Object-Oriented Programming

Kenjiro Taura

2024/04/28

Contents

What is Functional Programming?

1st attempt

> … *functional programming is a programming paradigm where programs are constructed by applying and composing functions.*
>
> — Wikipedia

- … well, I am almost always writing and applying functions, …

2nd attempt

- asked copilot: *describe the most important aspect of functional programming in less than 100 words*

The most important aspect of functional programming is immutability. In functional programming, data is not changed after it is created. Instead, functions return new data structures, ensuring that the original data remains unchanged. This leads to more predictable and reliable code, as it eliminates side effects and makes debugging easier. Additionally, immutability facilitates parallel processing, as data consistency is maintained across threads.

It is about Recursions!

My take

- The most important aspect is a cetain “way of thinking” when you solve a problem,
- $\approx$ formulate the solution to the problem using **recursion**

An example

- Q: write a function that computes the sum of elements in an array
- A: a “procedural” Python version

```python
def sum_array(a):
    n = len(a)
    s = 0
    for i in range(n):
        s = s + a[i]
    return s
```

Thinking behind the procedural version

- Well, to compute `a[0] + a[1] + ... + a[n-1]`,
  - start with `s = 0`, and

```
s = s + a[0]
s = s + a[1]
   ...
s = s + a[n-1]
```

- … now `s` should hold what we want
- Remember the time you were confused by the “equation”?

```
s = s + a[i]  # do you mean 0 = a[i] ??
```

A “functional” version

```python
# a[i] + a[i+1] + ... + a[j-1]
def sum_range(a, i, j):
    if i == j:
        return 0
    else:
        return a[i] + sum_range(a, i + 1, j)

def sum_array(a):
    return sum_range(a, 0, len(a))
```

A (superficial) characteristics of the functional version

- No **updates** to variables (like `s = s + ...`)
- No **loops**

… but the point is not about **lack** of something

The thinking behind the functional version

- The key observation:

$$(\text{sum of } a[0:n]) = a[0] + (\text{sum of } a[1:n])$$

and you can compute (sum of $a[1:n]$) by a recursive call

- As a minor note, we defined a function to compute sum of an array **range** $a[i:j]$ by:

$$(\text{sum of } a[i:j]) = a[i] + (\text{sum of } a[i+1:j]),$$

- plus a trivial base case (i.e., $i = j \Rightarrow$ the sum is zero)

Reasoning correctness $\approx$ induction

```
# a[i] + a[i+1] + ... + a[j-1]
def sum_range(a, i, j):
    if i == j:
        return 0
    else:
        return a[i] + sum_range(a, i + 1, j)
```

- We like to establish `sum_range(`a, i, j`)` in fact returns

$$a[i] + \ldots + a[j-1] \qquad (\star)$$

Reasoning correctness ≈ induction

1. $j - i = 0 \Rightarrow$ `sum_range(`a, i, j`)` returns `0` and $a[i] + \ldots + a[j-1] = 0$
2. Otherwise, assume the statement is true for $j - i < k$
   - Then, for $j - i = k$, `sum_range` returns

$$
\begin{aligned}
&a[i] + \mathrm{sum\_range}(a, i+1, j) \\
= &a[i] + (a[i+1] + \ldots + a[j-1]) \quad (\because \text{induction hypothesis}) \\
= &a[i] + \ldots + a[j-1] \qquad (\text{that is, } (\star))
\end{aligned}
$$

