Recursive Patterns

You are currently signed up as a listener in 6.101. While listeners have full access to the assignments, etc, they are not assigned recitation sections and are not allowed to make use of open lab hours or take quizzes.

This reading is relatively new, and your feedback will help us improve it! If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch during office hours or open lab hours, or via e-mail at `6.101-help@mit.edu`.

Table of Contents

- 1) Introduction
- 2) Definitions
  - 2.1) Programming
  - 2.2) Recursion in the Environment Model
- 3) Recursion in the Wild
- 4) Summing a List
  - 4.1) Aside: Doctest
  - 4.2) Getting the Base Cases Right
- 5) Summing a Nested List
  - 5.1) `sum_nested` Recursive Diagram Examples
- 6) Choosing the Right Decomposition For a Problem
  - 6.1) Aside: Canonical Output For Doctest
  - 6.2) One Way to Do Subsequences
  - 6.3) `subsequences` Recursive Call Diagram
- 7) Choosing the Right Recursive Subproblem
- 8) Summary

1) Introduction

In some sense, this week's reading represents a slight shift of focus as we move on to a new topic. The ideas we've been talking about so far won't go away, of course, but we will have a different central focus for the next few weeks: recursion.

We expect that recursion is something that everyone should have seen in the prerequisite for this course, but recursion can feel somewhat weird and uncomfortable until you build up some experience of using it and some understanding of how it works. So, we're going to spend a fair amount of time over the next couple of weeks with that goal in mind: developing facility and comfort with, and a deeper understanding of, recursion. As usual, we'll come at this from a couple of different perspectives -- we'll certainly look at how we can use recursion in our own programs, but we'll also focus on how recursion works behind the scenes (in terms of our environment model), as well as talking about how we can go about deciding whether a recursive solution is appropriate for a given problem (or whether some other approach might be better).

And don't worry if recursion feels strange or uncomfortable; that's a natural first reaction, and, with time and practice (which we're aiming to provide over the course of the next several readings, recitations, and labs), this idea will stop being quite so scary and start being a really powerful tool that we can use in our own programs.

2) Definitions

Before we can get too far into talking about using recursion, it's worth briefly talking about what recursion is. Generally speaking, **recursion** occurs when something is defined in terms of itself.

Although our focus will be on recursion from a programming perspective, recursion can also appear in other contexts. Here is an example from mathematics, a definition of the factorial operation. For nonnegative integer n,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

In general, a recursive definition will have multiple pieces:

- one or more **base cases** (terminating scenarios that do not need recursion to produce answers), and
- one or more **recursive cases** (sets of rules that reduce all other cases toward base cases)

In our example above, the case where $n = 0$ is a base case. We don't need recursion to find the answer there, since $0! = 1$ by definition.

The other case is our recursive case. We can see the recursion here -- the definition of $n!$ depends on the definition of $(n-1)!$. So the definition of factorial depends on the definition of factorial.

Let's see how this evolves in a mathematical context, before using it in programming. Suppose we want to determine $4!$ using the definition above. We can plug things into that definition and use something like the following sequence of steps to arrive at an answer, repeatedly applying the definition of factorial until we reach a base case:

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times 3 \times 2! \\ &= 4 \times 3 \times 2 \times 1! \\ &= 4 \times 3 \times 2 \times 1 \times 0! \\ &= 4 \times 3 \times 2 \times 1 \times 1 \\ &= 24 \end{aligned}$$

In this example, we can also see the important property that the recursive case reduces us down toward a base case, in the sense that, for any value n we want to compute the factorial of, the $(n-1)!$ is applying the factorial definition to a value that is closer to our base case of $n = 0$. In this way, we can rest assured that, for any nonnegative integer n, this process will eventually reach a base case, at which point we can find our overall answer.

2.1) Programming

It turns out that Python lets us define functions recursively (i.e., we can define functions in terms of themselves). As an example, here is a Python implementation of the definition of factorial from above (note that it is almost an exact translation):

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

We could try to write this more concisely, but this is a nice form to demonstrate how this definition parallels the mathematical definition from above. There are some key things to notice here:

- This is a recursive definition, since computing `factorial` depends on the definition of `factorial` (i.e., the function calls itself in the process of computing its result in some cases).
- We have one base case. When `n == 0`, we can compute the result without any recursion.
- We have one recursive case, which reduces down toward a base case.

Before we move on to the details of how this works in Python, it's worth mentioning again that, depending on the extent of your prior exposure to these ideas, it may feel uncomfortable writing a function like this. And a big part of the weirdness here is that, as we're writing `factorial`, we're using `factorial` as part of the body, despite the fact that *we've not finished writing `factorial` yet!* But, the right strategy is to write your recursive case(s) *under the assumption that you have a complete, working version of the function*, and to think about, under those conditions, how would I take the result from that recursive call and combine it with other things to produce the result I'm interested in? And we don't need to have blind faith here; if we've designed things such that we have our base cases set up, and such that our recursive calls are working down toward one of those bases cases, then things will work out for us.

Another source of weirdness here is that, in the process of evaluating some call to `factorial`, I'm going to need to call `factorial` again and maybe many times; and each of these calls has its own value of `n`. So, how can we be sure that Python is going to keep those things separate and not get confused?

2.2) Recursion in the Environment Model

Importantly, Python is able to keep those things separate, and its ability to do so is a natural effect of our normal rules for function evaluation (that is, Python does not treat recursive functions any differently from nonrecursive functions). Let's take a look at how this plays out using an environment diagram below, for evaluating `factorial(3)`. To start with, **only proceed through step 5 below, and then answer the following question**:

Moving from step 5 to step 6 in the environment diagram below, where will F_2's parent pointer go?

Global Frame

Submit

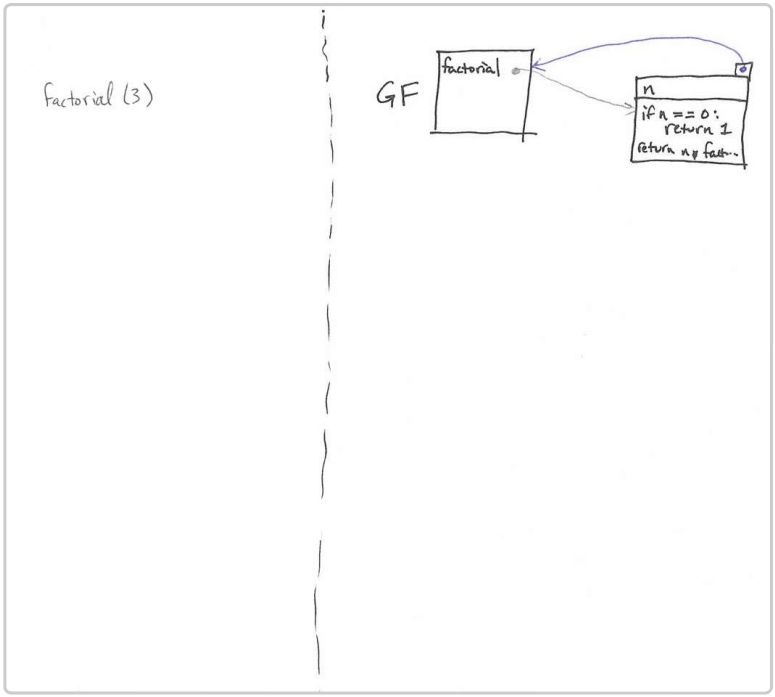
You have submitted this assignment 1 time.

Solution: Global Frame

Show/Hide Line Numbers

```
1 def factorial(n):
2   if n == 0:
3     return 1
4   else:
5     return n * factorial(n-1)
6 factorial(3)
```

<< First Step | < Previous Step | Next Step > | Last Step >>

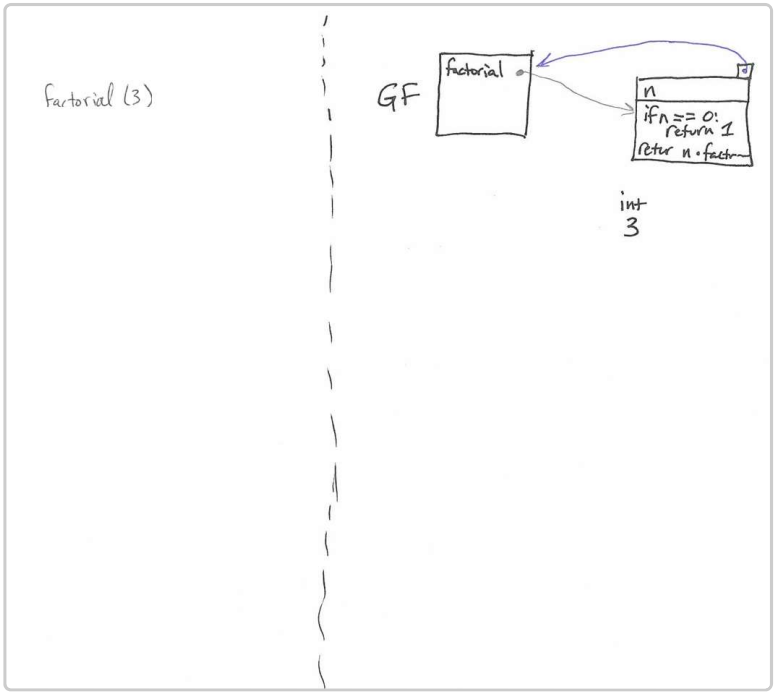


STEP 1

Here we have a diagram with two pieces: on the right-hand side, we have an environme diagram; and on the left-hand side, we have space to keep track of the expression we're evaluating. (The left-hand side is a recursive-call diagram, and we'll say more about thes in this reading.)

