Sequences and Higher Order Functions

Scaling a Sequence

- Given a sequence of numbers, how to scale every element?
- Let’s say, scale by 2

```
[5,1,4,9,11,22,12,55]
          ⬇
[10,2,8,18,22,44,24,110]
```

Scaling a Sequence

- Given a sequence of numbers, how to scale every element?

```
def seqScaleI(seq,n):
    output = []
    for i in seq:
        output.append(i*n)
    return output

def seqScaleR(seq,n):
    if not seq:
        return seq
    return [seq[0]*n]+seqScaleR(seq[1:],n)
```

Squaring a Sequence

- Given a sequence of numbers, how to square every element?

```
[5,1,4,9,11,22,12,55]
        ⬇
[25, 1, 16, 81, 121, 484, 144, 3025]
```

```python
def seqSquareI(seq):
    output = []
    for i in seq:
        output.append(i*i)
    return output
```

Squaring/Scaling a Sequence

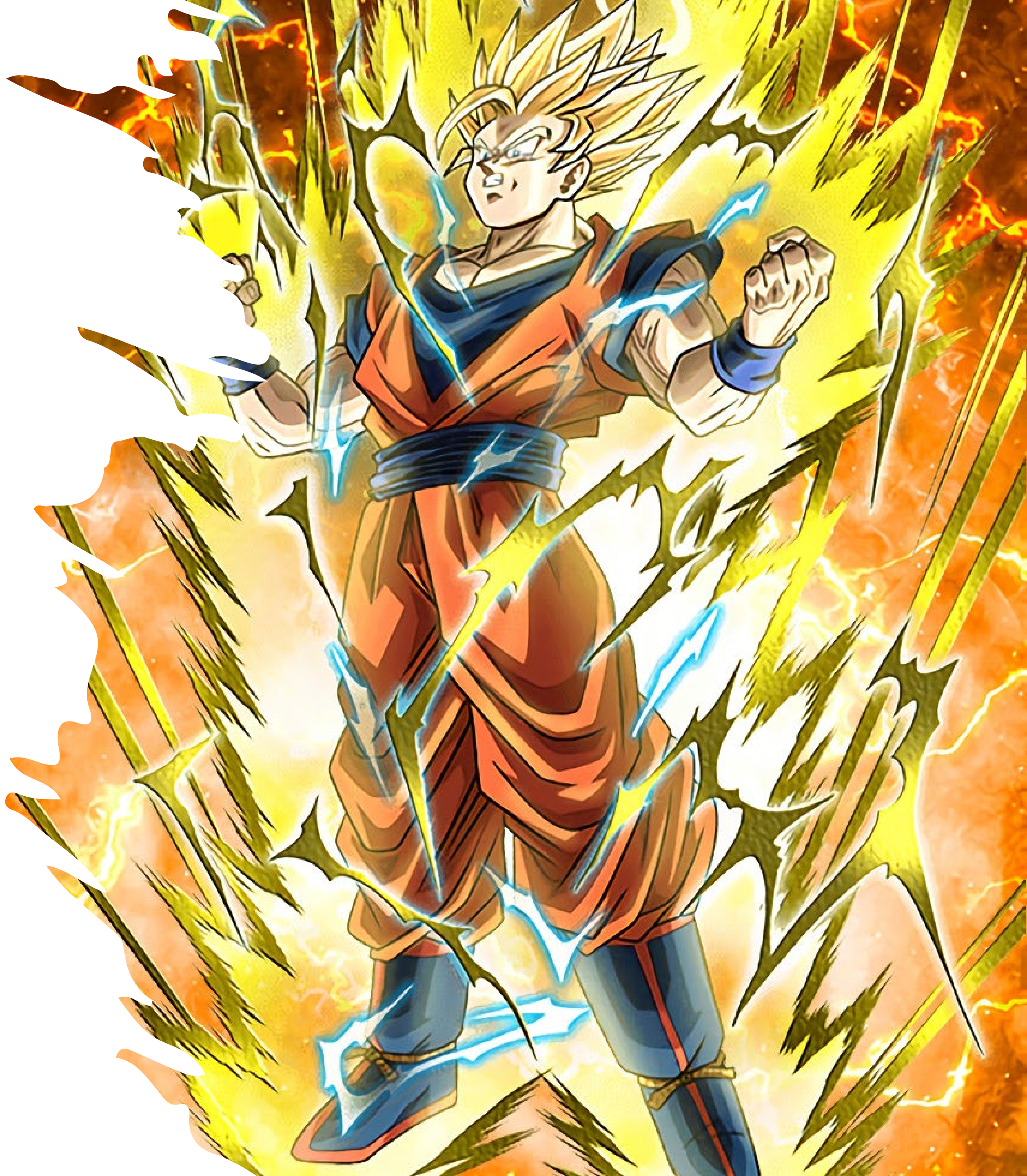
- Other than the function name (that can change to anything), what is the different?

```
def seqScaleI(seq,n):
    output = []
    for i in seq:
        output.append(i*n)
    return output
```

```
def seqSquareI(seq):
    output = []
    for i in seq:
        output.append(i*i)
    return output
```

- What should we do with other operations to a sequence?
  - E.g. cube, abs, etc.?

Power of Capturing Same Pattern



```python
def map(f,seq):
    output = []
    for i in seq:
        output.append(f(i))
    return output
```

```python
def seqScaleI(seq,n):
    output = []
    for i in seq:
        output.append(i*n)
    return output
```

era

```python
def seqSquareI(seq):
    output = []
    for i in seq:
        output.append(i*i)
    return output
```

Difference Operations on a Sequence

```
>>> lst = [5,1,4,9,11,22,12,55]
>>> map(square,lst)
[25, 1, 16, 81, 121, 484, 144, 3025]
>>> map(scale2,lst)
[10, 2, 8, 18, 22, 44, 24, 110]
>>> map(lambda x:x*x,lst)
[25, 1, 16, 81, 121, 484, 144, 3025]
>>> map(lambda x:2*x,lst)
[10, 2, 8, 18, 22, 44, 24, 110]
>>> map(lambda x:-x,lst)
[-5, -1, -4, -9, -11, -22, -12, -55]
```

Sequences and Higher Order Functions

Our `map()`

- However, our map() can only process list
  - Cannot work on other sequences like tuples, strings, etc.
- However, Python does have its original version of map()
  - But it will return a type “map” object
  - You can convert that object into other sequences like list or tuples

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61438>
>>> type(map1)
<class 'map'>
>>> map1Tuple = tuple(map1)
>>> map1Tuple
(1, 2, 3)
>>> map1List = list(map1)
>>> map1List
[]
```

Python's `map()`

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61358>
>>> type(map1)
<class 'map'>
>>> map1List = list(map1)
>>> map1List
[1, 2, 3]
>>> map1Tuple = tuple(map1)
>>> map1Tuple
()
```

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61438>
>>> type(map1)
<class 'map'>
>>> map1Tuple = tuple(map1)
>>> map1Tuple
(1, 2, 3)
>>> map1List = list(map1)
>>> map1List
[]
```

Python's `map()`

- The map object is actually an "iterable"
  - After you "took out" items from the map object, the items will be "gone"
- Conclusion
  - Conversion from a map object to a tuple or list only once

```
>>> tup = (1,-2,3)
>>> map1 = map(abs,tup)
>>> map1
<map object at 0x112e61438>
>>> type(map1)
<class 'map'>
>>> map1Tuple = tuple(map1)
>>> map1Tuple
(1, 2, 3)
>>> map1List = list(map1)
>>> map1List
[]
```

Python’s Filter

Python's Filter

- Python's map()
  - Apply a function `f` to every item `x` in the sequence
- Python's filter()
  - Apply a predicate function `f` to every item `x` in the sequence
    - A predicate is a function that return True or False
  - Return a iterable that
    - Keep the item if `f(x)` returns True
    - Remove the item otherwise

Python's `filter()`

```
>>> l = [1,2,3,'a',(1,2),('b',3)]
>>> filter(lambda x:type(x)==int,l)
<filter object at 0x112e618d0>
>>> list(filter(lambda x:type(x)==int,l))
[1, 2, 3]
>>> l = [1,2,'a',(1,2),6,('b',3),999]
>>> list(filter(lambda x:type(x)==int,l))
[1, 2, 6, 999]
>>> list(filter(lambda x:type(x)==str,l))
['a']
>>> l2 = [1,4,5,-4,9,-99,0,32,-9]
>>> list(filter(lambda x: x < 0 , l2))
[-4, -99, -9]
```

Counting a Sequence Shallowly or Deeply

How to Count the Number of Element in a Sequence?

```
>>> lst = [5,1,4,9,11,22,12,55]
>>> seqCountI(lst)
8
```

- Of course we can use `len()`
- But what if we want to implement it ourselves?

```
def seqCountR(seq):
    if not seq:
        return 0
    return 1 + seqCountR(seq[1:])
```

However, it's Shallow

```
>>> lst2 = [1,2,3,[4,5,6,7]]
>>> seqCountR(lst2)
4
```

- How to count ”deeply”?

