Lecture 21: Recursion and the Call Stack

Comp 102

Fakhir Shaheen

Forman Christian University

Stack Overflow

A Website Every Programmer Knows

- Stuck on a bug? Programmers ask on a website.



A Website Every Programmer Knows

- Stuck on a bug? Programmers ask on a website.
- Its name: **Stack Overflow**.



A Website Every Programmer Knows

- Stuck on a bug? Programmers ask on a website.
- Its name: **Stack Overflow**.
- Millions of programmers use it every day.



A Website Every Programmer Knows

- Stuck on a bug? Programmers ask on a website.
- Its name: **Stack Overflow**.
- Millions of programmers use it every day.



Why that name? What is a **stack**, and how does it **overflow**?

Write a guess on your paper (30 s). We answer it today.

Pictures Inside Pictures



Two mirrors face each other.

Pictures Inside Pictures



Two mirrors face each other.



A cocoa tin (1904) with a picture of the **same tin** on it.

Pictures Inside Pictures



Two mirrors face each other.



A cocoa tin (1904) with a picture of the **same tin** on it.

Each picture holds a **smaller copy of itself**.

Today

1. A function that **calls itself**.

Today

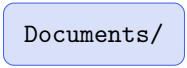
1. A function that **calls itself**.
2. The **stack** of frames Python keeps for every call.

Today

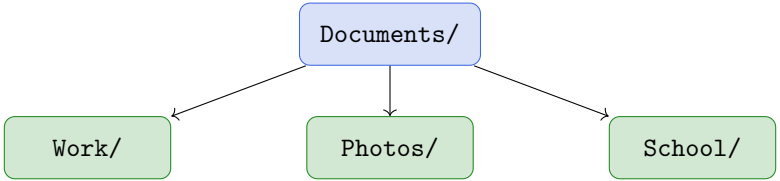
1. A function that **calls itself**.
2. The **stack** of frames Python keeps for every call.
3. The day the frames **run out**... and the name of the website.

Folders Inside Folders

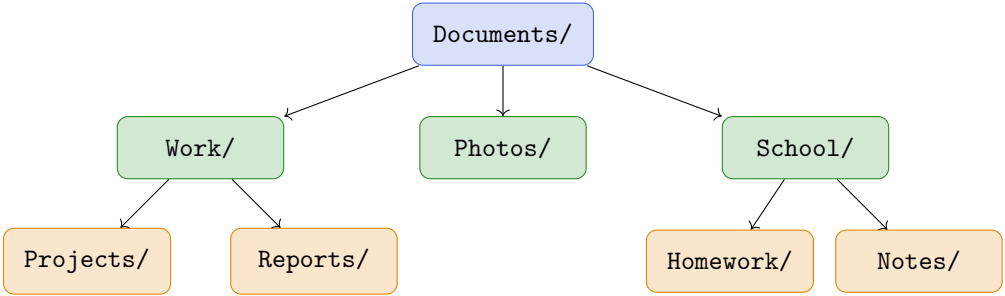
Folders Inside Folders



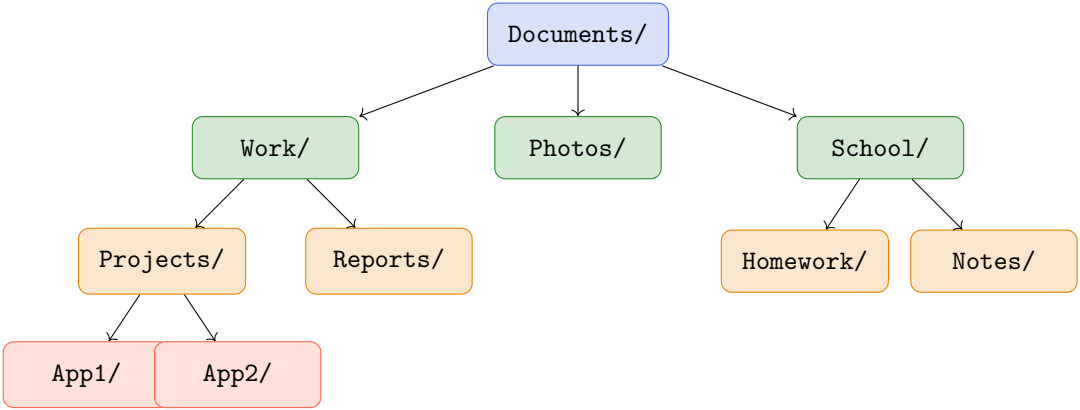
Folders Inside Folders



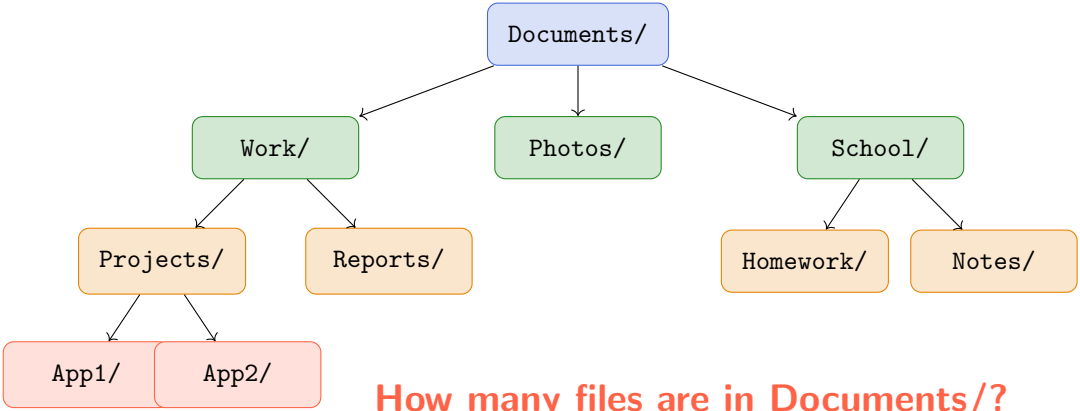
Folders Inside Folders



Folders Inside Folders



Folders Inside Folders



How many files are in Documents/?

We do not know how deep the folders go.

Counting the Files

To count the files in Documents/:

Counting the Files

To count the files in Documents/:

1. Count the files right here (say, 3).

Counting the Files

To count the files in Documents/:

1. Count the files right here (say, 3).
2. Count the files in Work/ ← same job, smaller!

Counting the Files

To count the files in Documents/:

1. Count the files right here (say, 3).
2. Count the files in Work/ ← same job, smaller!
3. Count the files in Photos/ ← same job, smaller!

Counting the Files

To count the files in Documents/:

1. Count the files right here (say, 3).
2. Count the files in Work/ ← same job, smaller!
3. Count the files in Photos/ ← same job, smaller!
4. Count the files in School/ ← same job, smaller!

Counting the Files

To count the files in Documents/:

1. Count the files right here (say, 3).
2. Count the files in Work/ ← same job, smaller!
3. Count the files in Photos/ ← same job, smaller!
4. Count the files in School/ ← same job, smaller!
5. Add them all up.

Counting the Files

To count the files in Documents/:

1. Count the files right here (say, 3).
2. Count the files in Work/ ← same job, smaller!
3. Count the files in Photos/ ← same job, smaller!
4. Count the files in School/ ← same job, smaller!
5. Add them all up.