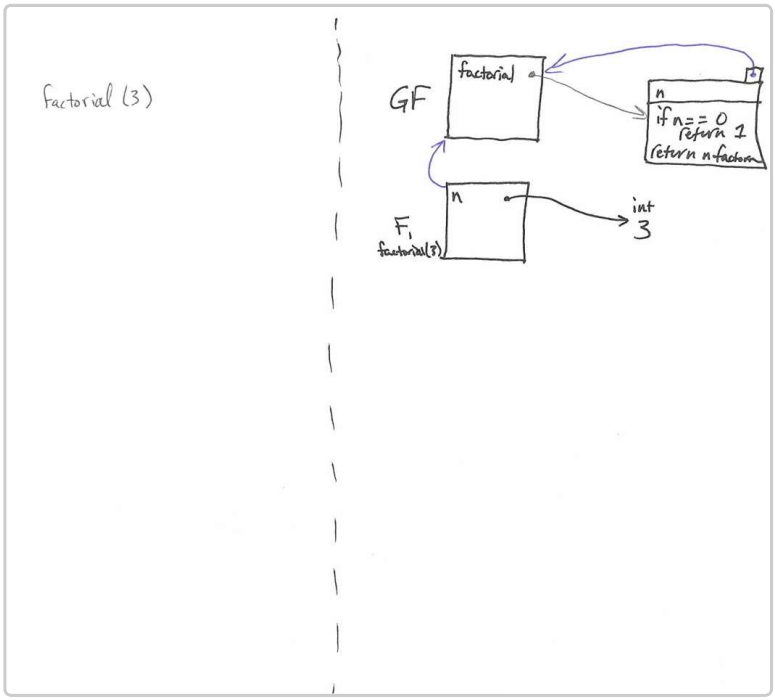
On the right-hand side, we see what the environment diagram looks like after executing function definition from above, and on the left, we see the expression we are going to e (`factorial(3)`), evaluated in the global frame (**GF**).

In order to perform this function call, we follow our usual steps. The first step is to figur what function we're calling (in this case, evaluating the name `factorial` to find the fu object on the right), as well as the arguments to the function (in this case, an integer `3`) results are shown on the next diagram.



STEP 2

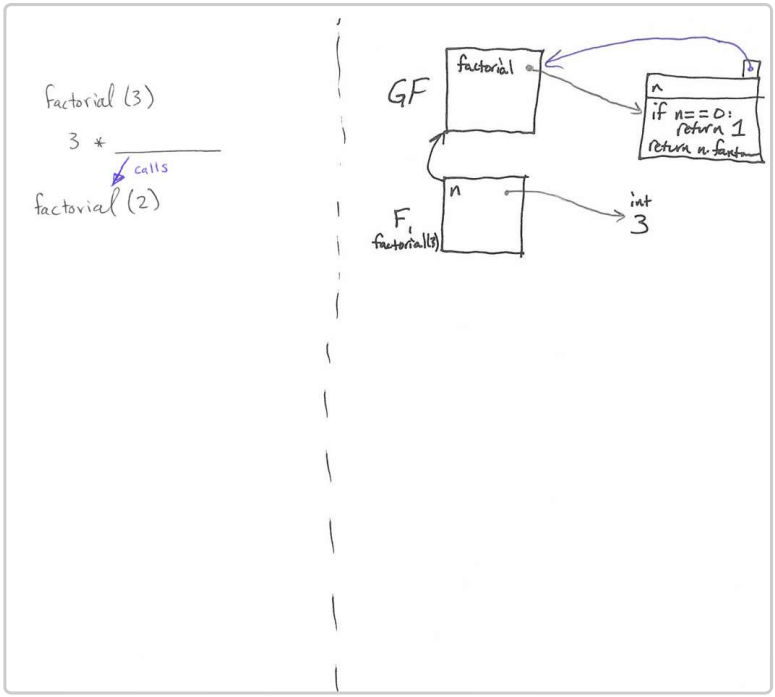
Notice that we now have a new integer `3` to work with. Our next step is to set up a new for this function call, the result of which is shown on the next diagram.



STEP 3

Here we have our new frame (**F1**, which I've also labeled with some information about t function we're calling there). Notice that we have given it a parent pointer (to the enclo frame of the function we're calling), and we've associated the name `n` with the argumer inside of **F1**.

With our new frame set up, we can now proceeed with evaluating the body of the funct respect to **F1**.



STEP 4

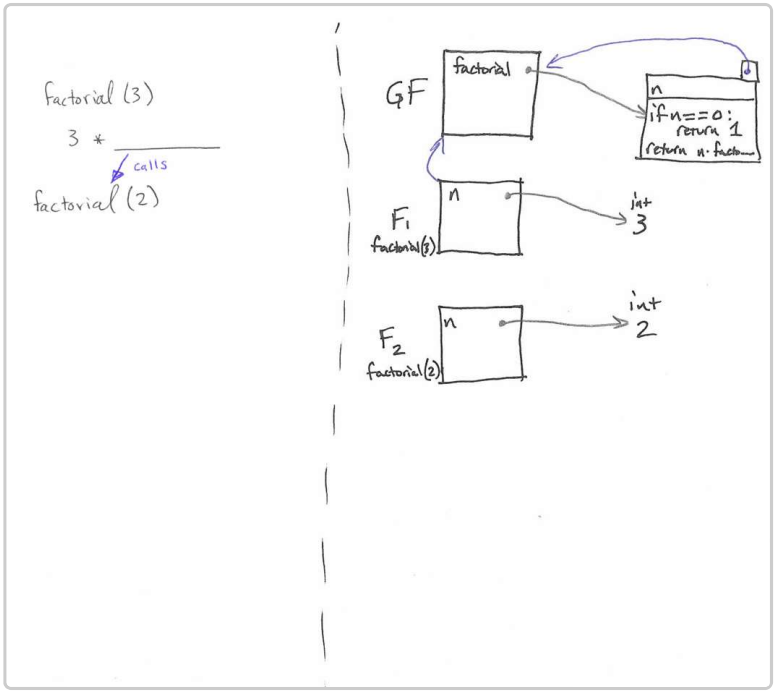
Evaluating the body, we start by checking `if n==0`. We evaluate `n` in **F1** (finding the a `3`), which is not equal to `0`, so we move on and hit `return n * factorial(n-1)`.

This is a complicated expression, but ultimately we need to multiply two things togethe integer `3` (which we get by evaluating the name `n` in **F1**) and the result of the function `factorial(n-1)`.

In order to evaluate that function call, we need to evaluate the name `factorial` in **F1** to figure out what function to call. Here, we don't find the name `factorial` in **F1**, so v its parent pointer to the global frame, where we find `factorial` bound to the functior at the top of the diagram; so that's the function we'll call!

Then we also need to determine the argument to this function call, which we do by eva `n-1` with respect to **F1**. If we're being pedantic, Python first evaluates `n` (finding `3`), the evaluates `1` (which makes a new integer `1`), then subtracts them to get a new integer `2`

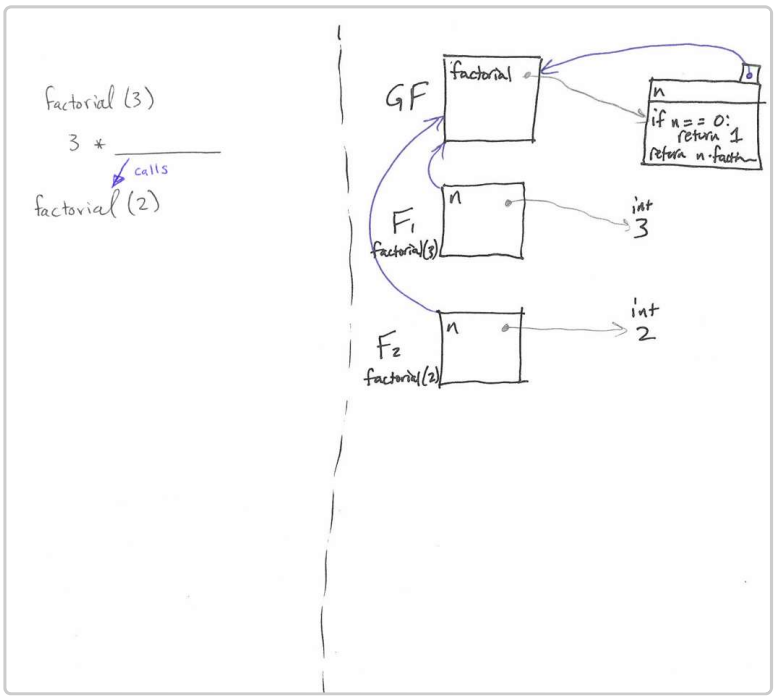
Once we know what function we're calling (the only function object drawn in the diagra what arguments to pass in (`2`), we can set up our new frame. The next slide shows the *complete* result of that step.



STEP 5

Here we have the result of setting up that new frame but with one key piece missing: ou frame's parent pointer.

STOP HERE FOR A MOMENT AND ANSWER THE QUESTION ABOVE THE DIAGRAM MOVING ON.



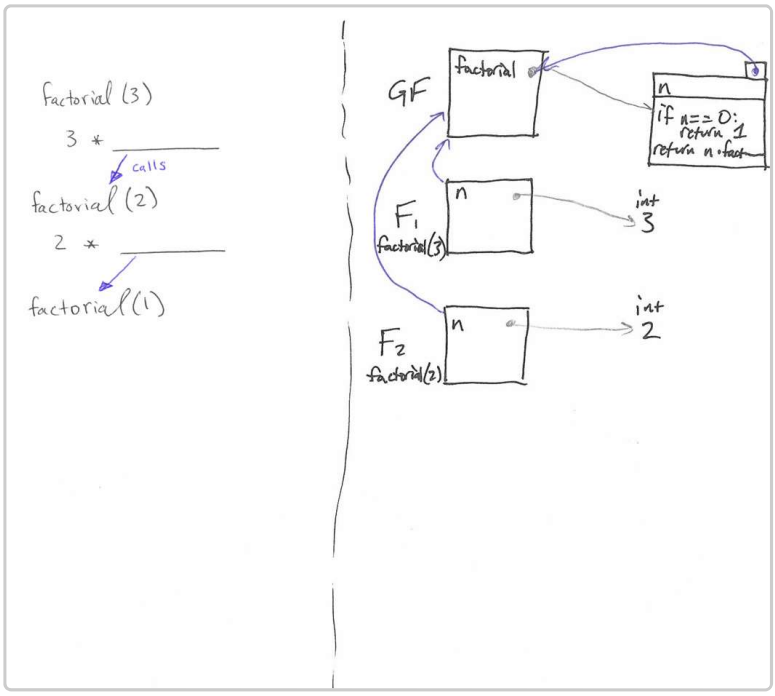
STEP 6

Here is where we need to be somewhat careful about the rules for function application! that when we make a new frame for a function call, the new frame's parent pointer goes *function's enclosing frame* (i.e., the frame where that function object was defined).

