Chapter Six
Functions: Advanced Topics

CSC 161: The Art of Computer Programming

Matt Post (grad TA; guest lecturer)

10/28/2009

REVIEW

Here we'll walk through the execution of a function. The slides will try to make explicit the facts that (a) when a function executes, the only information it has from its calling context are variables passed in from the calling function through its formal parameters and (b) the calling function knows nothing about how the function operates; all it gets is the return value.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

Here are three function definitions. We take main() to be the canonical starting point.

function definitions:

```
def square(x):
	return x * x

def distance(p1,p2):
	squarex = square(p2.getX() - p1.getX())
	squarey = square(p2.getY() - p1.getY())
	return math.sqrt(squarex + squarey)

def main():
	p1 = Point(2,3)
	p2 = Point(5,7)
	dist = distance(p1,p2)
	print dist
```

You run main()

Imagine we invoke the main() function from the python prompt. In the lower left is a box that will contain the output of the program as it is printed.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
    p1 = Point(2,3)
```

You run main()

The bold statement in the upper left means that the computer executes that line.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
    p1 = Point(2,3)
    p2 = Point(5,7)
```

You run main()

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
```

You run main()

We have now created two points. On the third line, we try to assign a value to the “dist” variable, but to do that, we need to first execute the distance() function.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
distance(p1 = main::p1, p2 = main::p2):
    squarex = square(3)
```

You run main()

scratch pad:

```
    p2.getX() - p1.getX()
=   5 - 2
=   3
```

distance() is called, and main()'s values of p1 and p2 (the "actual value") are bound to the variables listed in distance()'s definition (its "formal parameters"). We then try to execute the first line of distance to assign a value to squarex, but to do that, we need to execute the square() function. The "scratch pad" in the lower right is a conceptual device I'm using here to show that the computer needs to compute the value to send to square(). We take the equation from the first line of the definition of distance(), and compute its value to send to square().

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
distance(p1 = main::p1, p2 = main::p2):
square(x = 3):
    return 9
```

You run main()

scratch pad:

```
    x * x
=   3 * 3
=   9
```

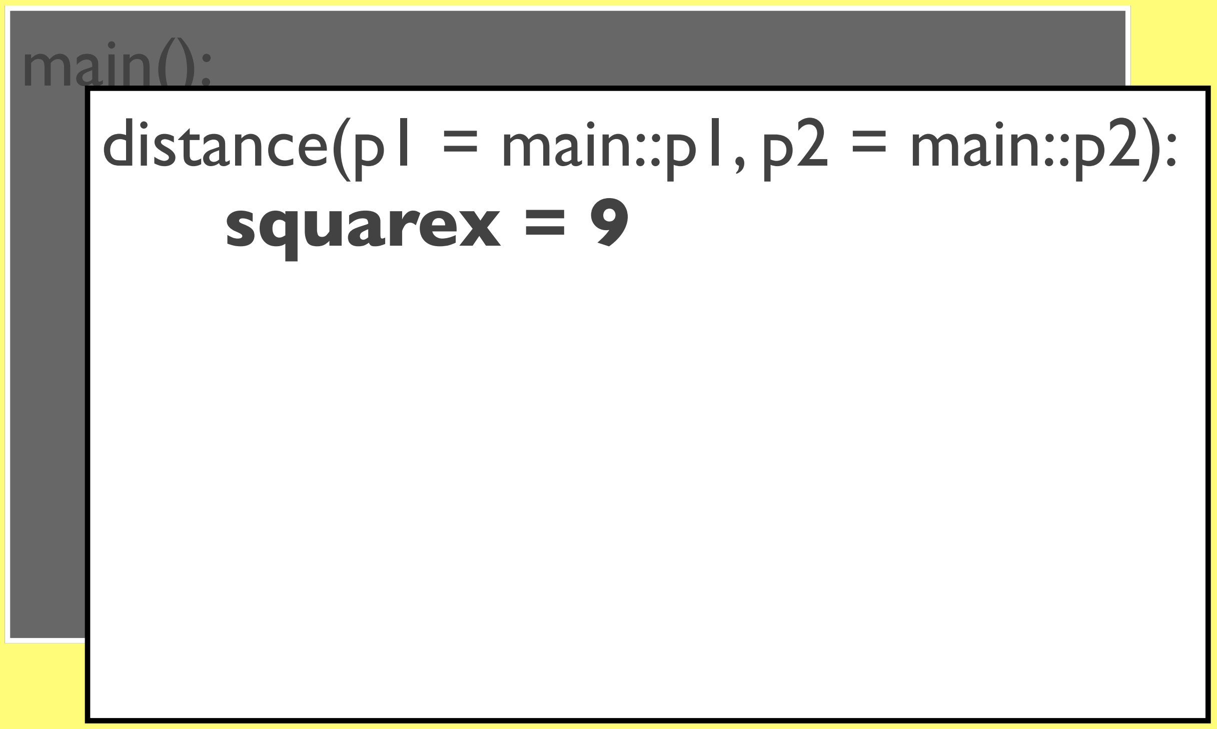
The value of 3 passed to square() is bound to square()'s formal parameter, **x**. On the scratch pad we compute the return value, and send it back to the calling function.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```



You run main()

One way to think about what's happening is to imagine that the function call gets replaced by its return value. Now that we have an actual number (the return value of calling distance()), we can assign the value to **squarex** and continue to the next line.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
distance(p1 = main::p1, p2 = main::p2):
    squarex = 9
    squarey = square(4)
```

You run main()

scratch pad:

```
    p2.getY() - p1.getY()
=   7 - 3
=   4
```

On the scratch pad we compute the value of p2.getY() – p1.getY() so we can pass that value as an argument to our next invocation of square().

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
  distance(p1 = main::p1, p2 = main::p2):
    square(x = 4):
        return 16
```

You run main()

scratch pad:

```
    x * x
=   4 * 4
=   16
```

Here we finish our call to square(), just as we did before.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
distance(p1 = main::p1, p2 = main::p2):
    squarex = 9
    squarey = 16
```

You run main()

When the square() function returns, we can assign its value to **squarey**.

function definitions:

```
def square(x):
	return x * x

def distance(p1,p2):
	squarex = square(p2.getX() - p1.getX())
	squarey = square(p2.getY() - p1.getY())
	return math.sqrt(squarex + squarey)

def main():
	p1 = Point(2,3)
	p2 = Point(5,7)
	dist = distance(p1,p2)
	print dist
```

```
main():
distance(p1 = main::p1, p2 = main::p2):
	squarex = 9
	squarey = 16
	return math.sqrt(25)
```

You run main()

scratch pad:

```
	squarex + squarey
=	9 + 16
=	25
```

On the scratch pad we compute the value of the argument to math.sqrt(). Note that this is another function call, which gets called (and executes) just like our calls to square(), etc. But for purposes of this demonstration, we won't step into the function, and just take its return value.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
```

```
distance(p1 = main::p1, p2 = main::p2):
    squarex = 9
    squarey = 16
    return 5
```

You run main()

math.sqrt() returns its value, and now distance() can return its own value to its caller, main().

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = 5
```

You run main()

distance() returns, so we can finish line 3 of main(). Again, you can think of the call to distance() being replaced in the code with its return value, after it completes. This value is then assigned to **dist**.

function definitions:

```
def square(x):
    return x * x

def distance(p1,p2):
    squarex = square(p2.getX() - p1.getX())
    squarey = square(p2.getY() - p1.getY())
    return math.sqrt(squarex + squarey)

def main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = distance(p1,p2)
    print dist
```

```
main():
    p1 = Point(2,3)
    p2 = Point(5,7)
    dist = 5
    print 5
```

You run main()

```
5
```

Now print can print the value of **dist**, and this is displayed in our output box.

TOPIC 1
functions that modify their parameters

This is the subject of Section 6.5.2 in your book. It's basically an exception or special case to what you've been told before.

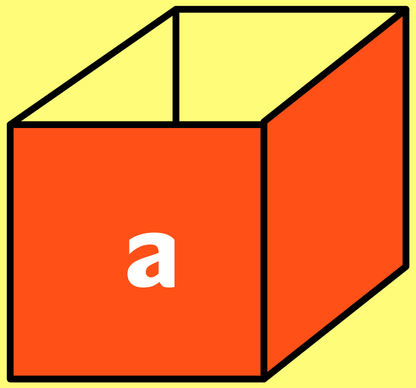
```
a = 17
```

```
b = [1, 2, 3, 4, 5]
```

Think of a variable as a box that can contain a value. A box usually has a name (the name of the variable), and a value (the value that has been assigned to it). A box can only have **one** value directly inside it. When you create a list, since a variable (box) can only have one value inside it, what happens implicitly behind the scenes is that a bunch of little anonymous boxes are created. The list variable contains all of those little boxes, and those little boxes contain the actual values. This way you obey the rule that a box can only directly have one value.

```
a = 17
```

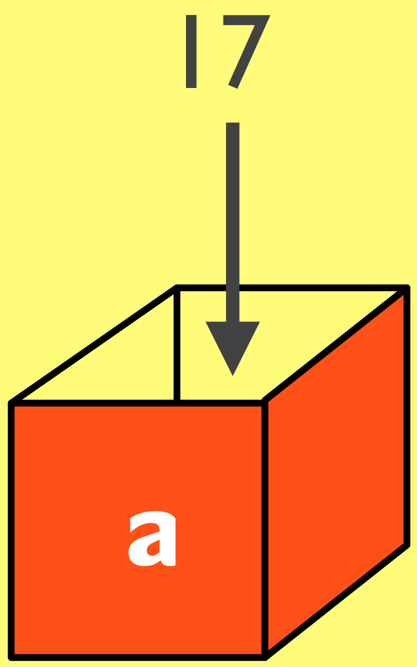
```
b = [1, 2, 3, 4, 5]
```



Think of a variable as a box that can contain a value. A box usually has a name (the name of the variable), and a value (the value that has been assigned to it). A box can only have **one** value directly inside it. When you create a list, since a variable (box) can only have one value inside it, what happens implicitly behind the scenes is that a bunch of little anonymous boxes are created. The list variable contains all of those little boxes, and those little boxes contain the actual values. This way you obey the rule that a box can only directly have one value.

```
a = 17
```

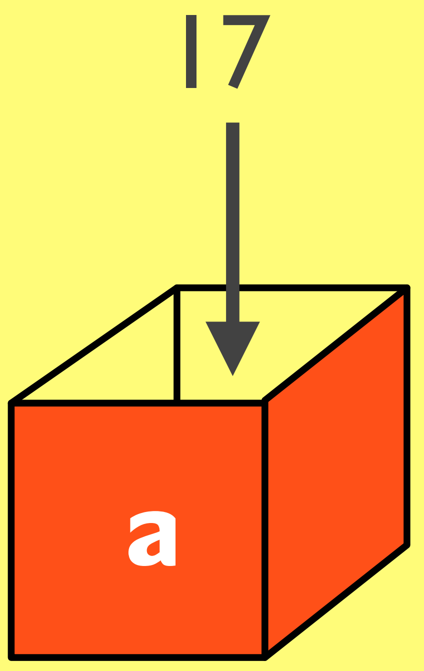
```
b = [1, 2, 3, 4, 5]
```

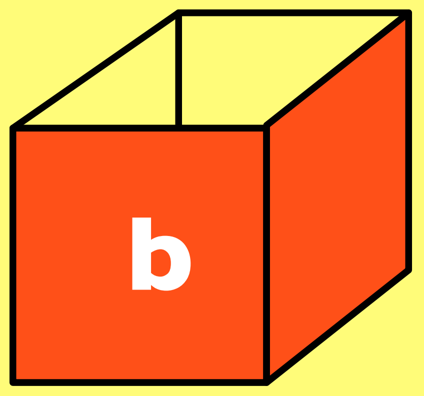


Think of a variable as a box that can contain a value. A box usually has a name (the name of the variable), and a value (the value that has been assigned to it). A box can only have **one** value directly inside it. When you create a list, since a variable (box) can only have one value inside it, what happens implicitly behind the scenes is that a bunch of little anonymous boxes are created. The list variable contains all of those little boxes, and those little boxes contain the actual values. This way you obey the rule that a box can only directly have one value.

```
a = 17
```

```
b = [1, 2, 3, 4, 5]
```

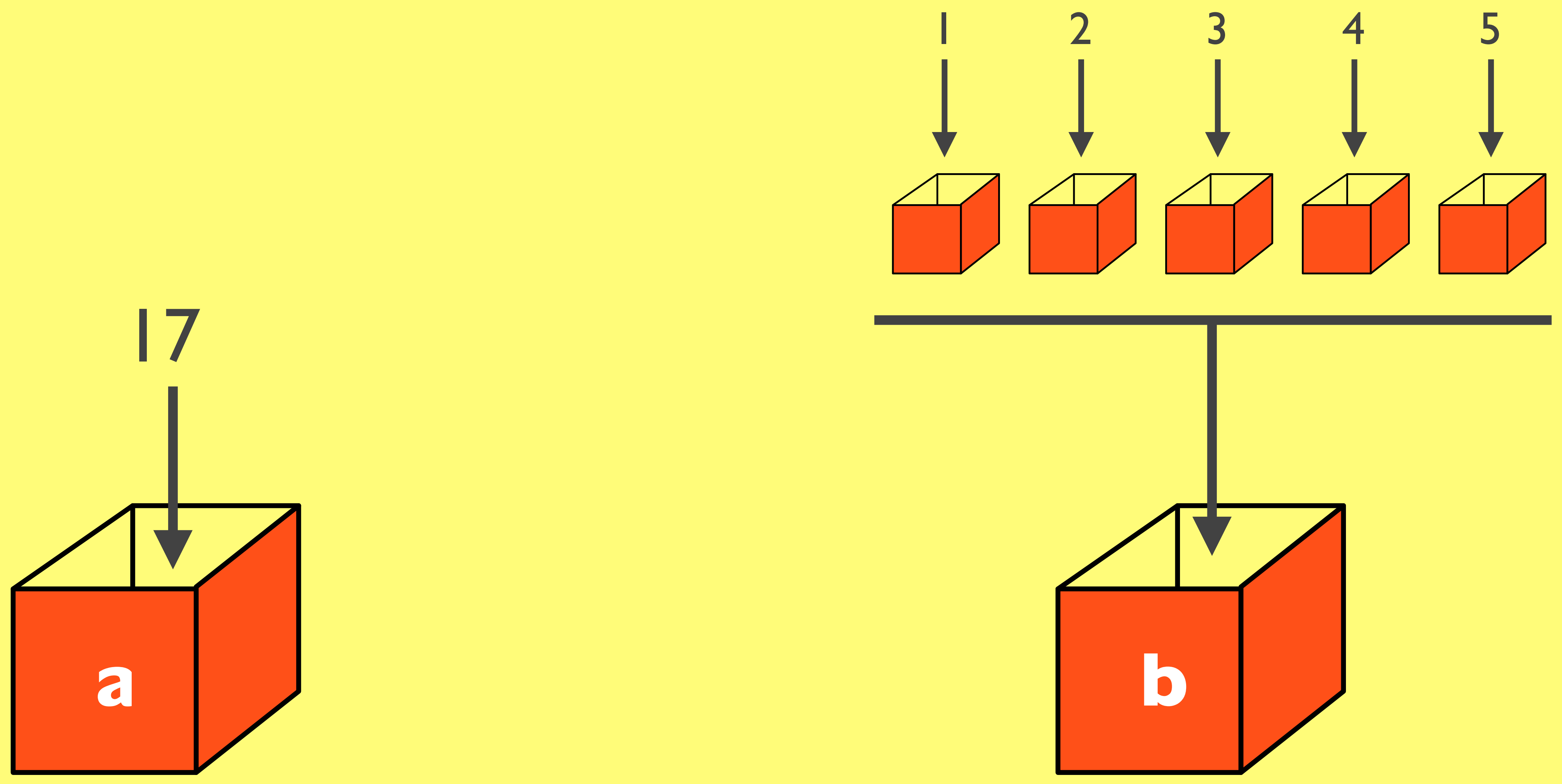




Think of a variable as a box that can contain a value. A box usually has a name (the name of the variable), and a value (the value that has been assigned to it). A box can only have **one** value directly inside it. When you create a list, since a variable (box) can only have one value inside it, what happens implicitly behind the scenes is that a bunch of little anonymous boxes are created. The list variable contains all of those little boxes, and those little boxes contain the actual values. This way you obey the rule that a box can only directly have one value.

```
a = 17
```

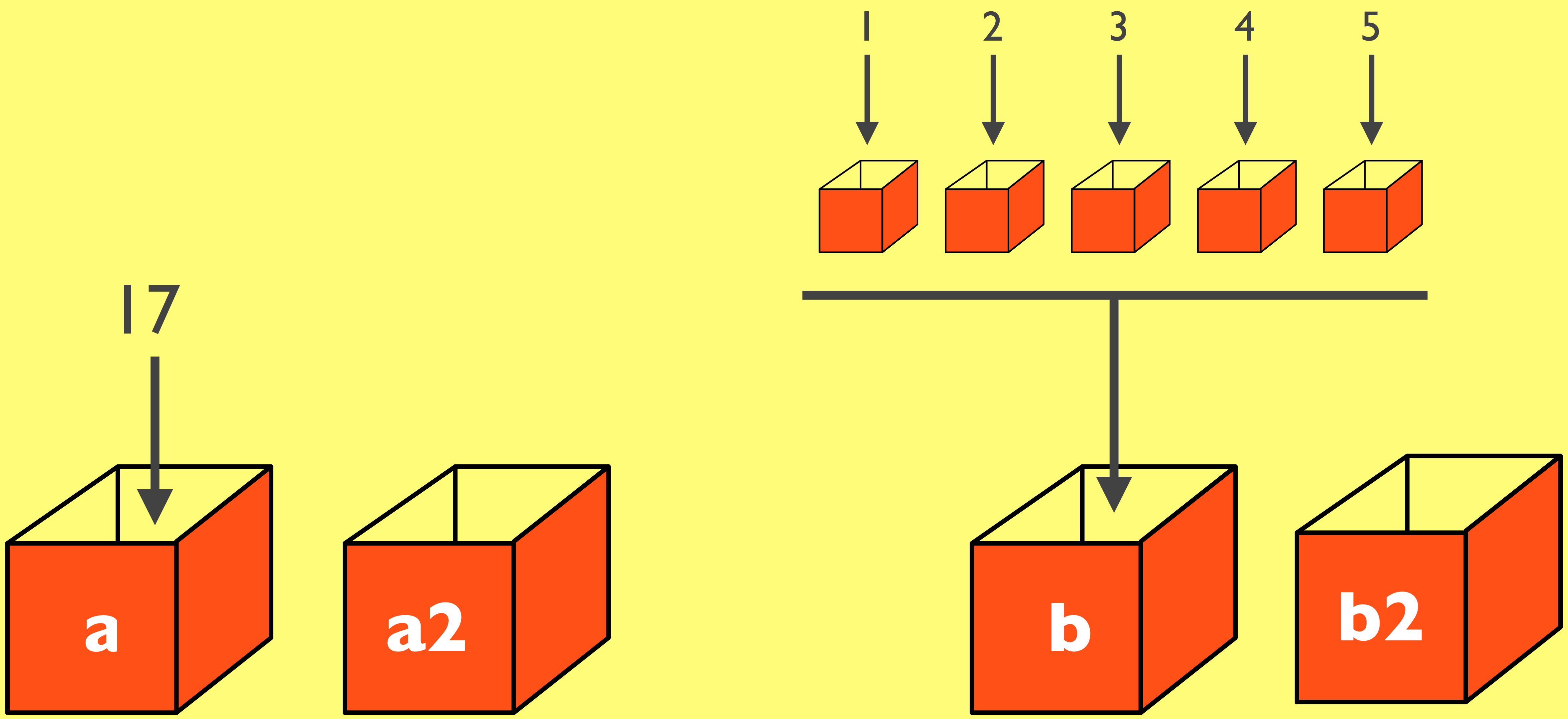
```
b = [1, 2, 3, 4, 5]
```



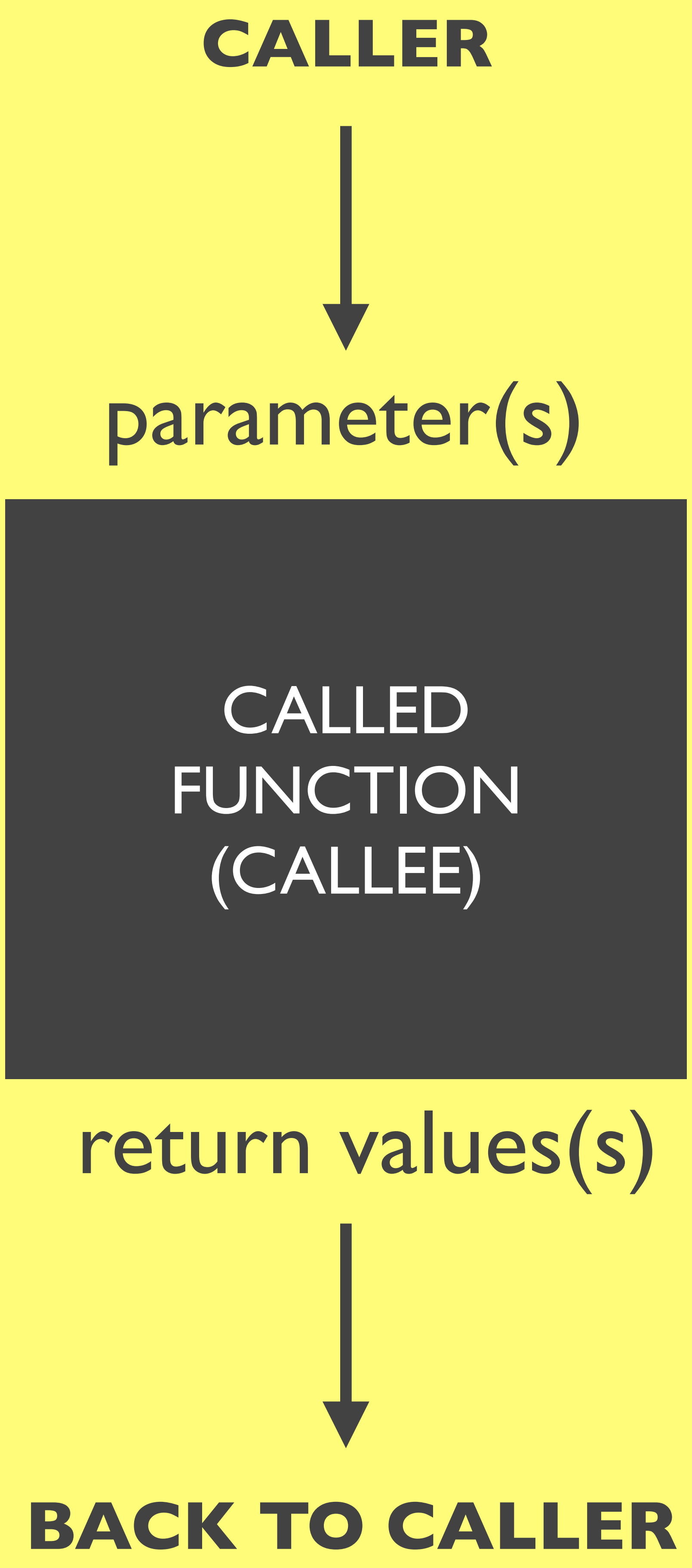
Think of a variable as a box that can contain a value. A box usually has a name (the name of the variable), and a value (the value that has been assigned to it). A box can only have **one** value directly inside it. When you create a list, since a variable (box) can only have one value inside it, what happens implicitly behind the scenes is that a bunch of little anonymous boxes are created. The list variable contains all of those little boxes, and those little boxes contain the actual values. This way you obey the rule that a box can only directly have one value.

```
a = 17
a2 = a
```

```
b = [1, 2, 3, 4, 5]
b2 = b
```

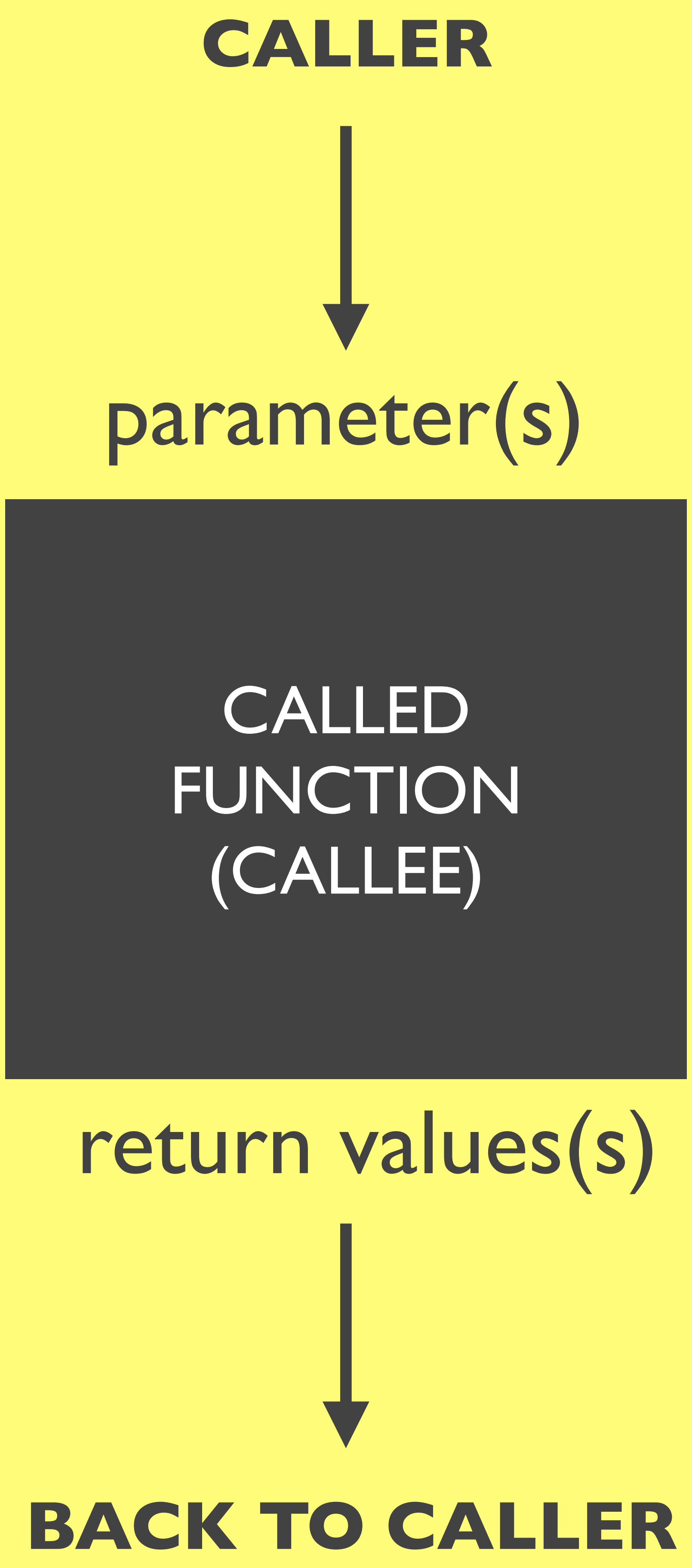


When you assign the value of a variable to another variable, it gets the contents of the box. When you assign **a2** = **a**, a new box **a2** gets the same value as **a**. They are independent; changing **a2** will not change **a**, nor vice versa. But when you assign **b2** = **b**, each contains the anonymous list of boxes, so changing the contents of one will change the other.



This is the old story, which we are about to amend. When variables are passed in to a function, an assignment (of the actual parameters to the function’s formal parameters) is made. This means that changes made within the function to certain kinds of parameters will actually be reflected in the caller’s variables, as well.

