Lecture 6: Recursion Foundations

Practice Problems & Debugging Recursion

Comp 111 — Programming 2

Forman Christian University

Warm-Up

30-Second Challenge!

1. What are the **TWO required parts** of any recursive function?

30-Second Challenge!

1. What are the **TWO required parts** of any recursive function?

   Answer: Base case + Recursive case

30-Second Challenge!

1. What are the **TWO required parts** of any recursive function?

   Answer: Base case + Recursive case

2. What error do you get without a base case?

30-Second Challenge!

1. What are the **TWO required parts** of any recursive function?

   Answer: Base case + Recursive case

2. What error do you get without a base case?

   Answer: RecursionError (infinite recursion → stack overflow!)

30-Second Challenge!

1. What are the **TWO required parts** of any recursive function?

   Answer: Base case + Recursive case

2. What error do you get without a base case?

   Answer: RecursionError (infinite recursion → stack overflow!)

3. In `sum_to(4)`, how many times does the function call itself?

30-Second Challenge!

1. What are the **TWO required parts** of any recursive function?

   Answer: Base case + Recursive case

2. What error do you get without a base case?

   Answer: RecursionError (infinite recursion → stack overflow!)

3. In `sum_to(4)`, how many times does the function call itself?

   Answer: 4 times — sum_to(3), sum_to(2), sum_to(1), sum_to(0)

Homework Review: Power Function

```python
def power(base, exp):
    # Base case: anything^0 = 1
    if exp == 0:
        return 1

    # Recursive case
    return base * power(base, exp - 1)

print(power(2, 3))  # Output: 8
```

Homework Review: Power Function

```python
def power(base, exp):
    # Base case: anything^0 = 1
    if exp == 0:
        return 1

    # Recursive case
    return base * power(base, exp - 1)

print(power(2, 3))  # Output: 8
```

Trace: power(2, 3)

```
power(2, 3)
 = 2 * power(2, 2)
  = 2 * 2 * power(2, 1)
   = 2 * 2 * 2 * power(2, 0)
    = 2 * 2 * 2 * 1

= 8
```

Designing Recursive Solutions

WISE: Your Recursion Recipe

For every problem, ask these 4 questions:

W — **W**hat's the simplest case? (base case)

WISE: Your Recursion Recipe

For every problem, ask these 4 questions:

W — **W**hat's the simplest case? (base case)

I — **I**f not simple, how do I shrink it?

WISE: Your Recursion Recipe

For every problem, ask these 4 questions:

W — **W**hat's the simplest case? (base case)

I — **I**f not simple, how do I shrink it?

S — **S**olve the smaller problem (trust it!)

WISE: Your Recursion Recipe

For every problem, ask these 4 questions:

W — **W**hat's the simplest case? (base case)

I — **I**f not simple, how do I shrink it?

S — **S**olve the smaller problem (trust it!)

E — **E**xtend small solution to full answer

You Try: Reverse a String

`reverse("hello")` → `"olleh"`

`reverse("cat")` → `"tac"`

Use WISE to **design** your solution:

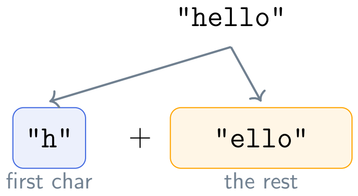
- What's the simplest string to reverse?
- How do you make a string smaller?
- How do you combine?

Take 4 minutes. Write pseudocode or Python.

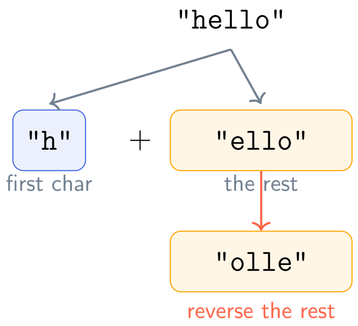
Think: What's the Recursive Insight?

```
"hello"
```

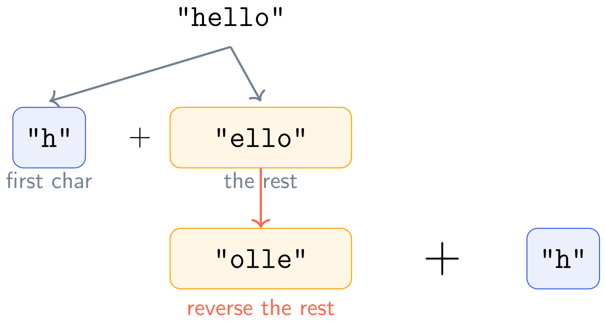
Think: What's the Recursive Insight?



Think: What's the Recursive Insight?

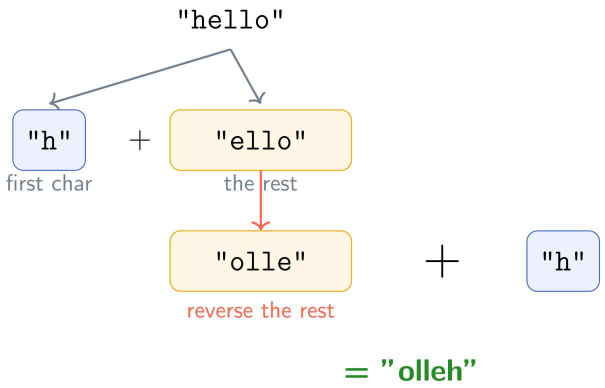


Think: What's the Recursive Insight?



