Lecture 5: Recursion Foundations

Base Cases & Call Stack Visualization

Comp 111 — Programming 2

Forman Christian University

What is Recursion?

The Infinite Mirror

The Infinite Mirror

Try Googling `"recursion"` ...

"Did you mean: recursion?"

Recursion is Everywhere!

File Folders Folders contain folders contain folders...

Recursion is Everywhere!

File Folders	Folders contain folders contain folders...
Family Tree	You have parents, who have parents...

Recursion is Everywhere!

File Folders	Folders contain folders contain folders...
Family Tree	You have parents, who have parents...
Social Media	Person shares → friends share → *their* friends share...

Recursion is Everywhere!

File Folders	Folders contain folders contain folders...
Family Tree	You have parents, who have parents...
Social Media	Person shares → friends share → *their* friends share...
GPS Directions	"Drive to the highway, then follow directions from there"

Recursion is Everywhere!

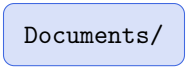
File Folders	Folders contain folders contain folders...
Family Tree	You have parents, who have parents...
Social Media	Person shares → friends share → *their* friends share...
GPS Directions	"Drive to the highway, then follow directions from there"
IKEA Assembly	"Build smaller shelf unit, attach to previous unit"

Recursion is Everywhere!

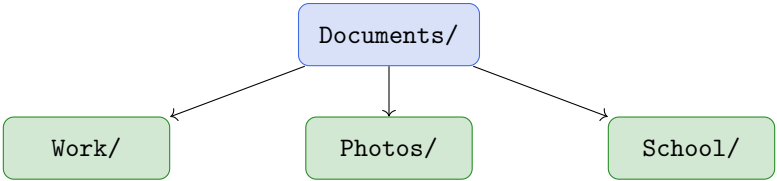
File Folders	Folders contain folders contain folders...
Family Tree	You have parents, who have parents...
Social Media	Person shares → friends share → *their* friends share...
GPS Directions	"Drive to the highway, then follow directions from there"
IKEA Assembly	"Build smaller shelf unit, attach to previous unit"

Recursion is everywhere. Today we learn to harness it in code.