Importantly, that is true regardless of what frame we're *calling* the function from. That is matters for this rule is the frame where the function was defined, not the frame from wh being called.

If you are unclear about this, it may help to review the rules for defining and calling a fu

---



STEP 7

Now that we've got our frame completely set up, we can proceed with evaluating the b the function inside of **F2**.

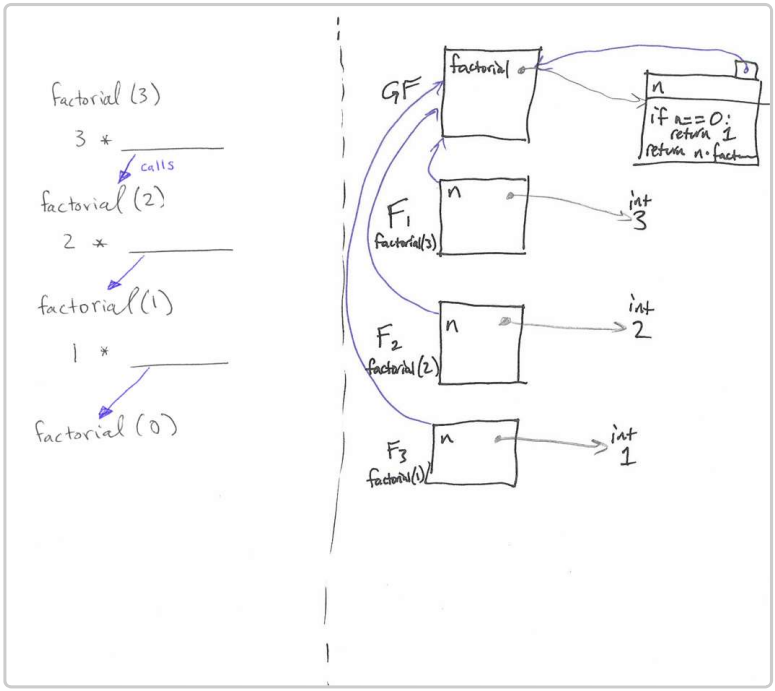
Evaluating the body, we first check `n==0` and find that that condition does not hold, so move on and evaluate `return n * factorial(n-1)` with respect to **F2**.

Evaluating that multiplication, we find `n` bound to `2`, and in order to figure out what to by, we need to evaluate `factorial(n-1)`. It's another function call! So we proceed wi same steps.

We start by figuring out what function we're calling and what arguments we're passing find the function by evaluating the name `factorial` in **F2**. Since we don't find it there follow the parent pointer to the global frame, where we find the function object drawn top of the diagram.

Then, evaluating `n-1` in **F2**, we get `1`. So now that we have our function and our argum can proceed with following our normal sequence of steps (setting up a new frame (with pointer), binding the parameters to the arguments that were passed in, and evaluating t of the function in that new frame).

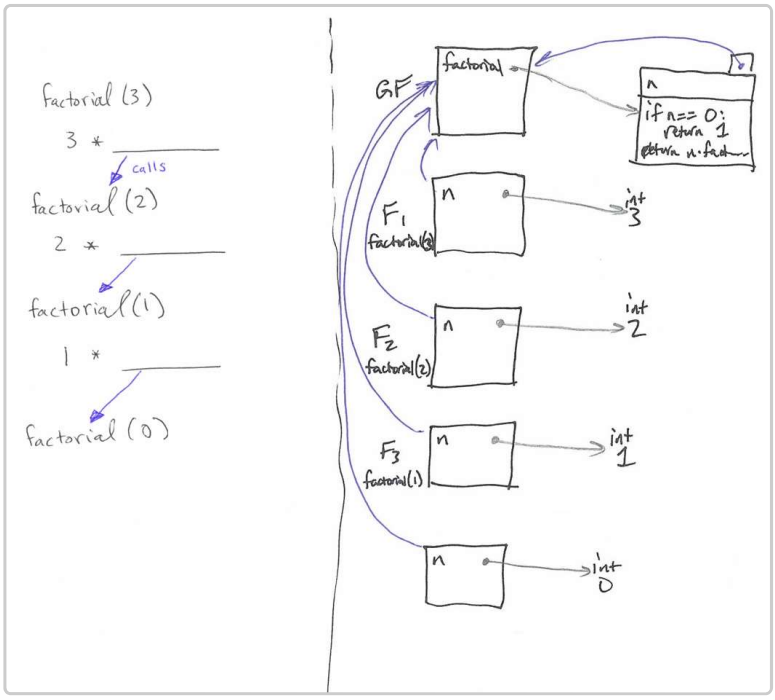
You may wish to pause here and try to draw the result of setting up the next frame for y Where should its parent pointer go? What names should be bound in it, and to what ob they bound? What will be the result of evaluating the body of the function in this new f



STEP 8

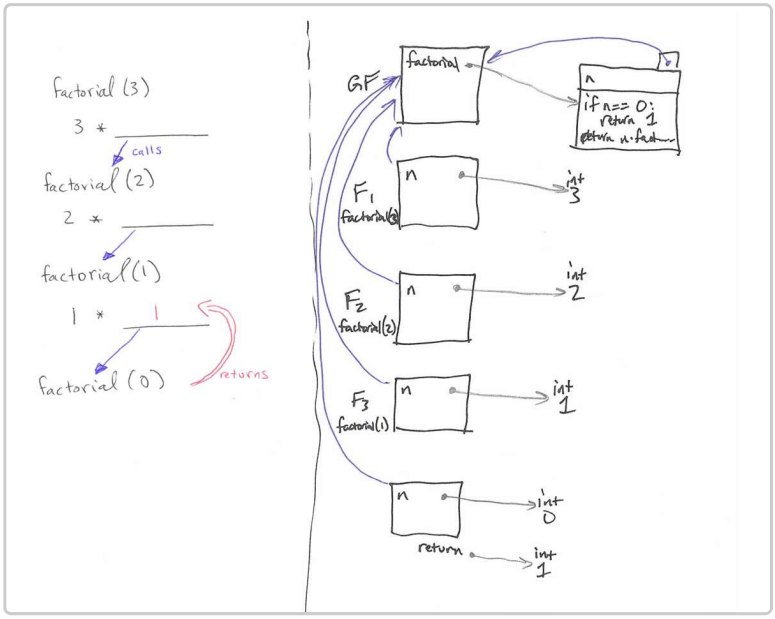
Here we see the result we arrive at after setting up our new frame **F3**. The process was t as what we just did for frame **F2**, so the parent frame is **GF**, but the name `n` is now bou value `1`.

When we evaluate the body of the function in **F3**, we first check `n==0`, and, since that e to `False`, we continue on to evaluating `n * factorial(n-1)` in **F3**. Here we have tc evaluate another function call, calling the same function but now with `0` passed in. So v up another frame, shown on the next step.



STEP 9

Notice that our new frame (which we'll call **F4** even though I forgot to write that label n above) has the global frame as its parent, and it has the name `n` bound to `0`. This has admittedly been a somewhat tedious process, but we've now arrived at a base case, and something very exciting is about to happen! Let's see what happens when we evaluate t of the function in **F4**.

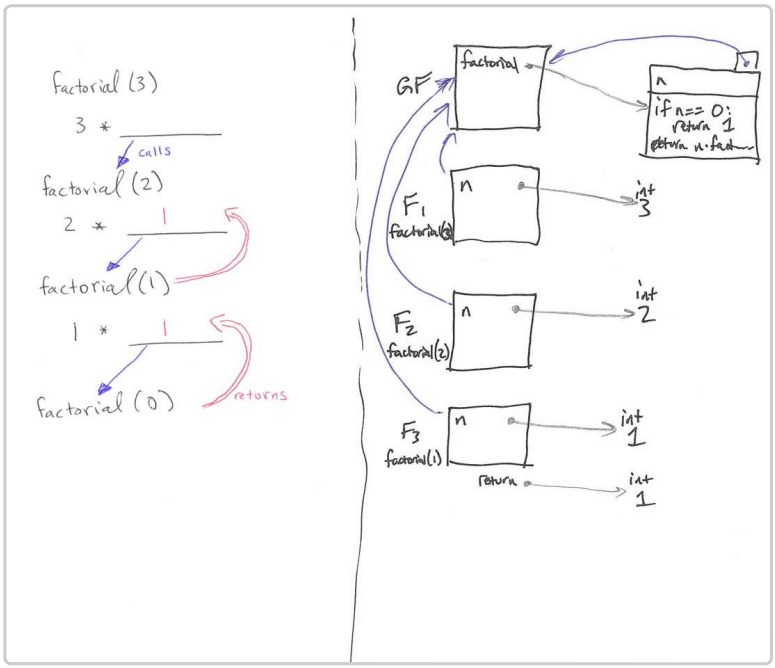


STEP 10

Here, we first check `if n == 0`, and, finally, it does! So we evaluate `1` in **F4** to get a ne integer object, which we return.

