Computational Thinking

Lecture 11: Recursion

University of Engineering and Technology
VIETNAM NATIONAL UNIVERSITY HANOI

Outline

- Recursion: repetition without for-loops
- Iterative and recursive arithmetic functions
- Base cases and recursive cases
- Infinite recursion
- Recursion with Multiple Base Cases
- Recursion on Nested Data
- Recursion vs. for-loops
- Recursive Programming Pattern

Reading for today: Think Python, 3rd ed., 5.8-10, 6.6-8

Repetition Without Loops

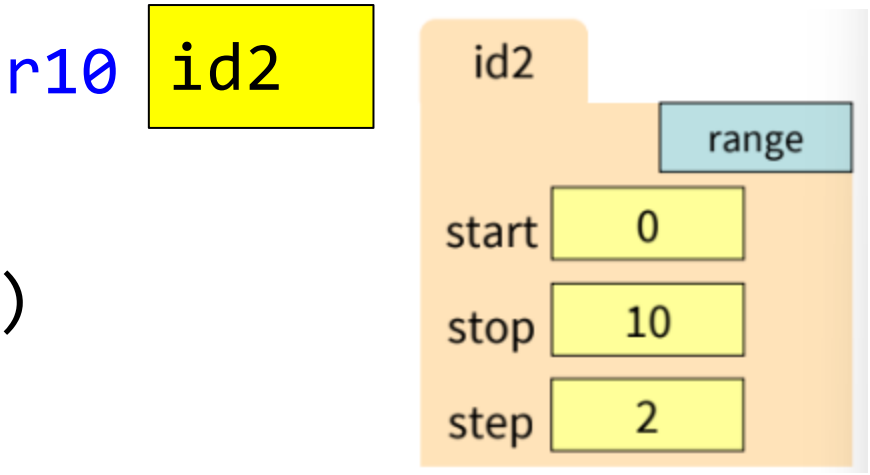
How to Print a Countdown — with Loop

```
def blast_off_loop(n):
    """Print a countdown starting at n.
        n: a non-negative int"""
    for n in range(n, 0, -1):
        print(n)
    print('BLAST OFF!')
```

New range() feature: **step**

New range feature: step

```
>>> r3 = range(3)
>>> r3.step
1
>>> r10 = range(0, 10, 2)
>>> r10.step
2
>>> list(r10)
[0, 2, 4, 6, 8]
>>> r_down = range(10, 0, -1)
>>> list(r_down)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```



How to Print a Countdown — with Loop

```
def blast_off_loop(n):
    """Print a countdown starting at n.
        n: a non-negative int"""
    for n in range(n, 0, -1):
        print(n)
    print('BLAST OFF!')
```

Could you do this **without** a loop?
Yes, and it doesn't require any new features!

How to Print a Countdown — without Loop

