section 05

links.cs61a.org/jasonxu (password: mutation)

Recursion Review

some things are very repetitive…

you're on a ladder… climbing down from the 10th step and the 1st step is the same instruction… but at a different state

instruction:

at your current step, take a step down

subset sum

given a list of numbers, does *some* combination of the #s add to *k*?

example set:
(1, 2, 3)

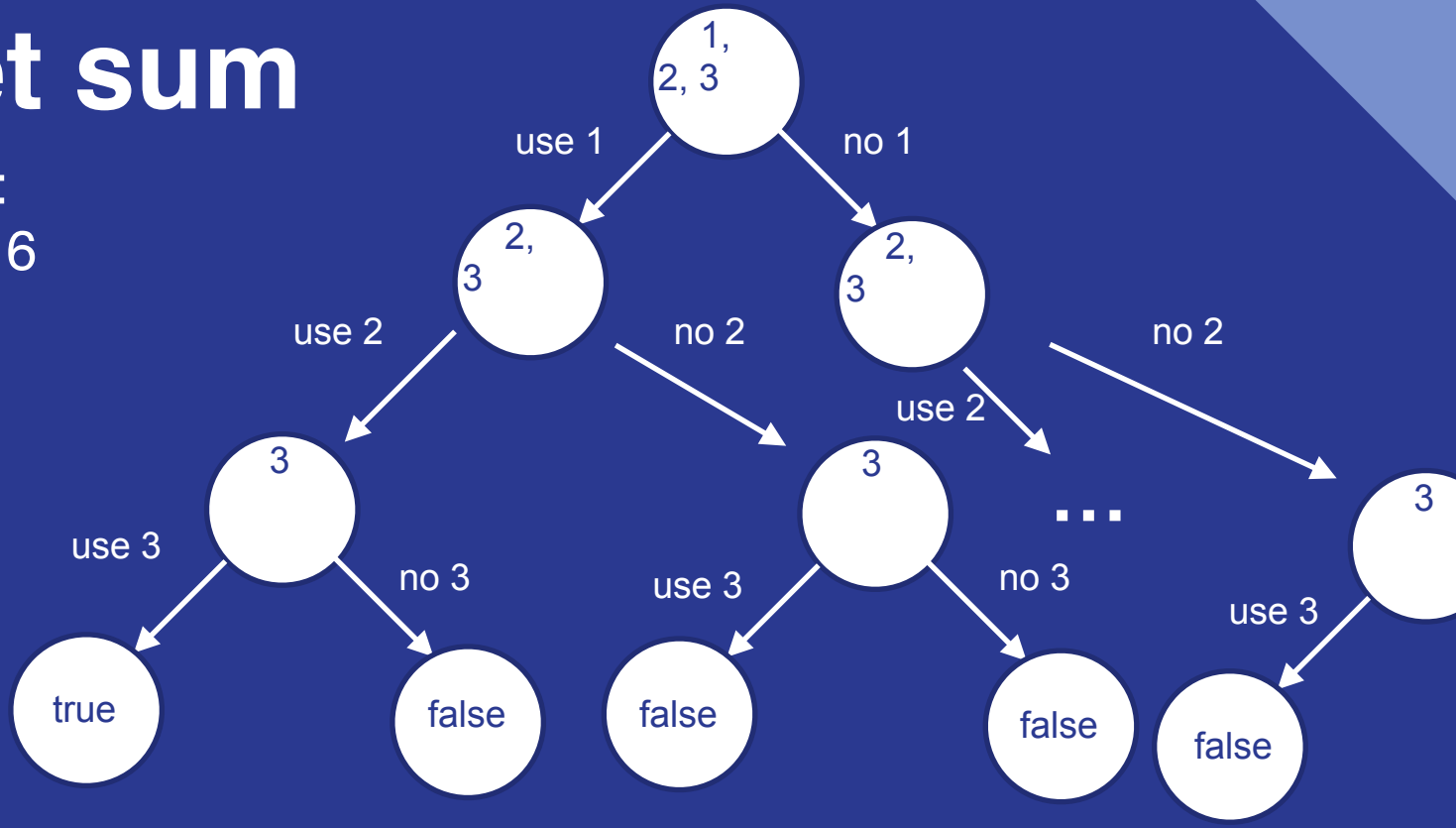
how to enumerate every subset (aka powerset)

(in, in, in), (in, in, out), (in, out, out), (out, in, in),
(out, out, in), (out, out, out), (in, out, in), (out, in, out)
or
{(in) + subsets((2, 3)), (out) + subsets((2, 3))}

look at the curr number, it is *either* in or out
we can always make a sum of 0 with any set?

