Recursion and Iteration

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

This reading examines recursion more closely by comparing and contrasting it with iteration. Both approaches create repeated patterns of computation. Recursion produces repeated computation by calling the same function recursively, on simpler or smaller subproblems. Iteration produces repeated computation using `for` loops or `while` loops.

If we can come up with an iterative version, do we need recursion at all? In one sense, no, we don't need recursion -- any function we can write recursively could also be written iteratively. But, some problems lend themselves *naturally* to a recursive solution. When we try to solve those kinds of problems iteratively instead, we find ourselves *simulating* recursion anyway, using an explicit agenda to keep track of what we need to do next. For those kinds of problems, it's more straightforward to express the recursive algorithm directly, letting Python handle the agenda bookkeeping using its call stack.

The converse is also true: any function we can write iteratively, with a loop, could also be written recursively. So, it's important to be able to think in *both* ways and choose the one that is most natural and appropriate to the problem we are trying to solve.

This reading looks at the essential equivalence between these approaches and some of their tradeoffs in simplicity and performance. We'll return to some of the functions we've written in previous readings, both recursive and iterative, and show how to write each using the respective other techniques.

2) Recursion vs. Iteration

Let's start with the simple example we used in the last reading: factorial. Mathematically, we could have used either of the following definitions of factorial:

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

or

$$n! = \prod_{i=1}^{n} i$$

The first definition is recursive, but the second is actually iterative -- it doesn't mention $n!$ on the right-hand side of the definition, and the right-hand side effectively describes a loop.

In terms of code, these definitions translate directly to either a recursive or iterative implementation:

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

```python
def factorial(n):
    out = 1
    for i in range(1, n+1):
        out *= i
    return out
```

Check Yourself:

Which of these implementations would you choose? What are the benefits and drawbacks of each approach?

Here are some points of comparison:

- The iterative version might be more efficient because it doesn't need to create new frames for the recursive calls.
- The recursive version might feel simpler, a better match for the mathematical definition of factorial.
- The two versions might behave differently with *illegal* inputs, like $n < 0$.

What does `factorial(-1)` do for the *iterative* version of `factorial` above?

○ the function returns the right value

○ the function returns an undesirable value

○ the function never returns

○ the function causes an error

What does `factorial(-1)` do for the *recursive* version of `factorial` above?

○ the function returns the right value

○ the function returns an undesirable value

○ the function never returns

○ the function causes an error

For either version, it would be better if the function produced an error right away when it gets an illegal input:

```python
def factorial(n):
    assert n >= 0, "need a nonnegative integer"
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

The result of `factorial(-1)` for the recursive version points at an important difference between recursion and iteration in many programming languages (including Python): recursion is limited by a maximum call-stack depth. In Python, this default limit is 1000 calls. When a function tries to call deeper than that, Python produces `RecursionError`. This limit can be increased using `sys.setrecursionlimit()`, but it can't be made unbounded, because the operating system itself puts a limit on Python's call stack.

What does `factorial(1001)` do for the *iterative* version of `factorial` above?

○ the function returns the right value

○ the function returns an undesirable value

○ the function never returns

○ the function causes an error

What does `factorial(1001)` do for the *recursive* version of `factorial` above (assuming default Python settings)?

○ the function returns the right value

○ the function returns an undesirable value

○ the function never returns

○ the function causes an error

3) List-Like Patterns

The rest of this reading is structured by looking at several common kinds of repeated computation, depending on the shape of the data it is processing: list-like (i.e., a sequence of values), tree-like, and graph-like. We will reach back to previous readings for a recursive or iterative algorithm that we have previously looked at for each of these forms, discussing how we might write the algorithm the other way.

Let's start with lists. Recall `sum_list` from the Recursion reading, which we wrote both iteratively and recursively:

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

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

The iterative version naturally uses a `for` loop.

The recursive version decomposes the list into its first element `x[0]` and the rest of the list `x[1:]`. The rest-of-list part is the smaller recursive subproblem. This **first/rest** decomposition is a very common way to deal with list-like data recursively.

Check Yourself:

What other recursive decompositions might be natural or plausible for list-like data?

And thinking back to what you learned in 6.100A/L:

- what decomposition does **binary search** (also known as bisection search) use?
- what decomposition does **merge sort** use?

Show/Hide

We might also imagine a decomposition that takes out the last element `x[-1]` and then recurses on the prefix of the list before it `x[:-1]`. This might be the right decomposition for a linear search from the end of the list, for example.

Binary search examines the element in the middle of a sorted list, say `m = len(x)//2`, and then recurses on either the left part `x[:m]` or the right part `x[m+1:]`, depending on whether the target element is less than or greater than `x[m]`. This decomposition might be called middle/left/right.

Merge sort sorts a list by first dividing it into a left half and a right half, recursively sorting each half, and then recombining the sorted sublists by merging them together.

Notice that, in each of these decompositions, the algorithm must be careful to make the recursive subproblems smaller. The first/rest and middle/left/right decompositions do it by peeling off one element of the list (the first or the middle) and excluding it from the recursive subproblems, so that they are definitely smaller than the original problem. This is how these decompositions guarantee that the subproblems are smaller than the original problem, so we will eventually terminate at the base case (which is often, but not always, the empty list).

Here is a version of `sum_list` that uses a left/right recursive decomposition:

```python
def sum_list(x):
    if not x:
        return 0
    else:
        m = len(x) // 2
        return sum_list(x[0:m]) + sum_list(x[m:])
```

What does `sum_list([1,2,3])` do for this version of `sum_list`?

○ the function returns the right value

○ the function returns an undesirable value

○ the function never returns

○ the function causes an error

3.1) Accumulating Results Using a Helper Function

One difference between the recursive and iterative versions of `sum_list` is the order in which they do the summation. Although both versions consider the list elements in order from beginning to end, the recursive version defers any addition until it has reached the end of the list, and then it does the sum backwards. So the computation ends up looking like this:

- iterative version: start with `0`, then add `x[0]`, then `x[1]`, then `x[2]`, ..., finally add `x[n-1]`
- recursive version: start with `0`, then add `x[n-1]`, then `x[n-2]`, then `x[n-3]`, ..., finally add `x[0]`

The recursive version is saving up the additions for its recombination step, when it combines the results of the recursive subproblems.

But, we don't have to write it this way. We can write a recursive version that does the addition as it goes, by defining a **recursive helper function** to which we pass the partial sum computed so far:

```python
def sum_list(x):
    def sum_helper(sum_so_far, lst):
        if not lst:
            return sum_so_far
        else:
            num = lst[0]
            rest = lst[1:]
            return sum_helper(sum_so_far + num, rest)
    return sum_helper(0, x)
```

The `sum_helper` function is a recursive helper function. It needs to be a different function from the original `sum_list` because it has a new parameter, `sum_so_far`, that keeps track of the partial sum computed so far. The recursive calls steadily add to `sum_so_far`, until finally reaching the end of the list (the base case), at which point the completed sum is returned as the only result. The body of `sum_list` gets the ball rolling by calling `sum_helper` with an initial value of `0` for `sum_so_far` (which was the value returned by the base case in the previous recursive version).

Hiding the recursive helper function as an internal function like this is a good idea because it's only needed inside `sum_list`. (It can also help for sharing parameters or local variables that don't need to change during the recursion, since all the calls to the recursive helper will automatically have access to variables in the enclosing function's frame, but we don't have any need for that in this case.)

How deep is the recursion for `sum_list([1,2,3,4])` --- specifically, how many calls to `sum_helper` are made?

This example demonstrates a general principle: an iterative loop can be transformed into a recursive helper function, by using parameters to carry the local variables that the iterative loop updates on each iteration.

In the loop in `sum_list`, the relevant local variable is `sum_so_far`, and here is how the loop is transformed into the recursive version shown above:

```
def sum_list(x):
    sum_so_far = 0          # becomes initial call to sum_helper(0, x)
    for num in x:           # becomes num = lst[0], rest = lst[1:]
        sum_so_far += num   # becomes recursion step
sum_helper(sum_so_far + num, rest)
    return sum_so_far       # becomes the base case
```

One trick that makes this pattern even shorter: we can eliminate the helper function `sum_helper` and instead recurse on `sum_list` itself, by giving the additional parameters default initial values. That way, the original call initializes the recursion, and subsequent recursive calls update the parameters:

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

3.2) Optimizations

List-like recursion in Python suffers from a couple of performance drawbacks:

1. Every recursive call creates a new frame, and the recursion depth is limited (to 1000 calls by default). For first/rest decompositions, this means that the length of the list that can be processed is similarly limited. (But, for other decompositions, like left-half/right-half, it's not a limitation at all. Why not? What is the maximum-length list that could be processed with successive halving if you are limited to 1000 recursive calls?)

2. Every recursive call in a first/rest decomposition needs to copy the rest of the list, using a slice like `lst[1:]`. Over the course of the entire computation, all this copying adds up to time proportional to the square of the length of the list. (Why? If the original list has length n, then the first call copies a slice of length $n-1$, the next recursive call copies a slice of length $n-2$, and so on until reaching the end of the list. Since $(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}$, this means $O(n^2)$ time is spent copying.)

Both of these issues can be fixed by clever optimizations:

1. The recursion-depth limit can be fixed by *tail-call optimization*. If the recursion is written so that the recursive call is the last thing done in the body of the function -- like the line `return sum_helper(...)` in the recursive version of `sum_list` above -- then this recursive call is called a *tail call*, coming as it does at the tail end of the work the function has to do. Tail-call optimization means that, when the runtime system encounters a tail call, it deduces that it will no longer need the frame for the current call and can simply *reuse* it for the new recursive call, rather than creating a new frame. With tail-call optimization, every recursive call to `sum_helper` simply reuses the same frame, the recursion depth never exceeds 1, and the performance of the recursive version is essentially like a loop.

   Tail-call optimization can't be applied to a recursive call that isn't at the very end of the function. If `sum_list` were written as we originally had it, with `return x[0] + sum_list(x[1:])`, then this is not a tail call, because the function still needs to do some more work (adding `x[0]`) after the recursive call comes back. Tail-call optimization is also blocked if the frame needs to be kept for a function object that was created during the call.

   Python unfortunately does not implement tail-call optimization, but other languages do.

2. The list-copying problem can be addressed by implementing the list with a linked list (which we will study in detail in a later week) rather than an array, so that the "rest" of a list can be obtained in constant time. Python doesn't do that, so another way to avoid list copying in Python is to use an index to represent the rest of the list, here called `i`:

```python
def sum_list(x, i=0, sum_so_far=0):
    if i >= len(x):
        return sum_so_far
    else:
        return sum_list(x, i+1, sum_so_far + x[i])
```

4) Tree-like Patterns

Having examined sequential, list-like data, let's turn now to tree-shaped data, which might go to arbitrary depth. For example, recall `sum_nested` from the Recursion reading:

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

Why do we think of the input to `sum_nested` as tree-shaped? Because each sublist in `x` is like an internal node of a tree, with children that might be further sublists, until we reach simple numbers, which are the leaves.

Tree-shaped data have a straightforward recursive decomposition that mirrors the tree structure: we make recursive calls to all children until we reach the leaves.

But, notice that `sum_nested` as written here is not only decomposing the tree structure recursively but also decomposing the list of children recursively.

When considering `x` like a tree, which of the recursive calls in `sum_nested` goes one level deeper in the tree?

○ `sum_nested(x[0])`

○ `sum_nested(x[1:])`

When considering `x` as a list of children, which of the calls in `sum_nested` are called on the *rest* of a first/rest decomposition of `x`?

○ `sum_nested(x[0])`

○ `sum_nested(x[1:])`

To convert this recursive function into an iterative loop, we will reach back to an idea that we used in earlier readings: an *agenda*. Here, our agenda will keep track of recursive calls that we haven't made yet. Specifically, each item on the agenda will consist of the parameters for a recursive call. The agenda is initialized with the original list (which we will rename to `original_x` so that we can keep using `x` for the rest of the code without confusion), and then, every time we encounter a recursive call to `sum_nested()`, we put it on the agenda instead, so that it will be handled in a later iteration of the loop. Here is the code:

```
def sum_nested(original_x):
    sum_so_far = 0
    agenda = [original_x] # agenda consists of parameters for pending recursive calls
    while agenda:
      x = agenda.pop(-1)
      if not x:
          sum_so_far += 0
      elif isinstance(x[0], list):
          agenda.append(x[0])
          agenda.append(x[1:])
      else:
          sum_so_far += x[0]
          agenda.append(x[1:])
    return sum_so_far
```

We opted in this iterative implementation to add and remove items at the end of the agenda, so that it explores the tree in the same order that the recursive algorithm does -- depth-first. But, we didn't necessarily have to make that choice -- how could we change the code so that the iterative version is traversing the tree breadth-first instead? Either way, it doesn't matter for `sum_nested`, since the numbers can be added in any order, but, for other problems, the order might matter.

Looking at the transformation we just did from the other point of view, there is an agenda data structure hiding inside the recursive version -- the agenda is just the call stack, managed by Python, and it is keeping track of the remaining work that needs to be done in the recursive process. So, recursion gives us a convenient and simple way to express a depth-first traversal without having to manage our own agenda data structure.

5) Graph-like Patterns

Now that we have explored the equivalence between recursive traversal and iterative (depth-first) traversal using an explicit agenda, let's go back to graph-traversal problems we explored in earlier readings. "Graph-like" data can be regarded as vertices connected by edges, but (unlike a tree) a graph may have multiple paths reaching the same vertex, or it may have cycles (paths that return to the same vertex again).

In previous readings, we solved graph-traversal problems using iteration, like this flood-fill function:

```python
def flood_fill(image, starting_location, new_color):
    # recall that "*location" is equivalent to "location[0],location[1]"
    original_color = get_pixel(image, *starting_location)

    agenda = [starting_location]  # agenda: all of the cells we need to color in
    visited = {starting_location}  # visited set: all pixels ever added to the agenda
    while agenda:
        location = agenda.pop(0)
        set_pixel(image, *location, new_color)
        for neighbor in get_neighbors(location):
            if (neighbor not in visited
                    and get_pixel(image, *neighbor) == original_color):
                visited.add(neighbor)
                agenda.append(neighbor)
```

Now that we know that we can use the recursive call stack as an agenda, how can we write this recursively? Everywhere we would put a work item on the agenda, let's make a call to a recursive helper function instead, using the work item as its parameter. This recursive helper function replaces the `while` loop:

```python
def flood_fill(image, starting_location, new_color):
    original_color = get_pixel(image, *starting_location)
    visited = {starting_location}  # visited set: all pixels ever added to the agenda
    def fill_from(location):
        set_pixel(image, *location, new_color)
        for neighbor in get_neighbors(location):
            if (neighbor not in visited
                    and get_pixel(image, *neighbor) == original_color):
                visited.add(neighbor)
                fill_from(neighbor)
    fill_from(starting_location)
```

Check Yourself:

Let's reflect on how we're using recursion here and think carefully about whether the recursion would actually stop.

What makes the recursive call `fill_from(neighbor)` a *smaller or simpler* subproblem?

And what is the base case of the recursion?

Show/Hide

Before we make the recursive call to `fill_from(neighbor)`, we first add `neighbor` to the `visited` set, so that we won't visit that pixel again. The graph of pixels we're trying to fill has effectively become one pixel smaller.

The base case happens when we have no recursive calls to make -- when we reach a pixel which is a dead end, because all its neighbor pixels are either not `original_color` or already `visited`.

Check Yourself:

The observation we just made leads to a further observation. For flood fill, we have another guarantee that the recursive subproblem is smaller. What is it?

Show/Hide

The fact that every recursive call changes a pixel's color from `original_color` to `new_color` *also* makes the problem smaller. So, we actually don't need a visited set when we solve flood fill recursively!