- *parameters* are passed from the caller to the callee by *value*, not by identity
- changes made to those parameters in the callee are not visible to the caller



This is the old story, which we are about to amend. When variables are passed in to a function, an assignment (of the actual parameters to the function’s formal parameters) is made. This means that changes made within the function to certain kinds of parameters will actually be reflected in the caller’s variables, as well.

```
def change(num):
      num = num * 2

def main():
      mynum = 7
      change(mynum)
      print mynum
```

```
def change(list):
      if (len(list)):
            list[0] = list[0] * 2;

def main():
      mylist = [7, 6, 5, 4, 3]
      change(mylist)
      print mylist[0]
```

running main() prints

running main() prints

This code exemplifies this point.

```
def change(num):
      num = num * 2

def main():
      mynum = 7
      change(mynum)
      print mynum
```

```
def change(list):
      if (len(list)):
            list[0] = list[0] * 2;

def main():
      mylist = [7, 6, 5, 4, 3]
      change(mylist)
      print mylist[0]
```

running main() prints

```
7
```

running main() prints

This code exemplifies this point.

```
def change(num):
      num = num * 2

def main():
      mynum = 7
      change(mynum)
      print mynum
```

running main() prints

```
7
```

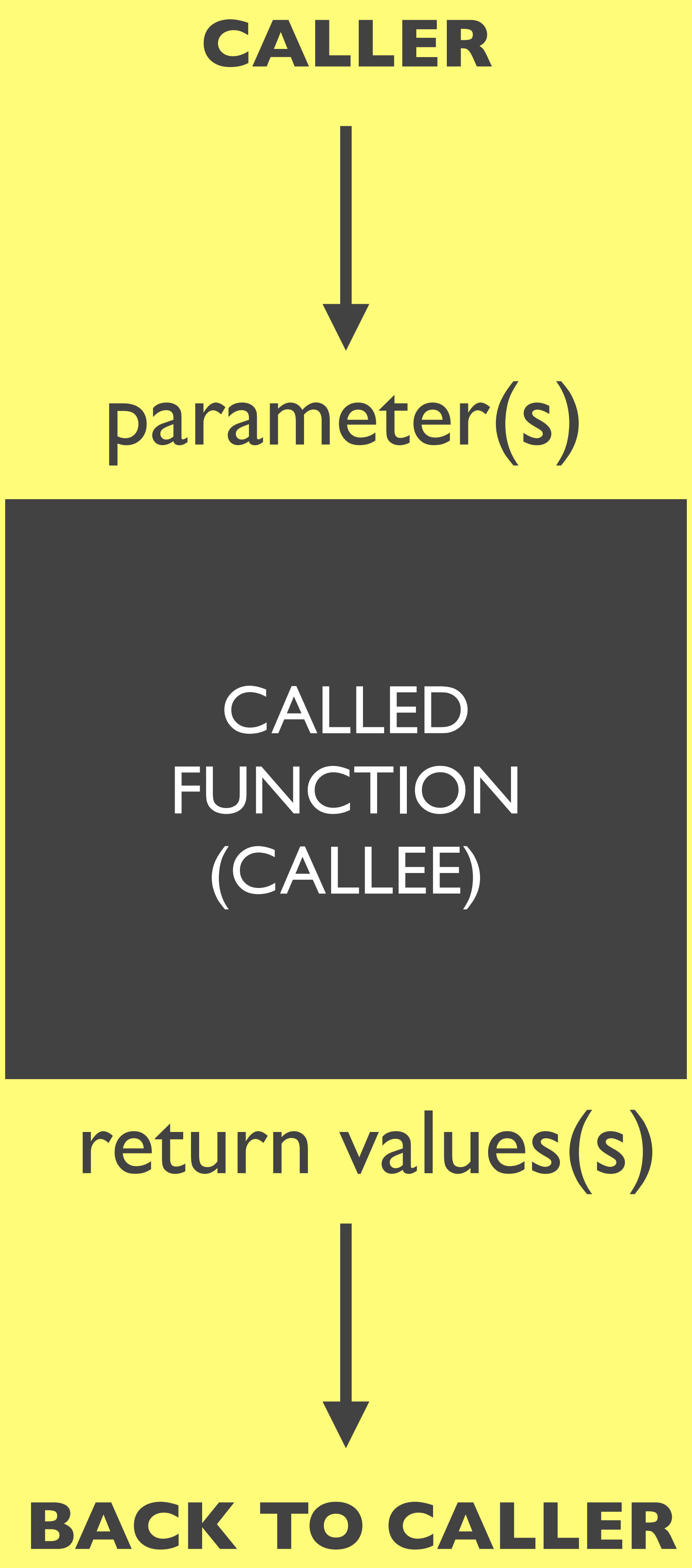
```
def change(list):
      if (len(list)):
            list[0] = list[0] * 2;

def main():
      mylist = [7, 6, 5, 4, 3]
      change(mylist)
      print mylist[0]
```

running main() prints

```
14
```

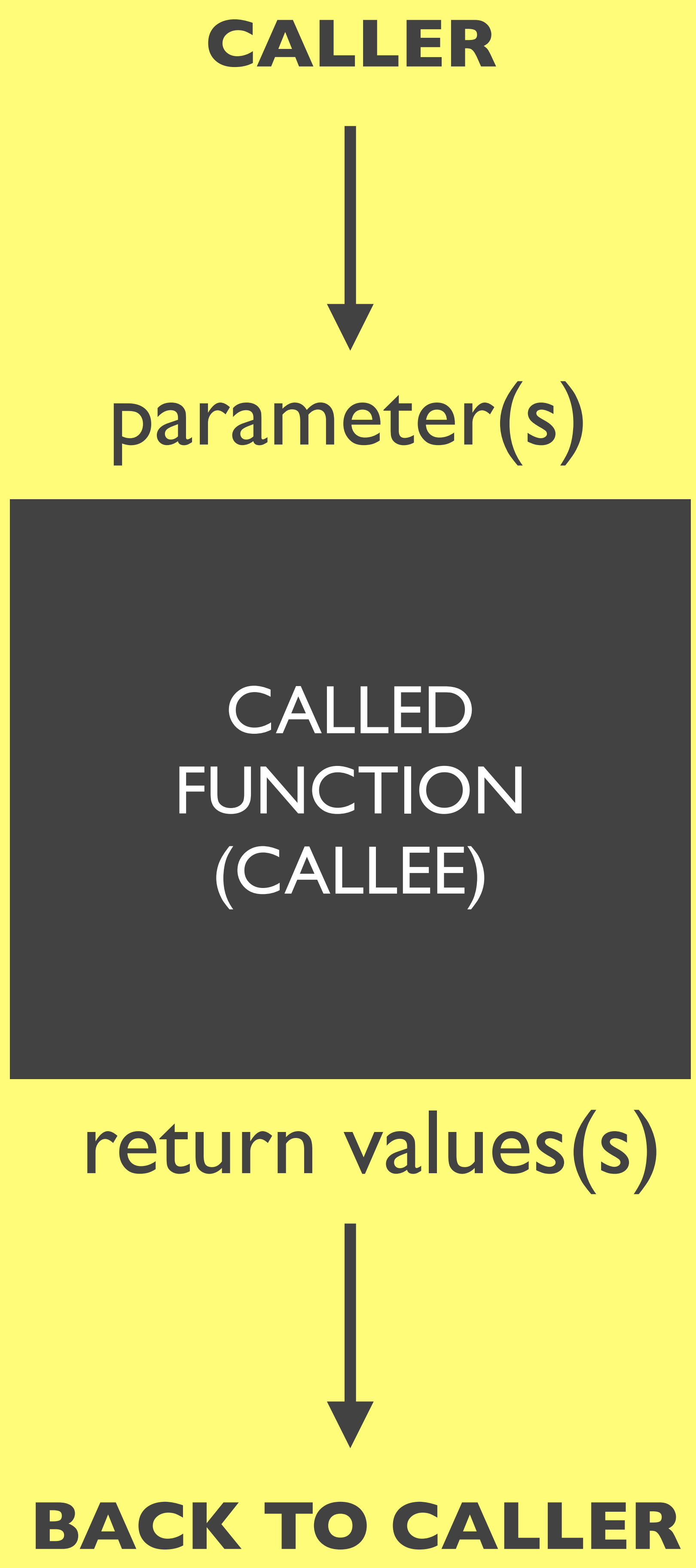
This code exemplifies this point.



Here is the new story for how information gets into and out of a function. See the next slide for which kinds of variables fall into which category.

old story

- *parameters* are passed from the caller to the callee by *value*, not by identity
- changes made to those parameters in the callee are not visible to the caller



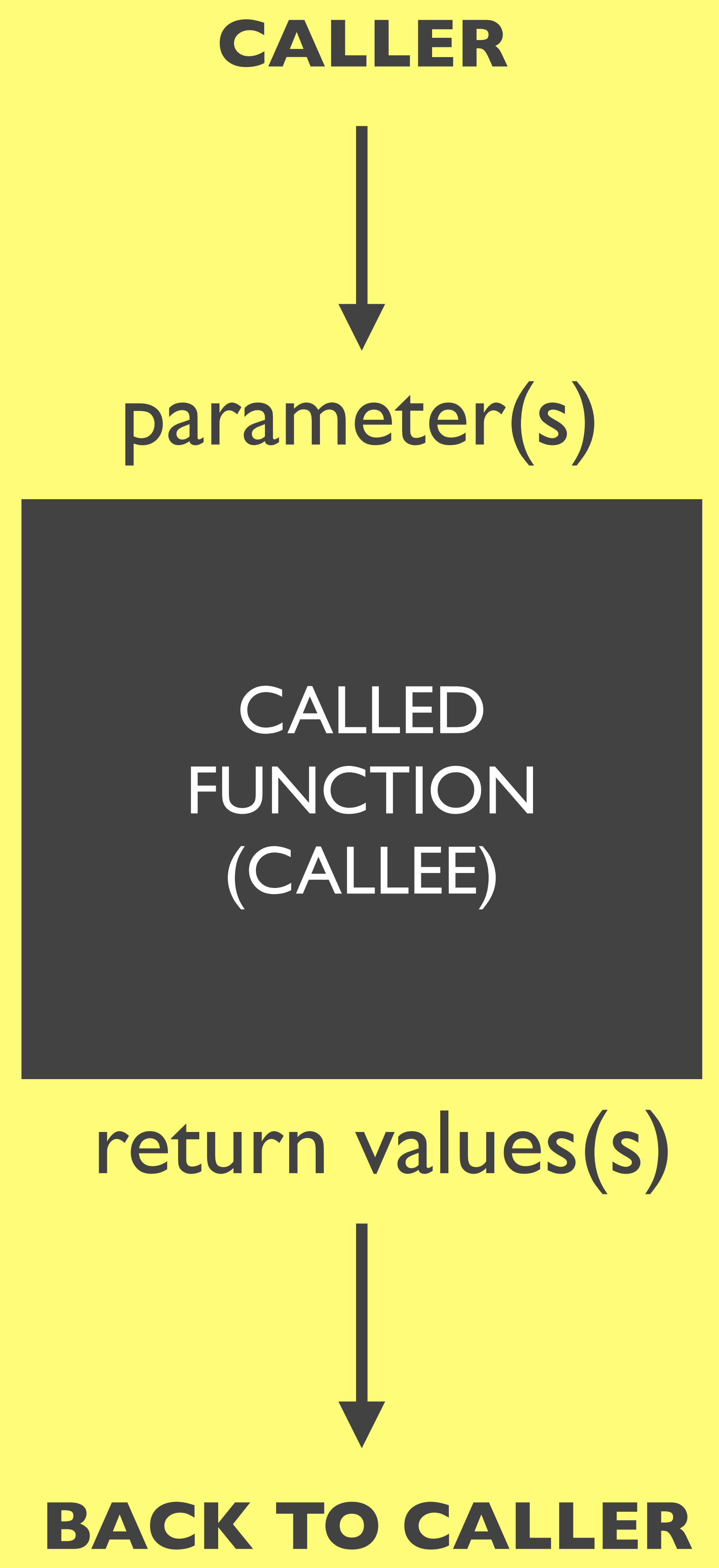
Here is the new story for how information gets into and out of a function. See the next slide for which kinds of variables fall into which category.

old story

- *parameters* are passed from the caller to the callee by *value*, not by identity
- changes made to those parameters in the callee are not visible to the caller

new story

- *some* parameters are passed by value
- others are not



Here is the new story for how information gets into and out of a function. See the next slide for which kinds of variables fall into which category.

immutable
(passed by value)

fixed types:

integer	29
float	29.0
tuple	(-4, 2.1, 'test')

http://www.stoneworld.co.uk/images/products/600.jpg
http://www.kansastravel.org/littlehouseontheprairie.htm

Immutable types (rocks) are atomic units. Assignment of these types of items creates independent copies, so that the assigner and assignee have different copies, and changes made to one will not be seen by the other. Mutable types (houses) will be shared. This is, for example, why you can't create a Rectangle object, draw it, assign it to a new variable, and then draw it again. You have to call clone(), a special function which actually does create a real copy.

immutable (passed by value)

mutable (can be modified)

fixed types:

integer	29
float	29.0
tuple	(-4, 2.1, ’test’)

objects:

class	Point()
list	[-4, 2.1, ‘test’]

http://www.stoneworld.co.uk/images/products/600.jpg
http://www.kansastravel.org/littlehouseontheprairie.htm

Immutable types (rocks) are atomic units. Assignment of these types of items creates independent copies, so that the assigner and assignee have different copies, and changes made to one will not be seen by the other. Mutable types (houses) will be shared. This is, for example, why you can’t create a Rectangle object, draw it, assign it to a new variable, and then draw it again. You have to call clone(), a special function which actually does create a real copy.

http://wirednewyork.com/landmarks/liberty/images/liberty.jpg

Let’s consider an example. Most of us have probably heard the story of how the names of immigrants were changed by inspectors at Ellis island to reflect a more assimilated Westernized spelling and pronunciation. These stories are probably somewhat apocryphal, but we’ll ignore that for the moment and write a function that helps out the inspectors.

```
def assimilate_foreigner(names):
    changed = 0
    for i in range(len(names)):
        if names[i] == ‘Nüchter’:
            names[i] = ‘Nickter’
            changed += 1
        elif names[i] == ‘Wojciechowski’:
            names[i] = ‘Smith’
            changed += 1
        elif names[i] == ‘Hitler’:
            names[i] = ‘Hunter’
            changed += 1

    return changed
```

This function takes a list of names that would be hard to pronounce for an 18th or 19th century English-speaking Westerner (or, in the case of Hitler, a name which would be undesirable at some point in the future), and changes them. Since names is a list, changes made to it in the function will be also changed in the caller’s list. We also count the number of changes we made.

```
def suggest_assimilated_names(names):
    # make an independent copy
    newnames = list(names)
    changed = 0
    for i in range(len(names)):
        if names[i] == ‘Nüchter’:
            newnames[i] = ‘Nickter’
            changed += 1
        elif names[i] == ‘Wojciechowski’:
            newnames[i] = ‘Smith’
            changed += 1
        elif names[i] == ‘Hitler’:
            newnames[i] = ‘Hunter’
            changed += 1

    return changed, newnames
```

This function just makes suggestions, instead of forcing the change. Notice that we avoid making copies in the caller’s list by explicitly creating a real copy with the list() function.

TOPIC 2

recursion

Recursion is the definition of a function or object in terms of itself. It is a relatively easy concept to understand, but to really be able to apply it in computer science takes some considerable work and persistence. If you don't understand this, it might help to know that it takes even computer science majors a lot of time to really be able to use it. Try writing your own functions, make diagrams and drawings, and go over it again and again.

this is seriously the most awesome concept in computer science

It really is.

recursion trivia

- a limitless fount of esoteric computer science humor
  - ❖ dictionary definition
    recursion, *n.* See "recursion".
  - ❖ try Googling "recursion" (after today's lecture)
- recursive acronyms
  - ❖ GNU = "GNU's not UNIX"
  - ❖ pine = "pine is not elm"
- also a property of language, is found in nature, and serves as the foundation for a class of mathematical proofs

Here are some examples of recursion.

function definitions:

```
def fun():
    fun()

def main():
    fun()
```

You run main()

In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.

function definitions:

```
def fun():
      fun()

def main():
      fun()
```

```
main():
      fun()
```

You run main()

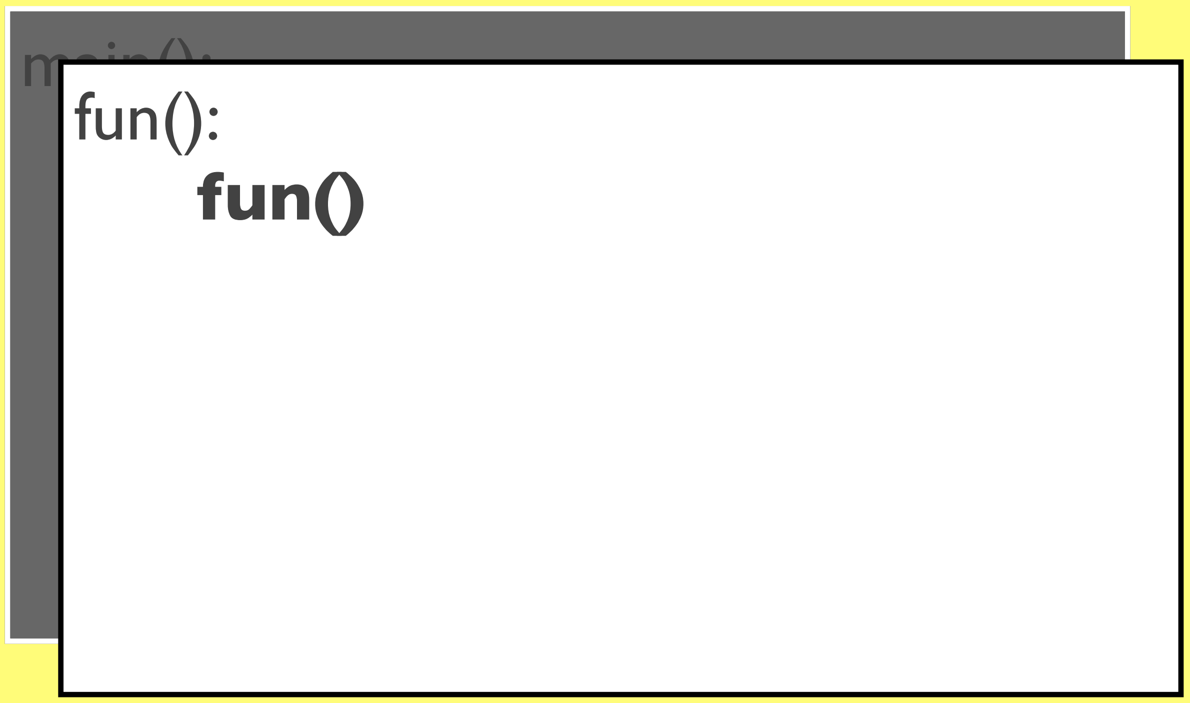
In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.

function definitions:

```
def fun():
    fun()

def main():
    fun()
```



You run main()

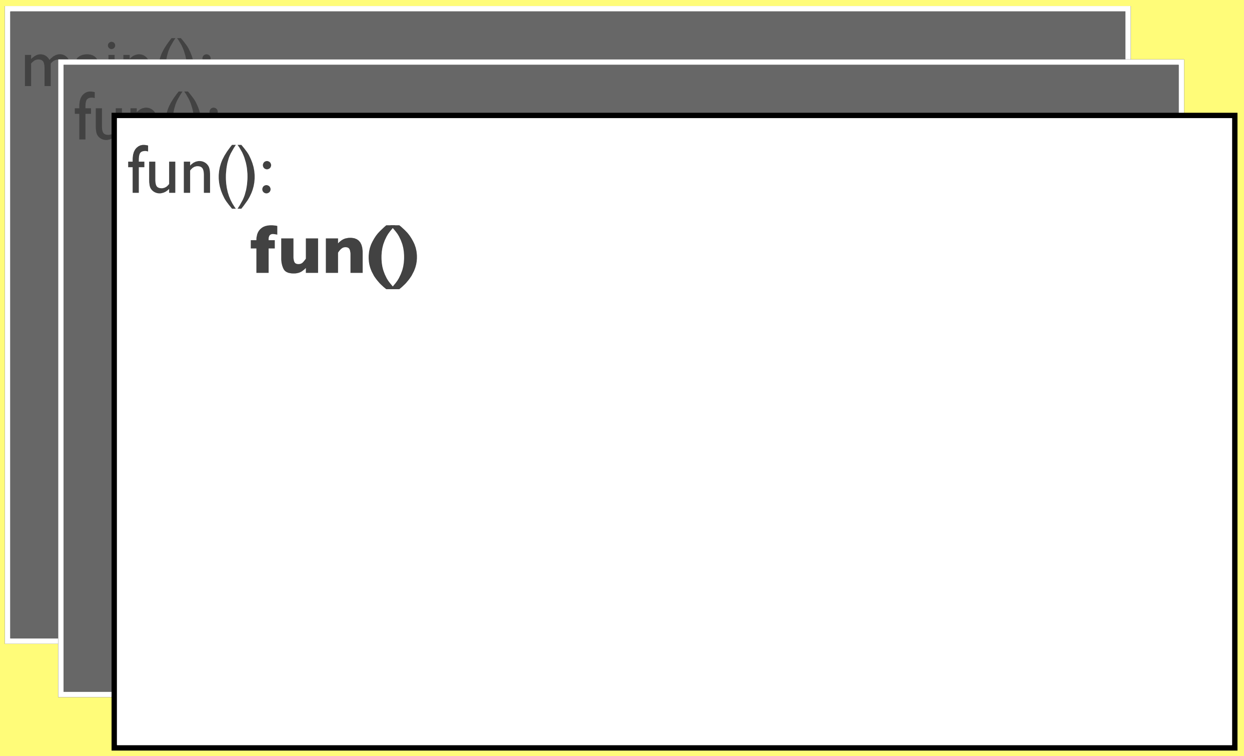
In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.

function definitions:

```
def fun():
      fun()

def main():
      fun()
```



You run main()

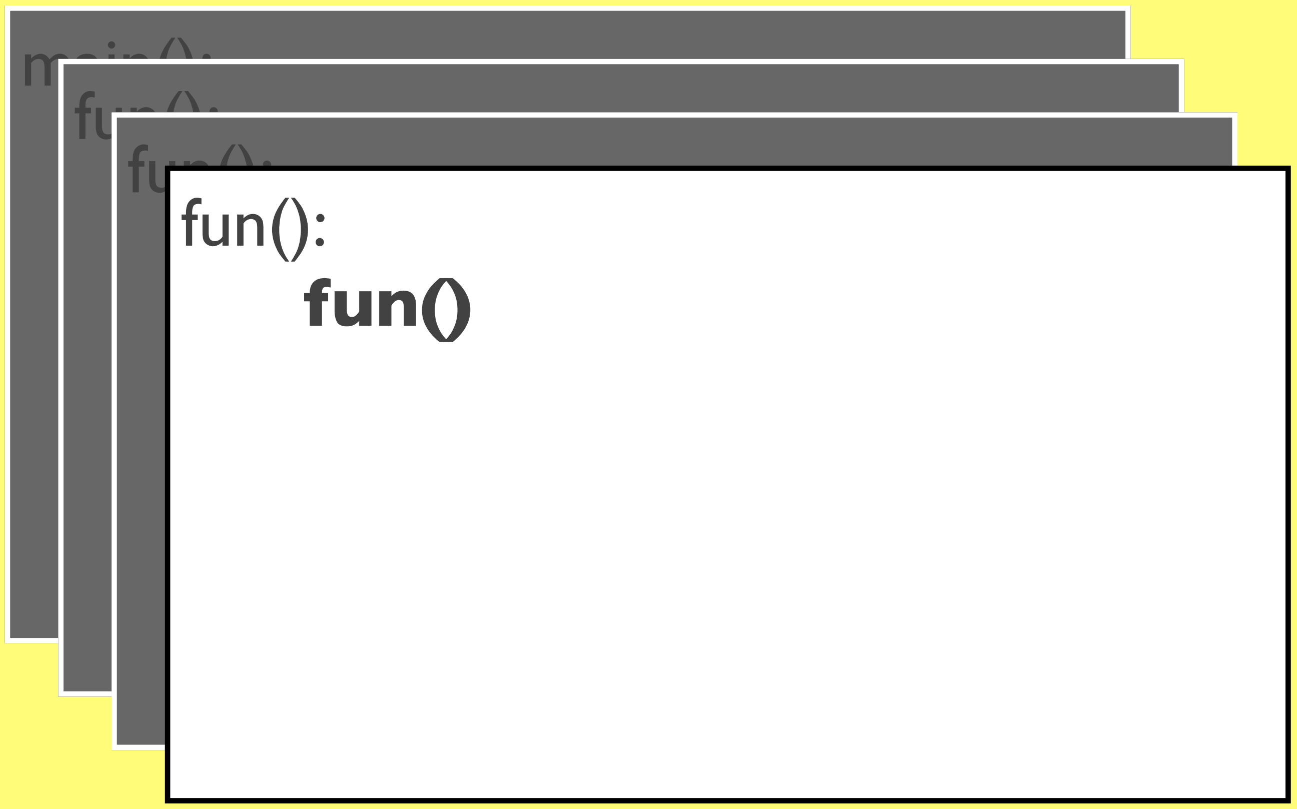
In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.

function definitions:

```
def fun():
    fun()

def main():
    fun()
```



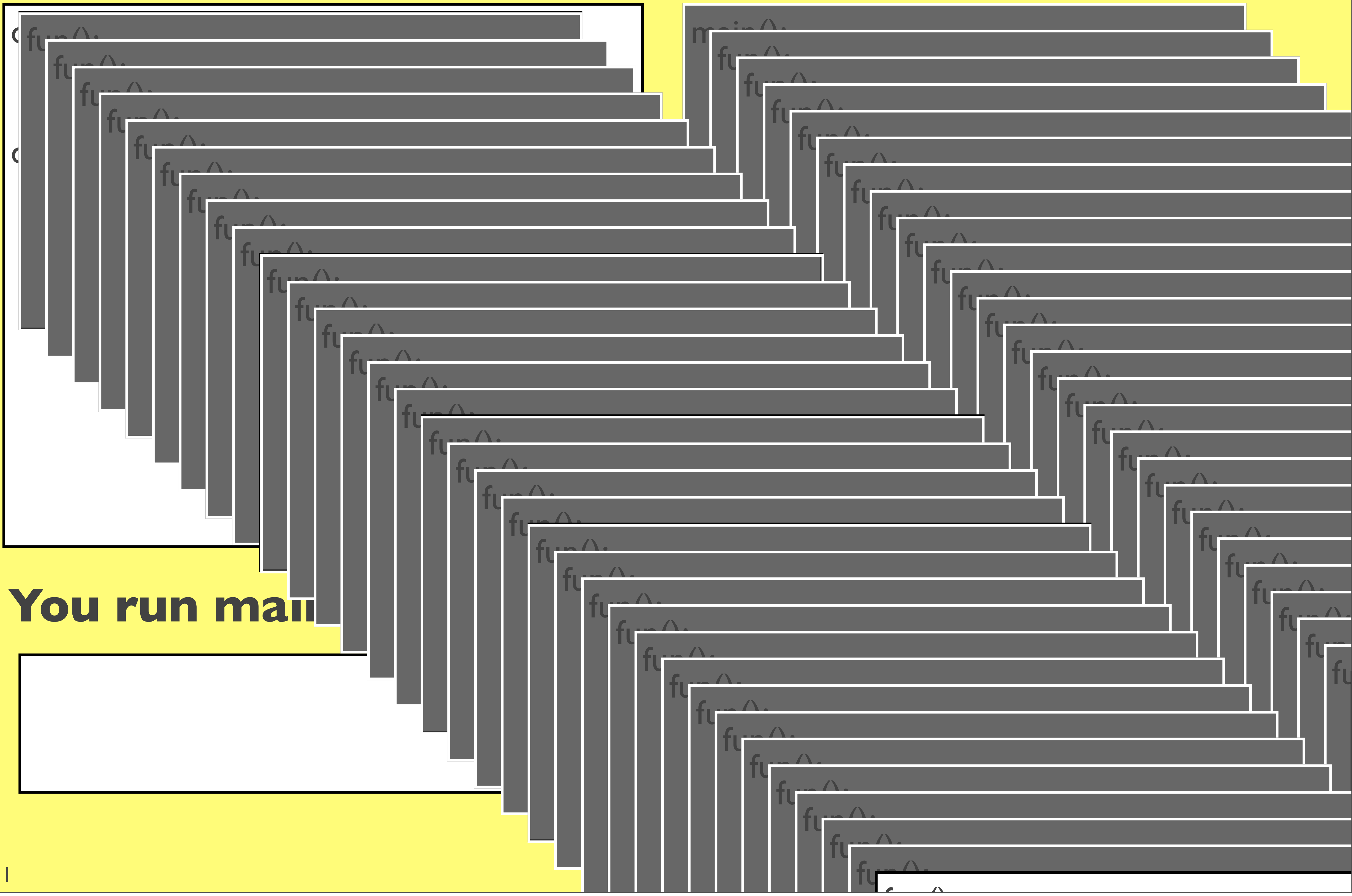
You run main()

In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.

function definitions:

You run mai

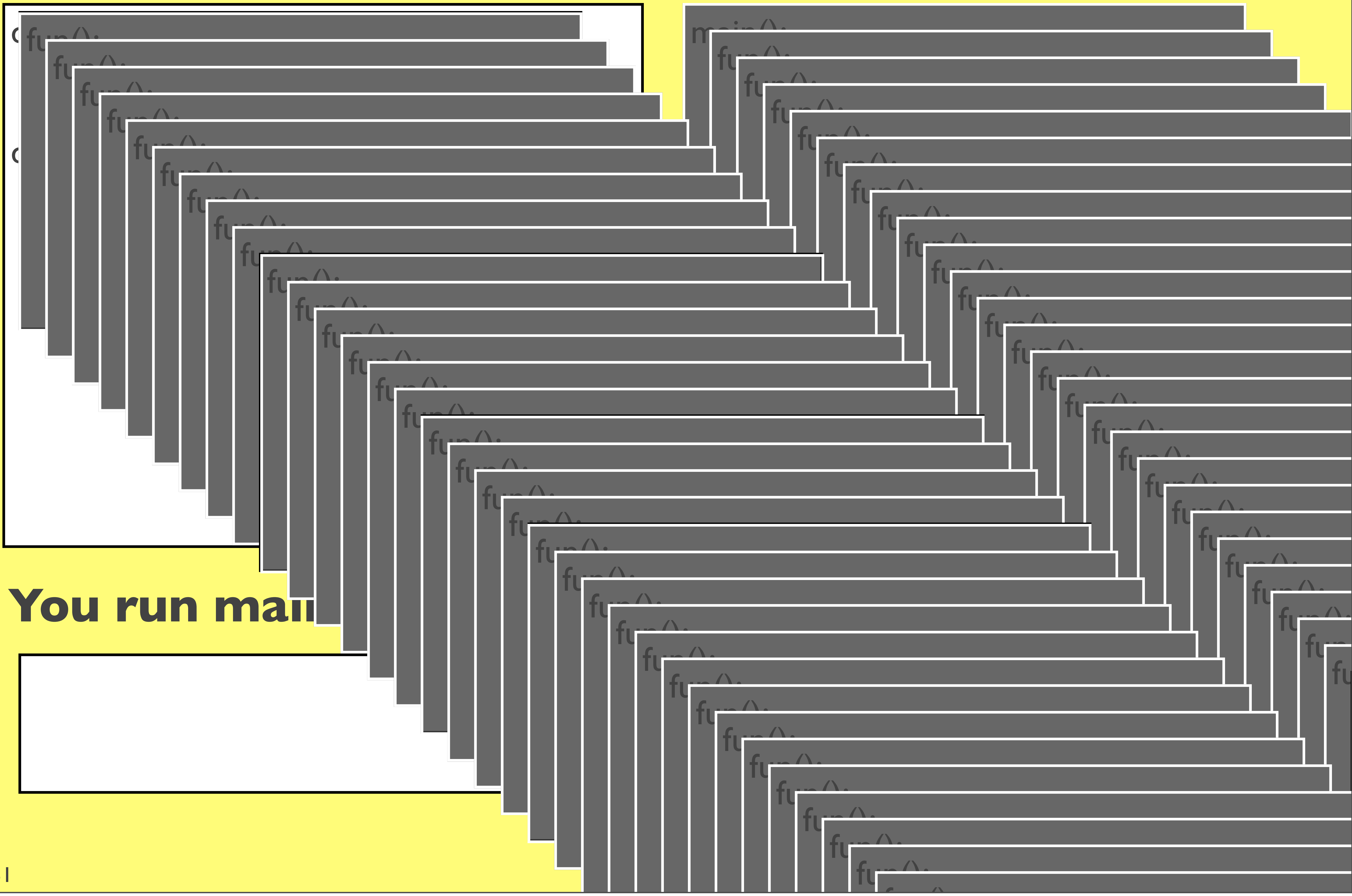


In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.

