

Recursion

Kelly Rivers and Stephanie Rosenthal

15-110 Fall 2019

Announcements

- Homework 3 Check-in 2 is due Monday at 12-noon!
- Homework 3 full is also due in 2 Mondays.

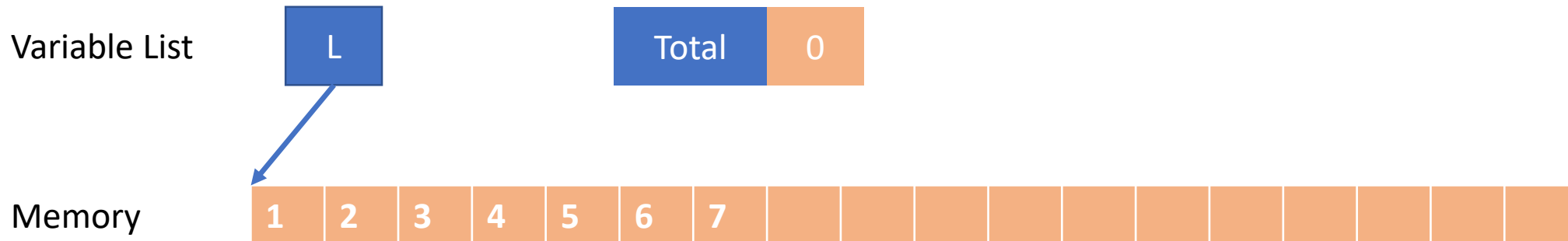
Learning Objectives

- To understand the purpose and definition of recursion
- To trace recursive algorithms
- To create simple recursive functions

Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

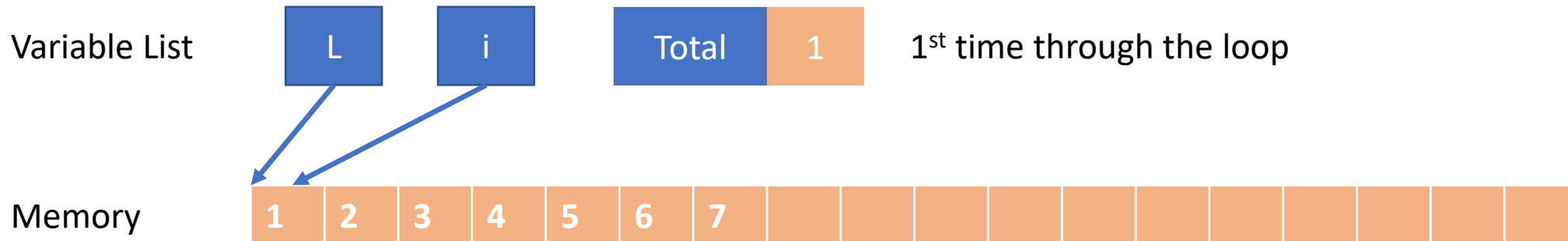
```
total = 0
for i in L:
    total = total + i
```



Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

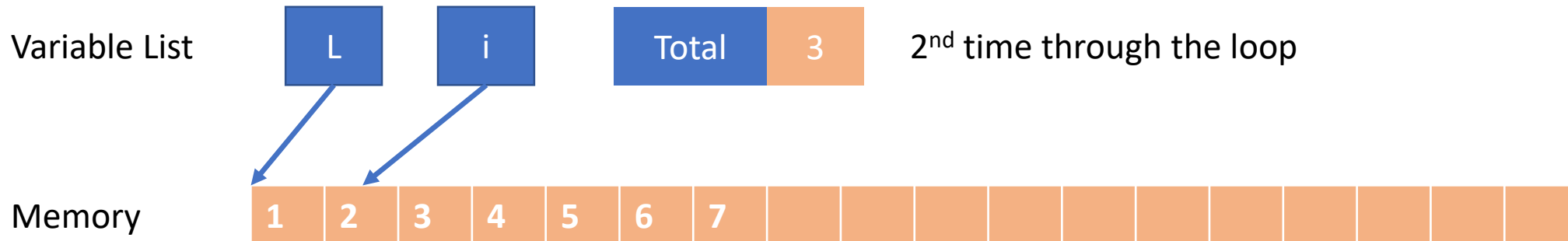
```
total = 0
for i in L:
    total = total + i
```



Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

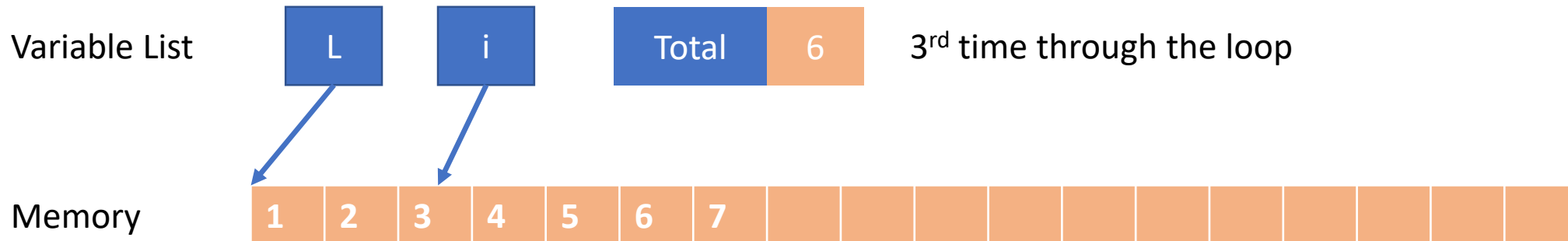
```
total = 0
for i in L:
    total = total + i
```



Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

```
total = 0
for i in L:
    total = total + i
```



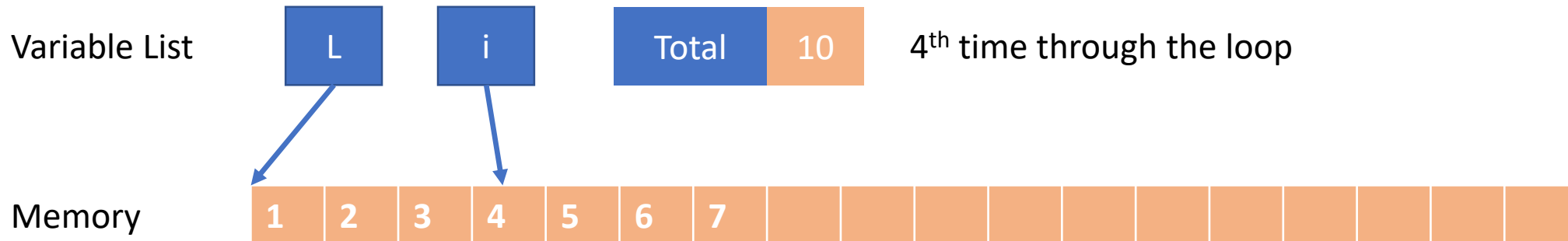
Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

```
total = 0
```

```
for i in L:
```

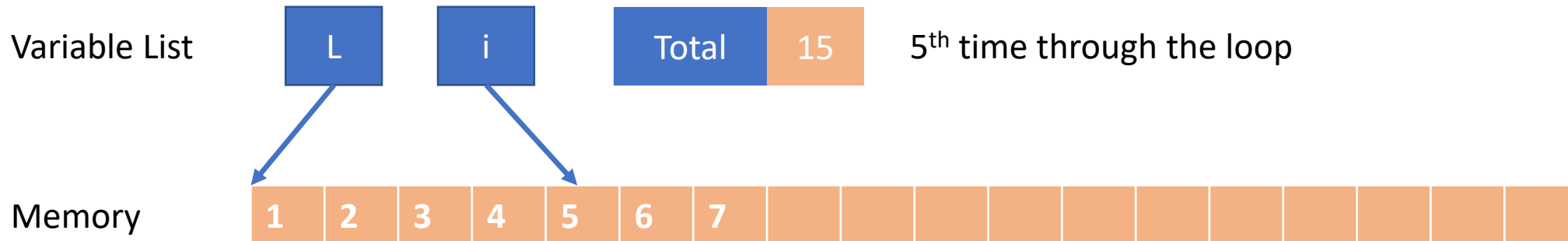
```
total = total + i
```



Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

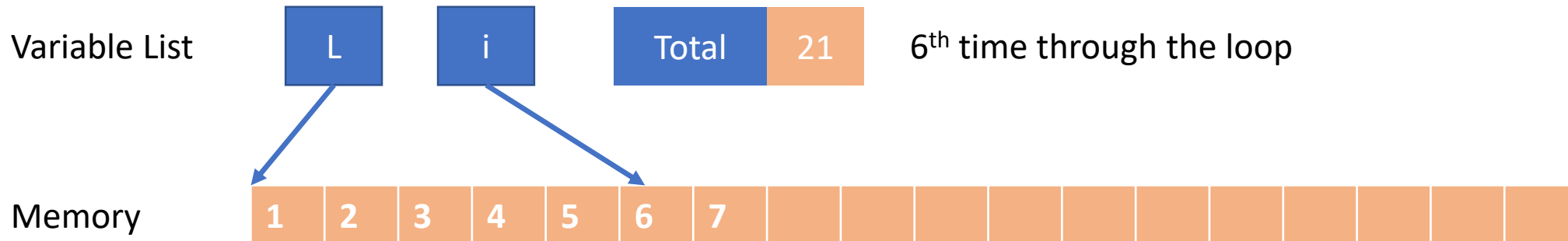
```
total = 0
for i in L:
    total = total + i
```



Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

```
total = 0
for i in L:
    total = total + i
```