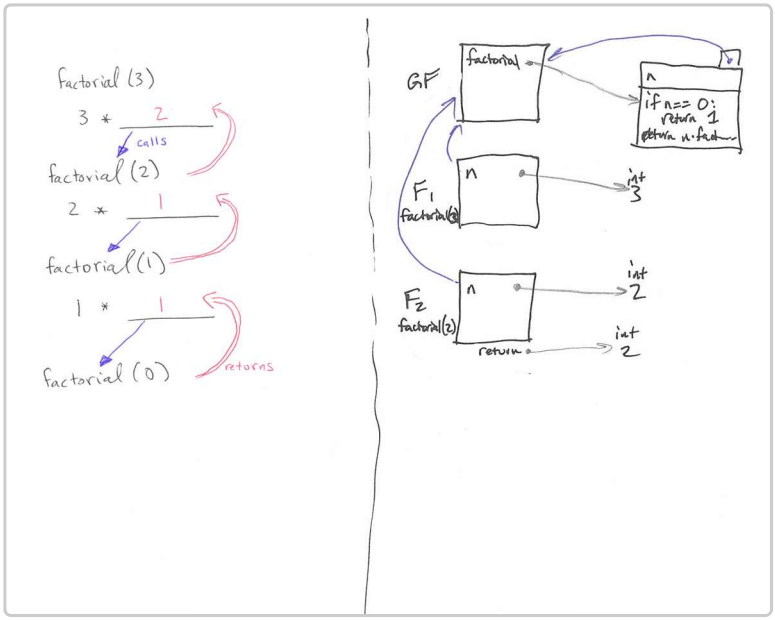
But this was just the return from our most recent function call! Let's remember how we We were evaluating `n * factorial(n-1)` in **F3**. So now we can go back and finish th computation, taking the result of our recursive call (`1`) and multiplying it by `1` to get ou

---



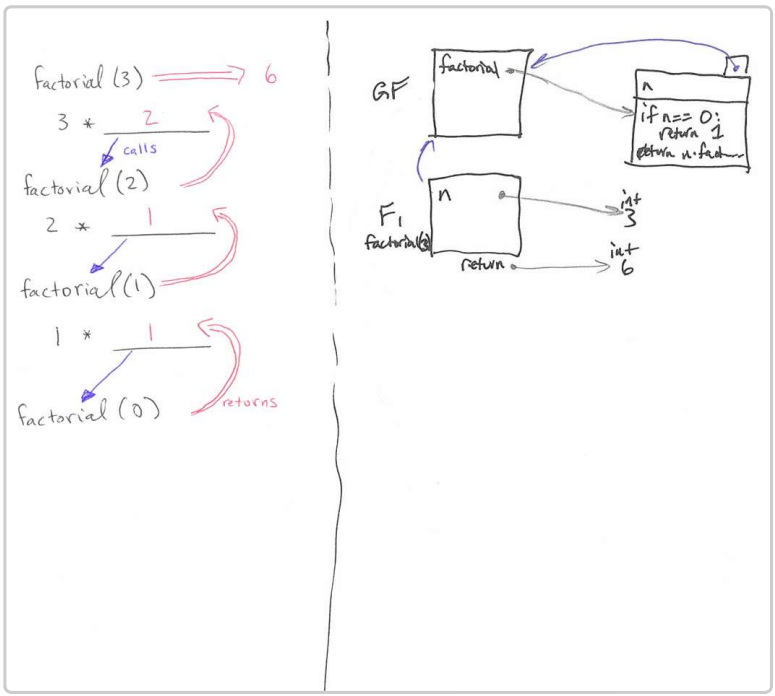
STEP 11

Notice that we're returning `1` from the function call that created **F3**. But how did we ge
We were evaluating `2 * factorial(1)` in **F2**! So we can take this result (`1` returned l
`factorial(1)`) and combine it with `2` to get the return value from **F2**!



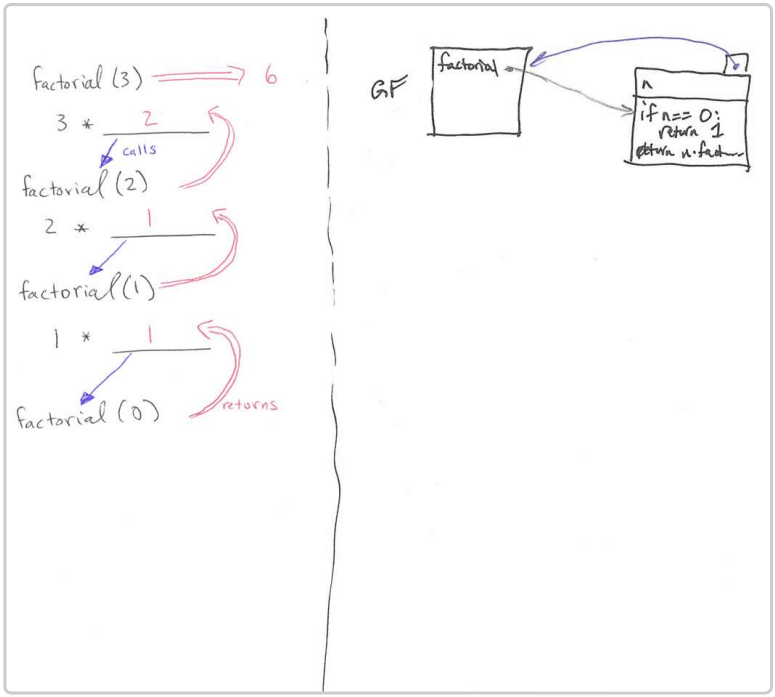
STEP 12

And now that we know *this* value, we can combine it with our `3` to get the overall result
(from our original function call).



STEP 13

And, finally, we have found the result of our original call! And the very last step is to cle



STEP 14

This was quite a process, but hopefully it was helpful to walk through the steps here to things were working themselves out as this program ran. Of course, if you have questio this process, please don't hesitate to ask them!

As a follow-on question, with factorial written as above, how many total frames (other than the global frame and the built-ins) are created when we call `factorial(7)`?

8

Submit

You have submitted this assignment 3 times.

Solution: 8

3) Recursion in the Wild

To reinforce the idea that some problems are naturally recursive, let's look at a quick example from a very practical program: CAT-SOOP, the software that is running the website you're looking at right now. CAT-SOOP is written in Python. One of its tasks is to store information about submissions that are made and other events that happen while you are using it, so it has a notion of a *log* that records information on disk about those events.

We don't want to get too much into the details about how CAT-SOOP does logging, but here is one part of the system that is particularly relevant to our discussion: a function that tests whether some arbitrary Python value can be stored in the log or not.

```
def can_log(x):
    """
    Checks whether a given value can be a log entry.
    Valid log entries are strings, ints, floats, complex numbers, None, or Booleans;
    _or_ lists, tuples, sets, frozensets, dicts, or OrderedDicts containing only valid log entries.
    """
```

Look at the docstring of this function: it gives us a nice description of which values can be valid log entries. What about this description suggests that recursion might be the best way to implement this function?

The docstring defines what a "valid log entry" is in a way that *depends on* the definition of a valid log entry!

> **Valid log entries** are strings, ints, floats, etc. *or* lists, tuples, sets, etc. containing only **valid log entries**.

Definitions like this are extremely common in programming, data representation, and software systems in general.

How would we implement this recursively? First notice that the definition has a base case:

```
    if isinstance(x, (str, bytes, int, float, complex, NoneType, bool)):
        return True
```

If `x` is one of these types, then we're done, we don't need to do any further recursion at all.

But, then, we've also got these recursive cases: if `x` is one of the collection types, like `list`, `tuple`, or `set`, then

```
    elif isinstance(x, (list, tuple, set, frozenset)):
        return all(can_log(v) for v in x)
```

Notice that the recursive call to `can_log` is applied to every element of the collection. (The built-in function `all` tests whether every element of an iterable is true, so it effectively `and`s together the results of all those recursive calls.) Most importantly, although those elements of the collection *could* be simple strings or ints (base cases), they could also be lists or sets or other collections themselves, thus needing still further recursion. The recursive implementation automatically takes care of that, no matter how complex the nesting gets.

We also need to handle dictionaries, which need two recursive calls for each (key,value) pair:

```
    elif isinstance(x, (dict, OrderedDict)):
        return all((can_log(k) and can_log(v)) for k,v in x.items())
```

And, finally, there's a second base case, which is if `x` does not have any of the types we checked for, then it's not a valid log entry, so we need to return `False`. Here's the full function:

```
def can_log(x):
    """
    Checks whether a given value can be a log entry.
    Valid log entries are strings/bytestrings, ints, floats, complex numbers,
    None, or Booleans; _or_ lists, tuples, sets, frozensets, dicts, or
    OrderedDicts containing only valid log entries.
    """
    if isinstance(x, (str, bytes, int, float, complex, NoneType, bool)):
        return True
    elif isinstance(x, (list, tuple, set, frozenset)):
```

```
        return all(can_log(v) for v in x)
    elif isinstance(x, (dict, OrderedDict)):
        return all((can_log(k) and can_log(v)) for k,v in x.items())
    return False
```

The details of this function don't matter as much as the big picture: this is a practical problem in which the structure of the problem was such that a recursive solution really jumps out as the simplest and most appropriate way to do it.

4) Summing a List

Let's dig into another example: summing a list of numbers:

```
def sum_list(x):
    """
    Compute the sum of a list of numbers, recursively.
    """
    pass
```

We could use the builtin function `sum()` to do this, of course, or we could write an iterative solution, which is what `sum()` is doing anyway:

```
def sum_list(x):
    out = 0
    for num in x:
        out += num
    return out
```

We'll talk more about the tradeoffs between iteration and recursion in the next reading. For now, though, let's try to do everything recursively. Intuition from these simple examples will help for more complicated situations where a builtin function doesn't exist or where an iterative solution is not the best way to do it.

Check Yourself:

First, let's think about what our base case(s) might look like. For what inputs to `sum_list` can we return the answer right away, without any work?

Show/Hide

Here's one possibility: if the list has only one element in it, then we don't have to do any addition. We can just return that element:

```
# BASE CASE
if len(x) == 1:
    return x[0]
```

This is not the only base case we might use, and maybe it's not even the best choice, but let's try it for now.