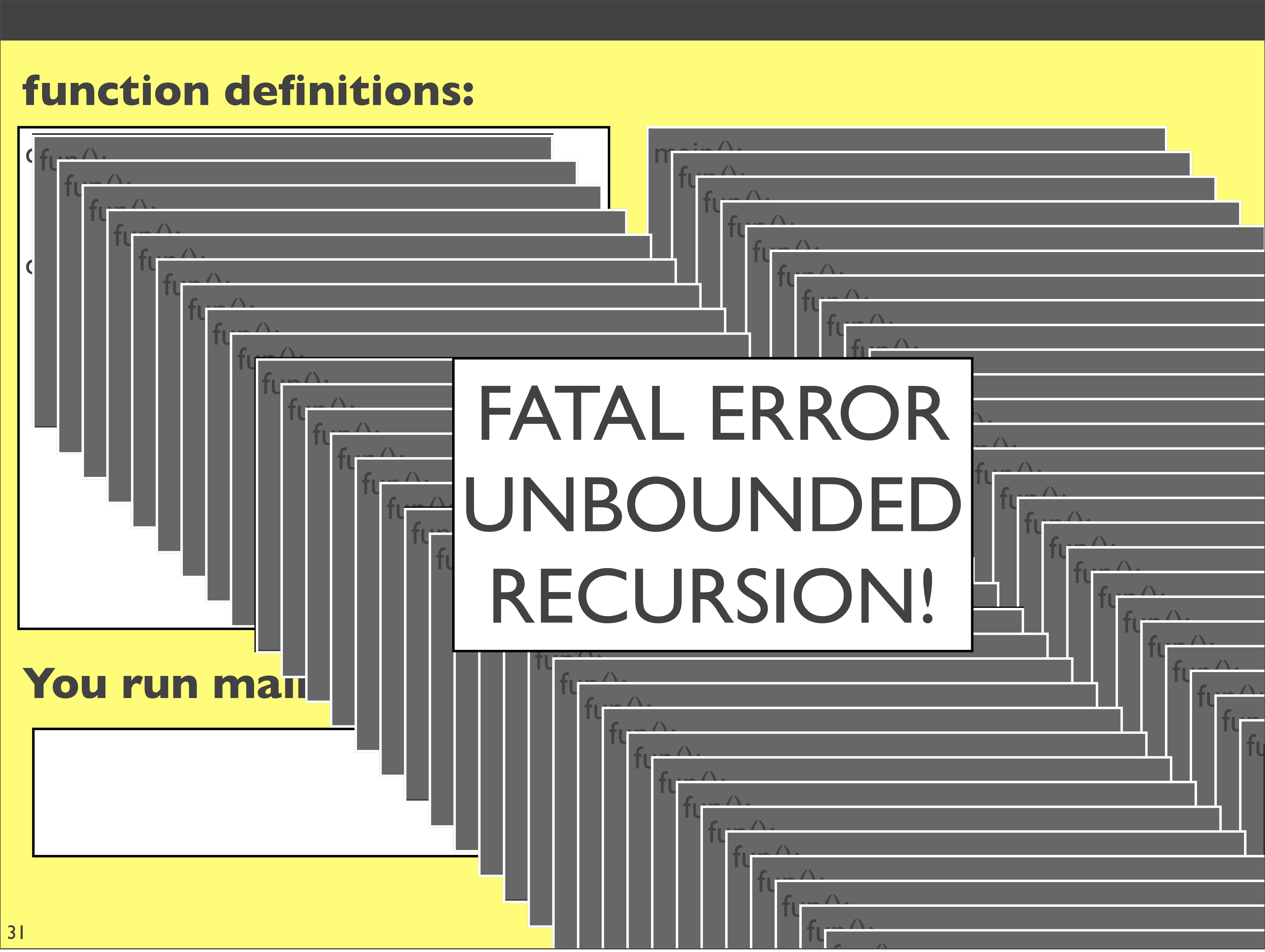
function definitions:

You run mai



In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.



function definitions:

main():

fun():

fun():

fun():

You run mai

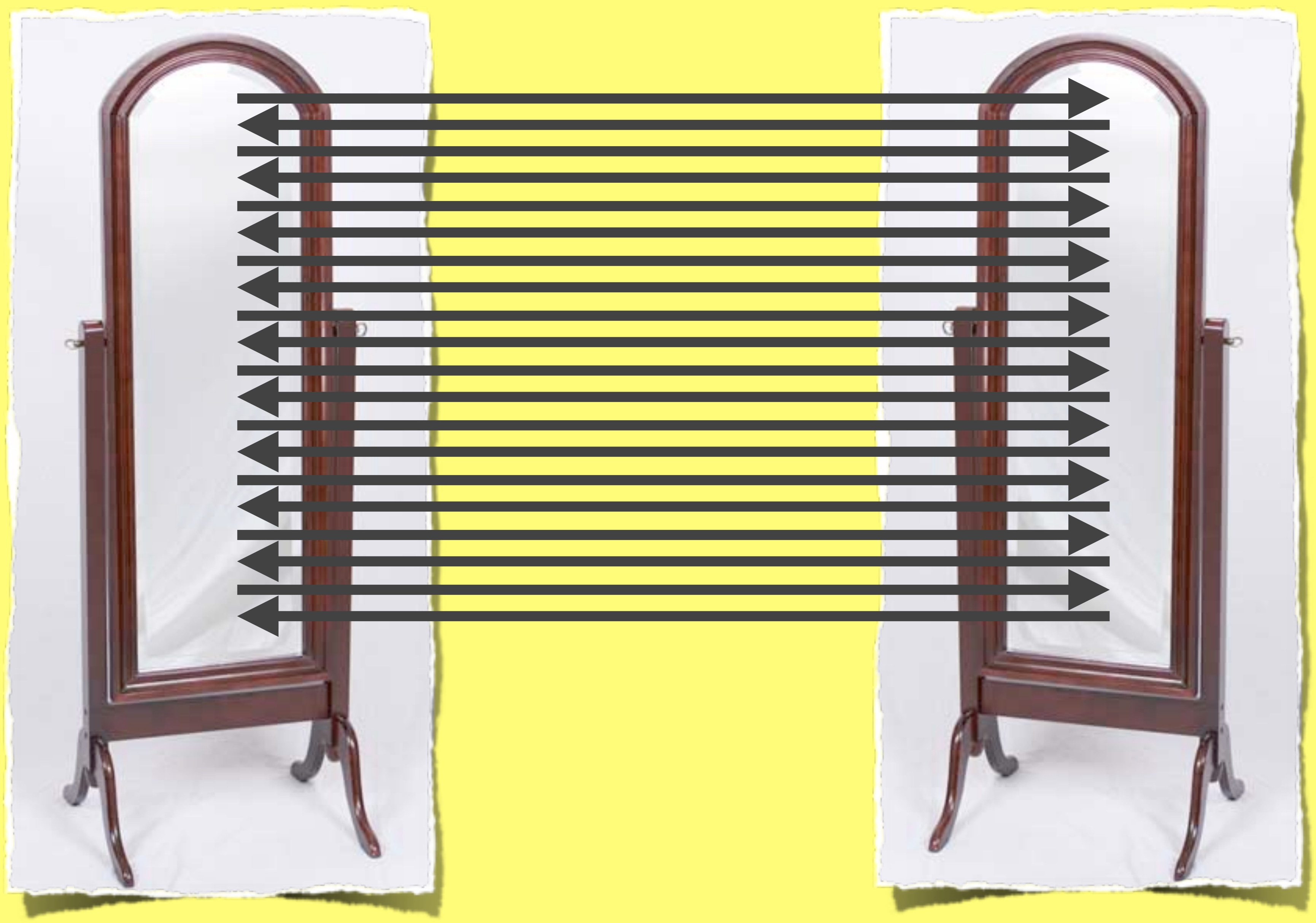
FATAL ERROR
UNBOUNDED
RECURSION!

In the remainder of the talk, we are going to build a function that computes the sum of the first **n** integers. You may recall that this can be computed directly with the function n * (n + 1) / 2 (e.g., the sum of the first 10 integers is 10 * 11 / 2 = 55).

We start with a simple but useless example of recursion. If you ran this program, it would never stop, because each instance of fun() just calls itself again. Eventually your computer will run out of memory and might give you an error like this one.

http://baysideproducts.com/store/images/Barrington%206112.jpg

These functions are a little bit like placing two mirrors opposite each other. What is the image that will be reflected in the first mirror? It's the image in the other mirror. What's the image in the other mirror? Well...



http://baysideproducts.com/store/images/Barrington%206112.jpg

These functions are a little bit like placing two mirrors opposite each other. What is the image that will be reflected in the first mirror? It's the image in the other mirror. What's the image in the other mirror? Well...

function definitions:

```
def fun(x):
      fun(x - 1)

def main():
      fun(10)
```

```
fun(x = 10):
      fun(9)
```

You run main()

scratch pad:

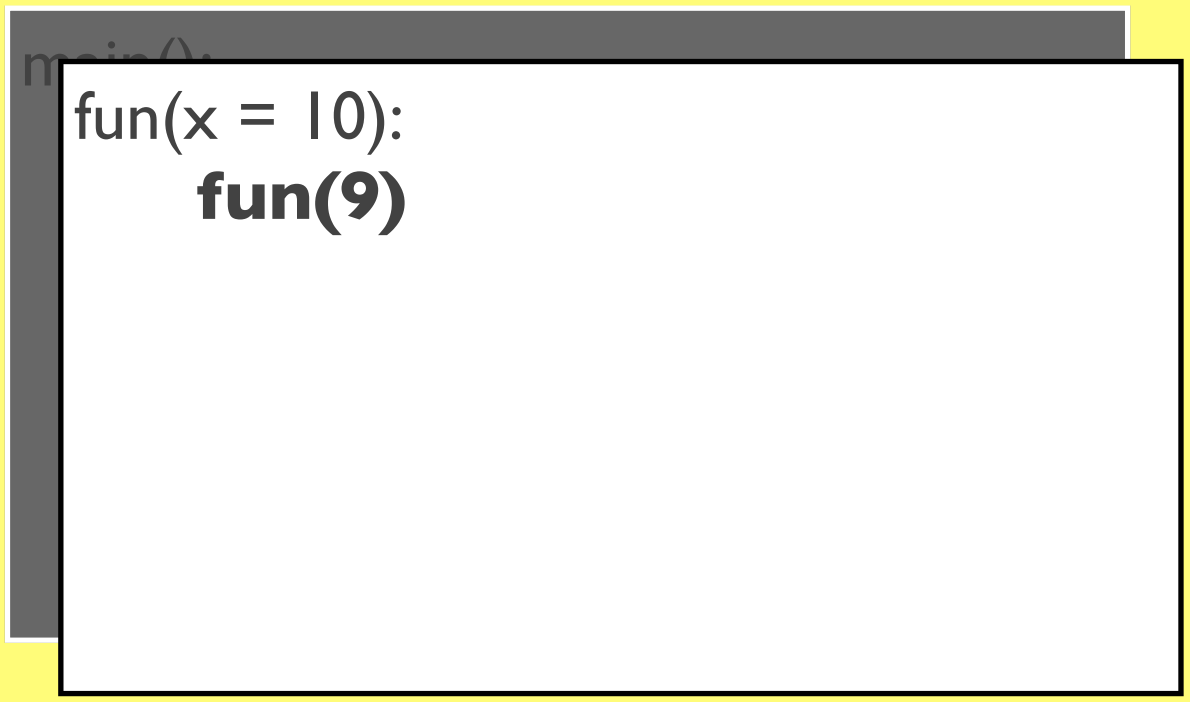
```
      x - 1
=     10 - 1
=     9
```

Let's change this function to be a little more useful. Here each instance of fun() will change the value of the parameter it is calling with.

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



You run main()

scratch pad:

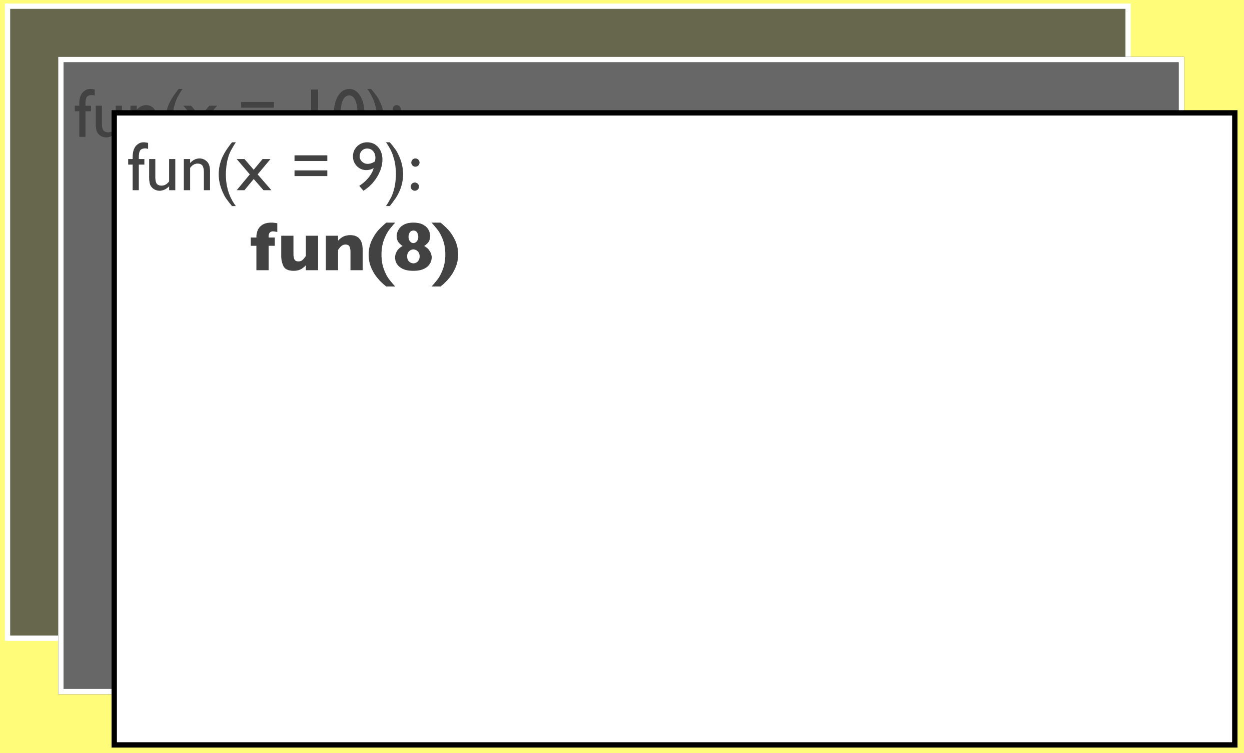
```
    x - 1
=   10 - 1
=   9
```

Let’s change this function to be a little more useful. Here each instance of fun() will change the value of the parameter it is calling with.

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 9):
    fun(8)
```

You run main()

scratch pad:

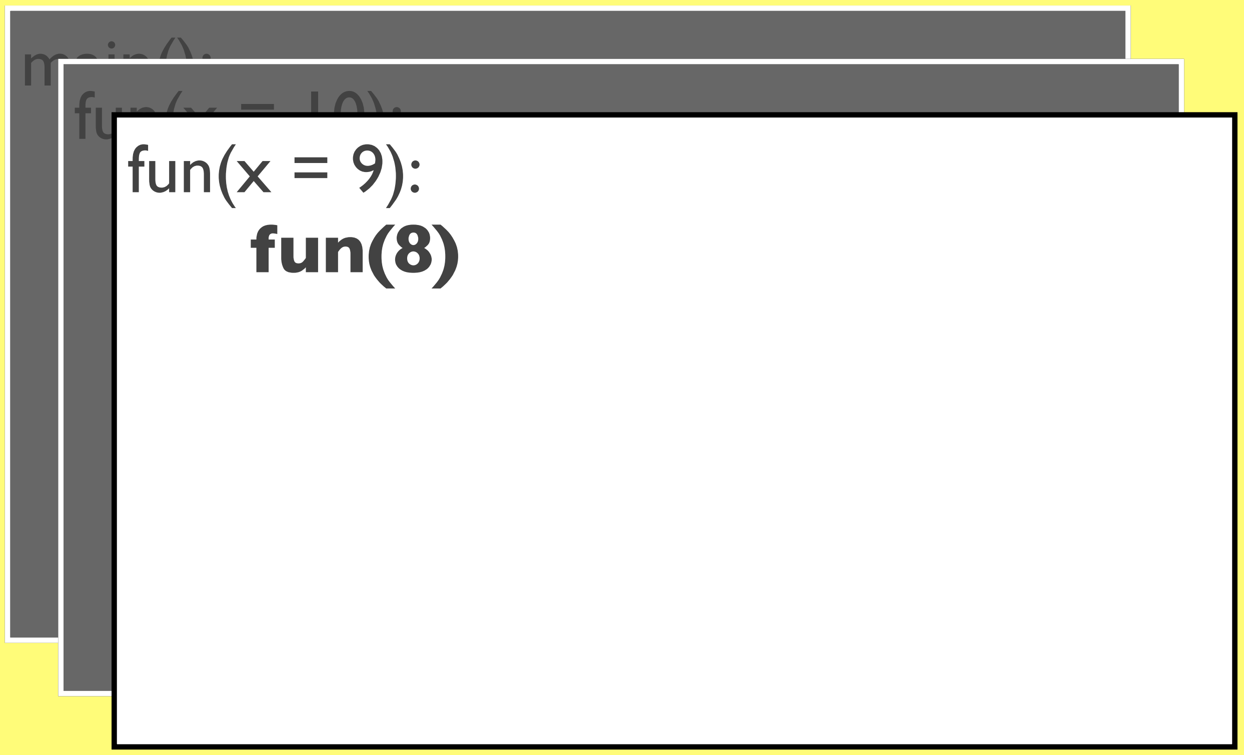
```
    x - 1
=   9 - 1
=   8
```

The value gets smaller

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
main():
fun(x = 10):
fun(x = 9):
    fun(8)
```

You run main()

scratch pad:

```
    x - 1
=   9 - 1
=   8
```

The value gets smaller

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```

```
fun(x = 8):
    fun(7)
```

You run main()

scratch pad:

```
    x - 1
=   8 - 1
=   7
```

and smaller

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```

```
main():
fun(x = 10):
fun(x = 9):
fun(x = 8):
    fun(7)
```

You run main()

scratch pad:

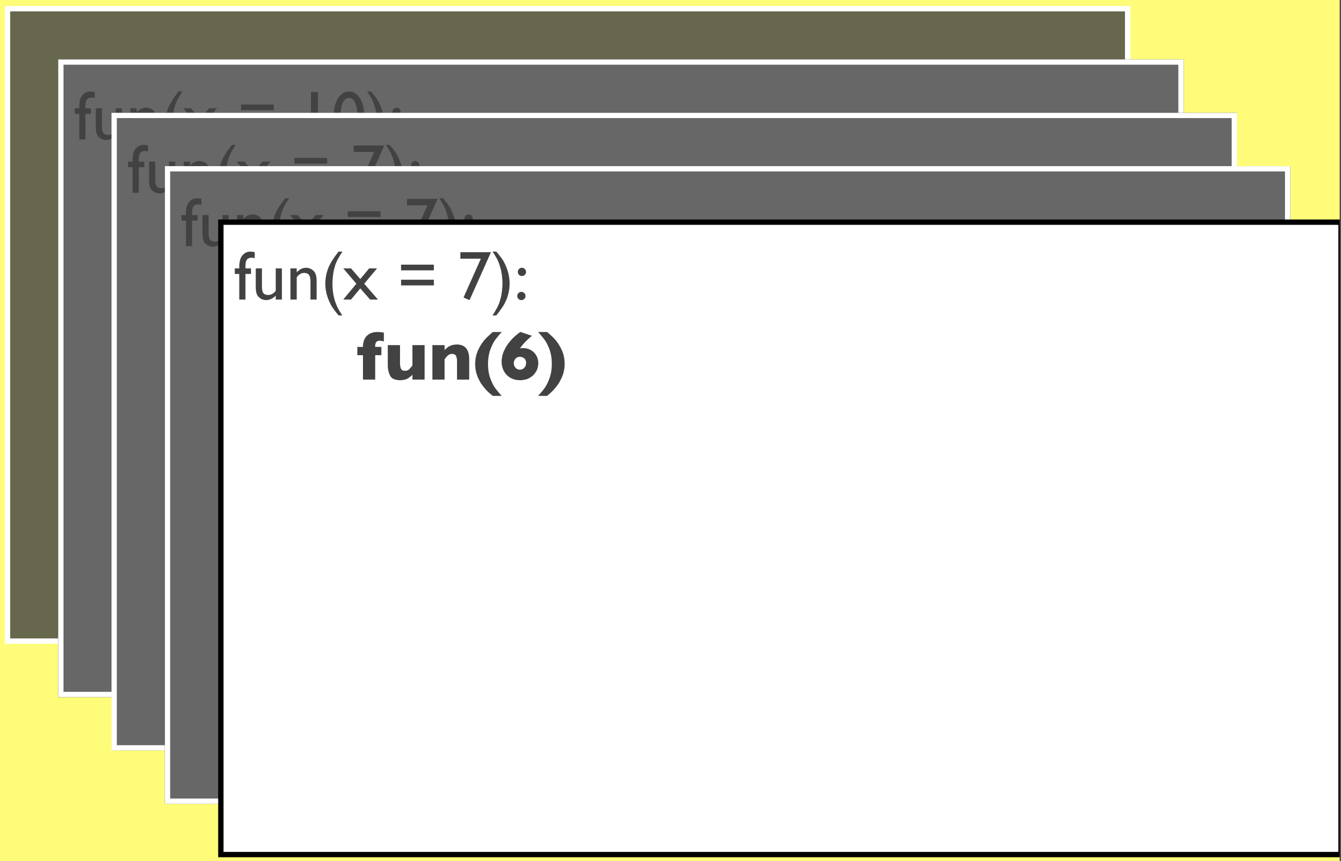
```
    x - 1
=   8 - 1
=   7
```

and smaller

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 7):
    fun(6)
```

You run main()

scratch pad:

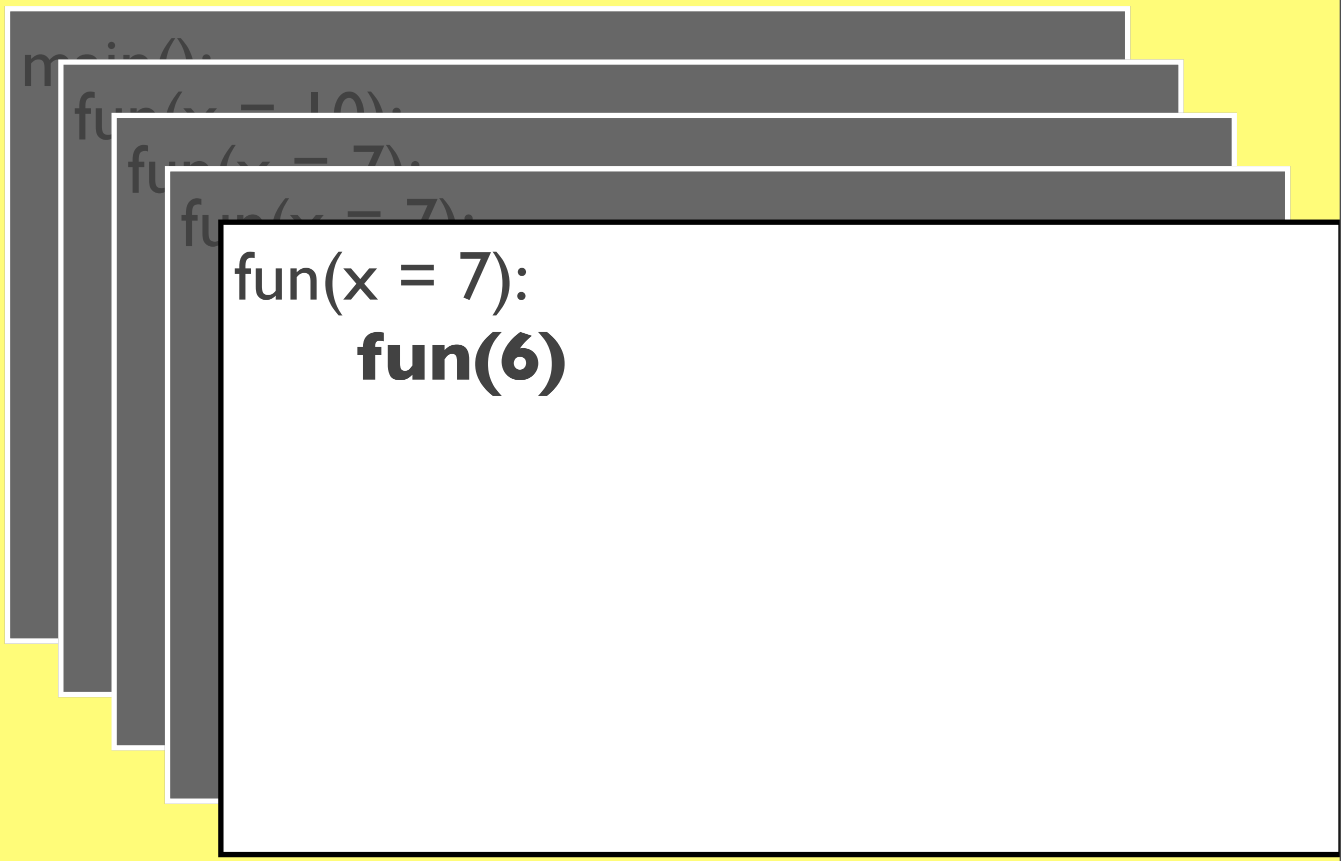
```
    x - 1
=   7 - 1
=   6
```

etc

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



You run main()

scratch pad:

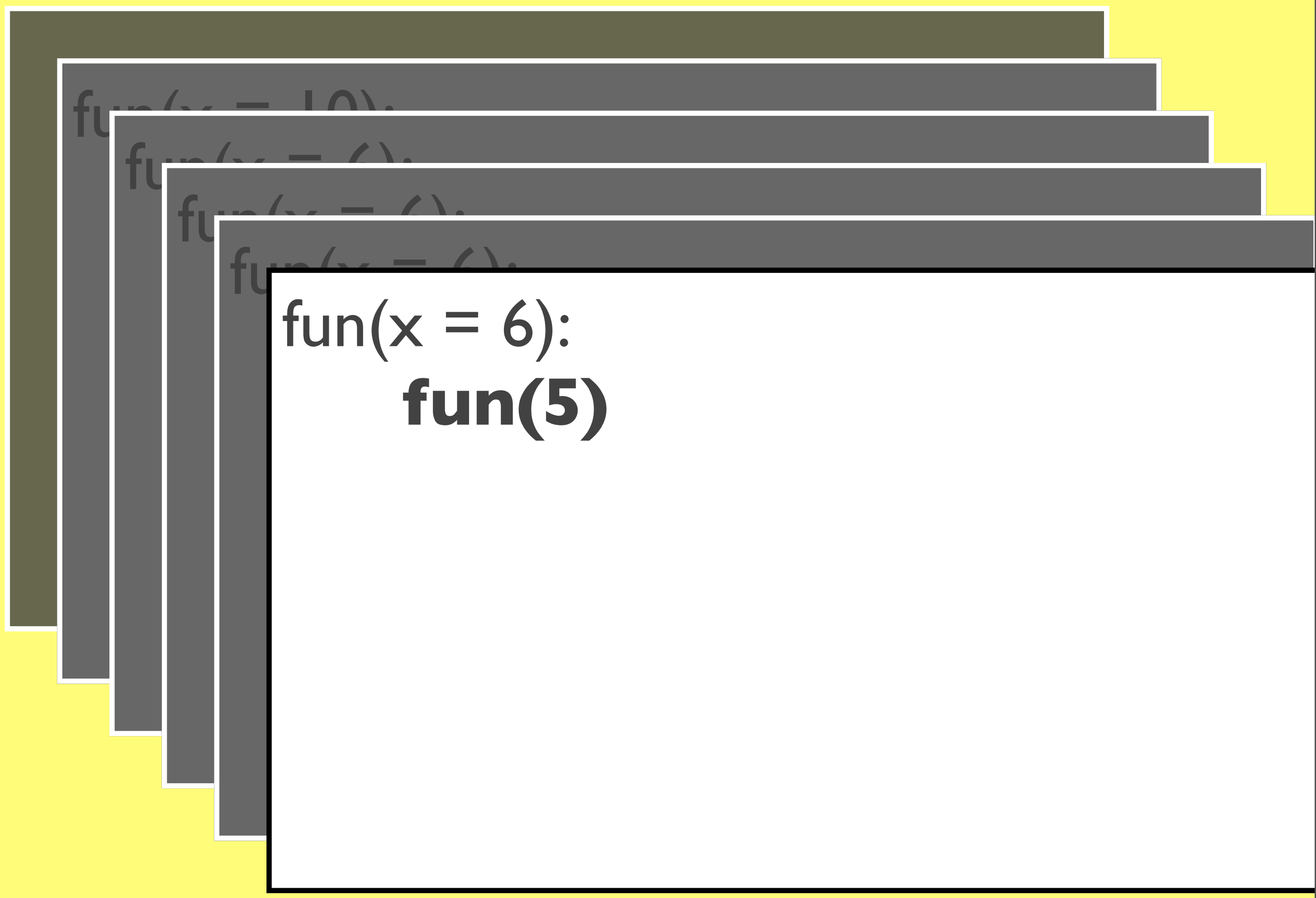
```
     x - 1
=    7 - 1
=    6
```

etc

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 6):
    fun(5)
```

You run main()

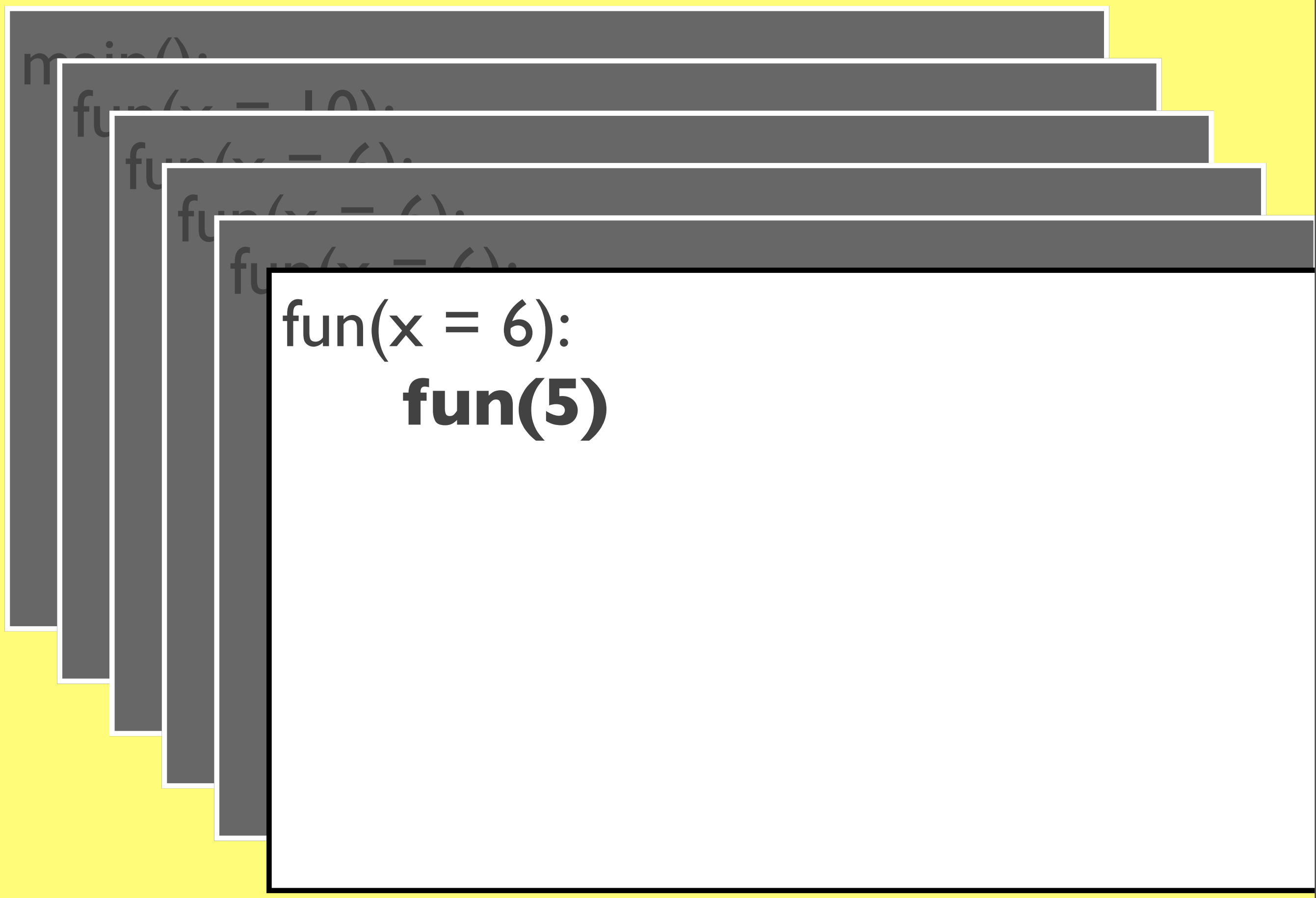
scratch pad:

```
    x - 1
=   6 - 1
=   5
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 6):
    fun(5)
```