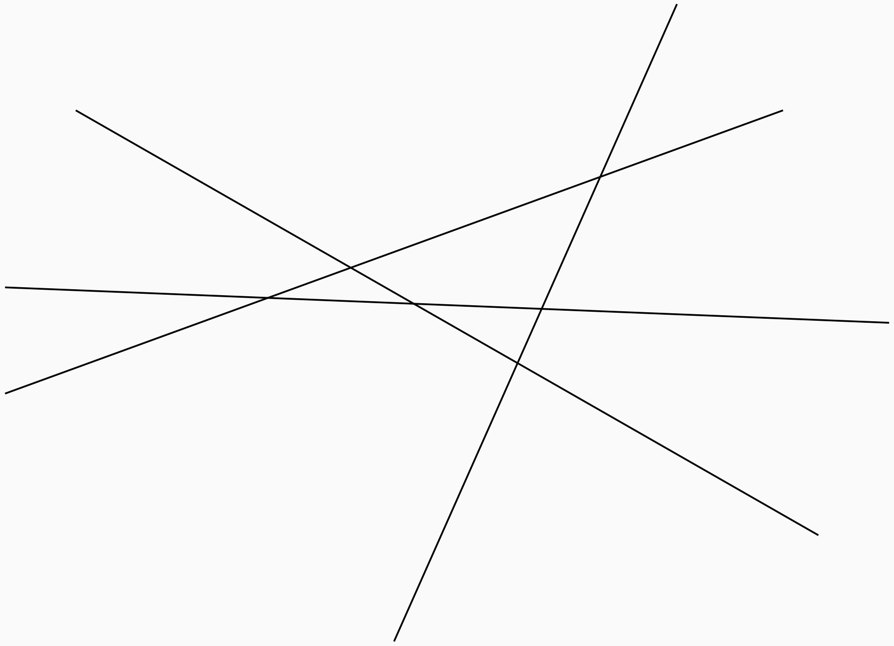
The “functional way” of problem solving

- $\approx$ solving a problem by recursive calls
- $\approx$ solving a problem, *assuming solutions to smaller cases are known*

very powerful for the same reason why solving math problems using **recurrence relation (漸化式)** and proving theorem by **induction (帰納法)** are very powerful

Solving problems with recurrence relation : an example

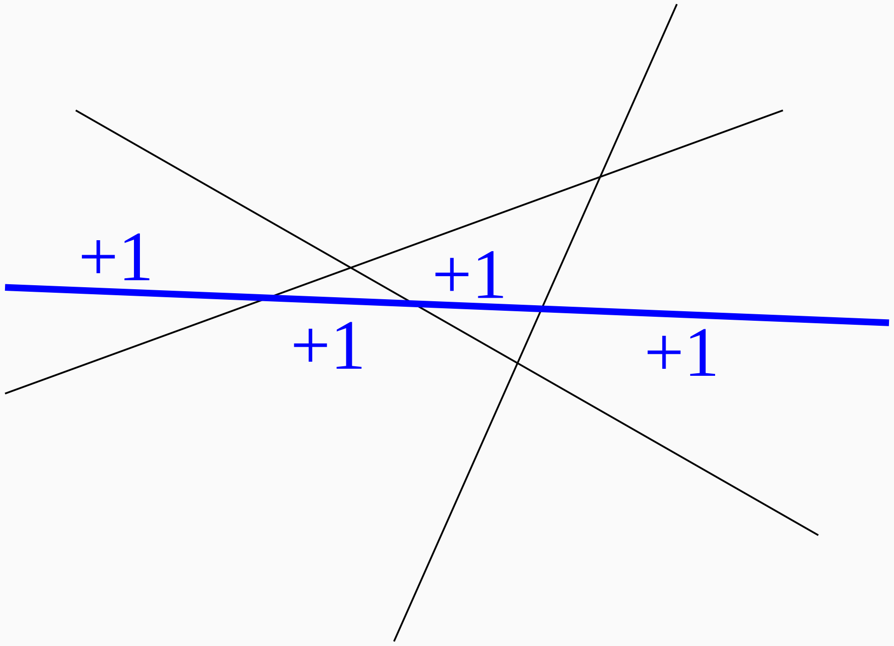
- Q: Draw n lines in a plane (no three lines intersect at a point). How many regions will result?



Solving problems with recurrence relation : an example

- Q: Draw n lines in a plane (no three lines intersect at a point). How many regions will result?

