Chapter 3: Recursion

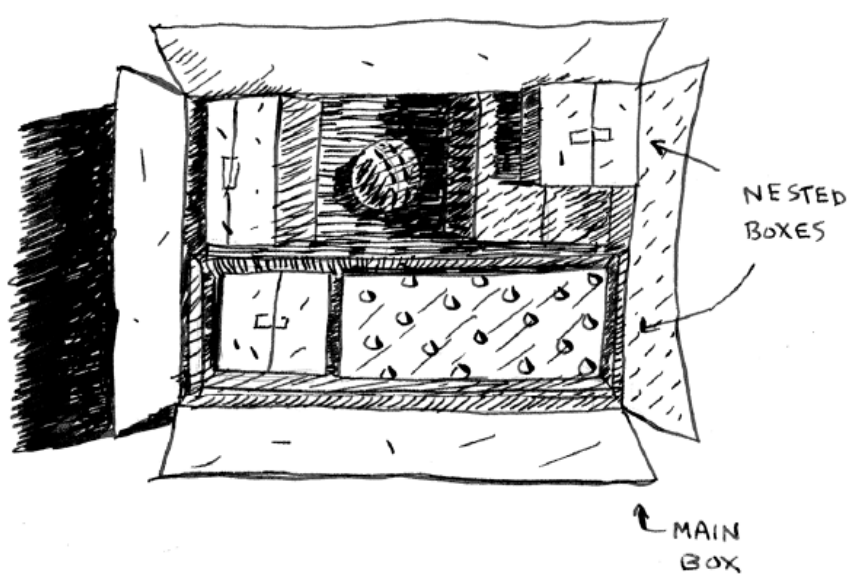
↳ Focuses:

↳ Learn about Recursion — coding technique

↳ Learn about what base cases & recursive cases are.

Recursion

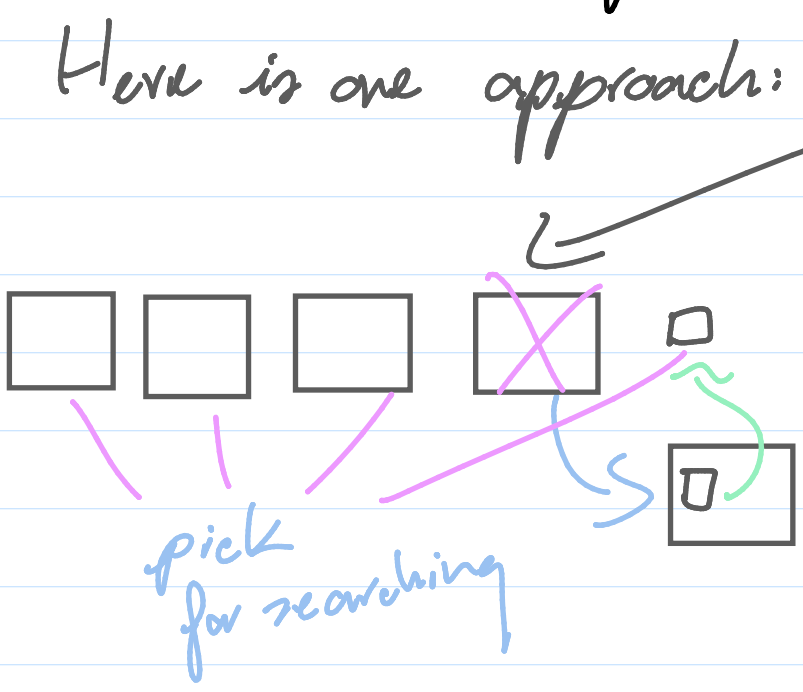
↳ Suppose you're going through a box to find a Key...

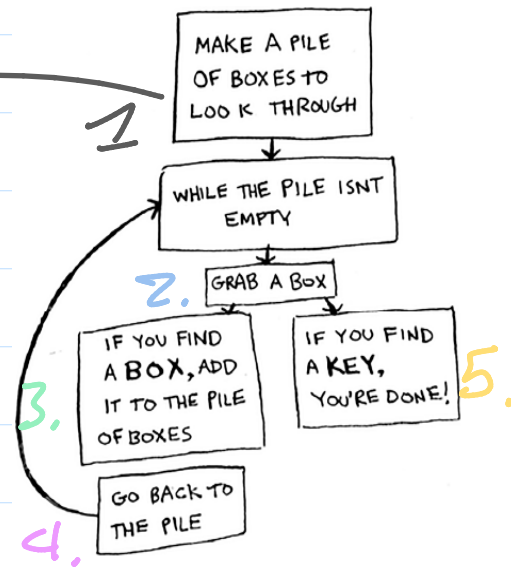


but to find this key you have to go through more boxes inside but there are more boxes inside those boxes!