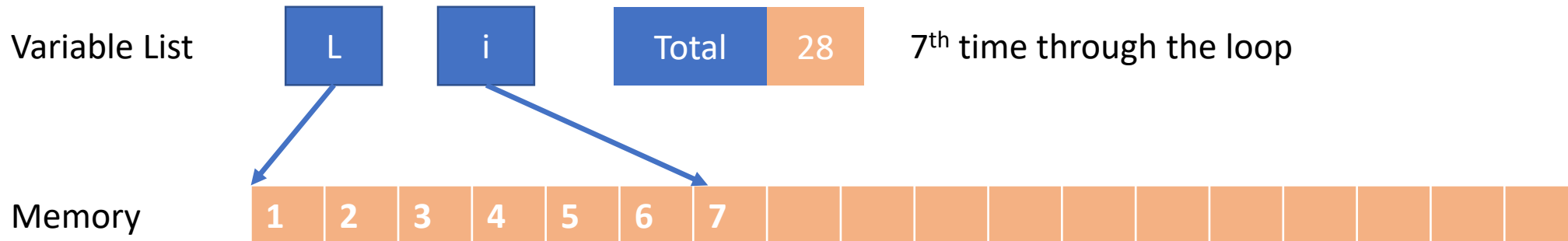
Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

```
total = 0
```

```
for i in L:
```

```
total = total + i
```



Iteration

Loops iterate through sequences one item at a time and we use other variables to compute information as the loop iterates.

```
total = 0
for i in L:
    total = total + i
```

Recursion

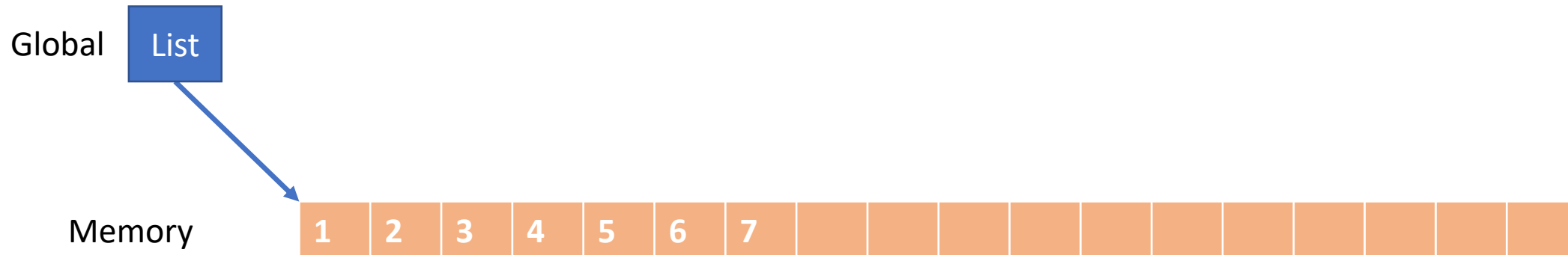
Recursion is another way to compute the same information by

- 1) breaking down the problem into smaller pieces
- 2) calling itself (the same function) on each smaller piece
- 3) combining the smaller answers back together in some way

Recursion

Example:

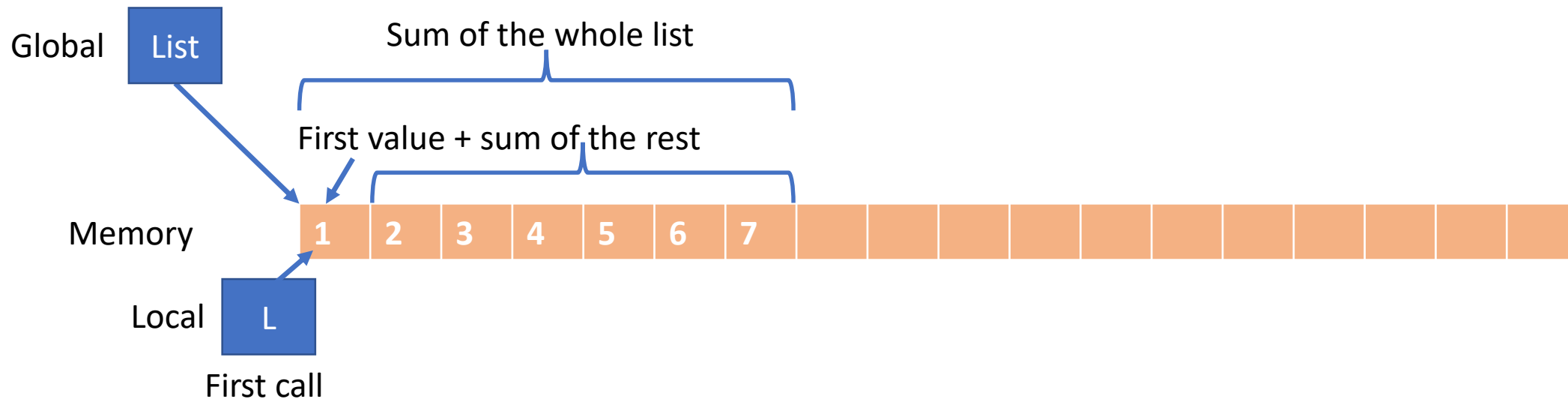
```
def recursiveAdd(L):  
    return L[0]+recursiveAdd(L[1:])
```



Recursion

Example:

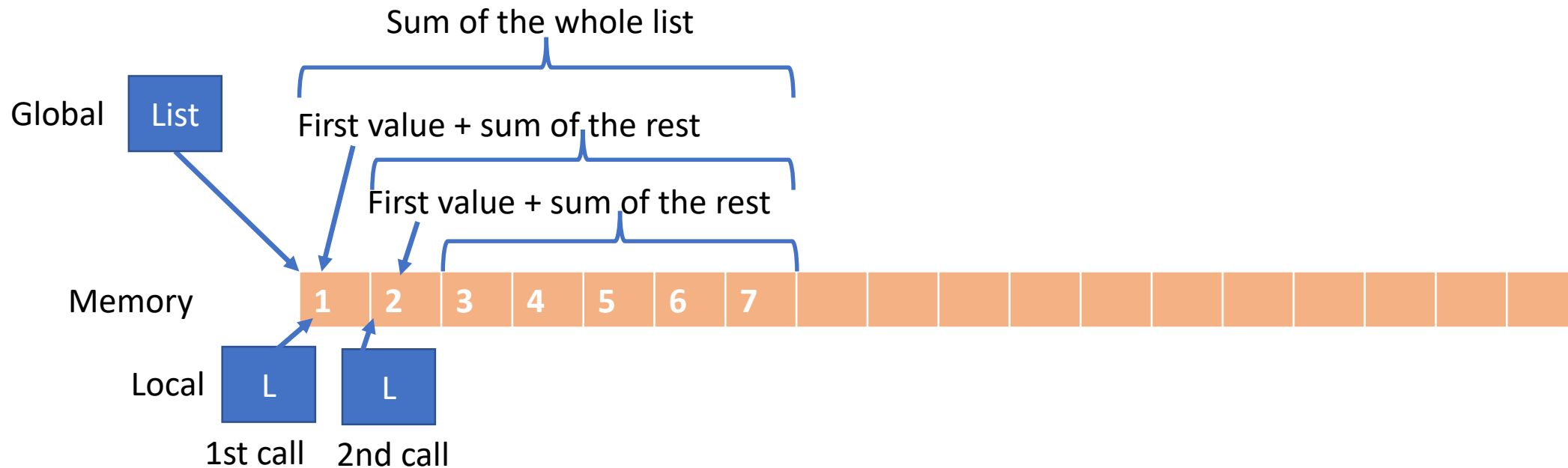
```
def recursiveAdd(L):  
    return L[0]+recursiveAdd(L[1:])
```



Recursion

Example:

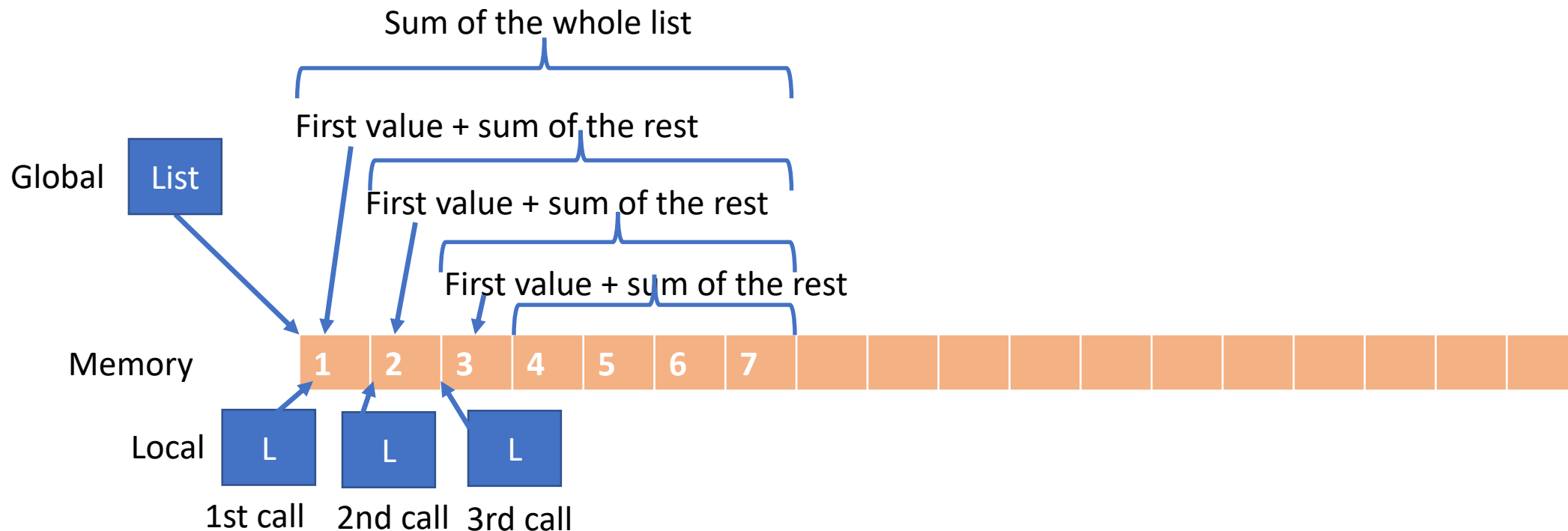
```
def recursiveAdd(L):  
    return L[0]+recursiveAdd(L[1:])
```



Recursion

Example:

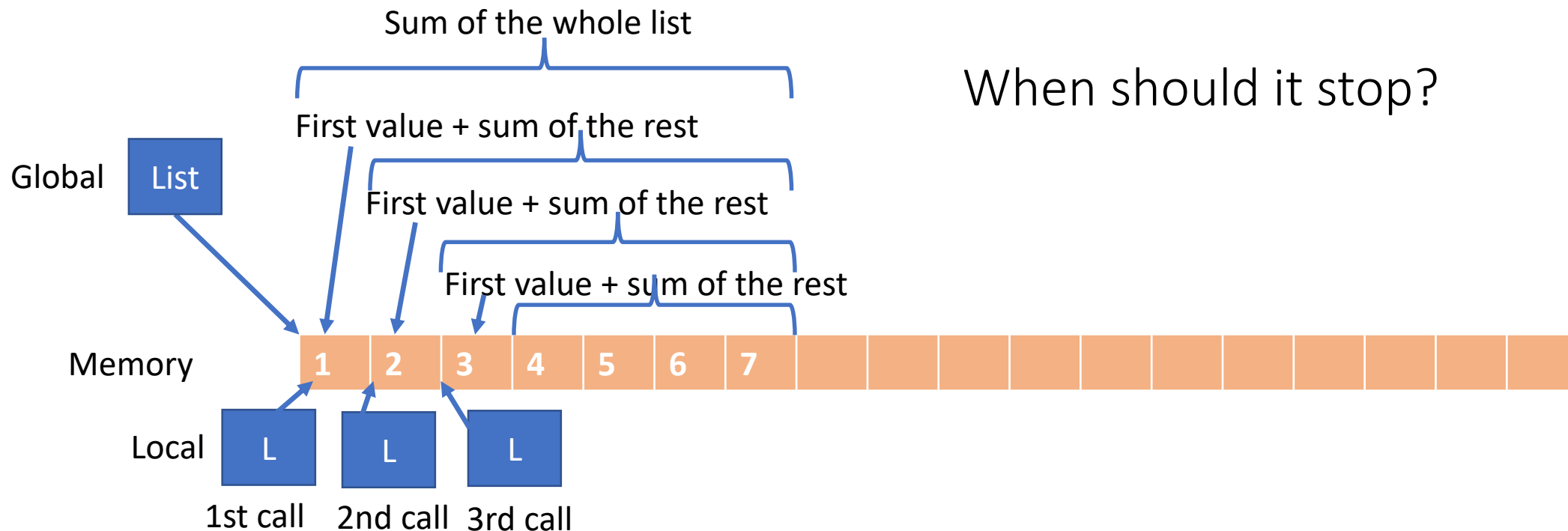
```
def recursiveAdd(L):  
    return L[0]+recursiveAdd(L[1:])
```



Recursion

Example:

```
def recursiveAdd(L):  
    return L[0]+recursiveAdd(L[1:])
```



Recursion

Recursion is another way to compute the same information by

- 1) breaking down the problem into smaller pieces
- 2) calling itself (the same function) on each smaller piece
- 3) combining the smaller answers back together in some way

We must define a **base case** - the value to return for the smallest input
it is NOT recursive because we can't break down the problem further.
Note: every possible input must eventually breakdown to this value(s)
Otherwise the recursion won't stop.

Recursion

Recursion is another way to compute the same information by

Inductive/Recursive case/step:

- 1) breaking down the problem into smaller pieces
- 2) calling itself (the same function) on each smaller piece
- 3) combining the smaller answers back together in some way

We must define a **base case** - the value to return for the smallest input

it is **NOT recursive** because we can't break down the problem further.

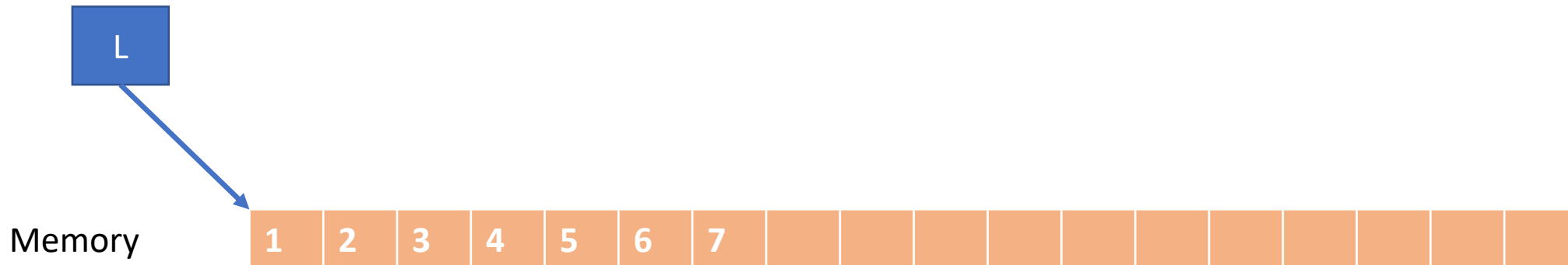
Note: every possible input must eventually breakdown to this value(s)

Otherwise the recursion won't stop.

Base Case: Piazza Poll

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return ?  
    return L[0]+recursiveAdd(L[1:])
```



What happens if L is Empty?

What is total in the iterative case?