```
def blast_off_no_loop(n):
    """Print a countdown starting at n.
        n: a non-negative int"""
    if n == 0:
        print('BLAST OFF!')
    else:
        print(n)
        blast_off_no_loop(n-1)
```

The Function Calls Itself

- A function can call **other** helper functions
- A function can call **itself** as a helper function
  - **Recursive call:** a call from a function to itself
  - **Recursive function:** a function that contains a recursive call
- Some will tell you that execution of recursive functions is mysterious or magical
  - We have been carefully teaching you exactly what you need to know, in part to reach this moment and understand how recursive functions work
  - **Execution diagrams**, especially call stacks, are the key to understanding

Recursion vs. Iteration

- **Recursion**: the programming pattern of using recursive functions
- **Iteration**: the programming pattern of using loops
- Both accomplish the task of repetition
  - For-loop: repeat body of loop
  - Recursion: repeat body of function
- Recursion is **strictly more powerful** than iteration with for-loops*
- Both can be found not just in programming but also in **high-school mathematics…**

* That is, over collections that do not change during the iteration. Recursion is equivalent in power to iteration with while-loops, which we will learn at the end of the semester.

Factorial

How many ways to order n distinct items?

n = 2:

- A, B
- B, A
- 2 ways to order = 2 * 1

n = 3:

- A, B, C
- A, C, B
- B, A, C
- B, C, A
- C, A, B
- C, B, A
- 6 ways to order = 3 * 2 * 1

In general:

- There are *n* ways to pick the first item.
- Then *n–1* ways to pick the second item.
- ...
- Then 2 ways to pick the next-to-last item.
- Then only 1 way to pick the last item.
- For a total of:
  n*(n–1)*...*2*1
  ways to order,
  aka permutations

The Factorial Operator/Function

$$n! = n * (n - 1) * \ldots * 2 * 1$$

or

$$n! = 1 * 2 * \ldots * (n - 1) * n$$

or

$$n! = n * (n - 1)!$$

and in all cases, $0! = 1$

Iterative Factorial

$$n! = 1 * 2 * \ldots * (n - 1) * n$$
$$0! = 1$$

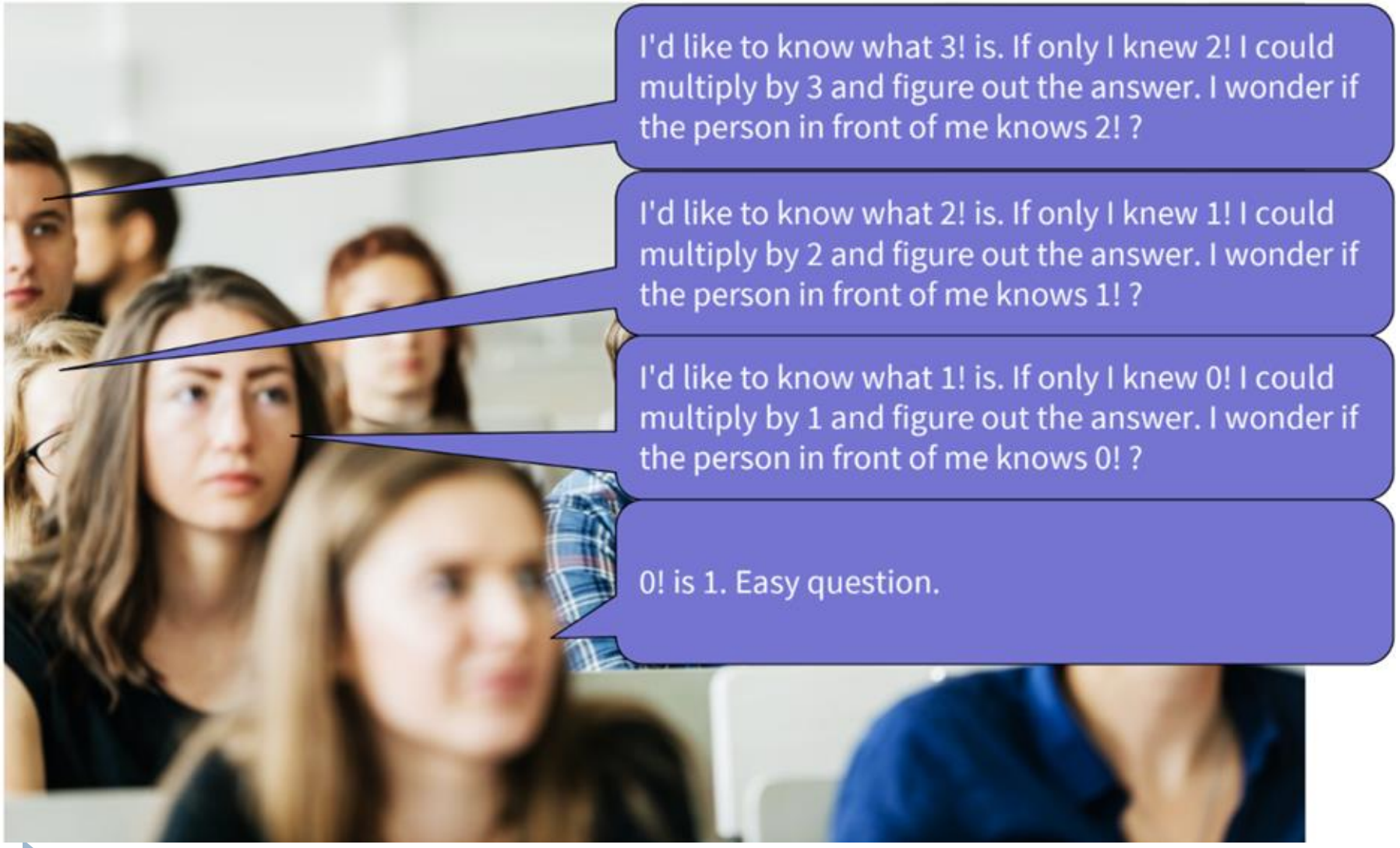
```
def factorial_iter(n):
    """Returns n!.
    n: a non-negative integer"""
    product = 1
    for i in range(1, n+1): # 1..n
        product = product * i
    return product
```

Recursive Factorial

$$n! = n * (n - 1)!$$
$$0! = 1$$

```
def factorial_rec(n):
    """Returns n!.
    n: a non-negative integer"""
    if n == 0:
        return 1
    else:
    n_minus_1_fact = factorial_rec(n-1)
    return n * n_minus_1_fact
```

An Analogy for the Call Frames



Quiz

What does this print?