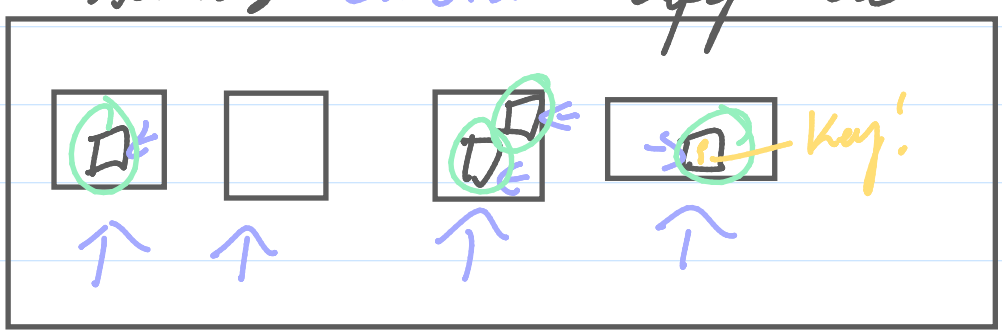
What should you do?

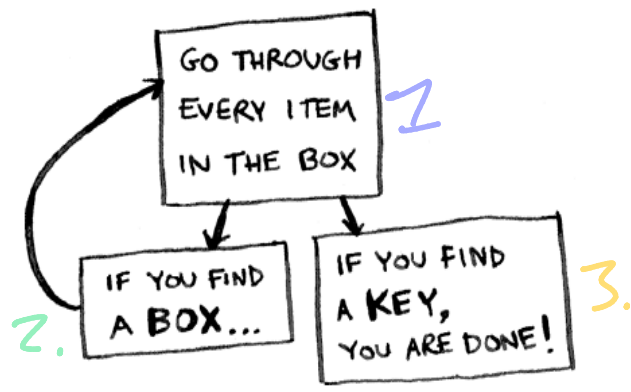
Here is one approach:





Here's another approach:





First Approach in Python:

```
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print("found the key!")
```

under while loop

← if it finds a box 1.

adds to pile 2.

3. Repeats until 4. finds key

Second Approach in Python:

```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(item)
    elif item.is_a_key():
      print("found the key!")
```

under new box!

function for box existence

for any item

If item is a box

Recursion!

calls itself again...

back here!

Recursion is where a function calls itself.

↳ Why?

↳ Simplifies the actions necessary to complete a task.

↳ Although, there is no performance benefit for using it.

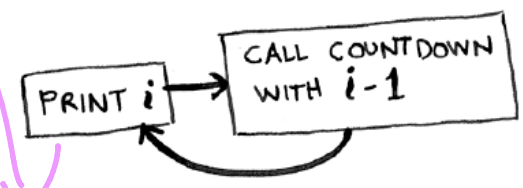
Quote from Leigh Caldwell: "Loops may achieve a performance gain for your program. Recursion may achieve a performance gain for your programmer.

Base Case and Recursive Case

- Using Recursive Functions is a good strategy to apply but since they call themselves repeatedly, how do we specify a clear condition that stops recursion?
  - Look at this recursive function...

```
def countdown(i):
  print(i)
  countdown(i-1)

countdown(3)
```




```
> 3...2...1...0...-1...-2...
```

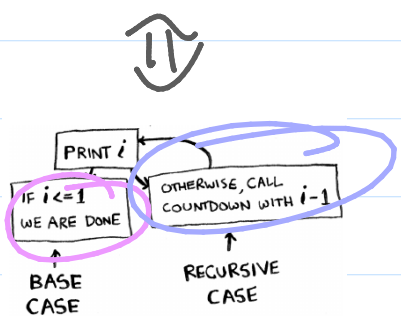
    - Since it's meant to countdown to 0, why does it go past 0?
      - Because there's no conditional that stops the recursion.
        - Means that it will go on forever.