Check Yourself:

Given that, what could we use as a recursive case?

Show/Hide

How about this:

```
# RECURSIVE CASE
else:
    return x[0] + sum_list(x[1:])
```

This corresponds to the mathematical expresion

$$\sum_{i=0}^{n-1} x[i] = x[0] + \sum_{i=1}^{n-1} x[i]$$

which is indeed true for $n > 1$.

So now our whole function looks like:

```
def sum_list(x):
    if len(x) == 1:
        return x[0]
    else:
        return x[0] + sum_list(x[1:])
```

4.1) Aside: Doctest

We're not done with `sum_list` yet, but let's take a moment to write some tests and see another useful Python tool that may not be familiar to you. There's a module built into Python called `doctest`, which lets us embed simple test cases inside the docstring of a function. The tests are formatted as if they were typed in the Python prompt, using `>>>` to indicate what should be typed, followed by what the expected output should be:

```
def sum_list(x):
    """
    Compute the sum of a list of numbers, recursively.

    >>> sum_list([1,2,3,4,5])
    15
    """
    if len(x) == 1:
        return x[0]
    else:
        return x[0] + sum_list(x[1:])
```

This is already helpful because it gives examples of using the function to a human reading the docstring. But, `doctest` goes a step further by automatically extracting these test cases and running them as automatic tests. If you call `doctest.testmod()`, e.g., in the main block of your file, then it will run all the docstring tests it can find in that file:

```
... # definition of sum_list

import doctest
if __name__ == '__main__':
    doctest.testmod(verbose=True)
```

Now, running the whole file will invoke all the docstring tests, and we'll see that it passed:

```
$ python sum_list.py
Trying:
    sum_list([1,2,3,4,5])
Expecting:
    15
ok
1 passed and 0 failed.
Test passed.
```

It's a good idea to include `verbose=True` as a parameter to `testmod()` -- otherwise, when all the tests pass, as in this case, `doctest` will simply succeed silently.

We can put as many doctests as we want into a docstring, so let's add another, which looks at what happens when we sum an empty list:

```
def sum_list(x):
    """
    Compute the sum of a list of numbers, recursively.

    >>> sum_list([1,2,3,4,5])
    15
    >>> sum_list([])
```

```
        0
        """
        if len(x) == 1:
            return x[0]
        else:
            return x[0] + sum_list(x[1:])


Trying:
    sum_list([1,2,3,4,5])
Expecting:
    15
ok
Trying:
    sum_list([])
Expecting:
    0
**********************************************************************
File "/Users/rcm/6.101/fall22/web/readings/recursion/sum_list.py", line 8, in __main__.sum_list
Failed example:
    sum_list([])
Exception raised:
    Traceback (most recent call last):
      File "/Users/rcm/.pyenv/versions/3.10.1/lib/python3.10/doctest.py", line 1346, in __run
        exec(compile(example.source, filename, "single",
      File "<doctest __main__.sum_list[1]>", line 1, in <module>
        sum_list([])
      File "/Users/rcm/6.101/fall22/web/readings/recursion/sum_list.py", line 14, in sum_list
        return x[0] + sum_list(x[1:])
    IndexError: list index out of range
**********************************************************************
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Oops! We have a failure. Can you see why it's going wrong? We'll see how to fix it in the next section.

One big caveat about doctest: it works by checking *printed output*, not by using `==` to compare the expected value with the actual value (which is what we do with `assert` in other kinds of tests we write). This has two unfortunate effects:

- if you put any `print` statements into your code to debug it, those prints will cause doctests to fail, because the docstring won't mention them as expected output. You have to remove or comment out all your debugging prints in order for the test to pass again.
- if there is any harmless variation in the output -- for example, the order in which a set or dictionary is printed -- then the test will fail, even if the set or dictionary is actually correct. We'll see one way to deal with this problem below.

That said, the convenience and understandability of having executable test cases right in a function's documentation can be a big win.

4.2) Getting the Base Cases Right

Let's add a base case to deal with the empty list:

```
def sum_list(x):
    if len(x) == 0:
```

```
        return 0
    elif len(x) == 1:
        return x[0]
    else:
        return x[0] + sum_list(x[1:])
```

Now the code works, but, if we reflect on it a little more, do we need *both* of these base cases? No! Now that we handle the length-0 list correctly, the length-1 base case is redundant with the recursive case. To see this more clearly, we can imagine having base cases for length-2, length-3, and so forth:

```
def sum_list(x):
    if len(x) == 0:
        return 0
    elif len(x) == 1:
        return x[0]
    elif len(x) == 2:
        return x[0] + x[1]
    elif len(x) == 3:
        return x[0] + x[1] + x[1]
    else:
        return x[0] + sum_list(x[1:])
```

This code is not DRY (that is, it doesn't follow the principle of "don't repeat yourself"). The more redundant base cases you have, the more places there are for bugs to hide. (In fact, there *is* a bug hiding in this example -- can you see it?)

So, let's simplify to the fewest base cases and recursive cases we can manage with. We'll also write the empty-list test slightly more pythonically, since empty lists are falsy:

```
def sum_list(x):
    if not x:
        return 0
    else:
        return x[0] + sum_list(x[1:])
```

5) Summing a Nested List

Now, let's think about a slightly different problem, summing lists of numbers where numbers may be inside sublists, down to an arbitrary depth:

```
def sum_nested(x):
    """
    >>> sum_nested([[1, 2], [3, [4, 5]], [[[[[6]]]]]])
    21
    """
    pass
```

The fact that the sublists can go down to arbitrary depth makes this problem feel hard. Fortunately, the empty list is still our base case:

```
    if not x:
        return 0
```

But, what about the recursive case? Adapting the recursive case from `sum_list`, which peels off the first element and then adds it to the recursive sum of the remaining elements, doesn't work here:

```
    return x[0] + sum_nested(x[1:])
```

The reason it doesn't work is that the first element `x[0]` is not just a number; it might be a list. If it's a list, we need to do something different, to turn the sublist `x[0]` into a number that we can add to the number coming from `sum_nested(x[1:])`.

So, let's try distinguishing those two cases with another `if` test:

```
    if isinstance(x[0], list):
        return SOMETHING + sum_nested(x[1:])
    else:
        return x[0] + sum_nested(x[1:])
```

Now, we have two recursive cases: one used when the first element is a list and another when the first element is just a number. That second case we can handle just as before. But, what should `SOMETHING` be, when `x[0]` is a sublist? We might try digging out elements of that sublist:

```
    if isinstance(x[0], list):
        return x[0][0] + sum_nested(x[1:])
```

But this won't be right if the sublist has more than one element (or less than 1 element!) We might fall back to iteration:

```
    if isinstance(x[0], list):
        subtotal = 0
        for x in x[0]:
            subtotal += x
        return subtotal + sum_nested(x[1:])
```

But we're trying to solve this *recursively*, and worse, this doesn't work either! Is `x` guaranteed to be a number here? No, it could itself be another sublist. `x[0]` could have sublists in it down to an arbitrary depth!

Let's take a step back. We already have a way to sum up `x[0]` if it's an arbitrarily deep list of lists. That way is `sum_nested` itself! We just have to lean in and trust the recursion:

```
    if isinstance(x[0], list):
        return sum_nested(x[0]) + sum_nested(x[1:])
```

Again, this may seem magical. We are now using `sum_nested` twice in the same line! Why doesn't this end up just calling itself endlessly and never returning? The key thing to observe is that every recursive call to `sum_nested` is using a list that is strictly *smaller* or *simpler* in some respect, heading towards some minimum represented by the base case. Here, we have peeled apart our original nested list into two pieces, `x[0]` and `x[1:]`, each of which is smaller.

Here is the full code:

Show/Hide Line Numbers

```
1  def sum_nested(x):
2      """
3      >>> sum_nested([[1, 2], [3, [4, 5]], [[[[6]]]]])
4      21
5      """
6      if not x:
7          return 0
8      elif isinstance(x[0], list):
9          return sum_nested(x[0]) + sum_nested(x[1:])
10     else:
11         return x[0] + sum_nested(x[1:])
```

5.1) `sum_nested` Recursive Diagram Examples

As we saw with the factorial environment diagram example, creating environment diagrams with recursion can take a lot of work. Additionally, because so many frames are created in the course of executing even a small recursive program like `factorial(3)`, it can be difficult to keep track of the course of program execution and determine which frame we are in, which frame we should return to, and where in the program to resume execution following the completion of a recursive call.

The diagram we drew on the left hand side of the environment diagram is called a recursive-call diagram. Regular environment diagrams are great for keeping track of the state of the stack, heap, object aliasing, and scoping for small programs. Recursive diagrams were designed to keep track of the tree of recursive function calls. In particular, they trace inputs, outputs, and what frame each recursive call originated from so we know where to return to. Recursive call diagrams won't be tested on exams, but they are a helpful tool for visualizing how small recursive programs work.

Now that we've explained a bit about what recursive call diagrams are, let's look at some example diagrams for `sum_nested` which should help explain how the function works:

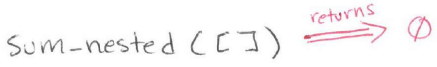
Show/Hide Line Numbers