You run main()

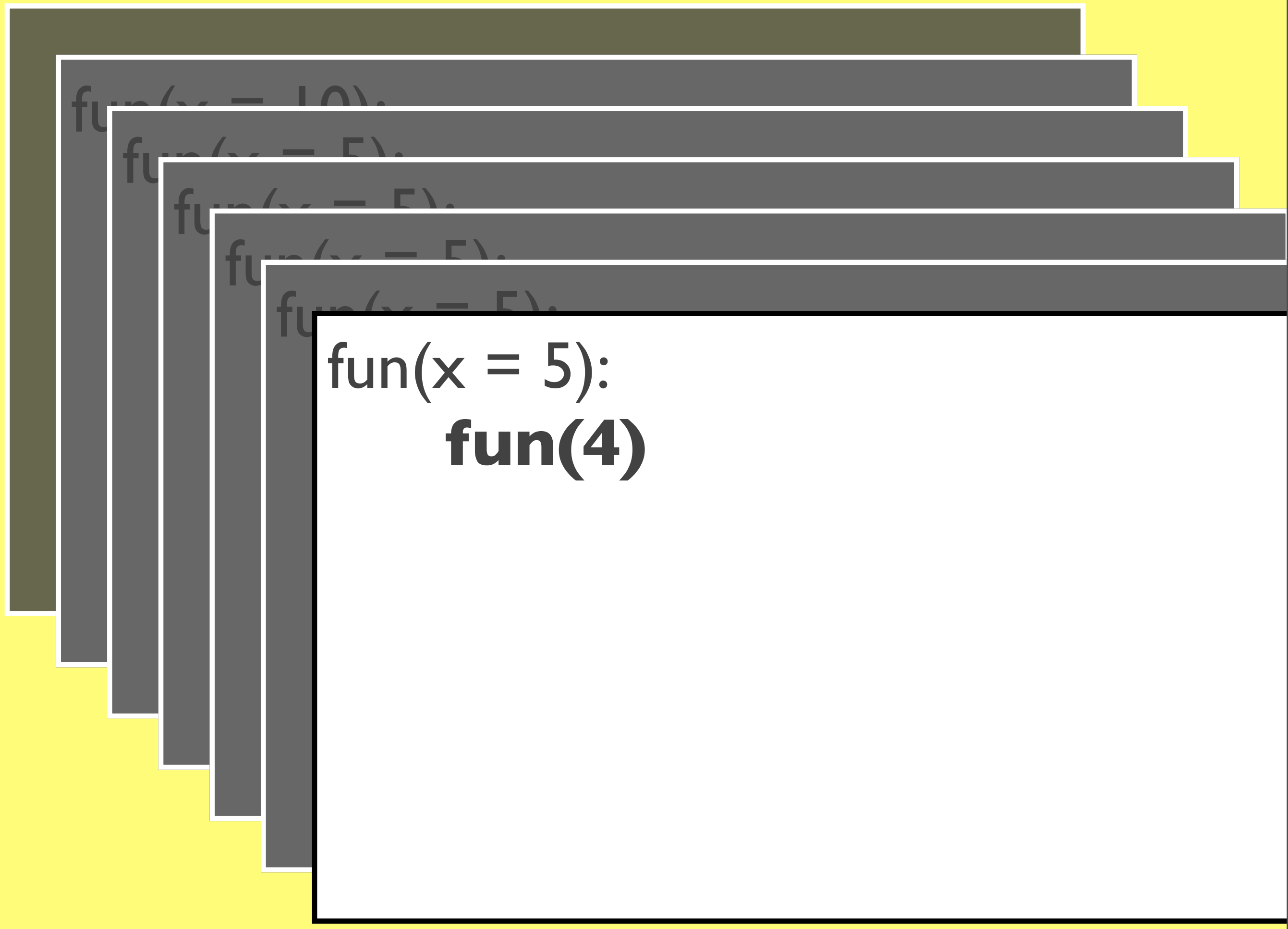
scratch pad:

```
    x - 1
=   6 - 1
=   5
```

function definitions:

```
def fun(x):
      fun(x - 1)

def main():
      fun(10)
```



```
fun(x = 5):
      fun(4)
```

You run main()

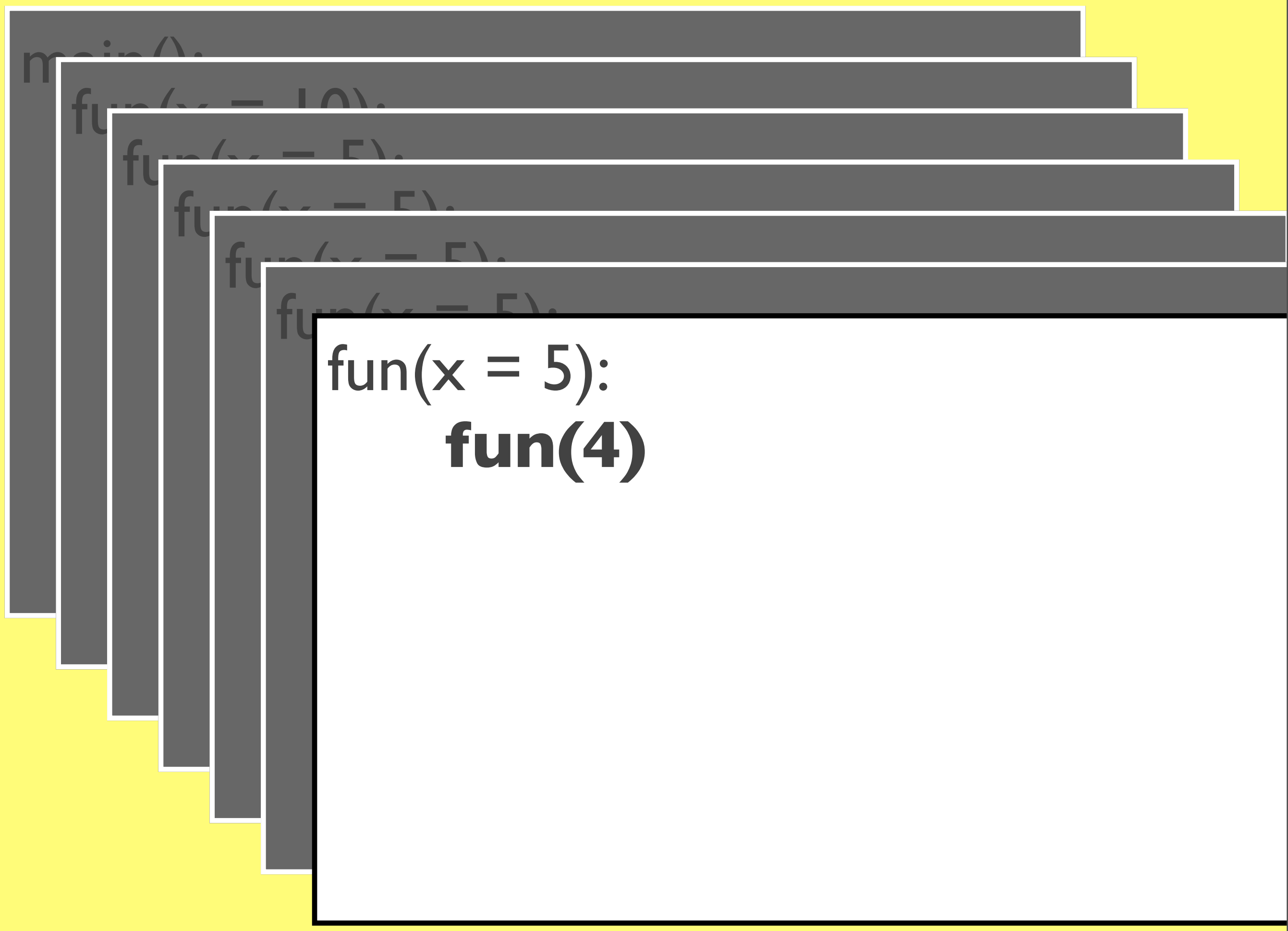
scratch pad:

```
     x - 1
=    5 - 1
=    4
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



You run main()

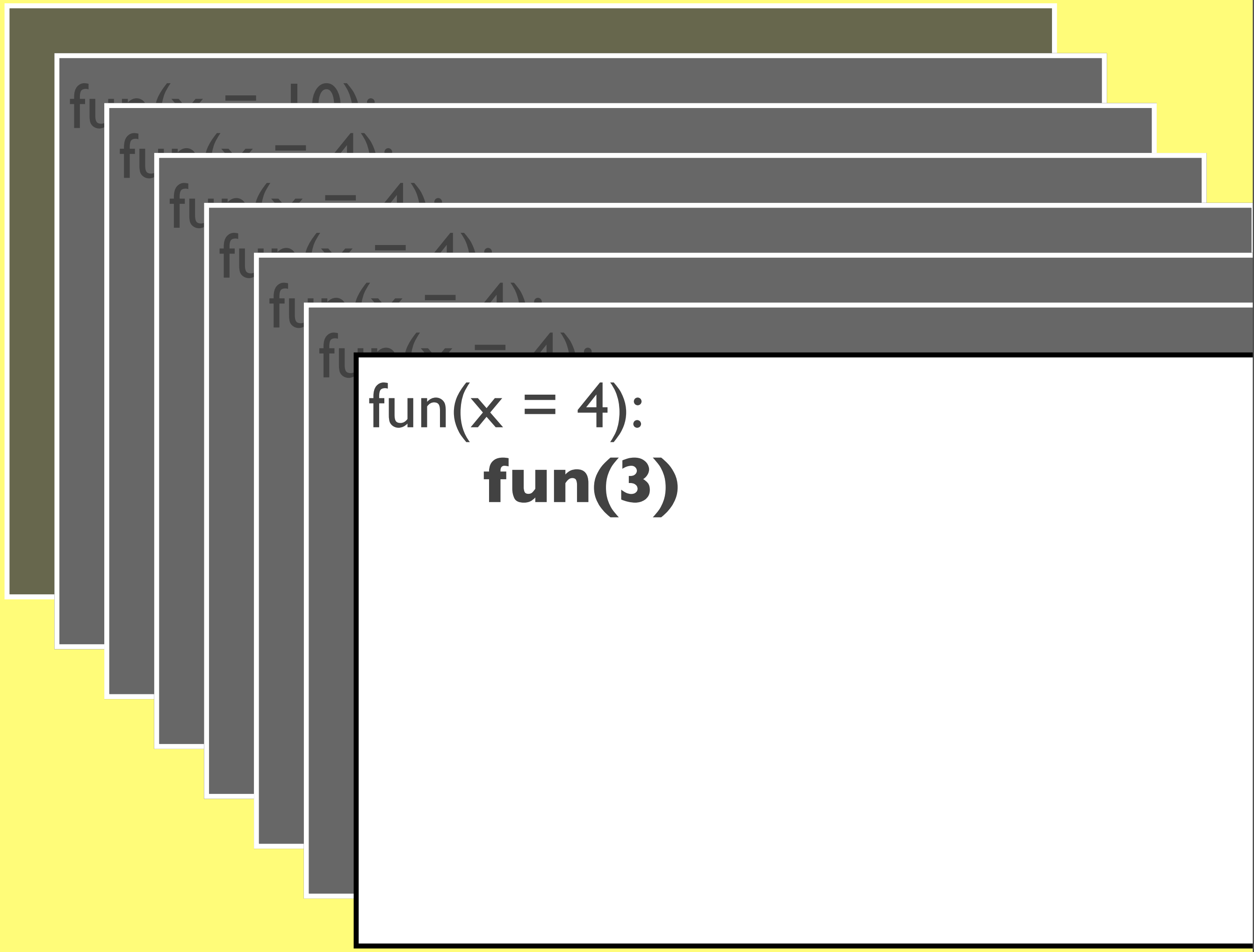
scratch pad:

```
     x - 1
=    5 - 1
=    4
```

function definitions:

```
def fun(x):
      fun(x - 1)

def main():
      fun(10)
```



```
fun(x = 4):
      fun(3)
```

You run main()

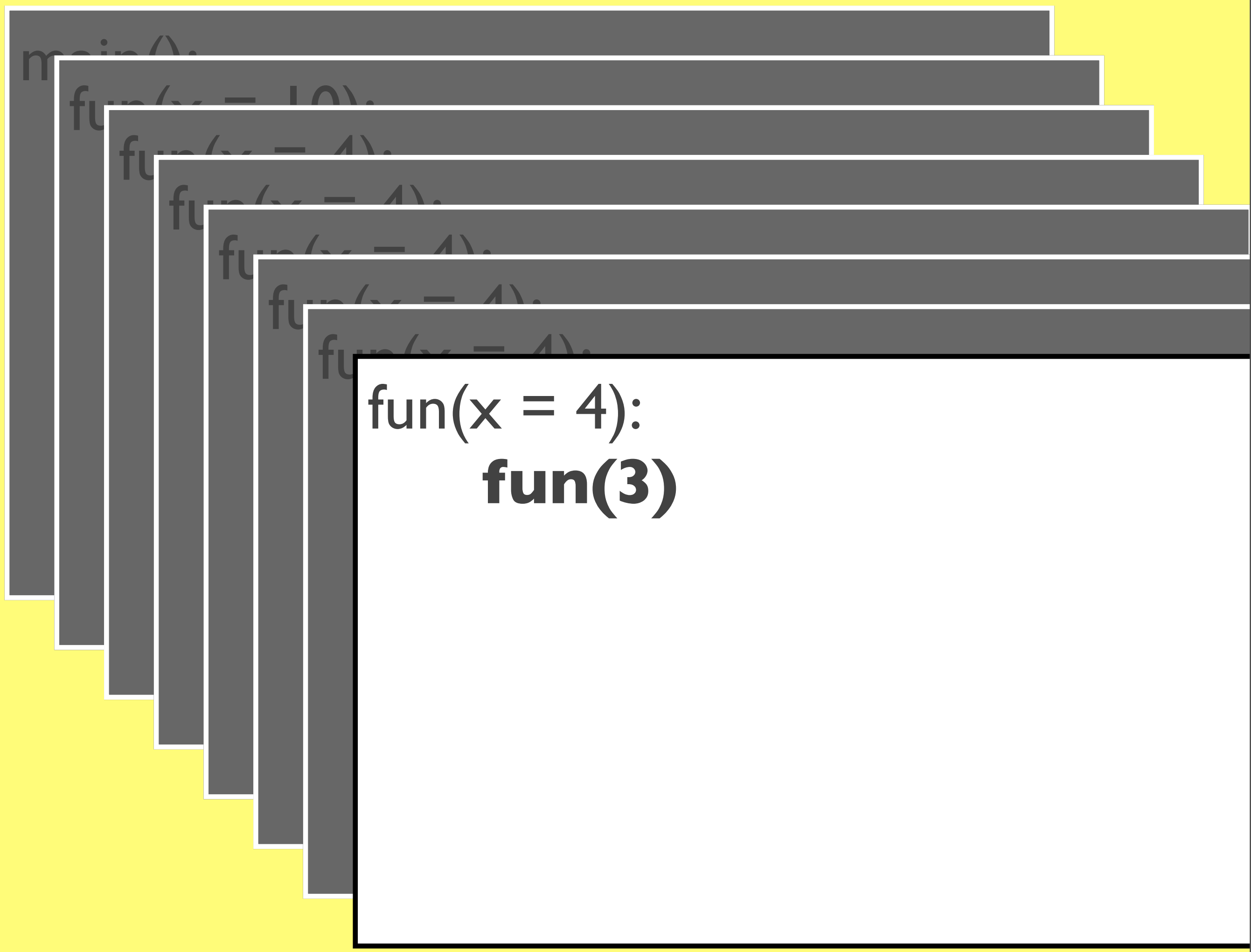
scratch pad:

```
     x - 1
=    4 - 1
=    3
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



You run main()

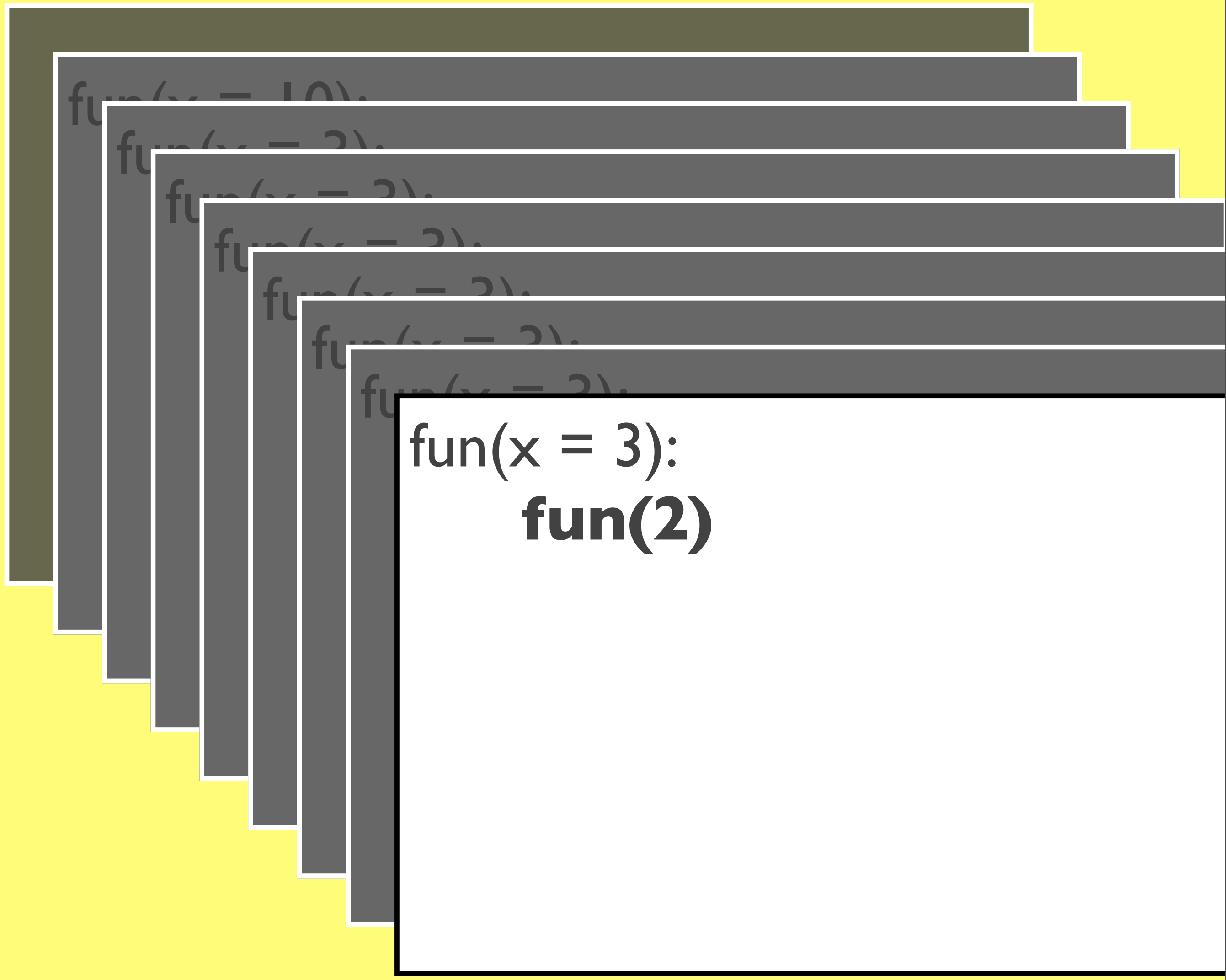
scratch pad:

```
    x - 1
=   4 - 1
=   3
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



You run main()

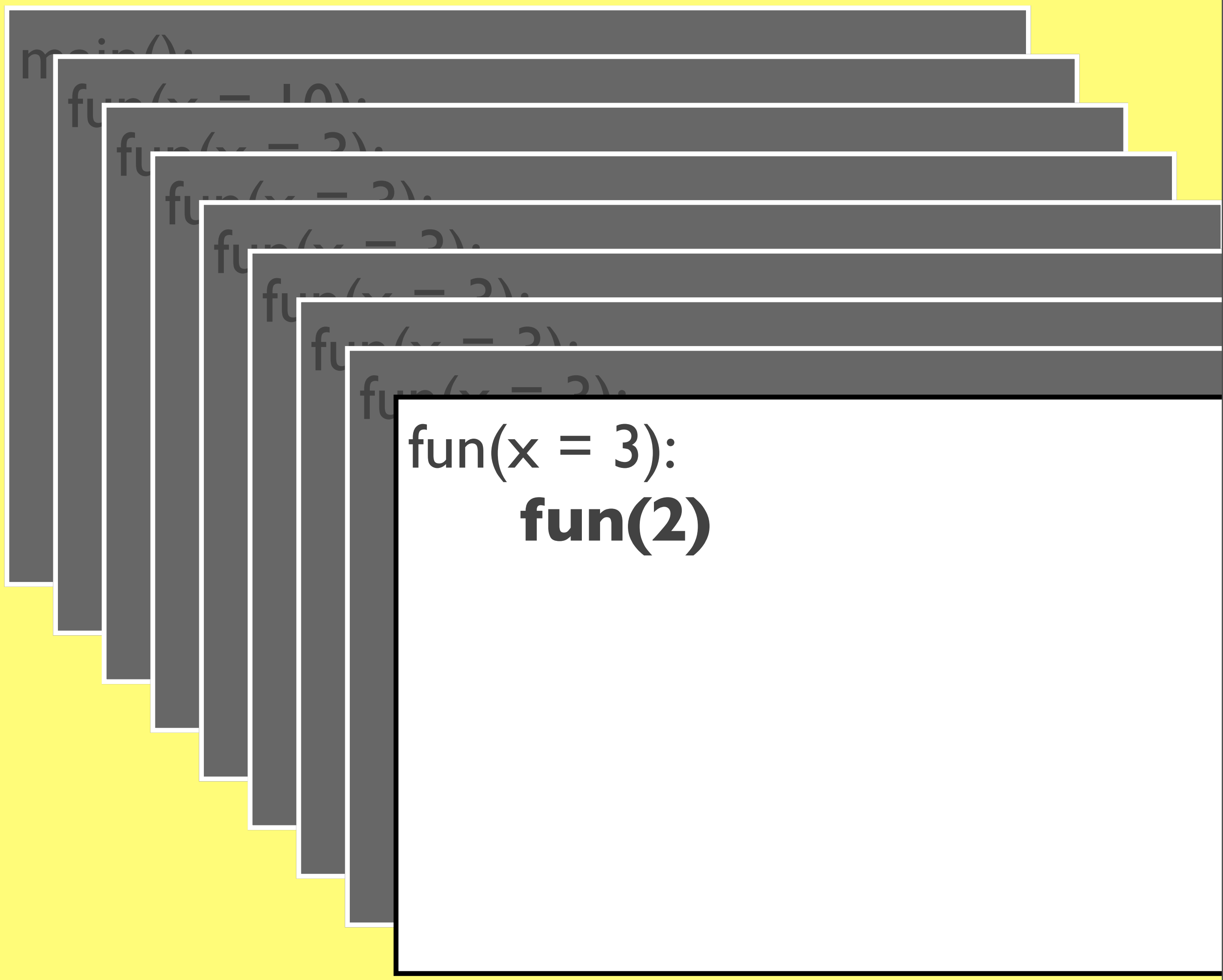
scratch pad:

```
    x - 1
=   3 - 1
=   2
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 3):
    fun(2)
```

You run main()

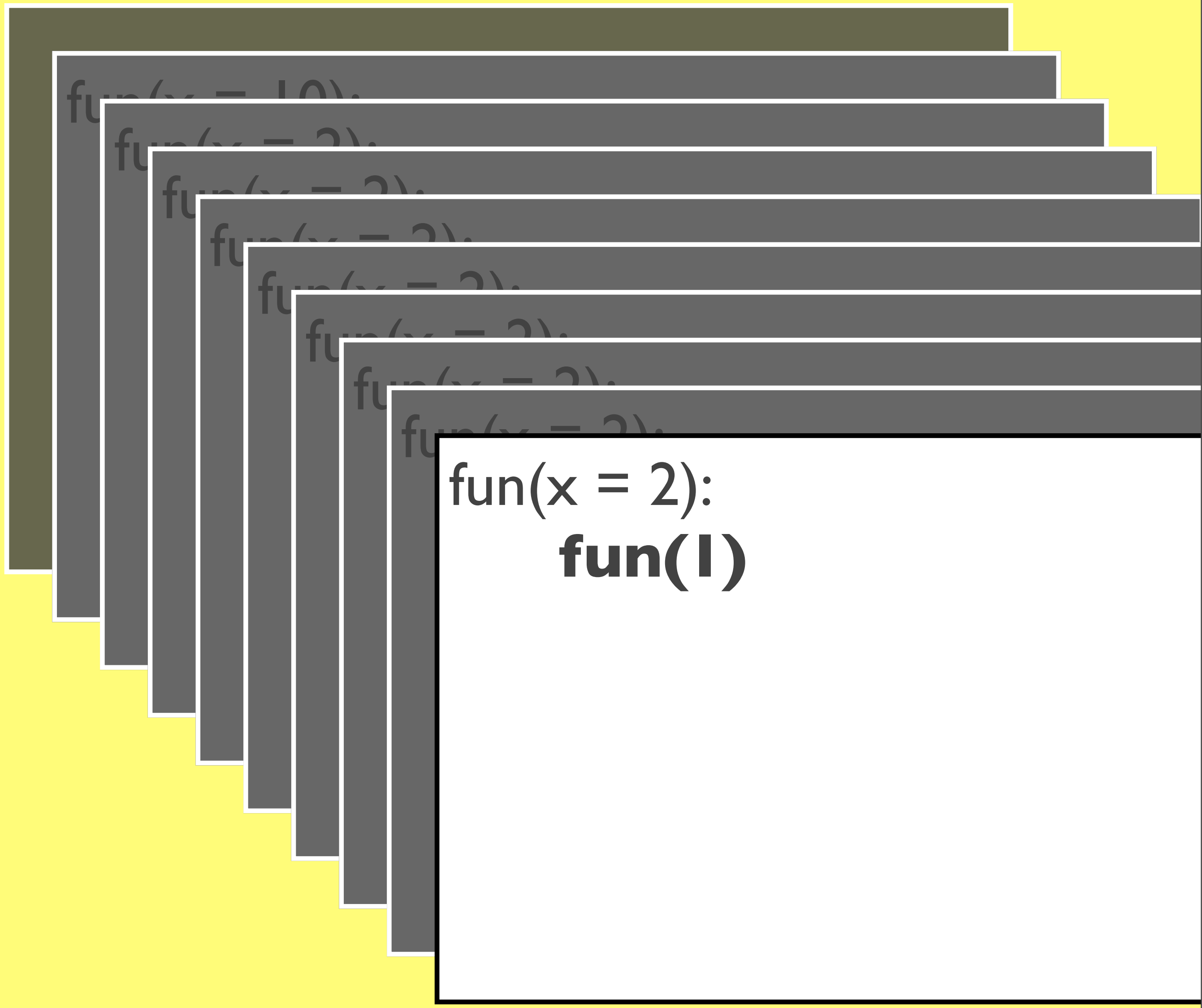
scratch pad:

```
    x - 1
=   3 - 1
=   2
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



You run main()