```
>>> deepcount(lst2)
7
```

- And what about a list like this

```
>>> lst3 = [1, 4, 9, [1, 4], [4, 9, 16, [1, 4, 9]], [9, 16, 25]]
>>> deepcount(lst3)
14
```

Counting Logic? Shallow Count

- Total count = count of the **first** item + count of the rest
  - But the count of the first item is always 1

```
def seqCountR(seq):
    if not seq:
        return 0
    return 1 + seqCountR(seq[1:])
```

- Can we do the same thing for deep count?
- What is the difference between deep and shallow count?
  - In deep count, the “length” of the first item may not be 1

Counting Logic? Deep Count

- Total count = count of the **first** item + count of the rest
  - But the count of the first item is only 1 if it's not a sequence
- `[1,2,3,4,[2,3,4],[1]]`
  - The first item has a count 1
- `[[1,2],3,4,5]`
  - The first item does **not** has a count 1
- Two questions:
  - How to tell the first item is a sequence or not?
  - What to do if the first item is a sequence?

First Question

- How to tell the first item is a list or not a list
  - Assuming we only have list
  - Not difficult to extend to tuples
- Check

```
type(seq[0])==tuple
```

- E.g.

```
>>> l1 = [1,2,3,4,[2,3,4],[1]]
>>> type(l1[0])==list
False
>>> l2 = [[1,2],3,4,5]
>>> type(l2[0])==list
True
```

Second Question

- If the first item is NOT a list, e.g. `[1,2,3,4,[2,3,4],[1]]`
    - count of the first item is 1
- If the first item IS a list, e.g. `[[1,2],3,4,5]`
    - recursively compute `deepcount()` of the first item!

- And this can handle if the first item is a list of a list of a list of ….
- e.g.
    - `[[[[[1,2],2],4],2],3,4,5]`

Second Question

- If the first item is NOT a list, e.g. `[1,2,3,4,[2,3,4],[1]]`
    - count of the first item is 1
- If the first item IS a list, e.g. `[[1,2],3,4,5]`
    - recursively compute `deepcount()` of the first item!

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```

Simple Test

deepcount([1])

⬇

1 deepcount(1) + deepcount([])

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```

Whenever we reached this line Count + 1

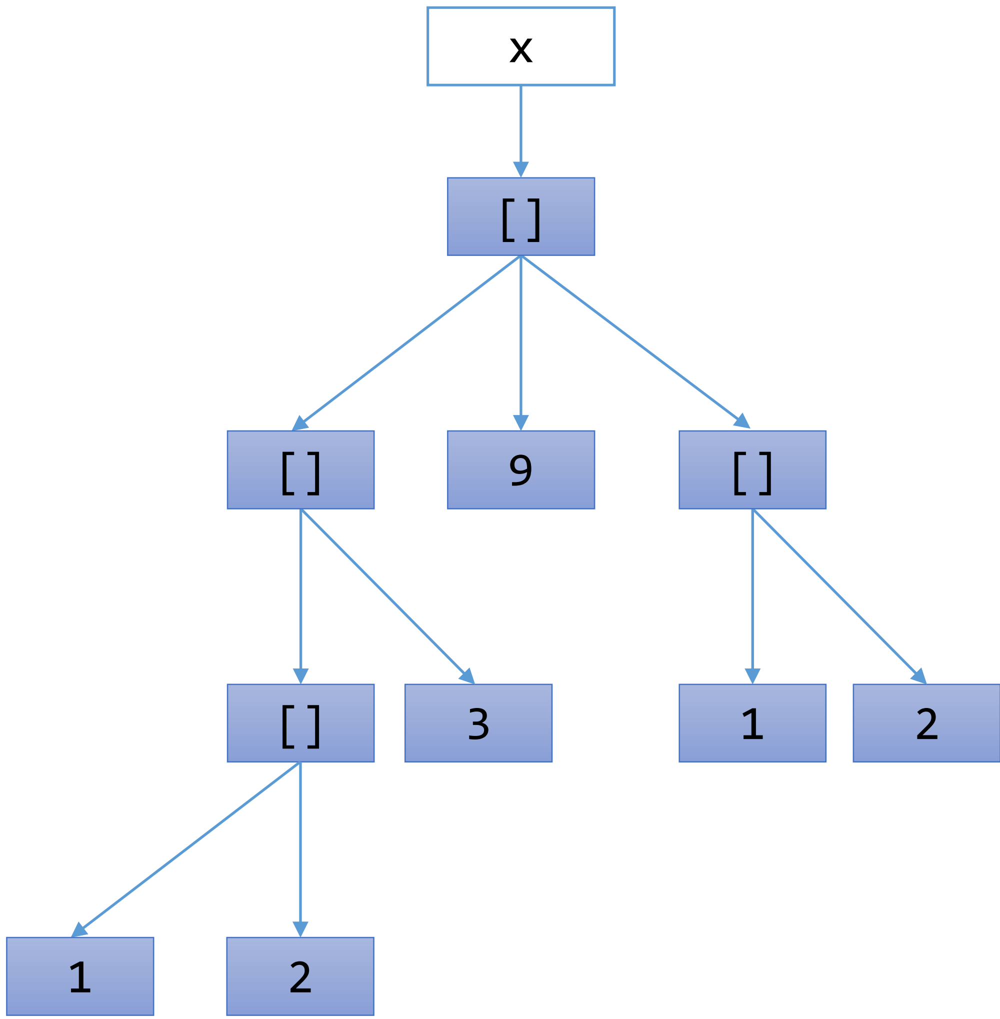
Simple Test

```
deepcount([1,2,3])
          ↓
deepcount(1) + deepcount([2,3])
   1                    ↓
                1 deepcount(2) + deepcount([3])
                                         ↓
                  deepcount(3) + deepcount([])
                1
```

Tracing the Code

- x = [[[1,2],3],9,[1,2]]

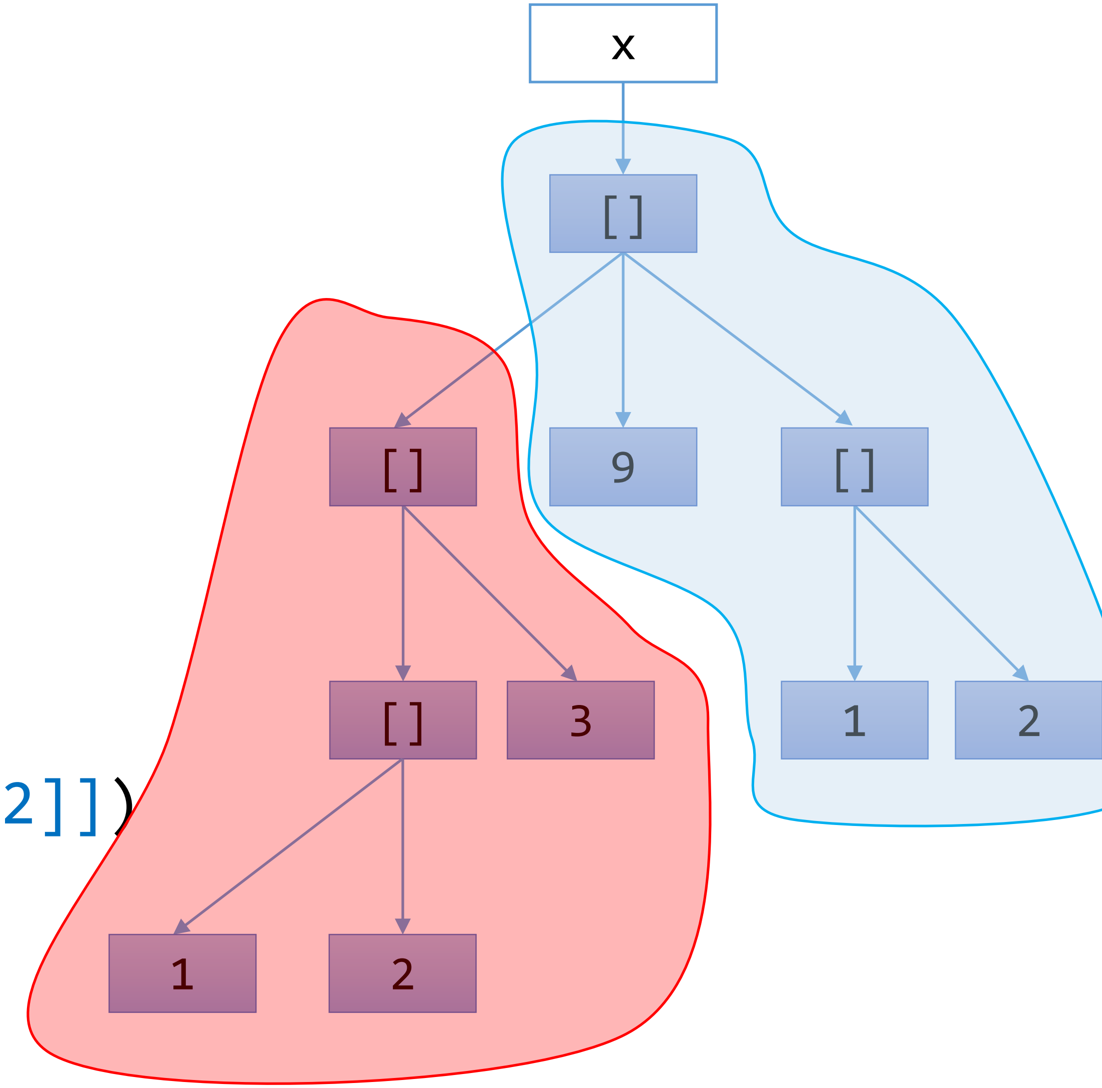
```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```