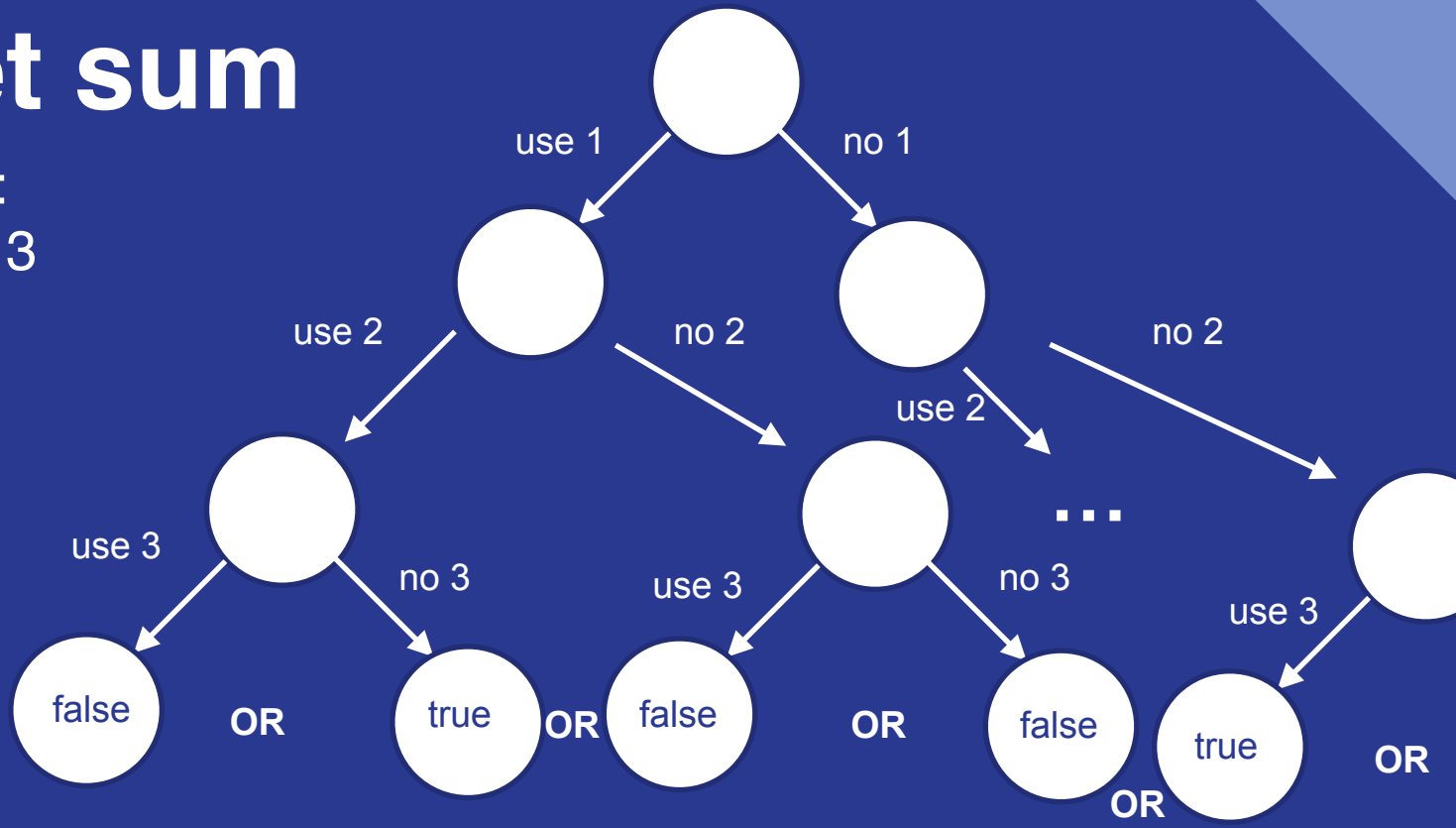
subset sum

example set:
(1, 2, 3), k = 6



subset sum

example set:
(1, 2, 3), k = 3



subset sum

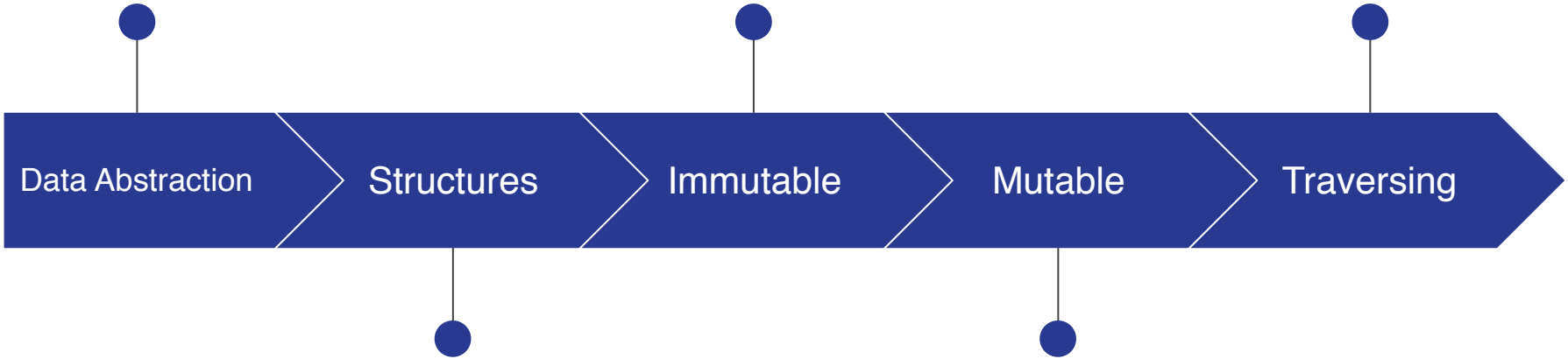
```
if k == 0:
    return True
elif lst == []:
    return False
else:
    return subset_sum(lst[1:], k - lst[0]) or \
