

1 Recursion

A *recursive* function is a function that is defined in terms of itself. A good example is the `factorial` function. Although we haven't finished defining `factorial`, we are still able to call it since the function body is not evaluated until the function is called. Note that when `n` is `0` or `1`, we just return `1`. This is known as the *base case*, and it prevents the function from infinitely recursing. Now we can compute `factorial(2)` in terms of `factorial(1)`, and `factorial(3)` in terms of `factorial(2)`, and `factorial(4)` – well, you get the idea.

There are **three** common steps in a recursive definition:

1. **Figure out your base case:** The base case is usually the simplest input possible to the function. For example, `factorial(0)` is `1` by definition. You can also think of a base case as a stopping condition for the recursion. If you can't figure this out right away, move on to the recursive case and try to figure out the point at which we can't reduce the problem any further.
2. **Make a recursive call with a simpler argument:** Simplify your problem, and assume that a recursive call for this new problem will simply work. This is called the “leap of faith”. For `factorial`, we reduce the problem by calling `factorial(n-1)`.
3. **Use your recursive call to solve the full problem:** Remember that we are assuming the recursive call works. With the result of the recursive call, how can you solve the original problem you were asked? For `factorial`, we just multiply $(n - 1)!$ by n .

Another way to understand recursion is by separating out two things: “internal correctness” and not running forever (known as “halting”).

A recursive function is internally correct if it always does the right thing assuming that every recursive call does the right thing. For example, the fibonacci function reproduced to the right is internally correct, since $F_0 = 0$ and $F_n = F_{n-1} + F_{n-2}$ are both true statements.

The bad fibonacci function to the right does not halt on all inputs, however, since `fib(1)` results in a call to `fib(-1)`, and then to `fib(-2)` and so on.

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

```
def fib(n): # WRONG!  
    if n == 0:  
        return 0  
    return fib(n - 1) + fib(n - 2)
```

A recursive function is correct if and only if it is both internally correct and halts; but you can check each property separately. The “recursive leap of faith” is temporarily placing yourself in a mindset where you only check internal correctness.

Questions

- 1.1 Write a function that takes two numbers `m` and `n` and returns their product. Assume `m` and `n` are positive integers. **Use recursion**, not `mul` or `*`!

Hint: $5 \times 3 = 5 + 5 \times 2 = 5 + 5 + 5 \times 1$.

For the base case, what is the simplest possible input for `multiply`?

If one of the inputs is one, you simply return the other input.

For the recursive case, what does calling `multiply(m - 1, n)` do? What does calling `multiply(m, n - 1)` do? Do we prefer one over the other?

The first call will calculate a value that is `n` less than the total, while the second will calculate a value that is `m` less.

Either recursive call will work, but only `multiply(m, n - 1)` is used in this solution.

```
def multiply(m, n):
```

```
    """
```

```
>>> multiply(5, 3)
```

```
15
```

```
    """
```

```
    if n == 1:
```

```
        return m
```

```
    else:
```

```
        return m + multiply(m, n - 1)
```

Video walkthrough

- 1.2 Below is the iterative version of `is_prime`, which returns `True` if positive integer `n` is a prime number and `False` otherwise:

```
def is_prime(n):
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

Implement the recursive `is_prime` function. Do not use a while loop, use recursion. As a reminder, an integer is considered prime if it has exactly two unique factors: 1 and itself.