```
def iterativeAdd(L):  
    total = 0  
    for i in L:  
        total = total + i
```

Answer: The total of an empty list is 0

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

Memory



Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

```
recursiveAdd( 

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

 )
```


Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

```
recursiveAdd( 

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

 )
```

```
Return 1+
```

Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

recursiveAdd(

1	2	3
---	---	---

)

Return 1+

recursiveAdd(

2	3
---	---

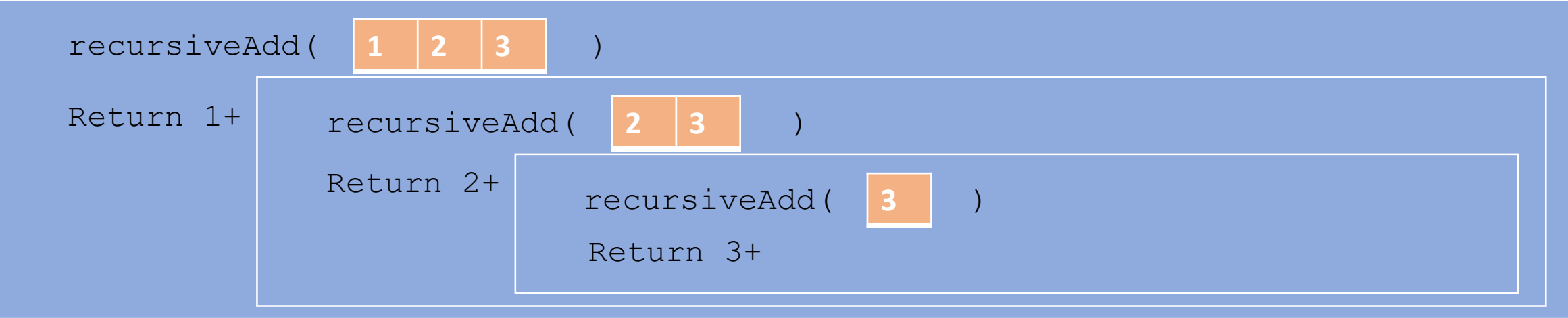
)

Return 2+

Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```



The diagram illustrates the recursive calls for the function `recursiveAdd` with the list `[1, 2, 3]`. It uses nested boxes to show the call stack. The outermost box represents the initial call with `[1, 2, 3]`. Inside it, a box for the second call with `[2, 3]` is shown, and inside that, a box for the third call with `[3]`. Each box shows the return value being calculated at that step: `Return 1+` for the first call, `Return 2+` for the second, and `Return 3+` for the third. The numbers 1, 2, and 3 in the function arguments are highlighted in orange boxes.

```
recursiveAdd( 1 2 3 )  
Return 1+ recursiveAdd( 2 3 )  
Return 2+ recursiveAdd( 3 )  
Return 3+
```

Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

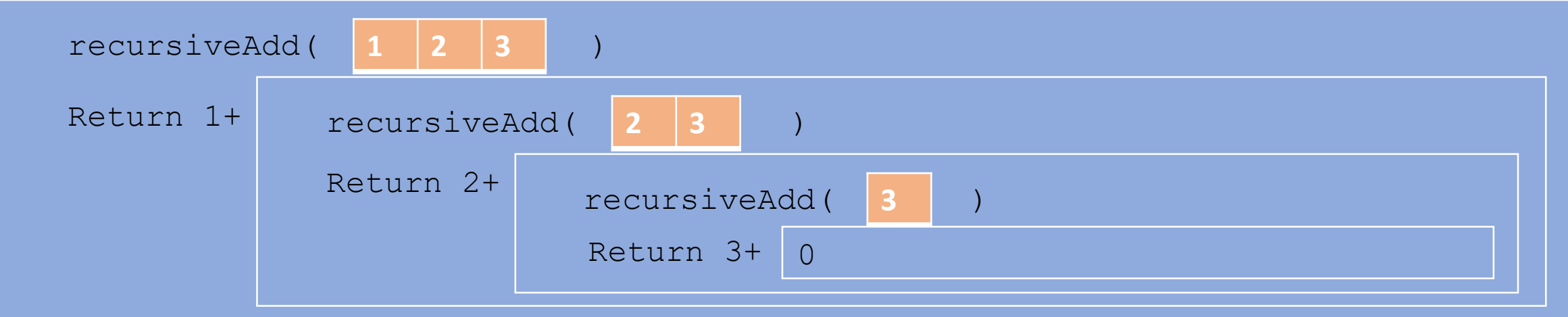
The diagram illustrates the recursive process of calculating the sum of the list [1, 2, 3] using the recursiveAdd function. It consists of four nested rectangular boxes, each representing a function call. The outermost box is labeled 'recursiveAdd(' and contains three orange boxes with the numbers 1, 2, and 3, followed by a closing parenthesis. To the left of this box is the text 'Return 1+'. Inside this box is a second box labeled 'recursiveAdd(' with two orange boxes containing 2 and 3, followed by a closing parenthesis. To the left of this box is the text 'Return 2+'. Inside the second box is a third box labeled 'recursiveAdd(' with one orange box containing 3, followed by a closing parenthesis. To the left of this box is the text 'Return 3+'. Inside the third box is a fourth box labeled 'recursiveAdd(' with an empty list '[]' followed by a closing parenthesis. To the left of this box is the text 'Return 0'. The boxes are nested, showing the sequence of calls from the initial call to the base case and back.