           subset_sum(lst[1:], k)
```

How to get information out of abstractions

Copying, slicing, new abstractions

Iterators!

Generators!



List abstraction

Tree abstraction

‘ordered’ objects

Pointers!

Append, extend, pop

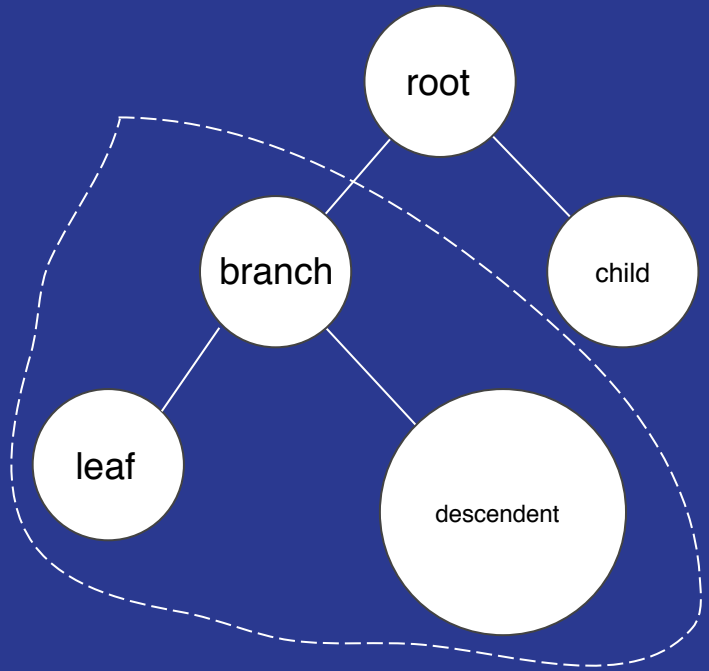
nonlocal

Trees

Recursive!

Abstraction barrier!