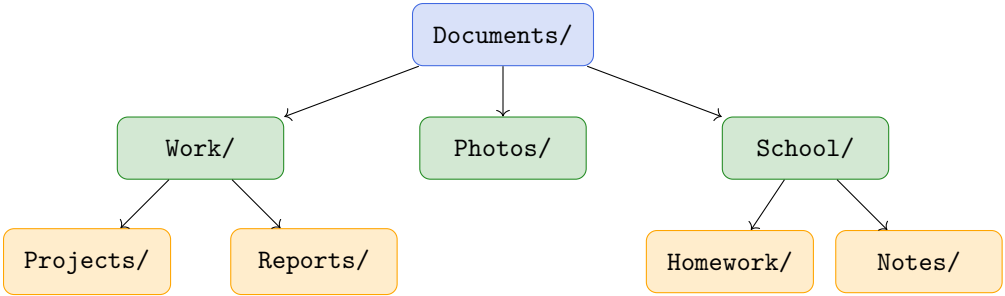
Example: File Folders



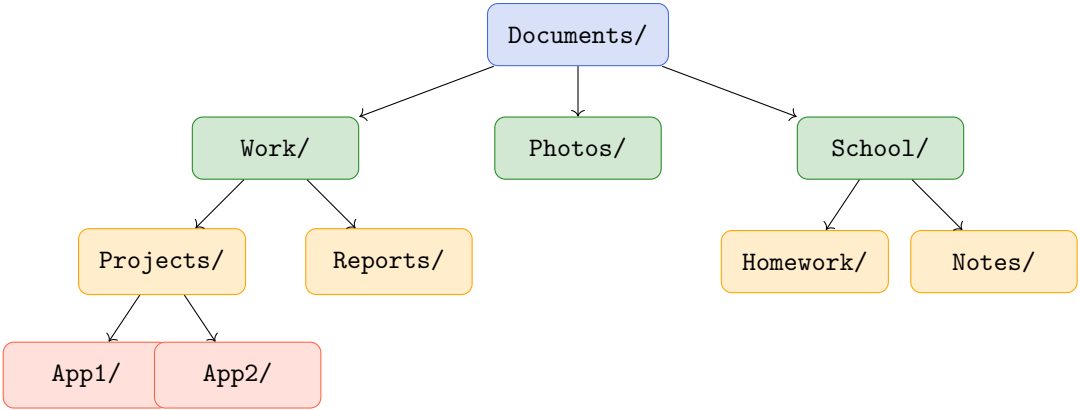
Example: File Folders



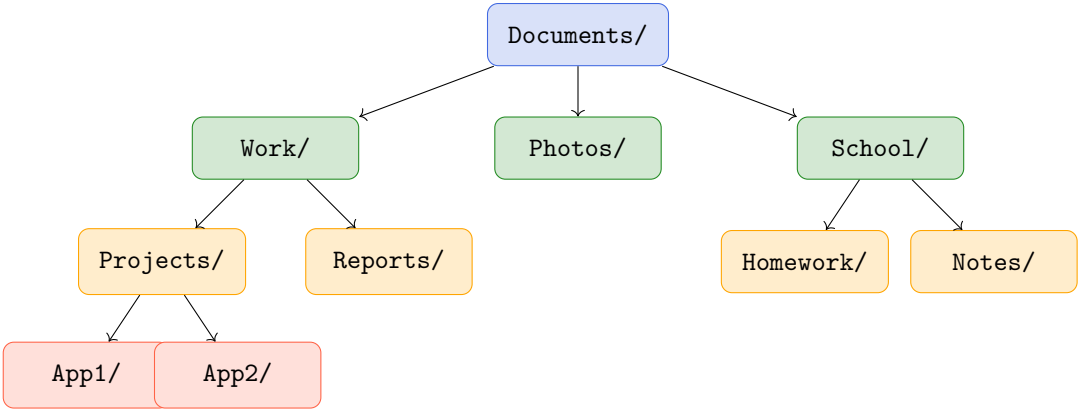
Example: File Folders



Example: File Folders



Example: File Folders



How do you count all files?

Count files here + count files in *each subfolder*

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

1. Count files directly in Documents/ (say, 3 files)

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

1. Count files directly in Documents/ (say, 3 files)
2. Count files in Work/ ← same problem, smaller!

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

1. Count files directly in Documents/ (say, 3 files)
2. Count files in Work/ ← same problem, smaller!
3. Count files in Photos/ ← same problem, smaller!

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

1. Count files directly in Documents/ (say, 3 files)
2. Count files in Work/ ← same problem, smaller!
3. Count files in Photos/ ← same problem, smaller!
4. Count files in School/ ← same problem, smaller!

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

1. Count files directly in Documents/ (say, 3 files)
2. Count files in Work/ ← same problem, smaller!
3. Count files in Photos/ ← same problem, smaller!
4. Count files in School/ ← same problem, smaller!
5. Add them all up!

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

1. Count files directly in Documents/ (say, 3 files)
2. Count files in Work/ ← same problem, smaller!
3. Count files in Photos/ ← same problem, smaller!
4. Count files in School/ ← same problem, smaller!
5. Add them all up!

Base case: Empty folder → return 0

Example: Counting Files

Task: Count all files in "Documents/"

To count files in Documents/:

1. Count files directly in Documents/ (say, 3 files)
2. Count files in Work/ ← same problem, smaller!
3. Count files in Photos/ ← same problem, smaller!
4. Count files in School/ ← same problem, smaller!
5. Add them all up!

Base case: Empty folder → return 0

The problem "looks like itself" at each level!

What is Recursion?

Recursion is when a function **calls itself** to solve a **smaller version** of the same problem.

What is Recursion?

Recursion is when a function **calls itself** to solve a **smaller version** of the same problem.

Key Insight: A problem that *"looks like itself"*

The Two Magic Ingredients

1. BASE CASE

When to **STOP**

(The answer is obvious)

The Two Magic Ingredients

1. BASE CASE
When to **STOP**
(The answer is obvious)

+

2. RECURSIVE CASE
Break into a **SMALLER**
version of the problem

The Two Magic Ingredients

1. BASE CASE
When to **STOP**
(The answer is obvious)

+

2. RECURSIVE CASE
Break into a **SMALLER**
version of the problem

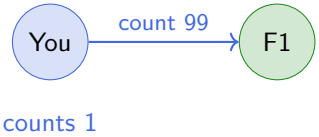
Every recursive solution needs **both**!

The Lazy Helper Analogy

"Count 100 sheep for me!"

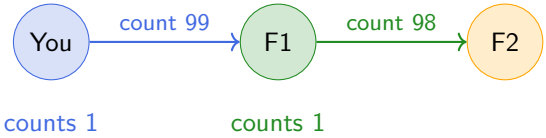
The Lazy Helper Analogy

"Count 100 sheep for me!"



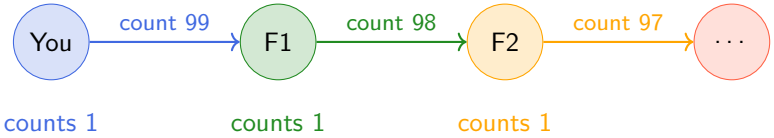
The Lazy Helper Analogy

"Count 100 sheep for me!"



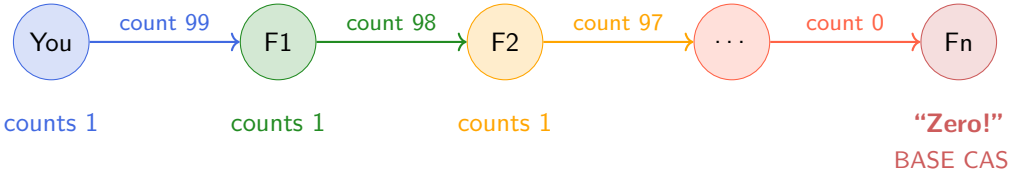
The Lazy Helper Analogy

"Count 100 sheep for me!"