```
recursiveAdd( 1 2 3 )  
Return 1+ recursiveAdd( 2 3 )  
Return 2+ recursiveAdd( 3 )  
Return 3+ recursiveAdd( [] ) Return 0
```

Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```



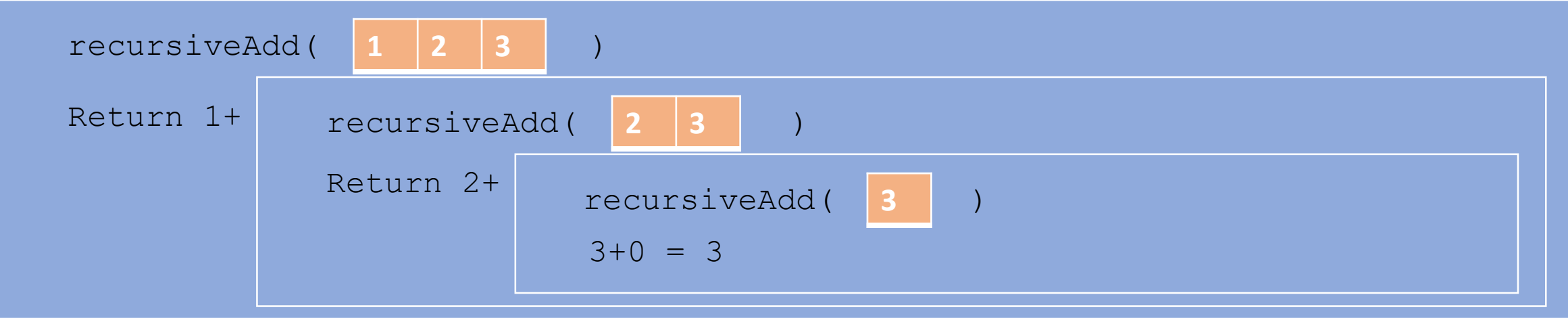
The diagram illustrates the recursive process of calculating the sum of the list [1, 2, 3]. It consists of three nested rectangular boxes, each representing a function call. The outermost box is labeled 'recursiveAdd(' followed by three orange boxes containing the numbers 1, 2, and 3, and a closing parenthesis. To the left of this box is the text 'Return 1+'. Inside this box is a second, smaller box labeled 'recursiveAdd(' followed by two orange boxes containing the numbers 2 and 3, and a closing parenthesis. To the left of this second box is the text 'Return 2+'. Inside the second box is a third, even smaller box labeled 'recursiveAdd(' followed by one orange box containing the number 3, and a closing parenthesis. To the left of this third box is the text 'Return 3+'. Inside the third box is a fourth, the smallest box, which contains the number 0. This visualizes the base case of the recursion where an empty list returns 0, and how the return values are propagated back up the call stack.

```
recursiveAdd( 1 2 3 )  
Return 1+  
    recursiveAdd( 2 3 )  
    Return 2+  
        recursiveAdd( 3 )  
        Return 3+ 0
```

Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```



The diagram illustrates the recursive process of adding the elements of the list [1, 2, 3]. It consists of three nested rectangular boxes, each representing a function call. The outermost box shows the initial call with the list [1, 2, 3]. Inside it, a box shows the recursive call with the list [2, 3]. Inside that, a box shows the recursive call with the list [3]. The innermost box shows the base case where the list is empty, and the result 3 is calculated as 3 + 0. The return values are propagated back up the stack: the innermost call returns 3, the middle call returns 2 + 3 = 5, and the outermost call returns 1 + 5 = 6.

```
recursiveAdd( [ 1 2 3 ] )  
Return 1+  
    recursiveAdd( [ 2 3 ] )  
    Return 2+  
        recursiveAdd( [ 3 ] )  
        3+0 = 3
```

Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

recursiveAdd(

1	2	3
---	---	---

)

Return 1+

recursiveAdd(

2	3
---	---

)

2+3 = 5

Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

```
recursiveAdd( 

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

 )
```

```
Return 1+5 = 6
```


Another Picture of What is Happening

Example:

```
def recursiveAdd(L):  
    if L == []:  
        return 0  
    return L[0]+recursiveAdd(L[1:])
```

```
recursiveAdd( 

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
|---|---|---|

 )
```

6

Recursion

General Form:

```
def recursiveFunction(X):  
    if X == ? #base case is some smallest case  
        return _____ #something not recursive  
    else:  
        #call itself on a smaller piece  
        result = recursiveFunction(Xsmaller)  
        #combine with some value and return  
        return _____
```

Example: Count Elements in List

```
def recursiveCount(L):  
    if L == 0 #base case is some smallest case  
        return 0 #something not recursive  
    else:  
        #call itself on a smaller piece  
        result = recursiveCount(L[1:])  
        #combine and return  
        return 1+result
```

Trace the function for L=["dog","cat","mouse"]

```
def recursiveCount(L):  
    if L == 0 #base case is some smallest case  
        return 0 #something not recursive  
    else:  
        #call itself on a smaller piece  
        result = recursiveCount(L[1:])  
        #combine and return  
        return 1+result
```

Write: Factorial ($n! = n * n-1 * n-2 * \dots * 2 * 1$)

Example: Factorial ($n! = n * n-1 * n-2 * \dots * 2 * 1$)