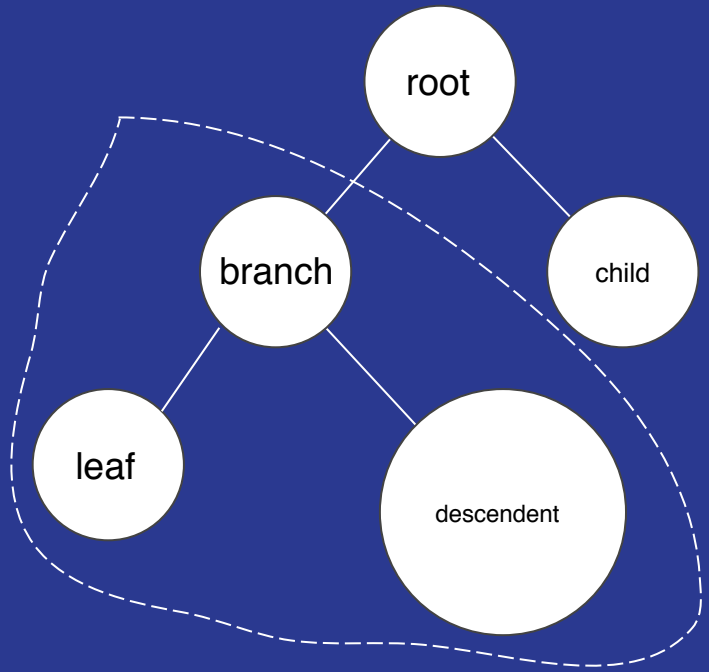
Will come back next week



List comprehension to apply recursive function onto each subtree… same thing as tree recursion except now physical representation.

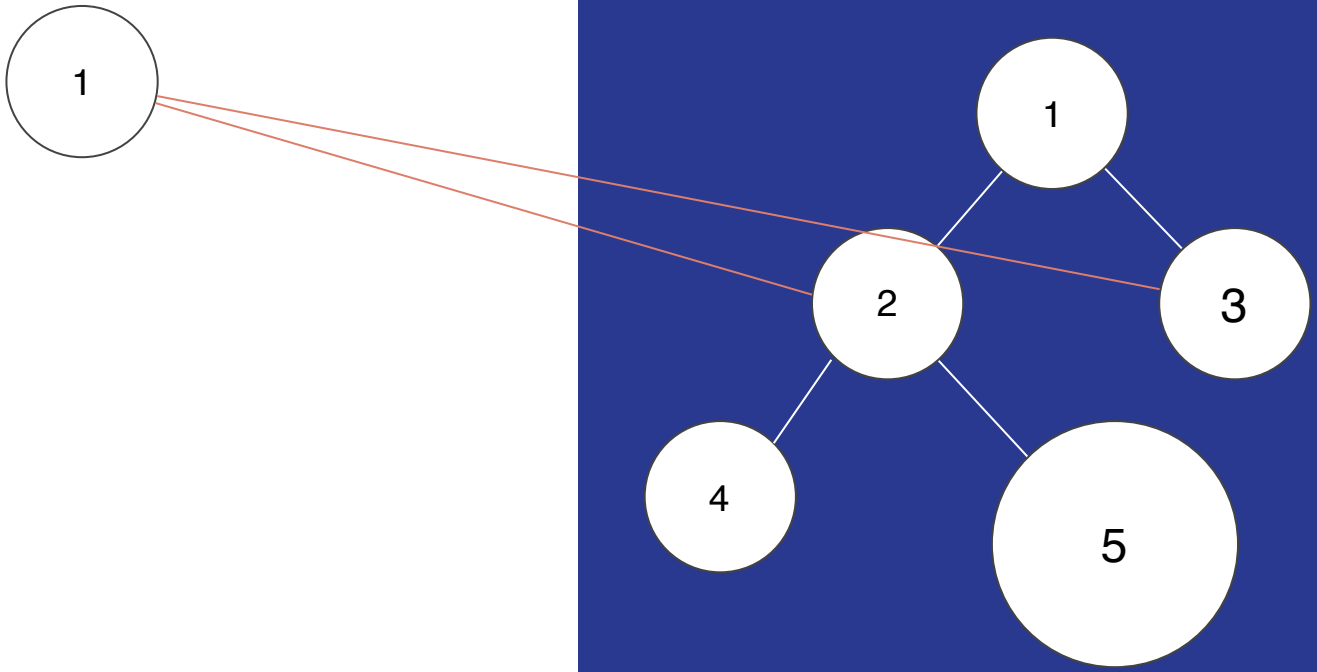
Can do so since branch is a ***'smaller'*** problem

```
def dumb_tree_func(t):
    if is_leaf(t):
        return tree(label(t))
    return tree(label(t), \
    [b for b in branches(t)])
```



