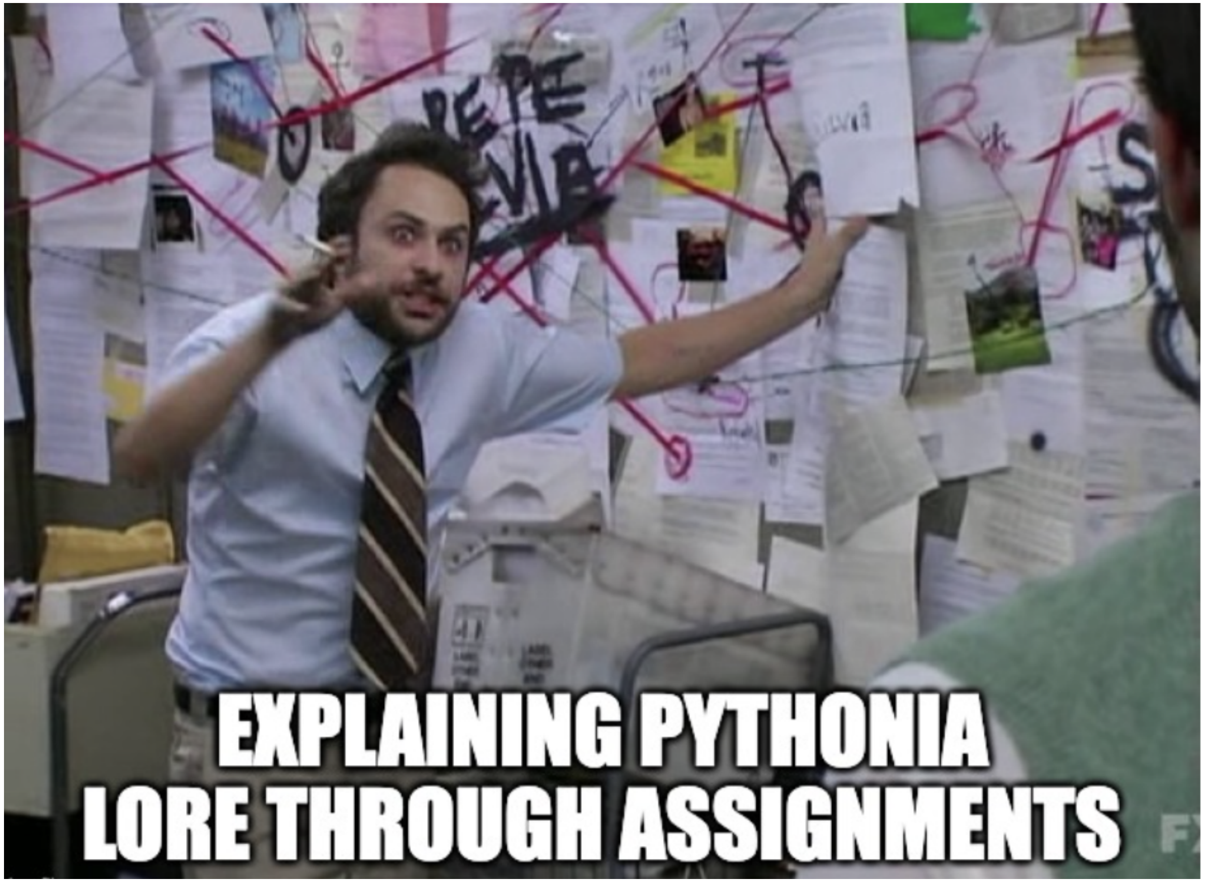
Discussion 6

DSC 20, Fall 2023

Meme of the Week



Agenda

- **Complexity** - Math Review, Interpretation, Calculation
- **Recursion** - Base Case, Recursive Calls, Logic

Complexity

Time complexity is an empirical method to measure the efficiency of code. Since everyone's computer is different, we can't compare code with actual numbers. Instead, we classify them into different runtime categories that allow us to infer its runtime relative to our input.

Complexity Fundamentals

1. Code should be quantified into big O values

2. Nested code will have compounded big O values

3. The largest term dictates the growth rate (i.e. only largest term matters)

4. Constants are irrelevant

5. big O analysis uses similar ideas to calculus and limits - reference your math knowledge

Complexity - Basic Interpretation

Operations that take a **constant** time to run and run at the same speed regardless of input size -> $O(1)$

```
In [ ]: 1 + 1 + 1 + 1 + 1 # O(1)
```

A statement that takes **linear** time to run will increase linearly with the size of input -> $O(n)$

```
In [ ]: for _ in range(n): # O(n)
            print(n)
```

Complexity - Basic Interpretation (Cont.)

