# CS61A Midterm 2 Review

Eric Atkinson
Kushal Ranjan
Victor Sutardja
Sagar Karandikar

Akshay Ramachandran
Chris Hall
Joy Jeng
Mark Miyashita

# Hello!

Because not all of the presenters are affiliated with the course, the standard disclaimer applies: **This review session is not officially endorsed by the course staff.**

All statements made by any course staff member involved in this session are made on behalf of HKN and not as a representative of the course.

Also, please fill out a feedback form on your way out.

# Topics

- Lists / Tuples, HOFs on sequences
- Recursion
- Nonlocal
- OOP
- Rlists
- Data Abstraction

# Iterables

- Lists - Sequences that are **mutable.** We can add, remove, and change the items of a list.

- Tuples - Sequences that are **immutable.** We **cannot** change the items in a tuple, only create new ones.

- Dictionaries - Stores data by mapping keys to values. Remember that they are unordered and the keys are unique!

- Range - Gives an object that represents an interval of elements between the input values that we can iterate over.

# List Comprehensions

```
[<result> for <item> in <thing to iterate over> if <condition>]
```

```
>>> [x for x in range(5) if x > 3]
[4]
```

List comprehensions give us a way to write a simple for loop and optional if statement in one line. For example, the above snippet is the same as:

```
>>> result = []
>>> for x in range(5):
...    if x > 3:
...        result.append(x)
...
>>> result
[4]
```

# List Comprehensions Example

```
>>> words = "I love CS61A!".split()
>>> words
['I', 'love', 'CS61A!']
>>> [len(w) for w in words]

_____
>>> [[w]*len(w) for w in words]

_____
```

# List Comprehensions Example

```
>>> words = "I love CS61A!".split()
>>> words
['I', 'love', 'CS61A!']
>>> [len(w) for w in words]
[1, 4, 6]
>>> [[w]*len(w) for w in words]
_____
```

# List Comprehensions Example

```
>>> words = "I love CS61A!".split()
>>> words
['I', 'love', 'CS61A!']
>>> [len(w) for w in words]
[1, 4, 6]
>>> [[w]*len(w) for w in words]
[['I'], ['love', 'love', 'love', 'love'], ['CS61A!',
'CS61A!', 'CS61A!', 'CS61A!', 'CS61A!', 'CS61A!']]
```

This is the same as:

```
[['I']*1, ['love']*4, ['CS61A!']*6]
```

# Higher Order Functions

- **Map** - Takes in a function and a sequence and applies the function to each of the elements of the sequence.
  - Input - function that takes in **one argument.**

    sequence that can be any iterable (list, tuple, etc.)
  - Output - sequence of the same length as the input.
- **Filter** - Takes in a function and a sequence and returns a new sequence with only the items for which the function returns True for.
  - Input - function that takes in **one argument.**

    sequence that can be any iterable (list, tuple, etc.)
  - Output - sequence that contains the elements that satisfy the function.
- **Reduce** - Takes in a function, a sequence, and an optional initial value and returns a single combined value. The result is accumulated as you iterate through the list.
  - Input - function that takes in **two arguments.**

    sequence that can be any iterable (list, tuple, etc.)

    (optional) starting value
  - Output - a single element that is found by combining the elements using the input function.

# Map and Filter

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> list(map(is_prime, filter(is_prime, fib)))


_____
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, filter(is_prime, fib)))


_____
```

# Map and Filter (cont.)

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> list(map(is_prime, ~~filter(is_prime, fib)~~))
                             [2, 3]

_____
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, filter(is_prime, fib)))


_____
```

# Map and Filter (cont.)

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> list(map(is_prime, ~~filter(is_prime, fib)~~))
                            [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, filter(is_prime, fib)))
```

_____

# Map and Filter (cont.)

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> list(map(is_prime, ~~filter(is_prime, fib)~~))
                            [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, ~~filter(is_prime, fib)~~))
                            [2, 3]
```

# Map and Filter (cont.)

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> list(map(is_prime, ~~filter(is_prime, fib)~~))
                           [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, ~~filter(is_prime, fib)~~))
                           [2, 3]
# intermediate step [get_fib(2), get_fib(3)]
```

# Map and Filter (cont.)

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> list(map(is_prime, ~~filter(is_prime, fib)~~))
                           [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, ~~filter(is_prime, fib)~~))
                           [2, 3]
# intermediate step [get_fib(2), get_fib(3)]
# intermediate step [fib[2], fib[3]]
```

# Map and Filter (cont.)

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> list(map(is_prime, ~~filter(is_prime, fib)~~))
                           [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, ~~filter(is_prime, fib)~~))
                           [2, 3]
# intermediate step [get_fib(2), get_fib(3)]
# intermediate step [fib[2], fib[3]]
[1, 2]
```

# Reduce

Given a list such as [1, 2, 3, 4, 5, 6], we want to reduce the list down to a single number which is the 'flattened' version of the list. The output for this particular list would be the number 123456.

```
>>> from functools import reduce
>>> t = [1, 2, 3, 4, 5, 6]
```

_____

# Reduce

Given a list such as [1, 2, 3, 4, 5, 6], we want to reduce the list down to a single number which is the 'flattened' version of the list. The output for this particular list would be the number 123456.

```
>>> from functools import reduce
>>> t = [1, 2, 3, 4, 5, 6]
>>> reduce(lambda have, curr: have*10 + curr, t)
123456
```

How?

# Reduce

```
>>> t = [1, 2, 3, 4, 5, 6]
>>> reduce(lambda have, curr: have*10 + curr, t)
123456
```

First time through:

have = 1

curr = 2

# Reduce

```
>>> t = [1, 2, 3, 4, 5, 6]
>>> reduce(lambda have, curr: have*10 + curr, t)
123456
```

First time through:
have = 1
curr = 2
result = 1*10 + 2 = 12

# Reduce

```
>>> t = [1, 2, 3, 4, 5, 6]
>>> reduce(lambda have, curr: have*10 + curr, t)
123456
```

First time through:
have = 1
curr = 2
result = 1*10 + 2 = 12

Next iteration:
have = 12
curr = 3
result = 12*10 + 3 = 123

# Dictionaries

Dictionaries are a way to store data by mapping keys to values. We can access this data by the following methods.

```
d = {'a': 1, 'b': 4}
d['c'] = 3          # add the key 'c' with the value 3
d['b'] = 2          # update the value of 'b' to 2
list(d.values())# returns [1, 2, 3]
list(d.keys())      # returns ['a', 'b', 'c']
list(d.items()) # returns [('a', 1), ('b', 2), ('c', 3)]
```

Remember that dictionaries have unique keys!

(If I try to add a key that already exists, it overrides the previous value with the new one.)

# Dictionaries Example

We can use list comprehension to construct dictionaries.

```
>>> d = {k : v for k, v in [(x, y) for x in range(3)
for y in range(4)]}
```

**Remember that dictionary keys are unique!**

```
>>> d
```

______

# Dictionaries Example

We can use list comprehension to construct dictionaries.

```
>>> d = {k : v for k, v in [(x, y) for x in range(3)
for y in range(4)]}
```

**Remember that dictionary keys are *unique*!**

```
>>> d
{0: 3, 1: 3, 2: 3}          # lol wat.
```

# Dictionaries Example

We can use list comprehension to construct dictionaries.

```
>>> d = {k : v for k, v in [(x, y) for x in range(3) for y in range(4)]}
```

For reference the list is:

[(0, 0), (0, 1), (0, 2), **(0, 3)**, (1, 0), (1, 1), (1, 2), **(1, 3)**, (2, 0), (2, 1), (2, 2), **(2, 3)**]

We only care about the last instance of the key (bolded). Why?

# Dictionaries Example

We can use list comprehension to construct dictionaries.