List comprehension to apply recursive function onto each subtree… same thing as tree recursion except now physical representation.

Can do so since branch is a ***'smaller'*** problem



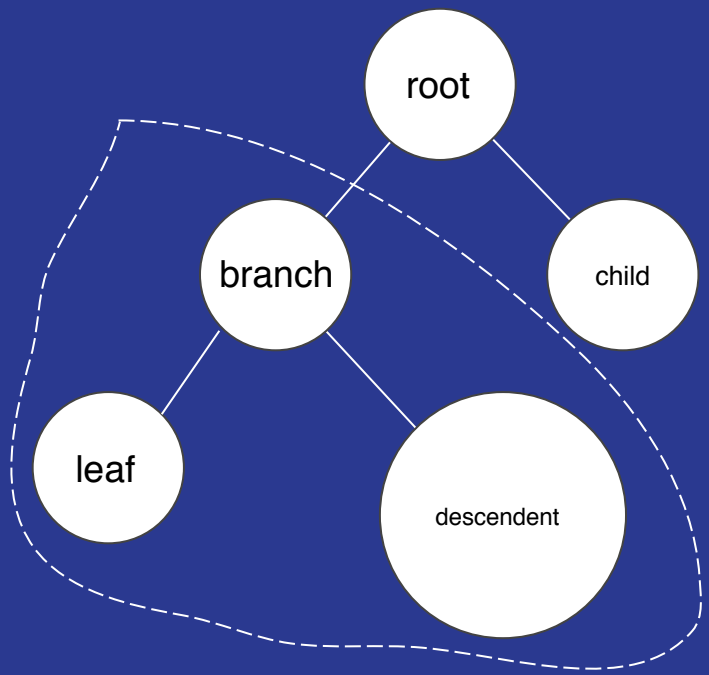
`dumb_tree_func(t):`

This tree gives you the same tree values, but it isn't a copy

Tree t

```
def dumb_tree_func(t):
    if is_leaf(t):
        return tree(label(t))
    return tree(label(t), \
    [d_t_func(b) for b in branches(t)])
```

Yay! Correctly copies a tree!

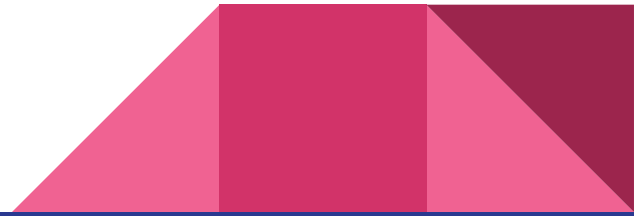


List comprehension to apply recursive function onto each subtree… same thing as tree recursion except now physical representation.

Can do so since branch is a ***'smaller'*** problem

Back in time

Uhhhhhh… what do the arrows mean in environment diagrams?



Back in time

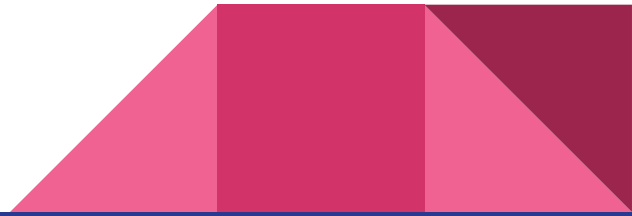
Probably heard me say ‘pointer’... time to formalize

Space efficiency, dealing with only 1 object, why make 100 copies?