Stop: a folder with no folders inside. Just count its files.

Counting the Files

To count the files in Documents/:

1. Count the files right here (say, 3).
2. Count the files in Work/ ← same job, smaller!
3. Count the files in Photos/ ← same job, smaller!
4. Count the files in School/ ← same job, smaller!
5. Add them all up.

Stop: a folder with no folders inside. Just count its files.

The job contains a smaller copy of the same job.

What is Recursion?

Recursion: a function **calls itself**

to do a **smaller copy** of the same job.

What is Recursion?

Recursion: a function **calls itself**

to do a **smaller copy** of the same job.

Like the cocoa tin: the picture holds a smaller copy of itself.

Two Parts, Always

1. BASE CASE

When to **STOP**

(the answer is easy)

Two Parts, Always

1. BASE CASE
When to **STOP**
(the answer is easy)

+

2. RECURSIVE CASE
Call yourself on a
SMALLER job

Two Parts, Always

1. BASE CASE
When to **STOP**
(the answer is easy)

+

2. RECURSIVE CASE
Call yourself on a
SMALLER job

Every recursive function needs **both**.

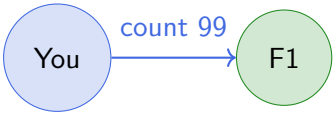
The Lazy Helpers

"Count these 100 sheep for me!"



The Lazy Helpers

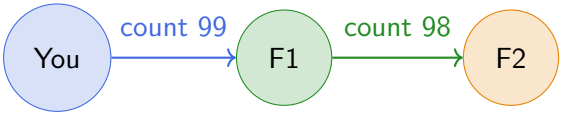
"Count these 100 sheep for me!"



takes 1 sheep

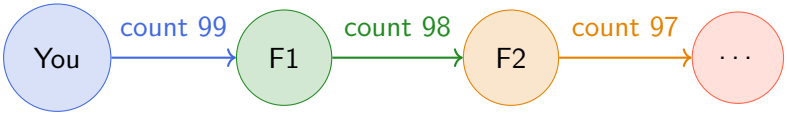
The Lazy Helpers

"Count these 100 sheep for me!"



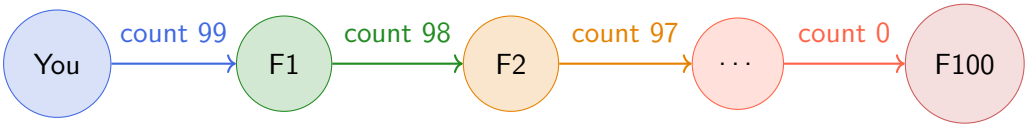
The Lazy Helpers

"Count these 100 sheep for me!"



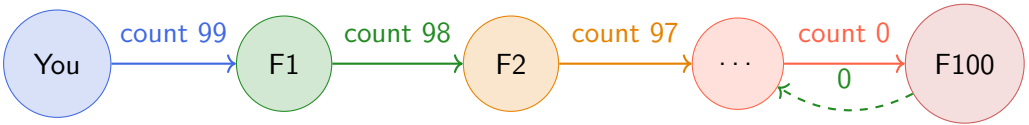
The Lazy Helpers

"Count these 100 sheep for me!"



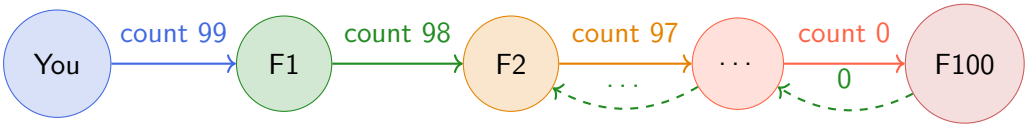
The Lazy Helpers

"Count these 100 sheep for me!"



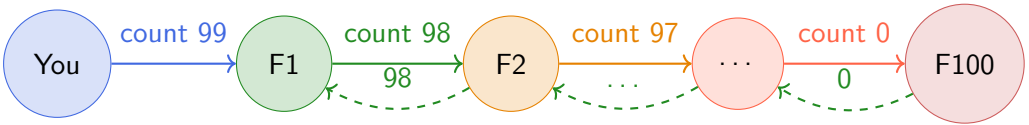
The Lazy Helpers

"Count these 100 sheep for me!"



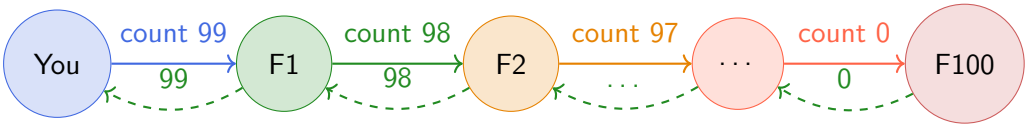
The Lazy Helpers

"Count these 100 sheep for me!"



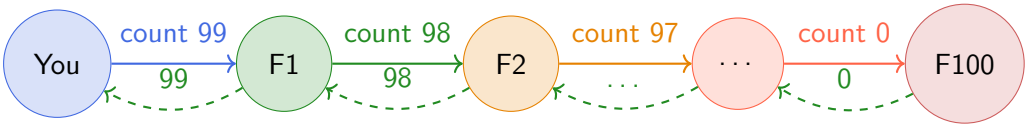
The Lazy Helpers

"Count these 100 sheep for me!"



The Lazy Helpers

"Count these 100 sheep for me!"



The answers come back up: 100!

First Example: Countdown

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

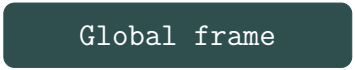
Output:

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

The Call Stack

Every Call Gets a Frame

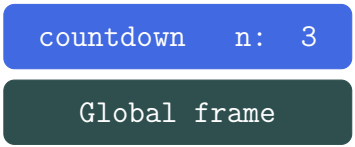
- Lecture 10: every function call makes a **new frame**.



Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.

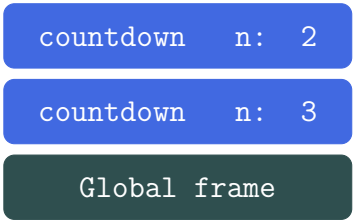
```
countdown(3)
```



Every Call Gets a Frame

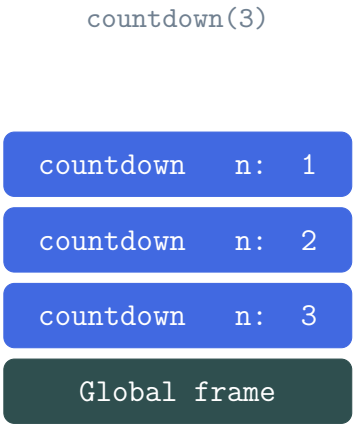
- Lecture 10: every function call makes a **new frame**.

`countdown(3)`



Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.



Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.



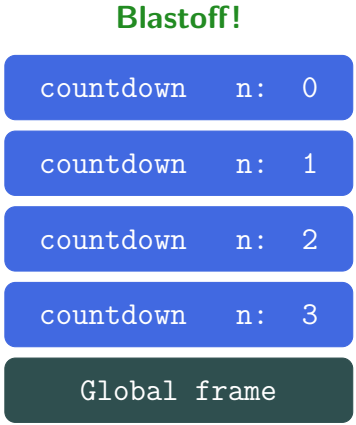
Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.
- Each frame has its **own** `n`.