A statement that takes **quadratic** time to run will increase quadratically with the size of input -> $O(n^2)$

In [ ]:

```
for i in range(x): # O(n^2)
    for j in range(y):
        print(i + j)
```

A statement that takes **logarithmic** time to run will increase logarithmically with the size of input -> $O(logn)$

In [ ]:

```
i = n
while i > 0: # O(log(n))
    i = i // 2
```

Complexity - Basic Interpretation (Cont.)

These are very basic examples. Just because there's a nested for loop does not necessarily mean that the runtime is O(n^2). Runtime depends on the number of iterations happening and the cost of each calculation. For example, every function you've used in this class so far has a specific runtime -> sorting is usually $O(nlogn)$

Checkpoint

What is the runtime of the following function?

In [24]:

```
def boo(lst):
    for i in lst:
        for j in 1000:
            print(i + j)
```

Checkpoint Solution

Solution: $O(n)$

The second for loop is a bait! The outer loop grows linearly with the input, but the second loop is constant.

In [ ]:

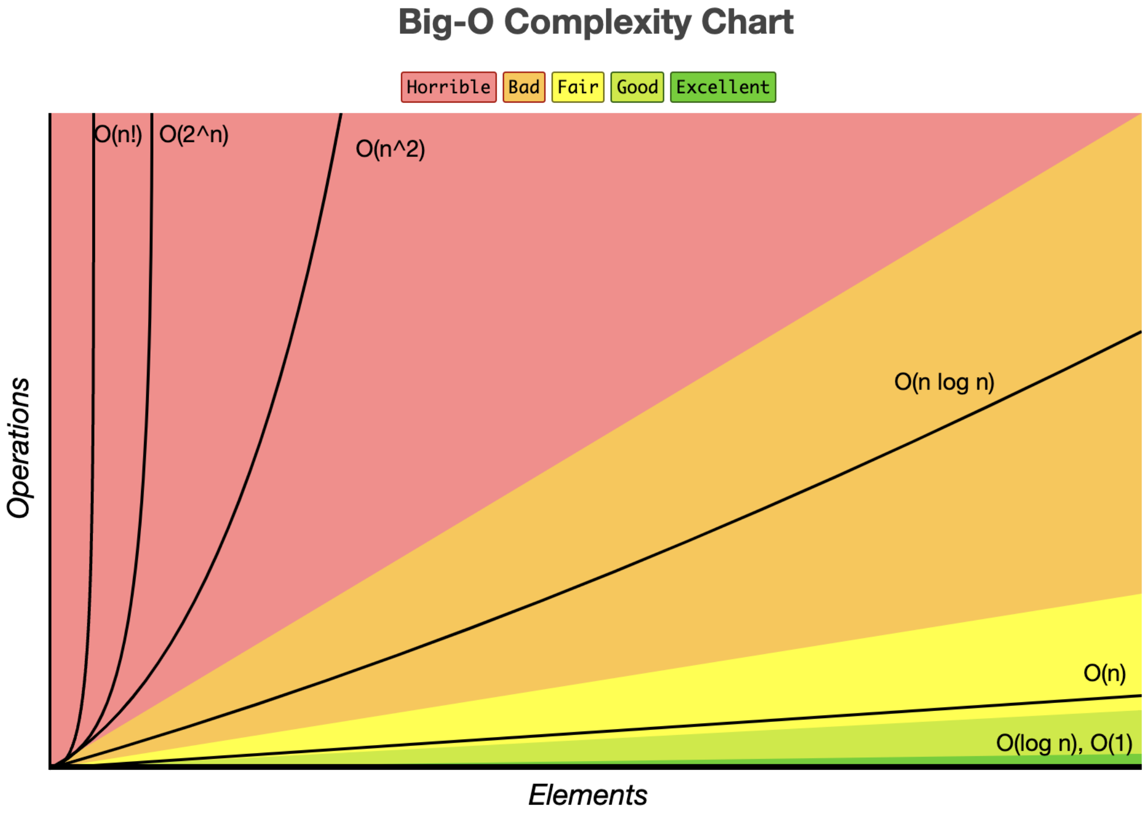
```
def boo(lst):
    for i in lst: # O(n)
        for j in 1000: # O(1)
            print(i + j) # O(1)
```

O(boo) = $O(n * 1000 * 1) = O(n)$

Complexity - Calculation

tips:

1. Always look for "hallmark features" (ex. if I see i // 2, there's probably log involved)
2. If there are loops, check the iteration count (don't assume n runtime)
3. Order of Growth follows mathematical principles. Only consider the largest term