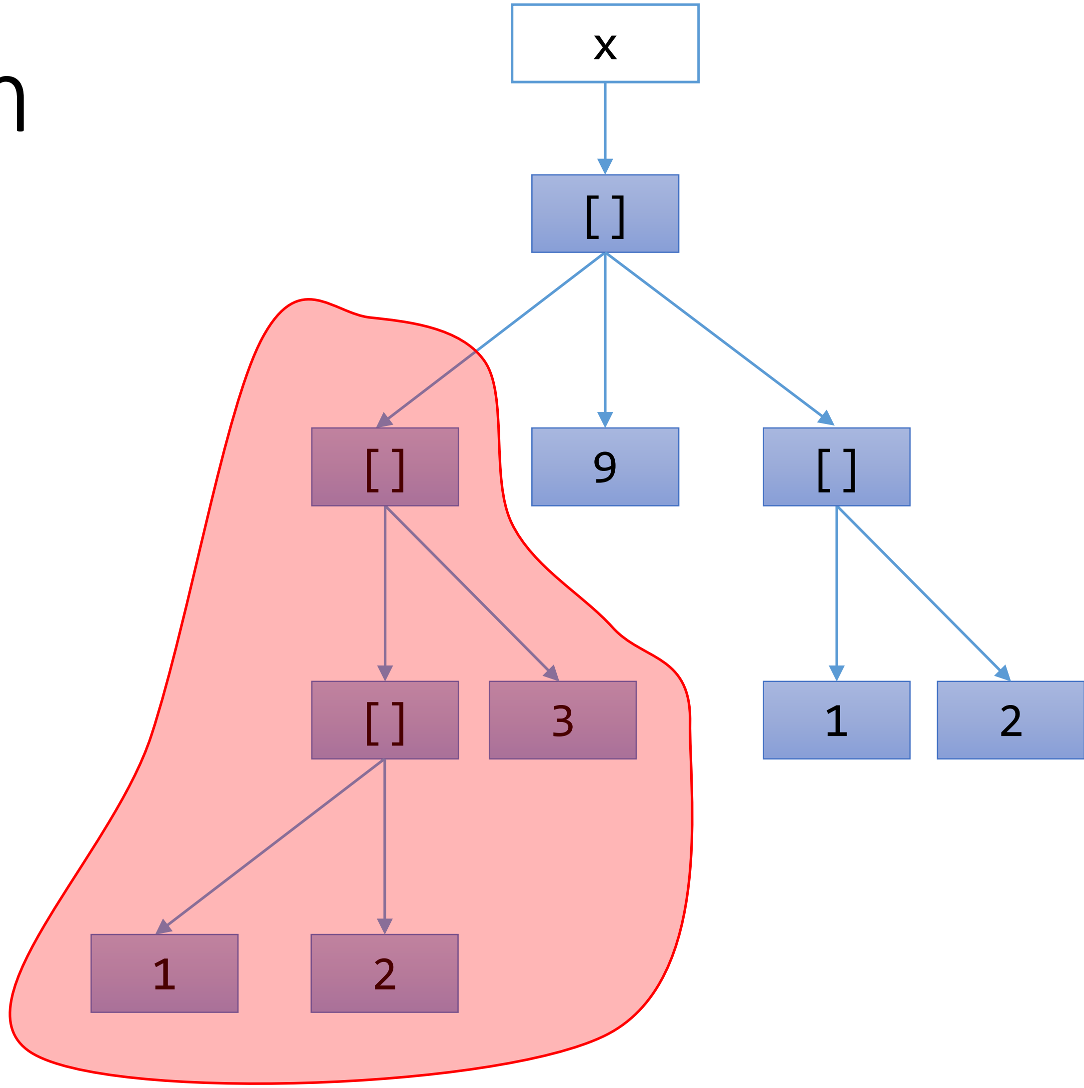
Tracing the Code

deepcount([[[1,2],3],9,[1,2]])

deepcount([[1,2],3])+deepcount([9,[1,2]])



Let’s Consider the Left Term




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

deepcount([1,2]) + deepcount([3])

deepcount(1) + deepcount([2])

deepcount(2)+deepcount([])
```

```
deepcount(3)+deepcount([])
```

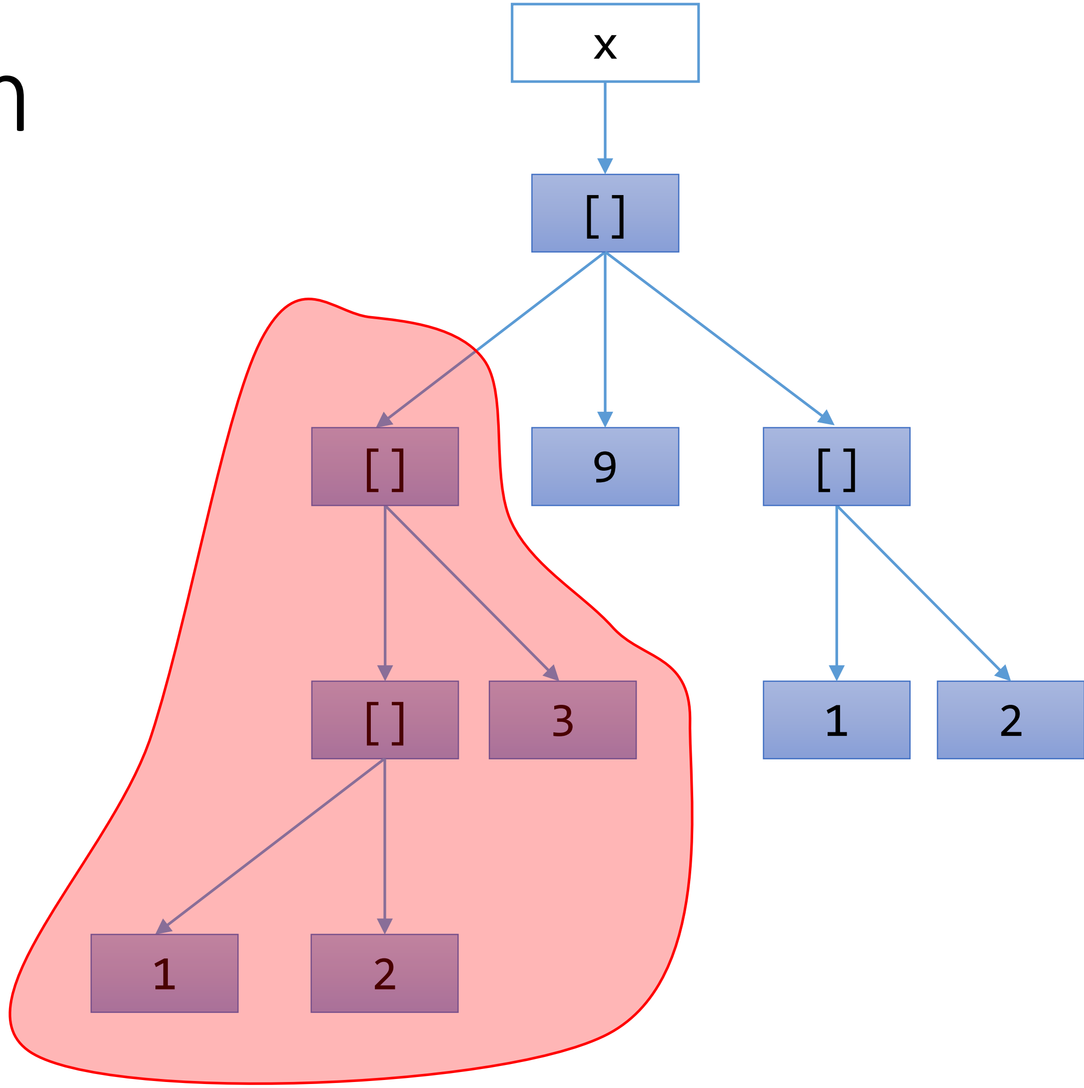
Let’s Consider the Left Term

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

deepcount([1,2]) + deepcount([3])

deepcount(1) + deepcount([2])

deepcount(2)+deepcount([])

deepcount(3)+deepcount([])
```