= "olleh"

Think: What's the Recursive Insight?



reverse(s) = reverse(s[1:]) + s[0]

Solution: Reverse String

```python
def reverse(s):
    # W: simplest case
    if len(s) <= 1:
        return s

    # I+S: shrink and solve
    # E: combine
    return reverse(s[1:]) + s[0]
```

Solution: Reverse String

```python
def reverse(s):
    # W: simplest case
    if len(s) <= 1:
        return s

    # I+S: shrink and solve
    # E: combine
    return reverse(s[1:]) + s[0]
```

WISE Breakdown:

W: `""` or `"x"` → return as-is

I: Remove first character

S: `reverse(s[1:])`

E: reversed rest + first char

Trace: reverse(”cat”)

```
reverse("cat")
```

Trace: reverse(”cat”)

```
reverse("cat")

 = reverse("at") + "c"
```

Trace: reverse(”cat”)

```
reverse("cat")

 = reverse("at") + "c"

  = reverse("t") + "a" + "c"
```

Trace: reverse("cat")

```
reverse("cat")
  = reverse("at") + "c"
    = reverse("t") + "a" + "c"
      = "t" + "a" + "c"
```
BASE CASE!

Trace: reverse(”cat”)

```
reverse("cat")
 = reverse("at") + "c"
  = reverse("t") + "a" + "c"
   = "t" + "a" + "c"BASE CASE!

= "tac"
```

Each call peels off the first char, then stacks it on the **right** going back up.

The Big Three Recursion Bugs

Bug	Symptom	Example	Fix
Missing return	Returns `None`	`f(n-1) + x` (no `return`)	Add `return`
Wrong base case	Wrong answer or infinite loop	`if n == 0` when n can be negative	Check edge cases
Not shrinking	RecursionError	`f(n)` calls `f(n)`	Ensure argument decreases

The Big Three Recursion Bugs

Bug	**Symptom**	**Example**	**Fix**
Missing return	Returns `None`	`f(n-1) + x` (no `return`)	Add `return`
Wrong base case	Wrong answer or infinite loop	`if n == 0` when n can be negative	Check edge cases
Not shrinking	RecursionError	`f(n)` calls `f(n)`	Ensure argument decreases

Memorize these. They explain 90% of recursion bugs.

Bug Hunt: Spot the Bugs!

This code has **2 bugs**. Find them both:

```python
def reverse_string(s):
    if s == "":
        print("")
    reverse_string(s[1:]) + s[0]
```

Discuss with a partner — 2 minutes!

Bug Hunt: Spot the Bugs!

This code has **2 bugs**. Find them both:

```
1 def reverse_string(s):
2     if s == "":
3         print("")
4     reverse_string(s[1:]) + s[0]
```

Discuss with a partner — 2 minutes!

Bug 1: `print("")` instead of `return ""` — **Missing return!**

Bug 2: Missing `return`, in line 4

Bug Hunt: Fixed

✗ BUGGY

```
def reverse_string(s):
    if s == "":
        print("")
    reverse_string(s[1:])
        + s[0]
```

✓ FIXED

```
def reverse_string(s):
    if s == "":
        return ""
    return reverse_string(s
        [1:]) + s[0]
```

Lesson: Always `return` from the base case — `print` is not `return`!

You Try: Count Vowels

`count_vowels("hello")` → 2

`count_vowels("aeiou")` → 5

`count_vowels("xyz")` → 0

Apply WISE:

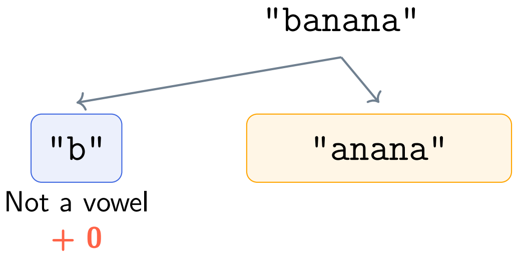
- What's the simplest string to count vowels in?
- How do you shrink the string?
- What do you do with the first character?

Take 4 minutes.