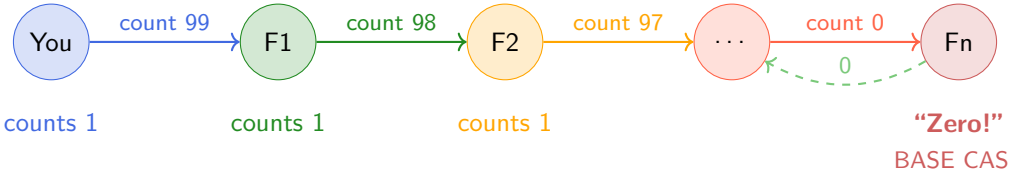
The Lazy Helper Analogy

"Count 100 sheep for me!"



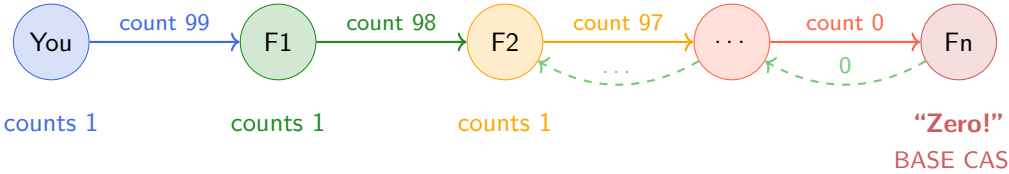
The Lazy Helper Analogy

"Count 100 sheep for me!"



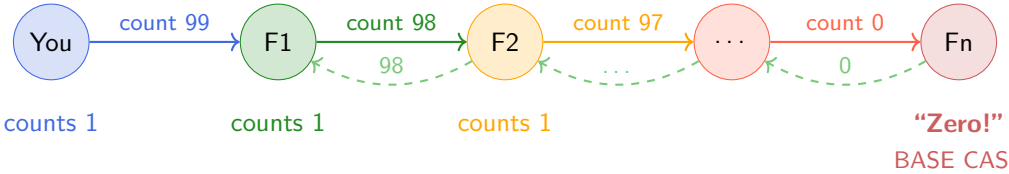
The Lazy Helper Analogy

"Count 100 sheep for me!"



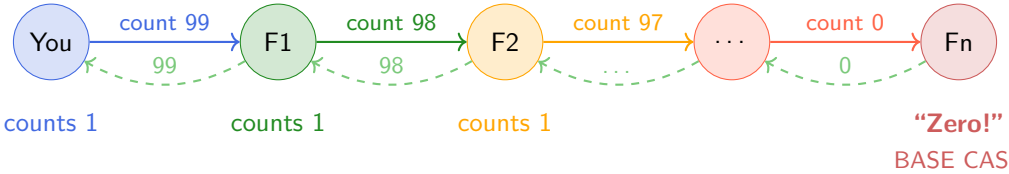
The Lazy Helper Analogy

"Count 100 sheep for me!"



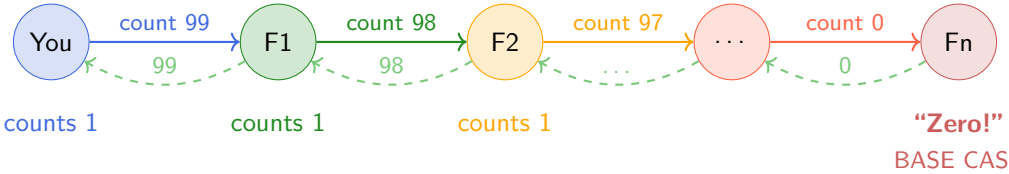
The Lazy Helper Analogy

"Count 100 sheep for me!"



The Lazy Helper Analogy

"Count 100 sheep for me!"



Answers bubble back up!

First Example: Countdown

```
1  def countdown(n):
2      # Base case
3      if n <= 0:
4          print("Blastoff!")
5          return
6
7      # Recursive case
8      print(n)
9      countdown(n - 1)
10
11 countdown(5)
```

Output:

```
5
4
3
2
1
Blastoff!
```

First Example: Countdown

```
1  def countdown(n):
2      # Base case
3      if n <= 0:
4          print("Blastoff!")
5          return
6
7      # Recursive case
8      print(n)
9      countdown(n - 1)
10
11 countdown(5)
```

Output:

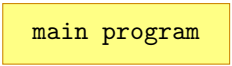
```
5
4
3
2
1
Blastoff!
```

The Call Stack

What is a Call Stack?

Analogy: Stack of Sticky Notes

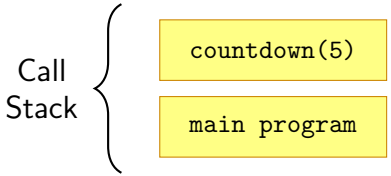
- Each function call = new sticky note



What is a Call Stack?

Analogy: Stack of Sticky Notes

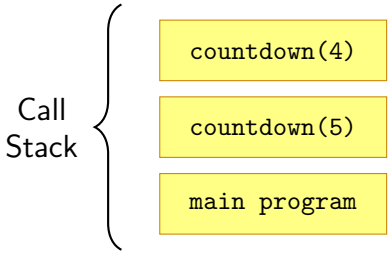
- Each function call = new sticky note
- Put it **on top** of the pile



What is a Call Stack?