```
>>> d = {k : v for k, v in [(x, y) for x in range(3)
for y in range(4)]}
```

For reference the list is:

[(0, 0), (0, 1), (0, 2), **(0, 3)**, (1, 0), (1, 1), (1, 2), **(1, 3)**, (2, 0), (2, 1), (2, 2), **(2, 3)**]

We only care about the last instance of the key (bolded). Why?

**Because dictionary keys are unique.**

# Recursion: Basics

- Recursion problems rely on two ideas:
  - Recursion problems have a simplest case (for which the solution is known)
  - A recursion problem can be reduced to its simplest case

# Recursion: Basics

- Recursion problems rely on two ideas:
  - Recursion problems have a simplest case (for which the solution is known)
  - A recursion problem can be reduced to its simplest case
- Recursive function: a function that may call itself
- Base case: simplest case(s) whose solution is known
- Recursive case: case in which problem must be reduced further

# Recursion

Identify the base case and give the recursive call.

1. Factorial(n), n>0

# Recursion

Identify the base case and give the recursive call.

1. Factorial(n), n>0
   a. Base: n==1 (or n == 2 or n < 2)
   b. Recursive: n*Factorial(n-1)

# Recursion

Identify the base case and give the recursive call.

1. Factorial(n), n>0
   a. Base: n==1 (or n == 2 or n < 2)
   b. Recursive: n*Factorial(n-1)
2. Search for an element in an RList

# Recursion

Identify the base case and give the recursive call.

1. Factorial(n), n>0
    a. Base: n==1 (or n == 2 or n < 2)
    b. Recursive: n*Factorial(n-1)
2. Search for an element in an RList
    a. Base: Element found OR next == None
    b. Recursive: Check next RList

# Recursion

3. Merge sort: Suppose you want to sort a list of n elements. Split the list into two halves, sort the two halves separately, and interleave the new sublists into a fully sorted list.

- Base: List is of length 2 (OR of length 1)
- Recursive: Split list in half, merge sort both halves.

# Recursion

Convert these iterative functions to recursive functions.

- Where would the base case condition appear in a while loop?
- What role does the body of the while loop play?

# Recursion

```
def foo(x, y):
    while y != 0:
        temp = y
        y = x % y
        x = temp
    return x
```

# Recursion

```
def foo(x, y):
   while y != 0:
      temp = y
      y = x % y
      x = temp
   return x
```

```
def foo(x, y):
   if y == 0:
      return x
   else:
      return foo(y,
         x%y)
```

Euclid's GCD Algorithm

# Recursion

```
def bar(n):
    a, b = 0, 1
    while n > 1:
        a = b
        b = b + a
        n = n - 1
    return b
```

# Recursion

```
def bar(n):
    a, b = 0, 1
    while n > 1:
        a = b
        b = b + a
        n = n – 1
    return b
```

```
def bar(n):
    if n == 1 or n == 2:
        return 1
    else
        return bar(n-1) +
        bar(n-2)
```

Fibonacci sequence with F (1) = 1 and F(2) = 1

# Recursion

Write a recursive function that takes two positive numbers, x and y, and returns the modulus of x by y. That is, mod(x, y) should return x%y. Do not use Python's built in division, floor division, or modulus operators.

```
>>> mod(5,2)
1
>>> mod(12,9)
3
>>> mod(24,3)
0
>>> mod(4,9)
4
>>> mod(18.5,5)
3.5
```

# Recursion

- Base case
  - $x < y$
  - return $x$
- Recursive case
  - So we have to decrease $x$ towards the base case WITHOUT changing the value of $x \% y$
  - We know that $x = ny + x\%y$, where $n$ is an integer
  - Subtract off $y$ until $x < y$

# Recursion

```
def mod(x,y):
    if y > x:
        return x
    else:
        return mod(x-y,y)
```

# Recursion

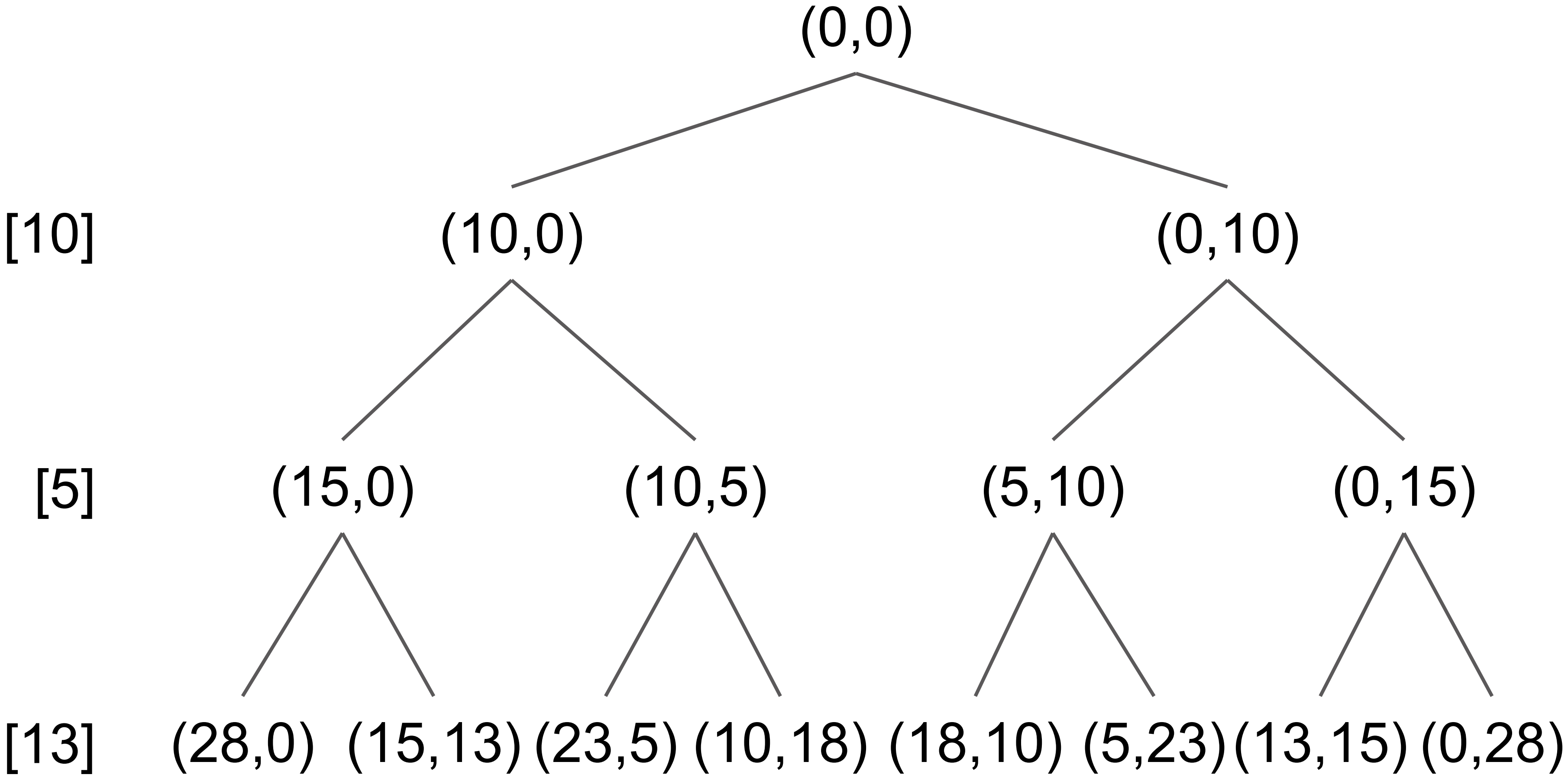
Write a function that, given a list of integers, returns True if and only if there is a way to place the integers into two groups such that the sum of one group equals the sum of the other. You may find it helpful to use a helper function.