1

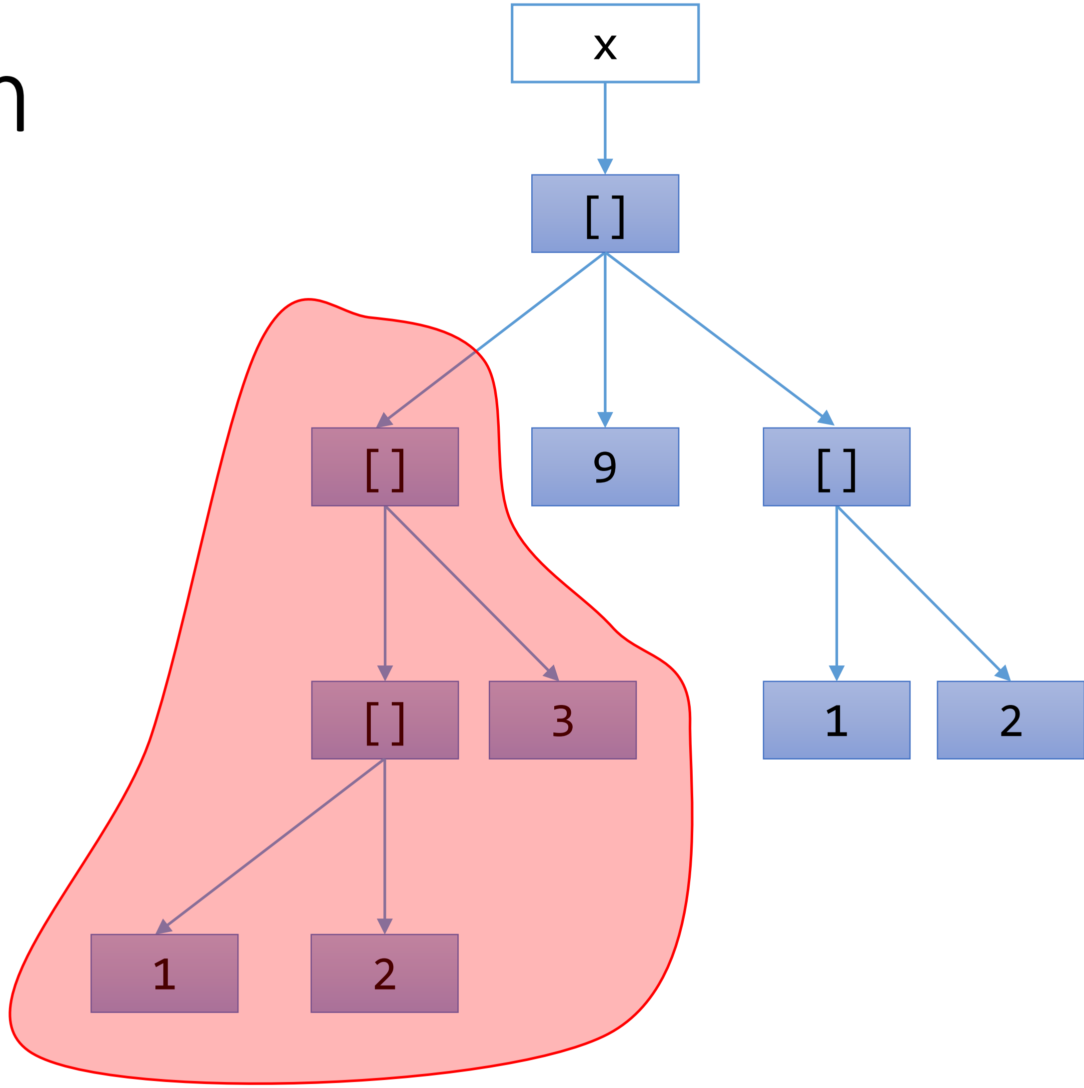
1

1



Let’s Consider the Left Term

```
deepcount([[1,2],3]]) ☞ 3
```



1

1

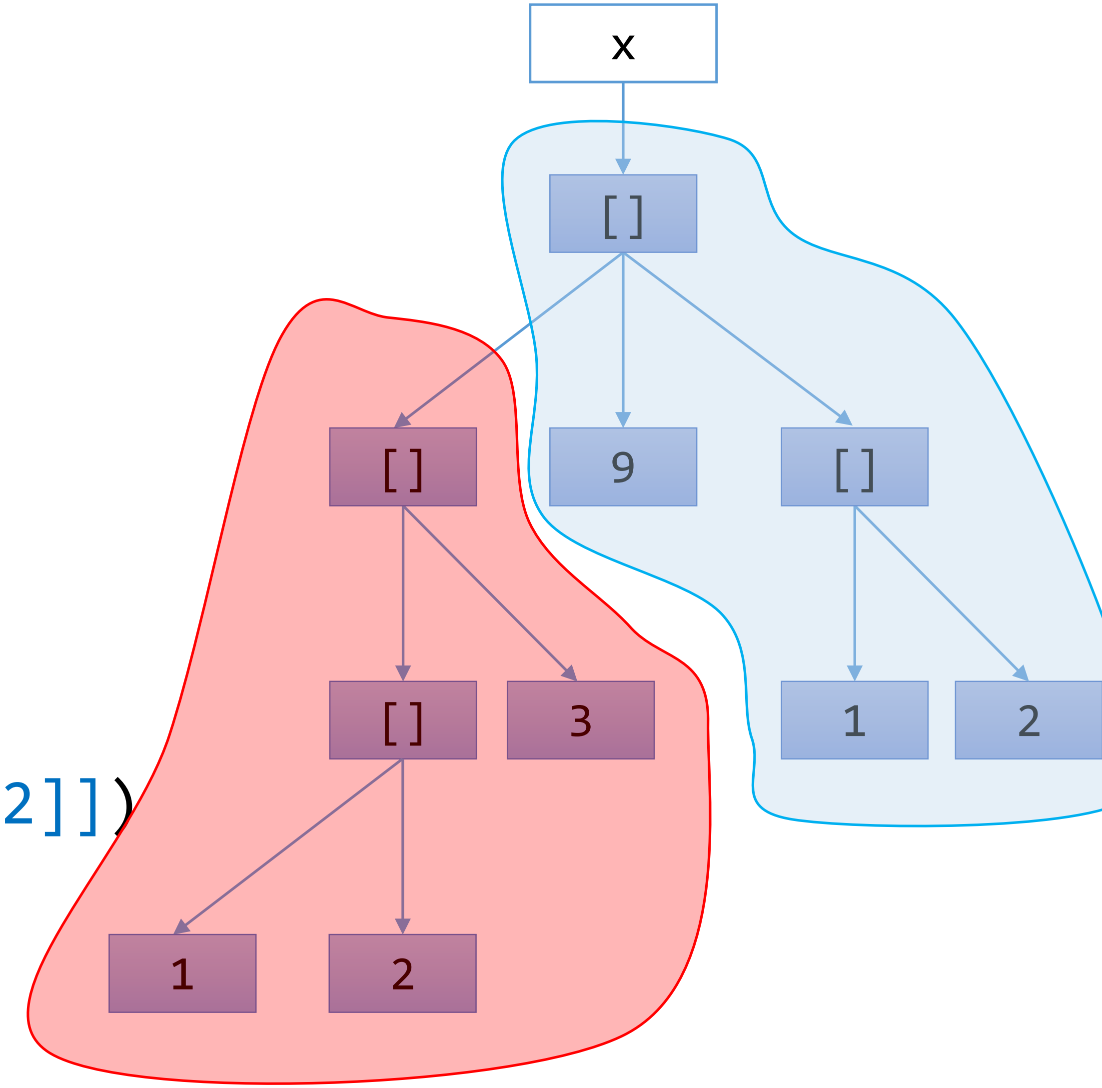
1

Tracing the Code

deepcount([[[1,2],3],9,[1,2]])

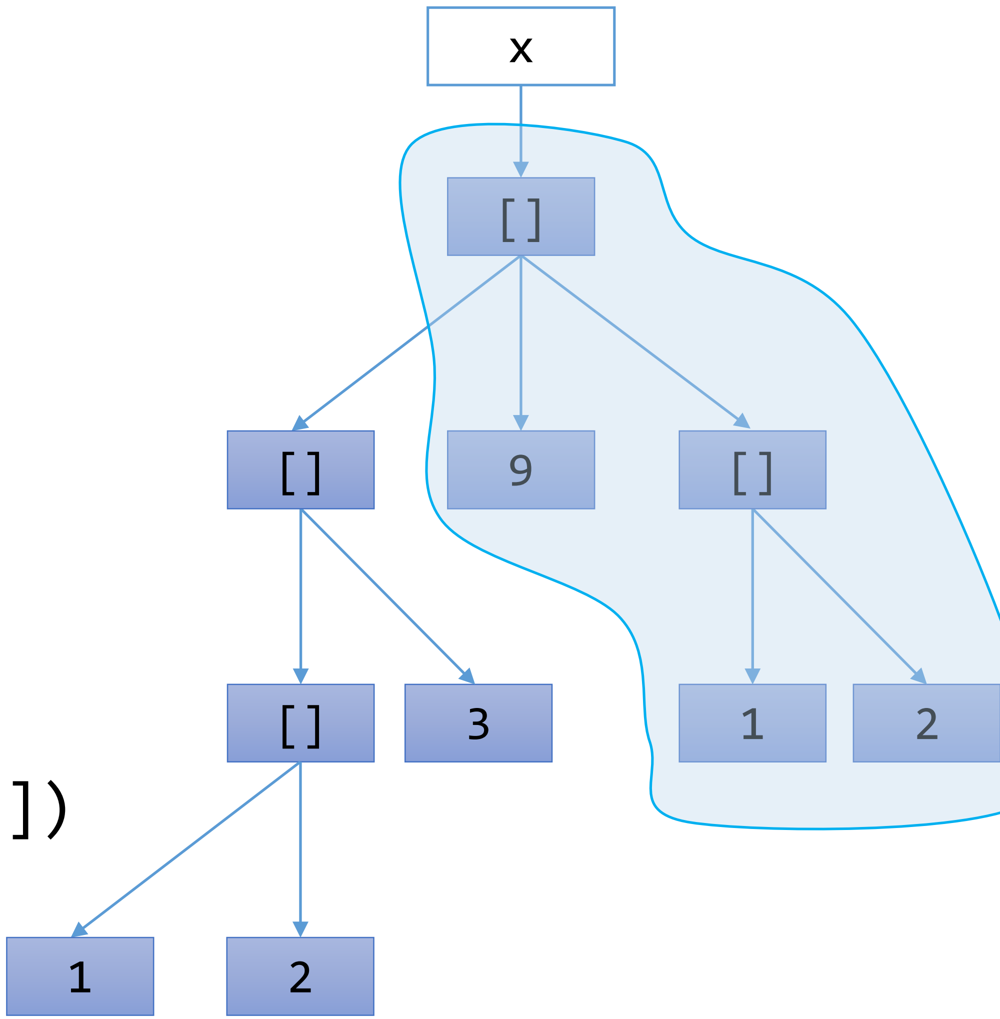
deepcount([[1,2],3]])+deepcount([9,[1,2]])

