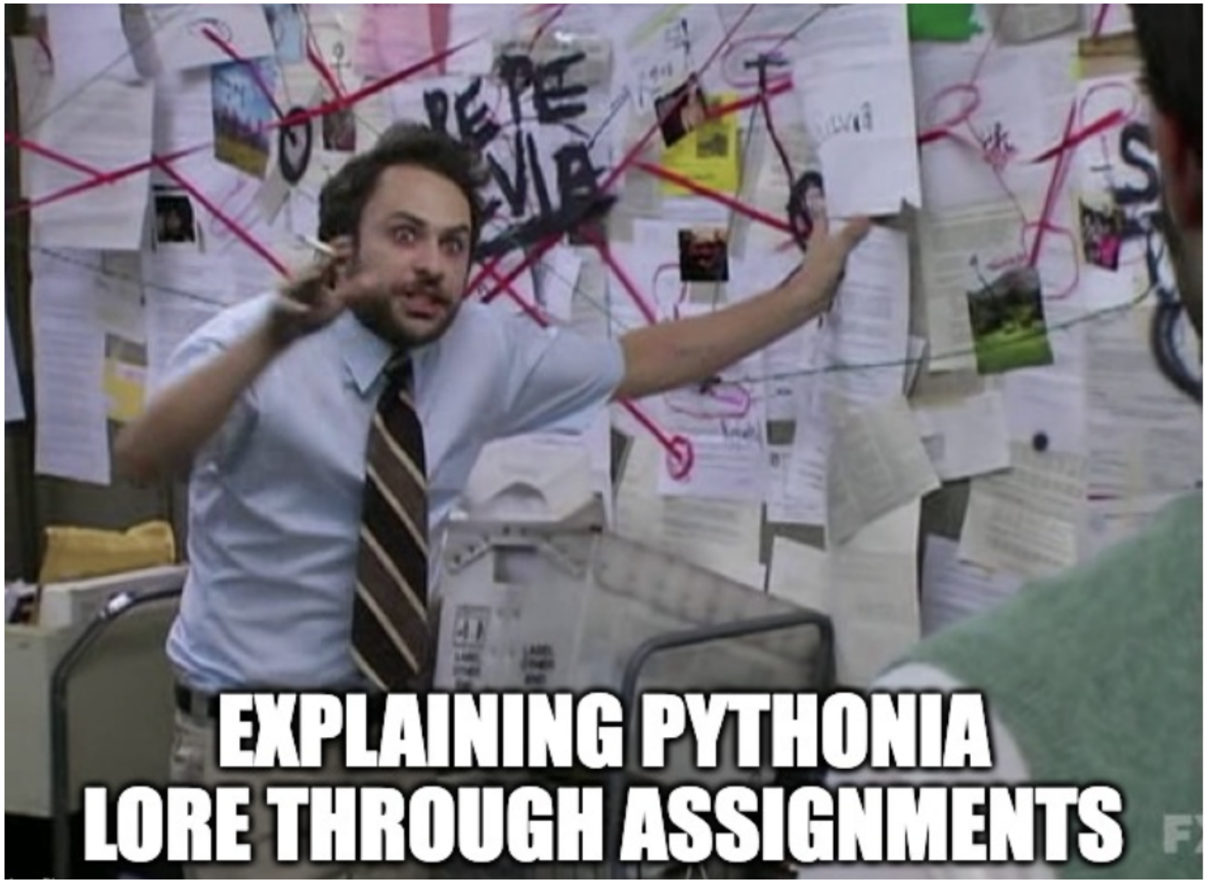
Discussion 6

DSC 20, Winter 2024

Meme of the Week



Agenda

- **Advanced Argument Passing** - Args, Kwargs, defaults
- **Recursion** - Base Case, Recursive Calls, Logic

*args

- Used when an unknown number of arguments will be passed into a function
- Denoted by * in the method header (IMPORTANT)
- processed in a similar manner to a list