```
def mystery(n):
    if n == 0:
        return 0
    else:
        n_minus_1_result = mystery(n-1)
        return n + n_minus_1_result
print(mystery(4))
```

A: 0
B: 1
C: 4
D: 7
E: 10

Vocabulary for Recursive Definitions

n! = n * (n – 1)!

0! = 1

Recursive Case

Base Case

```
def factorial_rec(n):
    """Returns n!.
    n: a non-negative integer"""
    if n == 0:
        return 1
    else:
        n_minus_1_fact = factorial_rec(n-1)
        return n * n_minus_1_fact
```

Vocabulary for Recursive Definitions

- **Base case:**
  - **Does not** make a recursive call
  - Is able to compute the answer **on its own**
- **Recursive case:**
  - **Does** make a recursive call
  - Needs **help** to compute the answer
  - It gets that help from itself...

How to Get Help from Yourself (without falling off a cliff)

The "Leap of Faith" with Recursion

- When recursive function f calls itself in the recursive case, f is making an **assumption** that f satisfies f's specification.
- But this is no different than when you call a library function f and make the **assumption** that f satisfies f's specification.
- In fact, the library function might be recursive! You'd never know without reading its source code.
- With recursion, some authors call this a **leap of faith**:* *a recursive function calls itself having faith that it will work.*
- But, the reason it works is reasonable: functions promise to meet their specifications.
  - You can **rely** on function to produce correct output.
  - As long as you **guarantee** precondition is met.

*[Søren Kierkegaard]: believing something but not based on reason

But How Can Something Be Defined in Terms of Itself?!

$$n! = n * (n - 1)!$$

$$0! = 1$$

```
def factorial_rec(n):
    """Returns n!.
    n: a non-negative integer"""
    if n == 0:
        return 1
    else:
    n_minus_1_fact = factorial_rec(n-1)
    return n * n_minus_1_fact
```

But How Can Something Be Defined in Terms of Itself?!

Q: Isn't that **circular** or **illogical**?

A1: Obviously it does work, as our demos show.

A2: But you're right to be suspicious — if we're not careful, defining something in terms of itself could lead to a big problem...**infinite recursion**.

"Infinite" Recursion

```
# recursionerrors.py
def infinite_recursion():
    infinite_recursion()
```

```
>>> import recursionerrors
>>> recursionerrors.infinite_recursion()
...
RecursionError: maximum recursion depth exceeded
```

"Infinite" Recursion

```
# recursionerrors.py
def bad_blast_off_v1(n):
    if n == 0:
        print('BLAST OFF!')
    else:
        print(n)
        bad_blast_off_v1(n) # BUG: need "n-1"

def bad_blast_off_v2(n):
    # BUG: no base case
    print(n)
    bad_blast_off_v2(n-1)
```

Python Prevents Infinite Recursion

- Number of stack frames is **limited**
- If your recursion is broken — is **circular** — your program will get stopped with a RecursionError
- Without that check by Python, such a broken program would potentially run forever — **"infinite"**
- So we don't actually observe infinite recursion; instead, we get RecursionErrors

"Infinite" Recursion

```
# recursionerrors.py
def infinite_recursion():
    infinite_recursion()
```

```
>>> import recursionerrors
>>> recursionerrors.infinite_recursion()
...
RecursionError: maximum recursion depth exceeded
```

Number of stack frames is **limited** (to 1000 by default).
So if your recursion would be infinite, your program will actually get stopped with a RecursionError.

Quiz

Which of the following could result in a RecursionError?
Precondition for both: n >= 0.

```
# A
def recurse(n):
    if n == 0:
        return
    recurse(n-1)
```

```
# B
def recurse(n):
    if n == 0:
        return
    recurse(n+1)
```

```
# C: Both of the above
```

Recursion with Multiple Base Cases

The Fibonacci Sequence

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
```

- Each number is the sum of the previous two
- Applications to:
  - Poetry: possible patterns of Sanskrit poems
  - Biology: branching in trees
  - Ecology: modeling population growth rate
  - And more [Wikipedia]

The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Recursive definition:

F(0) = 0

Base case

F(1) = 1

Base case

F(n) = F(n – 1) + F(n – 2) Recursive case

Recursive Fibonacci — Easy to Implement

F(0) = 0

Base Case

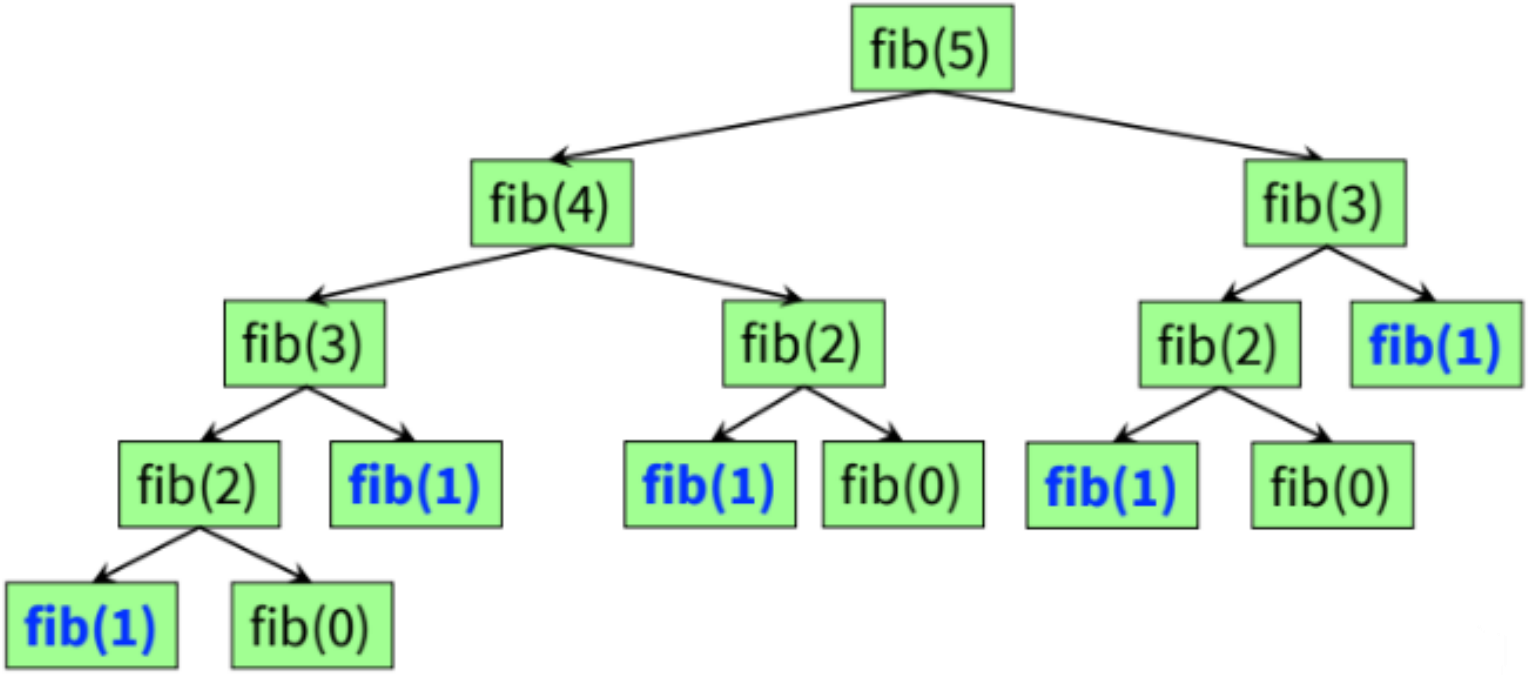
F(1) = 1

F(n) = F(n-1) + F(n – 2) Recursive Case

```
def fibo_rec(n):
    """Returns F(n).
    n: a non-negative integer"""
    if n == 0 or n == 1:
        return n
    else:
        return fibo_rec(n-1) + fibo_rec(n-2)
```