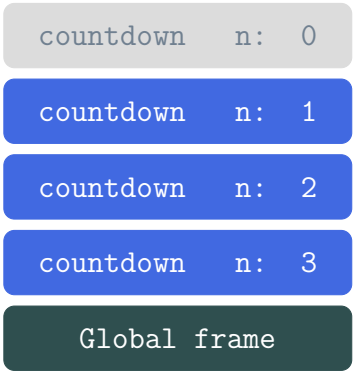
Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.
- Each frame has its **own** `n`.
- The newest frame is on **top**.



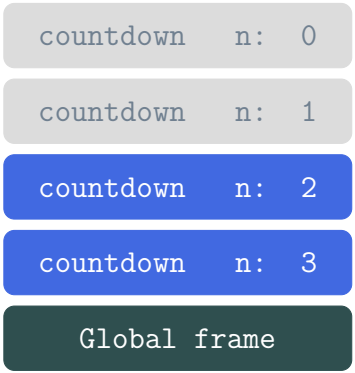
Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.
- Each frame has its **own** `n`.
- The newest frame is on **top**.
- When a call ends, its frame is **removed**.



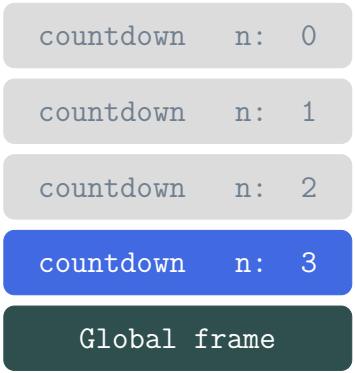
Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.
- Each frame has its **own** `n`.
- The newest frame is on **top**.
- When a call ends, its frame is **removed**.



Every Call Gets a Frame

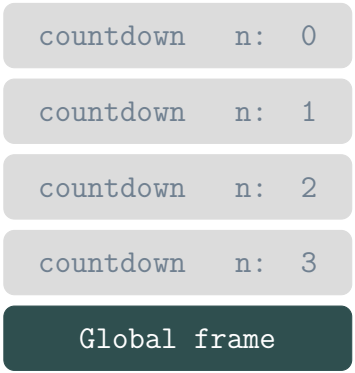
- Lecture 10: every function call makes a **new frame**.
- Each frame has its **own** `n`.
- The newest frame is on **top**.
- When a call ends, its frame is **removed**.



Every Call Gets a Frame

- Lecture 10: every function call makes a **new frame**.
- Each frame has its **own** `n`.
- The newest frame is on **top**.
- When a call ends, its frame is **removed**.

A pile like this is a **stack**: the last one on is the first one off, like plates.



Tracing sum_to(4): Going Down

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



Global frame

Python works out:

`result = sum_to(4)` ⇒ first call `sum_to(4)`

Python starts on line 8. First it must call `sum_to(4)`.

Tracing sum_to(4): Going Down

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

sum_to n: 4

Global frame

line 8: result = ?

Python works out:

sum_to(4) ⇒ n = 4

A new frame for this call. In it, n is 4.

Tracing sum_to(4): Going Down

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

sum_to n: 4

Global frame

line 8: result = ?

Python works out:

4 <= 0 ⇒ False

Is 4 <= 0? **No.** So Python skips line 4.

Tracing sum_to(4): Going Down

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



sum_to n: 4 | 4 + sum_to(3)

Global frame | line 8: result = ?

Python works out:

`n + sum_to(n - 1)` ⇒ `4 + sum_to(3)`

Line 6. To finish it, Python needs `sum_to(3)` first.

Tracing sum_to(4): Going Down

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

sum_to n: 3

sum_to n: 4 — line 6: 4 + ?

Global frame — line 8: result = ?

Python works out:

sum_to(3) ⇒ n = 3

sum_to(4) waits. A new frame: n is 3.

Tracing `sum_to(4)`: Going Down

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

sum_to n: 3	
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

3 <= 0 ⇒ False

Is 3 <= 0? **No.** So Python skips line 4.

Tracing sum_to(4): Going Down

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

sum_to n: 3	3 + sum_to(2)
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`n + sum_to(n - 1)` ⇒ `3 + sum_to(2)`

Line 6. To finish it, Python needs `sum_to(2)` first.

Tracing sum_to(4): Going Down

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

sum_to n: 2	
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`sum_to(2)` ⇒ `n = 2`

`sum_to(3)` waits. A new frame: `n` is 2.

Tracing sum_to(4): Going Down

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

sum_to n: 2	
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

2 <= 0 ⇒ False

Is 2 <= 0? **No.** So Python skips line 4.

Tracing sum_to(4): Going Down

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

sum_to n: 2	2 + sum_to(1)
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`n + sum_to(n - 1)` ⇒ `2 + sum_to(1)`

Line 6. To finish it, Python needs `sum_to(1)` first.

Tracing sum_to(4): Going Down

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

sum_to n: 1	
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`sum_to(1)` ⇒ `n = 1`

`sum_to(2)` waits. A new frame: `n` is 1.

Tracing sum_to(4): Going Down

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

sum_to n: 1	
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`1 <= 0` ⇒ `False`

Is 1 <= 0? **No.** So Python skips line 4.

Tracing sum_to(4): Going Down

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

sum_to n: 1	1 + sum_to(0)
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`n + sum_to(n - 1)` ⇒ `1 + sum_to(0)`

Line 6. To finish it, Python needs `sum_to(0)` first.

Tracing sum_to(4): Going Down

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

sum_to n: 0	
sum_to n: 1	line 6: 1 + ?
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`sum_to(0)` ⇒ `n = 0`

`sum_to(1)` waits. A new frame: `n` is 0.

Tracing `sum_to(4)`: Going Down

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

sum_to n: 0	
sum_to n: 1	line 6: 1 + ?
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`0 <= 0` ⇒ `True`

Is 0 <= 0? **Yes!** This is the base case.

Tracing sum_to(4): Going Down

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

sum_to n: 0	returns 0
sum_to n: 1	line 6: 1 + ?
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`return 0` ⇒ 0

sum_to(0) returns 0. No new call. Now we go back up.

Tracing `sum_to(4)`: Coming Back Up

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

sum_to n: 0	returns 0
sum_to n: 1	line 6: 1 + ?
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

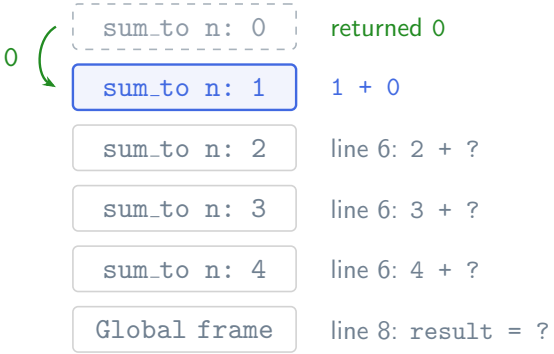
`return 0` ⇒ 0

Where we stopped: `sum_to(0)` returns 0.