Create a reference pointer to that object instead of duplicating

Useful now? **Lists are objects!**

`==` means *value-equality*
`is` means *object-equality*
`len(lst)` returns size of list

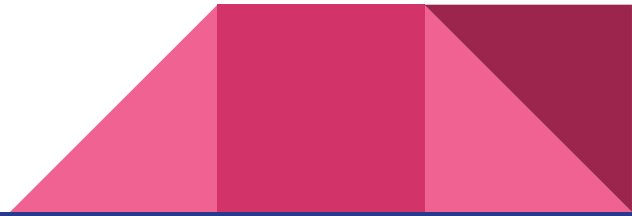


Back in time

```
lst = [1, 2, 3, [5, 6]]
#object is [1, 2, 3, [5, 6]], lst is a pointer!
#4th element is a pointer to [5, 6] object

a = lst
#a points to the same object as lst, not to lst
#a is lst

lst[0] = [5, 6]
#a is [[5, 6], 2, 3, [5, 6]]
```



So how do I make a copy?

I can’t just do an assignment! Why?

What does it mean to copy?

Shallow

Artificial copy, including copy of pointers, not objects just values

Deep

Copies values and objects!
recursion!

Easy: how?

Shallow

Artificial copy, including copy of pointers, not objects just ‘values’

Bit more involved… how do we copy trees?

Deep

Copies values and objects! recursion!

List Slicing -- **new *shallow* copy**

lst[*<start>*:*<end>*:*<step>*], default[0: len(lst), 1]

Bounds: [start, end)
No such thing as an invalid bound: returns **empty** list, **not error** if incorrect

Determine start & end depending on ***sign*** of step,

```
do slicing and return new list
```

Easy: how?

Shallow

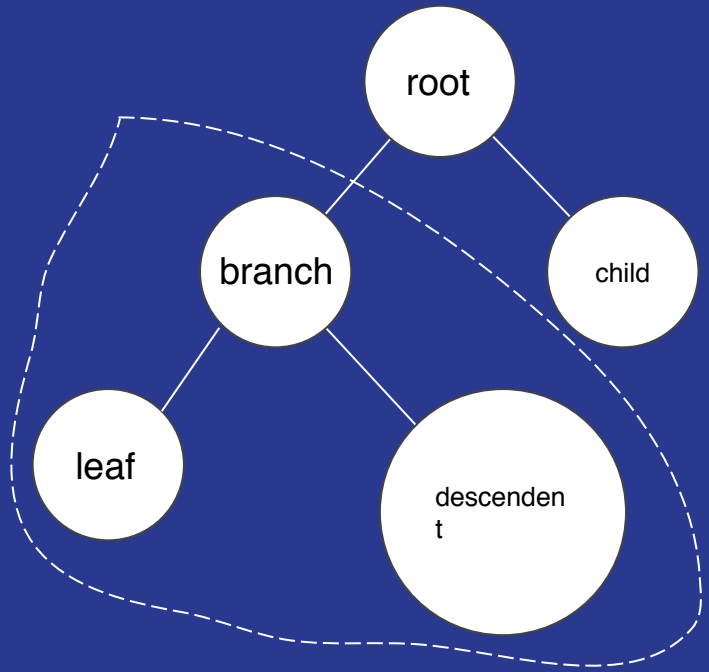
Artificial copy, including copy of pointers, not objects just values

Bit more involved… how do we copy trees?

Deep

Copies values and objects! recursion!

```
def dumb_tree_func(t):
    if is_leaf(t):
        return tree(label(t))
    return tree(label(t), \
    [d_t_func(b) for b in branches(t)])
```

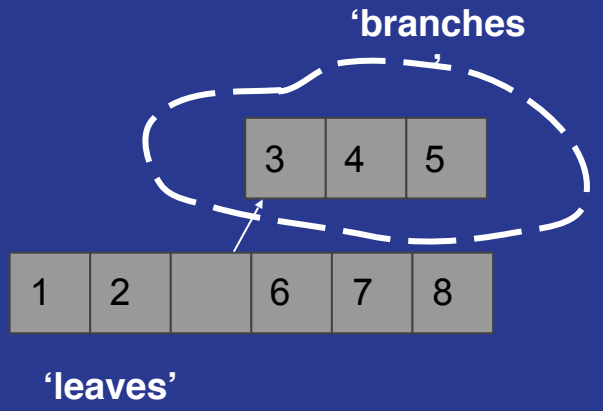


List comprehension to apply recursive function onto each subtree… same thing as tree recursion except now physical representation.