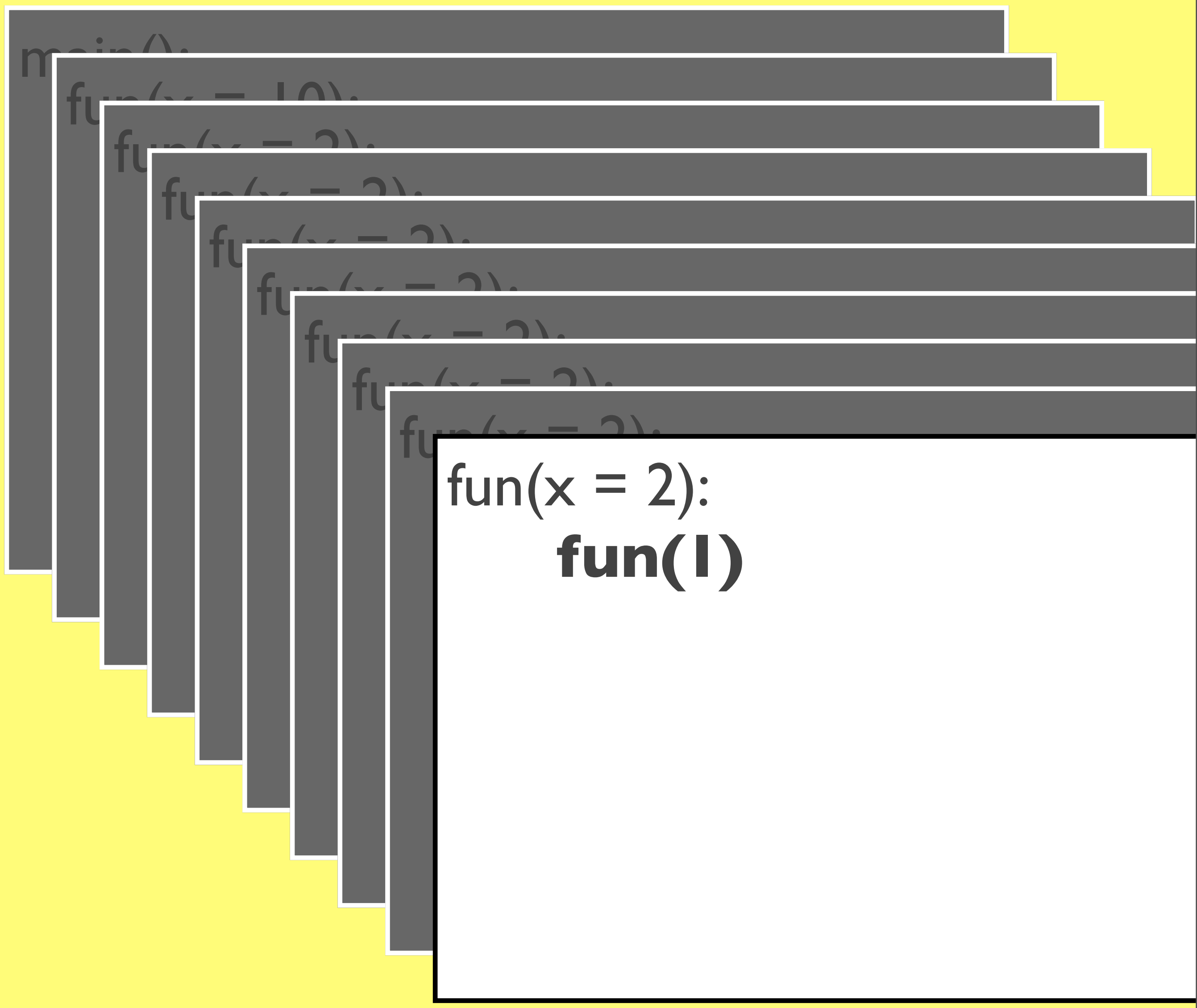
scratch pad:

```
    x - 1
=   2 - 1
=   1
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



You run main()

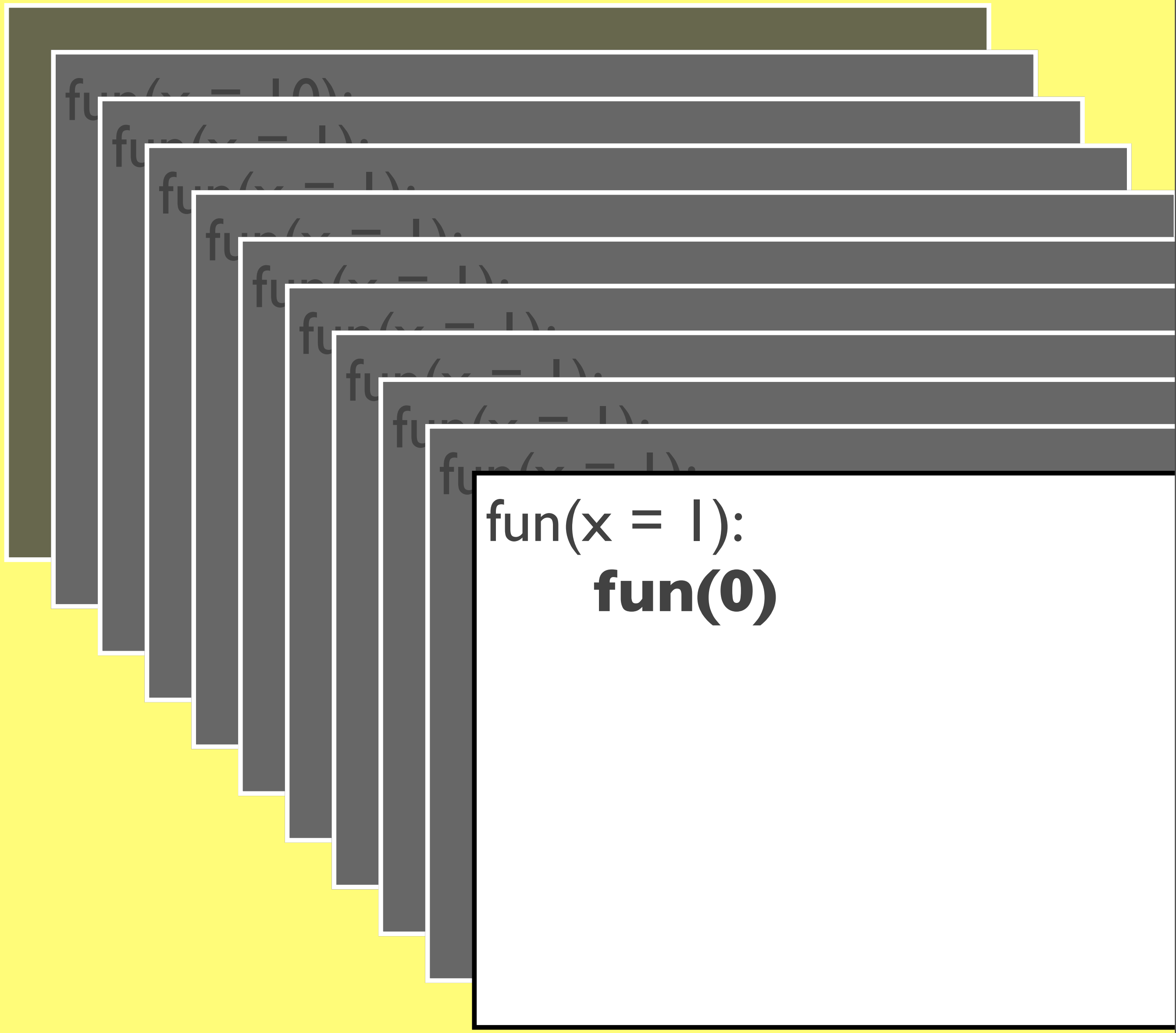
scratch pad:

```
     x - 1
=    2 - 1
=    1
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 1):
    fun(0)
```

You run main()

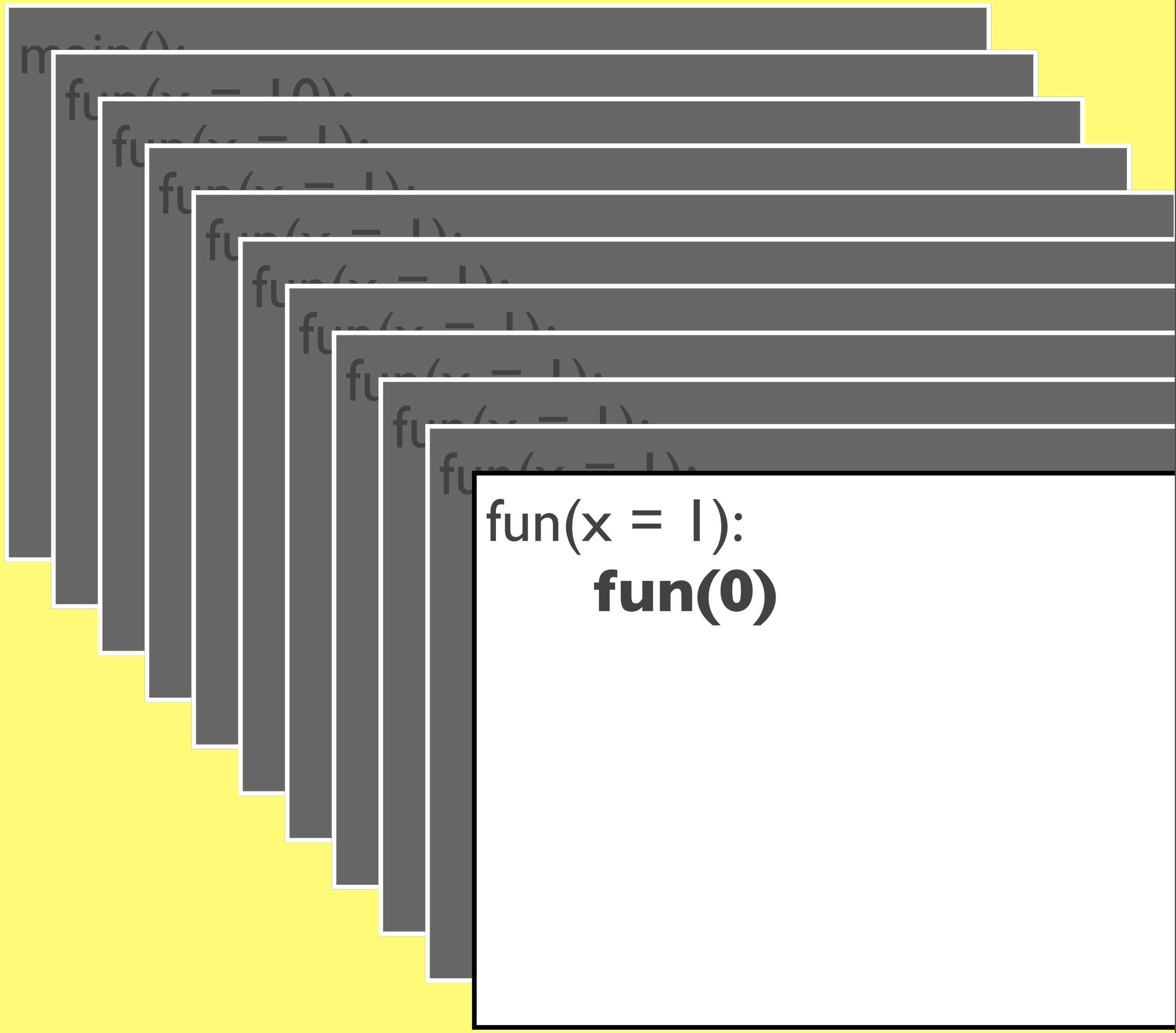
scratch pad:

```
    x - 1
=   1 - 1
=   0
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 1):
    fun(0)
```

You run main()

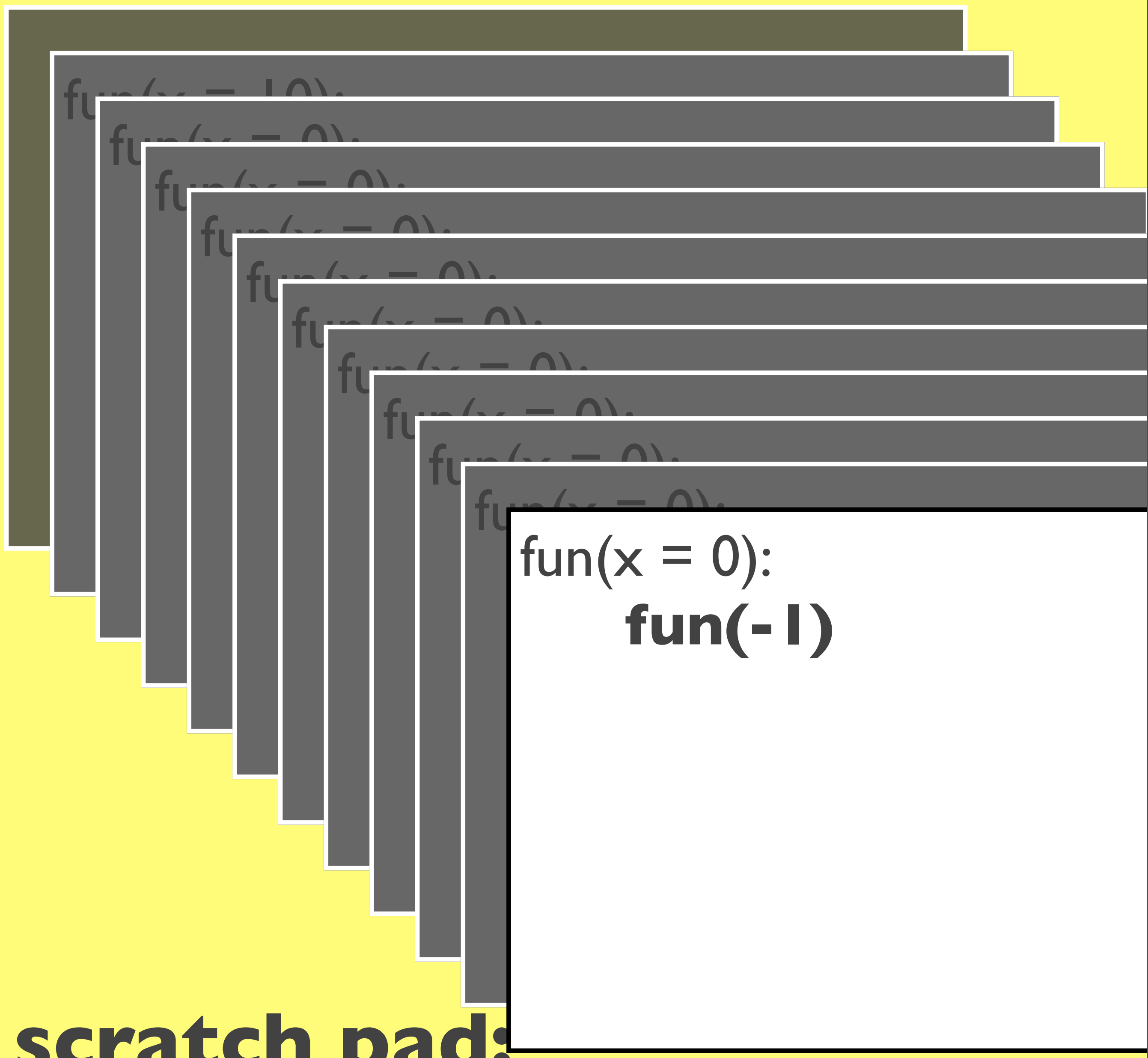
scratch pad:

```
    x - 1
=   1 - 1
=   0
```

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



```
fun(x = 0):
    fun(-1)
```

You run main()

scratch pad:

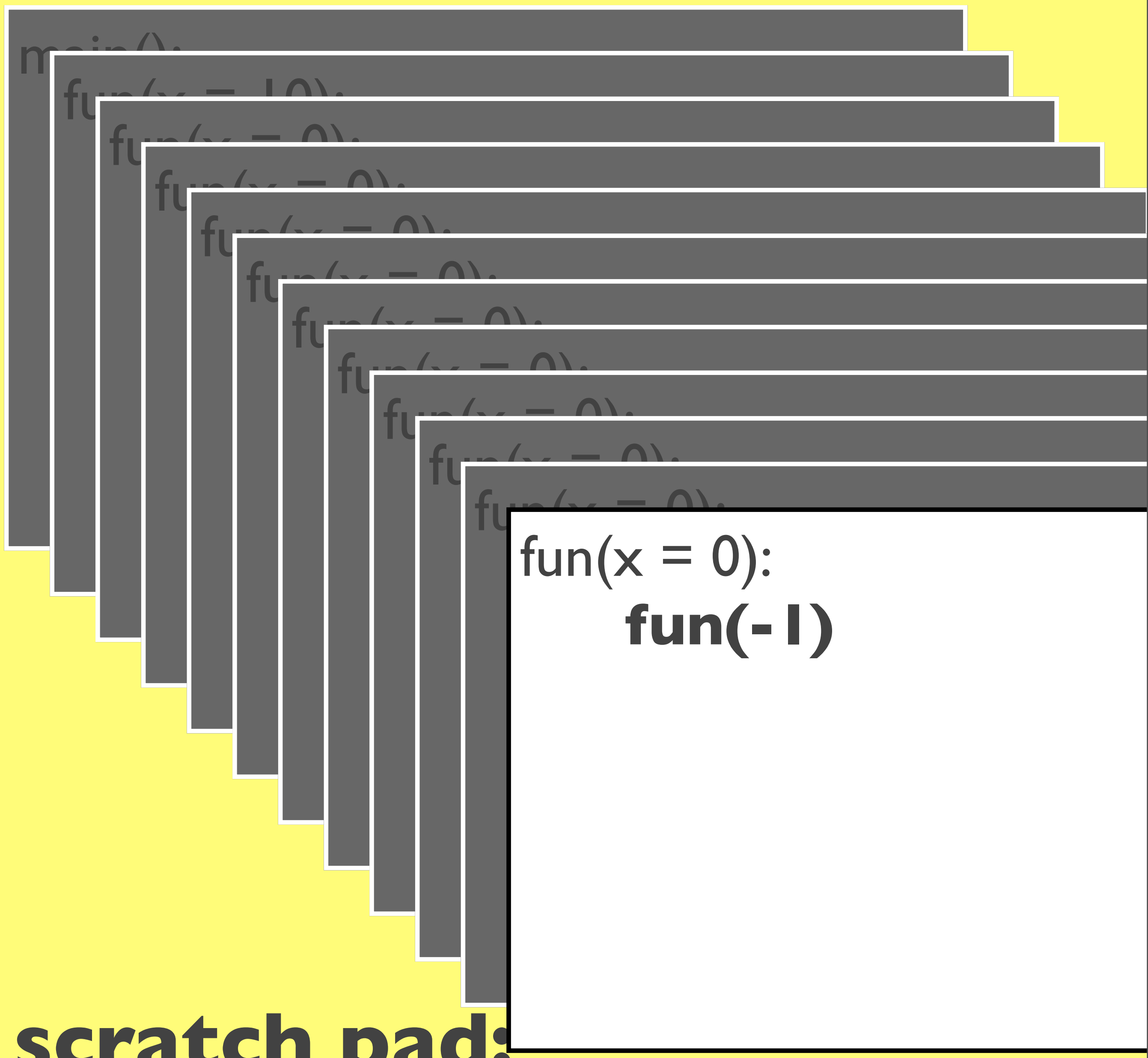
```
     x - 1
=    0 - 1
=    -1
```

but again, it never stops, and we get another instance of unbounded recursion.

function definitions:

```
def fun(x):
	fun(x - 1)

def main():
	fun(10)
```



You run main()

scratch pad:

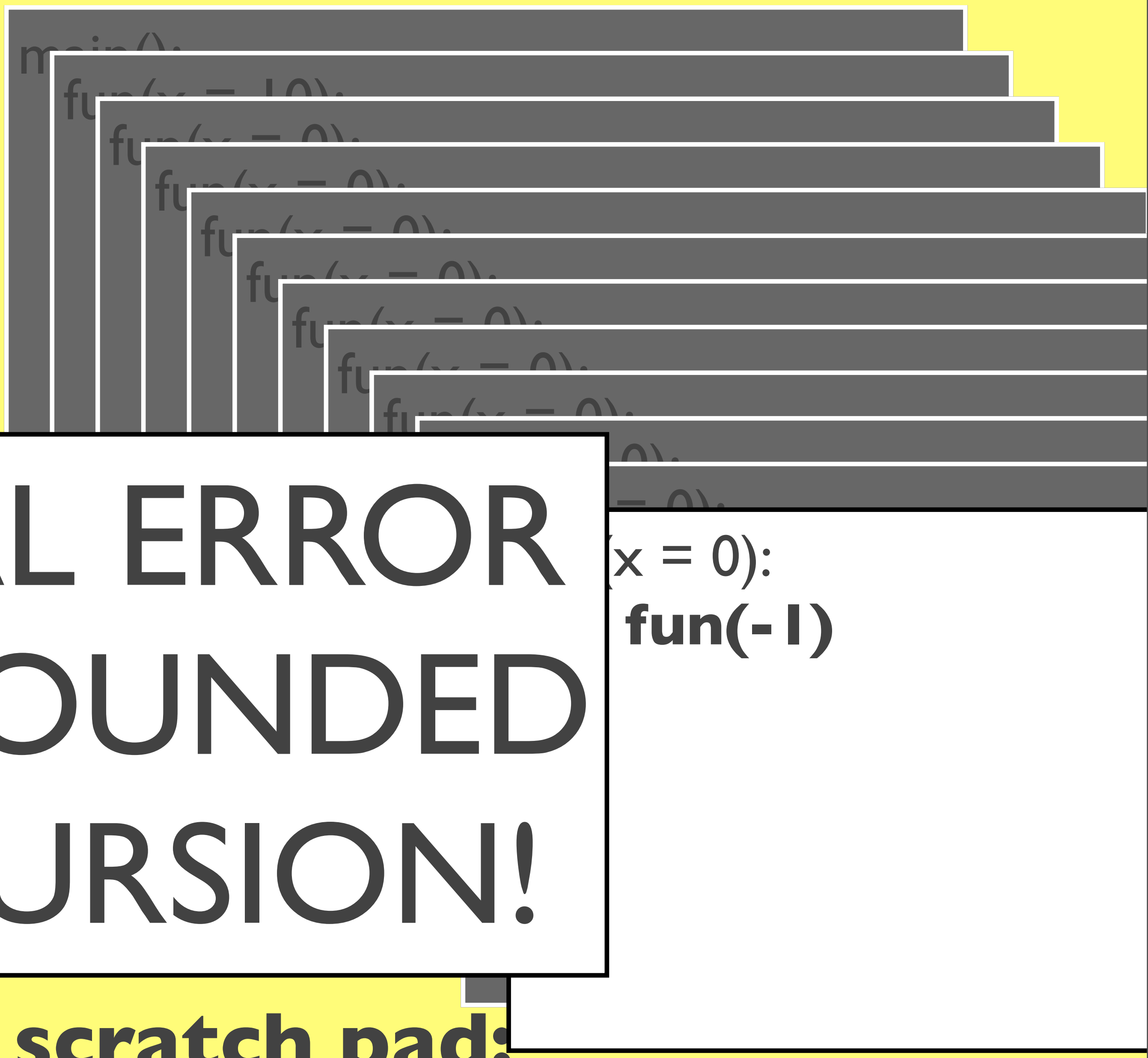
```
	x - 1
=	0 - 1
=	-1
```

but again, it never stops, and we get another instance of unbounded recursion.

function definitions:

```
def fun(x):
    fun(x - 1)

def main():
    fun(10)
```



FATAL ERROR
UNBOUNDED
RECURSION!

You run main()

scratch pad:

```
    x - 1
=   0 - 1
=   -1
```

but again, it never stops, and we get another instance of unbounded recursion.

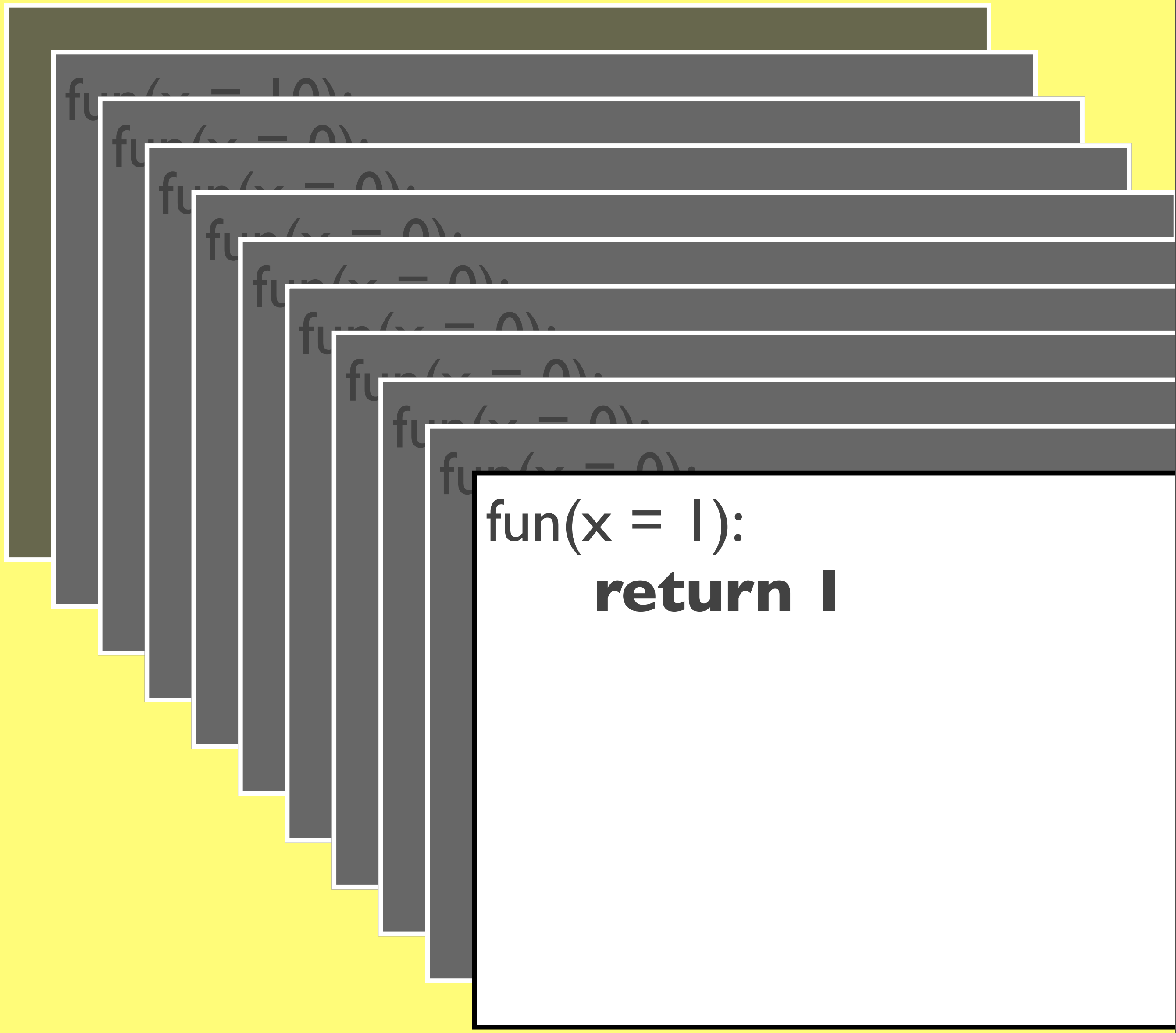
we need a **base case**

Recursion is still pretty useless to us. We need a way to allow a function to call itself, but also a way to eventually stop. This is called the **base case**.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    fun(x - 1)
    return None

def main():
    fun(10)
```



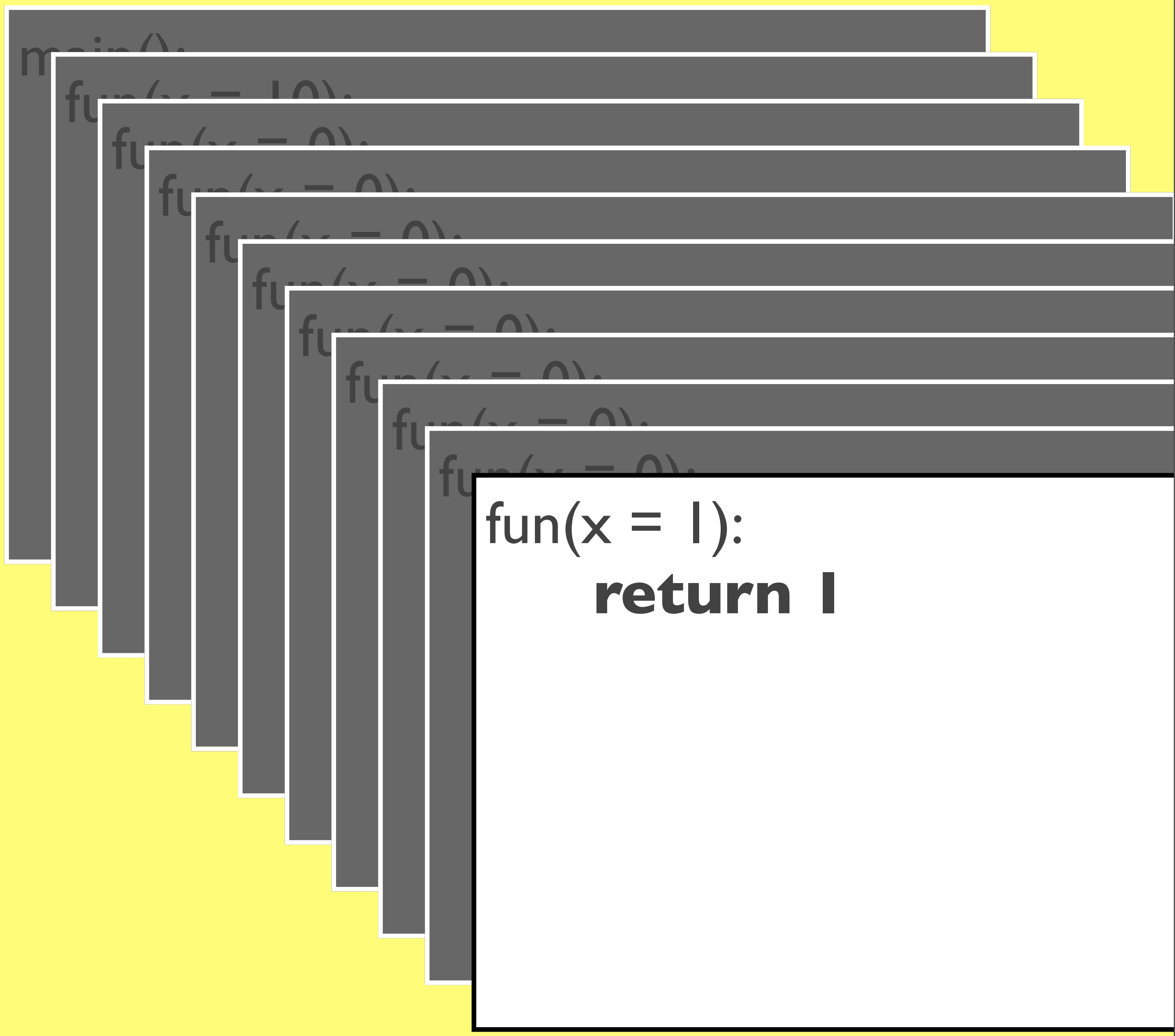
You run main()

Here we have added a base case: when the parameter is 1, the function returns 1. You will notice that this base case is useful: if we wanted to know the sum of the first 1 integers, we would call fun(1), and it would return 1, which is the correct answer. Even though this is trivial, identifying the base case like this is crucial to developing a recursive algorithm.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    fun(x - 1)
    return None

def main():
    fun(10)
```



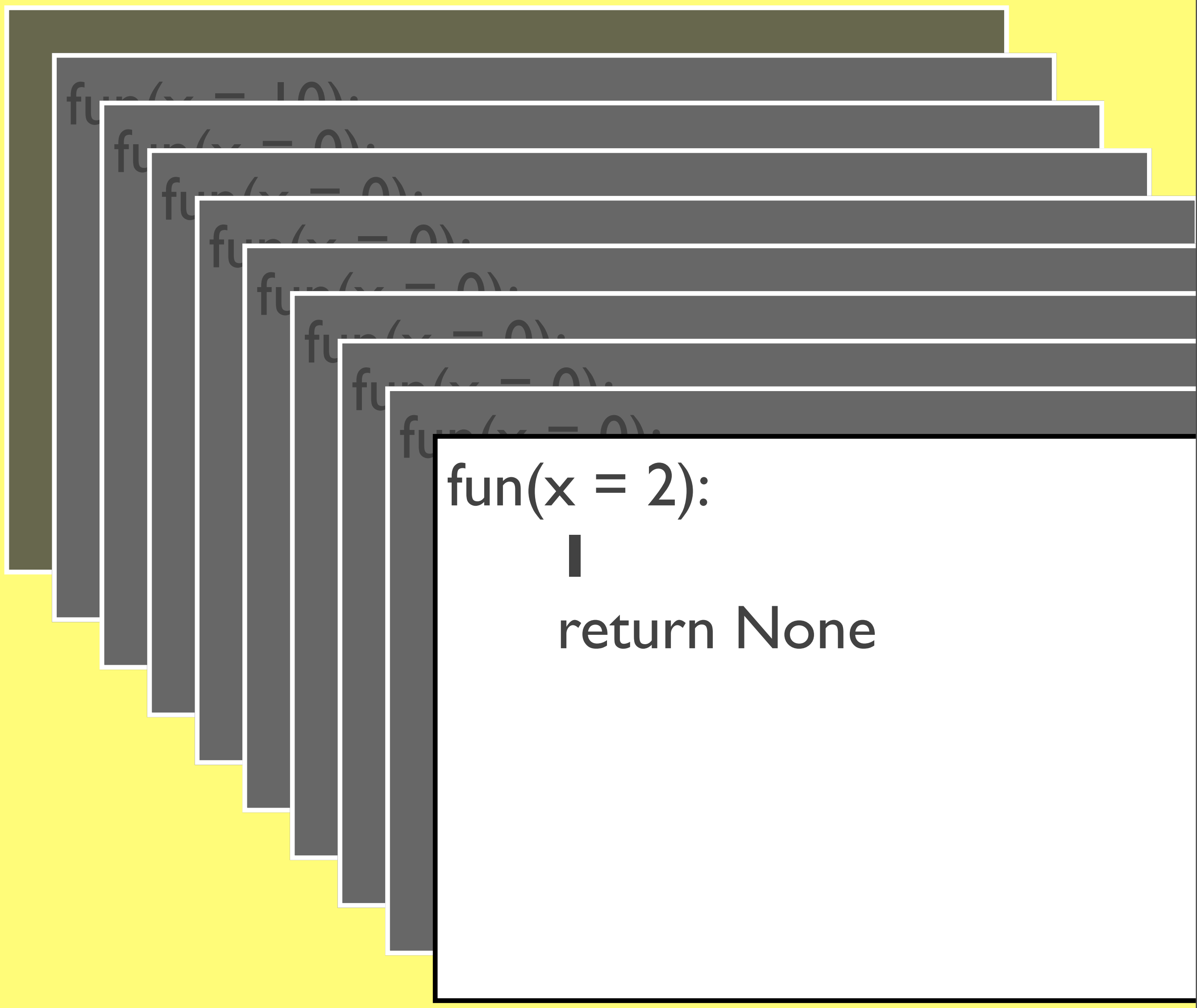
You run main()

Here we have added a base case: when the parameter is 1, the function returns 1. You will notice that this base case is useful: if we wanted to know the sum of the first 1 integers, we would call fun(1), and it would return 1, which is the correct answer. Even though this is trivial, identifying the base case like this is crucial to developing a recursive algorithm.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    fun(x - 1)
    return None

def main():
    fun(10)
```