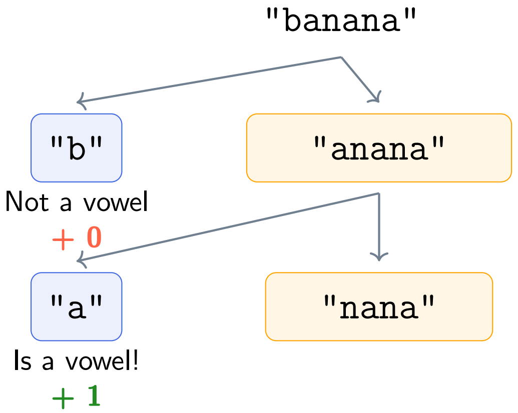
Think: What Changes at Each Step?

```
"banana"
```

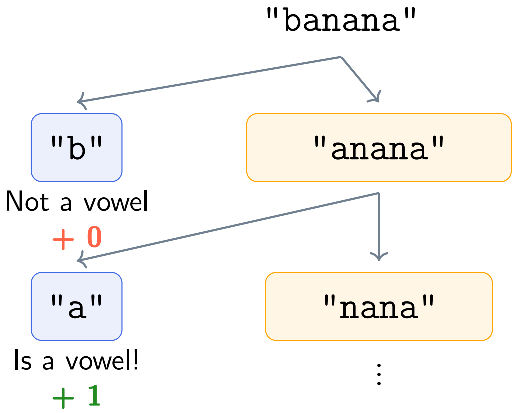
Think: What Changes at Each Step?



Think: What Changes at Each Step?



Think: What Changes at Each Step?



At each step:

Is first char a vowel?

Yes → 1 + count rest

No → 0 + count rest

Solution: Count Vowels

```python
def count_vowels(s):
    # W: empty string
    if s == "":
        return 0

    # I+S: check first, count rest
    rest = count_vowels(s[1:])

    # E: add 1 if vowel, 0 if not
    if s[0] in "aeiouAEIOU":
        return 1 + rest
    else:
        return rest
```

Solution: Count Vowels

```python
def count_vowels(s):
    # W: empty string
    if s == "":
        return 0

    # I+S: check first, count rest
    rest = count_vowels(s[1:])

    # E: add 1 if vowel, 0 if not
    if s[0] in "aeiouAEIOU":
        return 1 + rest
    else:
        return rest
```

WISE:

W: `""` $\rightarrow$ 0 vowels

I: Remove first char

S: `count_vowels(s[1:])`

E: $+1$ if vowel, $+0$ if not

Trace: count_vowels("hey")

```
count_vowels("hey")
```

Trace: count_vowels("hey")

```
count_vowels("hey")
 "h" is not a vowel
 0 + count_vowels("ey")
```

Trace: count_vowels("hey")

```
count_vowels("hey")
 "h" is not a vowel
 0 + count_vowels("ey")
  "e" IS a vowel
  1 + count_vowels("y")
```

Trace: count_vowels("hey")

```
count_vowels("hey")
 "h" is not a vowel
 0 + count_vowels("ey")
  "e" IS a vowel
  1 + count_vowels("y")
   "y" is not a vowel
   0 + count_vowels("")
```

Trace: count_vowels("hey")

```
count_vowels("hey")
 "h" is not a vowel
 0 + count_vowels("ey")
  "e" IS a vowel
  1 + count_vowels("y")
   "y" is not a vowel
   0 + count_vowels("")
    BASE! return 0
```

Trace: count_vowels("hey")

```
count_vowels("hey")
 "h" is not a vowel
 0 + count_vowels("ey")
  "e" IS a vowel
  1 + count_vowels("y")
   "y" is not a vowel
   0 + count_vowels("")
    BASE! return 0
```

Coming back up:

```
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1

Answer: 1
```

You Try: Palindrome Check

`is_palindrome("racecar")` → `True`

`is_palindrome("hello")` → `False`

`is_palindrome("aba")` → `True`

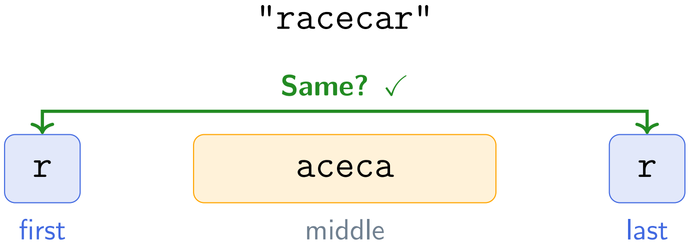
This is different from the previous problems.
The "smaller problem" isn't just removing the first character...

Take 4 minutes. Think about what makes a palindrome a palindrome.

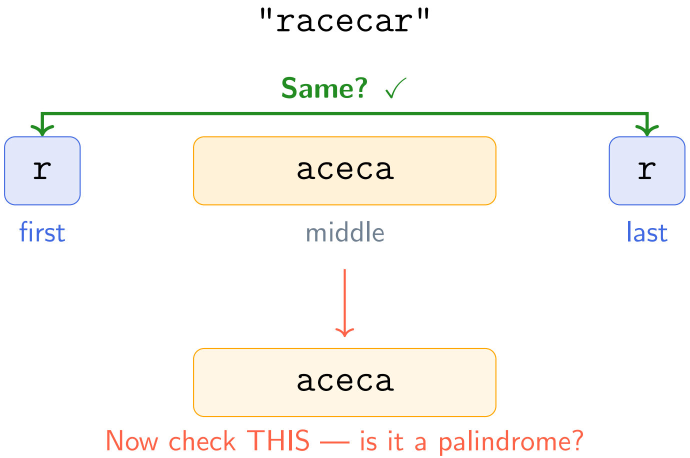
Think: How Does a Palindrome Shrink?