- A: Let the number of regions a_n. Then,



$$a_0 = 1,$$
$$a_n = a_{n-1} + n \quad (n > 0)$$

Into a code …

- Math:

$$a_0 = 1,$$
$$a_n = a_{n-1} + n \quad (n > 0)$$

- Code:

```
def n_regions(n):
    if n == 0:
        return 1
    else:
        return n_regions(n - 1) + n
```

Divide-and-conquer

- A powerful problem-solving paradigm that
  1. *divides* the input into smaller subproblems,
  2. solves (*conquers*) each subproblem recursively, and
  3. combines their solutions to yield the solution

```
def solve(D):
    if D is trivially small:
        return trivial_solve(D)
    D0, D1, ... = divide(D)
    A0 = solve(D0); A1 = solve(D1); ...
    return combine(A0, A1, ...)
```

A textbook example (quicksort)

- Input : an array/list of n elements A
- Output : sort A (i.e., $A[0] \leq A[1] \leq \ldots \leq A[n-1]$)

```
def qs(A):
  if len(A) <= 1:
    return A
  piv = A[0]
  # divide
  lower  = [x for x in A[1:n] if x < piv]
  higher = [x for x in A[1:n] if x >= piv]
  # conquer & combine
  return qs(lower) + [piv] + qs(higher)
```

Other examples

- merge sort
- Discrete Fourier Transform (DFT)
  - $O(n^2)$ algorithm is trivial
  - FFT is a divide-and-conquer algorithm of $O(n \log n)$
- polynomial multiplication of two n-degree polynomials
  - $O(n^2)$ algorithm is trivial
  - Karatsuba algorithm is a divide-and-conquer algorithm of $O(n^{\log_2 3})$ algorithm?

Other examples

- matrix multiplication of two $n \times n$ matrices
  - $O(n^3)$ algorithm is trivial
  - Strassen algorithm is a divide-and-conquer algorithm of $\left(O\left(n^{\log_2 7}\right)\right)$

For Your Exercise …

- maximum segment sum
  - given an array A of n numbers, find p and q that maximizes sum of $A[p : q]$
  - $O(n^2)$ algorithm is trivial
  - can you come up with $O(n \log n)$ or $O(n)$ algorithm?
- inversion count
  - given an array A of n numbers, count the number of (i, j) pairs for which $A[i] > A[j]$
  - can you come up with $O(n \log n)$ algorithm?

Abstracting Computation Patterns by Functions

Common “Patterns”

e.g.,

```python
def sum_square_pos(l):
  s = 0
  for x in l:
    if x > 0:
      s += x * x
  return s
```

- several common patterns in this code

1. go over each element of an array (`for x in l`)
2. do something when a condition is met (`if x > 0`)
3. calculate on each element (`x * x`)
4. reduce them into a single value (`s`)

Functional version (Python)

```python
def sum_square_pos(l):
  if l == []:
    return 0
  elif l[0] > 0:
    return l[0] * l[0] + sum_square_pos(l[1:])
  else:
    return sum_square_pos(l[1:])
```

Functional version (OCaml)

```
let rec sum_square_pos l = match l with
    [] -> 0
  | x :: r ->
    if x > 0
      x * x + sum_square_pos r
    else
      sum_square_pos r
```

- The same boilerplate for every different way of:
  - selecting elements (`x > 0`),
  - calculating a value for each selected element (`x * x`), and
  - reducing all values into one (`+`)

Higher-Order Functions on List (OCaml)

- `List.filter` $p\ l$ = list of elements x in l that satisfies $p\ x$
- `List.map` $f\ l$ = list of $f\ x$ for each x in l
- `List.fold_left` $r\ z\ l = r\ l_{n-1}\ (\ldots(r\ l_1\ (r\ l_0 z))))$
- `List.fold_right` $r\ z\ l = r\ l_0\ (\ldots(r\ l_{n-2}\ (r\ l_{n-1}\ z)))$
- With them and anonymous functions (`fun x -> ...`),

```
let rec sum_square_pos l =
  List.fold_left (fun x y -> x + y) 0
    (List.map (fun x -> x * x)
      (List.filter (fun x -> x > 0) l))
```

A Shorter Version

- OCaml supports:

1. "Function" versions of infix operators: `(+)`, `(<)`, `...`
   - i.e., `(+) x y` $\equiv$ `x + y`
   - $\therefore$ `(+)` $\equiv$ `fun x y -> x + y`
2. Partial applications. e.g.,
   - for `f x y = E`, two parameter function, `f x` $\equiv$ `fun y -> E`
   - $\therefore$ `(<) 0` $\equiv$ `fun y -> (<) 0 y` ($\equiv$ `fun y -> 0 < y`)
3. Pipeline operator
   - `x |> f` $\equiv$ `f x`

A Shorter Version

- Combined,

```
let sum_square_pos l =
  l |> List.filter ((<) 0)
    |> List.map (fun x -> x * x)
    |> List.fold_left (+) 0
```

- Note: the language you chose may or may not have similar functions builtin (you can roll it by yourself when it doesn't)

Deep Recursion, Stack Overflow, and Tail Recursion

Deep recursion may lead to stack overflow

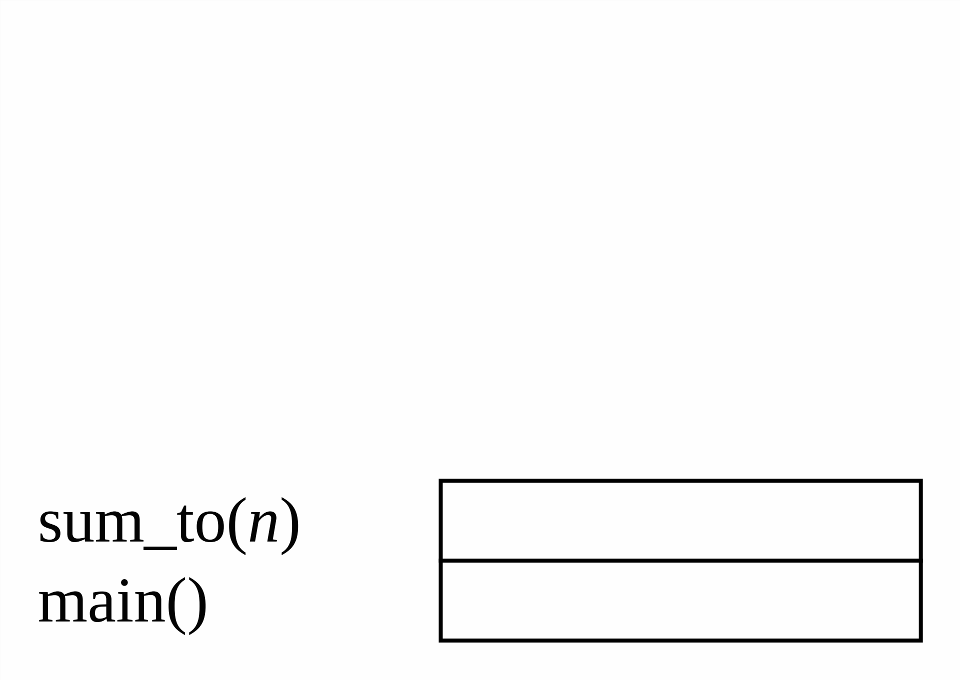
- e.g., to compute sum $1 + \ldots + n$

```python
def sum_to(n):
  if n == 0:
    return 0
  else:
    return n + sum_to(n - 1)
```

```
>>> sum_to(1000)
    ... <snip> ...
    return n + sum_to(n - 1)
               ^^^^^^^^^^^^^
  [Previous line repeated 996
more times]
RecursionError: maximum
recursion depth exceeded
```

