LOOP COMPOSITION

An Introduction to Computer Science

Let's learn about Loop Composition

Indirectly Looping in Functions

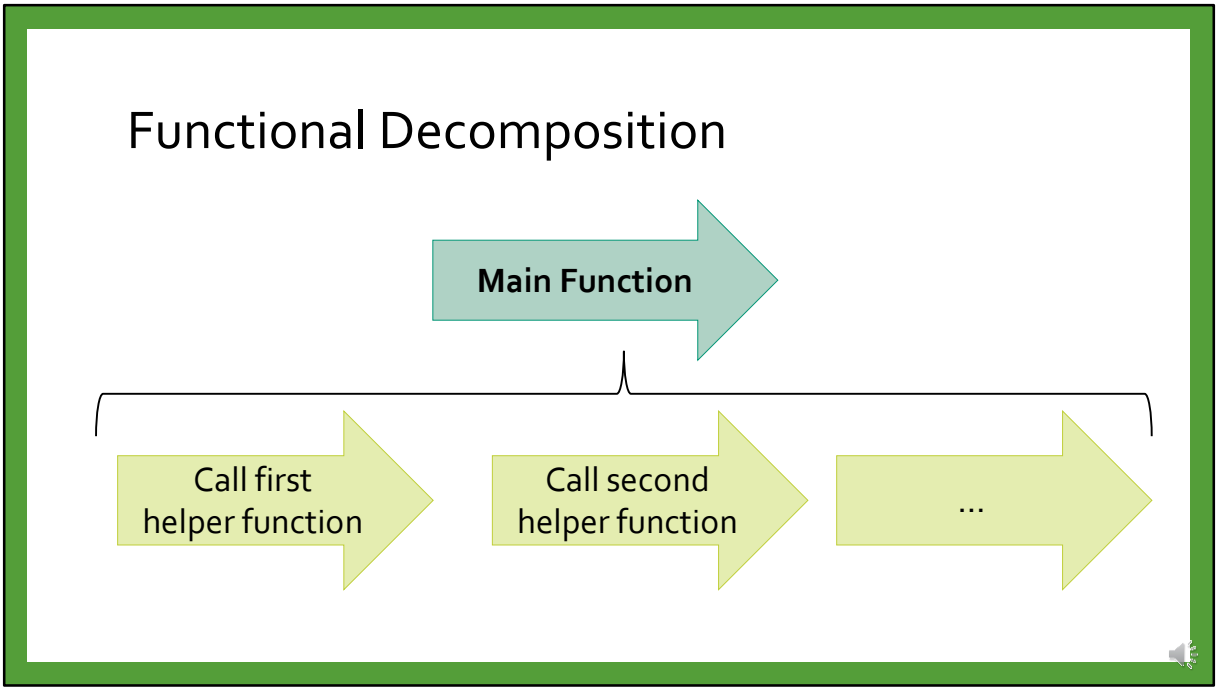
Looping Version

```
def average(nums: [int]) -> float:
    summed = 0
    counted = 0
    for num in nums:
        summed = summed + num
        counted = counted + 1
    return summed / counted
```