You run main()

scratch pad:

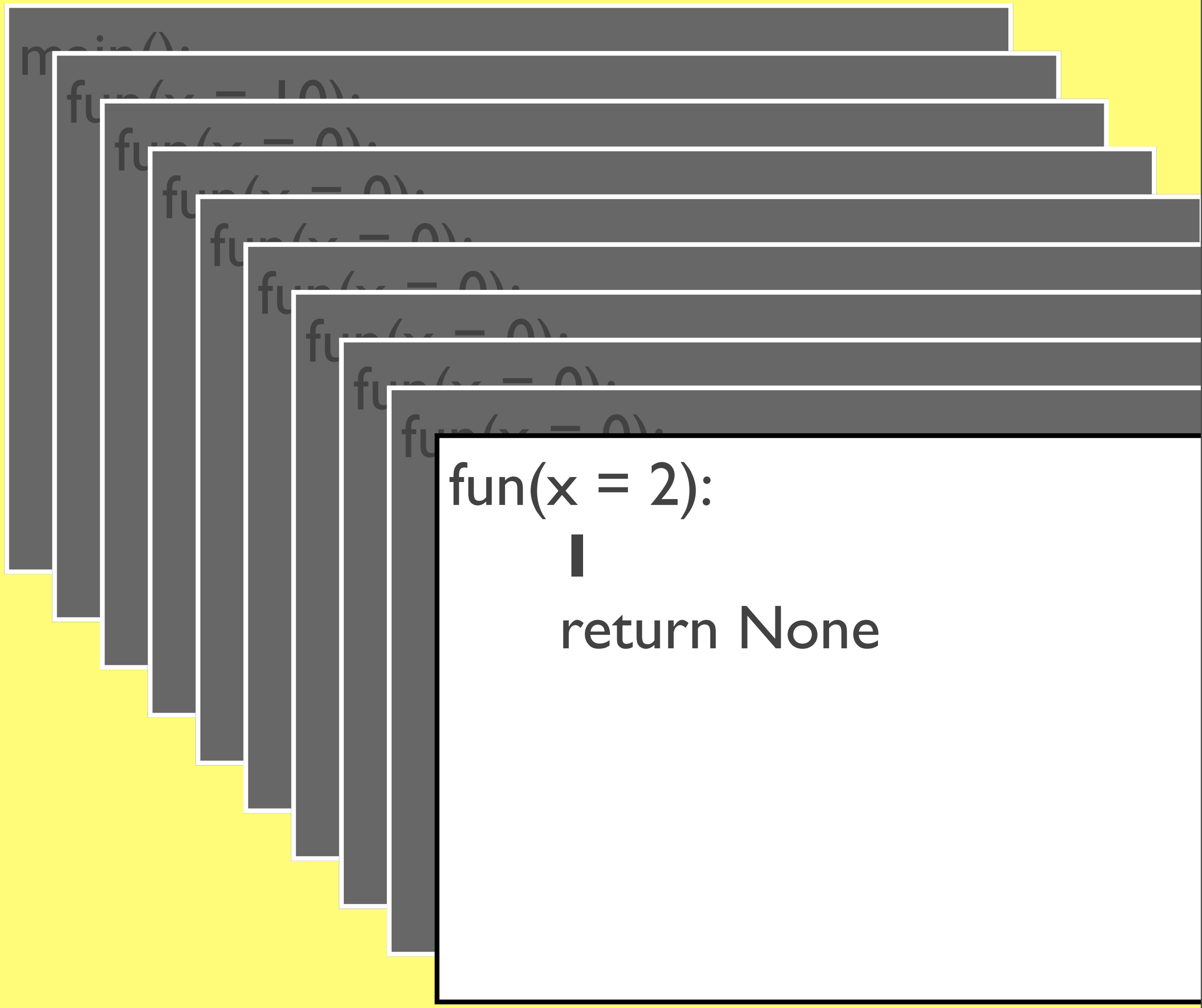
```
    fun(2-1)
=   fun(1)
=   1
```

Now, what happens when we call fun(2)? x is not 1, so we skip to the third line, call fun(2−1) (which is fun(1)). That returns 1, but we are not doing anything with it! Instead, we are returning None.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    fun(x - 1)
    return None

def main():
    fun(10)
```



You run main()

scratch pad:

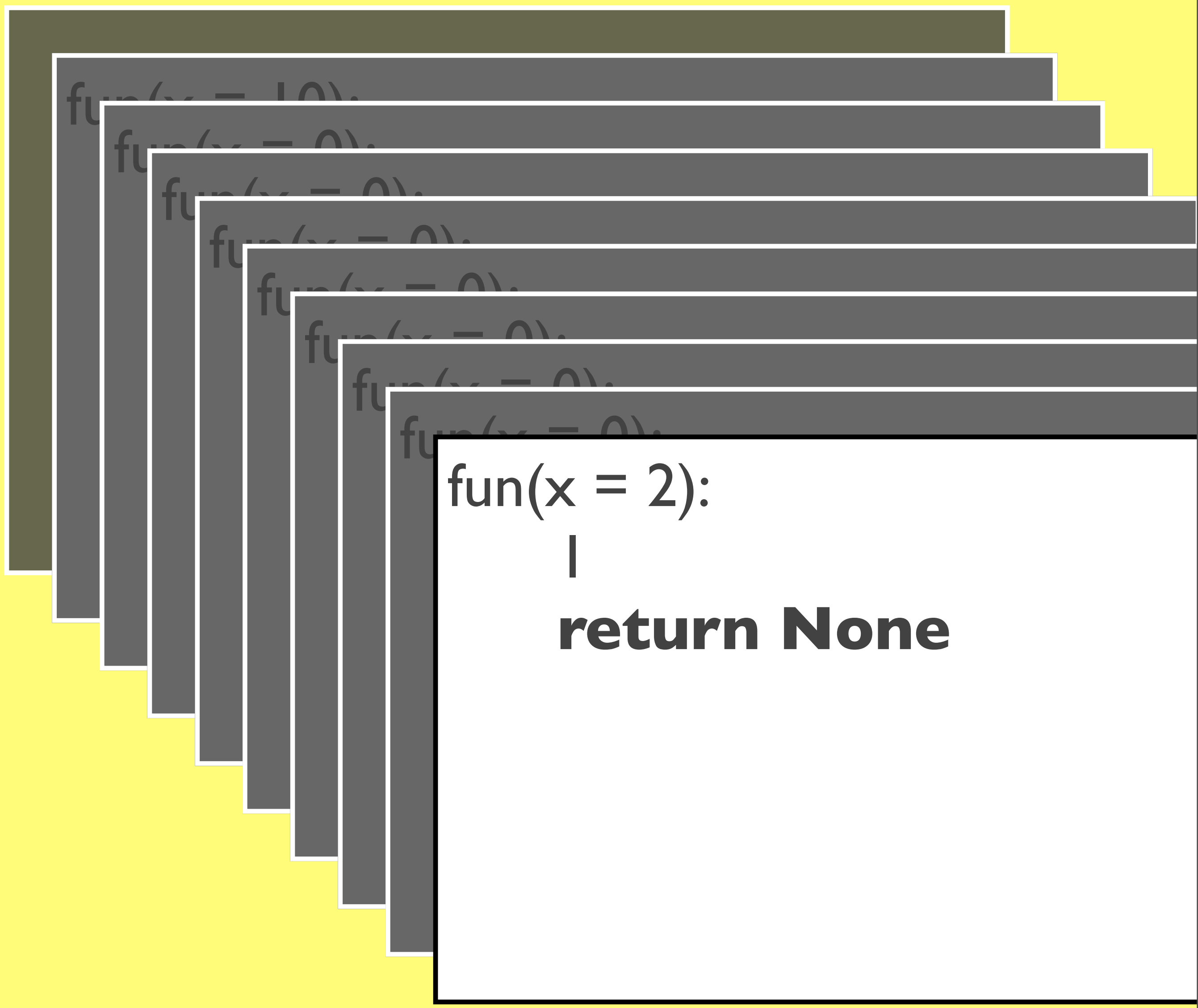
```
    fun(2-1)
=   fun(1)
=   1
```

Now, what happens when we call fun(2)? x is not 1, so we skip to the third line, call fun(2−1) (which is fun(1)). That returns 1, but we are not doing anything with it! Instead, we are returning None.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    fun(x - 1)
    return None

def main():
    fun(10)
```

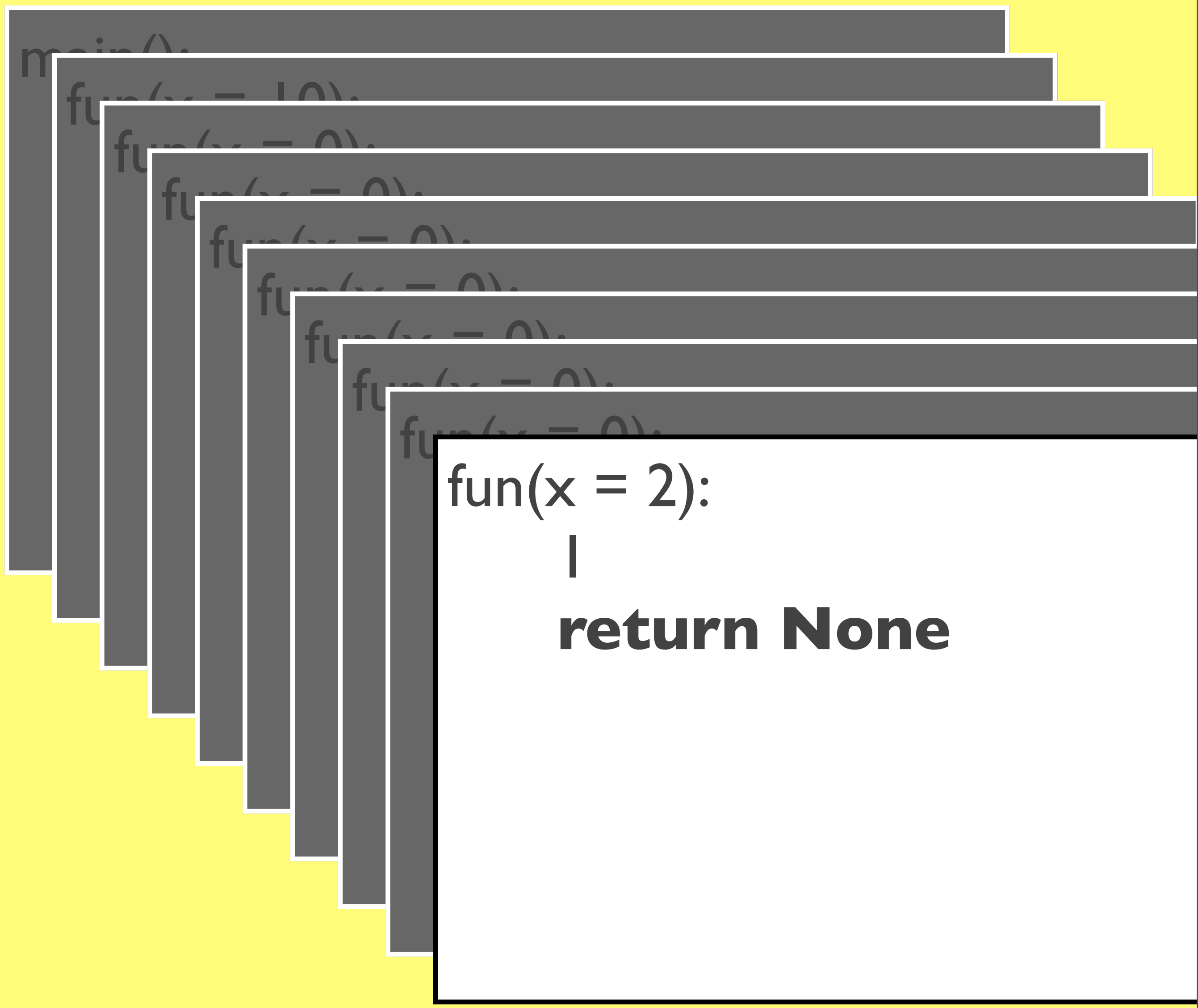


You run main()

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    fun(x - 1)
    return None

def main():
    fun(10)
```



You run main()

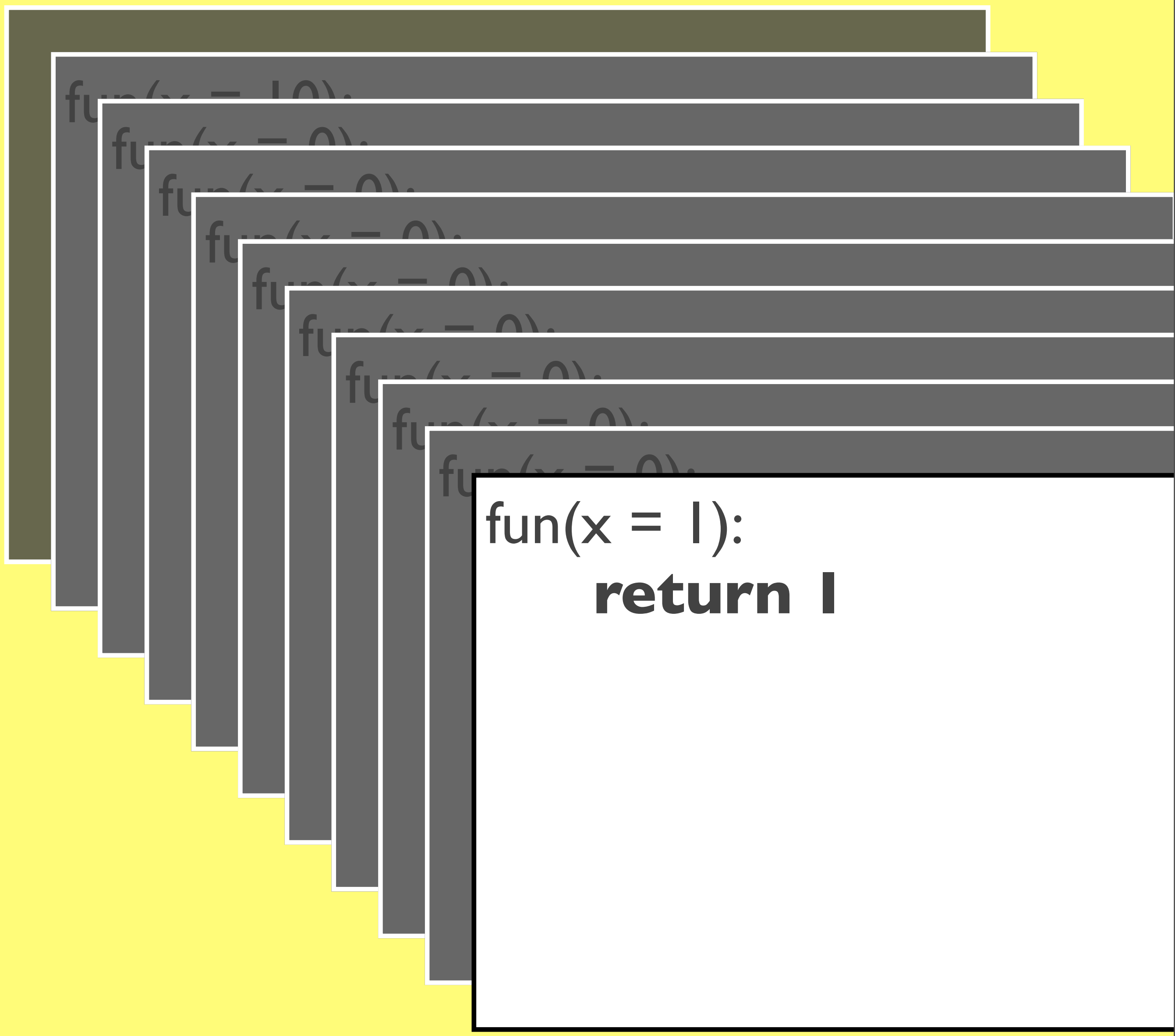
we need to return something in the recursive case, too

We need to return a value, not just in the base case, but in the recursive case as well.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return fun(x - 1)

def main():
    fun(10)
```



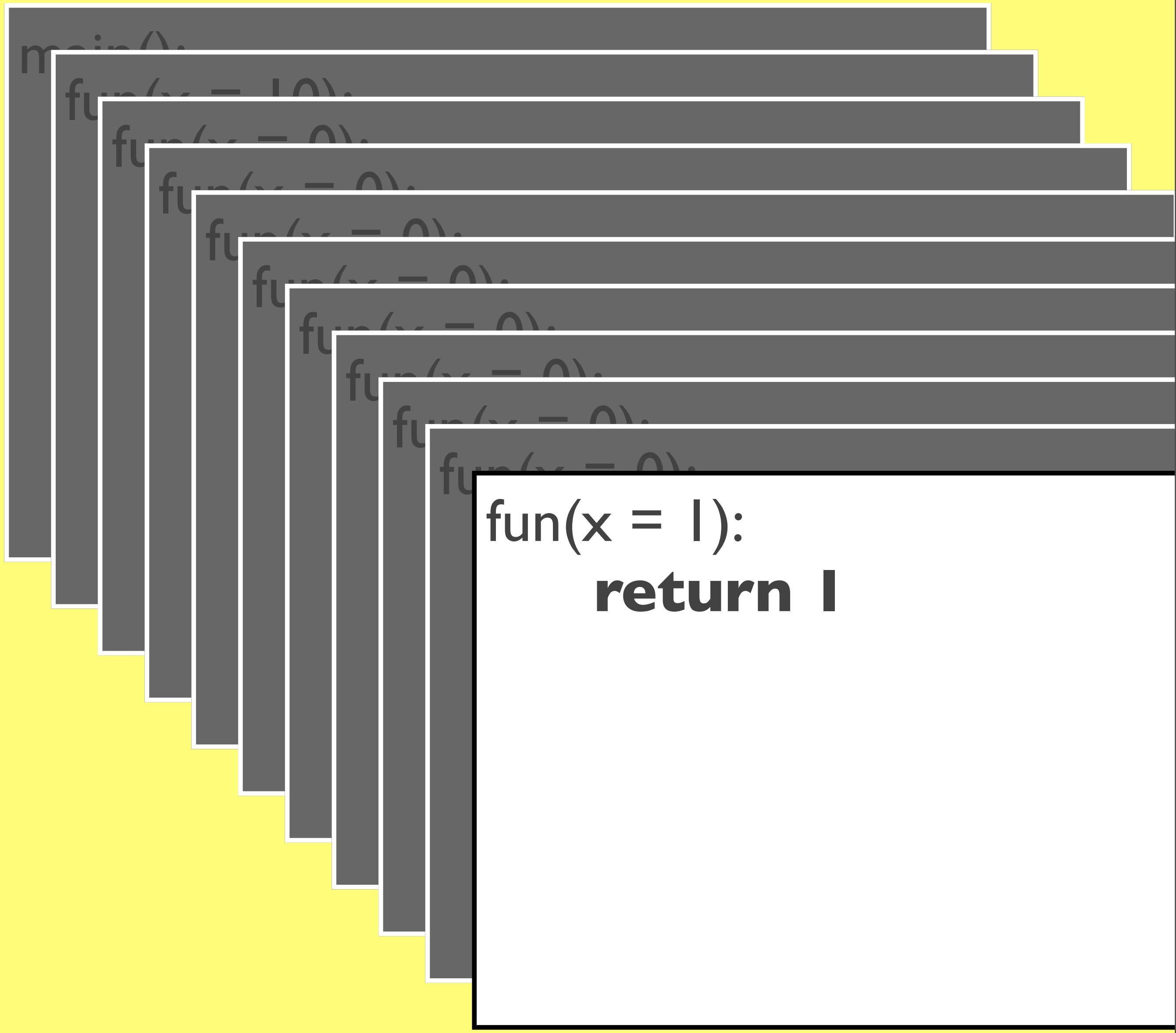
You run main()

Here we have made this change: now our function will always return something. However, it’s not quite the right value.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return fun(x - 1)

def main():
    fun(10)
```



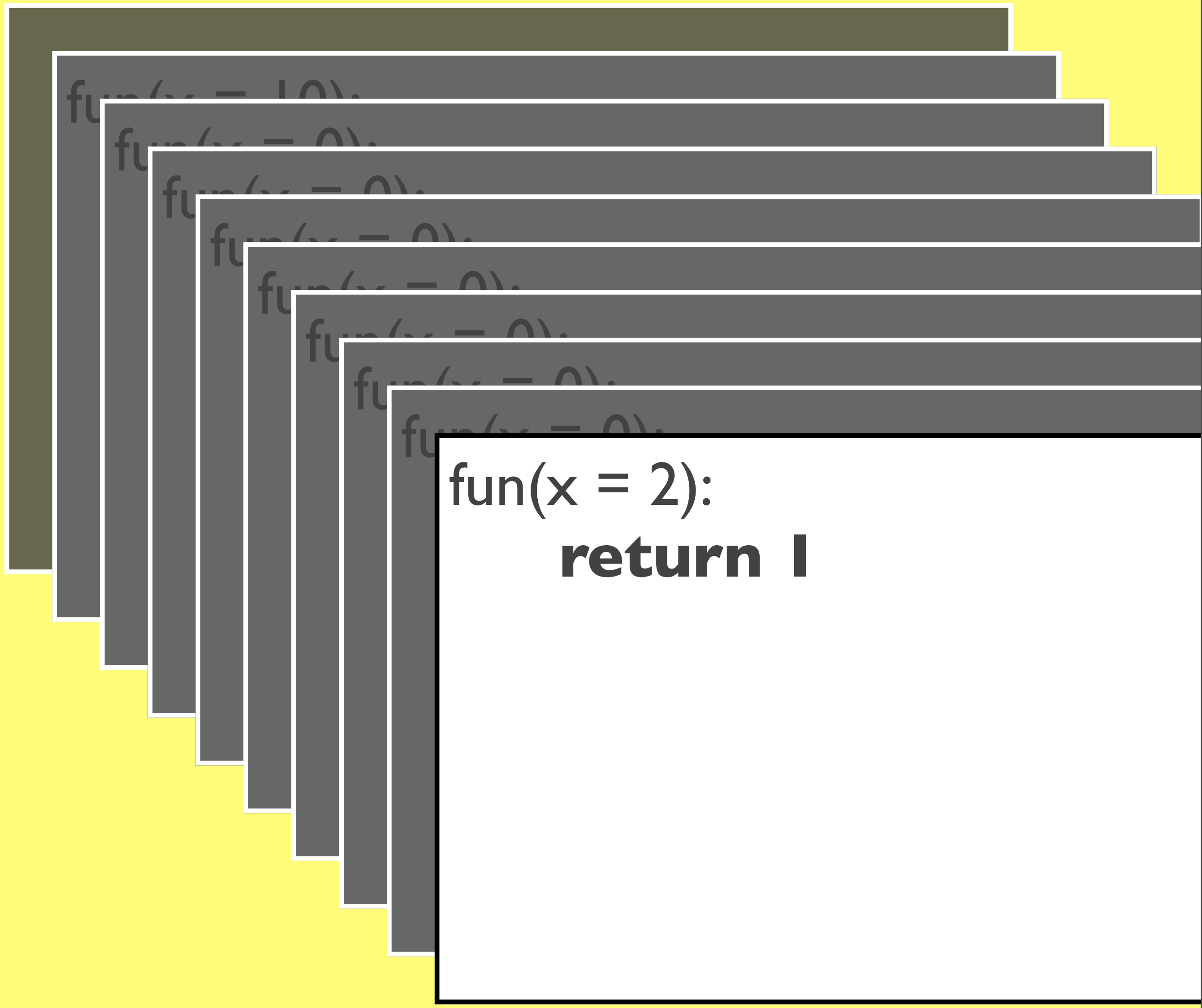
You run main()

Here we have made this change: now our function will always return something. However, it’s not quite the right value.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return fun(x - 1)

def main():
    fun(10)
```



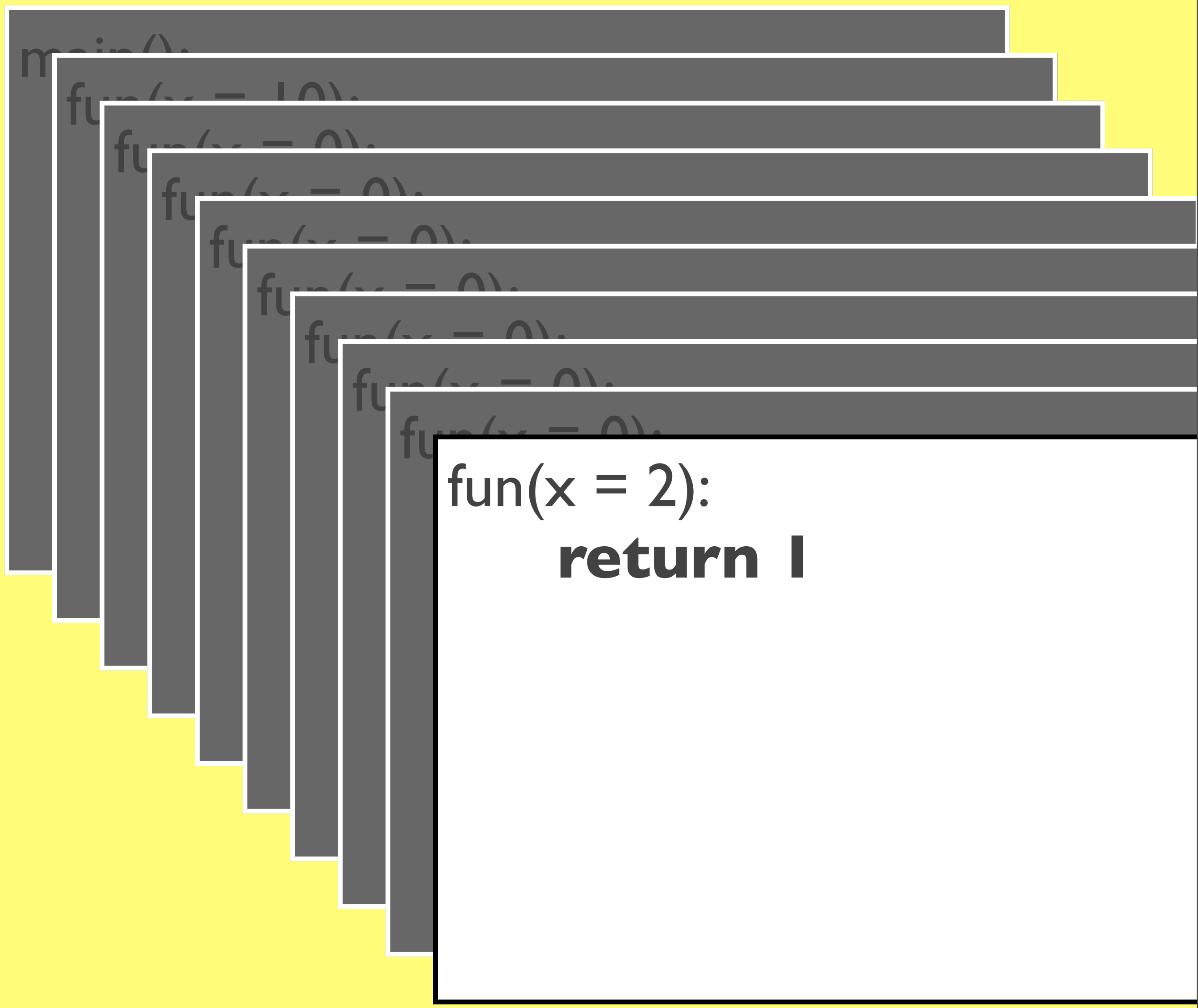
You run main()