```
def recursiveFactorial(n):  
    if n == 0 #base case is some smallest case  
        return 1 #something not recursive  
    else:  
        #call itself on a smaller piece  
        result = recursiveFactorial(n-1)  
        #combine and return  
        return n*result
```

Example: Exponentiation

```
def recursivePower(base, power):  
    if power == 0 #base case is some smallest case  
        return 1 #something not recursive  
    else:  
        #call itself on a smaller piece  
        result = recursivePower(base, power-1)  
        #combine and return  
        return base*result
```

Example: Counting Vowels

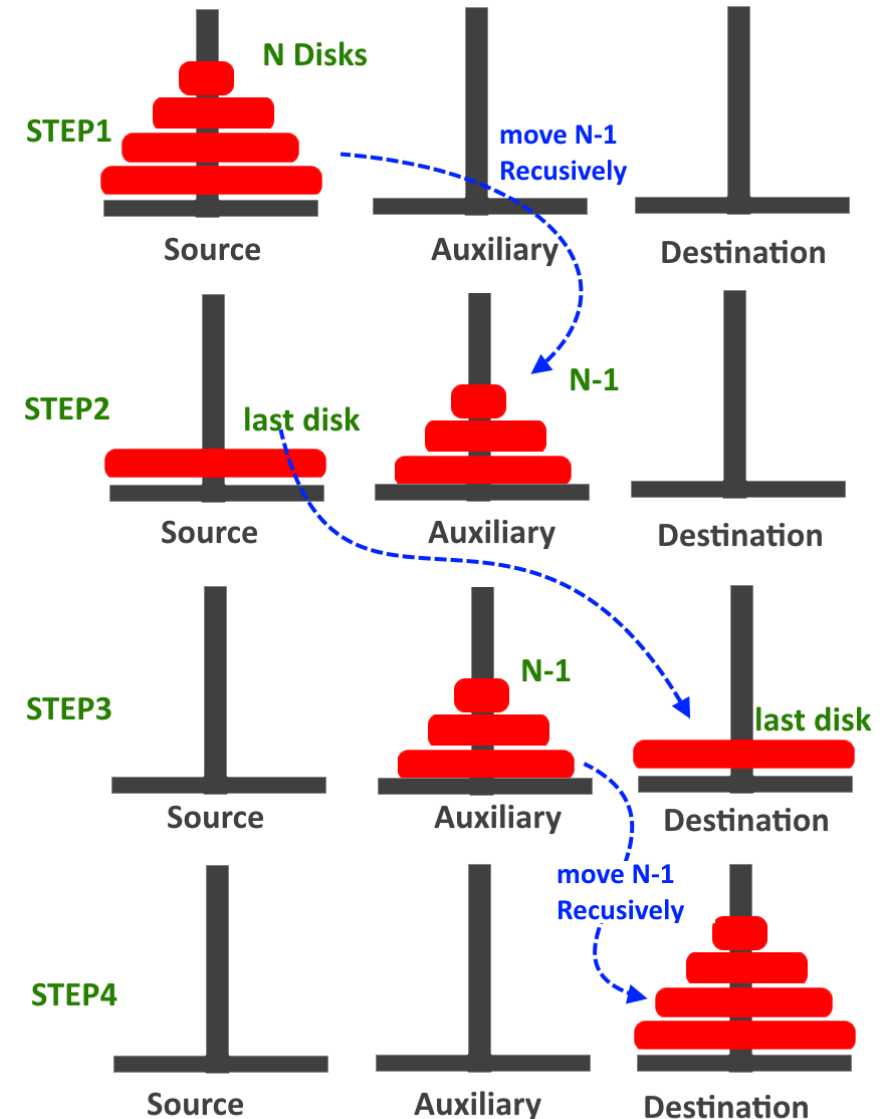
```
def vowelCount(s):  
    if (s == ''):  
        return 0  
    elif (s[0] in 'AEIOUaeiou'):  
        return 1 + vowelCount(s[1:])  
    else:  
        return vowelCount(s[1:])  
  
print(vowelCount("Isn't this amazing?")) # 5 (Iiaai)
```


Example: Fibonacci Numbers: $F(n) = F(n-1) + F(n-2)$

```
def fib(n):  
    if (n == 0):  
        return 1  
    elif (n == 1):  
        return 1  
    else:  
        return fib(n-1) + fib(n-2)  
  
print(fib(5))
```

Example: Tower of Hanoi

```
def move(n, source, dest, aux):  
    if (n == 1):  
        print((source, dest), end="")  
    else:  
        move(n-1, source, aux, dest)  
        move(1, source, dest, aux)  
        move(n-1, aux, dest, source)  
  
def hanoi(n):  
    print("Towers of Hanoi with n =", n)  
    move(n, 0, 1, 2)
```



Recursion Takeaways

- Alternative to iteration
- Recursive problems have base cases and inductive/recursive cases
- Need to break the problem into smaller pieces at each step
- Make sure all breakdowns end in the base case (otherwise infinite loop)