

# Recursive Function Call Stack

Supplement to Recursion lecture

# Tracing the Call Stack

Start with the original function call on the stack.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

```
addCards([5, 2, 7, 3])
```

## Call Stack

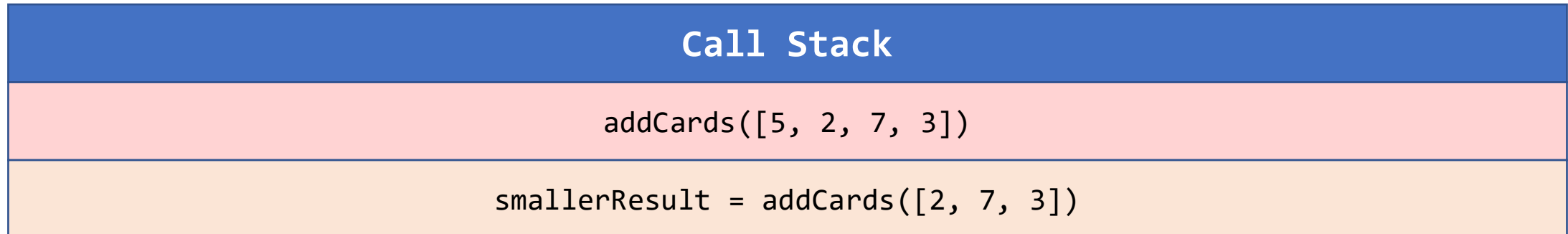
addCards([5, 2, 7, 3])

# Tracing the Call Stack

We go to the recursive case and set up a local variable `smallerProblem`. Then call `addCards` again on that variable, putting another level on the stack.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

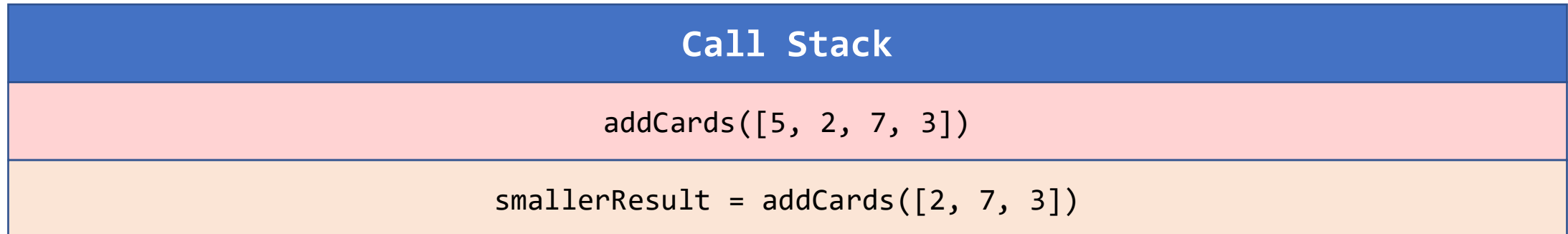
```
addCards([5, 2, 7, 3])
```



# Tracing the Call Stack

When we run through `addCards` a second time, there's a **new local state**. `cards` is now `[2, 7, 3]`. `smallerProblem` is now `[7, 3]`.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult  
  
addCards([5, 2, 7, 3])
```



# Tracing the Call Stack

Call `addCards` again, this time on `[7, 3]`. Note that the call stack helps us keep track of **all** previous calls.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult  
  
addCards([5, 2, 7, 3])
```

Call Stack
<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult = addCards([2, 7, 3])</code>
<code>smallerResult = addCards([7, 3])</code>

# Tracing the Call Stack

Now we run the function with `cards` set to `[7, 3]`. `smallerProblem` becomes `[3]`; we call the function again.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult  
  
addCards([5, 2, 7, 3])
```

Call Stack
<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult = addCards([2, 7, 3])</code>
<code>smallerResult = addCards([7, 3])</code>
<code>smallerResult = addCards([3])</code>

# Tracing the Call Stack

Run the function with `cards` set to `[3]`.  
`smallerProblem` becomes `[]`; run the function again.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult  
  
addCards([5, 2, 7, 3])
```

Call Stack
<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult = addCards([2, 7, 3])</code>
<code>smallerResult = addCards([7, 3])</code>
<code>smallerResult = addCards([3])</code>
<code>smallerResult = addCards([])</code>

# Tracing the Call Stack

Now we finally reach the base case. `addCards([])` returns `0` immediately, so `0` takes the place of the function call on the bottom level of the stack (in Call #5).

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

```
addCards([5, 2, 7, 3])
```

Call Stack
<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult = addCards([2, 7, 3])</code>
<code>smallerResult = addCards([7, 3])</code>
<code>smallerResult = addCards([3])</code>
<code>smallerResult = 0</code>



# Tracing the Call Stack

How does Python know what to do next? It remembers the **local state** of each level of the stack! On the green level, it knows that `cards` is `[3]` and `smallerResult` is now `0`.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

```
addCards([5, 2, 7, 3])
```

Call Stack
<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult = addCards([2, 7, 3])</code>
<code>smallerResult = addCards([7, 3])</code>
<code>smallerResult = addCards([3])</code>

# Tracing the Call Stack

Add `3 + 0` to get `3`; this can be sent back as the returned value to the previous level of the stack (in Call #3).

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

```
addCards([5, 2, 7, 3])
```

Call Stack
<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult = addCards([2, 7, 3])</code>
<code>smallerResult = addCards([7, 3])</code>
<code>smallerResult = 3</code>

# Tracing the Call Stack

At this level, `cards` is `[7, 3]` and `smallerResult` is `3`. `7 + 3` gives us a returned value of `10`.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

```
addCards([5, 2, 7, 3])
```

Call Stack
<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult = addCards([2, 7, 3])</code>
<code>smallerResult = 10</code>

# Tracing the Call Stack

Now `cards` is `[2, 7, 3]` and `smallerResult` is `10`. Add `2 + 10` to get a returned value of `12`.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

```
addCards([5, 2, 7, 3])
```

Call Stack	
	<code>addCards([5, 2, 7, 3])</code>
<code>smallerResult =</code>	<code>12</code>

# Tracing the Call Stack

We've finally reached the original call. `cards` is `[5, 2, 7, 3]`, and `smallerResult` is `12`. Add `5 + 12` to get `17`, which is returned at the top level as the final result of the original function call.

```
def addCards(cards):  
    if cards == []:  
        return 0  
    else:  
        smallerProblem = cards[1:]  
        smallerResult = addCards(smallerProblem)  
        return cards[0] + smallerResult
```

```
addCards([5, 2, 7, 3])
```