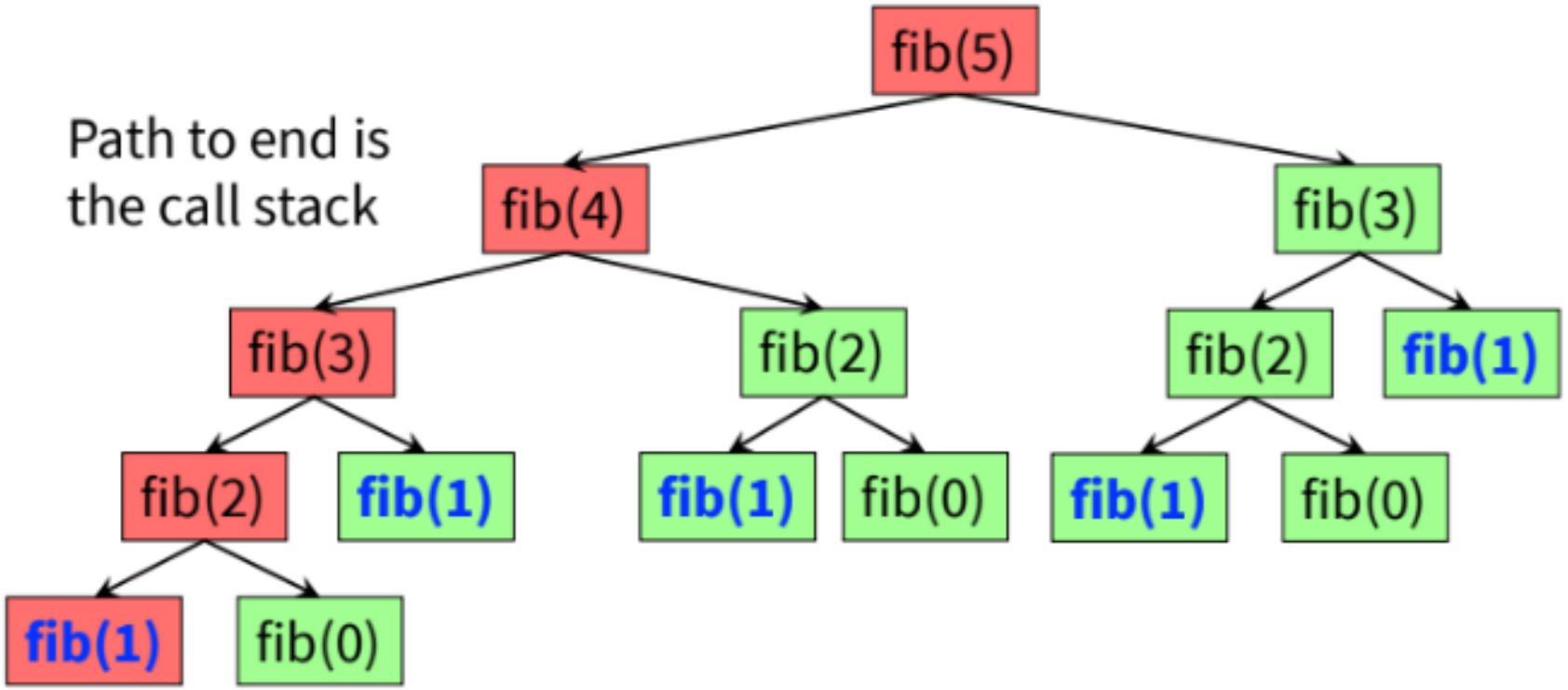
Rec. Fib.: # of Frames vs. # of Calls

`fibo_rec(n)` makes a lot of redundant calls



Rec. Fib.: # of Frames vs. # of Calls

`fibo_rec(n)` makes a lot of redundant calls



Iterative Fibonacci — Harder to Implement

```
def fibo_iter(n):
    if n == 0 or n == 1:
        return n
    else:
        a = 0
        b = 1
        for i in range(2, n+1):
            t = a
            a = b
            b = a + t
        return b
```

Quiz

```
def fibo_iter(n):
    if n == 0 or n == 1:
        return n
    else:
        a = 0
        b = 1
        for i in range(2, n+1):
                            # a = F(???), b
            t = a
            a = b
            b = a + t
            # ...
        return b
```

A: a = F(i-2),
b = F(i-1)

B: a = F(i-1),
b = F(i-2)

C: a = F(i-1),
b = F(i-1)

D: a = F(i-1),
b = F(i)

E: a = F(i),
b = F(i-1)

Iterative Fibonacci — Harder to Implement

```
def fibo_iter(n):
    if n == 0 or n == 1:
        return n
    else:
        a = 0
        b = 1
        # a = F(0), b = F(1)
        for i in range(2, n+1):
                        # a = F(i-2), b =
            t = a
            a = b
            b = a + t
            # a = F(i-1), b = F(i)
        # a = F(n-1), b = F(n)
        return b
```