```
>>> split([2,1,1]) # [2] and [1,1]
True
>>> split([2,1,2])
False
>>> split([10,5,13,6,12]) # [10,13] and [5,6,12]
True
>>> split([2,3,-1]) # [2] and [3,-1]
True
>>> split([4,5,6])
False
```

# Recursion

- Concept: Tree recursion
- Helper method
    - What are we interested in?
    - The SUM of the two lists
        - Actual elements in each list are not relevant
    - Parameters for the helper method?
        - What information is necessary to perform a single method call?

# Example: [10, 5, 13, 6, 12]



(0,0)

[10] (10,0) (0,10)

[5] (15,0) (10,5) (5,10) (0,15)

[13] (28,0) (15,13) (23,5) (10,18) (18,10) (5,23) (13,15) (0,28)

...and so on.

# Recursion

- Base case
    - Length of list is 0
        - True if left sum == right sum
        - False otherwise
- Recursive case
    - Add first element of list to each group in separate function calls
    - Call function using list excluding the first element
    - True if either function call returns True

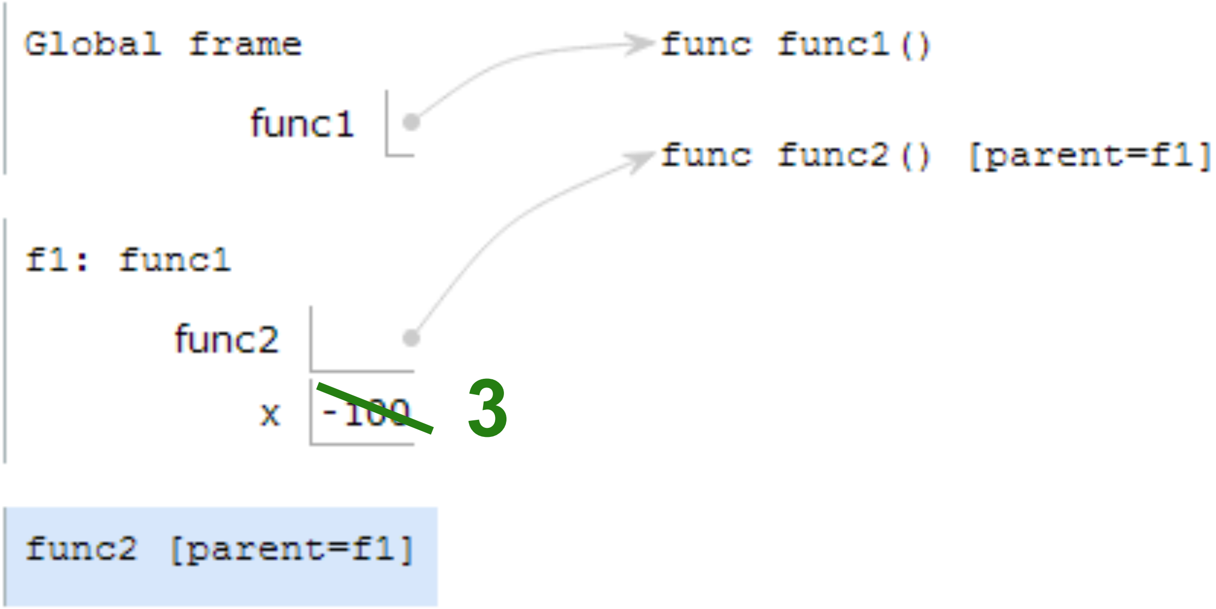
# Recursion

```
def split(s):
    def split_help(s, left, right):
        if len(s) == 0:
            if left == right:
                return True
            else:
                return False
        return split_help(s[1:],left+s[0],
        right) or split_help(s[1:],left,
        right+s[0])
    return split_help(s,0,0)
```

# Nonlocal in Environment Diagrams

```
def func1():
    x = -100
    def func2():
        nonlocal x
        x = 3
    func2()
func1()
```

If a variable is nonlocal, you must follow parents and look between (but not including) current frame and global



# Nonlocal in Environment Diagrams

```
x = 50
def func1():
    def func2():
        nonlocal x
        x = 3
    func2()
func1()
```

Does This Work?

No.

x is in the global frame

# Nonlocal in Environment Diagrams

```
def func1():
    def func2(x):
        nonlocal x
        x = 3
    func2(4)
func1()
```

Does This Work?

No.

nonlocal x, x in the same frame

# Nonlocal in Environment Diagrams

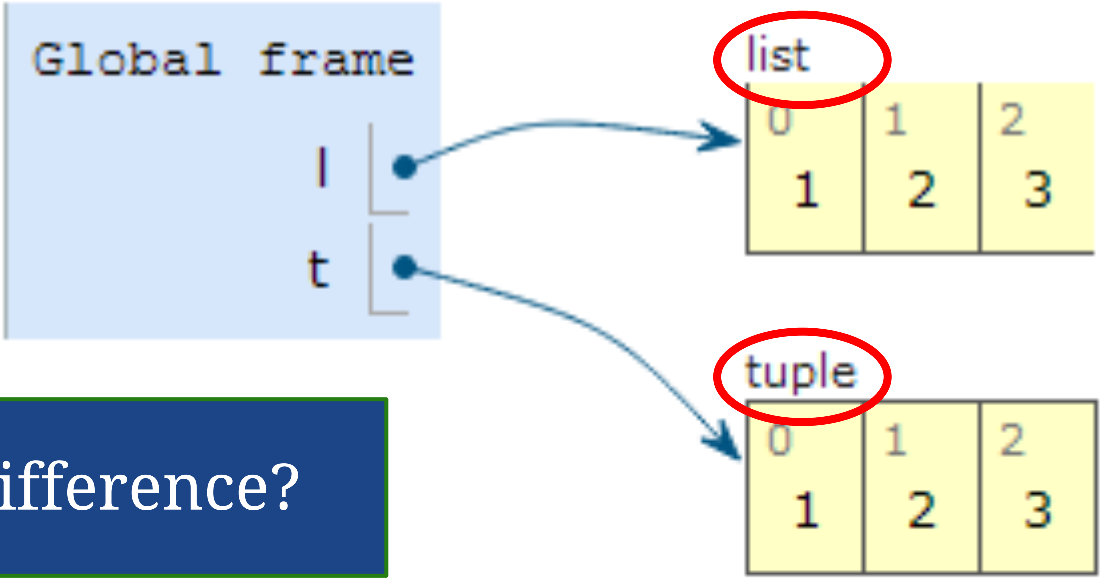
```
def func1():
    def func2():
        x = 4
        def func3():
            def func4():
                nonlocal x
                x = 3
            return func4()
        return func3()
    func2()
func1()
```

Does This Work?

Yes!

# Lists/Tuples in Environment Diagrams

```
>>> l = [1, 2, 3]
>>> t = (1, 2, 3)
```

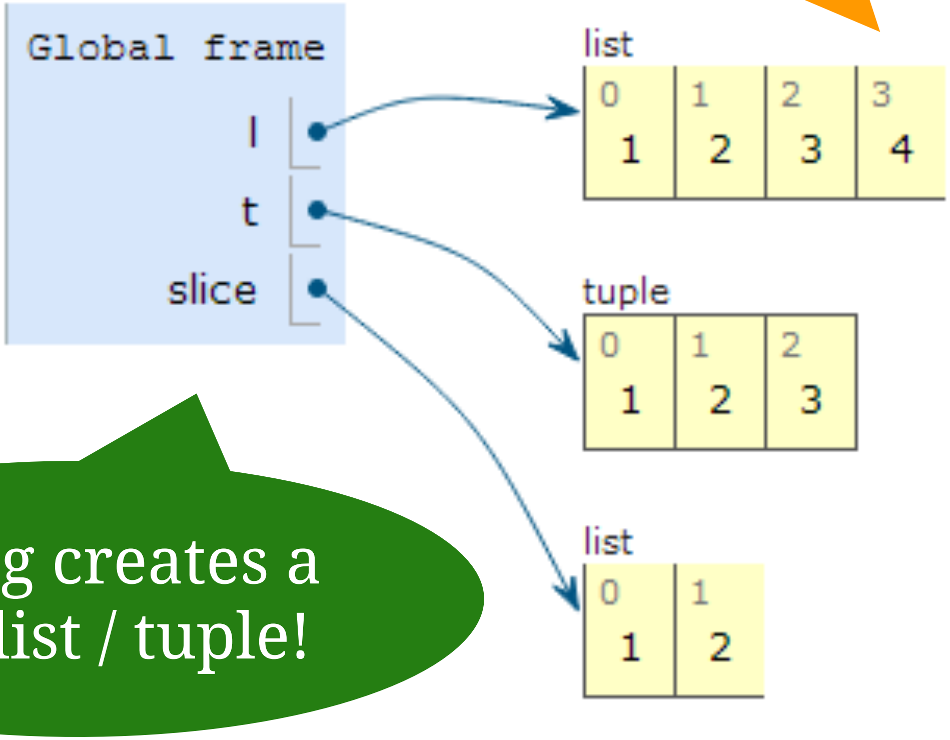


What's the difference?

# Lists/Tuples in Environment Diagrams

```
>>> l = [1, 2, 3]
>>> t = (1, 2, 3)
>>> slice = l[:2]
>>> l.append(4)
```

append() mutates the original list.



Slicing creates a new list / tuple!

# Draw the Environment Diagram

```
def a(x):
    def inner(y):
        x.extend(['<3','<3'])
        nonlocal x
        x = 5
        y[0][0] = 'cs61a'
        return y[0:2]
    return inner((x, 3, 6))

a(['cookies'])
```

# Environment Diagram Notes

```
def a(x):
    def inner(y):
        x.extend(['<3','<3'])
        nonlocal x
        x = 5
        y[0][0] = 'cs61a'
        return y[0:2]
    return inner((x, 3, 6))

a(['cookies'])
```

Don't need nonlocal to mutate something!

Need nonlocal to change what value a variable points to

Can't change what items a tuple points to, but you can mutate those values.

Slicing creates a new list with the same values.

# Nonlocal in Functions