Checkpoint

Determine the time complexity of the following function

In [9]:

```
def bubble_sort(arr):
    """
    function that sorts a list of values.
    """
    n = len(arr)
    for i in range(n):
        for j in range(n - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Checkpoint Solution

Solution: $O(n^2)$

Outer for loop, which runs n times. Inner for loop that runs $n - 1$ times. Within these 2 for loops, operations are all constant.

In [ ]:

```python
def bubble_sort(arr):
    n = len(arr) # O(1)
    for i in range(n): # O(n)
        for j in range(n - 1): # O(n-1)
            if arr[j] > arr[j + 1]: # O(1)
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

O(bubble_sort) = $O(1 + n * (n - 1) * 1) = O(n^2 - n + 1) = O(n^2)$

Checkpoint

Determine the time complexity of the following function

In [11]:

```python
def most_frequent_numbers(arr):
    """
    function to find the two numbers that
    appear most frequently in the list.
    """
    nums_dict = {}
    for num in arr:
        if num in nums_dict:
            nums_dict[num] += 1
        else:
            nums_dict[num] = 1
    most_frequent = sorted(nums_dict.items(), \
           key=lambda x: x[1], reverse=True)[:2]
    return [num[0] for num in most_frequent]
```

Checkpoint Solution

Solution: $O(nlogn)$

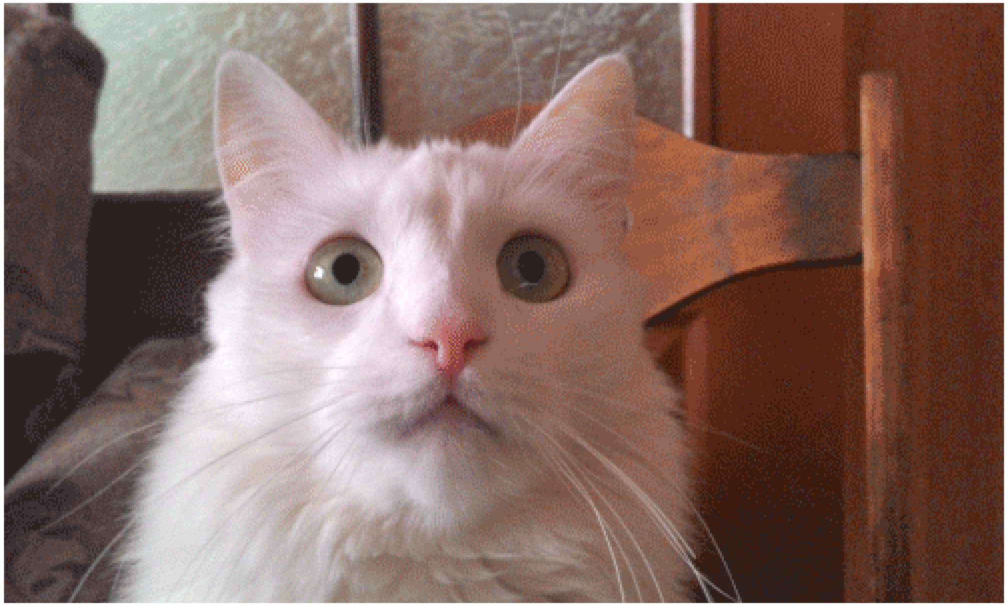
Outer for loop, which runs n times. Within the loop, operations are all $O(1)$. Outside of the loop, sorted() is called, which is a sorting algorithm with time complexity of $O(nlogn)$.

```
In [ ]: def most_frequent_numbers(arr):
            nums_dict = {}
            for num in arr: # O(n)
                if num in nums_dict: # O(1)
                    nums_dict[num] += 1
                else:
                    nums_dict[num] = 1
            #O(nlogn)
            most_frequent = sorted(nums_dict.items(), \
                   key=lambda x: x[1], reverse=True)[:2]
            return [num[0] for num in most_frequent]
```

O(most_frequent_numbers) = $O(n * 1 + nlogn) = O(nlogn)$

Recursion

Recursion is a design method for code - it refers to a class of functions that call on itself repeatedly. A lot of important ideas' optimal solutions are recursive and many algorithms depend on recursion to function correctly (ex. BFS, DFS, Dijkstra's algorithm, BST's). You can't escape it :)