`"racecar"`

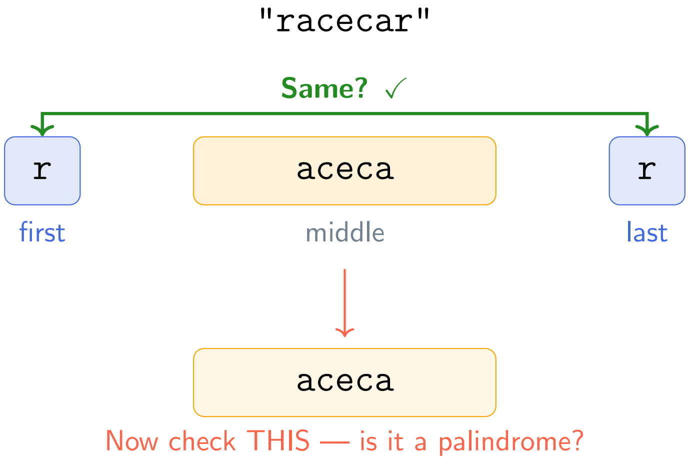
Think: How Does a Palindrome Shrink?



Think: How Does a Palindrome Shrink?



Think: How Does a Palindrome Shrink?



Shrinks from BOTH ends: `s[1:-1]`

Solution: Palindrome

```python
def is_palindrome(s):
    # W: 0 or 1 chars
    if len(s) <= 1:
        return True

    # I: compare ends
    if s[0] != s[-1]:
        return False

    # S+E: check middle
    return is_palindrome(s[1:-1])
```

Solution: Palindrome

```python
def is_palindrome(s):
    # W: 0 or 1 chars
    if len(s) <= 1:
        return True

    # I: compare ends
    if s[0] != s[-1]:
        return False

    # S+E: check middle
    return is_palindrome(s[1:-1])
```

WISE:

W: `""` or `"x"` → True

I: Remove both ends

S: `is_palindrome(s[1:-1])`

E: Only if ends match!

Key: Early return `False` if ends don't match.

Trace: is_palindrome(”racecar”)

```
is_palindrome("racecar")
```

Trace: is_palindrome(”racecar”)

```
is_palindrome("racecar")
 r == r ✓ → is_palindrome("aceca")
```

Trace: is_palindrome("racecar")

```
is_palindrome("racecar")
 r == r ✓ → is_palindrome("aceca")
  a == a ✓ → is_palindrome("cec")
```

Trace: is_palindrome("racecar")

```
is_palindrome("racecar")
 r == r ✓ → is_palindrome("aceca")
  a == a ✓ → is_palindrome("cec")
   c == c ✓ → is_palindrome("e")
```

Trace: is_palindrome("racecar")

```
is_palindrome("racecar")
 r == r ✓ → is_palindrome("aceca")
  a == a ✓ → is_palindrome("cec")
   c == c ✓ → is_palindrome("e")
    len ≤ 1 → True
```

BASE CASE!

Trace: is_palindrome(”racecar”)

```
is_palindrome("racecar")
 r == r ✓ → is_palindrome("aceca")
  a == a ✓ → is_palindrome("cec")
   c == c ✓ → is_palindrome("e")
    len ≤ 1 → True      BASE CASE!
```

True bubbles all the way back up. Every pair matched!

You Try: Find Maximum

Find the largest number **without** using `max()`.

`find_max([3, 7, 2, 9, 1])` → 9

New challenge: This time the data is a **list**, not a string.

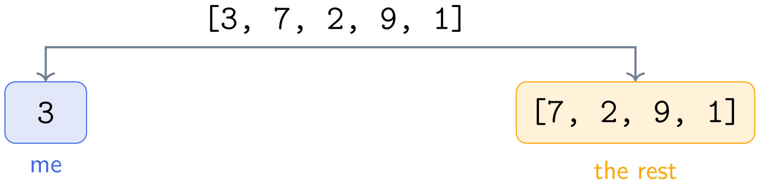
- What's the simplest list to find the max of?
- How do you make a list smaller?
- How do you combine "max of rest" with what you have?

Take 4 minutes.

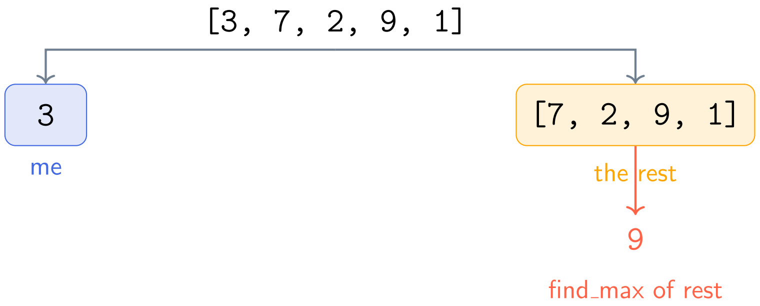
Think: Max = “Best of Me vs. the Rest”

```
[3, 7, 2, 9, 1]
```

