- Useful for typo detection!



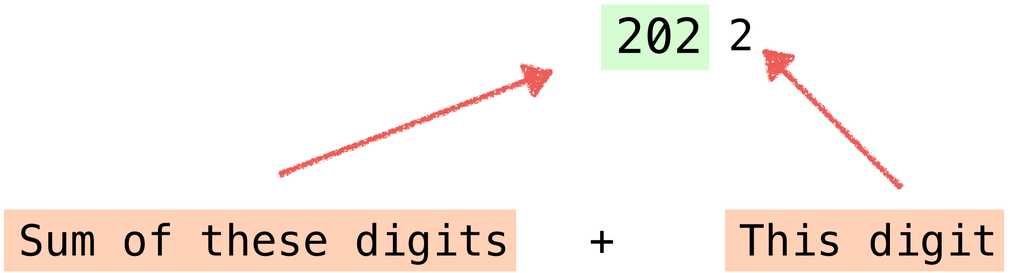
- Credit cards actually use the Luhn algorithm, which we'll implement after `sum_digits`

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.

We call this recursion.

Sum Digits Without a While Statement

```
def split(n):
    """Split positive n into all but its last digit and its last digit."""
    return n // 10, n % 10


def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        rest, last = split(n)
        return sum_digits(rest) + last
```

The Anatomy of a Recursive Function

- The `def statement header` is similar to other functions
- Conditional statements check for `base cases`
- Base cases are evaluated `without recursive calls`
- Recursive cases are evaluated `with recursive calls`

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        rest, last = split(n)
        return sum_digits(rest) + last
```

(Demo)

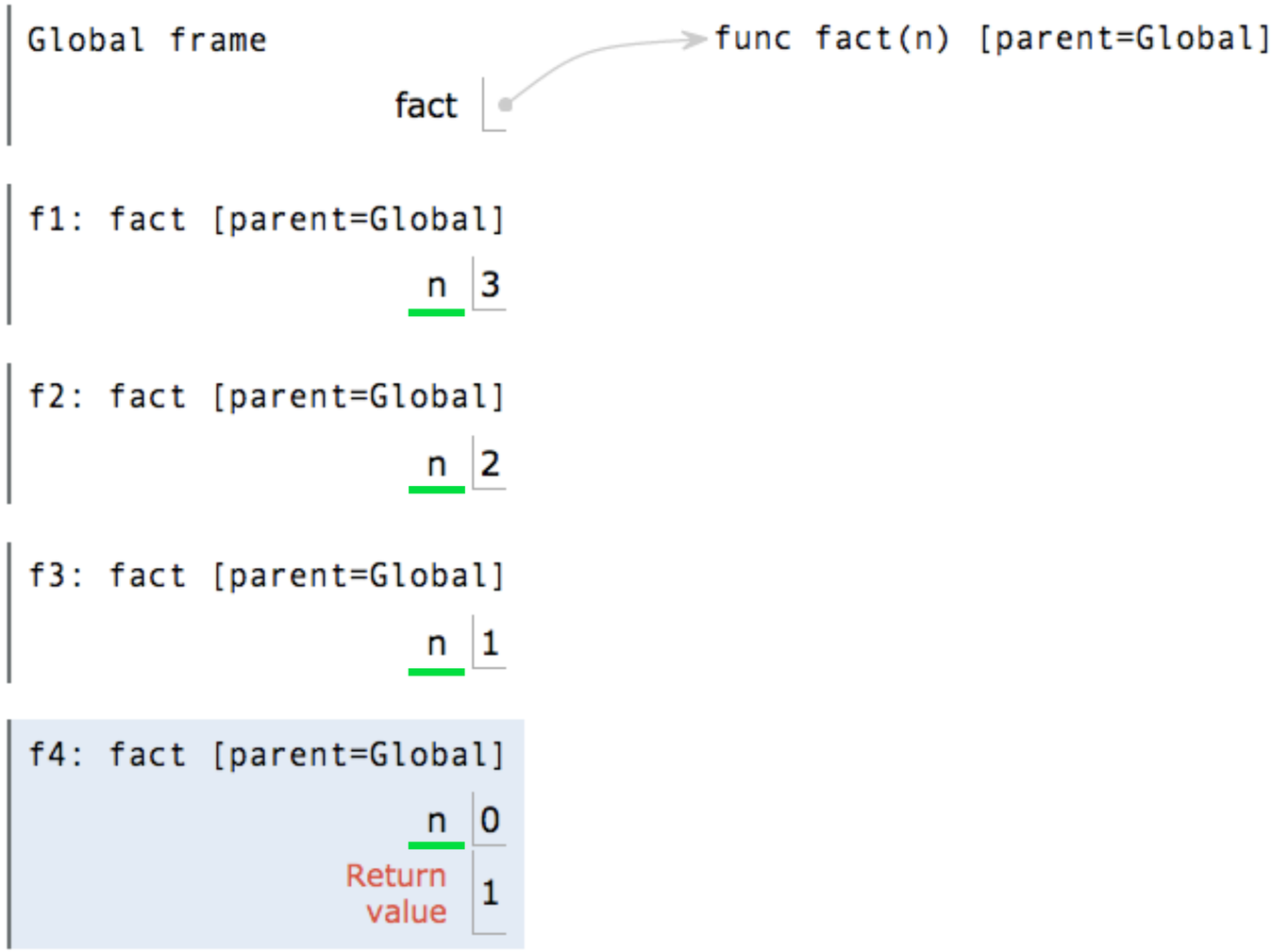
Recursion in Environment Diagrams

Recursion in Environment Diagrams

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n-1)
6
7  fact(3)
```