When we call fun(1), we still get the right answer, but fun(2) returns 1 as well, which isn’t correct.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return fun(x - 1)

def main():
    fun(10)
```



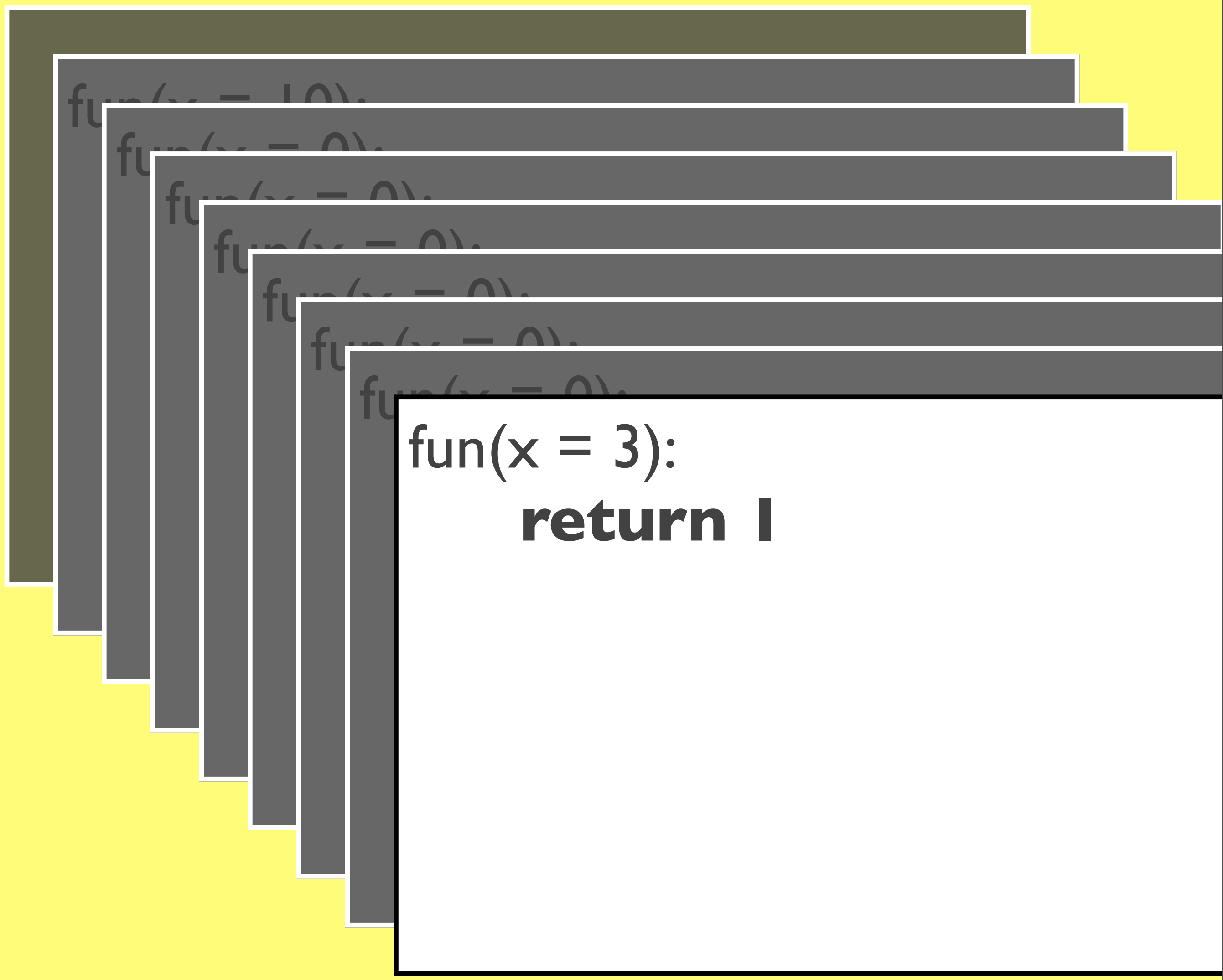
You run main()

When we call fun(1), we still get the right answer, but fun(2) returns 1 as well, which isn’t correct.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return fun(x - 1)

def main():
    fun(10)
```



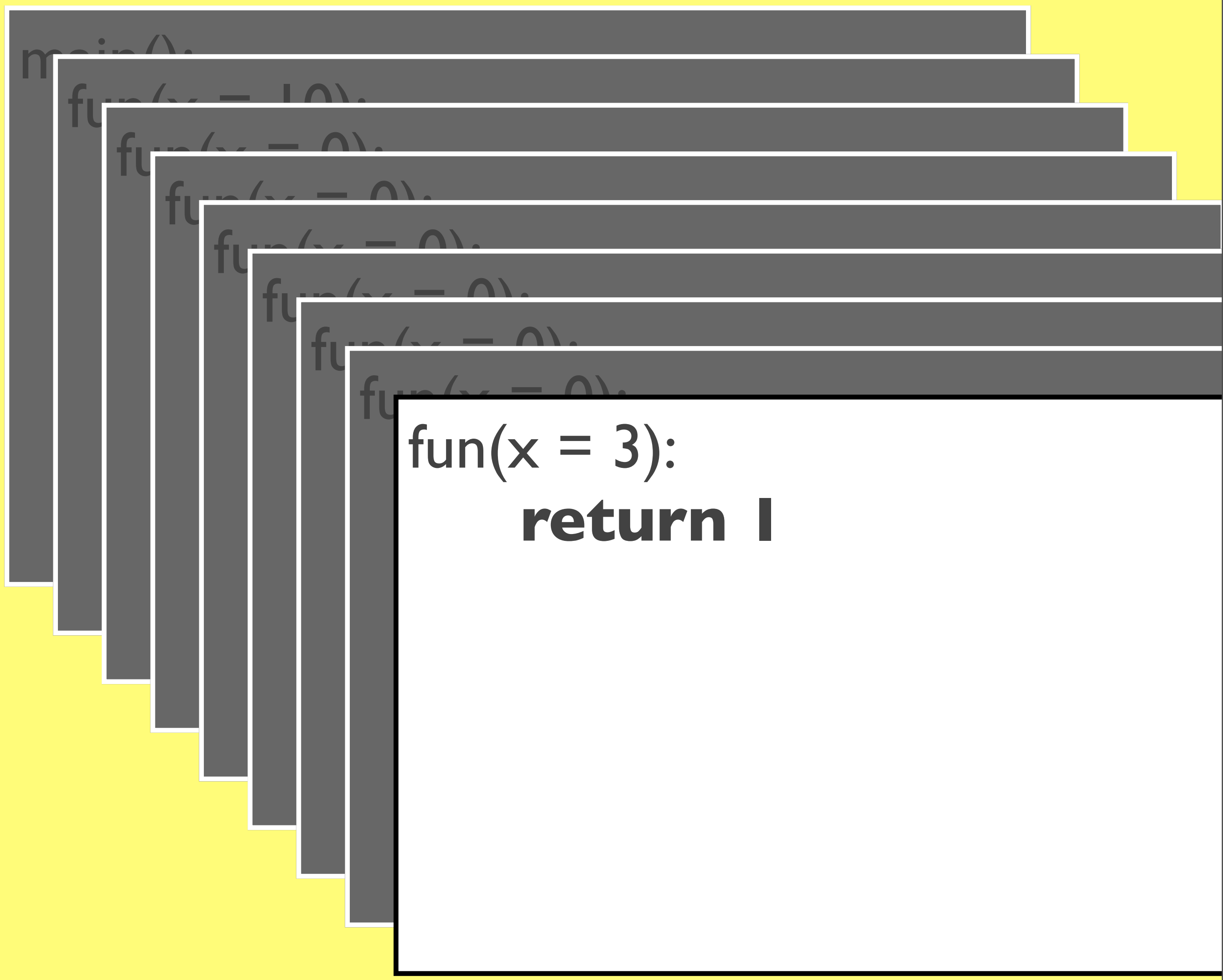
You run main()

Same for fun(3), and all the way back up to the original call fun(10).

function definitions:

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

def main():
    fun(10)
```



You run main()

Same for fun(3), and all the way back up to the original call fun(10).

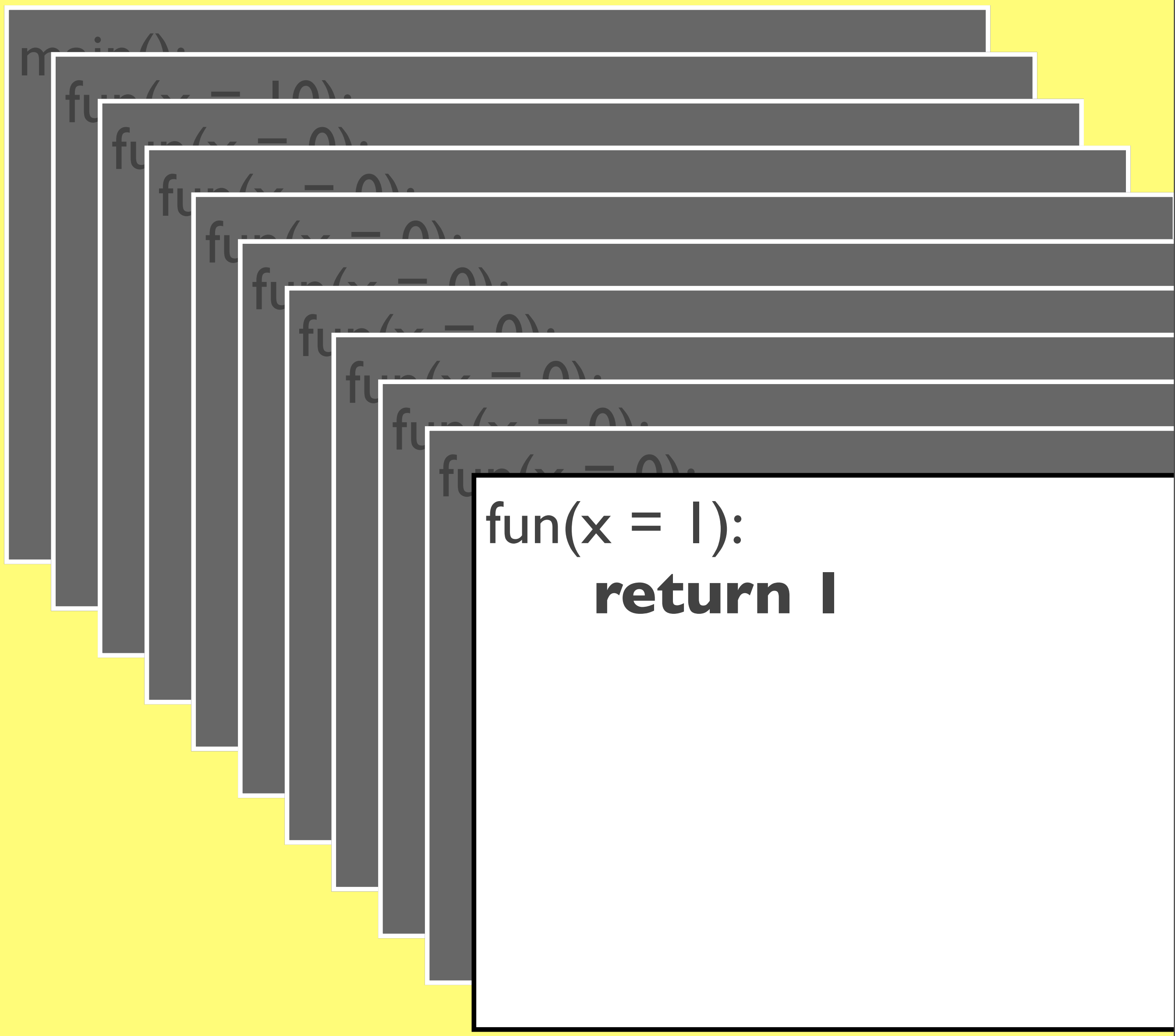
we need to actually *do something* with the results we've computed

The recursive case needs to return a different value. That is, it finds out the value of the subproblem, and then needs to add / multiply / otherwise manipulate that value so that it's caller can get a different answer.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



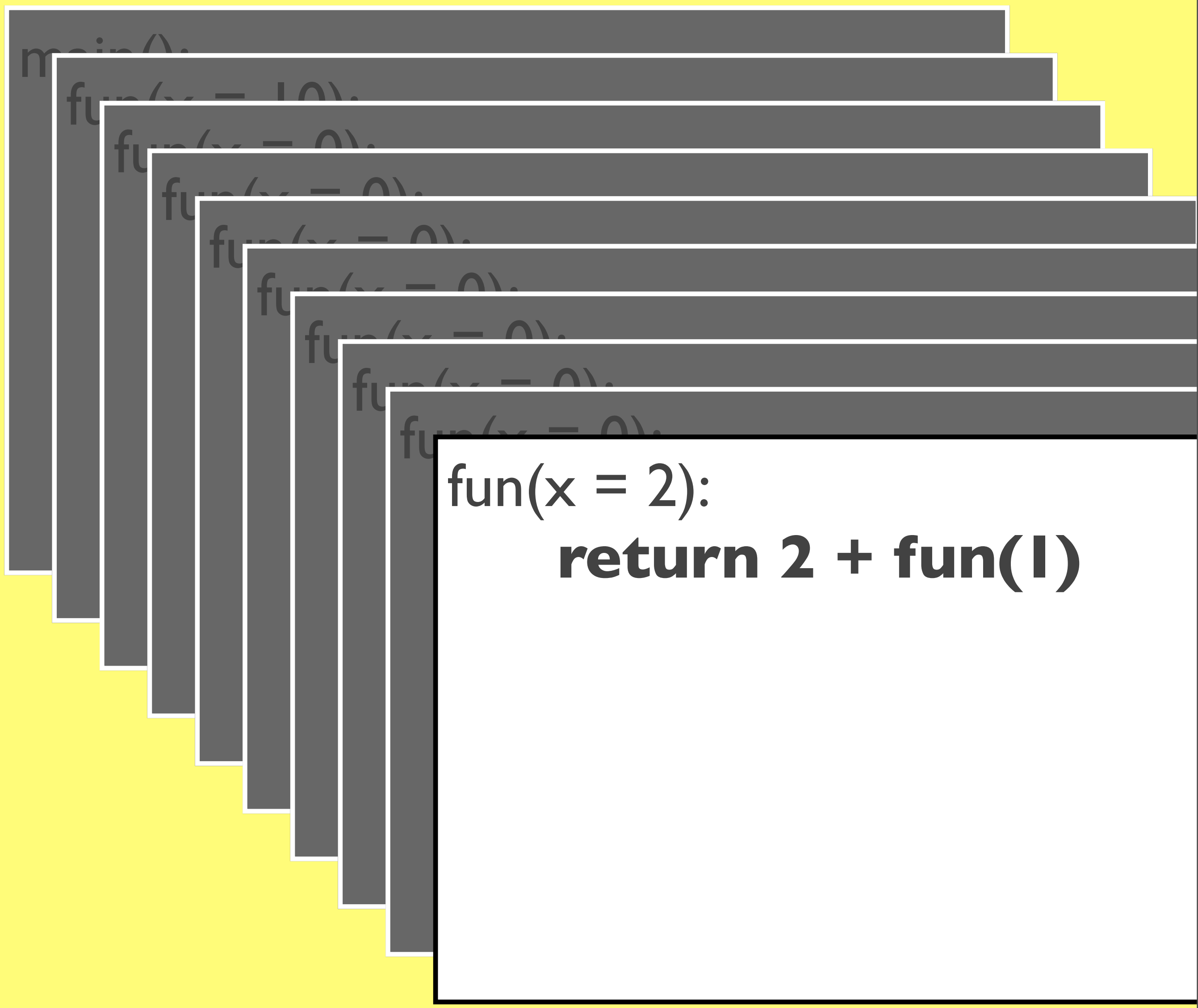
You run main()

The way to do that is to add our own value of x to that of the recursive call. Think of it this way. What is the sum of the first 1 integers? 1. What is the sum of the first two integers? 2 + 1. What is the sum of the first three integers? 3 + 2 + 1. First four? 4 + 3 + 2 + 1. You'll notice that in each case, the answer to "What is the sum of the first n integers?" is "n + the sum of the first (n−1) integers". That's our recursive formulation.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    **return x + fun(x - 1)**

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



You run main()

scratch pad:

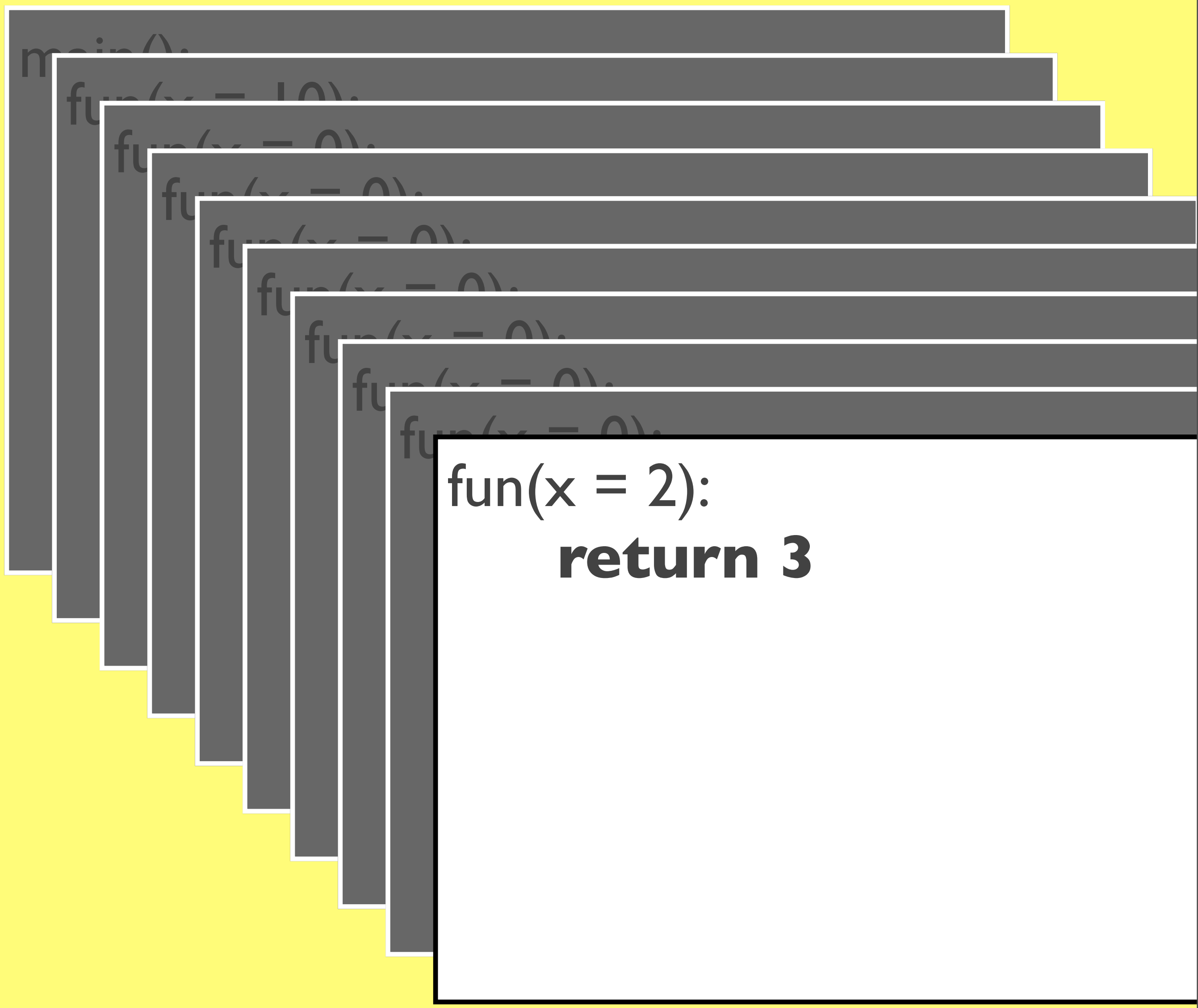
```
    2 + fun(1)
=   2 + 1
=   3
```

Here we see the correct value for fun(2) getting passed up to its caller, fun(3). The scratch pad computes the third line of the function fun(), which it can do now that it's subcall to fun() has returned.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



```
fun(x = 2):
    return 3
```

You run main()

scratch pad:

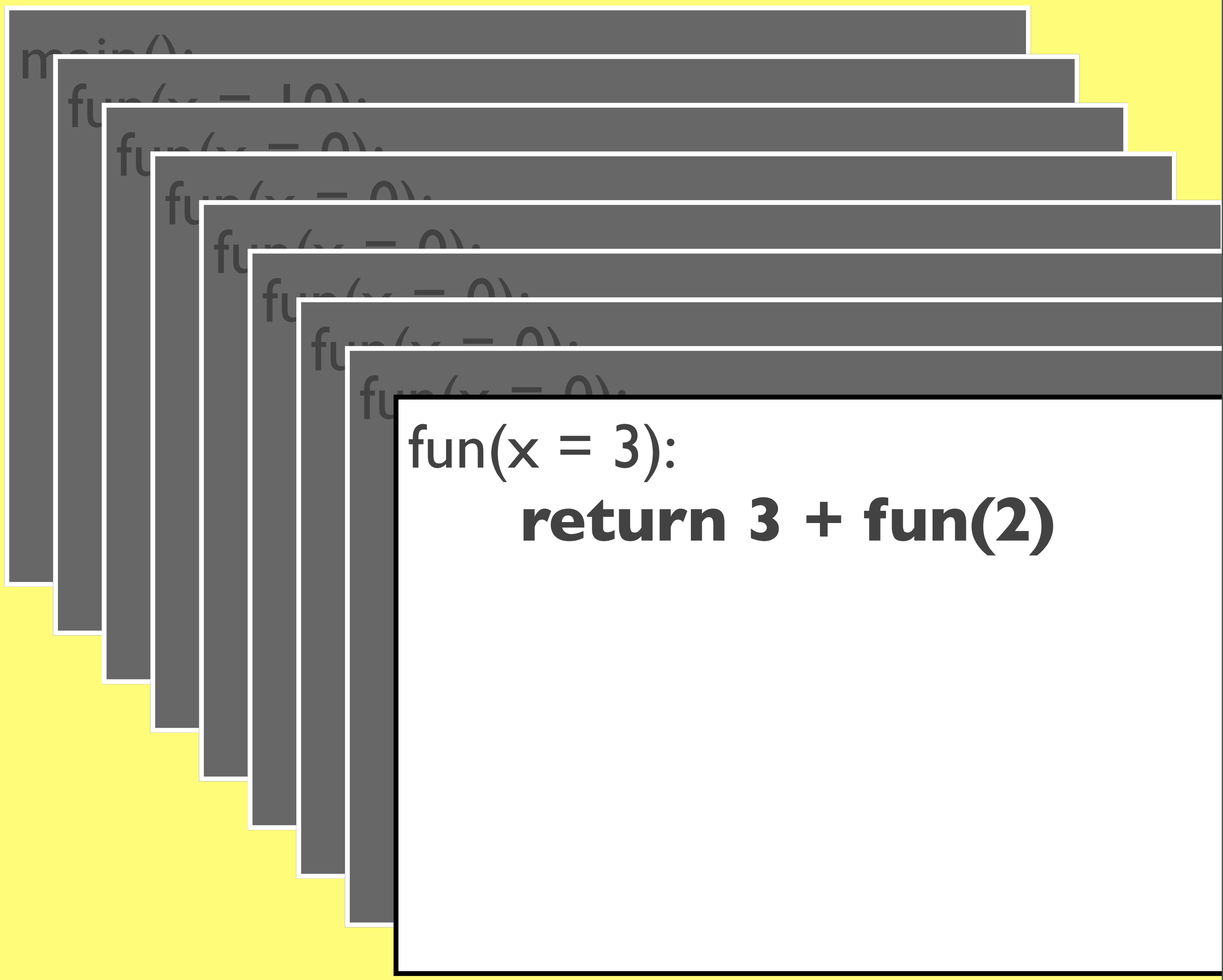
```
     2 + fun(1)
=    2 + 1
=    3
```

And so on up the chain of calls.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



You run main()

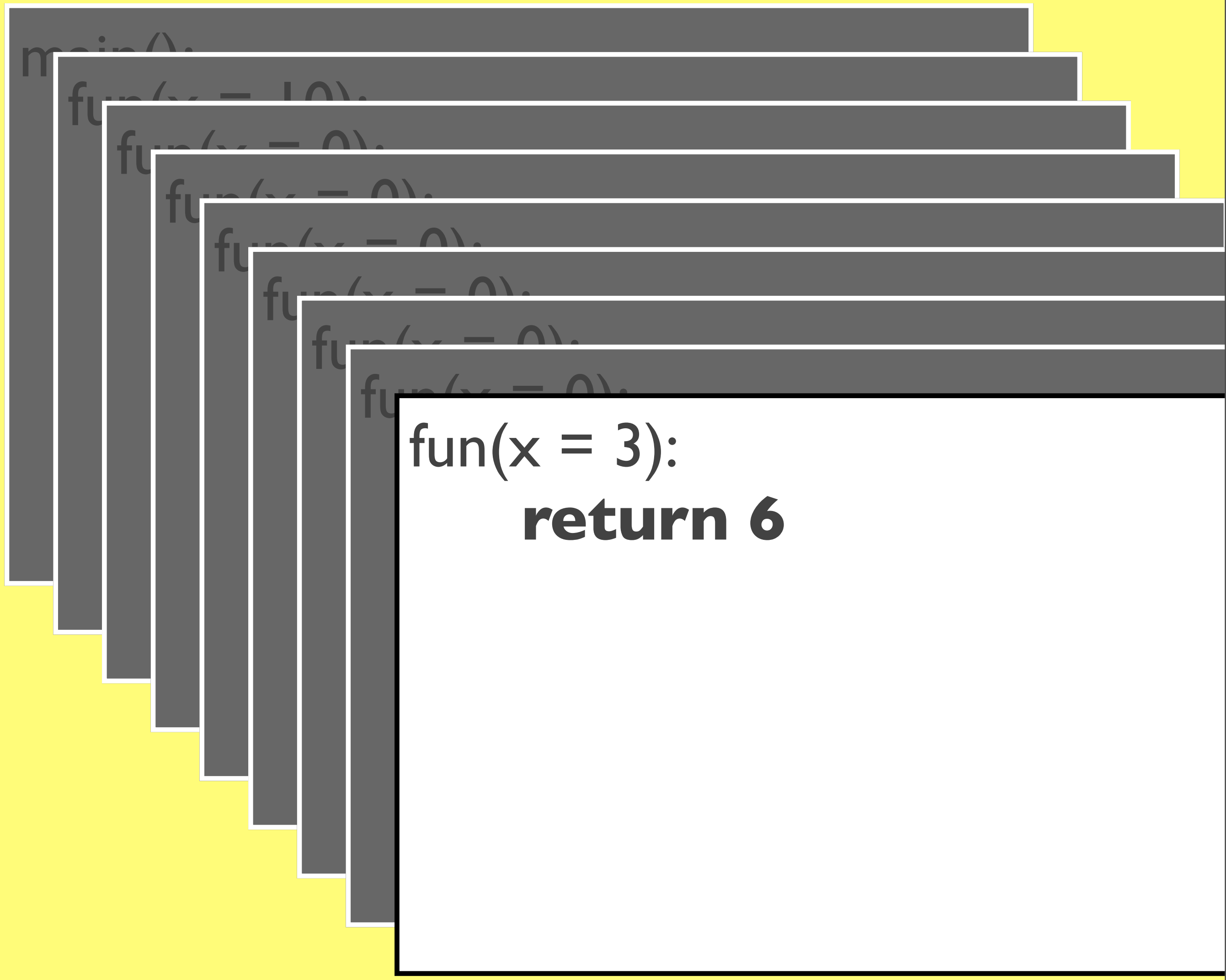
scratch pad:

```
     3 + fun(2)
=    3 + 3
=    6
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



You run main()

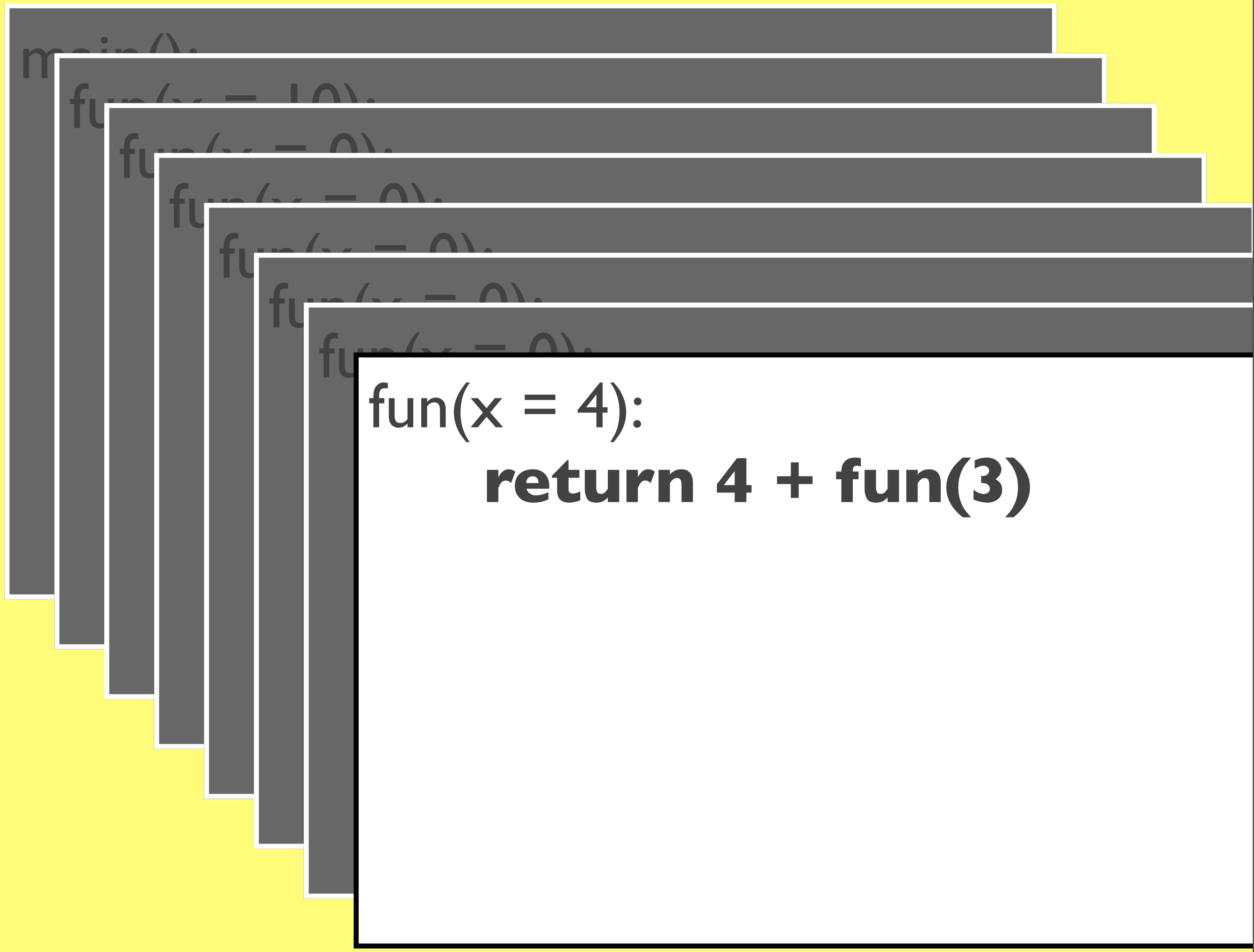
scratch pad:

```
     3 + fun(2)
=    3 + 3
=    6
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