Analogy: Stack of Sticky Notes

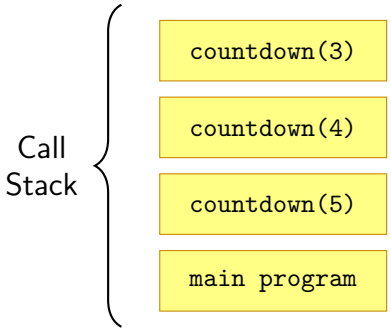
- Each function call = new sticky note
- Put it **on top** of the pile
- When function finishes, **remove** the note



What is a Call Stack?

Analogy: Stack of Sticky Notes

- Each function call = new sticky note
- Put it **on top** of the pile
- When function finishes, **remove** the note
- Go back to what's underneath

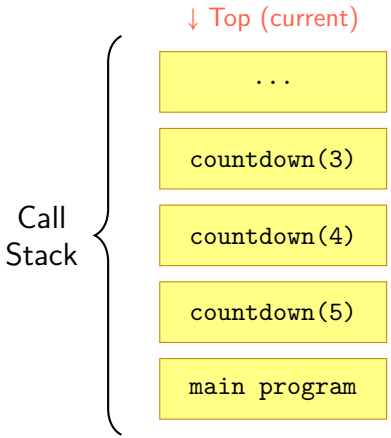


What is a Call Stack?

Analogy: Stack of Sticky Notes

- Each function call = new sticky note
- Put it **on top** of the pile
- When function finishes, **remove** the note
- Go back to what's underneath

LIFO: Last In, First Out



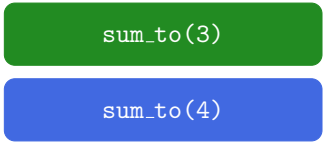
Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```

sum_to(4)

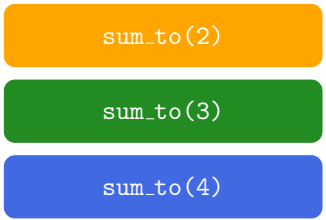
Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```



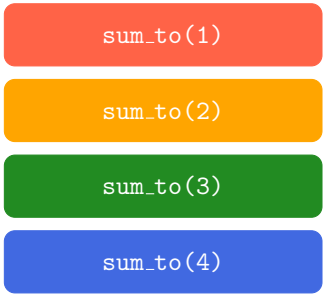
Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```



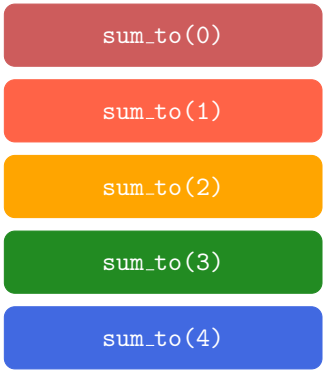
Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```



Tracing: Sum of Numbers

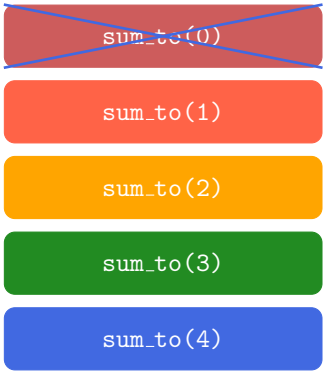
```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```



BASE!
returns 0

Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```

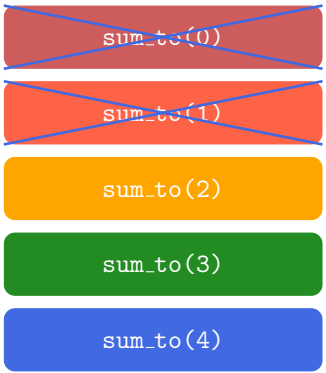


BASE!
returns 0

returns 1+0=1

Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```



BASE!
returns 0

returns 1+0=1

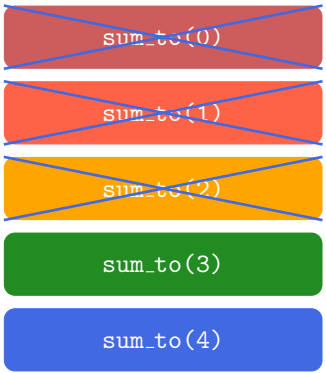
returns 2+1=3

Mathematical:

$sum\_to(n) = n + (n-1) + \cdots + 1 + 0$

Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```

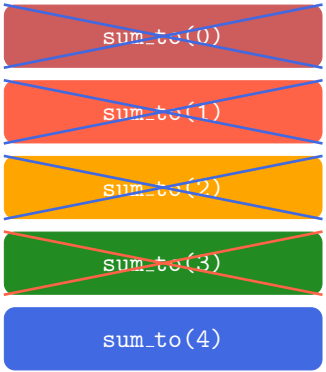


Mathematical:

$sum\_to(n) = n + (n-1) + \cdots + 1 + 0$

Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```

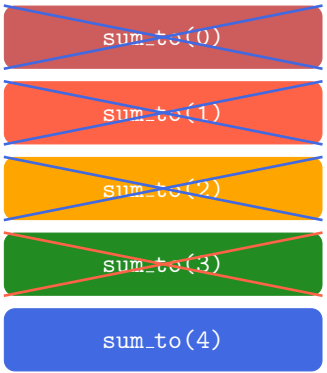


Mathematical:

$sum\_to(n) = n + (n-1) + \cdots + 1 + 0$

Tracing: Sum of Numbers