3 +deepcount([9,[1,2]])



Tracing the Code

```
deepcount([9,[1,2]])

deepcount(9)+deepcount([[1,2]])

        deepcount([1,2]) + deepcount([])

deepcount(1) + deepcount([2])

         deepcount(2) + deepcount([])
```



Tracing the Code

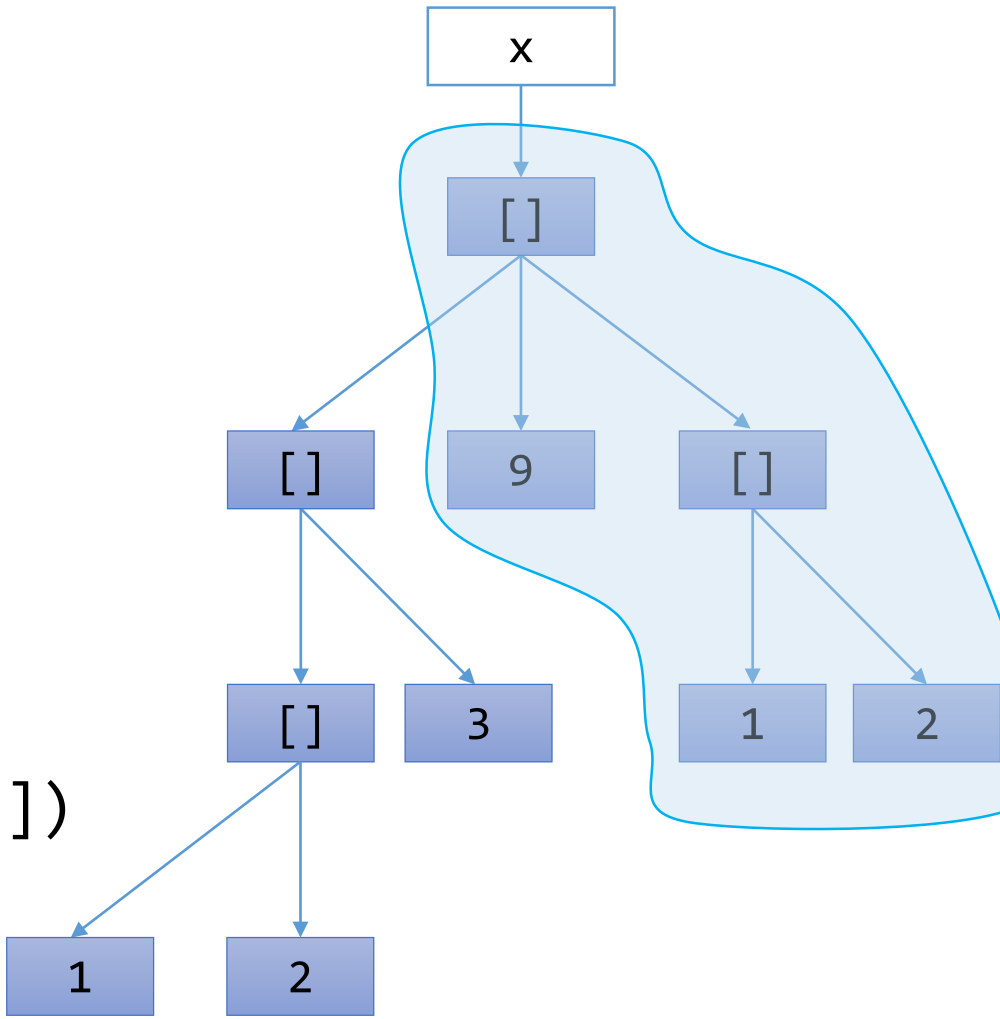
```
deepcount([9,[1,2]])

deepcount(9)+deepcount([[1,2]])
```

1

```
        deepcount([1,2]) + deepcount([])

deepcount(1) + deepcount([2])
```

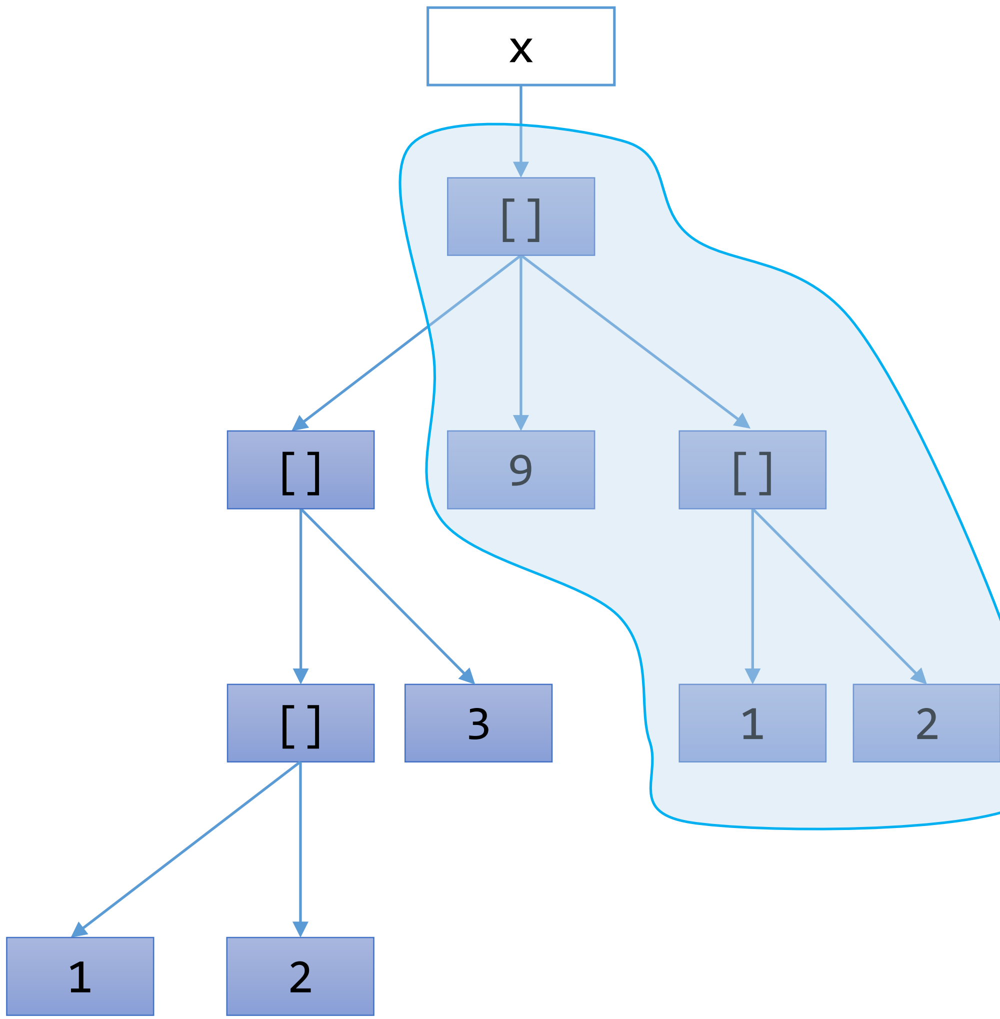
1

```
         deepcount 1 + deepcount([])
```