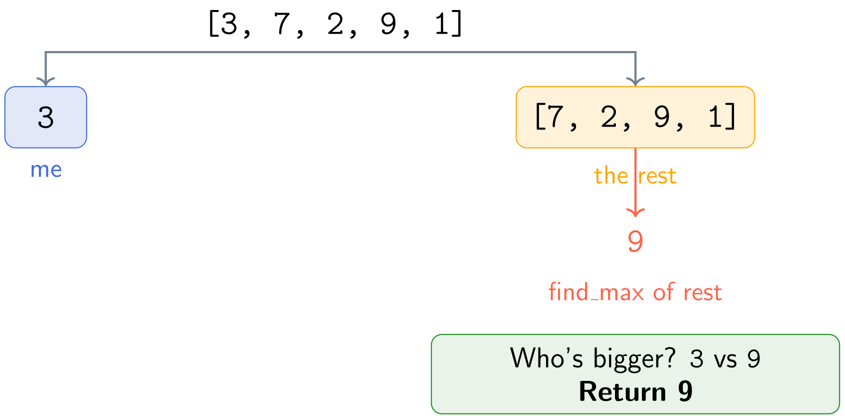
Think: Max = "Best of Me vs. the Rest"



Think: Max = "Best of Me vs. the Rest"



Think: Max = "Best of Me vs. the Rest"



Solution: Find Maximum

```python
def find_max(lst):
    # W: single element
    if len(lst) == 1:
        return lst[0]

    # I+S: max of the rest
    max_of_rest = find_max(lst[1:])

    # E: compare me vs rest
    if lst[0] > max_of_rest:
        return lst[0]
    else:
        return max_of_rest
```

Solution: Find Maximum

```python
def find_max(lst):
    # W: single element
    if len(lst) == 1:
        return lst[0]

    # I+S: max of the rest
    max_of_rest = find_max(lst[1:])

    # E: compare me vs rest
    if lst[0] > max_of_rest:
        return lst[0]
    else:
        return max_of_rest
```

WISE:

W: One element → that's the max

I: Remove first element

S: `find_max(lst[1:])`

E: Compare first vs max-of-rest

Trace: find_max([3, 7, 2])

Going down:

```
find_max([3, 7, 2])
```

Trace: find_max([3, 7, 2])

Going down:

```
find_max([3, 7, 2])
 find_max([7, 2])
```

Trace: find_max([3, 7, 2])

Going down:

```
find_max([3, 7, 2])
 find_max([7, 2])
  find_max([2])
```

Trace: find_max([3, 7, 2])

Going down:

```
find_max([3, 7, 2])
 find_max([7, 2])
  find_max([2])
   BASE! return 2
```

Trace: find_max([3, 7, 2])

Going down:

```
find_max([3, 7, 2])
 find_max([7, 2])
  find_max([2])
   BASE! return 2
```

Coming back up:

```
7 vs 2 → return 7
3 vs 7 → return 7

Answer: 7
```

The max "survives" each comparison on the way back up.

When Recursion, When Loops?

Recursion

- Recursive structure (trees, folders, nested lists)
- Divide & conquer
- Backtracking

Loops

- Simple counting / accumulation
- Performance critical
- Would recurse 1000+ times

When Recursion, When Loops?

Recursion

- Recursive structure (trees, folders, nested lists)
- Divide & conquer
- Backtracking

Loops

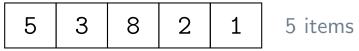
- Simple counting / accumulation
- Performance critical
- Would recurse 1000+ times

Python's recursion limit: ~1000 calls.
If you hit it, rewrite as a loop — don't increase the limit.

The Problem with Slicing

Every time we write `lst[1:]`, Python creates a **brand new list**:

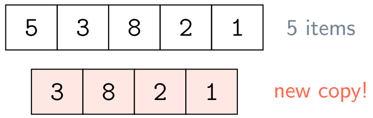
```
sum([5,3,8,2,1])
```



The Problem with Slicing

Every time we write `lst[1:]`, Python creates a **brand new list**:

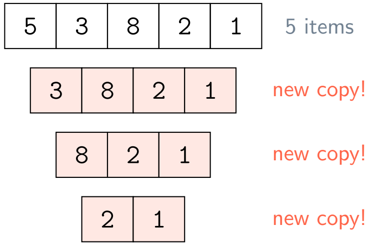
```
sum([5,3,8,2,1])

sum([3,8,2,1])
```



The Problem with Slicing

Every time we write `lst[1:]`, Python creates a **brand new list**:

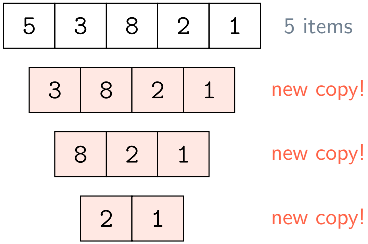
```
sum([5,3,8,2,1])

sum([3,8,2,1])

sum([8,2,1])

sum([2,1])
```



The Problem with Slicing

Every time we write `lst[1:]`, Python creates a **brand new list**:

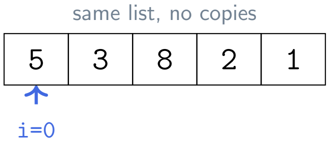
```
sum([5,3,8,2,1])

sum([3,8,2,1])

sum([8,2,1])

sum([2,1])
```



For a list of n items: $n + (n-1) + (n-2) + \cdots = O(n^2)$ copies!