In [10]:
```
def summation(*nums):
    return sum(nums)
print(summation())
print(summation(1,2,3,4,5))
```

```
0
15
```

Checkpoint

What is the result of this function call?

```
In [ ]: def generate_names(*name_parts):
            output = []
            for name in name_parts:
                output.append(name*2)
            return output
        generate_names('pika', 'dodo')
```

A. [['pikapika'], ['dodododo']]

B. ['pikapika', 'dodododo']

C. ['pika', 'dodo']

D.[['pika'], ['dodo']]

Checkpoint Solution

```
In [14]: def generate_names(*name_parts):
             output = []
             for name in name_parts:
                 output.append(name*2)
             return output
         generate_names('pika', 'dodo')
```

```
Out[14]: ['pikapika', 'dododo do']
```

**kwargs

- Used when an unknown number of **keyworded** arguments will be passed into a function
- Denoted by ** in the method header (IMPORTANT)
- processed in a similar manner to a dictionary

In [17]:
```
marina = {'marina':1}
def create_dct(**entry):
    return dict(entry)
print(create_dct())
print(create_dct(marina=1, langlois=2))
print(marina)
```

```
{}
{'marina': 1, 'langlois': 2}
{'marina': 1}
```

default_arguments

- Basically normal arguments, but with a default value
- if no value is passed, default value is set
- if a value is passed, default value is overwritten

In [16]:
```
def check_legal_age(age=18):
    return age>=21
print(check_legal_age())
print(check_legal_age(21))
```

```
False
True
```

Checkpoint

What is the result of this function call?

```
In [18]: def filter_dict(t=2, **items_in):
             return {k:v for k,v in items_in.items() if len(v)>t}
         filter_dict(temp=[1,2], test=[3,4,5], idk=[6,7,8,9])

Out[18]: {'test': [3, 4, 5], 'idk': [6, 7, 8, 9]}
```

A. {'idk': [6, 7, 8, 9]}

B. {'temp': [1, 2], 'test': [3, 4, 5], 'idk': [6, 7, 8, 9]}

C. {'test': [3, 4, 5], 'idk': [6, 7, 8, 9]}

D. {'temp': [1, 2], 'test': [3, 4, 5]}

note

complex argument ordering gets really messy

```
In [20]: def func(norm, *args, darg=2, **kwargs):
             return [norm, list(args), darg, dict(kwargs)]
         func(42,1,1,1,1,1,1,3,darg=4, test=1)

Out[20]: [42, [1, 1, 1, 1, 1, 1, 3], 4, {'test': 1}]

In [22]: def func(norm, darg=2, *args, **kwargs):
             return [norm, list(args), darg, dict(kwargs)]
         func(42,4,1,1,1,1,1,1,3,test=1)

Out[22]: [42, [1, 1, 1, 1, 1, 1, 3], 4, {'test': 1}]
```

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, DP, etc.). 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.