Tracing `sum_to(4)`: Coming Back Up

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





Python works out:

`1 + sum_to(0)` ⇒ `1 + 0`

Its frame is removed. Back in `n`: 1, on line 6. The **?** becomes 0.

Tracing sum_to(4): Coming Back Up

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

sum_to n: 1	returns 1
sum_to n: 2	line 6: 2 + ?
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

1 + 0 ⇒ 1

1 + 0 = 1. So sum_to(1) returns 1.

Tracing `sum_to(4)`: Coming Back Up

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





Python works out:

`2 + sum_to(1)` ⇒ `2 + 1`

Its frame is removed. Back in `n: 2`, on line 6. The **?** becomes 1.

Tracing sum_to(4): Coming Back Up

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

sum_to n: 2	returns 3
sum_to n: 3	line 6: 3 + ?
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

2 + 1 ⇒ 3

2 + 1 = 3. So sum_to(2) returns 3.

Tracing sum_to(4): Coming Back Up

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

3

sum_to n: 2	returned 3
sum_to n: 3	3 + 3
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

`3 + sum_to(2)` ⇒ `3 + 3`

Its frame is removed. Back in `n`: 3, on line 6. The **?** becomes 3.

Tracing `sum_to(4)`: Coming Back Up

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

sum_to n: 3	returns 6
sum_to n: 4	line 6: 4 + ?
Global frame	line 8: result = ?

Python works out:

3 + 3 ⇒ 6

3 + 3 = 6. So sum_to(3) returns 6.

Tracing `sum_to(4)`: Coming Back Up

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





Python works out:

`4 + sum_to(3)` ⇒ `4 + 6`

Its frame is removed. Back in `n: 4`, on line 6. The **?** becomes 6.

Tracing sum_to(4): Coming Back Up

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



sum_to n: 4 returns 10

Global frame line 8: result = ?

Python works out:

4 + 6 ⇒ 10

4 + 6 = 10. So sum_to(4) returns 10.

Tracing `sum_to(4)`: Coming Back Up

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




Python works out:

`result = sum_to(4)` ⇒ `result = 10`

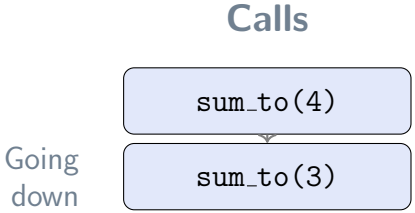
Its frame is removed. Back on line 8: `result` is **10**.