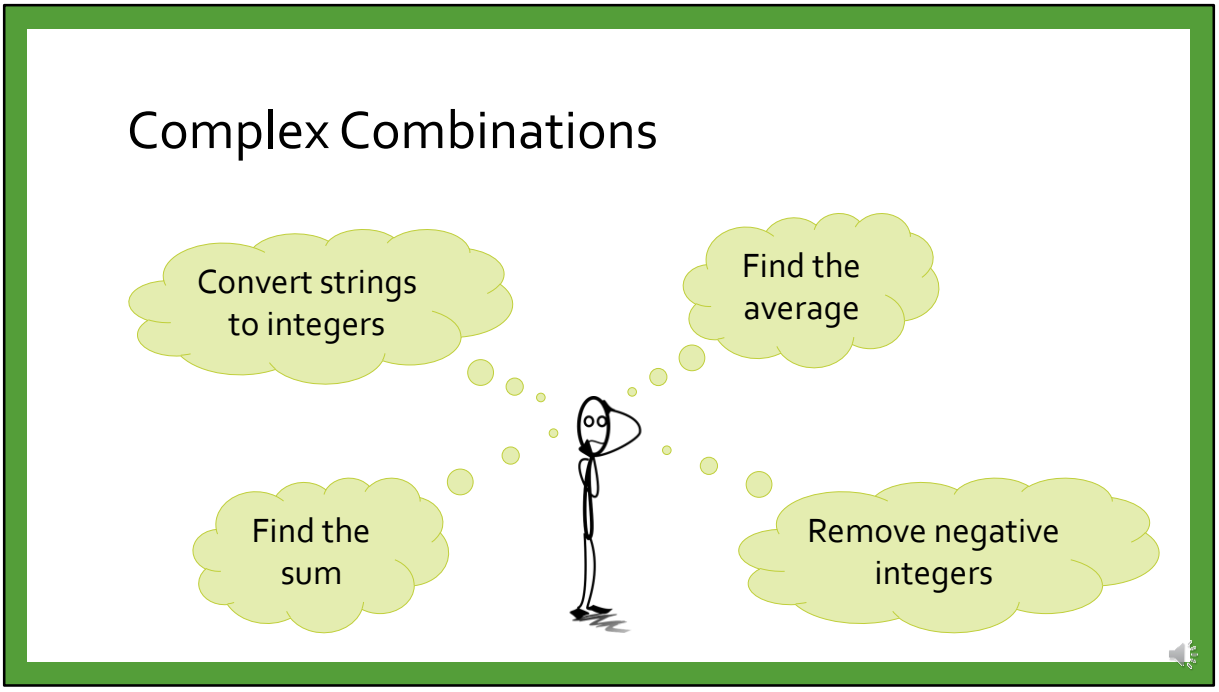
Indirectly Looping Version

```
def sum(nums: [int]) -> int:
    summed = 0
    for num in nums:
        summed = summed + num
    return summed

def count(nums: [int]) -> int:
    counted = 0
    for num in nums:
        counted = counted + 1
    return counted

def average(nums: [int]) -> float:
    return sum(nums)/count(nums)
```

You can calculate the mean, or average, of a list by adding together the elements and dividing by the number of elements.
Finding the average is a common operation for a list.
You could write this function by creating a loop that combines the Sum and Count patterns.
Alternatively, you could define functions to sum and count, and then use those functions in your Average function.
This version does not require a loop directly in the Average function, because the loop happens indirectly in a helper function.



Earlier, we learned to break up complex functions into multiple parts using functional decomposition.
With loops, we finally see why that becomes necessary.
The control flow for a function with a loop is complicated.
When you start nesting loops and if statements inside of functions, mistakes are likely.
Functional decomposition allows us to tame that complexity by focusing on one conceptual step at a time.



Let's say that we wanted to add up all the negative numbers in a list.
You could write a for loop that combines the Filter pattern and the Count pattern.
You might even succeed.
But what if you also needed to convert all the elements from strings to integers first?
What if you needed to also find the average?
Suddenly, combining all these patterns becomes difficult.

Composing Loop Patterns

```
def map_strs_to_ints(numbers: [str]) -> [int]:
    ...
def remove_negatives(numbers: [int]) -> [int]:
    ...
def count(numbers: [int]) -> int:
    ...
def sum(numbers: [int]) -> int:
    ...

def average_positives(numbers: [str]) -> float:
    positives = remove_negatives(map_strs_to_ints(numbers))
    average = sum(positives) / count(positives)
    return average
```

Instead, let's focus on defining functions to solve each part of the puzzle in turn.
We define a map_strs_to_ints function that maps string conversion over a list of strings.
We define a remove_negatives function that filters out negative integers.
We define a count and sum function that consume lists of integers to accumulate.
Then, we can compose those functions as needed into an average_positives function.
This may seem like a lot more work - suddenly we have to define and test 5 functions instead of 1.
But the advantage is that each function is relatively easy to define and test.
If you make a mistake, you will be able to track it down very quickly.