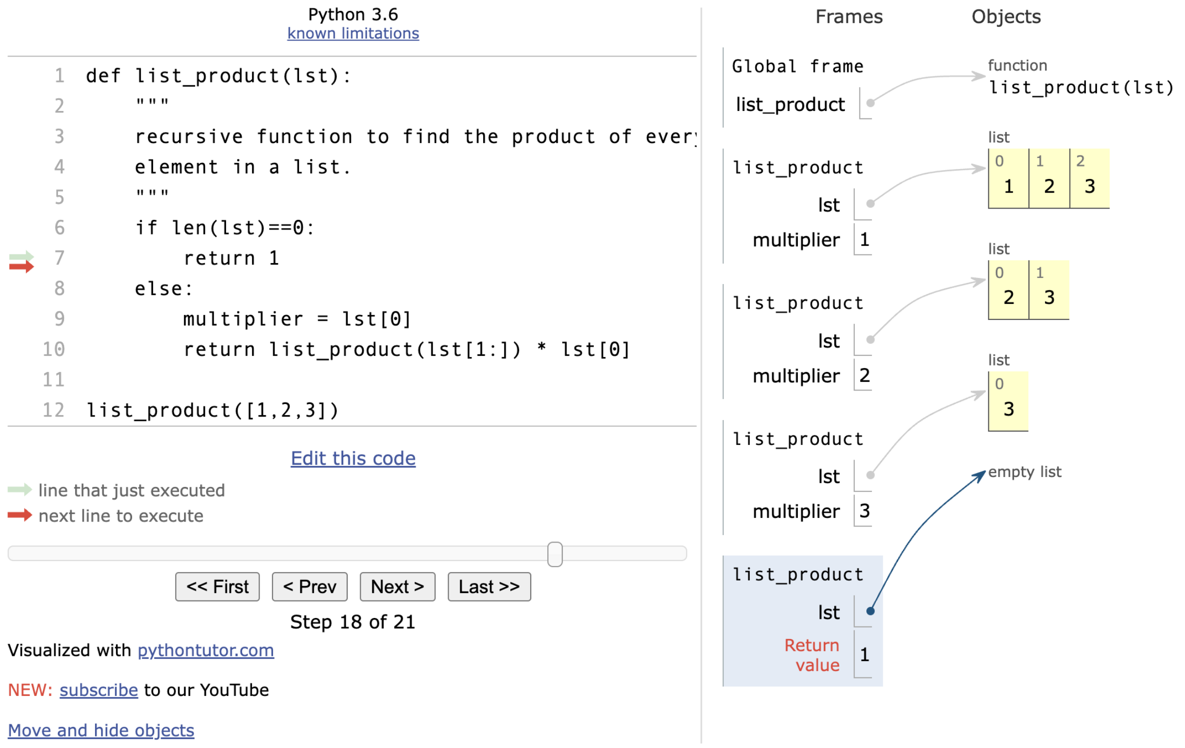
Recursion - Example

```
In [2]: 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 [4]: list_product([1,2,3])
```

```
Out[4]: 6
```

Recursion - Tracing Logic



Python 3.6
known limitations

```python
def list_product(lst):
    """
    recursive function to find the product of every
    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])
```

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)==0:
                return 0
            return list_product_wrong(lst[1:]) * lst[0]
        list_product_wrong([1,2,3])

Out[2]: 0
```

```
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?

```
In [28]: 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
```

A. `if len(s) == 0: return s`

B. `if len(s) == 1: return s`

C. `if len(s): return s`

D. `if len(s) != 1: return s`

Checkpoint Solution

```
In [32]: def reverse_recursive(s):
             if len(s) == 0: # can be 0 or 1
                 return s
```

marina langlois

Checkpoint

What should the recursive call be?

```
In [ ]: 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
            """
            if len(s) == 0: # can be 0 or 1
                return s
```

A. `return s[-1]`

B. `return s[1] + reverse_recursive(s[:1])`

C. `return reverse_recursive(s[:-1])`

D. `return s[-1] + reverse_recursive(s[:-1])`

Checkpoint Solution

```
In [8]:
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

```
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
```

```
In [21]: 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
```

```
In [25]: 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:
                 curr_max = recursive_max(lst[1:])
                 if lst[0] > curr_max:
                     return lst[0]
                 else:
                     return curr_max

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

         10
```

Python Tutor

```
In [27]: def count_len_lsts(*lists, counter=4):
             """
             Write a function that takes in an unknown number of lists
             and returns the sum of the length of the first 'counter'
             lists, default value of 4.

             Args:
                 lists(args): unknown number of lists
                 counter(int): number of lists length to count
             Returns:
                 sum of the lengths of the first counter lists

             >>> count_len_lsts([],[1],[2],[3],[4])
             3
             >>> count_len_lsts([],[],[1,2,3],[4,5], counter=2)
             0
             """
             return sum([len(x) for x in lists[:counter]])
         print(count_len_lsts([],[1],[2],[3],[4]))
         print(count_len_lsts([],[],[1,2,3],[4,5], counter=2))
```

```
3
0
```

Thanks for coming!