Lecture 6 - Recursion

Review: Abstraction

Describing Functions

```
def square(x):
        """Return X * X"""
```

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

x is a number

A function's range is the set of output values it might possibly return.

square returns a non-negative real number

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

square returns the square of x

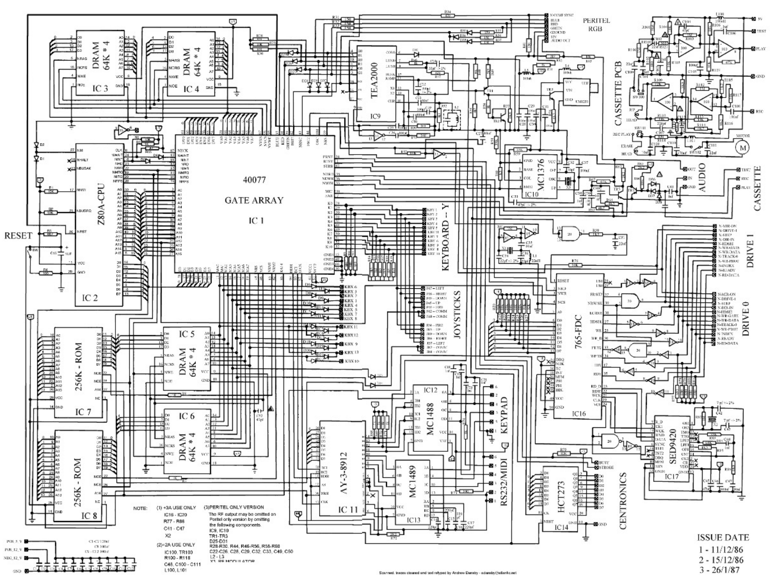
Functional Abstraction

Demo

Mechanics

How does Python execute this program line-by-line (e.g. Python Tutor)

What happens when you evaluate a call expression, what goes on its body, etc.



Use (**functional abstraction**)

- square(2) always returns 4
- square(3) always returns 9
- ...

Without worrying about *how* Python evaluates the function

Recursion

Suppose you're waiting in line for a concert.

You can't see the front of the line, but you want to know what your place in line is. Only the first 100 people get free t-shirts!

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

What should you do?

An **iterative algorithm** might say:

1. Ask my friend to go to the front of the line.
2. Count each person in line one-by-one.
3. Then, tell me the answer.

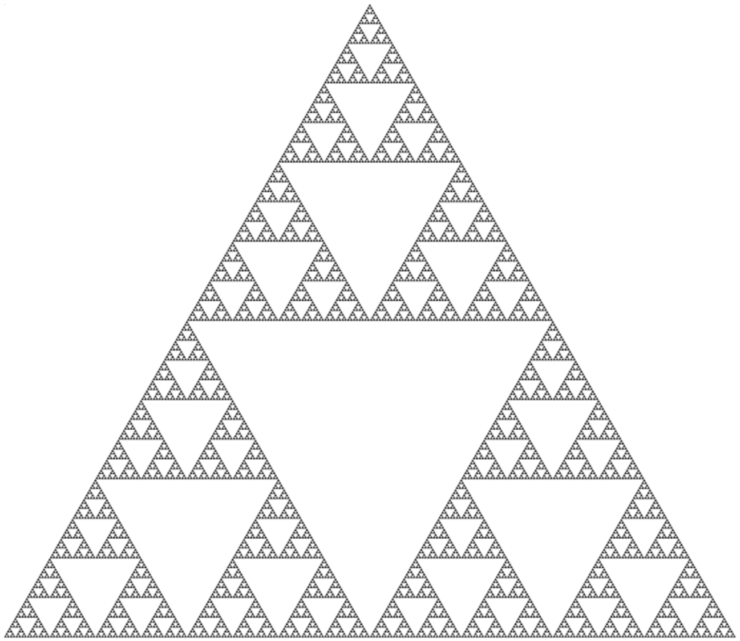
A **recursive algorithm** might say:

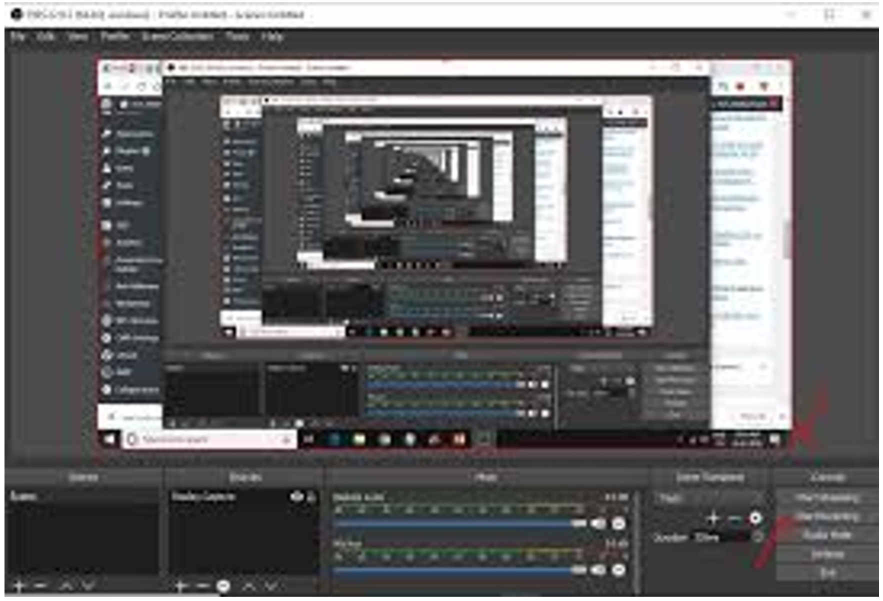
- If you're at the front, you know you're first.
- Otherwise, ask the person in front of you, **"What number in line are you?"**
- The person in front of you figures it out by asking the person in front of them who asks the person in front of them etc…
- Once they get an answer, they tell you and you add one to that answer.