Can do so since branch is a ***'smaller'*** problem

What happens if i don't put brackets here?

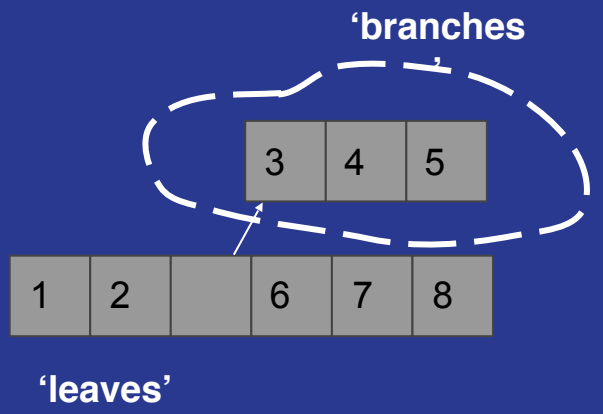
```
def dumb_list_func(lst):
    new_lst = []
    for item in lst:
        if is_list(item):
            new_lst += [dlf(item)]
        else:
            new_lst += [item]
     return new_lst
```



```
def list_copy(lst):
    if not is_list(lst):
        return lst
    return [lc(item) for item in lst]
```

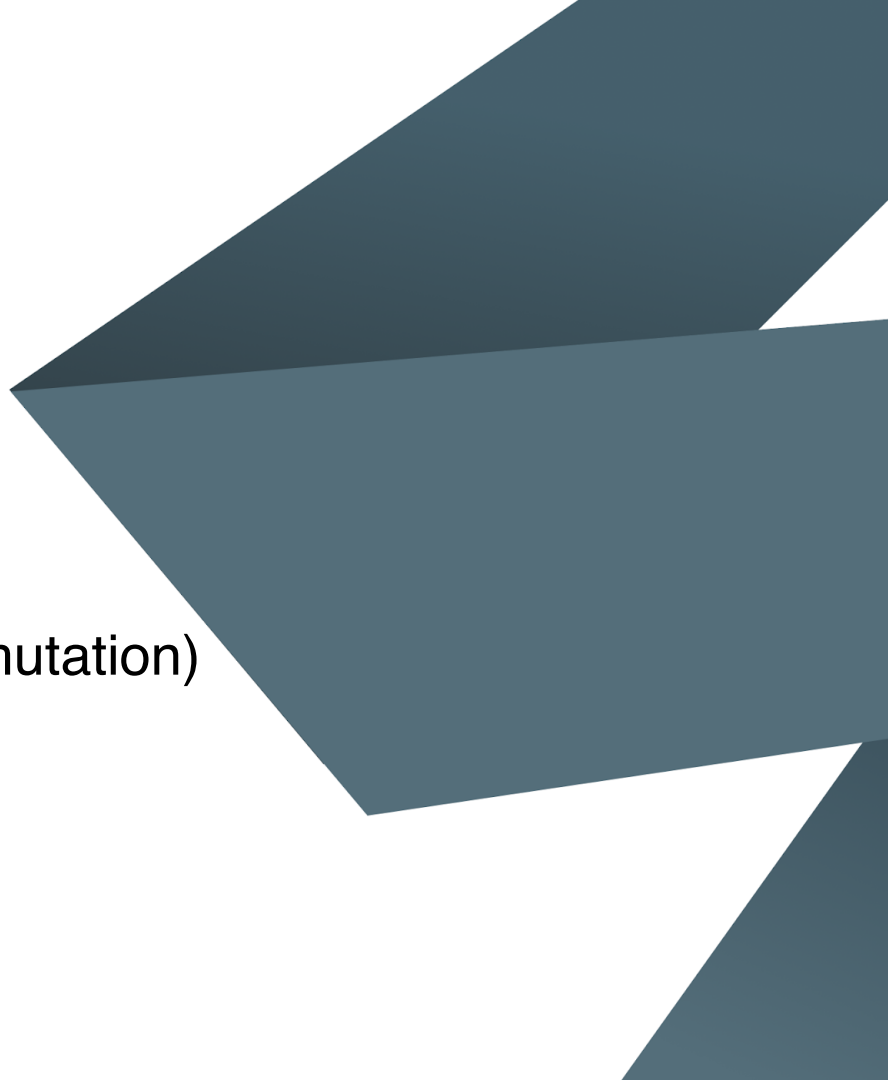
terrible design… why?