Have to think carefully about loop bounds

Have to think carefully about maintaining extra variables

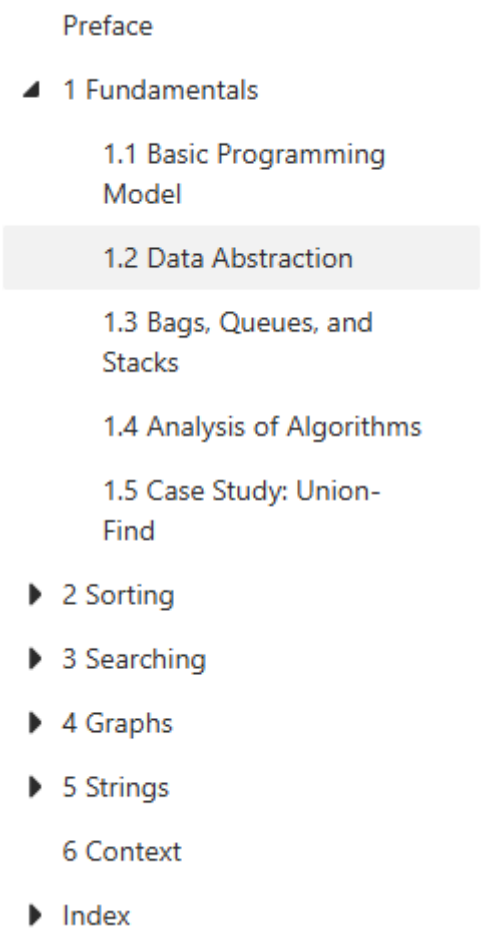
This implementation **does** have an advantage: It eliminates all the redundant calls of the recursive version.

Recursion vs. Iteration

- **Factorial**: easy to implement with both recursion and for-loops
- **Fibonacci**: easier to implement with recursion, but possible with for-loops
- Do we ever **need** recursion...?

Recursion on Nested Data

How to Store an Outline?




```
Book = [
   'Preface',
   '1 Fundamentals',
    ['1.1 Basic Programming Model',
     '1.2 Data Abstraction',
     '1.3 Bags, Queues, and Stacks',
     ['Bags',
      'Queues',
      'Stacks'],
     '1.4 Analysis of Algorithms',
     '1.5 Case Study: Union-Find'],
    '2 Sorting',
    '3 Searching',
    ...
]
```

How to Print an Outline?

```
# outline.py
def print_outline_v1(outline):
    for item in outline:
        print(item)
```

```
Preface
1 Fundamentals
['1.1 Basic Programming Model', '1.2 Data Abstraction',
'1.3 Bags, Queues, and Stacks', ['Bags', 'Queues',
'Stacks'], '1.4 Analysis of Algorithms', '1.5 Case Study:
Union-Find']
2 Sorting
3 Searching
```

How to Print an Outline?

```
def print_outline_v2(outline):
    for item in outline:
        # handle nested lists
        if isinstance(item, list):
            for subitem in item:
                print(' ' * 4 + str(subitem))
        else:
            print(item)
```

Preface
1 Fundamentals
    1.1 Basic Programming Model
    1.2 Data Abstraction
    1.3 Bags, Queues, and Stacks
    ['Bags', 'Queues', 'Stacks']
    1.4 Analysis of Algorithms
    1.5 Case Study: Union-Find
2 Sorting
3 Searching

str * n evaluates to n copies of str concatenated together
str * 2 is like str + str
str * 3 is like str + str + str
etc.

How to Print an Outline?

```
def print_outline_v3(outline):
    for item in outline:
        # handle nested lists
        if isinstance(item, list):
            for subitem in item:
                # handle nested nested lists
                if isinstance(subitem, list):
                    for subsubitem in subitem:
                        print(' ' * 8 + str(subsubitem))
                else:
                    print(' ' * 4 + str(subitem))
        else:
            print(item)
```

Preface
1 Fundamentals
- 1.1 Basic Programming Model
- 1.2 Data Abstraction
- 1.3 Bags, Queues, and Stacks
  - Bags
  - Queues
  - Stacks
- 1.4 Analysis of Algorithms
- 1.5 Case Study: Union-Find

2 Sorting
3 Searching

For-loops Do Not Suffice!

- What if we had an outline that nested yet another level? We'd need another nested for-loop!
- If an outline is nested to depth N, then we need N nested for-loops in the program
  - **Problem**: We can never write a program with enough for-loops to handle an outline of unknown nesting depth!
  - **Problem**: Even a small nesting depth (3) results in unwieldy code
- Recursion to the rescue...

How to Print an Outline!

```
def indent(level):
    return ' ' * level

def print_outline_rec(outline, level):
    for item in outline:
        if isinstance(item, list):
            print_outline_rec(item, level+1)
        else:
            print(indent(level) + item)
```

Recursive Case

Base Case

Now we can print outlines of any reasonable nesting depth with reasonable code!