```
1  def sum_to(n):
2      # Base case
3      if n <= 0:
4          return 0
5
6      # Recursive case
7      return n + sum_to(n-1)
8
9  result = sum_to(4)
```



BASE!
returns 0

returns 1+0=1

returns 2+1=3

returns 3+3=6

returns 4+6=10

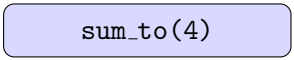
Mathematical:

$sum\_to(n) = n + (n-1) + \cdots + 1 + 0$

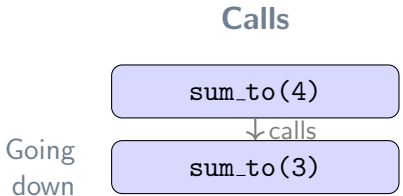
result = 10

Visualizing the Trace

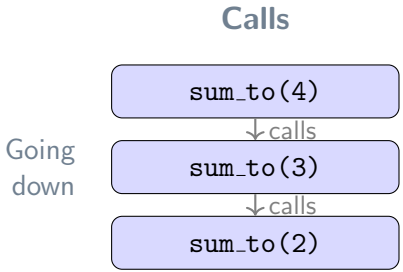
Calls



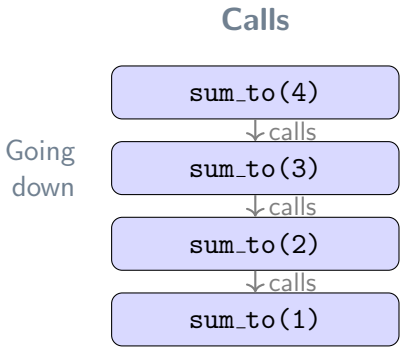
Visualizing the Trace



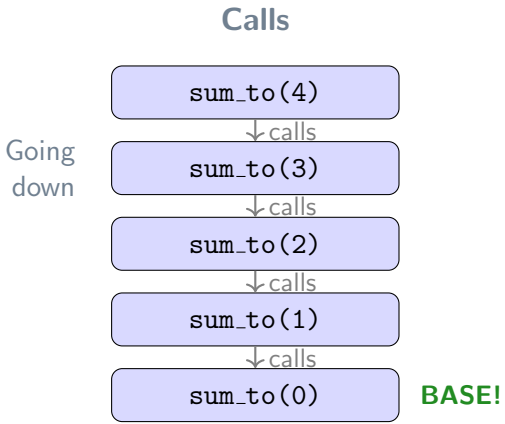
Visualizing the Trace



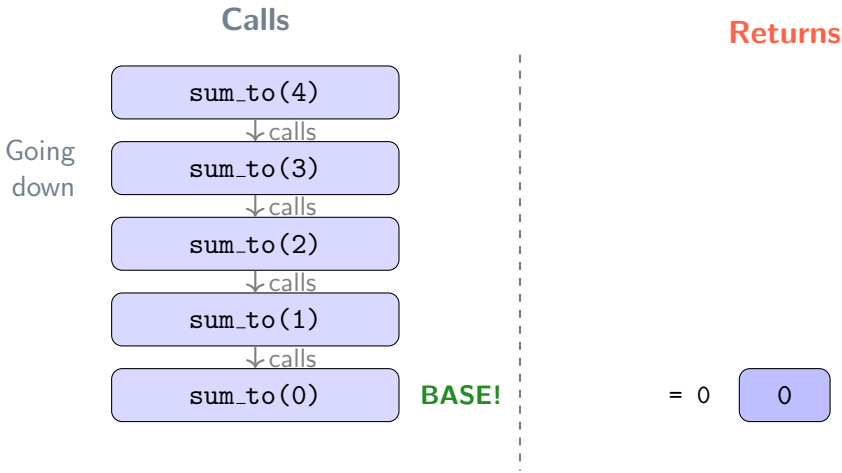
Visualizing the Trace



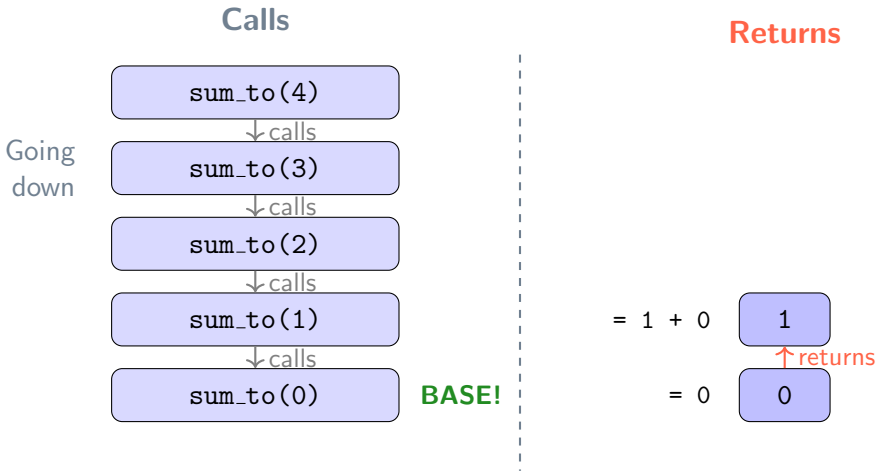
Visualizing the Trace



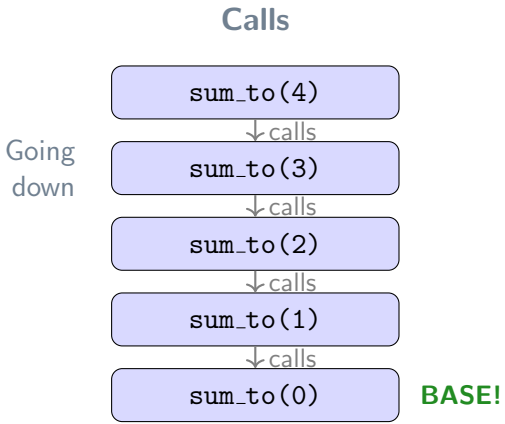
Visualizing the Trace



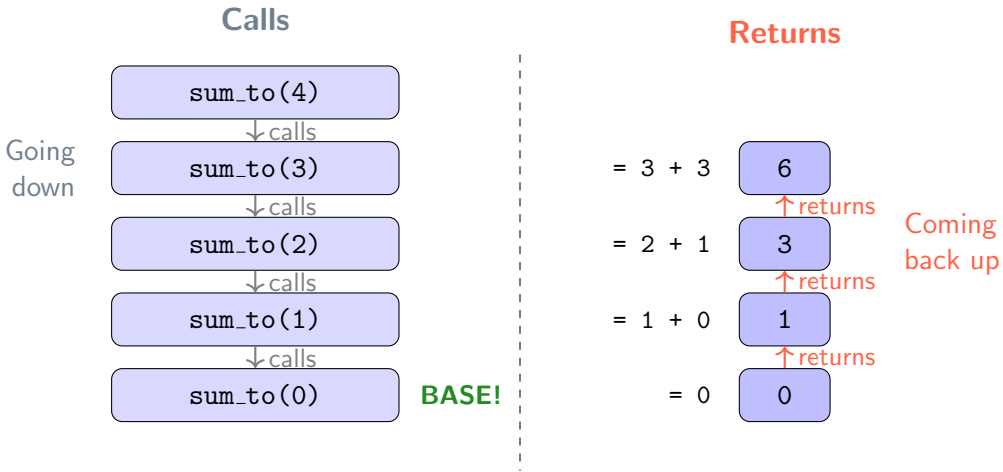
Visualizing the Trace



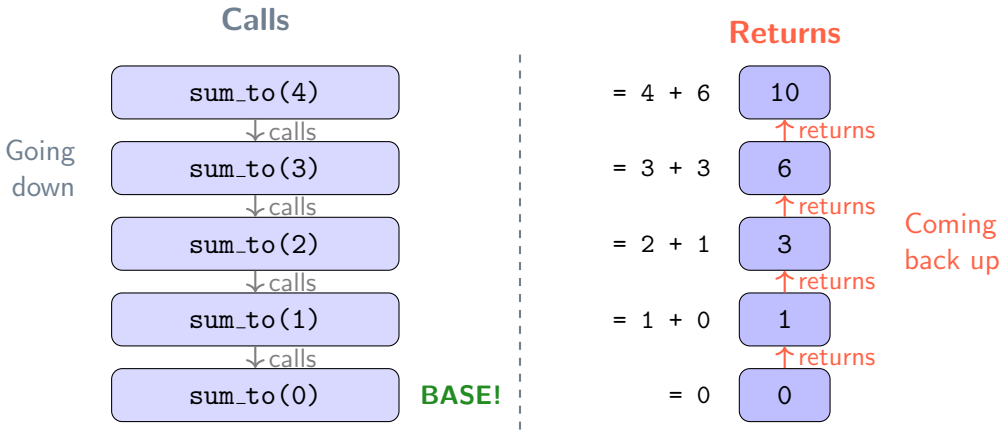
Visualizing the Trace



Visualizing the Trace



Visualizing the Trace



Why `n <= 0` and not `n == 1`?

Using `n <= 0` means our function handles:

Why `n <= 0` and not `n == 1`?

Using `n <= 0` means our function handles:

- `sum_to(0)` → returns **0** ✓

Why `n <= 0` and not `n == 1`?