Tracing the Code

`deepcount([9,[1,2]]) ☞ 3`

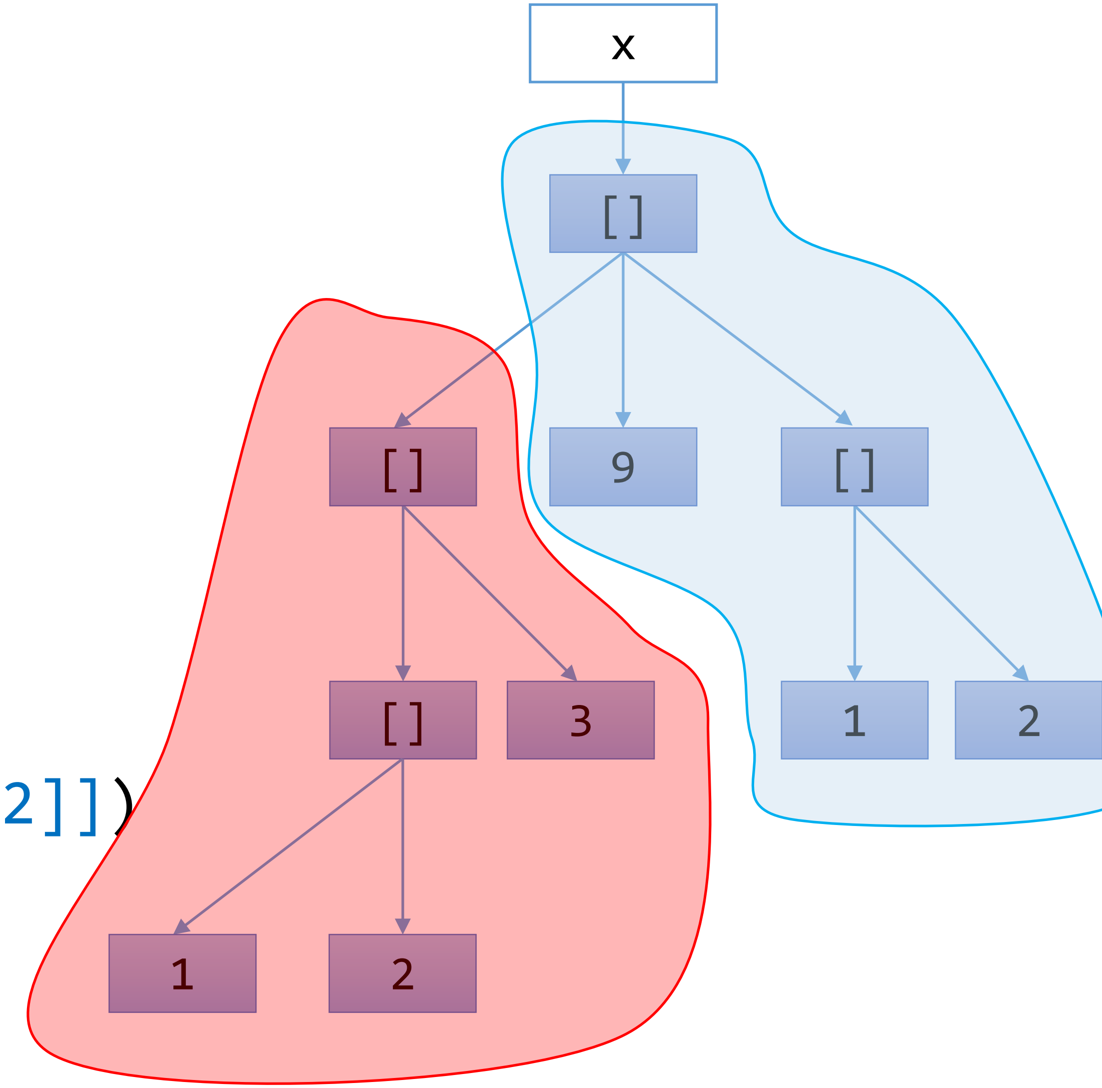


Tracing the Code

```
deepcount([[[1,2],3],9,[1,2]])
```

```
deepcount([[1,2],3]])+deepcount([9,[1,2]])
```

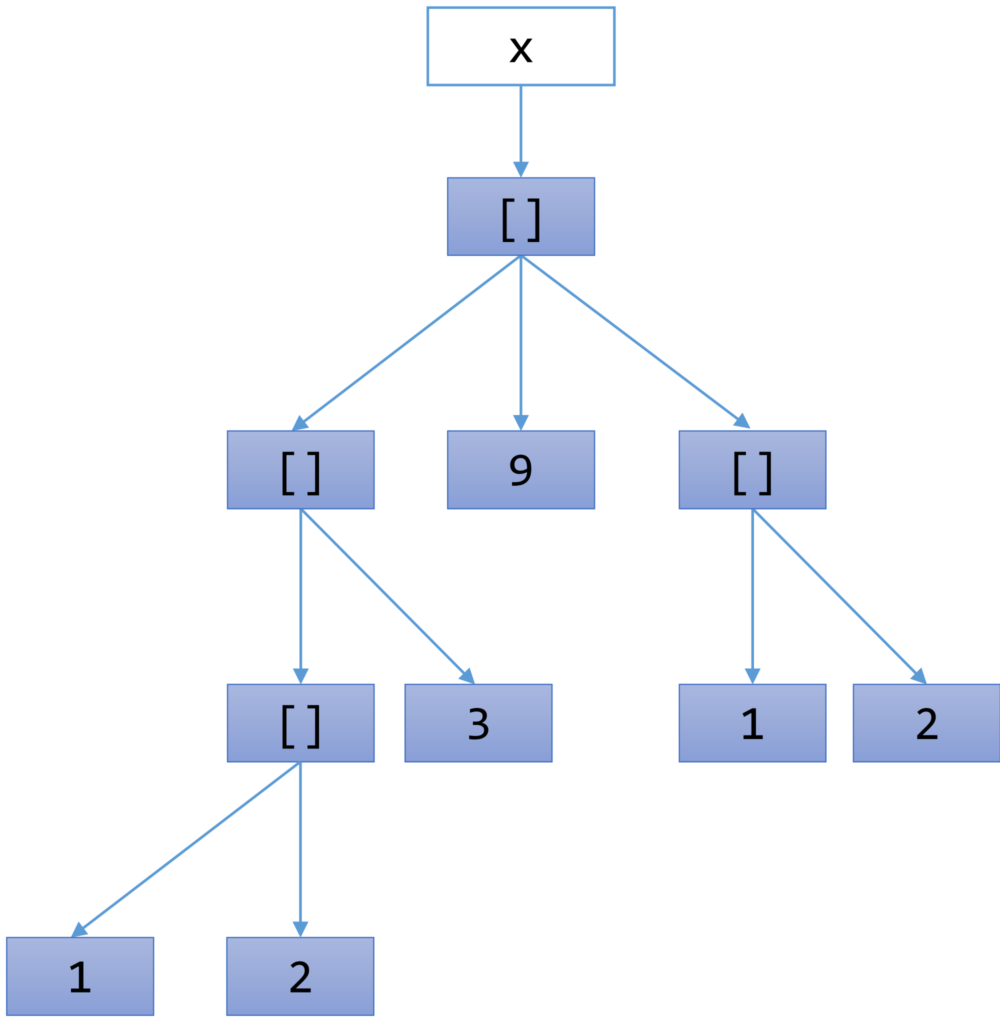
3 + 3 = 6



Tracing the Code

- x = [[[1,2],3],9,[1,2]]

```
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```



How about ….

- DeepSquare?

```
>>> l = [1, 4, 9, [1, 4], [4, 9, 16, [1, 4, 9]], [9, 16, 25]]
>>> deepSquare(l)
[1, 16, 81, [1, 16], [16, 81, 256, [1, 16, 81]], [81, 256, 625]]
```

- Deep Increment (by 1)?

```
>>> deepInc(l)
[2, 3, 4, [2, 3], [3, 4, 5, [2, 3, 4]], [4, 5, 6]]
>>> deepInc(deepInc(deepInc(l)))
[4, 5, 6, [4, 5], [5, 6, 7, [4, 5, 6]], [6, 7, 8]]
```

Get Some Insight from DeepCount?

```python
def deepcount(seq):
    if seq == []:
        return 0
    elif type(seq) != list:
        return 1
    else:
        return deepcount(seq[0])+ deepcount(seq[1:])
```

DeepSquare

- Spot the difference?

```
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])]+ deepSquare(seq[1:])
```

DeepSquare

- Base case is different
  - If seq is reduced to an empty list, return it as it is

```
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])]+ deepSquare(seq[1:])
```

DeepSquare

- The leaf case
  - If the item is not a list, return its square instead of 1

```
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])]+ deepSquare(seq[1:])
```

DeepSquare

- Otherwise
  - Return the list of recursive call of the first item
    - Huh? Why not return the "recursive call of the first item"? Why an extra "layer" of list?
  - And concatenate with the recursive call of the "rest" of the list