```
1  def sum_nested(x):
2    if not x:
3      return 0
4    elif isinstance(x[0], list):
5      return (sum_nested(x[0]) +
6              sum_nested(x[1:]))
7    else:
8      return x[0] + sum_nested(x[1:])
9
10 ans = sum_nested([])
```

<< First Step | < Previous Step | Next Step > | Last Step >>

STEP 1

Let's start with thinking about the simplest possible valid input. What happens when we call `sum_nested` with an empty list as its argument?



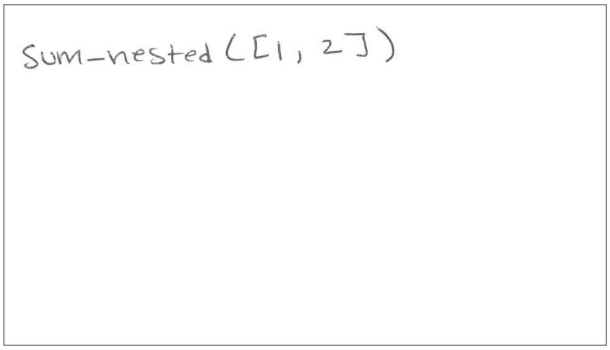
STEP 2

`sum_nested([])`
Because x is the empty list, the condition `if not x` evaluates to `True` and `0` is returned to the global frame and stored in `ans`. Because this input falls into the base case, no additional recursive calls are made.

Show/Hide Line Numbers

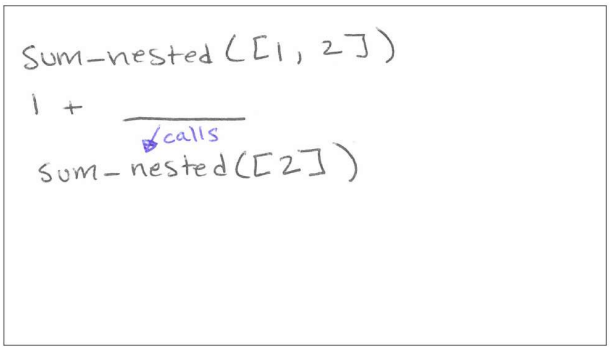
```
1  def sum_nested(x):
2    if not x:
3      return 0
4    elif isinstance(x[0], list):
5      return (sum_nested(x[0]) +
6              sum_nested(x[1:]))
7    else:
8      return x[0] + sum_nested(x[1:])
9
10 ans = sum_nested([1,2])
```

<< First Step | < Previous Step | Next Step > | Last Step >>



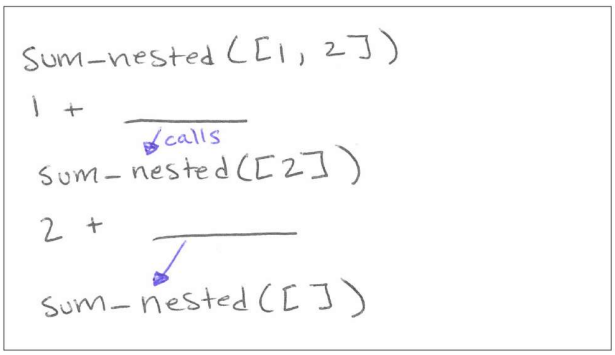
STEP 1

Ok now let's think about a slightly more complex example. What happens when the input to `sum_nested` is a non-empty flat list of numbers?



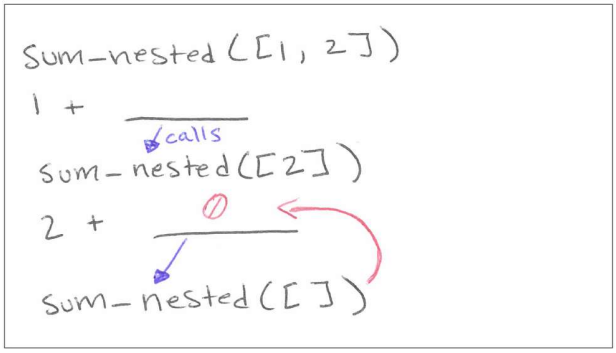
STEP 2

`sum_nested([1,2])`
Because `x[0]` is not a list, it falls into the second recursive else case which adds `1` to the result of the recursive call to `sum_nested([2])`.



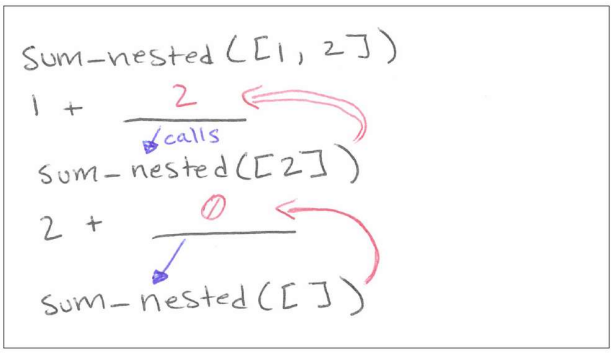
STEP 3

`sum_nested([2])`
`x[0]` is an integer, so it also falls into the second recursive case which adds `2` to the result of a recursive call on the empty list.



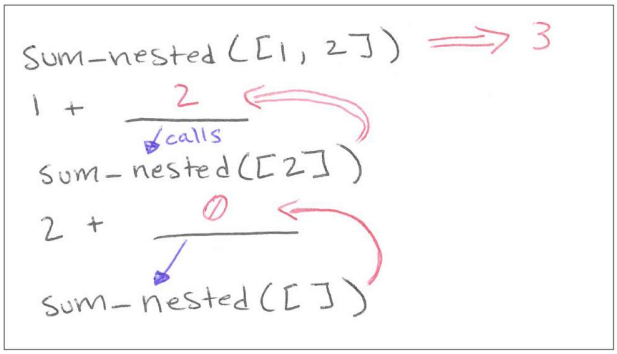
STEP 4

`sum_nested([])`
Now we have reached a base case! This returns `0` back to the `sum_nested([2])` frame that made the recursive call.



STEP 5

`sum_nested([2])`
Now that we have the result of `sum_nested([])`, the result of `2` is computed returned to the `sum_nested([1, 2])` frame that made the original recursive call.



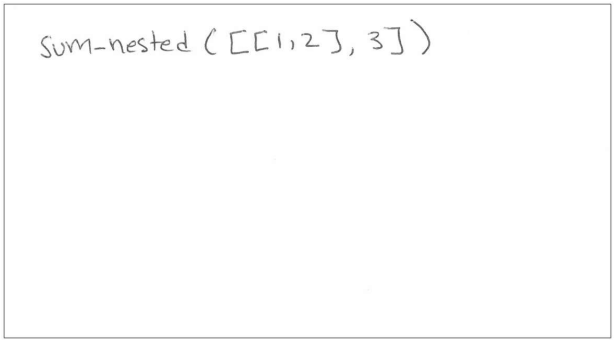
STEP 6

`sum_nested([1, 2])`
Now that we have the result of `sum_nested([2])`, the final result of `3` is computed and returned to back to the global frame where the original call to `sum_nested` was made in line 10.

Show/Hide Line Numbers

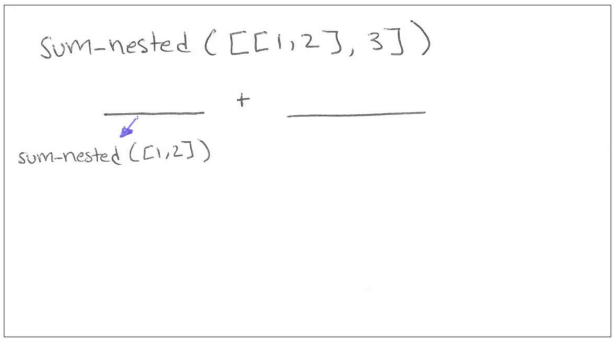
```
1  def sum_nested(x):
2    if not x:
3      return 0
4    elif isinstance(x[0], list):
5      return (sum_nested(x[0]) +
6              sum_nested(x[1:]))
7    else:
8      return x[0] + sum_nested(x[1:])
9
10 ans = sum_nested([[1,2], 3])
```

<< First Step | < Previous Step | Next Step > | Last Step >>



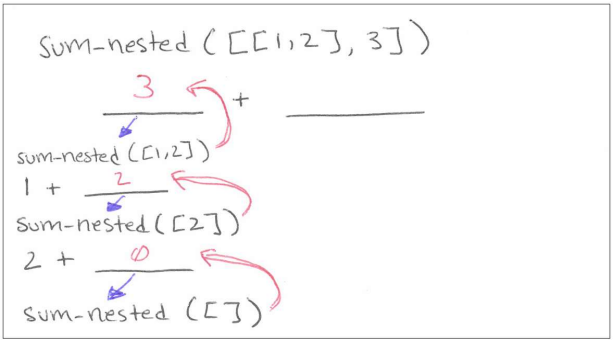
STEP 1

Let's look at one final example. What if the list is nested?



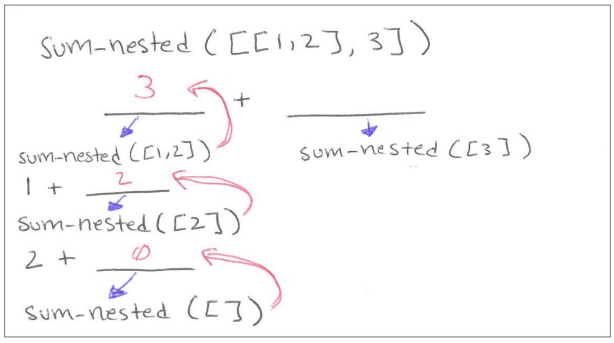
STEP 2

`sum_nested([[1, 2], 3])`
Because `x[0]` is a list in this example, it falls into the first recursive case which makes two separate recursive calls. Because Python executes expressions from left to right, the recursive call to `sum_nested(x[0])` is executed first.



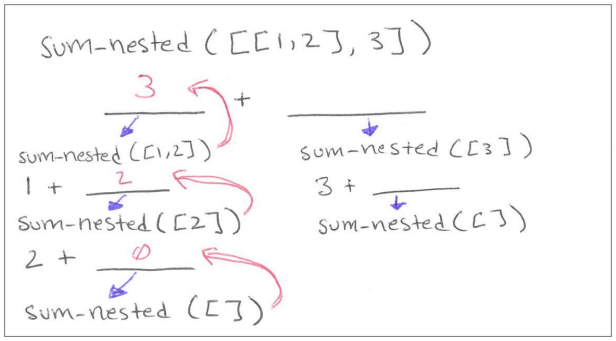
STEP 3

`sum_nested([1, 2])`
Because `x[0]` is an integer this falls into the second recursive case. Notice how this input is the same as the previous example. It will produce the same call structure as in the previous example, which computes and returns `3` back to the top-level recursive frame.



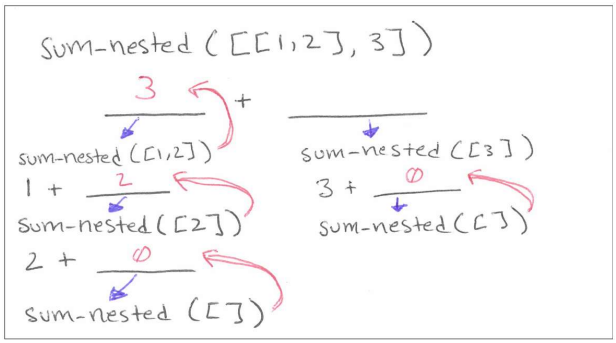
STEP 4

`sum_nested([[1, 2], 3])`
After the first recursive call `sum_nested([1,2])` finished executing and returned `3`, which allows the next recursive call to `sum_nested([3])` to start executing.



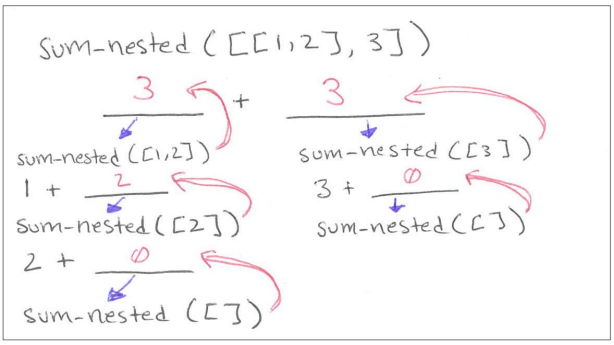
STEP 5

`sum_nested([3])`
`x[0]` is an integer, so it also falls into the second recursive case which adds `3` to the result of a recursive call on the empty list.



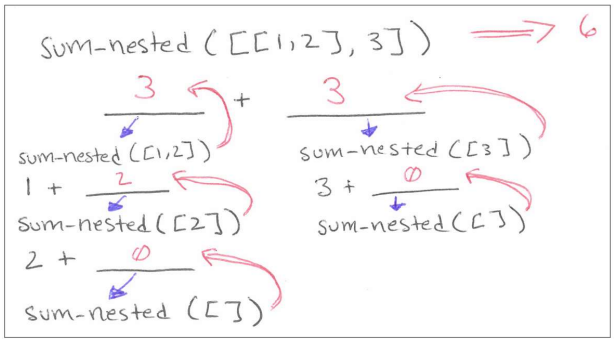
STEP 6

`sum_nested([])`
Base case, returns `0` to the `sum_nested([3])` frame that made the recursive call.



STEP 7

`sum_nested([3])`
Once the recursive call finished returning the result of `0`, this frame computes the result `3` and return it to the original `sum_nested` frame.



STEP 8

`sum_nested([[1, 2], 3])`
Now that both recursive calls have finished executing, the final result of `6` is computed and returned back to the global frame in line 10.

As a follow-on question, how many total calls to `sum_nested` will be made when we call `sum_nested([[[7, 2]]])`?

7

Submit

You have submitted this assignment 3 times.

Solution: 7

How many calls to `sum_nested` will be made with the empty list `[]` as the argument when we call `sum_nested([[1, 2], [3, [4, 5]], [[[[[6]]]]]])`?

9

Submit

You have submitted this assignment 31 times.

Solution: 9

Explanation:

Each list in the input will eventually reach the base case. Counting the number of square brackets, we find there are 9 total lists in the input, so `sum_nested([])` will get called nine times. Overall this example will call `sum_nested` 23 times!

6) Choosing the Right Decomposition For a Problem

Finding the right way to decompose a problem into smaller steps is an essential part of programming, and recursion is an elegant and simple decomposition for some problems. Suppose we want to implement this function:

```
def subsequences(seq):
    """
    Given a tuple or list or other iterable, returns a set of tuples
    consisting of all its subsequences.
    A subsequence is a sequence of elements from seq that are in the same
    order as in seq but not necessarily adjacent.

    >>> subsequences([4,2,3])
    {(4, 2, 3), (4, 2), (4, 3), (2, 3), (4,), (2,), (3,), ()}
    """
    pass
```

Fill in the blank for this additional doctest for `subsequences`:

```
>>> subsequences(["x"])
_____
```

{("x",), ()}

Submit

You have submitted this assignment 3 times.

Solution: `{('x',), ()}`

6.1) Aside: Canonical Output For Doctest

Oops, before we work on implementing `subsequences` recursively, let's fix a problem with these doctests. Doctest works by matching the actual printed output against what we wrote in the doctest, but a set can print in any order. All of the following would actually be the same set:

```
{(4,2,3), (4,2), (4,3), (2,3), (4,), (2,), (3,), ()}
{(4,), (4,2), (4,3), (4,2,3), (2,), (2,3), (3,), ()}
{(), (2,), (2,3), (3,), (4,), (4,2), (4,2,3), (4,3)}
```

... but only the first one would pass the doctest as we wrote it. So, let's make a small change to improve the test by putting its output in a *canonical* form, in this case putting the tuples in sorted order. To do that, we'll have to print a list instead of a set:

```
def subsequences(seq):
    """
    ...
    >>> sorted(subsequences([4,2,3]))
    [(), (2,), (2, 3), (3,), (4,), (4, 2), (4, 2, 3), (4, 3)]
    """
    pass
```

Note that tuples are sorted lexicographically, i.e., by comparing one element at a time and stopping as soon as an element is different or one tuple runs out of elements (in which case the shorter tuple wins).

This example shows that a doctest doesn't have to be just a single call to the function -- it can do some other work as well, in order to make the test more tolerant of harmless variation.

But, keep in mind that, even though our doctest converts the set into a sorted list for the sake of consistent printing, the `subsequences` function itself is still just working with sets.

Fill in the blank for this improved doctest for `subsequences`:

```
>>> sorted(subsequences(["x"]))
_____
```

[(), ('x',)]

Submit

You have submitted this assignment 7 times.

Solution: `[(), ('x',)]`

6.2) One Way to Do Subsequences

The `subsequences` problem lends itself to an elegant recursive decomposition. Consider the first element of `seq`. We can form one set of subsequences that *skip* that element, and we can form another set of subsequences that *include* that element, using the subsequences of the remaining elements. Those two sets generate all possible subsequences for the original input:

```
def subsequences(seq):
    if not seq:
        # base case:
        # subsequences of an empty sequence is the set of empty sequence (tuple)
        return {()}
    else:
        # recursive case:
        # split seq into two smaller sequences
        first = seq[0]
        rest = seq[1:]
        # recursively find all possible subsequences in rest
        rest_seq = subsequences(rest)
        # iteratively combine first with all subsequences in rest
        first_seq = {(first,) + sub_seq for sub_seq in rest_seq}
        # final result is combination of all subsequences with and without first
        return first_seq | rest_seq
```

What is the first recursive call made by `subsequences([4,2,3])`? Write your answer as the Python list that is passed as its argument.

[2,3]

Submit

You have submitted this assignment 2 times.

Solution: `[2, 3]`

What is the return value of that first recursive call? Write your answer as a Python expression.

{(),(2,),(2,3),(3,)}

Submit

You have submitted this assignment 2 times.

Solution: `{(2, 3), (2,), (3,), ()}`

We might try to write the computation of the `result` set in one line, like this:

```
    result = ({(first,) + subseq for subseq in subsequences(rest)} |
                {subseq for subseq in subsequences(rest)})
    return result
```

Which of the following describes the effect of this code change?

- [ ] It contains a Python syntax error.
- [ ] It will raise an exception when it runs.
- [x] It will make far more recursive calls than before.
- [ ] It will return the wrong answer because the result is being created in a different order.
- [ ] It will work as expected in about the same time.

Submit

You have submitted this assignment 1 time.

Solution:

- [ ] It contains a Python syntax error.
- [ ] It will raise an exception when it runs.
- [x] It will make far more recursive calls than before.
- [ ] It will return the wrong answer because the result is being created in a different order.
- [ ] It will work as expected in about the same time.

Explanation:

The code still works, but, instead of just making one recursive call in each recursive case, it calls `subsequences(rest)` twice. Each of those calls then makes 2 more calls, for a total of 4 at the next level; then 8 at the next level; then 16 at the next level. By the time we reach the base case, after peeling off all n elements from the original length-n sequence, we will have needed roughly 2^n calls to `subsequences`, compared to the n calls that the original recursive soluton used. It's very easy to inadvertently turn a linear recursive algorithm into an exponential one!

And, even though the `result` set is being put together in a different order than before, the order for a set doesn't matter, so the answer will still be the same.

6.3) `subsequences` Recursive Call Diagram

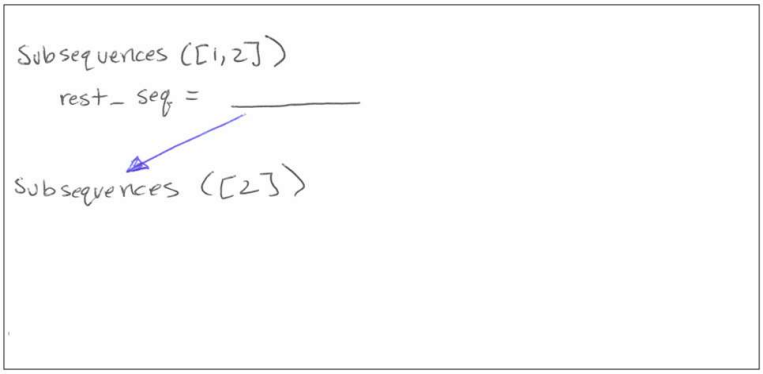
Let's consider how this function computes the result for `subsequences([1, 2])`.

Show/Hide Line Numbers

<< First Step | < Previous Step | Next Step > | Last Step >>

```
1 def subsequences(seq):
2   if not seq:
3     # base case:
4     return {()}
5   else:
```