Using `n <= 0` means our function handles:

- `sum_to(0)` → returns **0** ✓
- `sum_to(-5)` → returns **0** ✓

Why `n <= 0` and not `n == 1`?

Using `n <= 0` means our function handles:

- `sum_to(0)` → returns **0** ✓
- `sum_to(-5)` → returns **0** ✓
- Any edge case gracefully! ✓

Why `n <= 0` and not `n == 1`?

Using `n <= 0` means our function handles:

- `sum_to(0)` → returns **0** ✓
- `sum_to(-5)` → returns **0** ✓
- Any edge case gracefully! ✓

With `n == 1`: what happens if someone calls `sum_to(0)`?
It never hits the base case → RecursionError!

Why `n <= 0` and not `n == 1`?

Using `n <= 0` means our function handles:

- `sum_to(0)` → returns **0** ✓
- `sum_to(-5)` → returns **0** ✓
- Any edge case gracefully! ✓

With `n == 1`: what happens if someone calls `sum_to(0)`?
It never hits the base case → RecursionError!

Always ask: "What weird inputs could someone pass to my function?"

Python Tutor Demo

Let's visualize this live!

`https://pythontutor.com`

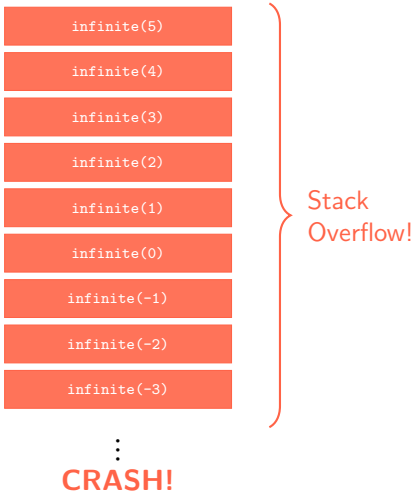
The Critical Base Case

What If We Forget the Base Case?

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4     # No base case!
5
6 infinite(5)
```