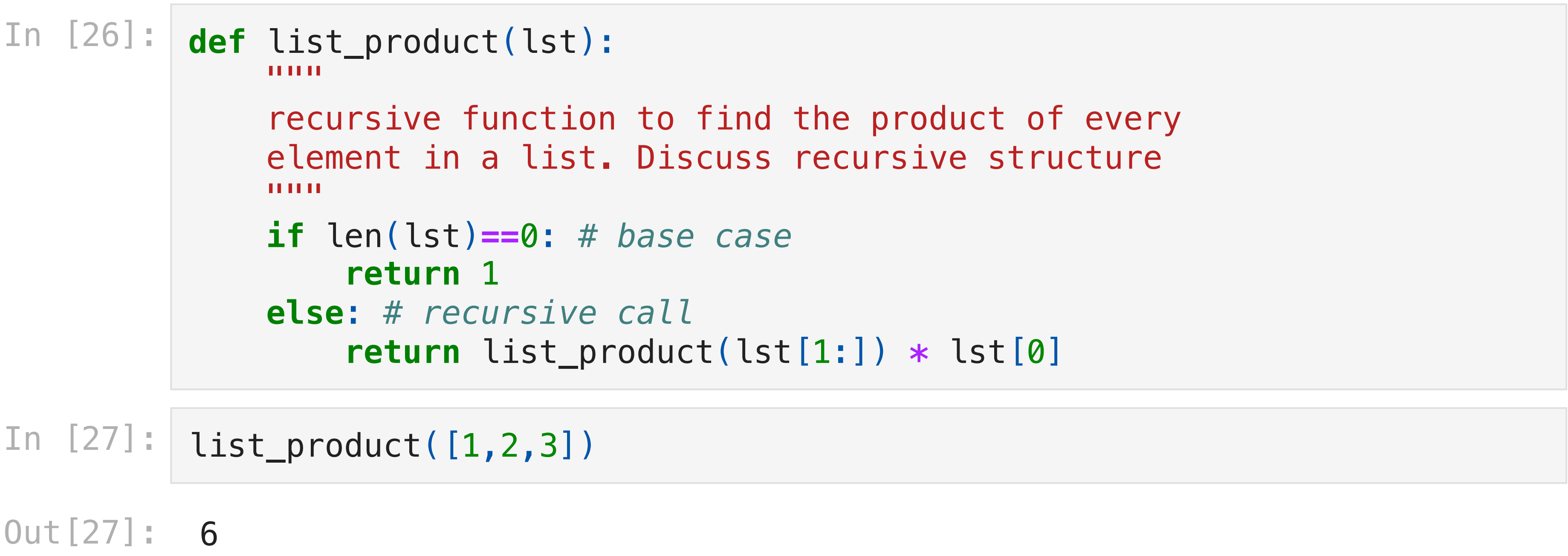
Recursion - Base Case

Base case(s) are regarded as the most important part of a recursive function. They determine the stop point for recursion and begin the argument passing up the "stack" of recursive calls. Without a well written base case, recursion will either never end or end incorrectly. When writing recursion questions, always start with determining the base case.

Recursion - Recursive Calls

The crux of a recursive function working is the recursive calls. These calls will repeat until the base case is reached, creating the "stack" of recursive calls that will begin resolving at the base case. Keep in mind when writing recursive calls that every call needs to trend towards the base case.

Recursion - Example



In [26]:

```python
def list_product(lst):
    """
    recursive function to find the product of every
    element in a list. Discuss recursive structure
    """
    if len(lst)==0: # base case
        return 1
    else: # recursive call
        return list_product(lst[1:]) * lst[0]
```