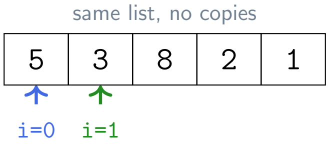
Better Idea: Use an Index

Instead of slicing, move an **index** through the **same list**:



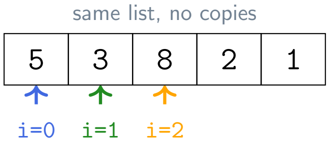
Better Idea: Use an Index

Instead of slicing, move an **index** through the **same list**:



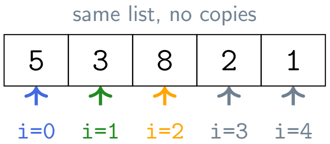
Better Idea: Use an Index

Instead of slicing, move an **index** through the **same list**:



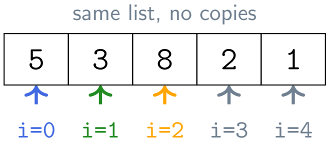
Better Idea: Use an Index

Instead of slicing, move an **index** through the **same list**:



Better Idea: Use an Index

Instead of slicing, move an **index** through the **same list**:



Zero copies — $O(n)$!

But now our function needs an extra parameter: `index`.
The caller shouldn't have to pass that...

First Attempt: Index as Parameter

Let's rewrite `sum_list` using an index instead of slicing:

```python
def sum_list(lst, index):
    if index == len(lst):
        return 0
    return lst[index] + sum_list(lst, index + 1)
```

First Attempt: Index as Parameter

Let's rewrite `sum_list` using an index instead of slicing:

```python
def sum_list(lst, index):
    if index == len(lst):
        return 0
    return lst[index] + sum_list(lst, index + 1)
```

No copies — but look at how you call it:

```python
sum_list([5, 3, 8], 0)  # What's the 0 for??
```

First Attempt: Index as Parameter

Let's rewrite `sum_list` using an index instead of slicing:

```python
def sum_list(lst, index):
    if index == len(lst):
        return 0
    return lst[index] + sum_list(lst, index + 1)
```

No copies — but look at how you call it:

```python
sum_list([5, 3, 8], 0)  # What's the 0 for??
```

Problem: The caller has to know to pass 0.
That's an **implementation detail** — it shouldn't be their problem.

Solution: Helper Functions

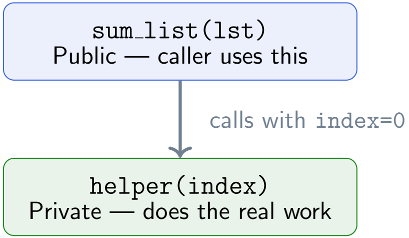
Wrap the index version inside a **clean public function**:

```
sum_list([5, 3, 8])        # Clean! No index needed.
```

Solution: Helper Functions

Wrap the index version inside a **clean public function**:

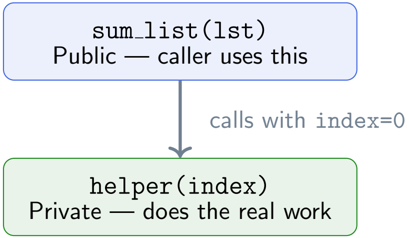
```
sum_list([5, 3, 8])        # Clean! No index needed.
```



Solution: Helper Functions

Wrap the index version inside a **clean public function**:

```
sum_list([5, 3, 8])        # Clean! No index needed.
```



The **helper** is a nested function that has access to `lst` and carries the `index` the caller never sees.

Helper Function: Solution

```python
def sum_list(lst):
    """Clean public interface."""
    def helper(index):
        if index == len(lst):
            return 0
        return lst[index] + helper(index + 1)

    return helper(0)
```

Helper Function: Solution

```python
def sum_list(lst):
    """Clean public interface."""
    def helper(index):
        if index == len(lst):
            return 0
        return lst[index] + helper(index + 1)

    return helper(0)
```

The user calls `sum_list([1,2,3])` — clean and simple.
The `index` is an **implementation detail** hidden inside.

Another Use Case: Binary Search

Helpers aren't just about slicing. Sometimes the recursion needs **extra parameters** the caller shouldn't know about.

Ugly: caller has to pass search boundaries:

```
binary_search([1,3,5,7,9], 7, 0, 4)  # low? high?
```

Another Use Case: Binary Search

Helpers aren't just about slicing. Sometimes the recursion needs **extra parameters** the caller shouldn't know about.

Ugly: caller has to pass search boundaries:

```
binary_search([1,3,5,7,9], 7, 0, 4)  # low? high?
```

Clean: hide boundaries inside a helper:

```
binary_search([1,3,5,7,9], 7)  # Just list + target!
```

Another Use Case: Binary Search

Helpers aren't just about slicing. Sometimes the recursion needs **extra parameters** the caller shouldn't know about.

