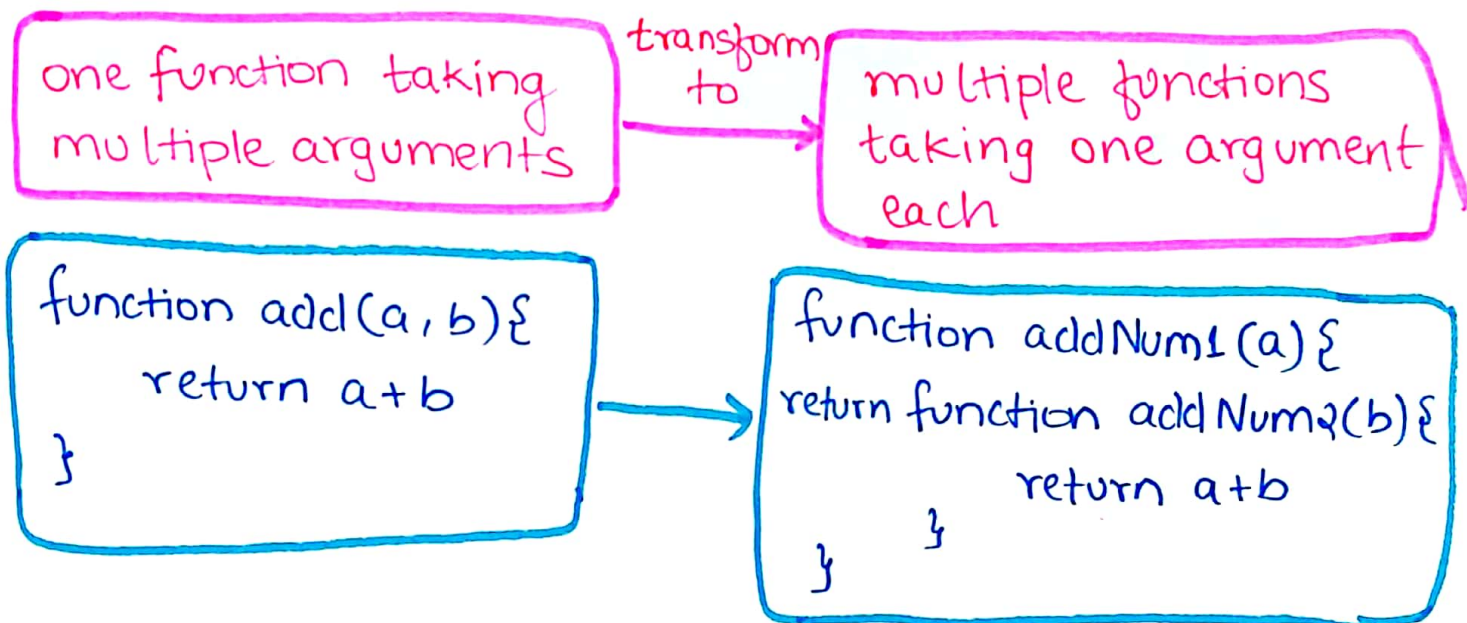


Currying

@codeWithSimran

Currying is a mechanism where we can translate the evaluation of a function that takes multiple arguments into evaluating a sequence of functions that take a single argument.



✱✱ The reason why we're able to achieve this is because function addNum2 has access to variable a due to closures

So now, how do we use this function?

► `addNum1(5)(3)` and not `addNum1(5,3)`

because `addNum1` only takes one argument and it returns a function so result of `addNum1(5)` is a function (`addNum2`) and we want to pass the second argument to `addNum2`

Alright, why would someone even do this in the first place??

Let's say we just call

@codeWithSimran

→ `addNum1(5)`

this will return us a function, so let's store it for future use

→ `const addToNum5 = addNum1(5)`

// sometime in future

→ `addToNum5(3)`

@codeWithSimran

→ `addToNum5(4)`

we're trying to run less code in future by storing ~~`addNum1`~~ `addToNum5` for future and not running it everytime

Compose

★ The data processing that we do should be obvious

Example.

Let's say we want to make a paste from 5 ingredients mixed together.

item1 add → item2 blend → newItem add → item3

Final Product ← blend |

So final product is made from item1, item2, item3 and the order doesn't matter

What compose says is we should be able to do the following

item2 add → item3 blend → newItem add → item1

Final Product blend |

★ Composability is a system design principle
A highly composable environment components can be assembled in various combinations and still get the right output

@codeWithSimran

Code example: -

@codeWithSimran

Let's say we have a negative number, we want to multiply it with another number and return the absolute value

$$-2 * 3 \rightarrow -6 \text{ absolute } 6$$

Let's compose these together

1) `const composedResult =`

`compose(multiplyBy3, returnAbsolute)`
↓

we need to define our own compose func.

~~`const compose = (f, g) =>`~~

2) `const compose = function(funcA, funcB) {`

3) `return function(data) {`

4) `return funcB(funcA(data))`
`}`

→ we can also do

`>> composedResult(-2)` `funcA(funcB(data))`
→ 6

Let's assume `multiplyBy3` and `returnAbsolute` are defined.

on line one we're calling `compose` function that takes 2 functions as arguments and returns a function (line 3), this function is stored in `composedResult`. we can now call `composedResult` with any data (-2) and it applies `funcB` on result of `funcA`