- The same function **fact** 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** solves a simpler problem than the last: smaller **n**

(Demo)



http://pythontutor.com/composingprograms.html#code=def%20fact%28n%29%3A%0A%20%20%20%20if%20n%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20return%201%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20n%20*%20fact%28n%20-%201%29%0A%20%20%20%20%20%20%20%20%0Afact%283%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Iteration vs Recursion

Iteration is a special case of recursion

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$$

Using while:

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

Using recursion:

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

Math:

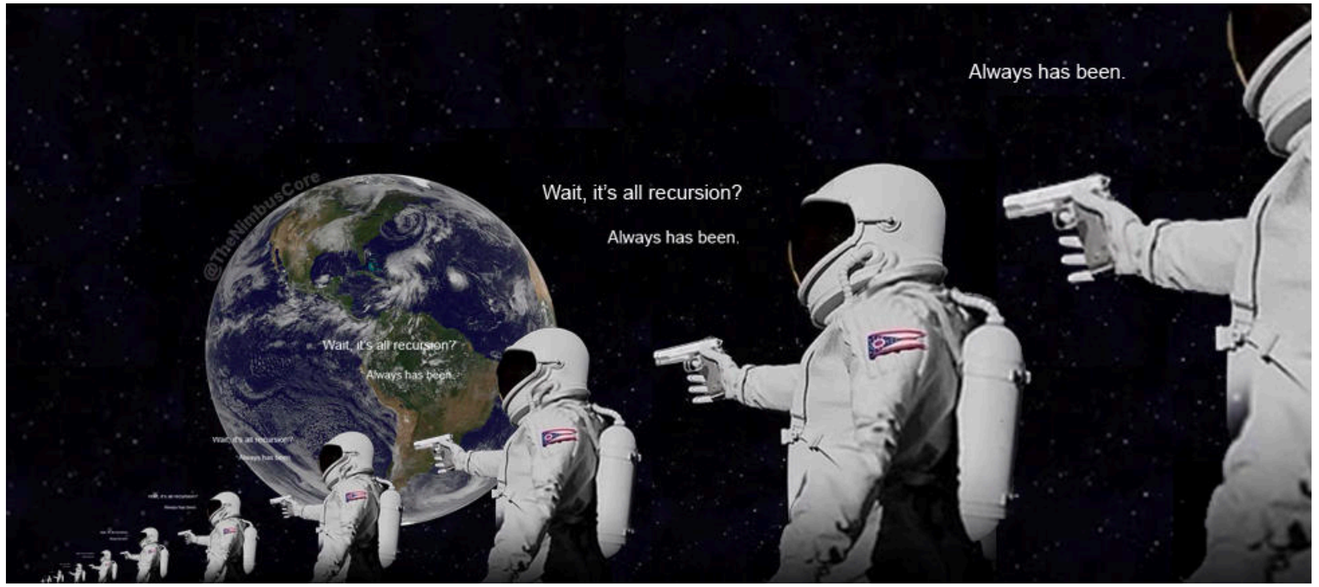
$$n! = \prod_{k=1}^{n} k$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Names: `n, total, k, fact_iter`

`n, fact`

Break


Always has been.
Wait, it's all recursion?
Always has been.
@TheNimbusCore
Wait, it's all recursion?
Always has been.