```
def example(start):
    def add(x):
        nonlocal start
        start += x
        return start
    return add

>>> add = example(1)
>>> add(5)

>>> add(4)
```

# Nonlocal in Functions

```
def example(start):
    def add(x):
        nonlocal start
        start += x
        return start
    return add

>>> add = example(1)
>>> add(5)
6
>>> add(4)
```

# Nonlocal in Functions

```
def example(start):
    def add(x):
        nonlocal start
        start += x
        return start
    return add

>>> add = example(1)
>>> add(5)
6
>>> add(4)
10
```

# Nonlocal in Functions - Side Question

```
def example(start):
    def add(x):
        nonlocal start
        start += x
        return start
    return add

>>> example(1)(6)

>>> example(1)(4)
```

# Nonlocal in Functions - Side Question

```
def example(start):
    def add(x):
        nonlocal start
        start += x
        return start
    return add

>>> example(1)(6)
7
>>> example(1)(4)
5
```

# Nonlocal: Rabbit Population

```
def rabbit_population(n, capacity):
    """ Ronny really likes bunnies, so she buys n to start with.
    They reproduce at the rate given by a function at every
    timestamp. However, if the function doesn't increase the
    number of bunnies, use the most recent one that did (start
    at doubling).  When the number reaches >= the capacity of
    her home, she needs to give 9/10 away to her friends.
    >>> timestamp = rabbit_population(5, 40)
    >>> timestamp(lambda x: x - 10)
    10
    >>> timestamp(lambda x: x * 4)
    40
    >>> timestamp(lambda x: x * 3)
    4
    """
```

# Nonlocal: Rabbit Population Solution

```
def rabbit_population(n, capacity):
    increase = lambda x: x * 2
    def timestamp(fn):
        nonlocal increase, n
        if fn(n) > n: # determine whether or not fn increases n
            increase = fn
        if n < capacity:
            n = increase(n)
        else:
            n = n // 10 # don't increase if too many
        return n
    return timestamp
```

# Nonlocal: Generate Lists

```
def gen_seq():
    """
    >>> update = gen_seq()
    >>> update()
    []
    >>> update()
    [1]
    >>> update()
    [1, 2]
    >>> update()
    [1, 2, 3]
    """
```

# Nonlocal: Generate Lists

```
def gen_seq():
    lst = None
    def update():
        if lst != None:
            lst.append(len(lst) + 1)
        else:
            lst = []
        return lst
    return update
```

Does This Work?

No.

need nonlocal to reassign

# Nonlocal: Generate Lists

```
def gen_seq():
    lst = []
    def update():
        temp = lst
        lst.append(len(lst) + 1)
        return temp
    return update
```

Does This Work?

No.

Both point to same list!
Might as well return `lst`.

# ~~Nonlocal~~: Generate Lists Solution

```
def gen_seq():
    lst = []
    def update():
        temp = list(lst)
        lst.append(len(lst) + 1)
        return temp
    return update
```

Sorry, trick question... we don't need nonlocal

# Nonlocal?: Generate Fib List

```
def make_fib():
    """ List of fibonacci numbers!
    >>> fib = make_fib()
    >>> fib()
    [0]
    >>> fib()
    [0, 1]
    >>> fib()
    [0, 1, 1]
    >>> fib()
    [0, 1, 1, 2]
    """
```

# ~~Nonlocal~~: Generate Fib List Soln

```
def make_fib():
    lst = []
    def update():
        if len(lst) < 2:
            lst.append(len(lst))
            return lst
        lst.append(lst[-1] + lst[-2])
        return lst
    return update
```

# Dog bark

```
class Dog(object):
    def bark(self):
        print('woof')
```

What would the following statements do?

```
a. fido.bark = 'bow wow'
   fido.bark()
```

error: 'str' object not callable

```
b. fido.bark = Dog.bark
   fido.bark()
```

error: bark() takes exactly 1 argument

```
c. fido.bark(fido)
```

woof

# Hungry Hungry Humans (lol)

```
class Animal(object):
    def __init__(self, name):
        self.n = name
        self.hunger = 0
    def eat(self, food):
        self.hunger+=1
        if self.hunger >= 2:
            print('Dead')
        else:
            self.hunger = 0
            print('eaten')
    def name(self):
        if self.hunger >= 3:
            return 'Dead'
        else:
            return self.n
```

```
class Person(Animal):
    dislikes = ['grass', 'mud']
    def eat(self, food):
        if food in Person.dislikes:
            self.hunger += 1
            print('No good')
        else:
            Animal.eat(self, food)
```

# Hungry Hungry Humans (cont)

```
c, p = Animal('cow'), Person('Mike')
```

What do the following lines print

a. `c.eat('grass')`

```
eaten
```

b.
```
for _ in range(3):
    p.eat('grass')
```

```
No good
No good
No good
```

c. `p.name()`

```
'Dead'
```

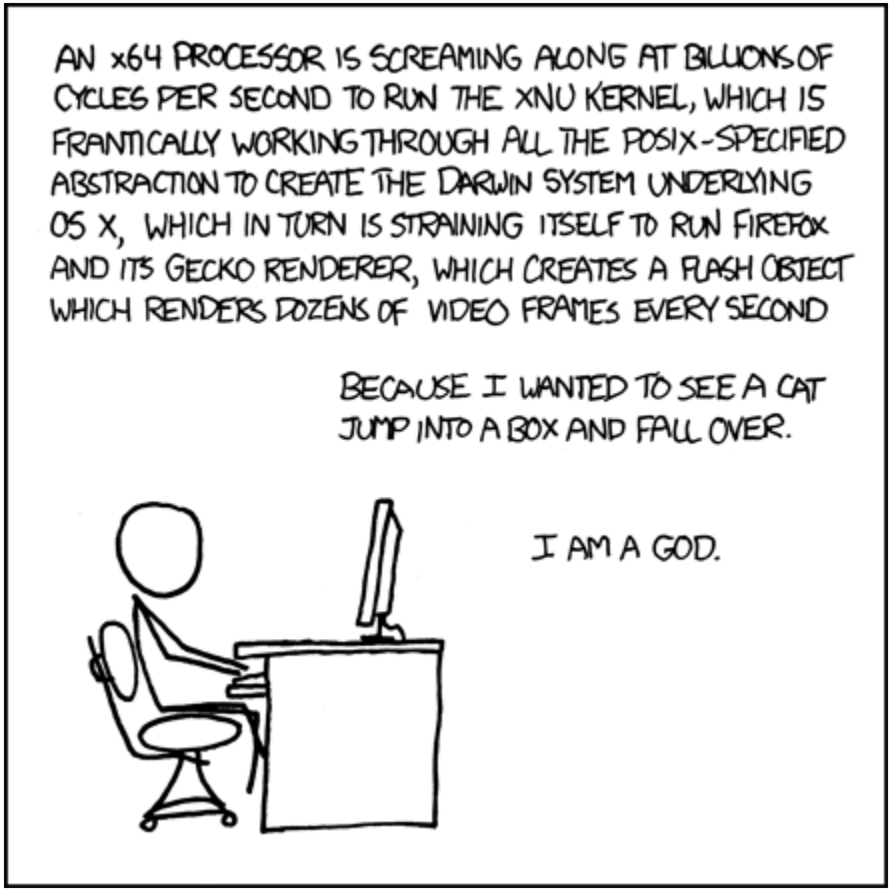
# Picky Eater

Use the above class definitions. Define a subclass of ‘Person’ called ‘PickyPerson’ that not only dislikes grass and mud, but adds a list of dislikes unique to each ‘PickyPerson’. If a food is in a PickyPerson’s dislikes, it prints “No good, I’m picky”, but prints the same thing as Person if it is in Person’s dislikes

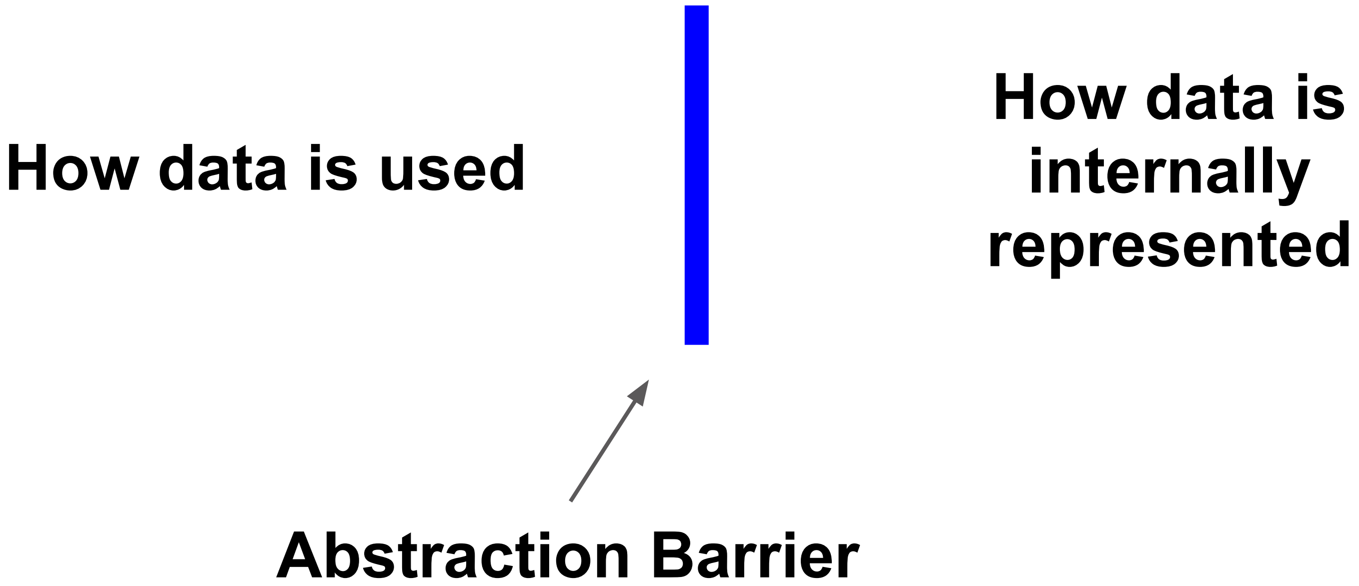
# Picky Eater (cont)