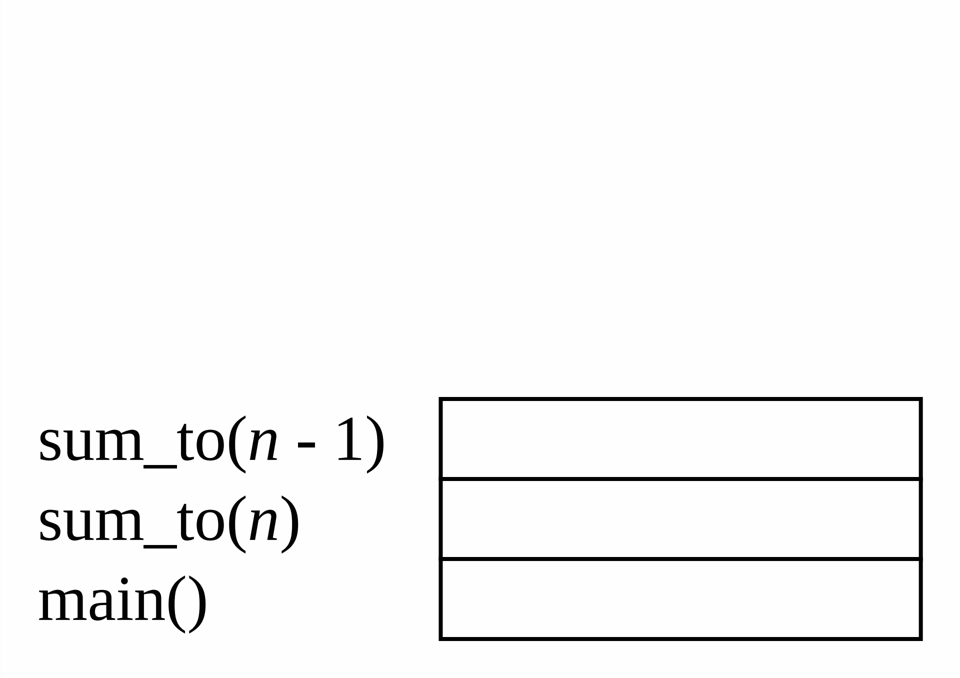
Why does it happen?

- A function call requires space for storing variables and intermediate values



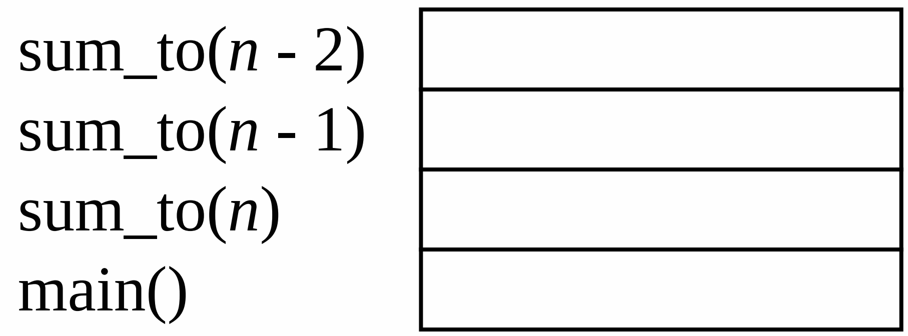
Why does it happen?

- A function call requires space for storing variables and intermediate values



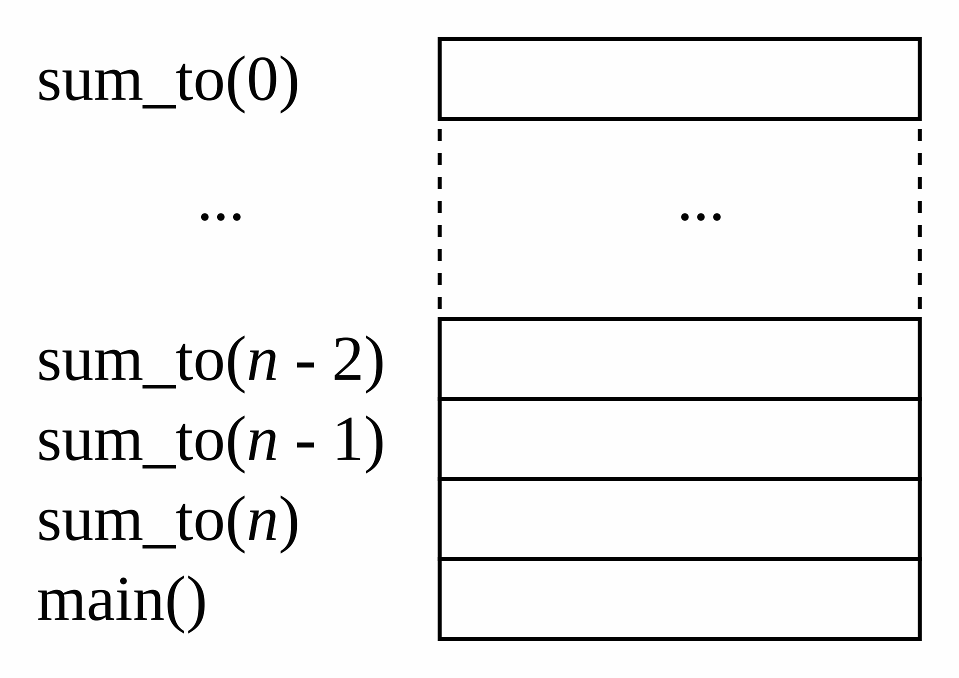
Why does it happen?