Going Down, Coming Back Up

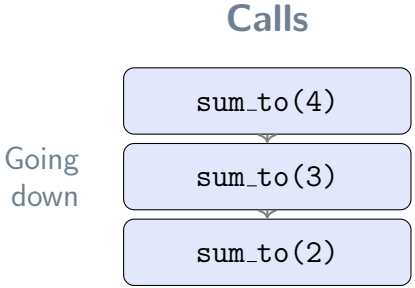
Calls

```
sum_to(4)
```

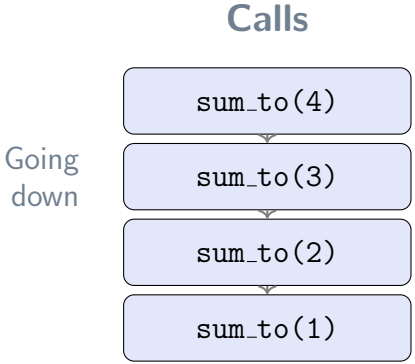
Going Down, Coming Back Up



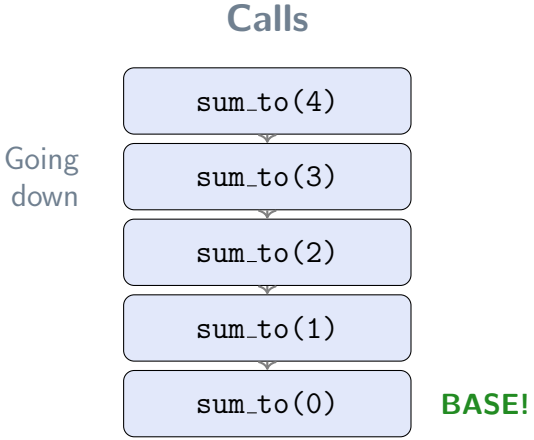
Going Down, Coming Back Up



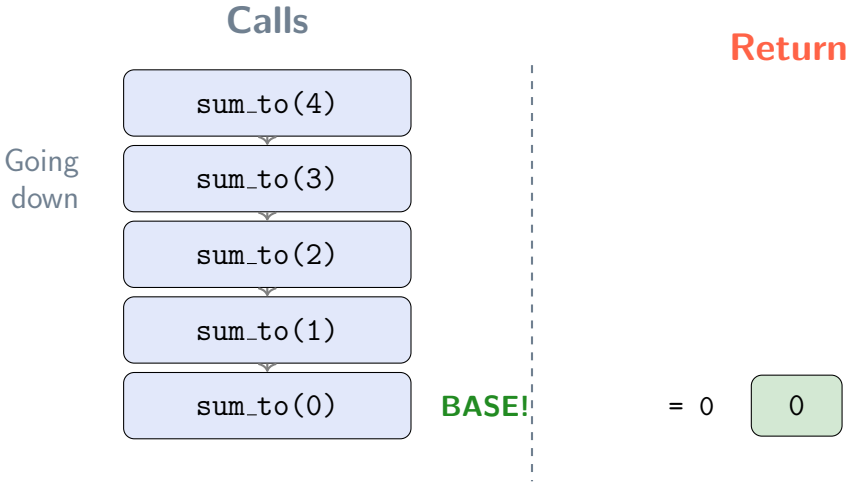
Going Down, Coming Back Up



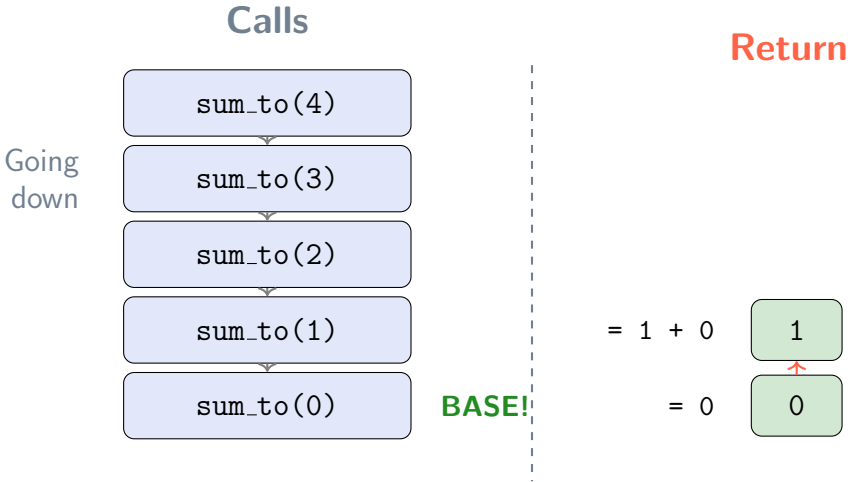
Going Down, Coming Back Up



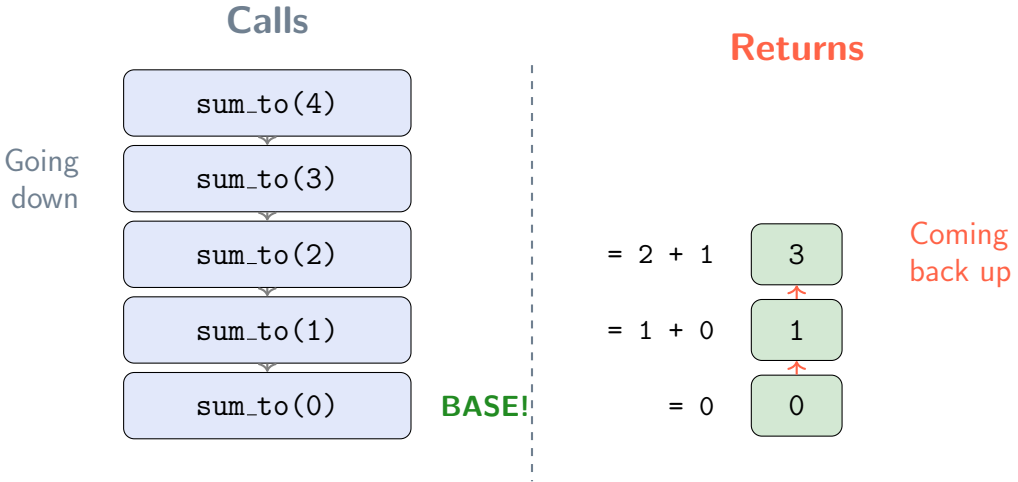
Going Down, Coming Back Up



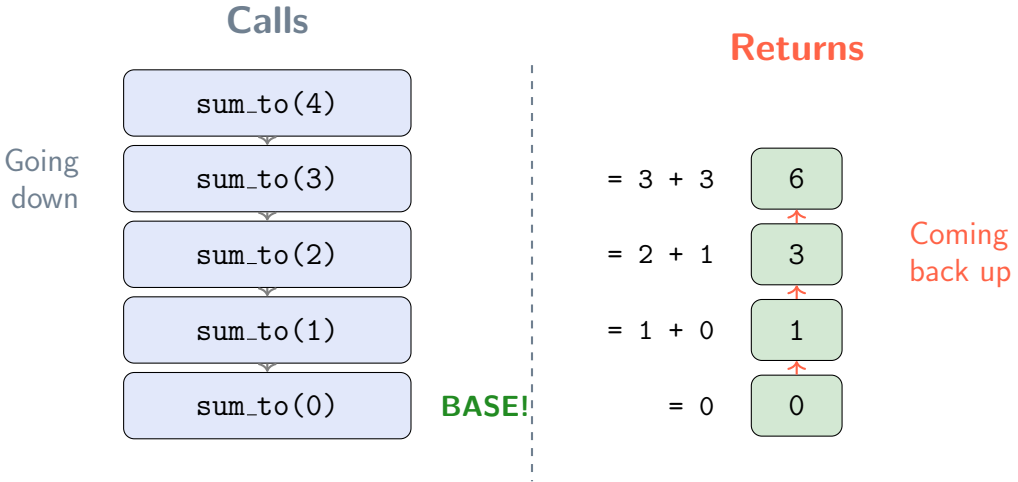
Going Down, Coming Back Up



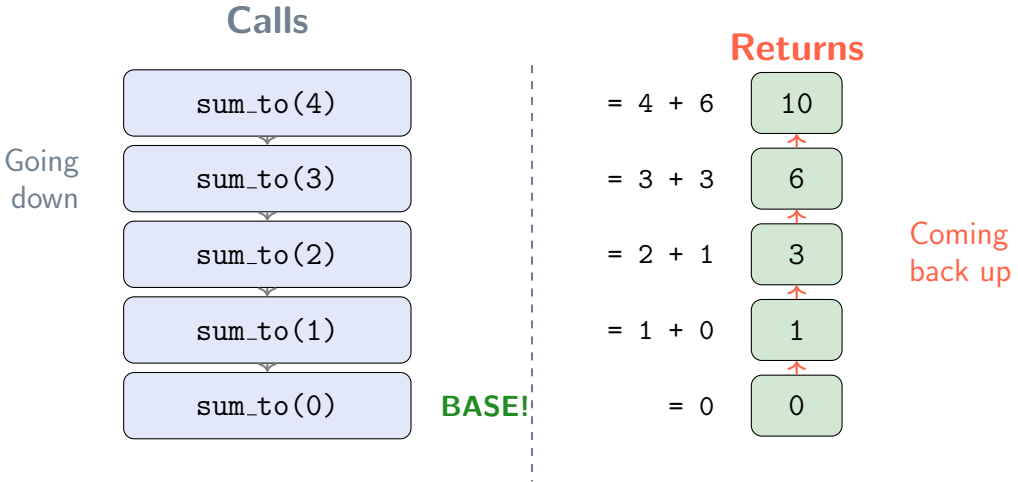
Going Down, Coming Back Up



Going Down, Coming Back Up



Going Down, Coming Back Up



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

With `if n <= 0:` the function also handles:

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

With `if n <= 0:` the function also handles:

- `sum_to(0)` → **0** ✓

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

With `if n <= 0:` the function also handles:

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

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

With `if n <= 0:` the function also handles:

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

With `if n == 1:`, try `sum_to(0)`. It goes $-1, -2, -3, \ldots$

It never reaches the base case.

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

With `if n <= 0:` the function also handles:

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

With `if n == 1:`, try `sum_to(0)`. It goes $-1, -2, -3, \ldots$

It never reaches the base case.

Always ask: which odd inputs could reach my function?

You Try! *(write the output, 1 min)*

The `print` is now **after** the call. What does `countup(3)` print?

```
1 def countup(n):
2     if n == 0:
3         return
4     countup(n - 1)
5     print(n)
6
7 countup(3)
```

You Try! *(write the output, 1 min)*

The `print` is now **after** the call. What does `countup(3)` print?

```
1 def countup(n):
2     if n == 0:
3         return
4     countup(n - 1)
5     print(n)
6
7 countup(3)
```

1
2
3

You Try! *(write the output, 1 min)*

The `print` is now **after** the call. What does `countup(3)` print?

```
1 def countup(n):
2     if n == 0:
3         return
4     countup(n - 1)
5     print(n)
6
7 countup(3)
```

```
1
2
3
```

Why? `countup(3)` must wait for `countup(2)` to finish before it prints 3. The **top** frame prints **first**.

Watch the Frames in Thonny

1. Open `lec21.py`.

Watch the Frames in Thonny

1. Open `lec21.py`.
2. **Debug current script** (the bug button, or Ctrl+F5).

Watch the Frames in Thonny

1. Open `lec21.py`.
2. **Debug current script** (the bug button, or Ctrl+F5).
3. Press **Step into** (F7) again and again.

Watch the Frames in Thonny

1. Open `lec21.py`.
2. **Debug current script** (the bug button, or Ctrl+F5).
3. Press **Step into** (F7) again and again.
4. Each call opens a **new window**: that window is a frame.