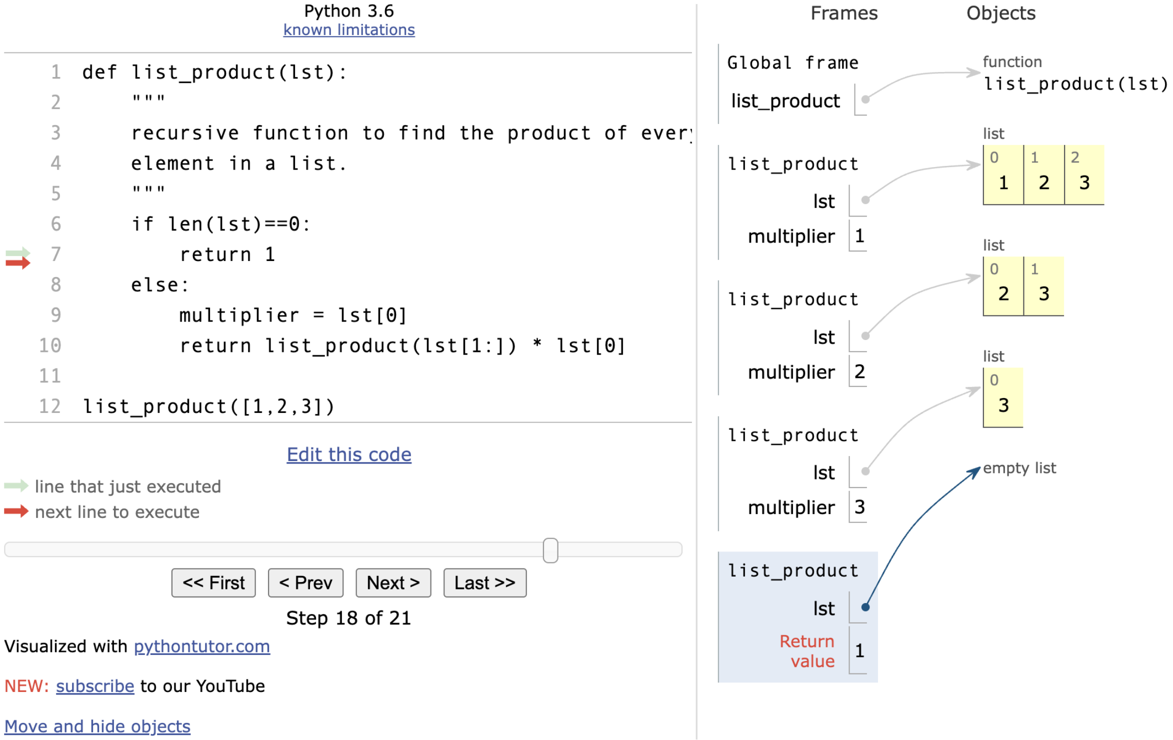
In [27]:

```python
list_product([1,2,3])
```

Out[27]:

```
6
```

Recursion - Tracing Logic



Python 3.6
known limitations

```
def list_product(lst):
    """
    recursive function to find the product of ever
    element in a list.
    """
    if len(lst)==0:
        return 1
    else:
        multiplier = lst[0]
        return list_product(lst[1:]) * lst[0]

list_product([1,2,3])
```