Output:

```
5
4
3
...
-993
-994
RecursionError!
```



Why Does Python Stop?

- Every function call uses **memory**
- Python has a **recursion limit** (~1000 calls)
- This **protects** your computer from crashing!
- Error message: `RecursionError: maximum recursion depth exceeded`

Why Does Python Stop?

- Every function call uses **memory**
- Python has a **recursion limit** (~1000 calls)
- This **protects** your computer from crashing!
- Error message: `RecursionError: maximum recursion depth exceeded`

Lesson: Always have a base case that will be reached!

Why Does Python Stop?

- Every function call uses **memory**
- Python has a **recursion limit** (~1000 calls)
- This **protects** your computer from crashing!
- Error message: `RecursionError: maximum recursion depth exceeded`

Lesson: Always have a base case that will be reached!

Practical tip: If your solution is correct but the input is too large, that's a signal to rewrite using a loop. We'll compare next lecture.

Common Base Case Mistakes

Forgetting base case
No `if` statement

→ Infinite recursion

Common Base Case Mistakes

Forgetting base case
No `if` statement

→ Infinite recursion

Wrong condition
`if n == 0` when `n` is negative

→ Never reaches base

Common Base Case Mistakes

Forgetting base case
No `if` statement

→ Infinite recursion

Wrong condition
`if n == 0` when `n` is negative

→ Never reaches base

No return value

```
if n == 1:  print("done")
```

→ Returns `None`

Common Base Case Mistakes

Forgetting base case
No `if` statement

→ Infinite recursion

Wrong condition
`if n == 0` when `n` is negative

→ Never reaches base

No return value
```
if n == 1:  print("done")
```

→ Returns `None`

Not getting smaller
```
return n + f(n)
```

→ Same size forever

You Try: Spot the Bug!

What's wrong with each function?

Problem 1:

```
1  def factorial(n):
2      return n * factorial(n-1)
```

Problem 2: What does `countdown(3)` print?

```
1  def countdown(n):
2      if n == 0:
3          return
4      countdown(n - 1)
5      print(n)
```

You Try: Spot the Bug!

What's wrong with each function?

Problem 1:

```
1  def factorial(n):
2      return n * factorial(n-1)
```

Bug: No base case!
Fix: Add `if n <= 1:  return 1`

Problem 2: What does `countdown(3)` print?

```
1  def countdown(n):
2      if n == 0:
3          return
4      countdown(n - 1)
5      print(n)
```

You Try: Spot the Bug!

What's wrong with each function?

Problem 1:

```
1 def factorial(n):
2     return n * factorial(n-1)
```

Bug: No base case!
Fix: Add `if n <= 1:  return 1`

Problem 2: What does `countdown(3)` print?

```
1 def countdown(n):
2     if n == 0:
3         return
4     countdown(n - 1)
5     print(n)
```

Surprise: Prints `1 2 3` (ascending!)
Why? `print` happens *after* the recursive call returns.

Guided Practice

Example: Factorial

Mathematical Definition:

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $n! = n \times (n-1)!$
- $0! = 1$ and $1! = 1$ (base cases)

```
1  def factorial(n):
2      # Base case
3      if n <= 1:
4          return 1
5      # Recursive case
6      return n * factorial(n-1)
7
8  print(factorial(5))  # 120
9  print(factorial(0))  # 1
```

Example: Factorial

Mathematical Definition:

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $n! = n \times (n-1)!$
- $0! = 1$ and $1! = 1$ (base cases)

```
1  def factorial(n):
2      # Base case
3      if n <= 1:
4          return 1
5      # Recursive case
6      return n * factorial(n-1)
7
8  print(factorial(5))  # 120
9  print(factorial(0))  # 1
```

```
factorial(5)
  = 5 * factorial(4)
    = 5 * 4 * factorial(3)
      = 5 * 4 * 3 * factorial(2)
        = 5 * 4 * 3 * 2 * 1

= 120
```

Example: String Length

How would you count characters **without** `len()`?