- A function call requires space for storing variables and intermediate values



Why does it happen?

- A function call requires space for storing variables and intermediate values



How to avoid it?

1. Use a knob to set the stack size if it easily fixes your problem

   …,
2. use a “balanced” recursion if possible, like:

```
def sum_range(a, b):
  if a == b:
    return 0
  else:
    c = (a + 1 + b) // 2
    return a + sum_range(a + 1, c) + sum_range(c, b)
```

3. … or use **“tail recursion”**

What is "tail recursion"?

- **Tail call** : if function f calls g, and f does nothing after g returns (other turn returns it), such a call to g is said "tail call"

```python
def f(x):
  if ...:
    return g(x)     # tail call
  else:
    return g(x) + 1 # not tail call
```

- **Tail recursion** $\equiv$ recursive call that is a tail call

Tail-recursive sum_to

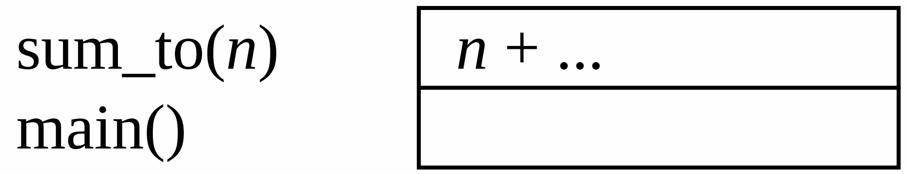
```
def sum_to_tail(n, s):
  if n == 0:
    return s
  else:
    return sum_to_tail(n - 1, n + s)

def sum_to(n):
  return sum_to_tail(n, 0)
```

- Confirm `sum_to_tail`(n, s) returns $(1 + ... + n) + s$

Why does it have anything to do with stack overflow?

- The true reason a function call requires space is to store values *required after the call*

```python
def sum_to(n):
  if n == 0:
    return 0
  else:
    return n + sum_to(n - 1)
```



Why does it have anything to do with stack overflow?

- The true reason a function call requires space is to store values *required after the call*

```
def sum_to(n):
  if n == 0:
    return 0
  else:
    return n + sum_to(n - 1)
```

sum_to(n - 1)	(n - 1) + ...
sum_to(n)	n + ...
main()	

Why does it have anything to do with stack overflow?

- The true reason a function call requires space is to store values *required after the call*