Look at return type



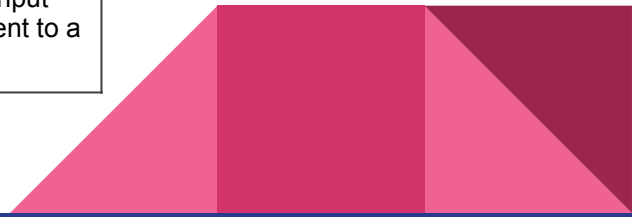
Break time?

links.cs61a.org/jasonxu (password: mutation)



List Operations

operation	domain	range	what it does
append()	any element can be string or list	None	Adds exactly one extra element to the list; uses pointer if the input doesn't “fit”
extend()	list	None	Mutates (puts all elements from the list and adds it directly to the end of the original list)
+= ***	lst	None	Mutates
lst = lst + otherlst	lst	None	Makes new list, assigns to lst
list()	iterable	List	Iterates through the input and adds each element to a (newly-made) list



Lists

```
lst = [1, 2, [3, 4, 5]]
```

```
>>>lst
[1, 2, [3, 4, 5]]
>>>lst[:]
[1, 2, [3, 4, 5]]
>>>a = lst[:]
>>>lst[1] = 9
>>>lst
[1, 9, [3, 4, 5]]
>>>a
[1, 2, [3, 4, 5]]
>>>a[2][2] = -1
>>>a
[1, 2, [3, 4, -1]]
>>>lst
[1, 9, [3, 4, -1]]
```

Lists

```
lst = [1, 2, 3, 4, 5]
```

```
>>>lst.extend(5)
Error
>>>lst.extend([5, 6])
>>>lst
[1, 2, 3, 4, 5, 5, 6]
>>>lst.extend((6, 6, 7))
[1, 2, 3, 4, 5, 5, 6, 6, 6, 7]
>>>lst = lst[5::2]
>>>lst
[5, 6, 7]
>>>lst.extend({'hi': 2, '1': 1})
>>>lst
[5, 6, 7, 'hi', '1']
>>>a = lst.append(10)
>>>a
>>>lst.append([100])
>>>lst
[5, 6, 7, 'hi', '1', 10, [100]]
```

Lists

.pop(<*i*>) removes and returns index i element.

```
>>>lst.pop()
5
>>>lst
[1, 2, 3, 4]
>>>lst.pop(0)
1
>>>lst
[2, 3, 4]
>>>a = lst
>>>a
[2, 3, 4]
>>>a = a + [lst.pop(2)]
>>>a
[2, 3, 4] #not [2, 3, 4, 4],
why?
>>>lst
[2, 3]
```

List Operations -- pt. 2

operation	domain	range	what it does
insert(i, x)	Index (if over, insert last, if under insert first) [negative indexing ok!], element	None	Adds exactly one extra element to the list at index i
remove(x)	item	None, Error	Takes out first instance of x in list, throws error otherwise

Tuples are immutable…

so are…?

This is why we erase certain values in environment diagrams

nonlocal

unbound local error

scoping -- essentially what my current frame can see, access, modify

```
def f():
        x = 5
    def g():
        print(x)
        x += 1
        return ‘success’
    return g()

def check():
        print(x)

>>>f()
Error
>>>f()
Error
```

nonlocal

unbound local error
RESOLVED

Notice placement of nonlocal

```
def f():
        x = 5
    def g():
        nonlocal x
        print(x)
        x += 1
        return ‘success’
    return g()

def check():
        print(x)

>>>f()
5
‘success’ #caution: return
value!
>>>f()
6
‘success’
```

global

explore a little!
same idea, though
exercise for the reader

```
x = 5
def f()
     global x
     print(x)
     x += 1
     return ‘success’

def check():
          print(x)

>>>f()
5
‘success’ #caution: return
value!
>>>check()
6
```