Watch the Frames in Thonny

1. Open `lec21.py`.
2. **Debug current script** (the bug button, or Ctrl+F5).
3. Press **Step into** (F7) again and again.
4. Each call opens a **new window**: that window is a frame.

At home you can also try `https://pythontutor.com`.

The Day the Frames Run Out

No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

infinite(5)

No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

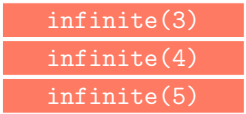
infinite(4)
infinite(5)

No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

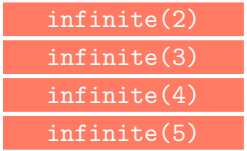


No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

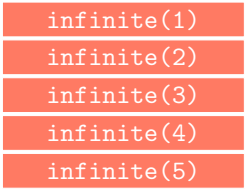


No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

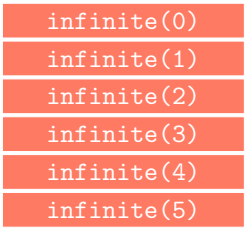


No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

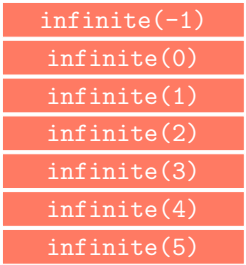


No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

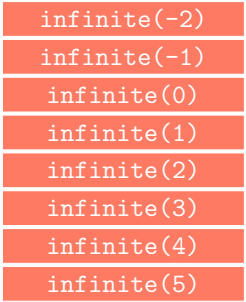


No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

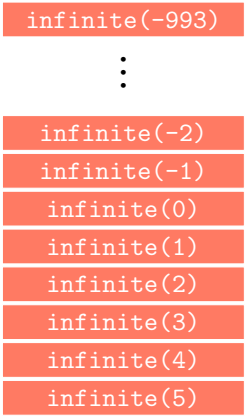


No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```

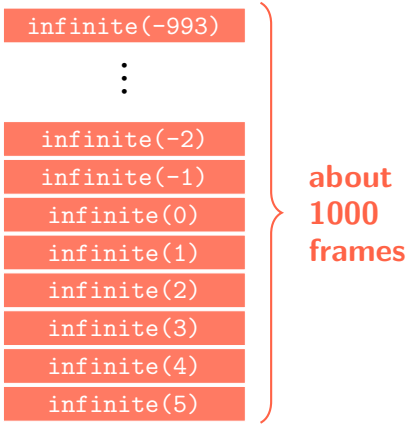


No Base Case

```
1 def infinite(n):
2     print(n)
3     infinite(n - 1)
4
5 infinite(5)
```

Output:

```
5
4
...
-993
RecursionError
```



The Pile Has a Limit

- Every frame takes some **memory**.

The Pile Has a Limit

- Every frame takes some **memory**.
- Python allows about **1000** frames on the stack.

The Pile Has a Limit

- Every frame takes some **memory**.
- Python allows about **1000** frames on the stack.
- One more, and Python stops the program:

```
RecursionError: maximum recursion depth exceeded
```

The Pile Has a Limit

- Every frame takes some **memory**.
- Python allows about **1000** frames on the stack.
- One more, and Python stops the program:

```
RecursionError: maximum recursion depth exceeded
```

A stack of frames that grows past its limit
is a **stack overflow**.

The Pile Has a Limit

- Every frame takes some **memory**.
- Python allows about **1000** frames on the stack.
- One more, and Python stops the program:

```
RecursionError: maximum recursion depth exceeded
```

A stack of frames that grows past its limit
is a **stack overflow**.

Programmers named their website after the bug they **all** meet.

Bug Hunt! *(write the line number and the fix, 30 s)*

`power(b, e)` should give b^e. For example, `power(2, 3)` is 8.

```
1 def power(b, e):
2     if e == 0:
3         return 1
4     return b * power(b, e)
5
6 print(power(2, 3))
```

Symptom:

```
RecursionError: maximum
recursion depth exceeded
```

Bug Hunt! *(write the line number and the fix, 30 s)*

`power(b, e)` should give b^e. For example, `power(2, 3)` is 8.

```
1 def power(b, e):
2     if e == 0:
3         return 1
4     return b * power(b, e)
5
6 print(power(2, 3))
```

Symptom:

```
RecursionError: maximum
recursion depth exceeded
```

Line 4. `e` never gets smaller, so `e == 0` never comes.

Fix: `power(b, e - 1)`

Bug Hunt! *(write the line number and the fix, 1 min)*

Count down in **twos**, then say `Blastoff!`

```
1  def twos(n):
2      if n == 0:
3          print("Blastoff!")
4          return
5      print(n)
6      twos(n - 2)
```

Symptom:

`twos(6)` prints `6 4 2 Blastoff!`
`twos(7)` prints `7 5 3 1 -1 -3...`
then `RecursionError`

Bug Hunt! *(write the line number and the fix, 1 min)*

Count down in **twos**, then say `Blastoff!`

```
1  def twos(n):
2      if n == 0:
3          print("Blastoff!")
4          return
5      print(n)
6      twos(n - 2)
```

Symptom:

`twos(6)` prints `6 4 2 Blastoff!`
`twos(7)` prints `7 5 3 1 -1 -3...`
then `RecursionError`

Line 2. From 7, `n` goes $1, -1$: it **jumps over** 0.
Fix: `if n <= 0:`

Bug Hunt! *(write the line number and the fix, 1 min)*

```
1 def sum_to(n):
2     if n <= 0:
3         return 0
4     n + sum_to(n - 1)
5
6 print(sum_to(3))
```

Symptom:

```
TypeError: unsupported
operand type(s) for +:
'int' and 'NoneType'
```

Bug Hunt! *(write the line number and the fix, 1 min)*

```
1 def sum_to(n):
2     if n <= 0:
3         return 0
4     n + sum_to(n - 1)
5
6 print(sum_to(3))
```

Symptom:

```
TypeError: unsupported
operand type(s) for +:
'int' and 'NoneType'
```

Line 4. It works out `n + ...` and throws it away. No `return`, so the call gives `None`.

Fix: `return n + sum_to(n - 1)`

Bug Hunt! *(write the line number and the fix, 1 min)*

```
1 def sum_to(n):
2     if n <= 0:
3         return 0
4     n + sum_to(n - 1)
5
6 print(sum_to(3))
```

Symptom:

```
TypeError: unsupported
operand type(s) for +:
'int' and 'NoneType'
```

Line 4. It works out `n + ...` and throws it away. No `return`, so the call gives `None`.

Fix: `return n + sum_to(n - 1)`
`sum_to(1)` gives `None`. Then `sum_to(2)` tries `2 + None`: **TypeError**.