```
fun(x = 4):
    return 4 + fun(3)
```

You run main()

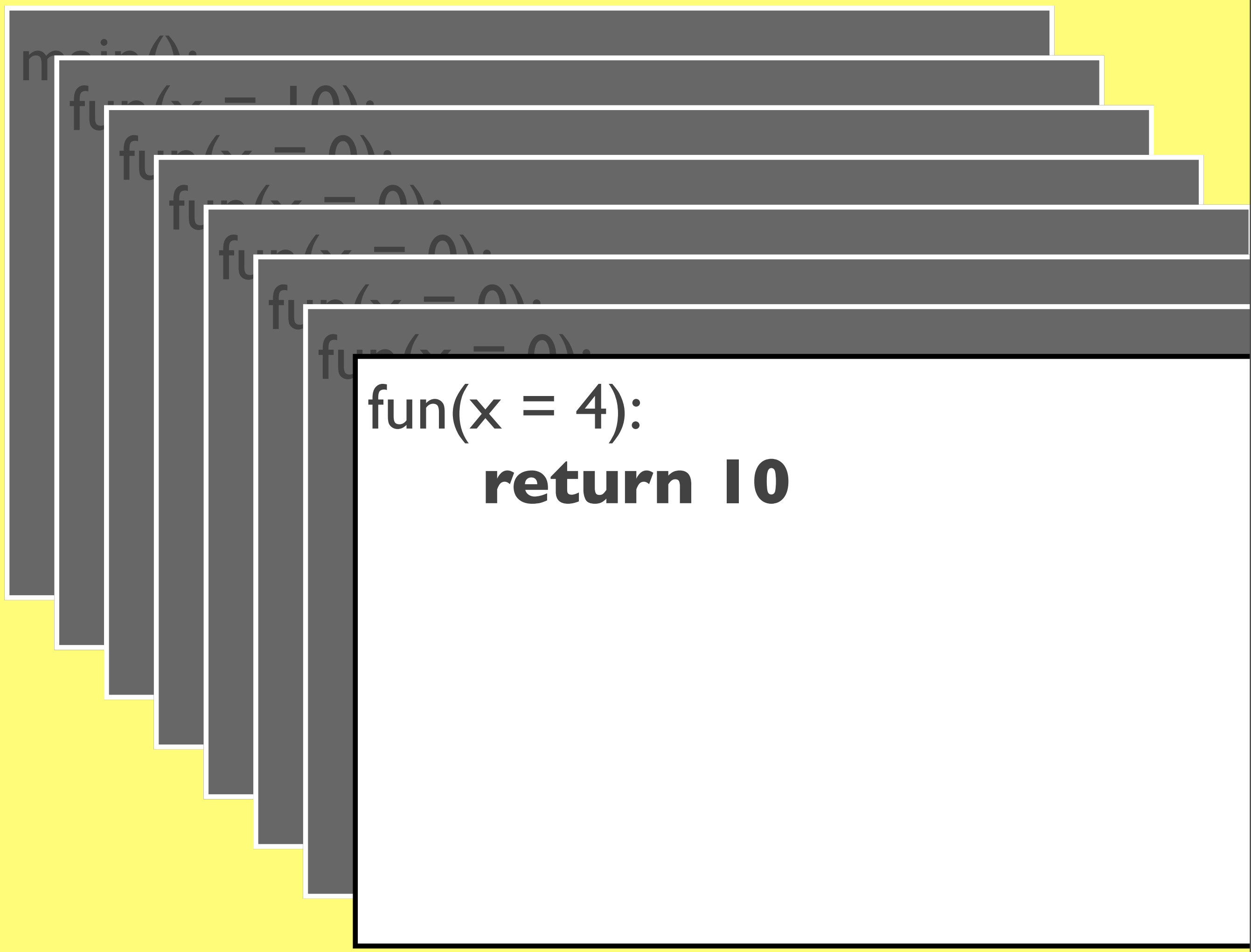
scratch pad:

```
    4 + fun(3-1)
=   4 + 6
=   10
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    **return x + fun(x - 1)**

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



You run main()

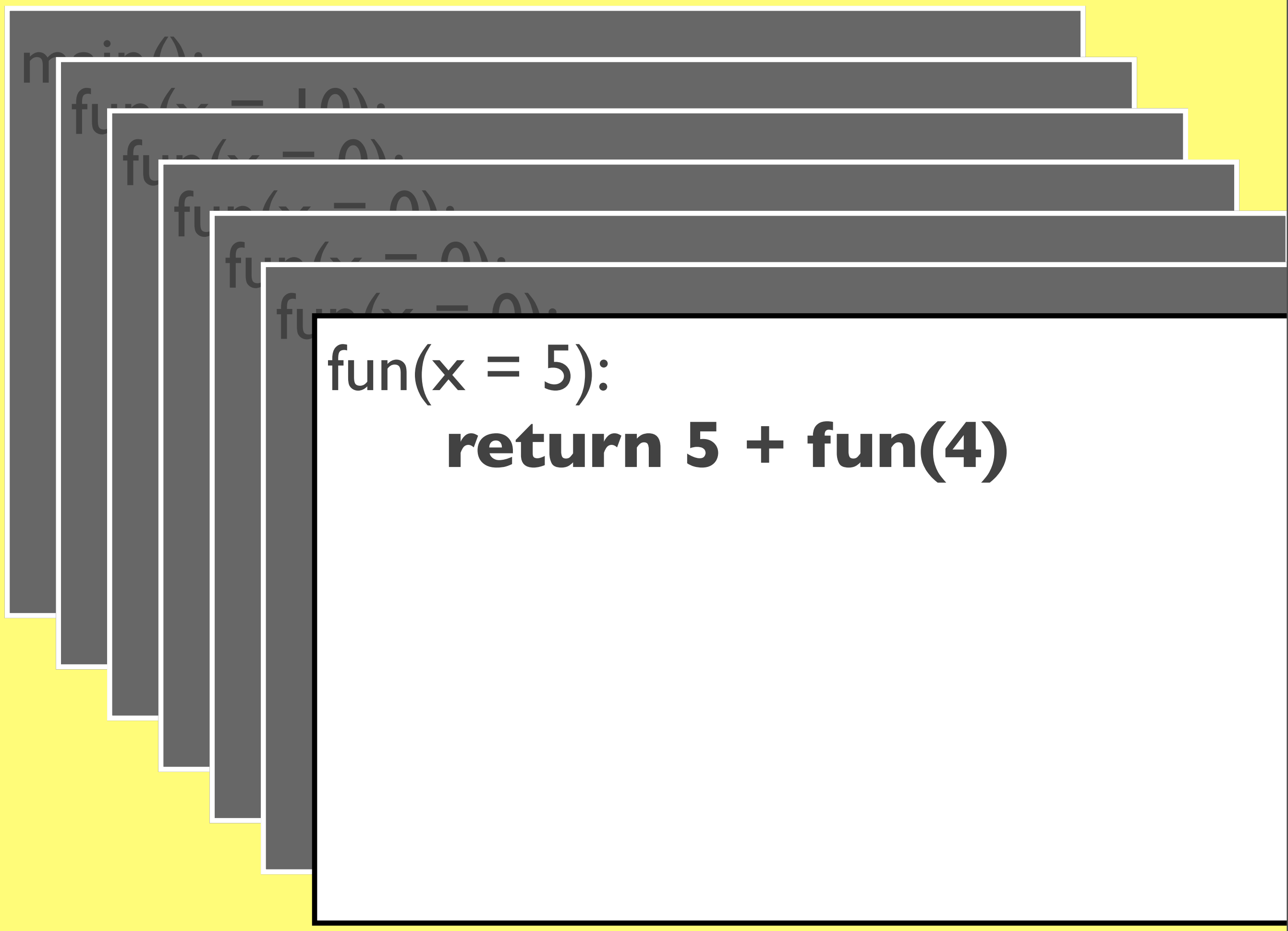
scratch pad:

```
    4 + fun(3-1)
=   4 + 6
=   10
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    **return x + fun(x - 1)**

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



You run main()

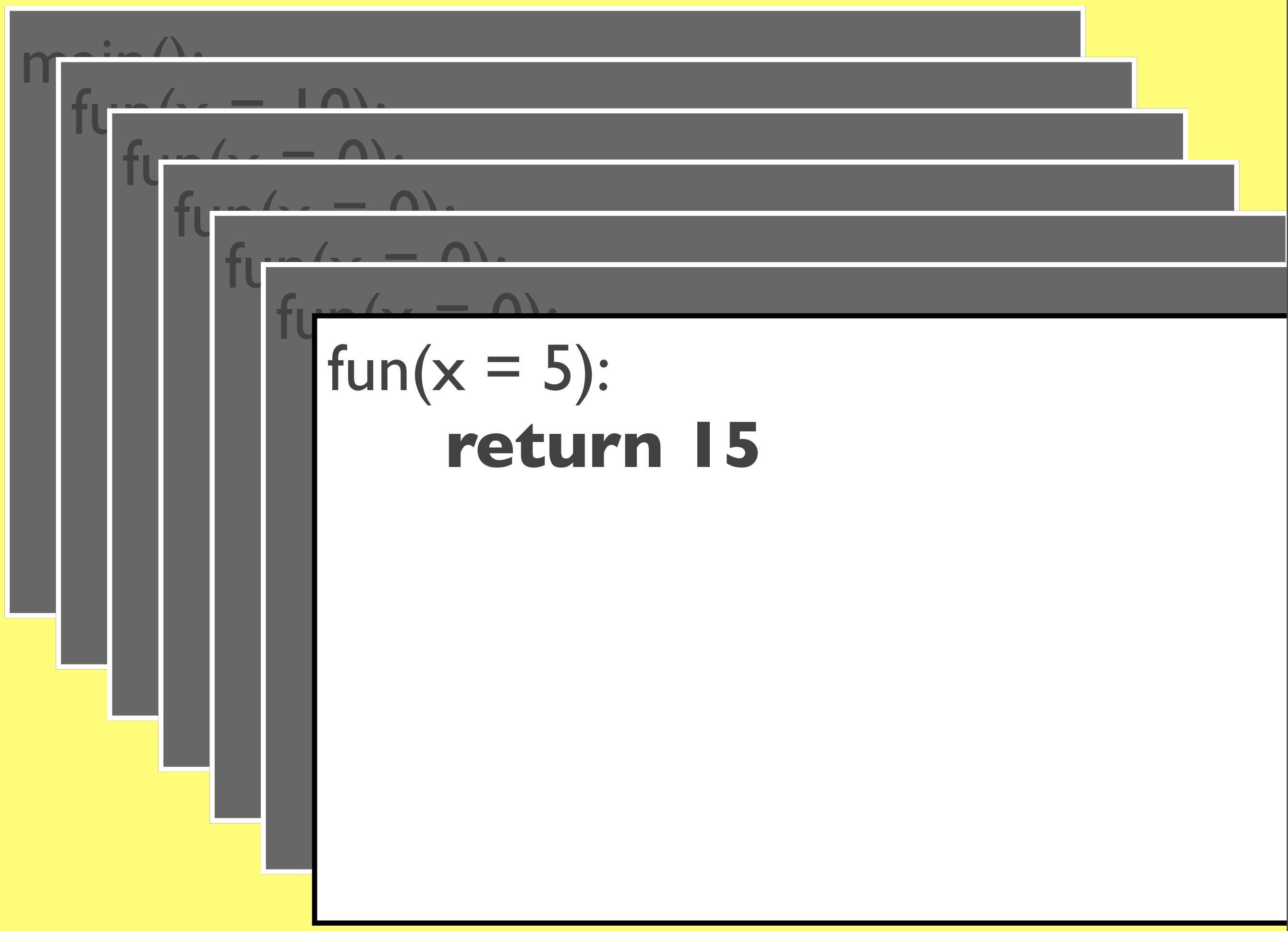
scratch pad:

```
     5 + fun(4)
=    5 + 10
=    15
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    **return x + fun(x - 1)**

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



You run main()

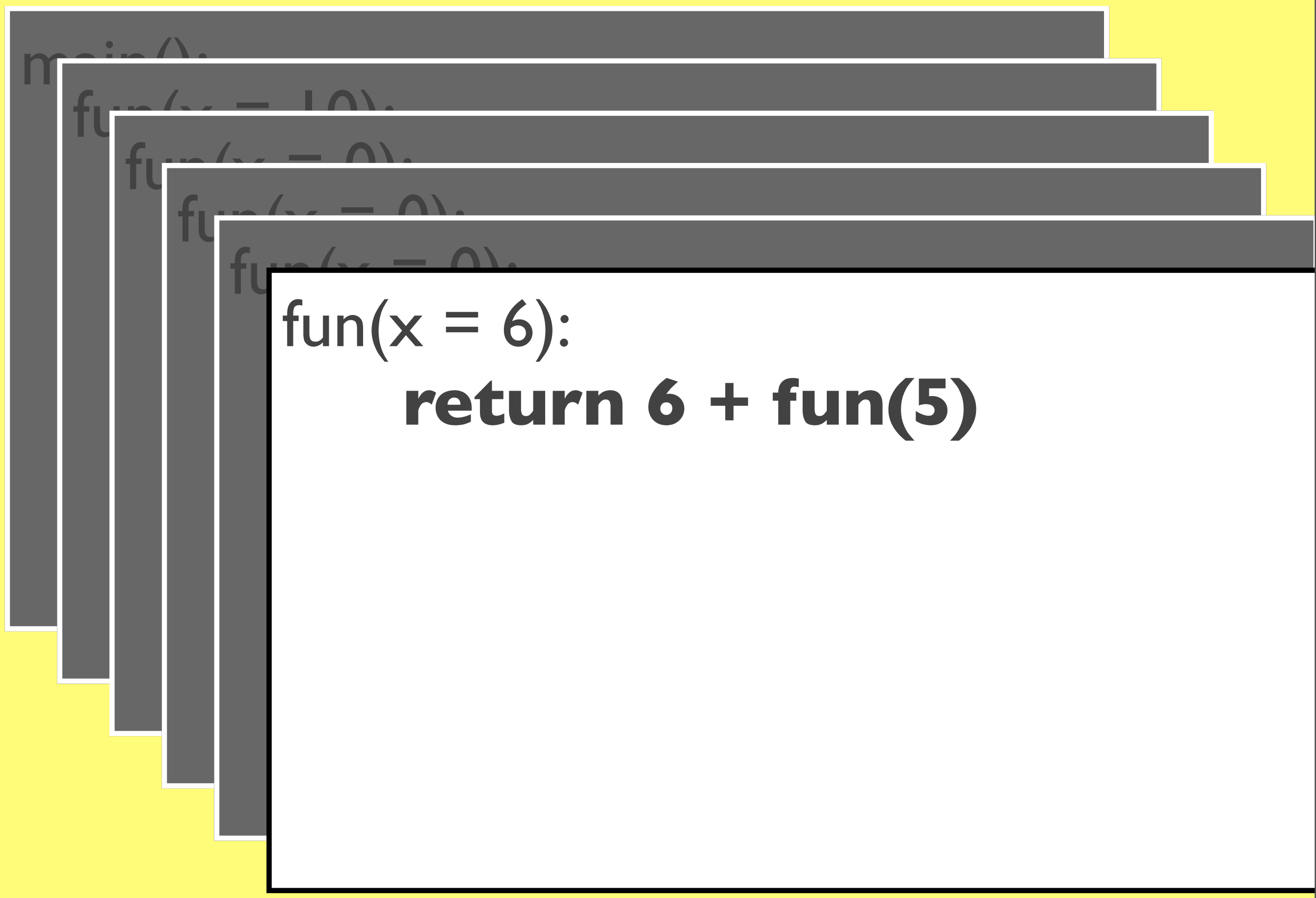
scratch pad:

```
    5 + fun(4)
=   5 + 10
=   15
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



```
fun(x = 6):
    return 6 + fun(5)
```

You run main()

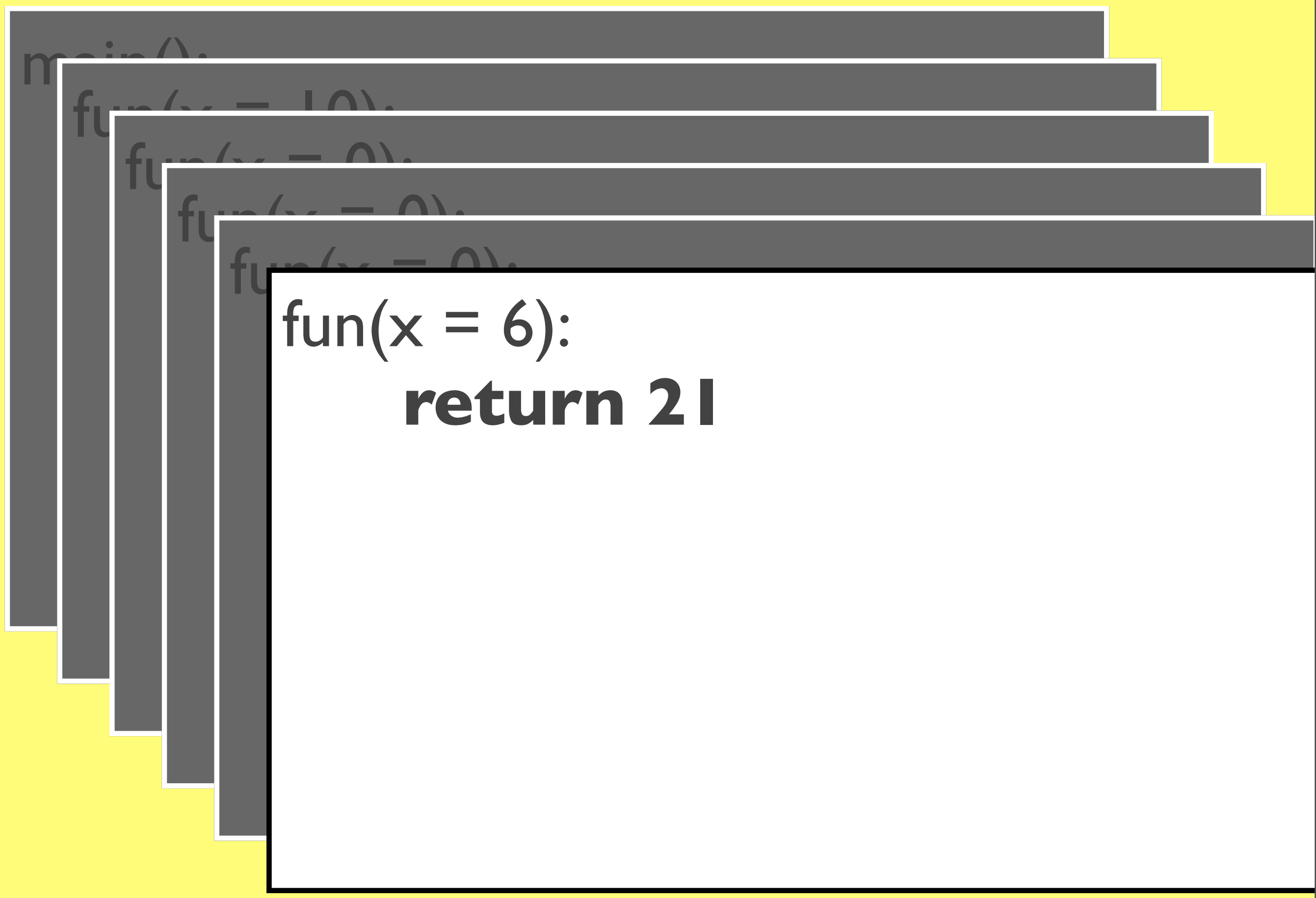
scratch pad:

```
     6 + fun(5)
=    6 + 15
=    21
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



You run main()

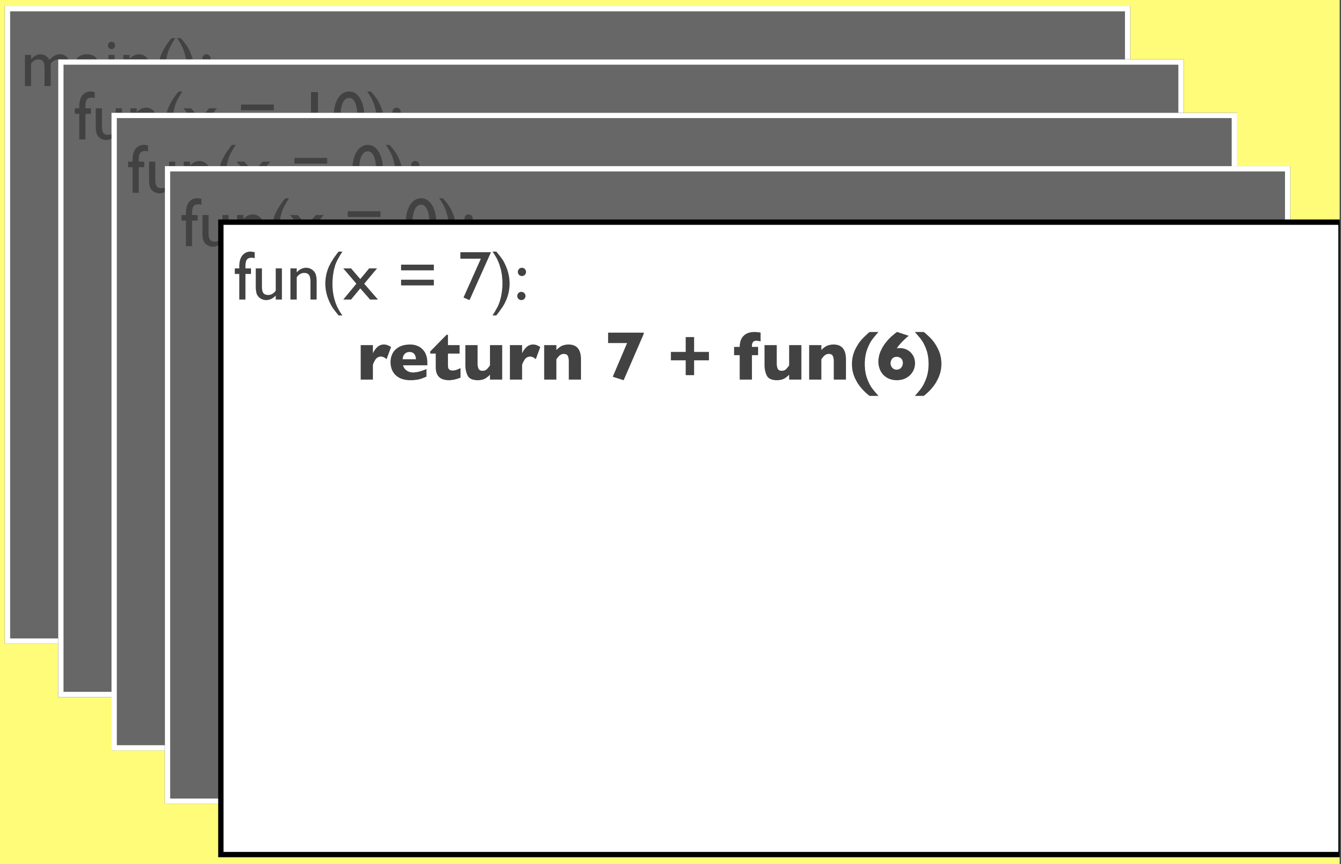
scratch pad:

```
    6 + fun(5)
=   6 + 15
=   21
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    **return x + fun(x - 1)**

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



```
fun(x = 7):
    return 7 + fun(6)
```

You run main()

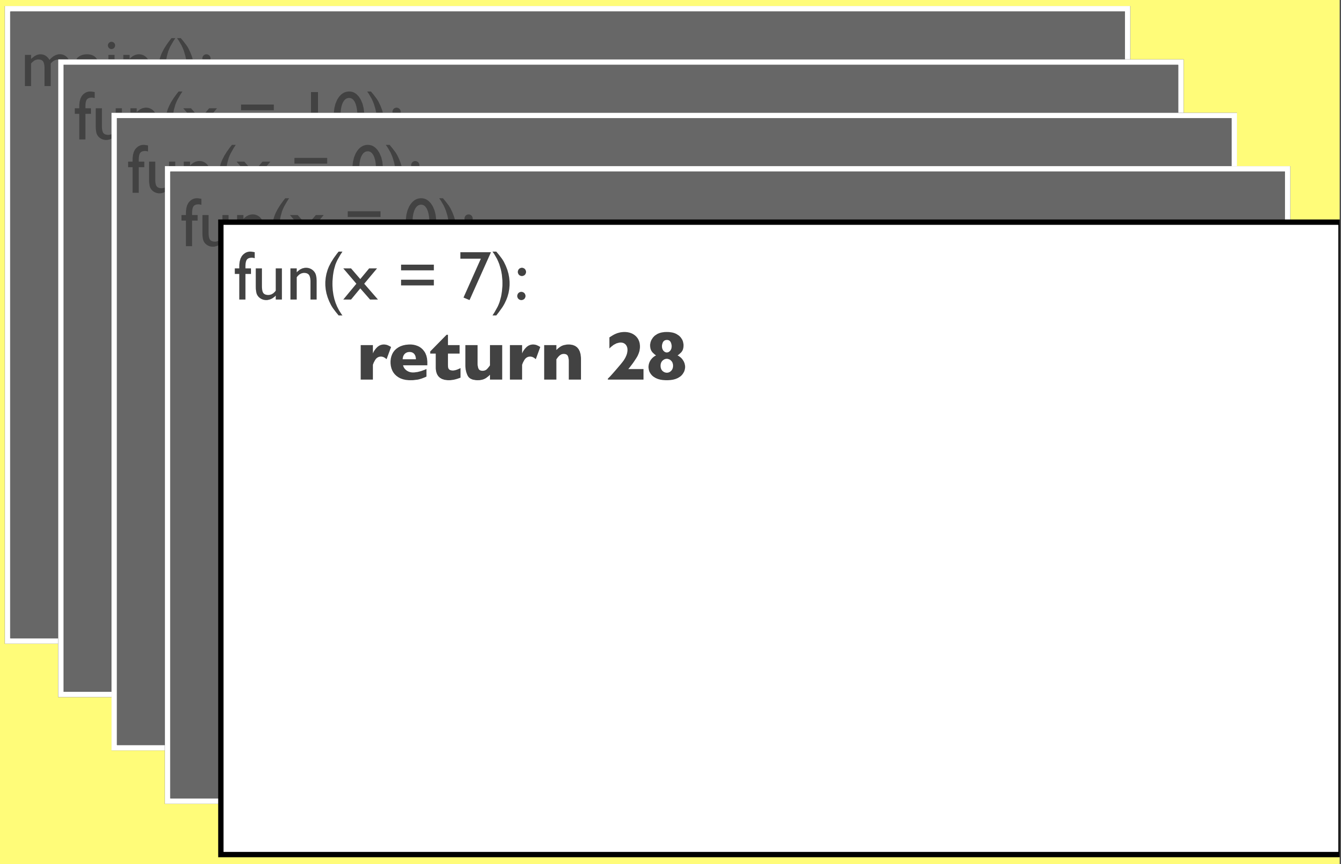
scratch pad:

```
     7 + fun(6-1)
=    7 + 21
=    28
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print “sum of first”, n,
    print “integers is”, sum
```



```
fun(x = 7):
    return 28
```

You run main()

scratch pad:

```
    7 + fun(6-1)
=   7 + 21
=   28
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    **return x + fun(x - 1)**

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```

main():

fun(x = 10):

fun(x = 9):

fun(x = 8):

    return 8 + fun(7)

You run main()

scratch pad:

```
    8 + fun(7)
=   8 + 28
=   36
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```

```
main():
fun(x = 10):
fun(x = 9):
fun(x = 8):
    return 36
```

You run main()

scratch pad:

```
    8 + fun(7)
=   8 + 28
=   36
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```

```
fun(x = 9):
    return 9 + fun(8)
```

You run main()

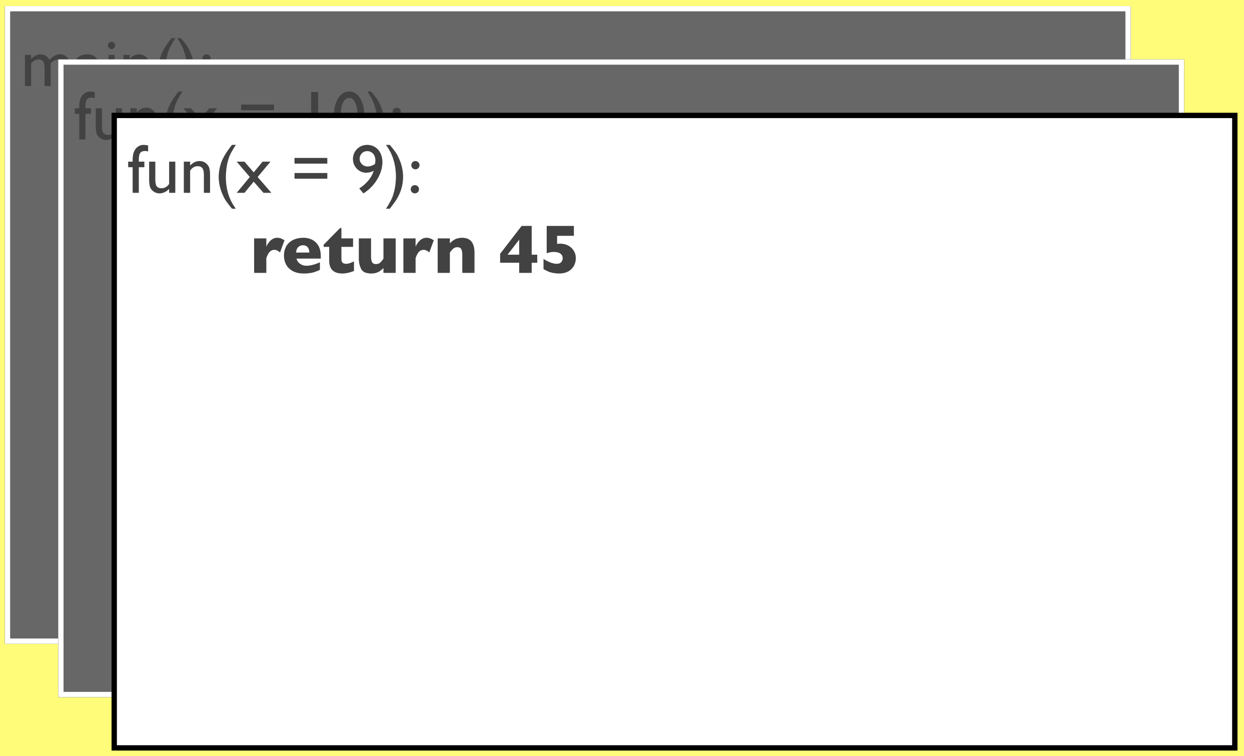
scratch pad:

```
    9 + fun(8)
=   9 + 36
=   45
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```



```
fun(x = 9):
    return 45
```

You run main()

scratch pad:

```
     9 + fun(8)
=    9 + 36
=    45
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```

```
main():
fun(x = 10):
    return 10 + fun(9)
```

You run main()

scratch pad:

```
    10 + fun(9-1)
=   10 + 45
=   55
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```

```
main():
fun(x = 10):
    return 55
```

You run main()

scratch pad:

```
     10 + fun(9-1)
=    10 + 45
=    55
```

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```

```
main():
    n = 10
    sum = 55
```

You run main()

Finally, the initial call to fun(10) returns with the correct value. This is assigned to the variable **sum**, which is then printed out.

function definitions:

```
def fun(x):
    if x == 1:
        return 1
    return x + fun(x - 1)

def main():
    n = 10
    sum = fun(n)
    print "sum of first", n,
    print "integers is", sum
```

```
main():
    n = 10
    sum = 55
    print "sum of first", n,
    print "integers is", sum
```

You run main()

```
sum of first 10 integers is 55
```

summary

- function parameters: *fixed, atomic types* vs *object types*
  - objects passed to functions are mutable
  - fixed types are not
- recursion
  - solves a larger problem in terms of a smaller one
  - argument must change each time (to avoid an infinite loop)
  - the changing argument value must be approaching the base case
  - should return the result of the sub function call plus something else