Quiz

Consider this call. How many times will `print_outline_rec` be called in total, including the initial call?

```
outline = ['Intro', ['Part 1',
    ['Detail 1.1', 'Detail 1.2'], 'Part 2'], 'Conclusion']
print_outline_rec(outline, 0)
```

```
def print_outline_rec(outline, level):
    for item in outline:
        if isinstance(item, list):
            print_outline_rec(item, level+1)
        else:
            print(indent(level) + item)
```

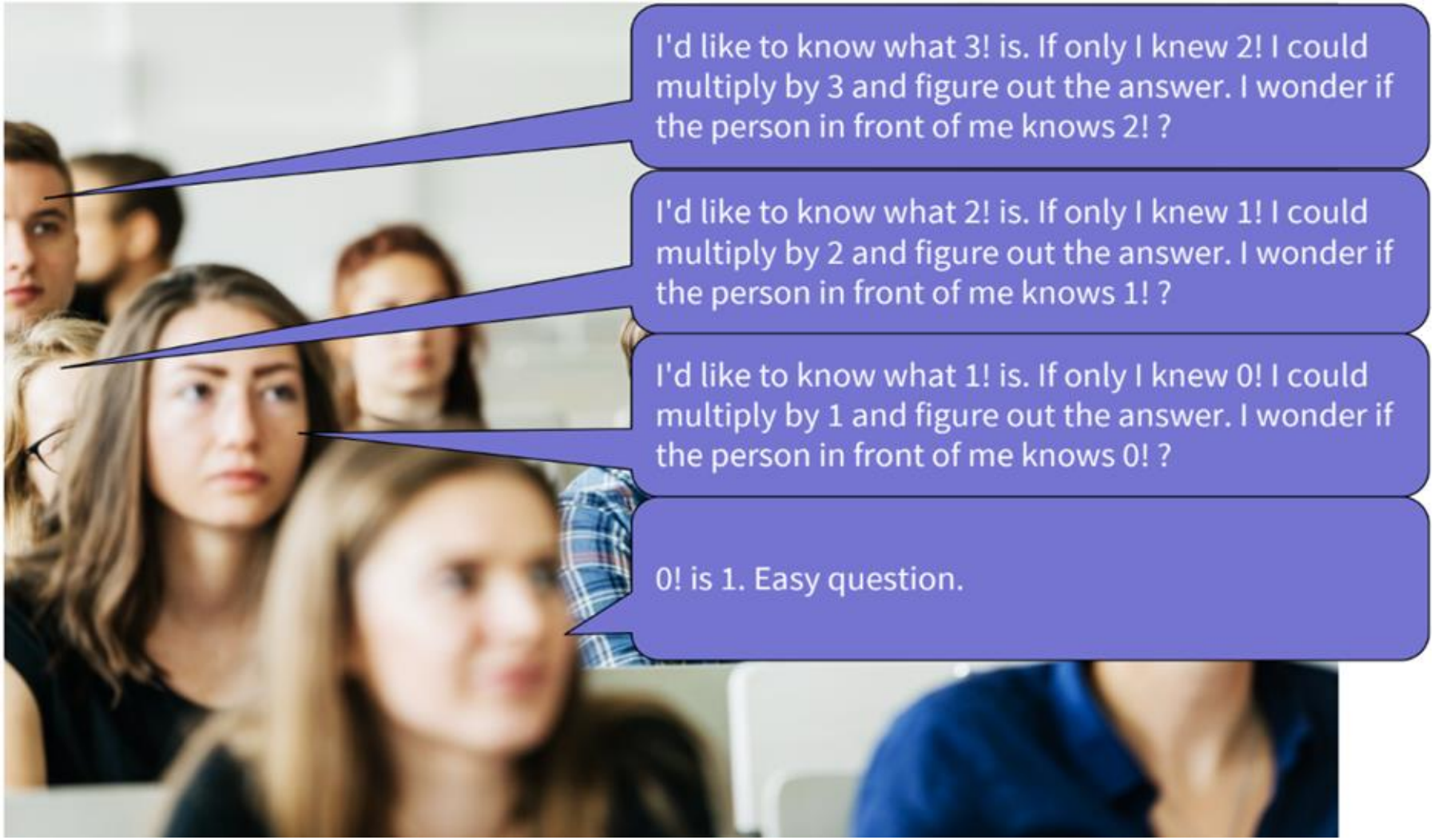
A: 1
B: 2
C: 3
D: 6
E: 7

Recursion vs. Iteration

- **Factorial**: easy to implement with both recursion and for-loops
- **Fibonacci**: easier to implement with recursion, but possible with for-loops
- **Processing nested data**: requires recursion to reach data nested at unknown depths