Edit this code

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 18 of 21

Visualized with pythontutor.com

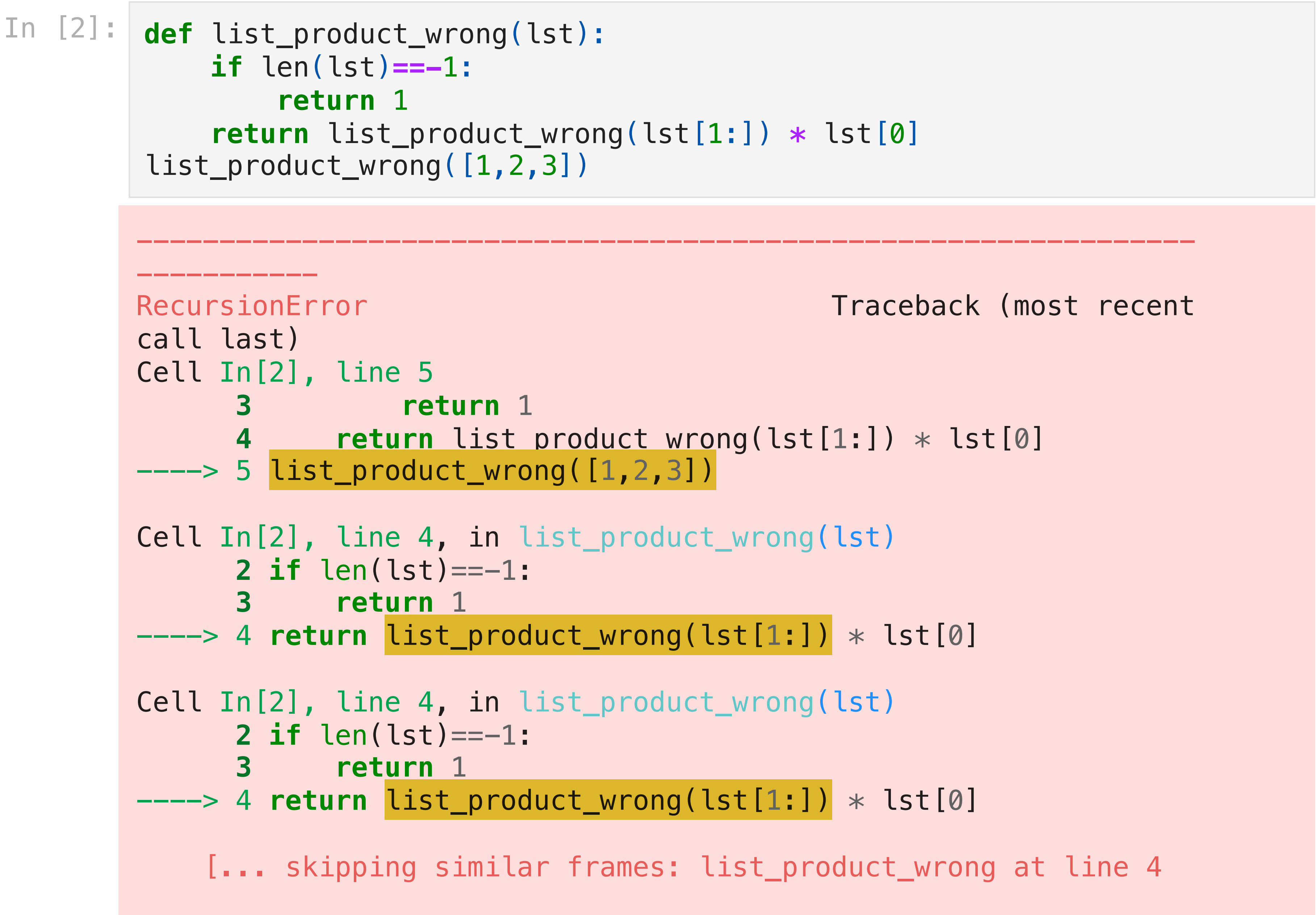
NEW: subscribe to our YouTube

Move and hide objects

The recurisve calls of the function generates a "stack" of recursive functions to resolve, each waiting for the result from the next until it can be solved. No resolution can happen until the base case is reached and returns the iniital value.

link

Recursion - Bad Base Case



```
In [2]: def list_product_wrong(lst):
            if len(lst)==-1:
                return 1
            return list_product_wrong(lst[1:]) * lst[0]
        list_product_wrong([1,2,3])
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent
call last)
Cell In[2], line 5
      3         return 1
      4     return list_product_wrong(lst[1:]) * lst[0]
----> 5 list_product_wrong([1,2,3])

Cell In[2], line 4, in list_product_wrong(lst)
      2 if len(lst)==-1:
      3     return 1
----> 4 return list_product_wrong(lst[1:]) * lst[0]

Cell In[2], line 4, in list_product_wrong(lst)
      2 if len(lst)==-1:
      3     return 1
----> 4 return list_product_wrong(lst[1:]) * lst[0]

    [... skipping similar frames: list_product_wrong at line 4
(2969 times)]

Cell In[2], line 4, in list_product_wrong(lst)
      2 if len(lst)==-1:
      3     return 1
----> 4 return list_product_wrong(lst[1:]) * lst[0]

Cell In[2], line 2, in list_product_wrong(lst)
      1 def list_product_wrong(lst):
----> 2     if len(lst)==-1:
      3         return 1
      4     return list_product_wrong(lst[1:]) * lst[0]

RecursionError: maximum recursion depth exceeded while calling a
Python object
```

Checkpoint

What should the base case be? What should the recursive call be?

In [28]:

```python
def reverse_recursive(s):
    """
    Recursive function to reverse a string

    args:
        s(string): string to be reversed
    returns:
        reversed string

    >>> reverse_recursive('siolgnal aniram')
    marina langlois
    """
    # your implementation here
```

Checkpoint Solution

In [32]:

```
def reverse_recursive(s):
    if len(s) == 0: # can be 0 or 1
        return s
    else:
        return s[-1] + reverse_recursive(s[:-1])

print(reverse_recursive('siolgnal aniram'))
```

```
marina langlois
```

practice question solutions

What do the following runtime expressions evaluate to?

$$O(2n + nlogn + \sqrt{n} + 100) = O(nlogn)$$

$$O(n! + 2^n + n^2 + 999 + n^n) = O(n^n)$$

$$O(n^3 + n^2\sqrt{n} + 9999n) = O(n^3)$$

What is the runtime complexity of the following function?

In [38]:

```python
def complexity(x):
    total = 1
    k_sum = 0

    for i in range(x):
        for j in range(x**2, x**2 + 5*x - 50):
            total += j
        k = x
        while k > 0:
            k = k // 2
            k_sum += k

    return total, k_sum
```

Solution: $O(n^2)$

Outer for loop which runs n times. Inner for loop that runs for $(n^2 + 5 * n - 50) - n^2 = 5 * n - 50$ times. Inner while loop that runs $logn$ times. All other operations are constant.