That's called an infinite loop

  - How do we stop this?
    - Every recursive function has two parts.
      - Recursive Case
        - When a function calls itself
      - Base Case
        - The end condition, when the function stops calling itself (stops an eventual infinite loop)

```
def countdown(i):
  print(i)
  if i <= 1:          <-- Base case
    return
  else:               <-- Recursive case
    countdown(i-1)

countdown(3)
```



Call Stacks

↳ Think of a to-do list, but instead of putting them side-by-side, instead, each to-do note/item are stacked on top of each other.

↳ Like this:

↳ Now, the unique thing about the call stack is that you only read the topmost (most recently placed) element in a stack, and other elements CANNOT be accessed nor read until you pop the other elements on top of the chosen element.

↳ The only element that can be read & accessed is the last/topmost element on the stack

What is "pop"?

↳ One of the two options available for a stack.

Available actions for a Stack (call stack)

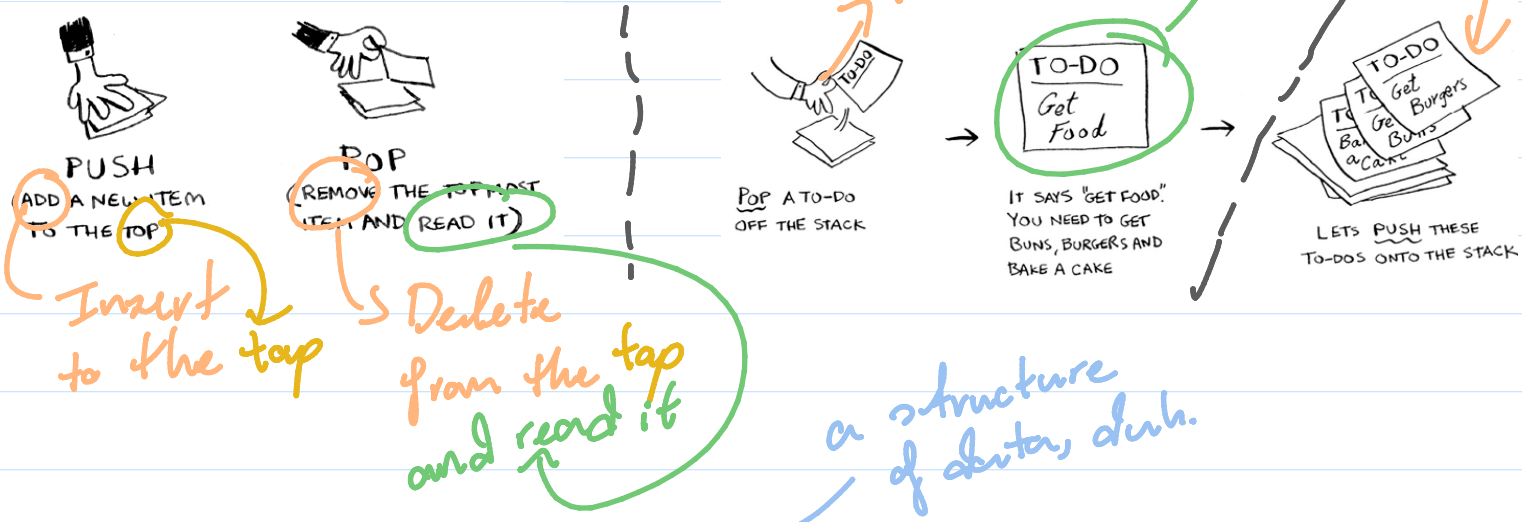
↳ Push

↳ "Push-ing" some element means to insert a new element to the last/topmost place in the stack, thus becoming the only element to be read and/or accessed until it's popped.

↳ Pop

↳ "Pop-pping" an element means to remove/delete an element from the topmost/latest position in the stack, it also returns the value/data in that topmost element.

Push vs Pop in Images.



Stacks are a data structure, very simple.

a structure of data, duh.