The Big Three Recursion Bugs

Bug	Symptom	Fix

The Big Three Recursion Bugs

Bug	Symptom	Fix
Not getting smaller	`RecursionError`	`e - 1`, not `e`

The Big Three Recursion Bugs

Bug	Symptom	Fix
Not getting smaller	`RecursionError`	`e - 1`, not `e`
Base case jumped over	`RecursionError`	`<=`, not `==`

The Big Three Recursion Bugs

Bug	Symptom	Fix
Not getting smaller	`RecursionError`	`e - 1`, not `e`
Base case jumped over	`RecursionError`	`<=`, not `==`
Missing `return`	`None` or `TypeError`	`return` the answer

The Big Three Recursion Bugs

Bug	Symptom	Fix
Not getting smaller	`RecursionError`	`e - 1`, not `e`
Base case jumped over	`RecursionError`	`<=`, not `==`
Missing `return`	`None` or `TypeError`	`return` the answer

Seen a `RecursionError`? Ask:

does every call get closer to the base case?

Writing Recursive Functions

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \qquad = 5 \times 4!$

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \quad = 5 \times 4!$

```
1  def factorial(n):
2      if n <= 1:
3          return 1
4      return n * factorial(n - 1)
5
6  print(factorial(5))
7  print(factorial(0))
```

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \quad = 5 \times 4!$

```
1  def factorial(n):
2      if n <= 1:
3          return 1
4      return n * factorial(n - 1)
5
6  print(factorial(5))
7  print(factorial(0))
```

factorial(5)

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \quad = 5 \times 4!$

```
1  def factorial(n):
2      if n <= 1:
3          return 1
4      return n * factorial(n - 1)
5
6  print(factorial(5))
7  print(factorial(0))
```

```
factorial(5)
  = 5 * factorial(4)
```

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \quad = 5 \times 4!$

```
1  def factorial(n):
2      if n <= 1:
3          return 1
4      return n * factorial(n - 1)
5
6  print(factorial(5))
7  print(factorial(0))
```

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

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \quad = 5 \times 4!$

```
1  def factorial(n):
2      if n <= 1:
3          return 1
4      return n * factorial(n - 1)
5
6  print(factorial(5))
7  print(factorial(0))
```

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

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \quad = 5 \times 4!$

```
1  def factorial(n):
2      if n <= 1:
3          return 1
4      return n * factorial(n - 1)
5
6  print(factorial(5))
7  print(factorial(0))
```

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

Factorial

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \quad = 5 \times 4!$

```
1  def factorial(n):
2      if n <= 1:
3          return 1
4      return n * factorial(n - 1)
5
6  print(factorial(5))
7  print(factorial(0))
```

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

= 120

`factorial(0)` is 1.

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1 def length(s):
2     if s == "":
3         return 0
4     return 1 + length(s[1:])
5
6 print(length("hello"))
```

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1 def length(s):
2     if s == "":
3         return 0
4     return 1 + length(s[1:])
5
6 print(length("hello"))
```

```
length("hello")
```

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1 def length(s):
2     if s == "":
3         return 0
4     return 1 + length(s[1:])
5
6 print(length("hello"))
```

```
length("hello")
 = 1 + length("ello")
```

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1 def length(s):
2     if s == "":
3         return 0
4     return 1 + length(s[1:])
5
6 print(length("hello"))
```

```
length("hello")
 = 1 + length("ello")
  = 1+1 + length("llo")
```

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1  def length(s):
2      if s == "":
3          return 0
4      return 1 + length(s[1:])
5
6  print(length("hello"))
```

```
length("hello")
 = 1 + length("ello")
  = 1+1 + length("llo")
   = 1+1+1 + length("lo")
```

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1 def length(s):
2     if s == "":
3         return 0
4     return 1 + length(s[1:])
5
6 print(length("hello"))
```

```
length("hello")
 = 1 + length("ello")
  = 1+1 + length("llo")
   = 1+1+1 + length("lo")
    = 1+1+1+1 + length("o")
```

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1  def length(s):
2      if s == "":
3          return 0
4      return 1 + length(s[1:])
5
6  print(length("hello"))
```

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

Length of a String, Without `len`

- The empty string `""` has length **0**. (base case)
- Any other string: **1** + the length of the rest, `s[1:]`.

```
1  def length(s):
2      if s == "":
3          return 0
4      return 1 + length(s[1:])
5
6  print(length("hello"))
```

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

= 5

You Try! *(write the function on paper, 3 min)*

Write `multiply(a, b)` for whole numbers $b \geq 0$. Use **only** +, no $*$.

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

You Try! *(write the function on paper, 3 min)*

Write `multiply(a, b)` for whole numbers $b \geq 0$. Use **only** +, no $*$.

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

```
1  def multiply(a, b):
2      if b == 0:
3          return 0
4      return a + multiply(a, b - 1)
5
6  print(multiply(3, 4))      # 12
```

You Try! *(write the function on paper, 3 min)*

Write `multiply(a, b)` for whole numbers $b \geq 0$. Use **only** +, no *.

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

```
1  def multiply(a, b):
2      if b == 0:
3          return 0
4      return a + multiply(a, b - 1)
5
6  print(multiply(3, 4))      # 12
```

Base case: anything × 0 is 0.

Smaller job: `b - 1`.

The Leap of Faith

To write `factorial(n)`, **trust** that `factorial(n - 1)` already works.

Then ask only: how do I use it to get `factorial(n)`?

The Leap of Faith

To write `factorial(n)`, **trust** that `factorial(n - 1)` already works.

Then ask only: how do I use it to get `factorial(n)`?

Don't trace every call in your head.

Do check the base case, and check that each call is smaller.

Checklist

✓ A **base case**: when to stop.

Checklist

✓ A **base case**: when to stop.

✓ A **recursive case**: call yourself on a smaller job.

Checklist

- ✓ A **base case**: when to stop.
- ✓ A **recursive case**: call yourself on a smaller job.
- ✓ Every call moves **toward** the base case.

Checklist

- ✓ A **base case**: when to stop.
- ✓ A **recursive case**: call yourself on a smaller job.
- ✓ Every call moves **toward** the base case.
- ✓ Odd inputs (0, negatives, "") still stop.

Checklist

- ✓ A **base case**: when to stop.
- ✓ A **recursive case**: call yourself on a smaller job.
- ✓ Every call moves **toward** the base case.
- ✓ Odd inputs (0, negatives, `""`) still stop.
- ✓ Every path ends in a `return`.

You Try! *(write the output, 2 min)*

One `print` **before** the call, one **after**.

(a) What does `echo(2)` print? **(b)** At most, how many `echo` frames?

```
1 def echo(n):
2     if n == 0:
3         print("base")
4         return
5     print("down", n)
6     echo(n - 1)
7     print("up", n)
8
9 echo(2)
```

You Try! *(write the output, 2 min)*

One `print` **before** the call, one **after**.

(a) What does `echo(2)` print? **(b)** At most, how many `echo` frames?

```
1 def echo(n):
2     if n == 0:
3         print("base")
4         return
5     print("down", n)
6     echo(n - 1)
7     print("up", n)
8
9 echo(2)
```

```
down 2
down 1
base
up 1
up 2
```

You Try! *(write the output, 2 min)*

One `print` **before** the call, one **after**.

(a) What does `echo(2)` print? **(b)** At most, how many `echo` frames?