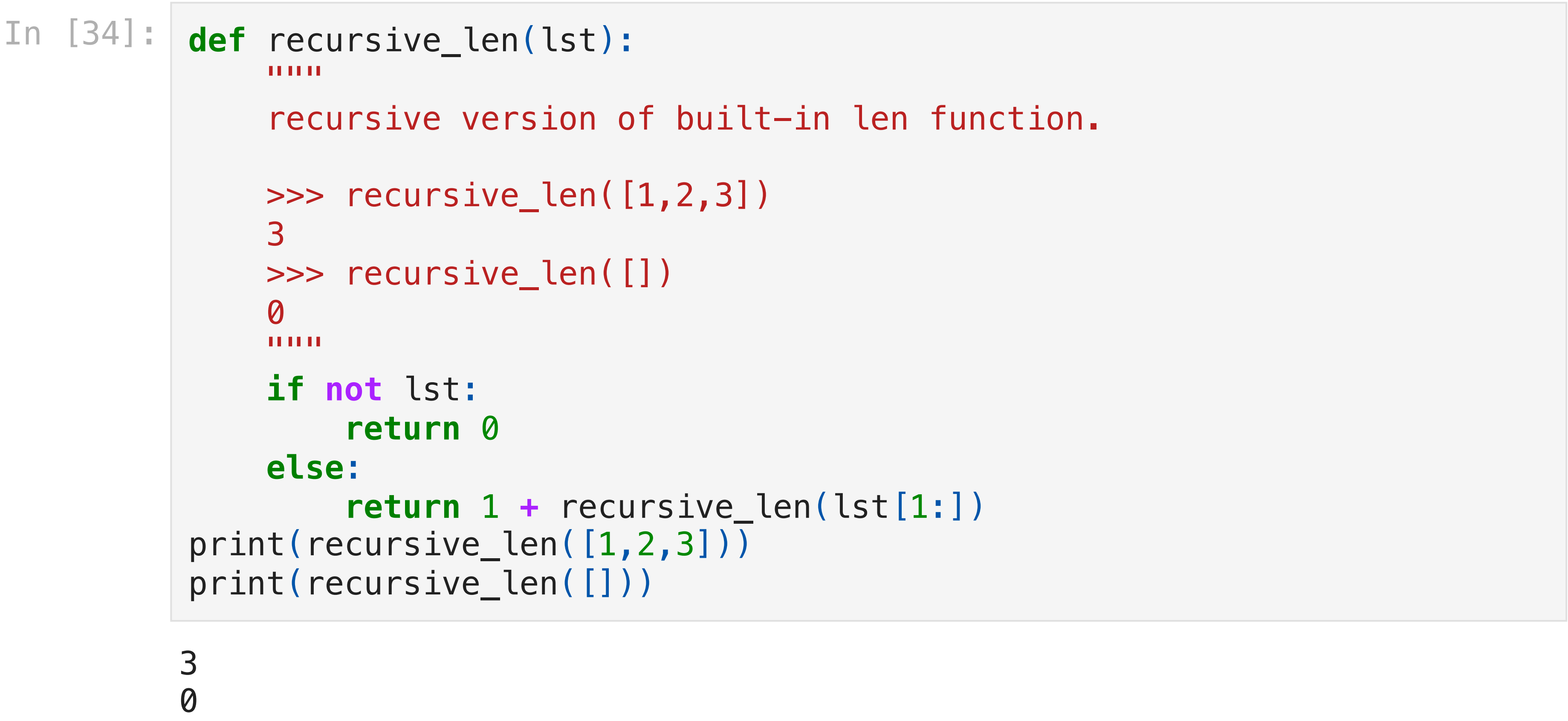
In [37]:

```
def complexity(x):
    total = 1
    k_sum = 0

    for i in range(x): # O(n)
        for j in range(x**2, x**2 + 5*x - 50): # O(n)
            total += j
        k = x
        while k > 0: # O(logn)
            k = k // 2
            k_sum += k

    return total, k_sum
```