Stacks described in code

↳ This is a function, greet() that could expand two more functions, greet2() and bye().

greet()

```
def greet(name):
    print("hello, " + name + "!")
    greet2(name)
    print("getting ready to say bye...")
    bye()
```

greet2()

```
def greet2(name):
    print("how are you, " + name + "?")
```

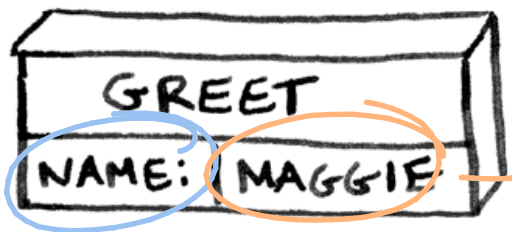
bye()

```
def bye():
    print("ok bye! ")
```

Suppose you could greet("david"), firstly the computer will allocate a "box of memory" for greet("david") specifically.

memory allocated ONLY FOR greet("david")

We currently have allocated memory but we haven't used it. So let's set the parameter: "david" for memory to use.

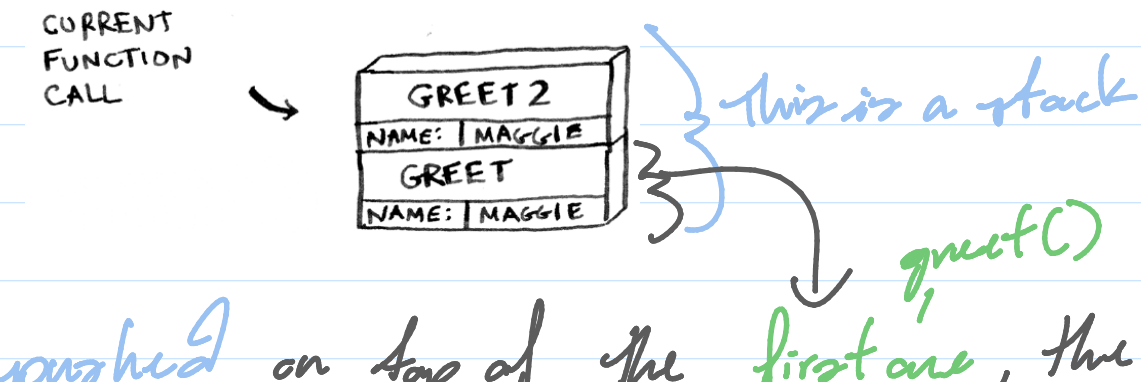


(variable) (value of variable name) also the parameter of greet

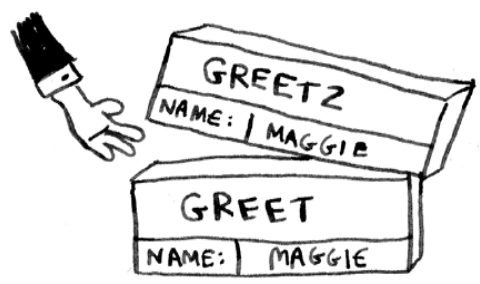
for the book's example, NOT OURS

computer executes functions

Every time you make a function/call, the computer saves the values for ALL variables for any function call. Next, it prints "hello, david!", then from greet()
greet2("david") is called to which more memory is allocated for greet2()



The 2nd box is pushed on top of the first one, the 2nd box prints out "How are you, david?", then returns back to the function call (greet()), when this happens, the box, greet2() gets popped off the stack.



What we just coded was a function executed to it's completion ontop of a function that was partially completed until the newly executed function is completed.

- The original function was paused until the second function completed it's task.
- Same process goes with the bye() function.