```
1  def echo(n):
2      if n == 0:
3          print("base")
4          return
5      print("down", n)
6      echo(n - 1)
7      print("up", n)
8
9  echo(2)
```

```
down 2
down 1
base
up 1
up 2
```

3 frames

- echo n: 0
- echo n: 1
- echo n: 2
- Global frame

You Try! *(write the output, 2 min)*

One `print` **before** the call, one **after**.

(a) What does `echo(2)` print?

```
1  def echo(n):
2      if n == 0:
3          print("base")
4          return
5      print("down", n)
6      echo(n - 1)
7      print("up", n)
8
9  echo(2)
```

(b) At most, how many `echo` frames?

```
down 2
down 1
base
up 1
up 2
```

3 frames

- echo n: 0
- echo n: 1
- echo n: 2
- Global frame

Why? The **downs** print going down.

The **ups** print coming back up.

You Try! *(2 min)*

How many times can you halve `n` before you reach 1?

```
1  def halvings(n):
2      if n == 1:
3          return 0
4      return 1 + halvings(n // 2)
```

(a) What does `halvings(8)` return?

(b) What happens with `halvings(0)`?

(c) Fix it. Change **one** line.

You Try! *(2 min)*

How many times can you halve `n` before you reach 1?

```
1  def halvings(n):
2      if n == 1:
3          return 0
4      return 1 + halvings(n // 2)
```

```
halvings(8)
= 1 + halvings(4)
= 1 + 1 + halvings(2)
= 1 + 1 + 1 + halvings(1)
= 1 + 1 + 1 + 0 = 3
```

(a) What does `halvings(8)` return?

(b) What happens with `halvings(0)`?

(c) Fix it. Change **one** line.

You Try! *(2 min)*

How many times can you halve `n` before you reach 1?

```
1  def halvings(n):
2      if n == 1:
3          return 0
4      return 1 + halvings(n // 2)
```

(a) What does `halvings(8)` return?

(b) What happens with `halvings(0)`?

(c) Fix it. Change **one** line.

```
halvings(8)
= 1 + halvings(4)
= 1 + 1 + halvings(2)
= 1 + 1 + 1 + halvings(1)
= 1 + 1 + 1 + 0 = 3
```

(b) `0 // 2` is 0. `n` never reaches 1: `RecursionError`.

You Try! *(2 min)*

How many times can you halve `n` before you reach 1?

```
1  def halvings(n):
2      if n == 1:
3          return 0
4      return 1 + halvings(n // 2)
```

(a) What does `halvings(8)` return?

(b) What happens with `halvings(0)`?

(c) Fix it. Change **one** line.

```
halvings(8)
= 1 + halvings(4)
= 1 + 1 + halvings(2)
= 1 + 1 + 1 + halvings(1)
= 1 + 1 + 1 + 0 = 3
```

(b) `0 // 2` is 0. `n` never reaches 1: `RecursionError`.

(c) Line 2: `if n <= 1:`

You Try! *(write the function on paper, 3 min)*

Write `countdown_list(n)`. It **returns** the list `[n, ..., 2, 1]`.

`countdown_list(3)` → `[3, 2, 1]` `countdown_list(0)` → `[]`

Hint: `[3, 2, 1]` is `[3]` + `[2, 1]`

You Try! *(write the function on paper, 3 min)*

Write `countdown_list(n)`. It **returns** the list `[n, ..., 2, 1]`.

`countdown_list(3)` → `[3, 2, 1]` `countdown_list(0)` → `[]`

Hint: `[3, 2, 1]` is `[3] + [2, 1]`

```
1 def countdown_list(n):
2     if n == 0:
3         return []
4     return [n] + countdown_list(n - 1)
5
6 print(countdown_list(3))
```

You Try! *(write the function on paper, 3 min)*

Write `countdown_list(n)`. It **returns** the list `[n, ..., 2, 1]`.

`countdown_list(3)` → `[3, 2, 1]` `countdown_list(0)` → `[]`

Hint: `[3, 2, 1]` is `[3]` + `[2, 1]`

```
1 def countdown_list(n):
2     if n == 0:
3         return []
4     return [n] + countdown_list(n - 1)
5
6 print(countdown_list(3))
```

Base case: 0 gives the empty list `[]`.

Smaller job: `n - 1`.

Summary

Summary

- **Recursion:** a function calls itself on a smaller job.
- **Two parts:** a base case and a recursive case.
- **Call stack:** every call gets a frame. The newest is on top.
- **No base case:** about 1000 frames, then `RecursionError`.
- **Big Three bugs:** not smaller, base case jumped over, missing `return`.

Big Idea

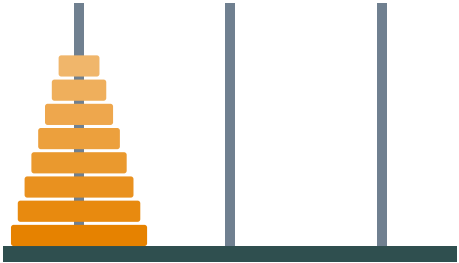
A function may call itself...

if each call is **smaller**
and a **base case** stops it.

Without a base case, the frames pile up until they **overflow the stack**.

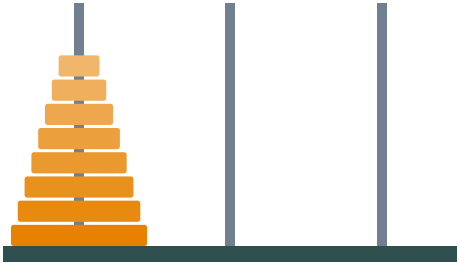
Next Time: Thinking Recursively

- A temple. Three needles. **64** golden disks.



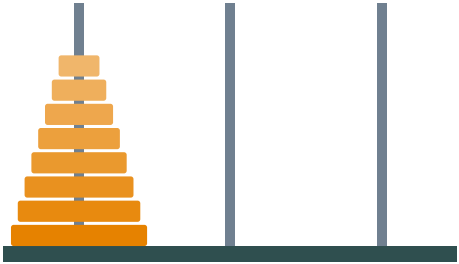
Next Time: Thinking Recursively

- A temple. Three needles. **64** golden disks.
- The priests move one disk at a time.



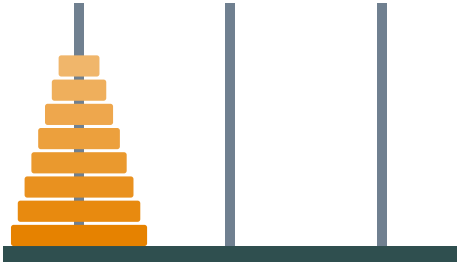
Next Time: Thinking Recursively

- A temple. Three needles. **64** golden disks.
- The priests move one disk at a time.
- When they finish, the story says, **the world ends**.



Next Time: Thinking Recursively

- A temple. Three needles. **64** golden disks.
- The priests move one disk at a time.
- When they finish, the story says, **the world ends**.



Should we worry? Recursion will tell us.

Questions?

Exit Ticket

Today's question *(2 minutes)*

(a) What does `f(3)` return?

```
def f(n):
    if n == 0:
        return 0
    return n * 2 + f(n - 1)
```

(b) Why is the website called **Stack Overflow**? One line.

COMP 102 — Exit Ticket #21

Name: _____

Roll #: _____ Date: _____

Answer:

Hand it to me on your way out.
No name, no attendance.