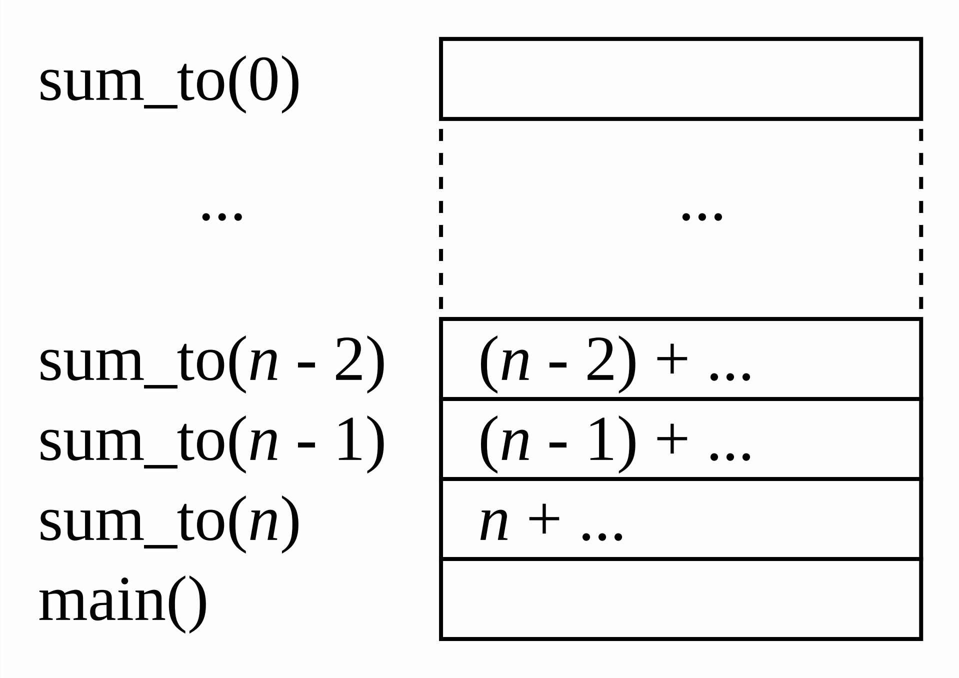
```python
def sum_to(n):
  if n == 0:
    return 0
  else:
    return n + sum_to(n - 1)
```

sum_to(n - 2)	(n - 2) + ...
sum_to(n - 1)	(n - 1) + ...
sum_to(n)	n + ...
main()	

Why does it have anything to do with stack overflow?

- The true reason a function call requires space is to store values *required after the call*

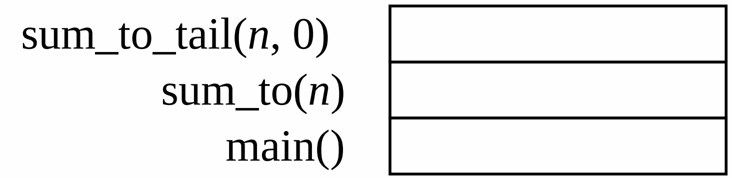
```
def sum_to(n):
  if n == 0:
    return 0
  else:
    return n + sum_to(n - 1)
```



Why does it have anything to do with stack overflow?

- But for tail calls, no space is required for computation after the call!

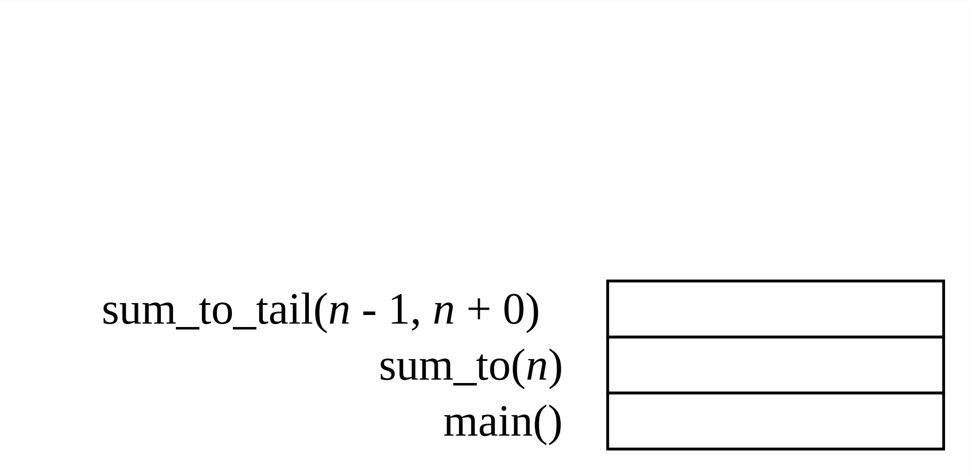
```
def sum_to_tail(n, s):
  if n == 0:
    return s
  else:
    return sum_to_tail(n - 1,
n + s)
```



Why does it have anything to do with stack overflow?