```
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])]+ deepSquare(seq[1:])
```

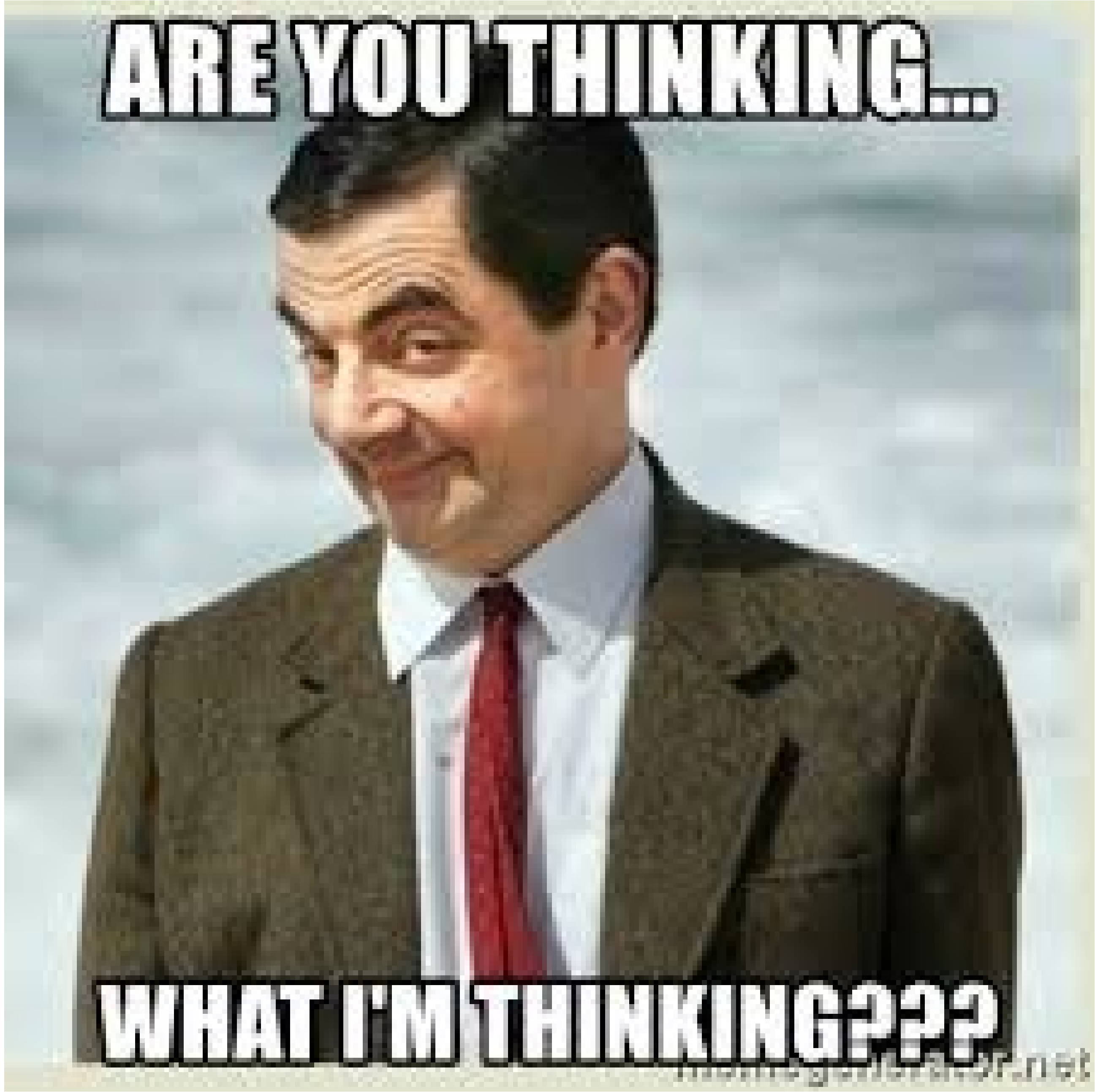
```
deepSquare([2])

[deepSquare(2)] + deepSquare([])

   [2*2]        +        []          ☞          [4]
```

```python
def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])]+ deepSquare(seq[1:])
```

DeepInc

- How different from DeepSquare?

```
def deepInc(seq):
    if seq == () or seq == []:
        return seq
    elif type(seq) != list:
        return seq+1
    else:
        return [deepInc(seq[0])]+ deepInc(seq[1:])
```


ARE YOU THINKING...
WHAT I'M THINKING???

```python
def deepInc(seq):
    if seq == () or seq == []:
        return seq
    elif type(seq) != list:
        return seq+1
    else:
        return [deepInc(seq[0])]+ deepInc(seq[1:])

def deepSquare(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return seq*seq
    else:
        return [deepSquare(seq[0])]+ deepSquare(seq[1:])
```

deepMap !!!!

```
def deepMap(func,seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return func(seq)
    else:
        return [deepMap(func,seq[0])]+ deepMap(func,seq[1:])
```

deepMap!!!

```
>>> l = [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
>>> deepMap(square,l)
[1, 4, 9, [1, 4], [4, 9, 16, [1, 4, 9]], [9, 16, 25]]
>>> deepMap(str,l)
['1', '2', '3', ['1', '2'], ['2', '3', '4', ['1', '2', '3']],
['3', '4', '5']]
>>> deepMap(lambda x:x/2,l)
[0.5, 1.0, 1.5, [0.5, 1.0], [1.0, 1.5, 2.0, [0.5, 1.0, 1.5]],
[1.5, 2.0, 2.5]]
```

Remember List Copy by copy()?