Recursion

Recursion is useful for solving problems with a naturally repeating structure - they are defined in terms of themselves

It requires you to find patterns of smaller problems, and to define the smallest problem possible





Recursion in Evaluation

`f(g(h(2), True), h(x))`

`g(h(2), True)`

`h(x)`

A call expression is composed of smaller (call) expressions!

`h(2)`

Stop once you reach a number, boolean, name, etc.

Recursive Functions

Recursive Functions

- A function is called **recursive** if the body of that function calls itself, either directly or indirectly
- This implies that executing the body of a recursive function may require **applying that function multiple times**
- Recursion is inherently tied to functional abstraction

Structure of a Recursive Function

1. One or more **base cases**, usually the smallest input.
    - "If you're at the front, you know you're first."

1. One or more ways of **reducing the problem**, and then **solving the smaller problem using recursion**.
    - "Ask the person in front, 'What number in line are you?'"

1. One or more ways of **using the solution to each smaller problem** to solve our larger problem.
    - "When the person in front of you figures it out and tells you, **add one to that answer**."

Demo

Functional Abstraction & Recursion

Expression	Value
fact(1)	1
fact(3)	6 (3 * 2 * 1)
fact(4)	24 (4 * 3 * 2 * 1)
fact(n - 1)	n-1 * n-2 * ... * 1
fact(n)	~~n * n-1 * n-2 * .. * 1~~
	n * fact(n - 1)

Verifying factorial

Is factorial correct?

1. Verify the **base cases**.
    - Are they **correct**?
    - Are they **exhaustive**?

Now, harness the power of **functional abstraction**!

1. Assume that **factorial(n-1)** is correct.
2. Verify that **factorial(n)** is correct.

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

Functional abstraction: don't worry that fact is recursive and just assume that factorial gets the right answer!