```
6       # recursive case:
7       first = seq[0]
8       rest = seq[1:]
9       rest_seq = subsequences(rest)
10      first_seq = {
11        (first,) + sub_seq
12          for sub_seq in rest_seq
13      }
14      return first_seq | rest_seq
15
16  ans = subsequences([1, 2])
```

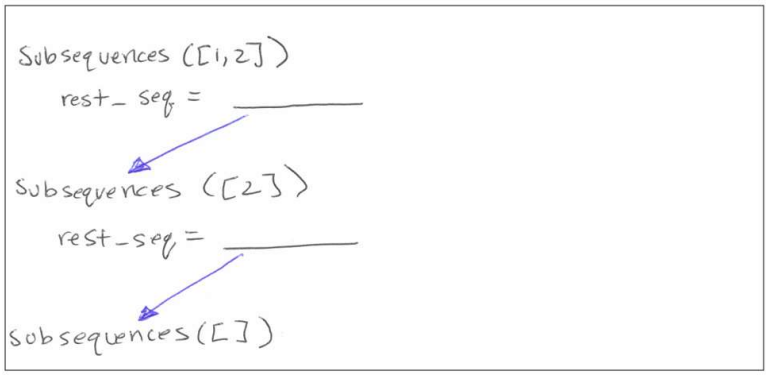


STEP 1

`subsequences([1, 2])`
Because seq is non-empty, it falls into the recursive `else` case. In lines 7 and 8, the sequence is split into:

```
first = 1
rest = [2]
```

Then in line 9, the recursive call on `subsequences(rest)` starts executing.



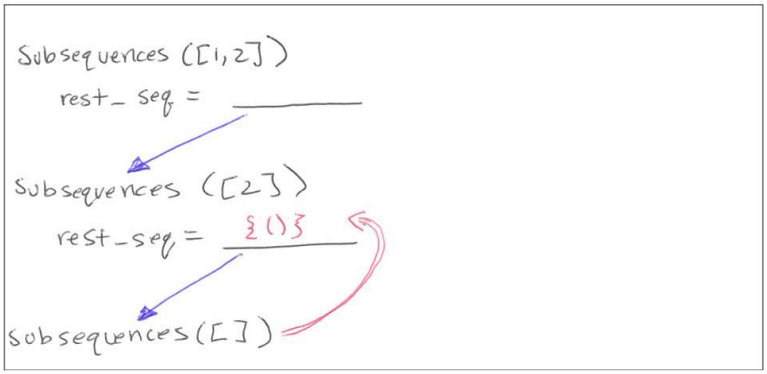
STEP 2

`subsequences([2])`
Because seq is non-empty, it falls into the recursive `else` case. In lines 7 and 8, the sequence is split into:

```
first = 2
rest = []
```

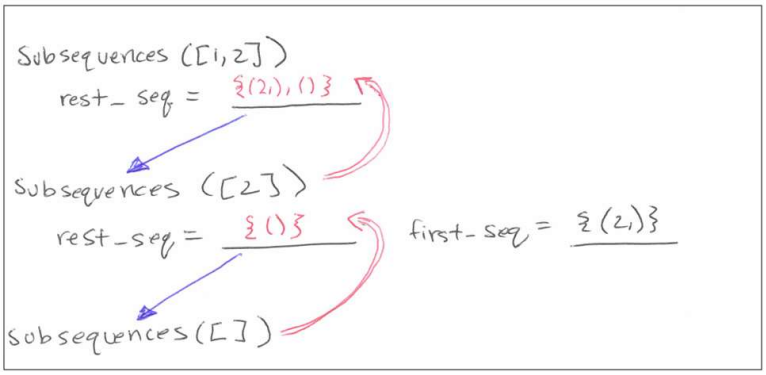
Note that slicing a list out of bounds like `seq[1:]` will return an empty list (but indexing a list like `seq[1]` would cause an `IndexError`.)

Then in line 9, the recursive call on `subsequences(rest)` starts executing.



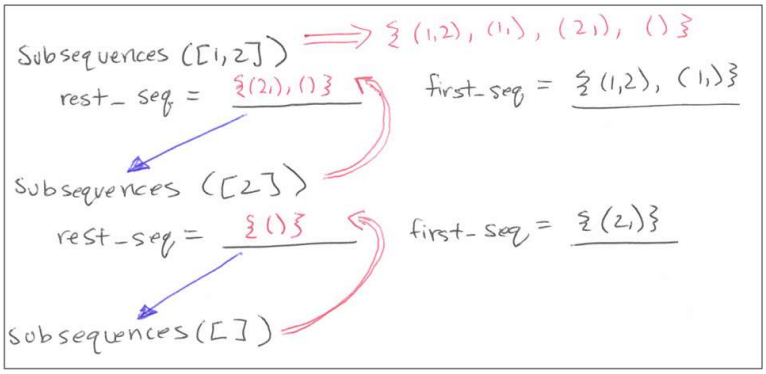
STEP 3

`subsequences([])`
Because seq is empty, it falls into the base case and returns the set containing an empty tuple to the `subsequences([2])` frame that made the recursive call.



STEP 4

`subsequences([2])`
Now that we know what `rest_seq` is, `first_seq` can be computed iteratively. Then in line 14, `first_seq` and `rest_seq` are combined into a new set `{(2,), ()}` which is returned to the original `subsequences([1, 2])` call.



STEP 5

`subsequences([1, 2])`
Now that we know what `rest_seq` is, `first_seq` can be computed iteratively by concatenating `(1,)` with the tuples from `rest_seq`. Then in line 14, `first_seq` and `rest_seq` are combined into a new set which is

returned as the final result `{(1, 2), (1,), (2,), ()}` and stored in `ans` in the global frame in line 16.

How many total calls to `subsequences` will be made in the course of evaluating `subsequences([4, 3, 2])`?

4

Submit

You have submitted this assignment 1 time.

Solution: 4

Explanation:

We end up making a linear number of calls to `subsequence` --- one for each element in the list plus one call with the empty list to reach the base case.

7) Choosing the Right Recursive Subproblem

Let's look at another example. Suppose we want to convert an integer to a string representation with a given base, following this spec:

```
def number_to_string(n, b):
    """
    Given an integer n and base b (where 2 <= b <= 10),
    returns n represented as a string in base-b notation,
    without any unnecessary leading zeroes.

    >>> number_to_string(-829, 10)
    "-829"
    >>> number_to_string(5, 2)
    "101"
    >>> number_to_string(0, 10)
    "0"
    """
```

Let's develop a recursive implementation of this function. One recursive case here is straightforward -- we can handle negative integers simply by recursively calling for the representation of the corresponding positive integer:

```
if n < 0:
    return "-" + number_to_string(-n, b)
```

This shows that the recursive subproblem can be smaller or simpler in more subtle ways than just the value of a numeric parameter or the size of a list parameter. We have still effectively reduced the problem by reducing it from all possible integers to just positive integers.

The next question is, given that we have a positive `n`, say `n = 829` in base 10, how should we decompose it into a recursive subproblem? Thinking about the number as we would write it down on paper, we could either start with 8 (the leftmost or highest-order

digit) or 9 (the rightmost or lowest-order digit). Starting at the left end seems natural, because that's the direction we write, but it's harder in this case because we would need to first find the number of digits in the number to figure out how to extract the leftmost digit. Instead, a better way to decompose `n` is to take its remainder modulo `b` (which gives the *rightmost* digit) and also divide by `b` (which gives the subproblem, the remaining higher-order digits):

```
digits = "0123456789"
return number_to_string(n // b, b) + digits[n % b]
```

In general, **think about several ways to break down the problem and try to write the recursive cases.** You want to find the one that produces the simplest, most natural recursive case.

It remains to figure out what the base case is and include an `if` statement that distinguishes the base case from this recursive case. Here is the function:

```
def number_to_string(n, b):
    """
    Given an integer n and base b (where 2 <= b <= 10),
    returns n represented as a string in base-b notation.

    >>> number_to_string(-829, 10)
    "-829"
    >>> number_to_string(5, 2)
    "101"
    >>> number_to_string(0, 10)
    "0"
    """
    digits = "0123456789"
    if n < 0:
        return "-" + number_to_string(-n, b)
    elif IS_BASE_CASE:
        BASE_CASE
    else:
        return number_to_string(n // b, b) + digits[n % b]
```

Which of the following can be substituted for the `IS_BASE_CASE` and `BASE_CASE` to make the code correct? It may help to think through the doctest cases shown in the docstring.

- [ ]
```
elif n == 0:
    return "0"
```

- [ ]
```
elif n == 0:
    return ""
```

- [x]
```
elif n < b:
    return str(n)
```

- [x]
```
elif n < b:
    return digits[n]
```

Submit

You have submitted this assignment 18 times.

Solution:

- [ ]
```
elif n == 0:
    return "0"
```

- [ ]
```
elif n == 0:
    return ""
```

- [x]

```
        elif n < b:
         return str(n)
```



```
        elif n < b:
        return digits[n]
```

Explanation:

The first choice puts a leading zero in front of every number -- it would make `number_to_string(-829, 10)` return `"-0829"`, failing that test case.

The second choice fixes the leading-zero problem but outputs the empty string for `number_to_string(0, 10)`, when we want `"0"`.

The third and fourth choices do the right thing -- once `n` fits in a single digit (in base `b`), it outputs that digit, which avoids the leading-zero problem and still produces `0` when that's what we want.

8) Summary

We've looked at several functions here and showed how to write them recursively, sometimes in different ways. Along the way, we've seen that:

- every recursive function needs one or more base cases and one or more recursive cases
- the recursive cases should make the problem smaller or simpler in some way

We've focused closely on recursion in this reading, aiming to write every function recursively, even when there was an *iterative* approach that might have been just as natural. Next week's reading will delve more deeply into the question of recursion vs. iteration, showing that they are interchangeable (every iterative algorithm can be made recursive, and vice versa) and discuss when it's more natural to use one or the other.