```
class PickyPerson(Person):

   def __init__(self, name, more):
       Person.__init__(self, name)
       self.dislikes = more

   def eat(self, food):
       if food in self.dislikes:
           self.hunger += 1
           print("No good I'm picky")
       else:
           Person.eat(self, food)
```

# Data Abstraction


AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND
BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.
I AM A GOD.

# Data Abstraction



How data is used

How data is internally represented

Abstraction Barrier

# Example: Points

```
def make_point(x, y):
   return(x,y)

def x(point):
   return point[0]

def y(point):
   return point[1]

def dist(point1, point2):
   return sqrt((x(point2) - x(point1)) ** 2 + \
               (y(point2) - y(point1)) ** 2)
```

# Question 1 part a

Use the point abstraction to implement a line segment abstraction that has the following functions:

`make_segment(start, end)`

- Constructs a line segment between start and end

`length(seg)`

- Returns the distance between seg's start and end

`slope(seg)`

- Returns the slope of the line through seg's start and end

`consecutive(seg1, seg2)`

- Returns true if seg1's end is the same as seg2's start, or false otherwise

For reference, the point data abstraction:

```
make_point(x,y)
x(point)
y(point)
dist(point1,point2)
```

# Question 1 part a (soln.)

```
def make_segment(start,end):
    return (start, end)
```

# Question 1 part a (soln.)

```
def make_segment(start,end):
    return (start, end)
def length(seg):
    return dist(seg[0], seg[1])
```

# Question 1 part a (soln.)

```
def make_segment(start,end):
    return (start, end)
def length(seg):
    return dist(seg[0], seg[1])
def slope(seg)
    dy = y(seg[1]) - y(seg[0])
    dx = x(seg[1]) - x(seg[0])
    if dx == 0:
        return None
    return dy/dx
```

# Question 1 part a (soln.)

```
def make_segment(start,end):
    return (start, end)
def length(seg):
    return dist(seg[0], seg[1])
def slope(seg)
    dy = y(seg[1]) - y(seg[0])
    dx = x(seg[1]) - x(seg[0])
    if dx == 0:
        return None
    return dy/dx
def consecutive(seg1, seg2):
    return seg1[1] == seg2[0]
```

# Question 1 part a (soln.)

```
def make_segment(start,end):
    return (start, end)
def length(seg):
    return dist(seg[0], seg[1])
def slope(seg)
    dy = y(seg[1]) - y(seg[0])
    dx = x(seg[1]) - x(seg[0])
    if dx == 0:
        return None
    return dy/dx
def consecutive(seg1, seg2):
    ~~return seg1[1] == seg2[0]~~
    return (x(seg1[1]) == x(seg2[0])) and (y(seg1[1]) == y(seg2[0]))
```

# Question 1 part b

Your friend has written a function to compute the total length of a path of line segments, but she has broken some abstraction barriers in doing so. Rewrite this function so that it does the same thing but uses the line segment abstraction properly.

```
#Assume path is a tuple of line segments
def path_length(path):
    prev = path[0][1]
    ret = 0
    for (s, cur) in path[1:]:
        if s != prev:
            return None
        else:
            ret += dist(s,cur)
        prev = cur
    return ret
```

# Question 1 part b (soln.)

Your friend has written a function to compute the total length of a path of line segments, but she has broken some abstraction barriers in doing so. Rewrite this function so that it does the same thing but uses the line segment abstraction properly.

```
#Assume path is a tuple of line segments
def path_length(path):
    prev = path[0]
    ret = 0
    for cur in path[1:]:
        if not consecutive(prev,cur):
            return None
        else:
            ret += length(cur)
        prev = cur
    return ret
```

# Question 2

In this question, we will implement a hacky version of the iterator abstraction. This abstraction has only one function -- make_iter -- which takes an RList and returns a function which walks through each value of the RList:

```
>>> iter = make_iter(rlist(1, rlist(2, rlist(3, None))))
>>> iter()
1
>>> iter()
2
>>> iter()
3
>>> iter()
None
```

# Question 2 part a

Use the iterator abstraction to implement reduce.

```
def reduce(combiner, iter, start):
```

# Question 2 part a (soln.)

Use the iterator abstraction to implement reduce.

```
def reduce(combiner, iter, start):
    acc = start
    val = iter()
    while val != None:
        acc = combiner(acc, val)
        val = iter()
    return acc
```

# Question 2 part b

Implement `make_iter`. Hint: you may need to have some mutable data.

```
>>> iter = make_iter(rlist(1, rlist(2, rlist(3, None))))
>>> iter()
1
>>> iter()
2
>>> iter()
3
>>> iter()
None
```

# Question 2 part b (soln.)

```
def make_iter(lst):
    def ret():
        nonlocal lst
        if lst == None:
            return None

        cur = first(lst)
        lst = rest(lst)
        return cur

    return ret
```

# Quicksorting Recursive Lists

Implement a quicksort algorithm for immutable rlists.

Quicksort algorithm picks an item B in the list and creates:

- a list A of items from the original list less than B
- a list C of items from the original list greater than or equal to B

These two lists are once again sorted using quicksort.

Finally, we append a list containing only B to A, and then append C to the result of the first appending.

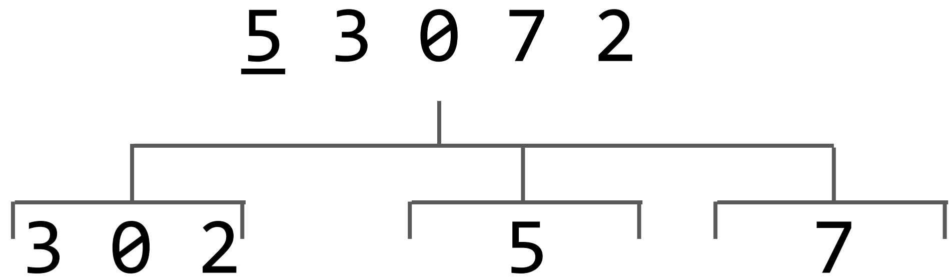
# Quicksort Example

Given a list:

```
5 3 0 7 2
```

# Quicksort Example

Given a list:



5 3 0 7 2

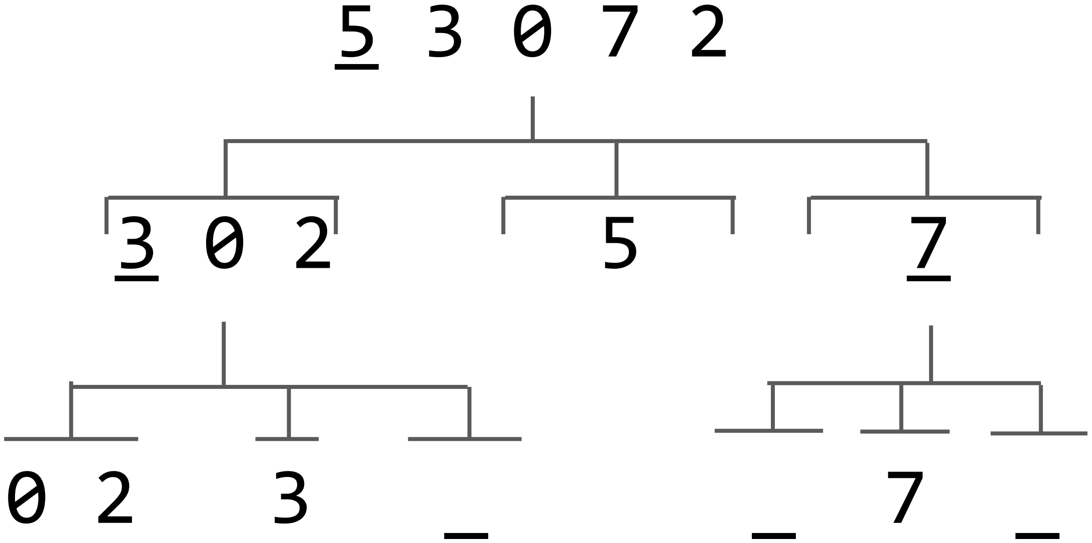
3 0 2

5

7

# Quicksort Example

Given a list:



5 3 0 7 2

3 0 2

5

7

0 2

3

_

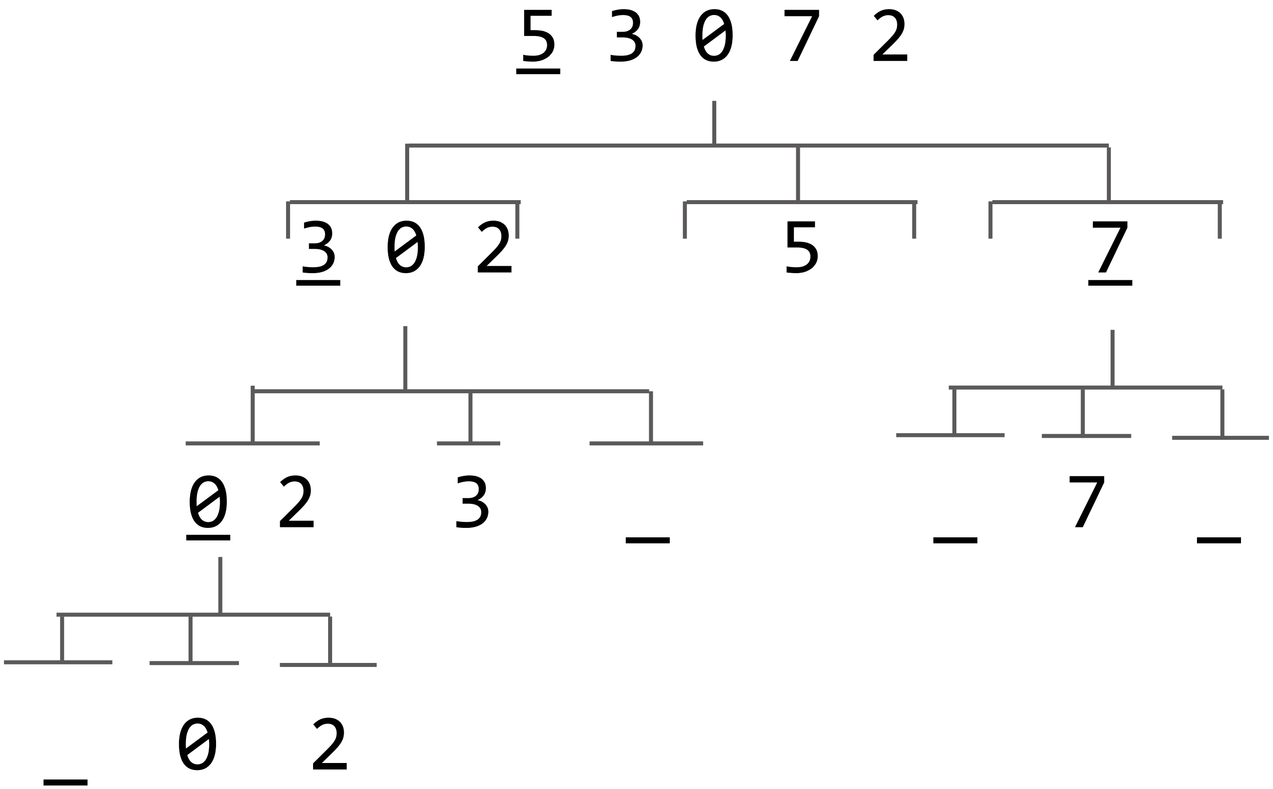
_

7

_

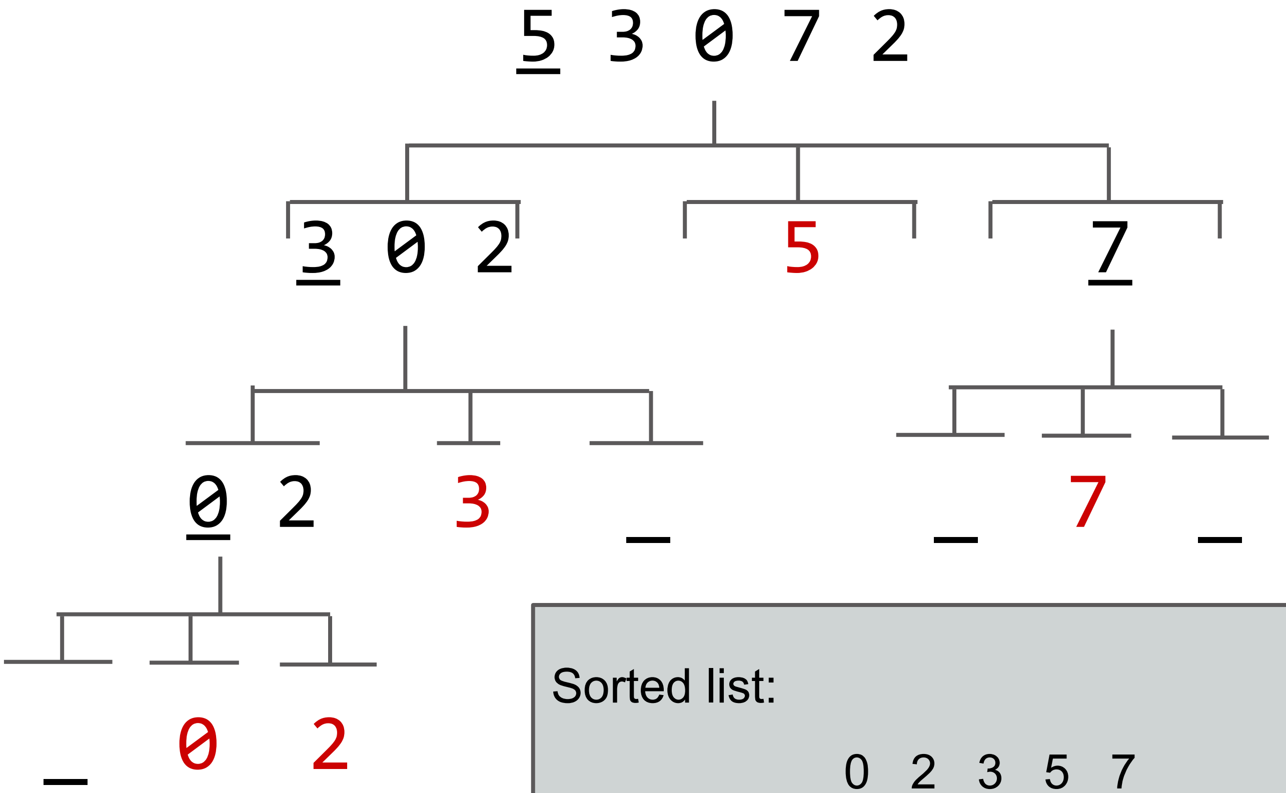
# Quicksort Example

Given a list:



# Quicksort Example

Given a list:



Sorted list:

0 2 3 5 7

# Rlist Implementation

This is the data abstraction that we will be using for our immutable rlists:

```
empty_rlist = None

def rlist(first, rest):
   """Construct a recursive list from its first element and the rest."""
   return (first, rest)

def first(s):
   """Return the first element of a recursive list s."""
   return s[0]

def rest(s):
   """Return the rest of the elements of a recursive list s."""
   return s[1]
```

# Breaking Down the Problem

We will want to create helper functions to break down the process into several key steps.

First, write two functions to create such lists A and C from an rlist r given a value n.

Then we should write a function that will allow us to append the contents of one rlist to another.

Finally, we will put these pieces together.

# Split Up the List