- But for tail calls, no space is required for computation after the call!

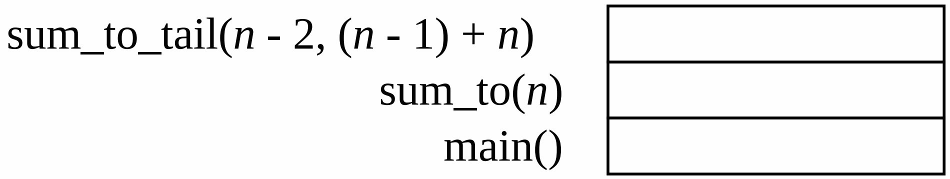
```python
def sum_to_tail(n, s):
  if n == 0:
    return s
  else:
    return sum_to_tail(n - 1,
n + s)
```



Why does it have anything to do with stack overflow?

- But for tail calls, no space is required for computation after the call!

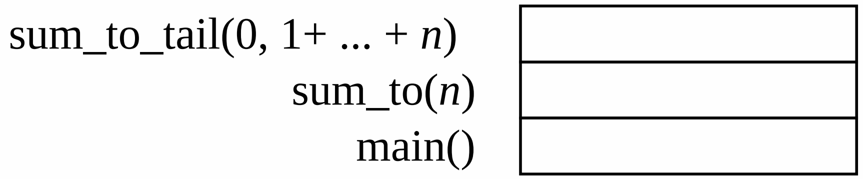
```python
def sum_to_tail(n, s):
  if n == 0:
    return s
  else:
    return sum_to_tail(n - 1, n + s)
```



Why does it have anything to do with stack overflow?

- But for tail calls, no space is required for computation after the call!

```
def sum_to_tail(n, s):
  if n == 0:
    return s
  else:
    return sum_to_tail(n - 1,
n + s)
```



How to come up with a tail-recursive version?

1. there is no universal formula
2. adding an extra parameter storing “partial result” often does it
   - e.g., `sum_to`$(n) \rightarrow$ `sum_to_tail`(n, s)
   - … and slightly change the spec
   - `sum_to_tail`$(n, s) = (1 + \ldots + n) + s$
3. there is a general template for converting “loop” into tail recursion

Loop to tail-recursion

- following is a *general* template

```
x = x0
y = y0
while E(x, y):
  x = F(x, y)
  y = G(x, y)
return ...
```

$\Rightarrow$

```
let rec loop x y =
  if not (E x y) then
    ...
  else
    let x' = F x y in
    let y' = G x' y in
    loop x' y'
in
loop x0 y0
```

An example (sum_to)

- the natural for loop

```
s = 0
for i in range(1, n+1):
  s += i
return s
```

- while-loop version

```
i = 1
s = 0
while i <= n:
  s = s + i
  i = i + 1
return s
```

Tail recursion

```
i = 1
s = 0
while i <= n:
  s = s + i
  i = i + 1
return s
```

$\Rightarrow$

```
let rec sum_to_tail i n s =
  if i > n then
    s
  else
    sum_to_tail (i + 1) n (s + i)
in
sum_to_tail 1 n 0
```

A final remark about stack overflow

- Stack overflow is an *implementation artifact* (avoidable)
- In principle, we should be able to grow (stack + heap) up to the computer memory
- Yet, stack tends to overflow much earlier than that (e.g., a few MB, when you can use $>$ GB for other data)
- It is actually an *unnecessary* constraint, imposed by a typical/ traditional memory management strategy that allocates stack separately from heap as a contiguous memory whose maximum size is set when a program (or a thread) is started

A final remark about stack overflow

- A suitable language implementation can avoid such unnecessary overflow altogether by allocating stack more flexibly (e.g., dynamically growing it by allocating stack frames from heap)
- Few language implementations (e.g., Standard ML New Jersey) do it