```
>>> l2 = l.copy()
>>> l2
[1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
>>> l[3][0] = 999
>>> l2
[1, 2, 3, [999, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
>>> l
[1, 2, 3, [999, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
```

- Shallow copy!!!!!!!! (Please refer to tutorials)

deepCopy()

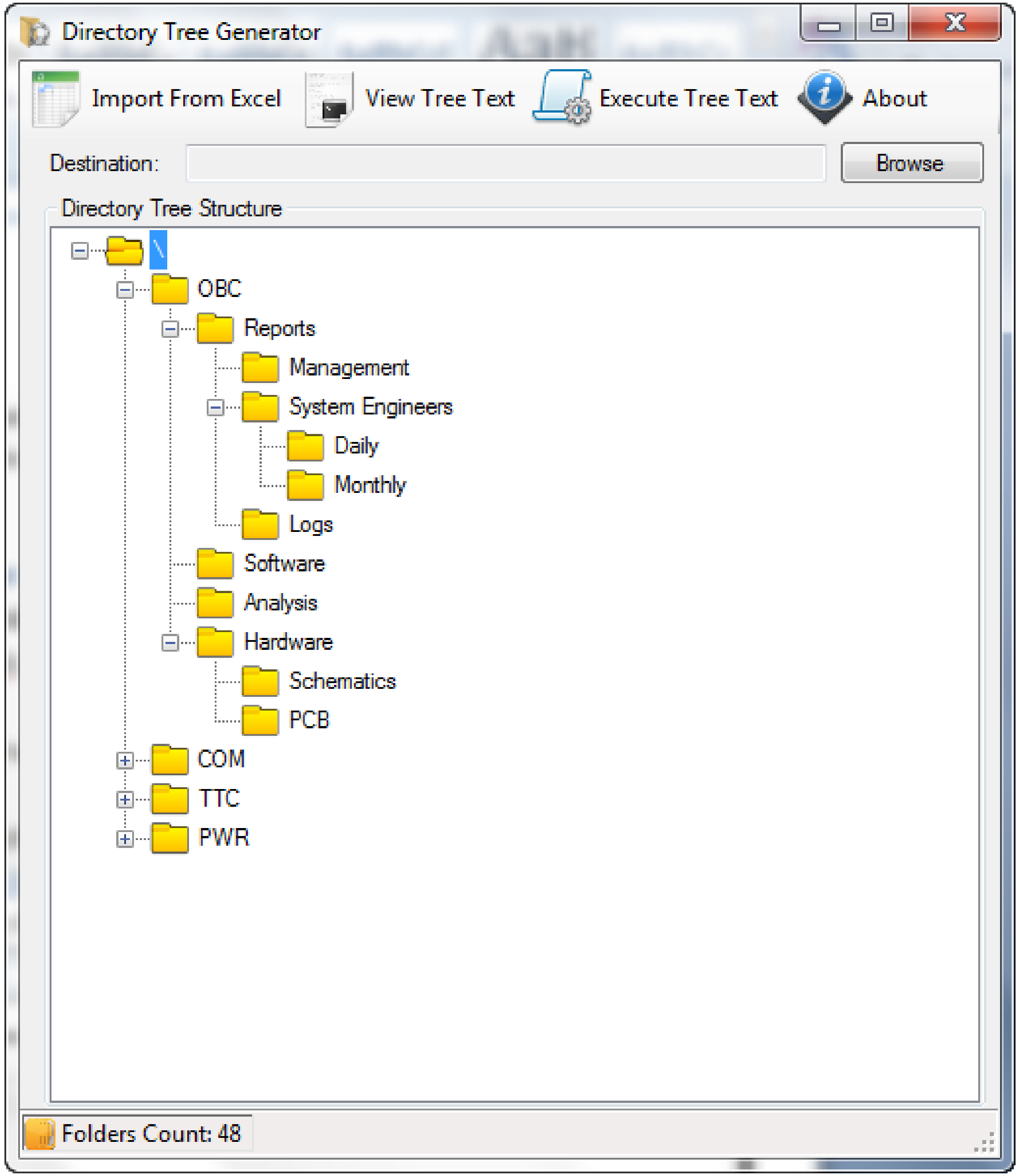
```
>>> l1 = [1,2,3,[1,2],[2,3,4,[5,6]]]
>>> l2 = deepMap(lambda x:x, l1)
>>> l1[3][0] = 999
>>> l1
[1, 2, 3, [999, 2], [2, 3, 4, [5, 6]]]
>>> l2
[1, 2, 3, [1, 2], [2, 3, 4, [5, 6]]]
```

- And it works!

Why Do I Want to Go “Deep”?

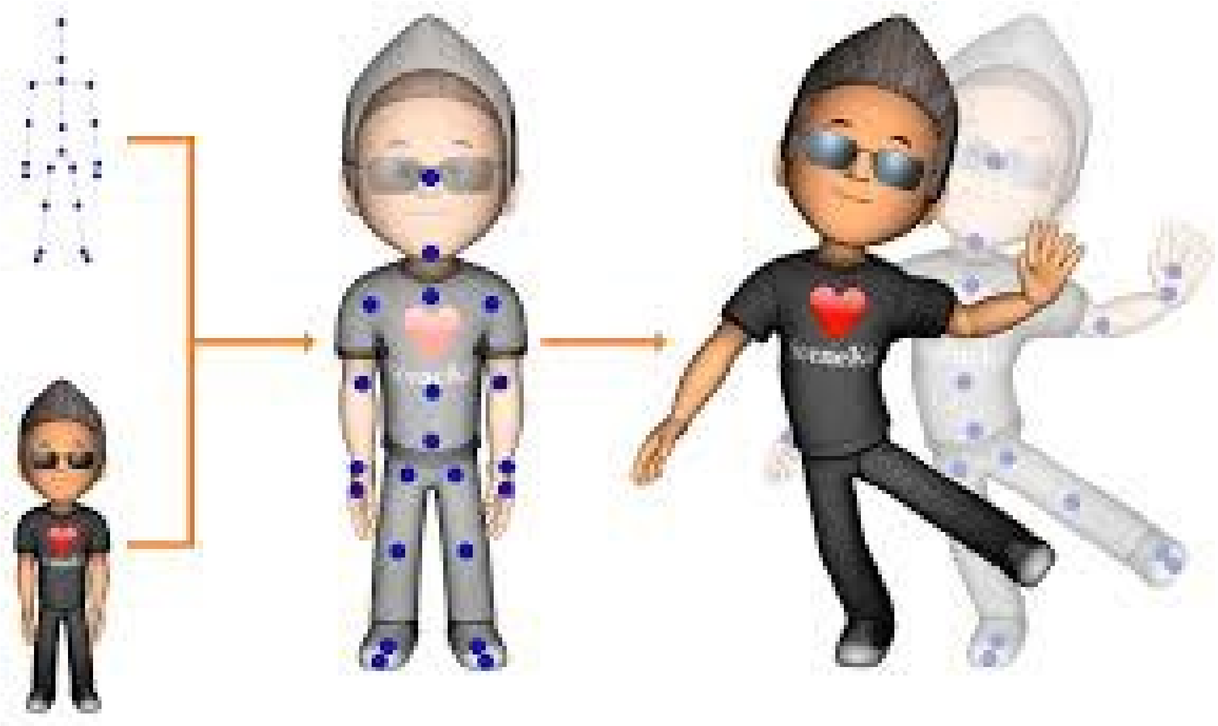
Copying a Directory

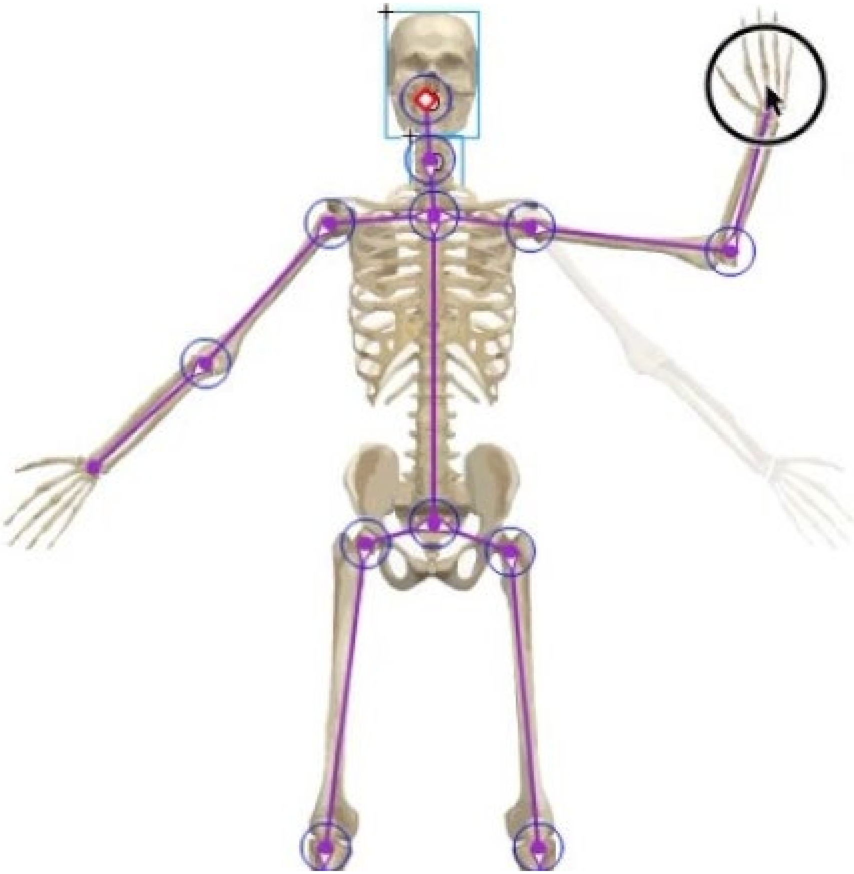
- When the directory contains a lot of files in many subdirectories



Computer Animation

- Skeleton animation





Shortest Path Tree

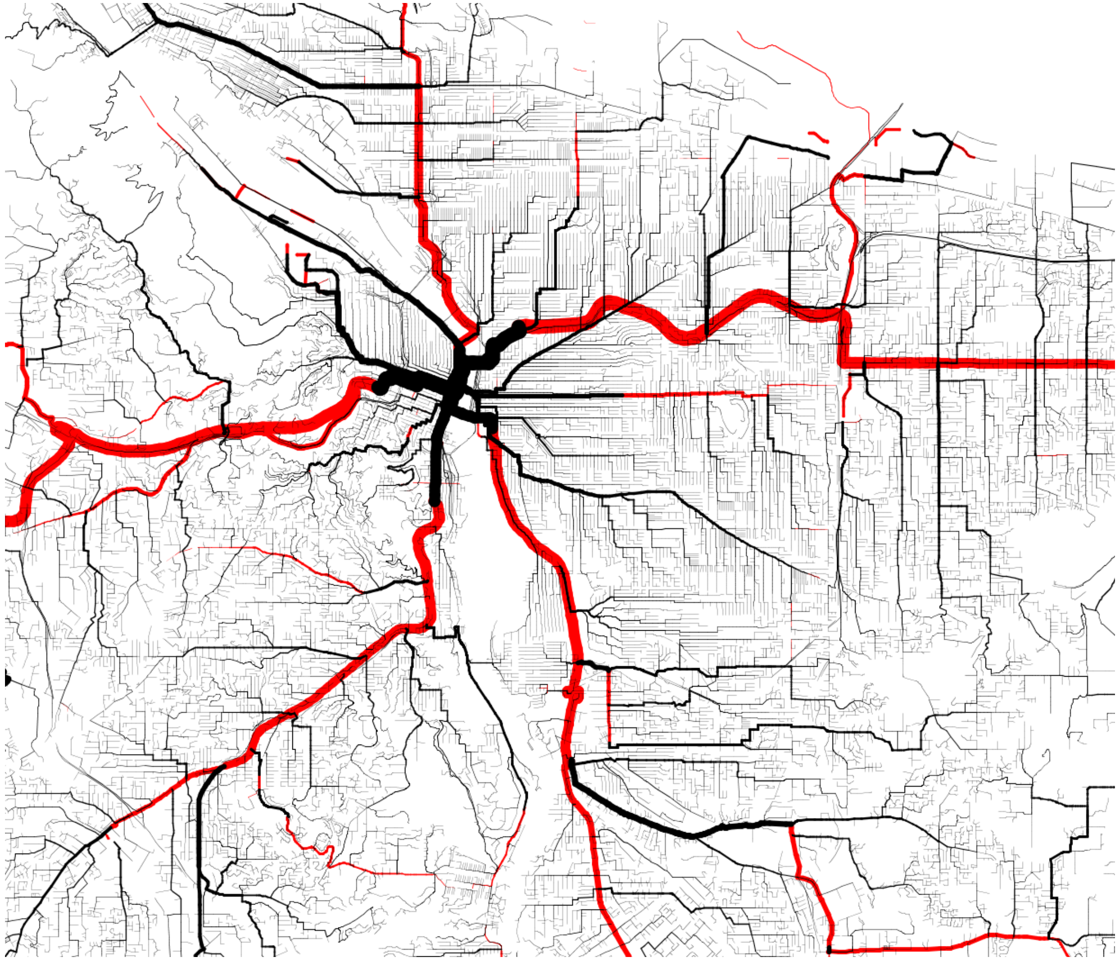


Image Processing

- An image is a list of lists of lists
  - The first level list: rows
  - The second level of lists: column
  - The third level of lists: RGB values
- Map a function to change certain values
  - E.g. change colors

How about if I just want to be shallow?

How about if I just want to be shallow?

- Given a nested list, output a list with all the elements but without any

```
>>> l = [1, 2, 3, [1, 2], [2, 3, 4, [1, 2, 3]], [3, 4, 5]]
>>> flatten(l)
[1, 2, 3, 1, 2, 2, 3, 4, 1, 2, 3, 3, 4, 5]
```

Flatten()

```python
def flatten(seq):
    if seq == []:
        return seq
    elif type(seq) != list:
        return [seq]
    else:
        return flatten(seq[0])+ flatten(seq[1:])
```

```
flatten([[[1]]])
flatten([[1]]) + flatten([])
flatten([1]) + flatten([]) + flatten([])
flatten(1) + flatten([]) + flatten([]) + flatten([])
[1]        + flatten([]) + flatten([]) + flatten([])
```

Conclusions

- `map()` is a powerful tools in Python
  - Allows you to perform a lot of operations with less redundant code
- Deep operations are useful to solve problems with non-linear data
  - E.g. Trees, n-dim arrays, graphs
- Using recursive functions wisely is the key for algorithms
  - Higher level of coding