```
def less_rlist(n, r):
    """Construct an rlist containing only values from r less than n."""
```

```
def greater_rlist(n, r):
    """Construct an rlist containing only values from r greater than or
    equal to n."""
```

# Split Up the List

```
def less_rlist(n, r):
    """Construct an rlist containing only values from r less than n."""
    if r == empty_rlist:
        return r
```

```
def greater_rlist(n, r):
    """Construct an rlist containing only values from r greater than or
    equal to n."""
```

# Split Up the List

```
def less_rlist(n, r):
    """Construct an rlist containing only values from r less than n."""
    if r == empty_rlist:
        return r
    if first(r) < n:
        return rlist(first(r), less_rlist(n, rest(r)))


def greater_rlist(n, r):
    """Construct an rlist containing only values from r greater than or
    equal to n."""
```

# Split Up the List

```
def less_rlist(n, r):
    """Construct an rlist containing only values from r less than n."""
    if r == empty_rlist:
        return r
    if first(r) < n:
        return rlist(first(r), less_rlist(n, rest(r)))
    return less_rlist(n, rest(r))

def greater_rlist(n, r):
    """Construct an rlist containing only values from r greater than or
    equal to n."""
```

# Split Up the List

```
def less_rlist(n, r):
    """Construct an rlist containing only values from r less than n."""
    if r == empty_rlist:
        return r
    if first(r) < n:
        return rlist(first(r), less_rlist(n, rest(r)))
    return less_rlist(n, rest(r))

def greater_rlist(n, r):
    """Construct an rlist containing only values from r greater than or
    equal to n."""
    if r == empty_rlist:
        return r
    if first(r) >= n:
        return rlist(first(r), greater_rlist(n, rest(r)))
    return greater_rlist(n, rest(r))
```

# Appending rlists

```
def append_rlist(r1, r2):
    """Append the contents of rlist r2 to the end of rlist r1."""
```

# Appending rlists

```
def append_rlist(r1, r2):
    """Append the contents of rlist r2 to the end of rlist r1."""
    if r1 == empty_list:
        return r2
    if r2 == empty_list:
        return r1
```

# Appending rlists

```
def append_rlist(r1, r2):
    """Append the contents of rlist r2 to the end of rlist r1."""
    if r1 == empty_list:
        return r2
    if r2 == empty_list:
        return r1
    return rlist(first(r1), append_rlist(rest(r1), r2))
```

# Sorting Everything Out

```
def quicksort_rlist(r):
    """Sort the rlist r in ascending order using a quicksort algorithm.
    >>> r = rlist(5, rlist(3, rlist(3, rlist(8, rlist(2, rlist(4, rlist(9,
        rlist(-1, rlist(7, rlist(6, empty_rlist))))))))))
    >>> r
    (5, (3, (3, (8, (2, (4, (9, (-1, (7, (6, None))))))))))
    >>> quick_sort_rlist(r)
    (-1, (0, (3, (3, (4, (5, (6, (7, (8, (9, None))))))))))
    """
```

# Sorting Everything Out

```
def quicksort_rlist(r):
    """Sort the rlist r in ascending order using a quicksort algorithm.
    >>> r = rlist(5, rlist(3, rlist(3, rlist(8, rlist(2, rlist(4, rlist(9,
        rlist(-1, rlist(7, rlist(6, empty_rlist))))))))))
    >>> r
    (5, (3, (3, (8, (2, (4, (9, (-1, (7, (6, None))))))))))
    >>> quick_sort_rlist(r)
    (-1, (0, (3, (3, (4, (5, (6, (7, (8, (9, None))))))))))
    """
    if r == empty_list or rest(r) == empty_list:
        return r
```

# Sorting Everything Out

```
def quicksort_rlist(r):
    """Sort the rlist r in ascending order using a quicksort algorithm.
    >>> r = rlist(5, rlist(3, rlist(3, rlist(8, rlist(2, rlist(4, rlist(9,
        rlist(-1, rlist(7, rlist(6, empty_rlist))))))))))
    >>> r
    (5, (3, (3, (8, (2, (4, (9, (-1, (7, (6, None))))))))))
    >>> quick_sort_rlist(r)
    (-1, (0, (3, (3, (4, (5, (6, (7, (8, (9, None))))))))))
    """
    if r == empty_list or rest(r) == empty_list:
        return r
    n = first(r)
    less =                 less_rlist(n, rest(r))
    more =                 greater_rlist(n, rest(r))
```

# Sorting Everything Out

```
def quicksort_rlist(r):
    """Sort the rlist r in ascending order using a quicksort algorithm.
    >>> r = rlist(5, rlist(3, rlist(3, rlist(8, rlist(2, rlist(4, rlist(9,
        rlist(-1, rlist(7, rlist(6, empty_rlist))))))))))
    >>> r
    (5, (3, (3, (8, (2, (4, (9, (-1, (7, (6, None))))))))))
    >>> quick_sort_rlist(r)
    (-1, (0, (3, (3, (4, (5, (6, (7, (8, (9, None))))))))))
    """
    if r == empty_list or rest(r) == empty_list:
        return r
    n = first(r)
    less = quicksort_rlist(less_rlist(n, rest(r)))
    more = quicksort_rlist(greater_rlist(n, rest(r)))
```

# Sorting Everything Out

```
def quicksort_rlist(r):
    """Sort the rlist r in ascending order using a quicksort algorithm.
    >>> r = rlist(5, rlist(3, rlist(3, rlist(8, rlist(2, rlist(4, rlist(9,
        rlist(-1, rlist(7, rlist(6, empty_rlist))))))))))
    >>> r
    (5, (3, (3, (8, (2, (4, (9, (-1, (7, (6, None))))))))))
    >>> quick_sort_rlist(r)
    (-1, (0, (3, (3, (4, (5, (6, (7, (8, (9, None))))))))))
    """
    if r == empty_list or rest(r) == empty_list:
        return r
    n = first(r)
    less = quicksort_rlist(less_rlist(n, rest(r)))
    more = quicksort_rlist(greater_rlist(n, rest(r)))
    return append_rlist(append_rlist(less, rlist(n, empty_rlist)), more)
```

# Removing Duplicates

Write a function that takes an rlist and removes duplicate values from that rlist. Do **not** create and return a new rlist.

We will need to keep track of the values that the list already contains as we recurse through the rlist, so we should make a helper function.

# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
```

# Removing Duplicates

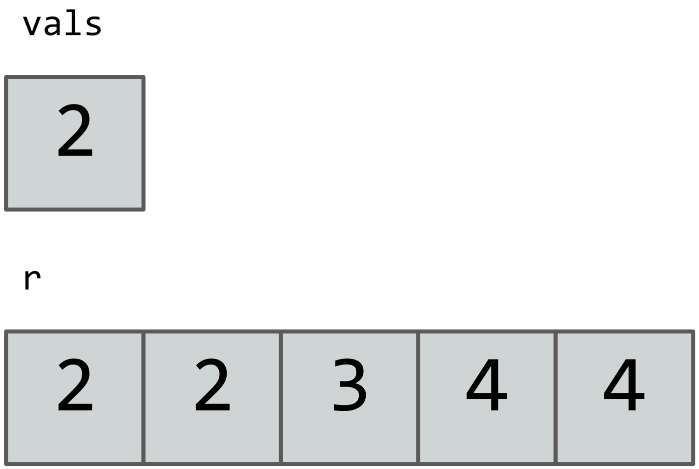
```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
```

# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
```

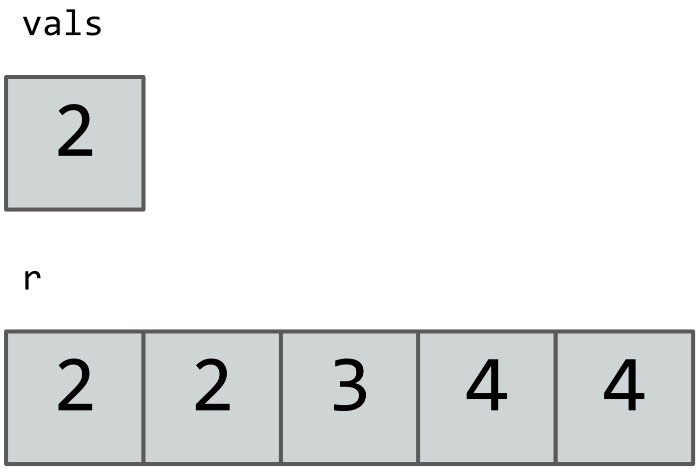
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return





    dedupe_helper(r, [r.first])
```