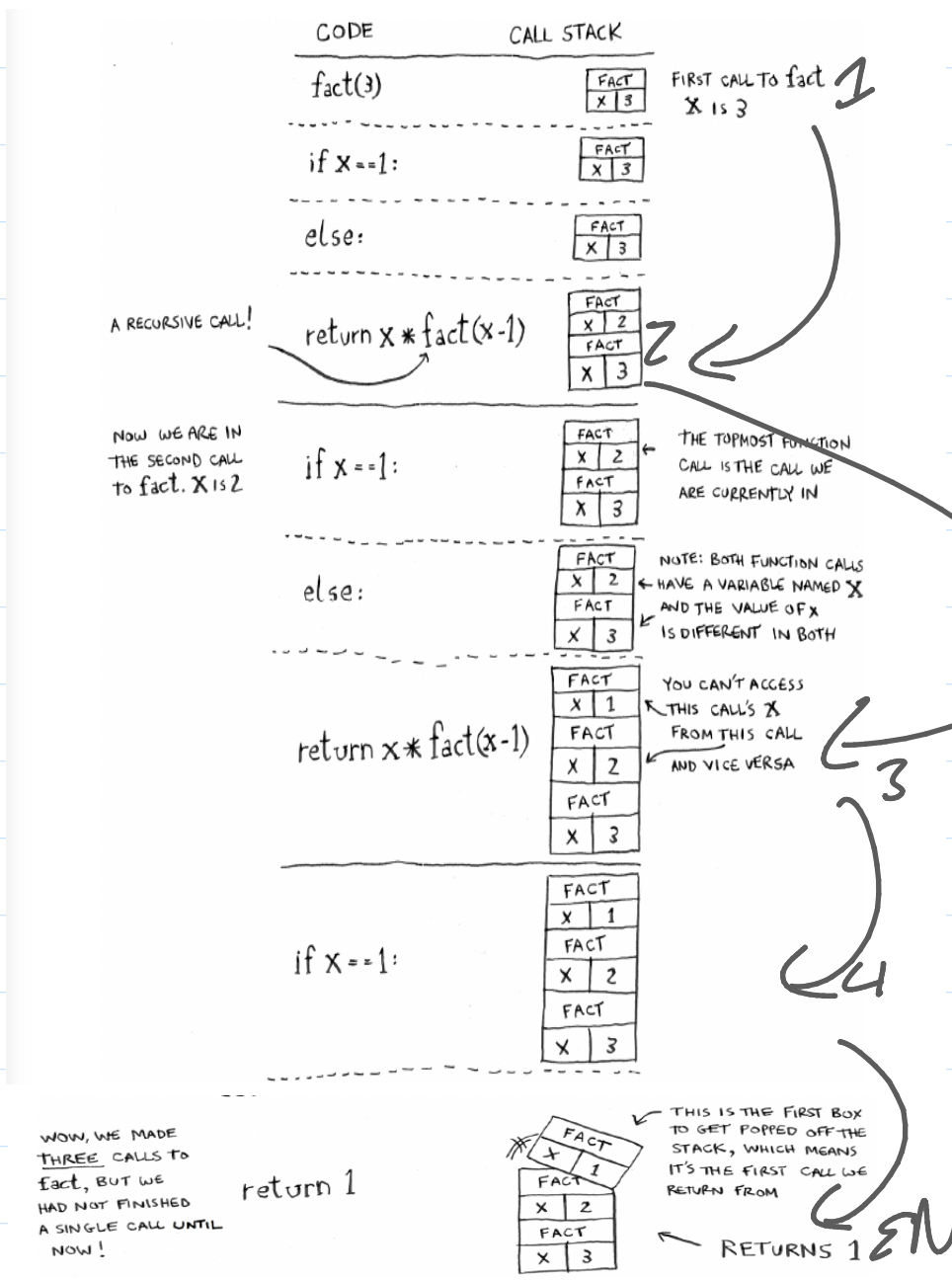
Stacks that are used to save variables for multiple functions are called call stacks.

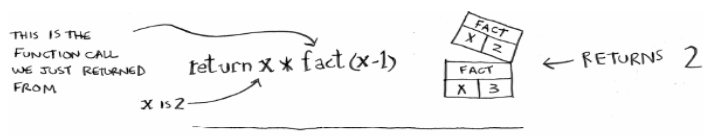
Call Stacks w/ Recursion

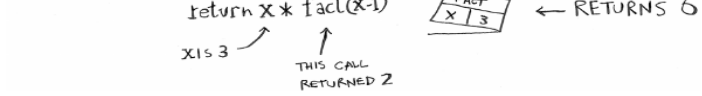
We get a recursive function that works to output a factorial for any number n.

```
def fact(x):
  if x == 1:
    return 1
  else:
    return x * fact(x-1)
```

Here's a line-by-line demonstration of how the call stack changes:







Each call stack has its own copy of x. You cannot access a different function's copy of x.

Using the Stack is convenient, but at a cost.

- Saving all that info takes up a lot of memory.
  - A remedy for this could be:
    - Rewrite your code to use a for loop
    - use tail recursion
    - advanced topic
- The good thing about Stacks is that
  - You don't have to keep track of all those nested functions functions executed by a function
    - the stack does it for you.