Break

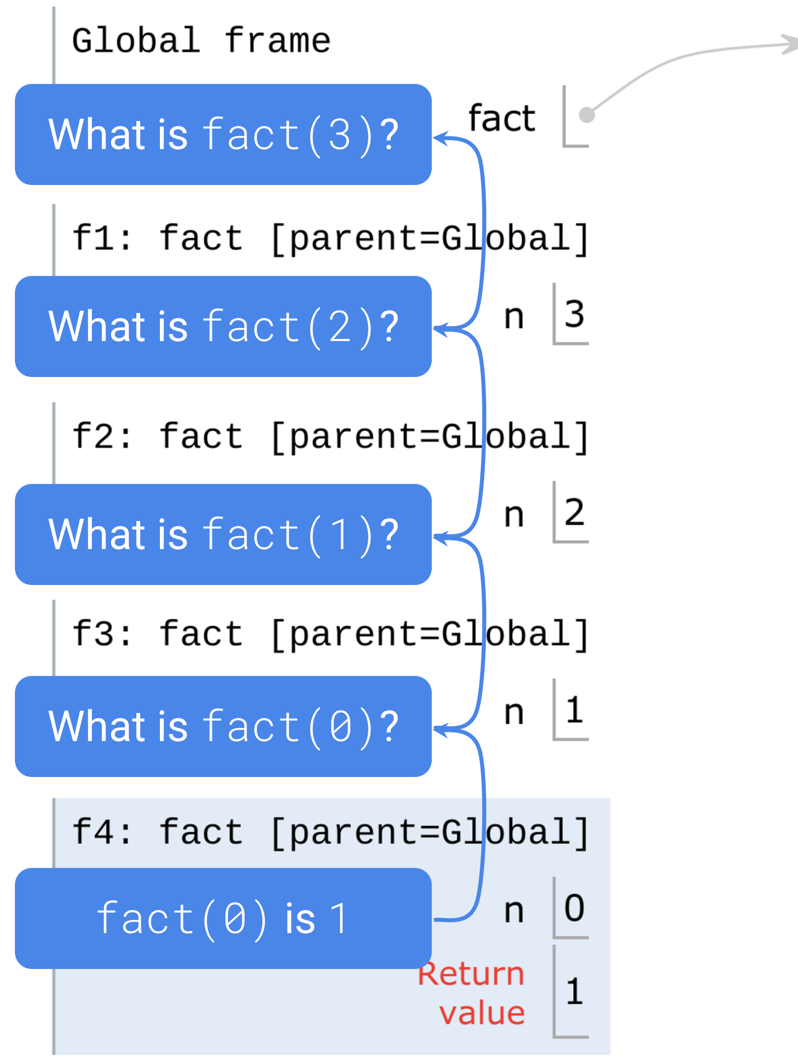
Visualizing Recursion

Demo

Recursion in Environment Diagrams

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

fact(3)
```



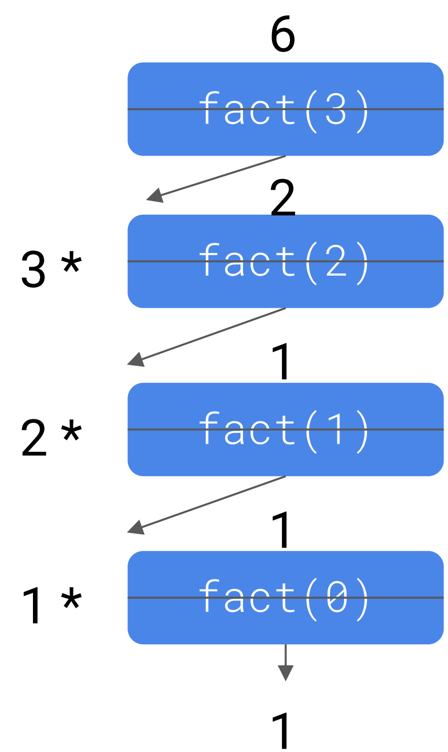
The same function fact is called multiple times, each time solving a simpler problem

All the frames share the same parent - only difference is the argument

What n evaluates to depends upon the **current environment**

Recursive tree - another way to visualize recursion

```python
def fact(n):
    """Calculates n!"""
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```



How to Trust Functional Abstraction

Assume this all works!

Look at how we computed fact(3)

- Which required computing fact(2)
    - Which required computing fact(1)
        - Which required computing fact(0)
            - Which we know is 1, thanks to the base case!

Verifying the correctness of recursive functions

1. Verify that the base cases work as expected
2. For each larger case, verify that it works by **assuming the smaller recursive calls are correct**

```
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 * 4 * 3 * 2 * 1
    elif n == 6:
        return 6 * fact(5)
    else:
       return n * fact(n-1)
```

Identifying Patterns

Is factorial correct?

1. List out all the cases.
2. Identify **patterns** between each case.
3. Simplify repeated code with **recursive calls**.

Examples

Count Up

Demo

Let's implement a recursive function to print the numbers from 1 to `n`. Assume `n` is positive .

```
def count_up(n):
    """Prints the numbers from
    1 to n.
        >>> count_up(1)
        1
        >>> count_up(2)
        1
        2
        >>> count_up(4)
        1
        2
        3
        4
    """
        "*** YOUR CODE HERE
***"
```

1. One or more **base cases**
2. One or more **recursive calls** with simpler arguments.
3. **Using the recursive call** to solve our larger problem.

Count Up - Summary

1. Base case
    - What is the smallest number where we don't have to do any work?
        - We know `n` is positive so the the smallest positive integer is 1 and if n = 1, print it out and do nothing else.
2. Recursive call with smaller arguments
    - Have access to the largest number, so try printing smaller numbers
3. Use recursive call to solve the problem
    - Once we've printed up to n - 1, what value is left?

Demo

Sum Digits

Let’s implement a recursive function to sum all the digits of `n`. Assume `n` is positive .

```
def sum_digits(n):
    """Calculates the sum of the digits `n`.
    >>> sum_digits(9)
    9
    >>> sum_digits(19)
    10
    >>> sum_digits(2019)
    12
    """
    "*** YOUR CODE HERE ***"
```

1. One or more **base cases**
2. One or more **recursive calls** with simpler arguments.
3. **Using the recursive call** to solve our larger problem.

Sum Digits Discussion

What's our:

Input?

Number

Output?

Sum of all the digits

Base case?

A single digit

Smaller problem?

Sum of all digits but one

Larger problem?

Sum of all digits but one plus the digit that was left out

Iteration vs. Recursion

- Iteration and recursion are somewhat related
- Converting **iteration to recursion** is formulaic, but converting **recursion to iteration** can be more tricky

Iterative

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

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

Names: n, total, k, fact_iter

Recursive

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

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

Names: n, fact

Summary

- Recursive functions are functions that call themselves in their body one or more times
    - This allows us to break the problem down into smaller pieces
    - Using functional abstraction, we do not have to worry about how those smaller problems are solved
- A recursive function has a base case to define its smallest problem, and one or more recursive calls
    - If we know the base case is correct, and that we get the correct solution assuming the recursive calls work, then we know the function is correct
- Evaluating recursive calls follow the same rules we’ve talked about so far