O(complexity) =

$$O(n((n^2 + 5n - 50 - n^2) + logn)) = O(n((5n - 50) + logn))$$

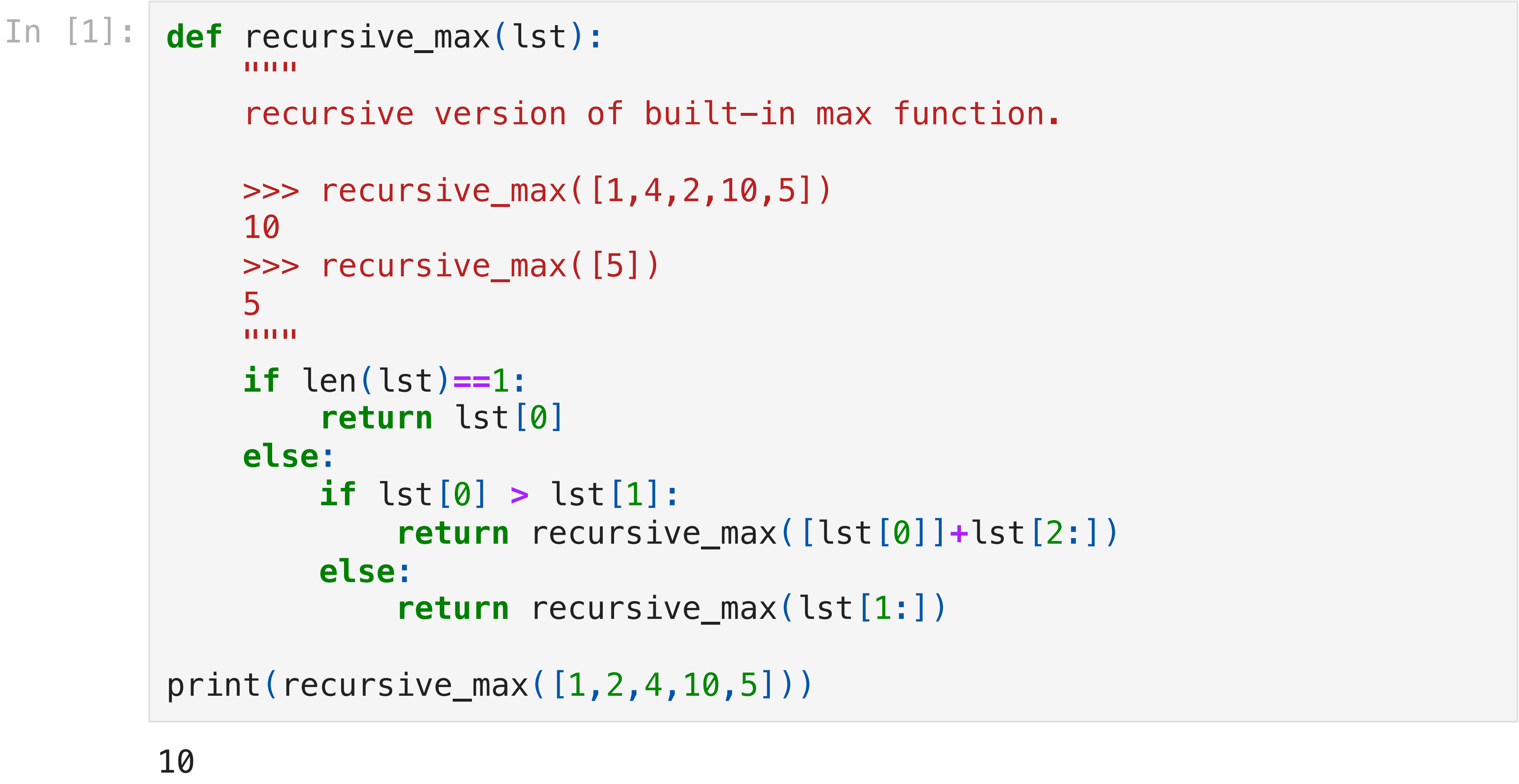
$$= O(5n^2 - 50n + nlogn) = O(n^2)$$



In [34]:

```
def recursive_len(lst):
    """
    recursive version of built-in len function.

    >>> recursive_len([1,2,3])
    3
    >>> recursive_len([])
    0
    """
    if not lst:
        return 0
    else:
        return 1 + recursive_len(lst[1:])
print(recursive_len([1,2,3]))
print(recursive_len([]))
```

```
3
0
```



```python
def recursive_max(lst):
    """
    recursive version of built-in max function.

    >>> recursive_max([1,4,2,10,5])
    10
    >>> recursive_max([5])
    5
    """
    if len(lst)==1:
        return lst[0]
    else:
        if lst[0] > lst[1]:
            return recursive_max([lst[0]]+lst[2:])
        else:
            return recursive_max(lst[1:])

print(recursive_max([1,2,4,10,5]))
```

```
10
```

pythontutor

Discussion Attendance

Take 2 minutes and head to gradescope to complete discussion attendance. The assignment is called Discussion 6 Participation.

Thanks for coming!