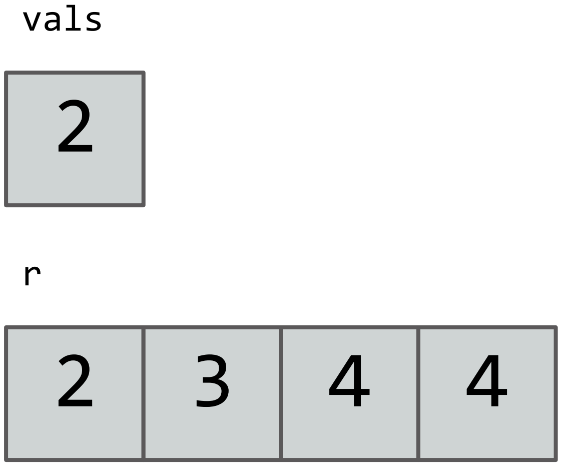
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)



    dedupe_helper(r, [r.first])
```



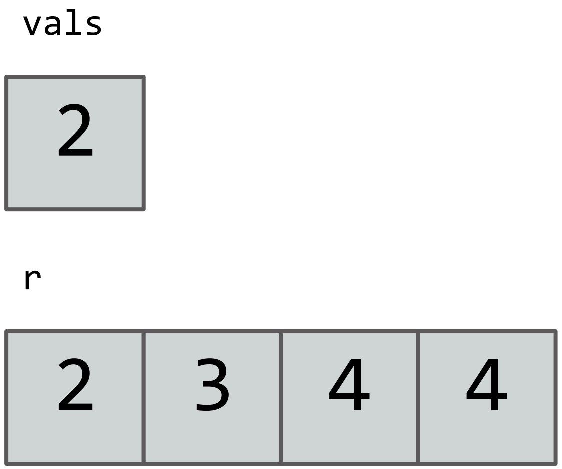
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)



    dedupe_helper(r, [r.first])
```



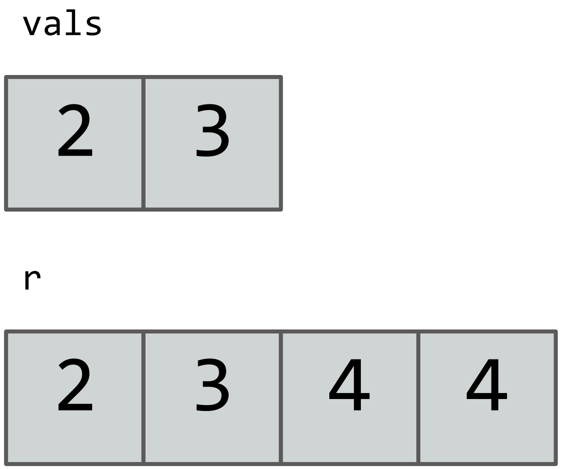
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)
        else:
            vals += [r.rest.first]
            dedupe_helper(r.rest, vals)
    dedupe_helper(r, [r.first])
```



# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)
        else:
            vals += [r.rest.first]
            dedupe_helper(r.rest, vals)
    dedupe_helper(r, [r.first])
```


vals

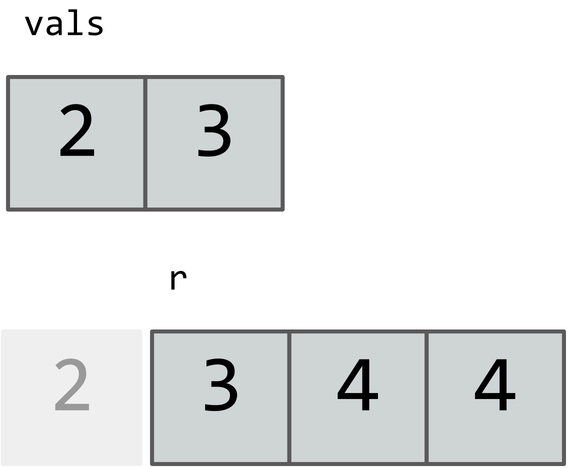
2 3

r

2 3 4 4

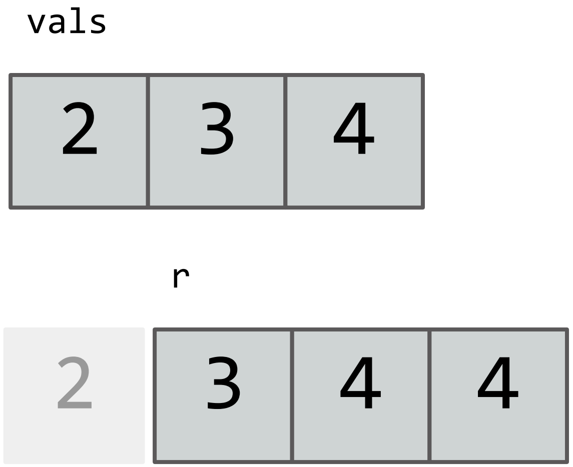
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)
        else:
            vals += [r.rest.first]
            dedupe_helper(r.rest, vals)
    dedupe_helper(r, [r.first])
```



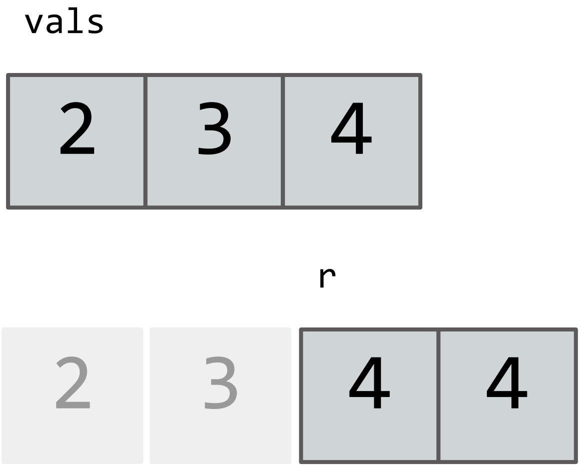
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)
        else:
            vals += [r.rest.first]
            dedupe_helper(r.rest, vals)
    dedupe_helper(r, [r.first])
```



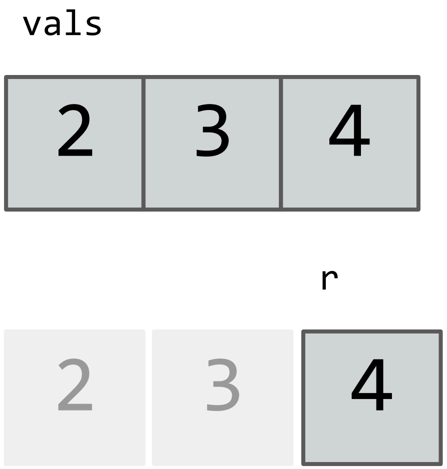
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)
        else:
            vals += [r.rest.first]
            dedupe_helper(r.rest, vals)
    dedupe_helper(r, [r.first])
```



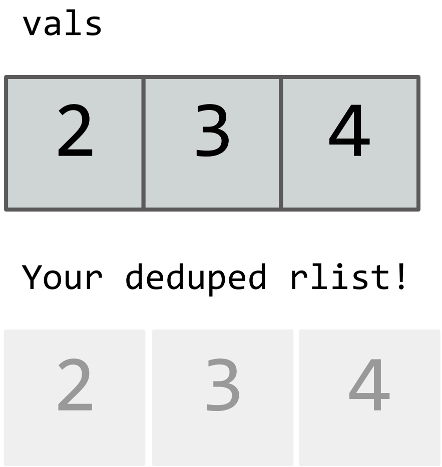
# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)
        else:
            vals += [r.rest.first]
            dedupe_helper(r.rest, vals)
    dedupe_helper(r, [r.first])
```



# Removing Duplicates

```
def dedupe(r):
    """Take a mutable rlist r and remove all duplicate values from it.
    Do NOT create and return a new list."""
    def dedupe_helper(r, vals):
        if r.rest == Rlist.empty:
            return
        if r.rest.first in vals:
            r.rest = r.rest.rest
            dedupe_helper(r, vals)
        else:
            vals += [r.rest.first]
            dedupe_helper(r.rest, vals)
    dedupe_helper(r, [r.first])
```



vals

| | | |
|---|---|---|
| 2 | 3 | 4 |

Your deduped rlist!

| | | |
|---|---|---|
| 2 | 3 | 4 |