Ugly: caller has to pass search boundaries:

```
binary_search([1,3,5,7,9], 7, 0, 4)  # low? high?
```

Clean: hide boundaries inside a helper:

```
binary_search([1,3,5,7,9], 7)  # Just list + target!
```

The helper carries `low` and `high` —
the caller never thinks about them.

Binary Search with Helper

(we'll cover this in detail next lecture)

```python
def binary_search(lst, target):
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2
        if lst[mid] == target:
            return True
        elif lst[mid] < target:
            return helper(mid + 1, high)
        else:
            return helper(low, mid - 1)

    return helper(0, len(lst) - 1)
```

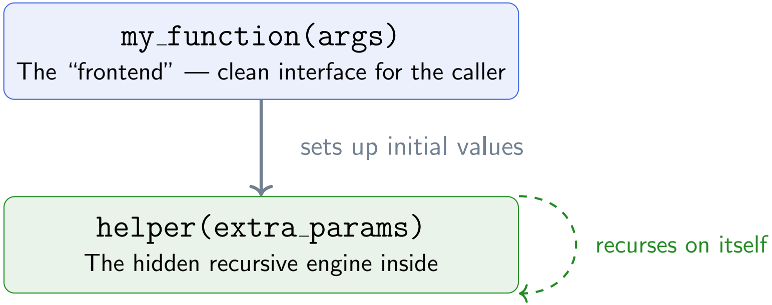
Binary Search with Helper

(we'll cover this in detail next lecture)

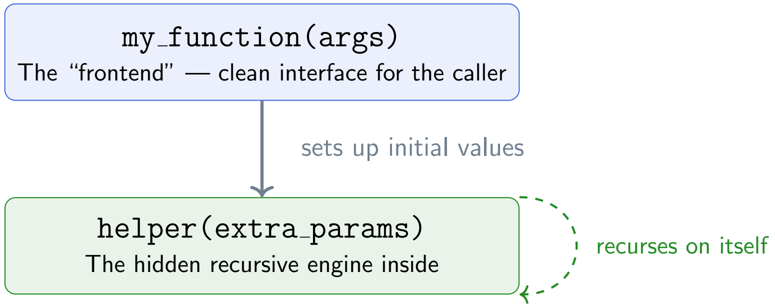
```python
def binary_search(lst, target):
    def helper(low, high):
        if low > high:
            return False
        mid = (low + high) // 2
        if lst[mid] == target:
            return True
        elif lst[mid] < target:
            return helper(mid + 1, high)
        else:
            return helper(low, mid - 1)

    return helper(0, len(lst) - 1)
```

Pattern: outer function sets up initial values, inner `helper` does the recursive work.

The Helper Pattern


my_function(args)
The “frontend” — clean interface for the caller
sets up initial values
helper(extra_params)
The hidden recursive engine inside
recurses on itself

The Helper Pattern


my_function(args)
The "frontend" — clean interface for the caller
sets up initial values
helper(extra_params)
The hidden recursive engine inside
recurses on itself

Use a helper when:

- The recursion needs parameters the caller shouldn't provide
- You want to avoid expensive slicing with an index

Capstone: Flatten Nested Lists

This is where recursion **really shines** — a loop can't do this!

A list can contain numbers **or other lists**, nested arbitrarily deep:

```
[1, [2, 3], [4, [5, 6]]]    →    [1, 2, 3, 4, 5, 6]

[1, [2, [3, [4, [5]]]]]     →    [1, 2, 3, 4, 5]
```

Capstone: Flatten Nested Lists

This is where recursion **really shines** — a loop can't do this!

A list can contain numbers **or other lists**, nested arbitrarily deep:

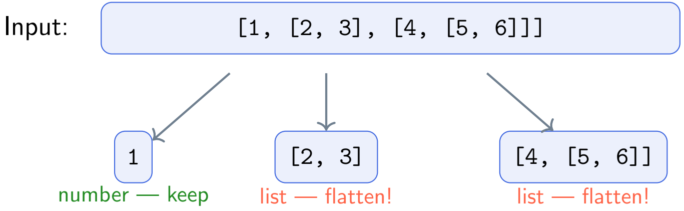
```
[1, [2, 3], [4, [5, 6]]]   →   [1, 2, 3, 4, 5, 6]

[1, [2, [3, [4, [5]]]]]    →   [1, 2, 3, 4, 5]
```

Why can't a loop handle this?
You don't know how many levels deep the nesting goes!

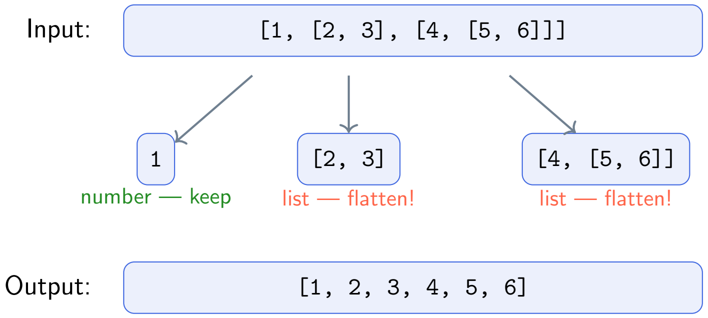
Think: What Does Flatten Do at Each Element?