```
def flood_fill(image, starting_location, new_color):
    original_color = get_pixel(image, *starting_location)
    if original_color == new_color:
      return # nothing to do
    def fill_from(location):
        set_pixel(image, *location, new_color)
        for neighbor in get_neighbors(location):
            if get_pixel(image, *neighbor) ==
original_color:
                fill_from(neighbor)
    fill_from(starting_location)
```

Suppose we wrote a recursive graph-traversal algorithm (not flood fill) that recursively visits the neighbors of a vertex, but we neglect to use a visited set to remember which vertices we already visited. What could happen if the graph has cycles?

○ the recursive function runs forever without stopping

○ the recursive function raises an error

6) When To Use Recursion and When To Use Iteration

How might you decide to use recursion?

- When the problem is easy to express recursively (as factorial)
- When the data itself is defined recursively (like a tree often is)
- When depth-first traversal is fine

- When the recursion won't get too deep (this constraint may be relaxed for programming languages other than Python)

Recursive solutions are often shorter and simpler, by moving agenda bookkeeping into the programming language rather than representing it explicitly.

As you gain comfort with recursion, you may find that a recursive solution is easier to express at first, because of natural decompositions (like first/rest, or left-half/right-half, or recursing into children or graph neighbors). So, it may be good to *start* with a recursive approach and then convert it to an iterative algorithm (using one of the strategies here) if you run into problems like recursion depth limits or excessive copying.

7) Common Mistakes in Recursion

Here are three common ways that recursion can go wrong:

- The base case may be missing entirely, or the problem needs more than one base case, but not all the base cases are covered.
- The recursive step doesn't reduce to a smaller subproblem, so the recursion doesn't converge.
- Aliases to a mutable data structure are inadvertently shared, and mutated, among the recursive calls.

We saw the first two issues in an exercise above, when we tried to convert `sum_list` to use a left/right decomposition. When the subdivided list reached length 1, the decomposition was no longer able to make two smaller subproblems, so we needed to make that into a new base case.

Aliases to mutable data are sometimes *necessary* in recursion, as we saw above when we shared the `visited` set across all calls in a recursive traversal of a graph. For flood fill, of course, sharing the mutable `image` is necessary because the whole point of the recursion is to fill in more pixels. But, when you are doing a recursive depth-first search without a `visited` set (which, as we saw with flood fill, is often possible when a move in the search automatically makes the problem smaller or simpler), it can be tempting to keep the search state mutable. This becomes tricky, because then multiple pending recursive calls have aliases to the same mutable state. We will see an example of this in the next reading, when we use recursive search to solve Sudoku puzzles.

Look for these mistakes when you're debugging. A `RecursionError` is a sign of the first two. The third bug can be harder to spot, because it's fundamentally an aliasing bug.

8) Generators

In the last part of this reading, we'll introduce a really useful feature of Python called a *generator*. Like ordinary functions, generators can be recursive or iterative. Generators will be very important for a future lab, so we will start discussing them now in order to give time to practice using them in class.

So far in 6.101, we've talked a lot about functions, and we know by now what happens when we call a function: Python sets up a new frame to keep track of local variables for the function; and, when the function finishes, we get rid of those local variables and return a value to whomever it was that called the function.

Generators are a special kind of function that work in a similar way, except that they give us a way to *pause* their execution such that they produce a value, but we keep their local variables around and have the ability to resume their execution if and when we want to.

Before we get too much further, let's look at an example of a regular function, and we'll see how using generators can help us out. Here's the first example we'll look at today, a basic reimplementation of Python's `range` function:

```
# this is just a regular function, NOT a generator
def list_range(start, stop, step=1):
    assert step >= 1
    out = []
    current = start
    while current < stop:
        out.append(current)
        current += step
    return out
```

Perhaps surprisingly, when we try to run the following code (which makes use of this function), we don't see values printed to the screen immediately; rather, the program seems to hang:

```
for i in list_range(0, 1_000_000_000_000_000, 1):
    if i > 100:
        break
    print(i**2)
```

Check Yourself:

Why do we not see numbers printed to the screen right away? What is going on?

Importantly, **there is no infinite loop in this code**. But, why does our little program appear to do nothing?

Well, it turns out that it *is* doing something; when we first start running this program, Python tried to evaluate `list_range(0, 1_000_000_000_000_000, 1)`, in order to build a list that we can then loop over. This process takes a *whole bunch* of time and consumes a *whole bunch* of memory in order to create that list, and only after we're done making the list will Python actually be able to loop over it and start printing values. In fact, this process is going to take enough time and memory that it is not worth waiting for it to try to finish.

So here, we're wasting a lot of time to build that list, and we're also wasting a whole bunch of memory making a list containing *lots* of numbers, just so that we can print one after the other (and only for the first 100!). This function is a prime candidate for replacement by a generator; it would be great if, rather than building the whole list first, we could ask for the next number in our range, then pause the execution of the function until we're ready for another value.

It turns out that we can do this by writing *almost* exactly the same code, but with a few tweaks:

```
# this is a GENERATOR!
def gen_range(start, stop, step=1):
    assert step >= 1
    current = start
    while current < stop:
        yield current
        current += step
```

Let's note that the core logic here is precisely the same as our function above. We're looping until some condition is satisfied, adding `step` to a counter each time through the loop, etc. But, rather than creating an empty list, adding elements of interest to that list, and then returning that list at the end, we use a special keyword called `yield` on the elements of interest.

Whenever Python sees `yield` as part of the body of a function definition, Python will create a generator object when we call that function, rather than running our normal function-evaluation process. We can try this out to discover some things:

```
>>> g = gen_range(0, 1_000_000_000_000_000, 1)  # this resolves _right
away_, no wait
>>> g  # g is now a generator!
<generator object gen_range at 0x7f06be1a95f0>
```

Importantly, when we run the above, Python has still set up a new local frame for this generator, but it has not yet run *any* of the code in the body of the generator. There are multiple ways to get elements out of a generator, but the first we'll look at is the built-in `next` function. Calling `next` with a generator passed in will cause Python to start running the body of the code, run until we see a `yield`, pause the execution again, and give us the value that was yielded from the generator. As an example:

```
>>> next(g)
0
```

If we then call `next` again on the same generator, execution picks up where we had left off, and we again run the generator until we hit the next `yield` statement, at which point `current` is equal to `1`, so we get a `1` out:

```
>>> next(g)
1
```

`next` is a good way to get individual elements from a generator, but generators really exist to be looped over. So we can also write code like the following:

```
for i in gen_range(0, 1_000_000_000_000_000, 1):  # loop over the
generator
    if i > 100:
        break
    print(i**2)
```

Check Yourself:

Try running this code -- what happens?

In that code, whenever we reach the top of the loop, we need a new value for `i`; in order to get that value, Python will run our generator until we hit a `yield` statement and then set `i` to be the value that was yielded, before entering the body of the loop.

One important thing to note is that the code above started to print values *right away* because, in contrast to our list example from earlier, we didn't need to gather all of the elements together before starting in on our `for` loop. And, perhaps more importantly, we didn't need to waste memory building a list containing all of these numbers before considering any of them; we consider one of them at a time instead, saving ourselves a lot of memory overhead.

8.1) Another Example

The generator examples we've seen so far have been iterative, in the sense that the `yield` statement is inside a `while` loop. Let's consider an example where the generator is recursive instead.

Consider the following code, which finds the set of words that can be made by combining a subset of letters from some starting word in an arbitrary order:

```
def subwords(word):
    """
    Return all valid words that can be made from a subset of the letters
in the
    original given word (regardless of order).

    >>> result = subwords('cat')
    >>> expected = {'cat', 'act', 'at'}
    >>> result == expected
    True
    """
    return {w for w in subwords_helper(word) if w in ALL_WORDS}


def subwords_helper(word):
    """
    Helper function, returns all possible permutations of subsequences of
x,
    regardless of whether they are real words or not.

    >>> result = subwords_helper('cat')
    >>> expected = {'', 'c', 'a', 't',
    ...             'ca', 'ac', 'ct', 'tc', 'ta', 'at',
    ...             'cat', 'cta', 'tac', 'tca', 'act', 'atc'}
    >>> result == expected
    True
    """
    if not word:
        return {''}

    first = word[0]
    rest = word[1:]

    out = set()
    for w in subwords_helper(rest):
        out.add(w)
        for ix in range(len(w)+1):
            out.add(w[:ix] + first + w[ix:])
    return out
```

Notice that `subwords_helper` is a recursive helper function, using a list-like decomposition. (In this case, we chose to define `subwords_helper` at the top level, outside of `subwords`, because it has doctests. Using doctests for a function requires it to be defined at the top level.)

This code works nicely, but it takes a really long time for some inputs. For example, calling `subwords('artichokes')` likely takes more than 10 seconds on most machines.

And we can think about why this takes so much time, too! Well, one reason is just that this problem is hard and takes a lot of time. But, we're also spending a lot of time (and memory!) building up a structure containing all of the possible combinations of those letters (9,864,100 possibilities in the case of `'artichokes'`), which we then filter down (to 517 results in the case of `'artichokes'`)! So, perhaps, we could do better by not storing all of those possibilities in a big set, but rather using a generator for that part, so that we can consider the possible matches one-at-a-time instead.

You may wish to try writing this code for yourself as a little self-exercise, but our implementation is given below:

Show/Hide

Note how similar the structure is to the code from above; only a few places differ (we don't accumulate possibilities in a set and return it in the end; rather, we `yield` possibilities as we notice them).

Note also, though, that if we run the generator to completion, the values that we ultimately yield are exactly the same as the elements that we would have added to a set before returning it in our original version.