```
def is_prime(n):
    """
    >>> is_prime(7)
    True
    >>> is_prime(10)
    False
    >>> is_prime(1)
    False
    """
```

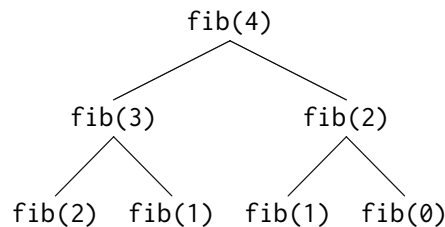
```
def prime_helper(index):
    if index == n:
        return True
    elif n % index == 0 or n == 1:
        return False
    else:
        return prime_helper(index + 1)
return prime_helper(2)
```

2 Tree Recursion

Consider a function that requires more than one recursive call. A simple example is the recursive `fibonacci` function:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

This type of recursion is called **tree recursion**, because it makes more than one recursive call in its recursive case. If we draw out the recursive calls, we see the recursive calls in the shape of an upside-down tree:



We could, in theory, use loops to write the same procedure. However, problems that are naturally solved using tree recursive procedures are generally difficult to write iteratively. It is sometimes the case that a tree recursive problem also involves iteration: for example, you might use a while loop to add together multiple recursive calls.

As a general rule of thumb, whenever you need to try multiple possibilities at the same time, you should consider using tree recursion.

How to diagram Tree Recursion

Questions

- 2.1 You want to go up a flight of stairs that has n steps. You can either take 1 or 2 steps each time. How many different ways can you go up this flight of stairs? Write a function `count_stair_ways` that solves this problem. Assume n is positive.

When there is only 1 step, there is only one way to go up the stair. When there are two steps, we can go up in two ways: take a two-step, or take 2 one-steps.

Before we start, what's the base case for this question? What is the simplest input?

Our first base case is where there are no steps left. This means that we took an action in the previous recursive step that led to our goal of reaching the top. Our second base case is where we have overstepped. This means that the action we took is not valid, as it caused us to step over our goal.

What do `count_stair_ways(n - 1)` and `count_stair_ways(n - 2)` represent?

`count_stair_ways(n - 1)` represents the number of different ways to go up the last $n - 1$ stairs (this is the case where we take 1 step as our move). `count_stair_ways(n - 2)` represents the number of different ways to go up the last $n - 2$ stairs (this is the case where we take 2 steps as our move). Our base cases will take care of if there are no steps left or if we overstepped.

Fill in the code for `count_stair_ways`:

```
def count_stair_ways(n):

    if n == 0:
        return 1
    elif n < 0:
        return 0
    return count_stair_ways(n-1) + count_stair_ways(n-2)
```

[Video walkthrough \(Leap of Faith\)](#)

[Video Walkthrough \(Diagramming Trees\)](#)

- 2.2 Consider a special version of the `count_stairways` problem, where instead of taking 1 or 2 steps, we are able to take **up to and including** `k` steps at a time.

Write a function `count_k` that figures out the number of paths for this scenario. Assume `n` and `k` are positive.

```
def count_k(n, k):
    """
    >>> count_k(3, 3) # 3, 2 + 1, 1 + 2, 1 + 1 + 1
    4
    >>> count_k(4, 4)
    8
    >>> count_k(10, 3)
    274
    >>> count_k(300, 1) # Only one step at a time
    1
    """

    if n == 0:
        return 1
    elif n < 0:
        return 0
    else:
        total = 0
        i = 1
        while i <= k:
            total += count_k(n - i, k)
            i += 1
        return total
```

[Video Walkthrough](#)

3 Lists

A *sequence* is an ordered collection of values. It has two fundamental properties: length and element selection. In this discussion, we'll explore one of Python's data types, the *list*, which implements this abstraction.

In Python, we can have lists of whatever values we want, be it numbers, strings, functions, or even other lists! Furthermore, the types of the list's contents need not be the same. In other words, the list need not be homogeneous.

Lists can be created using square braces. Their elements can be accessed (or *indexed*) with square braces. Lists are zero-indexed: to access the first element, we must index at 0; to access the i th element, we must index at $i - 1$.

We can also index with negative numbers. These begin indexing at the end of the list, so the index -1 is equivalent to the index `len(list) - 1` and index -2 is the same as `len(list) - 2`.

Let's try out some indexing:

```
>>> fantasy_team = ['aaron rodgers', 'desean jackson']
>>> print(fantasy_team)
['aaron rodgers', 'desean jackson']
>>> fantasy_team[0]
'aaron rodgers'
>>> fantasy_team[len(fantasy_team) - 1]
'desean jackson'
>>> fantasy_team[-1]
'desean jackson'
```

List Slicing

If we want to access more than one element of a list at a time, we can use a *slice*. Slicing a sequence is very similar to indexing. We specify a starting index and an ending index, separated by a colon. Python creates a new list with the elements from the starting index up to (but not including) the ending index.

We can also specify a step size, which tells Python how to collect values for us. For example, if we set step size to 2, the returned list will include every **other** value, from the starting index until the ending index. A negative step size indicates that we are stepping backwards through a list when collecting values.

You can also choose not to specify any/all of the slice arguments. Python will perform some default behaviour if this is the case:

- If the step size is left out, the default step size is 1.
- If the start index is left out, the default start index is the beginning of the list.
- If the end index is left out, the default end index is the end of the list.
- If the step size is negative, the default start index becomes the end of the list, and the default end index becomes the beginning of the list.

Thus, `lst[:]` creates a list that is identical to `lst` (a copy of `lst`). `lst[::-1]` creates a list that has the same elements of `lst`, but reversed. Those rules still apply if more than just the step size is specified e.g. `lst[3::-1]`.