Recursive Thinking:

- Empty string `""` has length **0** (base case)

Example: String Length

How would you count characters **without** `len()`?

Recursive Thinking:

- Empty string `""` has length **0** (base case)
- Any other string: **1** + length of the rest

Example: String Length

How would you count characters **without** `len()`?

Recursive Thinking:

- Empty string `""` has length **0** (base case)
- Any other string: **1** + length of the rest

```
1  def string_length(s):
2      # Base case: empty string
3      if s == "":
4          return 0
5      # Recursive case
6      return 1 + string_length(s[1:])
7
8  print(string_length("hello"))
```

Example: String Length

How would you count characters **without** `len()`?

Recursive Thinking:

- Empty string `""` has length **0** (base case)
- Any other string: **1** + length of the rest

```
1  def string_length(s):
2      # Base case: empty string
3      if s == "":
4          return 0
5      # Recursive case
6      return 1 + string_length(s[1:])
7
8  print(string_length("hello"))
```

```
string_length("hello")
 = 1+ str_len("ello")
  = 1+1 + str_len("llo")
   = 1+1+1 + str_len("lo")
    = 1+1+1+1 + str_len("o")
     = 1+1+1+1+1 + 0

= 5
```

You Try: Multiply Using Addition

Write a recursive function to multiply two positive integers **using only addition**.

Hint: $3 \times 4 = 3 + 3 + 3 + 3 = 3 + (3 \times 3)$

You Try: Multiply Using Addition

Write a recursive function to multiply two positive integers **using only addition**.

Hint: $3 \times 4 = 3 + 3 + 3 + 3 = 3 + (3 \times 3)$

```
1  def multiply(a, b):
2      # Base case: anything times 0 is 0
3      if b == 0:
4          return 0
5      # Recursive case: a * b = a + a * (b-1)
6      return a + multiply(a, b - 1)
7
8  print(multiply(3, 4))  # 12
```

Summary

Recursion Checklist

✓ Define your **BASE CASE** (when to stop)

Recursion Checklist

✓ Define your **BASE CASE** (when to stop)

✓ Define your **RECURSIVE CASE** (smaller subproblem)

Recursion Checklist

✓ Define your **BASE CASE** (when to stop)

✓ Define your **RECURSIVE CASE** (smaller subproblem)

✓ Make sure you're moving **TOWARD** the base case

Recursion Checklist

- ✓ Define your **BASE CASE** (when to stop)
- ✓ Define your **RECURSIVE CASE** (smaller subproblem)
- ✓ Make sure you're moving **TOWARD** the base case
- ✓ Handle **edge cases** (0, negatives, empty inputs)

Recursion Checklist

- ✓ Define your **BASE CASE** (when to stop)
- ✓ Define your **RECURSIVE CASE** (smaller subproblem)
- ✓ Make sure you're moving **TOWARD** the base case
- ✓ Handle **edge cases** (0, negatives, empty inputs)
- ✓ **Trust** that the recursive call will work!

The Recursive Leap of Faith

When writing recursive code:

Trust that your function will work for smaller inputs.

If your base case is correct and you're making the problem smaller, **it WILL work**.

The Recursive Leap of Faith

When writing recursive code:

Trust that your function will work for smaller inputs.

If your base case is correct and you're making the problem smaller, **it WILL work**.

Don't try to trace every call in your head!

Do trust the recursive structure.

Key Takeaways

- **Recursion** = function calling itself on smaller problem

Key Takeaways

- **Recursion** = function calling itself on smaller problem
- Two essential parts: **Base case** + **Recursive case**

Key Takeaways

- **Recursion** = function calling itself on smaller problem
- Two essential parts: **Base case** + **Recursive case**
- **Call stack** tracks each function call
  - Builds up as we recurse down
  - Unwinds as answers bubble up

Key Takeaways

- **Recursion** = function calling itself on smaller problem
- Two essential parts: **Base case** + **Recursive case**
- **Call stack** tracks each function call
  - Builds up as we recurse down
  - Unwinds as answers bubble up
- Without base case → `RecursionError`

Key Takeaways

- **Recursion** = function calling itself on smaller problem
- Two essential parts: **Base case** + **Recursive case**
- **Call stack** tracks each function call
  - Builds up as we recurse down
  - Unwinds as answers bubble up
- Without base case → `RecursionError`
- Use **Python Tutor** to visualize!

Next Lecture Preview

Recursion Practice & Debugging

- More practice problems
- Debugging recursive functions
- When to use recursion vs. loops

Next Lecture Preview

Recursion Practice & Debugging

- More practice problems
- Debugging recursive functions
- When to use recursion vs. loops

Homework:

1. Trace `factorial(4)` on paper (draw the call stack!)
2. Write recursive `power(base, exp)` $\rightarrow 2^3 = 8$
3. Write recursive `count_down_to_zero(n)` that prints `n` down to 0
4. **Bonus:** Write recursive `reverse_string(s)`

Questions?

Practice on `pythontutor.com`