```
def subwords(word):
    """
    Return all valid words that can be made from a subset of the
letters in the
    original given word (regardless of order).

    >>> result = subwords('cat')
    >>> expected = {'cat', 'act', 'at'}
    >>> result == expected
    True
    """
    return {w for w in subwords_helper(word) if w in ALL_WORDS}


# GENERATOR VERSION!
def subwords_helper(word):
    """
    Helper function, returns all possible permutations of
subsequences of x,
    regardless of whether they are real words or not.

    >>> result = set(subwords_helper('cat'))
    >>> expected = {'', 'c', 'a', 't',
    ...              'ca', 'ac', 'ct', 'tc', 'ta', 'at',
    ...              'cat', 'cta', 'tac', 'tca', 'act', 'atc'}
    >>> result == expected
    True
    """
    if not word:
        yield ''
        return  # return here means stop the generator entirely
(no more values to yield)

    first = word[0]
    rest = word[1:]

    for w in subwords_helper(rest):  # recursive call to a
generator!
        yield w

        for ix in range(len(w)+1):
            yield w[:ix] + first + w[ix:]
```

(If you want to try this code yourself, you can get the `ALL_WORDS` set from the word-ladder example in the Graph Search reading.)

This new version takes about half as much time to run `subwords('artichokes')`! Importantly, generators are not *always* faster than using other structures like lists or sets; we need to be careful when deciding whether to use them or not, but we've just seen a couple of examples of cases where they can make a big difference!

8.2) The `yield from` Keyword

Often, we want a generator to take elements from another iterable object and yield them one after the other, something like the following:

```
def gen1():
    yield 1
    yield 2

def gen2():
    yield 9
    yield 8
```

```python
def gen3():
    for val in gen1():
        yield val

    for val in gen2():
        yield val
```

We can accomplish the same thing by using the `yield from` keyword, which yields each element from any iterable object in order:

```python
def gen3():
    yield from gen1()
    yield from gen2()
```

8.3) Questions

The following code box contains a function that takes a list of numbers as input and returns a list containing the negatives of those values. Rewrite the code so that `negate_elements` is a generator that yields those values instead. You **should not** make a list containing all of the elements at any point in your code.

```python
def negate_elements(x):
    out = []
    for val in x:
        out.append(-val)
    return out
```

Consider the following generator:

```
import math

def sine(period):
    n = 0
    while True:
        yield round(math.sin(n * 2 * math.pi / period), 6)
        n = (n + 1) % period
```

How many elements will the generator `sine(5)` produce if we loop over it?

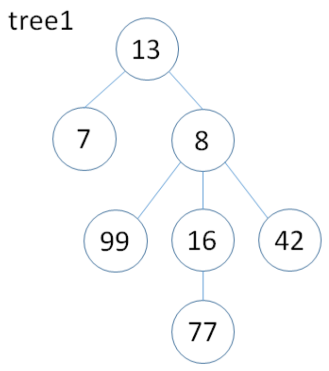
--

Write a generator that yields all of the elements from a given tree, in an arbitrary order.

A node in the tree is represented as a list with the first element being the node value, and the rest of the list being the node's children (each of which is also a tree). That is to say, our tree structure is a nested list structure.

For example, the following code (`tree1`) and picture represent the same tree.

```
tree1 = [13,
          [7],
          [8,
            [99],
            [16,
              [77]],
            [42]]]
```



Your generator should take in a tree in this format, and it should yield each of the node values in an abitrary order.

```python
def all_values(tree):
    pass  # your code here

```

9) Summary

We have explored several kinds of problems, based on the shape of the data or repeated computation that needs to be done (list-like, tree-like, graph-like), and have seen how to do the repeated computation both recursively and iteratively.

Along the way, we've seen more strategies for thinking recursively, including:

- recursive decompositions that are commonly used for list-like data (first/rest, left-half/right-half), tree-like data (recursing into children), and graph-like data (recursing to neighbors);

- using a recursive helper function with parameters that accumulate partial results;
- using recursion for a depth-first traversal of a tree or graph.

We've also explored the idea of a generator, a kind of function that produces a sequence of values without having to hold the entire sequence in memory. A generator pauses its computation whenever a value is produced (so that the caller can use it) and then resumes its computation when it's time to get the next value.