```
>>> directors = ['jenkins', 'spielberg', 'bigelow', 'kubrick']
>>> directors[:2]
['jenkins', 'spielberg']
>>> directors[1:3]
['spielberg', 'bigelow']
>>> directors[1:]
['spielberg', 'bigelow', 'kubrick']
>>> directors[0:4:2]
['jenkins', 'bigelow']
>>> directors[::-1]
['kubrick', 'bigelow', 'spielberg', 'jenkins']
```

List Comprehensions

A **list comprehension** is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

It might be helpful to note that we can rewrite a list comprehension as an equivalent for statement. See the example to the right.

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x % 2 == 1` (1, 3, and 5). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

Note: The `if` clause in a list comprehension is optional.

Questions

3.1 What would Python display?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
```

1 3

```
>>> len(a)
```

5

```
>>> 2 in a
```

False

```
>>> 4 in a
```

True

```
>>> a[3][0]
```

2

Video walkthrough

3.2 Write a function that takes a list `s` and returns a new list that keeps only the even-indexed elements of `s` and multiplies them by their corresponding index.

```
def even_weighted(s):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """

    return [_____]

    return [i * s[i] for i in range(len(s)) if i % 2 == 0]
```

The key point to note is that instead of iterating over each element in the list, we must instead iterate over the indices of the list. Otherwise, there's no way to tell if we should keep a given element.

One way of solving these problems is to try and write your solution as a for loop first, and then transform it into a list comprehension. The for loop solution might look something like this:

```
result = []
for i in range(len(s)):
    if i % 2 == 0:
        result = result + [i * s[i]]
return result
```

[Video walkthrough](#)

- 3.3 Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```
def max_product(s):
    """Return the maximum product that can be formed using non-consecutive
    elements of s.

    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """

    if s == []:
        return 1
    elif len(s) == 1: # Base case optional
        return s[0]
    else:
        return max(max_product(s[1:]), s[0] * max_product(s[2:]))
```

At each step, we choose if we want to include the current number in our product or not:

- If we include the current number, we cannot use the adjacent number.
- If we don't use the current number, we try the adjacent number (and obviously ignore the current number).

The recursive calls represent these two alternate realities. Finally, we pick the one that gives us the largest product.

[Video walkthrough](#)

1. Whole Numbers

- (a) A *hole number* is a number in which every other digit dips below the digits immediately adjacent to it. For example, the number 968 would be considered a hole number because the number 6 is smaller than both of its surrounding digits. Assume that we only pass in numbers that have an odd number of digits. Define the following function so that it properly identifies hole numbers.

```
def check_hole_number(n):
    """
    >>> check_hole_number(123)
    False
    >>> check_hole_number(3241968)
    True
    >>> check_hole_number(3245968)
    False
    """

    if n // 10 == 0:

        return True

    # The \ symbol just allows me to continue this line of code on a new line.

    # It's only included to make sure all the code stays on the page

    return ((n // 10) % 10) < (n % 10) and ((n // 10) % 10) < ((n // 100) % 10) \

        and check_hole_number(n // 100)
```

- (b) Define the following function so that it properly identifies mountain numbers. A *mountain number* is a number that either
- has digits that strictly decrease from right to left OR strictly increase from right to left
 - has digits that increase from right to left up to some point in the middle of the number (not necessarily the exact middle digit). After reaching the maximum digit, the digits to the left of the maximum digit should strictly decrease.

```
def check_mountain_number(n):
    """
    >>> check_mountain_number(103)
    False
    >>> check_mountain_number(153)
    True
    >>> check_mountain_number(123456)
    True
    >>> check_mountain_number(2345986)
    True
    """

    def helper(x, is_increasing):

        if x // 10 == 0:

            return True
```

```
    if is_increasing and (x % 10) < ((x // 10) % 10):  
        return helper(x // 10, is_increasing)  
    return (x % 10) > ((x // 10) % 10) and helper(x // 10, False)  
return helper(n, True)
```