Input: `[1, [2, 3], [4, [5, 6]]]`

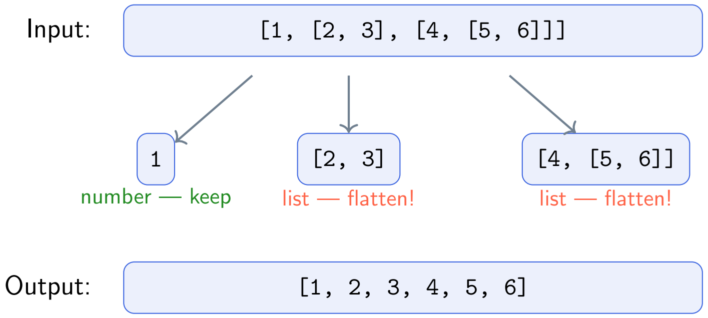
Think: What Does Flatten Do at Each Element?



Think: What Does Flatten Do at Each Element?



Think: What Does Flatten Do at Each Element?



For each item: is it a **list**? Recurse. Is it a **number**? Keep it.

Same idea as file folders — recursion handles **arbitrary depth** naturally.

Solution: Flatten

```
def flatten(lst):
    # W: empty list
    if lst == []:
        return []

    # I: look at first item
    first = lst[0]
    rest = flatten(lst[1:])

    # S+E: recurse or keep
    if isinstance(first, list):
        return flatten(first) + rest
    else:
        return [first] + rest
```

Solution: Flatten

```python
def flatten(lst):
    # W: empty list
    if lst == []:
        return []

    # I: look at first item
    first = lst[0]
    rest = flatten(lst[1:])

    # S+E: recurse or keep
    if isinstance(first, list):
        return flatten(first) + rest
    else:
        return [first] + rest
```

WISE:

W: [] → return []

I: Split into first + rest

S: Flatten rest; if first is a list, flatten it too

E: Concatenate results

Trace: flatten([1, [2, [3]]])

```
flatten([1, [2, [3]]])
```

Trace: flatten([1, [2, [3]]])

```
flatten([1, [2, [3]]])
  1 is a number → [1] + flatten([[2, [3]]])
```

Trace: flatten([1, [2, [3]]])

```
flatten([1, [2, [3]]])
 1 is a number → [1] + flatten([[2, [3]]])
  [2,[3]] is a list → flatten([2,[3]]) + flatten([])
```

Trace: flatten([1, [2, [3]]])

```
flatten([1, [2, [3]]])
 1 is a number → [1] + flatten([[2, [3]]])
  [2,[3]] is a list → flatten([2,[3]]) + flatten([])
   2 is a number → [2] + flatten([[3]])
```

Trace: flatten([1, [2, [3]]])

```
flatten([1, [2, [3]]])
 1 is a number → [1] + flatten([[2, [3]]])
  [2,[3]] is a list → flatten([2,[3]]) + flatten([])
   2 is a number → [2] + flatten([[3]])
    [3] is a list → flatten([3]) → [3]
```

Trace: flatten([1, [2, [3]]])

```
flatten([1, [2, [3]]])
 1 is a number → [1] + flatten([[2, [3]]])
  [2,[3]] is a list → flatten([2,[3]]) + flatten([])
   2 is a number → [2] + flatten([[3]])
    [3] is a list → flatten([3]) → [3]
= [1, 2, 3]
```

All flattened!

Lists get opened up recursively; numbers get kept. Depth doesn't matter!

Wrap-Up

Key Takeaways

WISE: What (base) – If (shrink) – Solve (recurse) – Extend (combine)

Key Takeaways

WISE: What (base) – If (shrink) – Solve (recurse) – Extend (combine)

Big Three Bugs: Missing return, wrong base case, not shrinking

Key Takeaways

WISE: What (base) – If (shrink) – Solve (recurse) – Extend (combine)

Big Three Bugs: Missing return, wrong base case, not shrinking

Design first: Break down the problem before writing code

Key Takeaways

WISE: What (base) – If (shrink) – Solve (recurse) – Extend (combine)

Big Three Bugs: Missing return, wrong base case, not shrinking

Design first: Break down the problem before writing code

Helper functions hide extra parameters from the caller

Homework

1. `remove_char(s, c)` — remove all occurrences of a character
   `remove_char("hello", "l")` → `"heo"`
2. `all_digits_even(n)` — True if every digit is even
   `all_digits_even(2468)` → `True`
3. Rewrite `count_vowels` with a **helper function** (index instead of slicing)
4. **Challenge:** `is_sorted(lst)` — check if list is in ascending order

Next Week Preview

Divide & Conquer!

- What happens when a function calls itself **TWICE**?
- Split big problems **in half**
- **Binary search** — find any item in a million with just 20 checks!

Teaser: What if we cut the problem **in HALF**?

Questions?

Practice on `pythontutor.com`