Verifying Recursive Functions

The Recursive Leap of Faith

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

Is fact implemented correctly?

1. Verify the base case

2. Treat **fact** as a ***functional abstraction***!

3. Assume that **fact**(**n**-1) is correct

4. Verify that **fact**(**n**) is correct

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

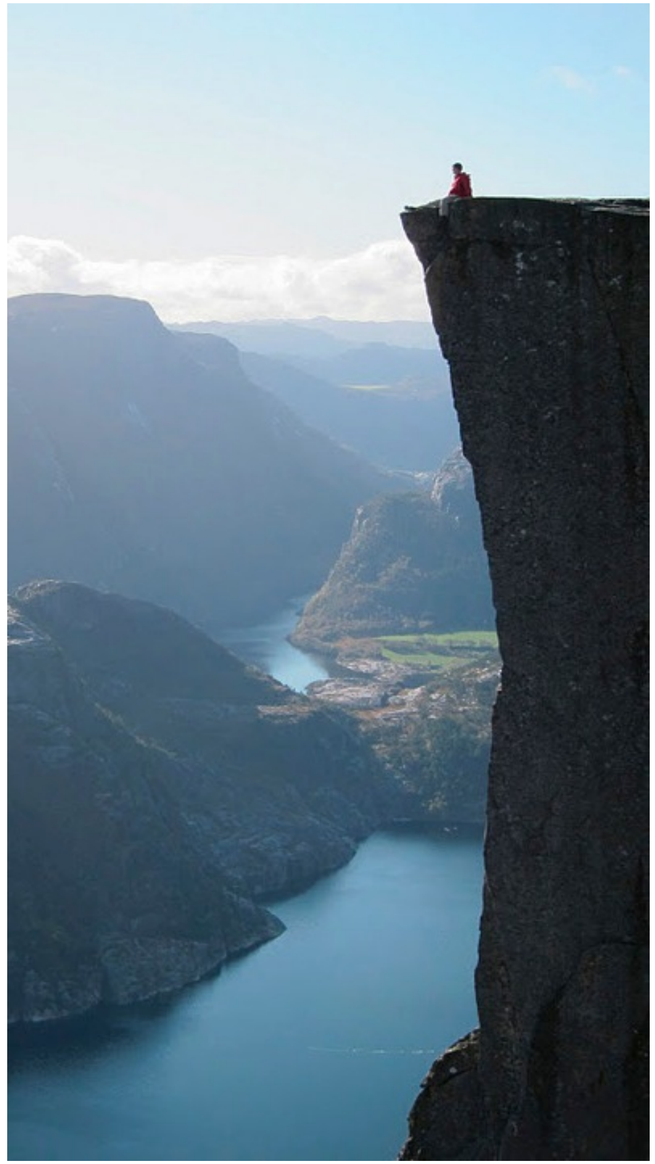


Photo by Kevin Lee, Preikestolen, Norway

Arms-length Recursion

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

```
def fact(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(4)
    else:
        return n * fact(n-1)
```

Is this implementation of fact correct?

Yes

To avoid arms-length recursion, simplify repeated code with recursive calls.

Recursion Practice

Example: Luhn Algorithm

The Luhn Algorithm

Used to verify credit card numbers

From Wikipedia: http://en.wikipedia.org/wiki/Luhn_algorithm

- **First:** From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if product of this doubling operation is greater than 9 (e.g., 7 * 2 = 14), then sum the digits of the products (e.g., 10: 1 + 0 = 1, 14: 1 + 4 = 5)

- **Second:** Take the sum of all the digits

1	3	8	7	4	3
2	3	1+6=7	7	8	3

= 30

The Luhn sum of a valid credit card number is a multiple of 10 (Demo)

Recursion and Iteration

Converting Recursion to Iteration

Can be tricky: Iteration is a special case of recursion.

Idea: Figure out what **state** must be maintained by the iterative function.

```
def sum_digits(n):
    """Return the sum of the digits of positive integer n."""
    if n < 10:
        return n
    else:
        all_but_last, last = split(n)
        return sum_digits(all_but_last) + last
```

A partial sum

What's left to sum

(Demo)

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