Recursive Programming Pattern

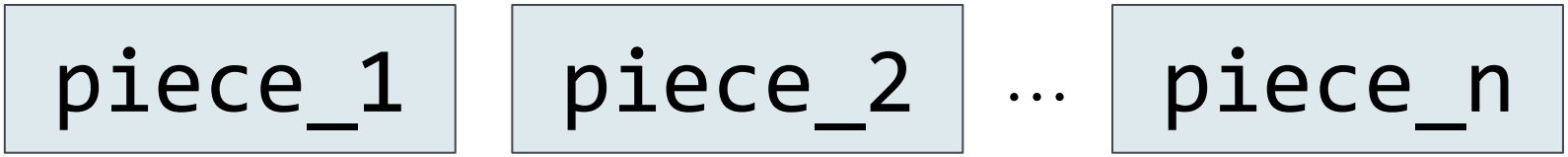
Review: An Analogy for the Call Frames



Recursive Programming Pattern

Goal: Solve problem P on a piece of data:

Pattern: Decompose data into one or more smaller pieces:



Base case: If piece is small enough the answer is easy to find.

Recursive case: If piece is big the answer requires computing P on that piece, then **recombining** with other data to get answer.

Example: Outline

Goal: Print outline from nested lists:

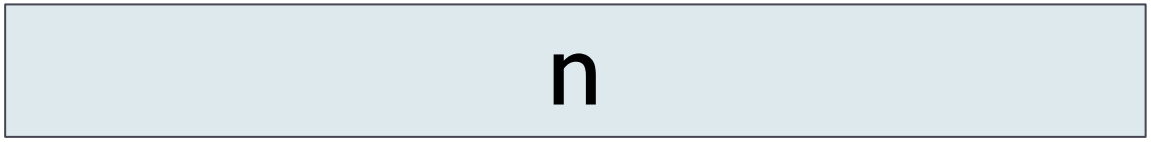
Pattern: Decompose into each of its n items

Base case: If piece is small enough — if it is a string — the line is easy to print. It's just the string, prefixed with appropriate indent.

Recursive case: If piece is big enough — if it is a list — the answer requires printing it as an outline with a greater amount of indent. The **recombination** occurs on the command line as all the output is printed.

Example: Fibonacci

Goal: Compute F(n) on data n:



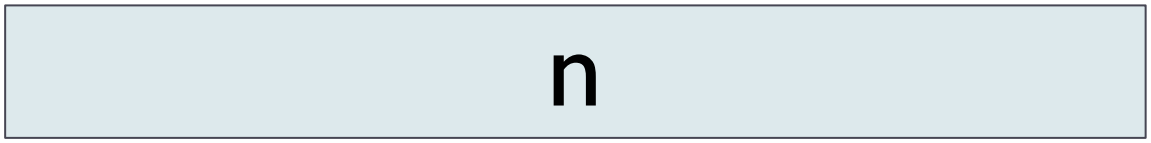
Pattern: Decompose into two smaller pieces, n-1 and n-2:

Base case: If piece is small enough — if it is 0 or 1 — the answer is easy to find. It's just 0 or 1, respectively.

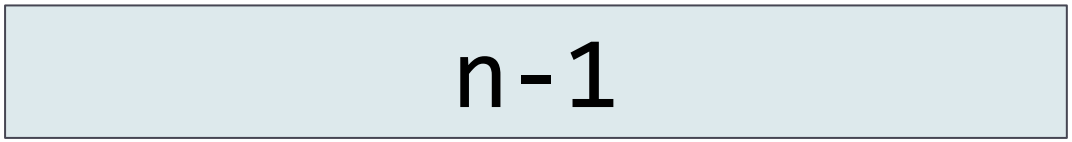
Recursive case: If piece is big enough — if it is >=2 — the answer requires computing F(n-1) and F(n-2) then **recombining** using addition: F(n-1) + F(n-2).

Example: Factorial

Goal: Compute n! on data n:



Pattern: Decompose into a smaller piece, n-1:



Base case: If n-1 is small enough — if it is 0 — the answer is easy to find. It's just 1.

Recursive case: If n-1 is big enough — if it is >0 — the answer requires computing (n-1)! then **recombining** it using multiplication: n * (n-1)!

How to Solve Problems with Recursion

- Decide how to **decompose** data
- Decide how to solve **small (base)** cases
- Decide how to **recombine big (recursive)** cases

This problem-solving technique is also called **"divide and conquer"**.