CS61A Midterm 2 Review

Davis Foote
Brian Hou
Jonathan Kotker

Hello!

Poll Everywhere: http://goo.gl/Rn6Vlv
or text “263002 ” + your_msg to 22333

Hosted by HKN (hkn.eecs.berkeley.edu)

Office hours from 11AM to 5PM in 290 Cory, 345 Soda

Check our website for exam archive, course guide, course surveys, tutoring schedule.

This is an unofficial review session and HKN is not affiliated with this course. All of the topics we are reviewing will reflect the material you have covered, our experiences in CS61A, and past midterms. We make no promise that what we cover will necessarily reflect the content of the midterm.

Also, please help keep this room clean - no food or drink allowed. Thank you!

Agenda

- Lists, Tuples, Dictionaries
- HOFs on sequences
- Data Abstraction
- Nonlocal
- Object-Oriented Programming
- Inheritance
- Recursive lists
- Trees
- Orders of Growth

Follow along at http://goo.gl/YGWBCH!

bonus

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; we can only create new tuples.

- *Dictionaries*: Objects that map keys to values. Remember that the keys are unordered and unique!

- *Ranges*: Objects that represent an interval of elements between two values.

List Comprehensions

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

```
>>> [x + 2 for x in range(5) if x > 3]
[6]
```

List comprehensions provide a way to write a simple `for`-loop and optional `if`-statement in one line. For example, the above snippet is equivalent to:

```
>>> result = []
>>> for x in range(5):
...     if x > 3:
...         result.append(x + 2)
...
>>> result
[6]
```

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]
```

bonus

Higher Order Functions

Map - Takes in a function and a sequence, and applies the function to each element of the sequence.

Input - Function that takes in **one argument**, which is any iterable sequence (list, tuple, etc.).

Output - Sequence of the same length as the input.

For example:

```
>>> list(map(lambda x: x*x, [2, 3, 4]))
[4, 9, 16]
```

bonus

Higher Order Functions

Filter - Takes in a function and a sequence, and returns a new sequence that contains only the items for which the function returns `True`.

Input - Function that takes in **one argument**, which is any iterable sequence (list, tuple, etc.).

Output - Sequence that contains the elements that satisfy the function.

For example:

```
>>> list(filter(lambda x: x % 2 == 0, [2, 3, 4]))
[2, 4]
```

bonus

Higher Order Functions

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**: an iterable sequence (list, tuple, etc.) and an (optional) starting value.

Output - Single element that is determined by combining the elements of the sequence using the input function.

For example:

```
>>> reduce(lambda sofar, curr: sofar+curr, [2, 3, 4]))
9
```

Reduce

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

```
>>> from functools import reduce
>>> t = [1, 2, 3, 4, 5, 6]
>>> reduce(______________________________, _____)
123456
```

Reduce

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

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

How?

Reduce

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

First iteration:

```
so_far = 1
curr = 2
```

Reduce

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

First iteration:

```
so_far = 1
curr = 2
result = 1*10 + 2 = 12
```

Reduce

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

Next iteration:

```
so_far = 12
curr = 3
result = 12*10 + 3 = 123
```

and so on.

Map and Reduce

```
>>> from functools import reduce
>>> cool = 'hilfinger'
>>> story = [cool[i:2*i] for i in range(6)]
>>> story

_____
>>> bro = map(len, story)
>>> bro

_____
>>> print(reduce(print, bro), list(bro))

_____
```

Map and Reduce

```
>>> from functools import reduce
>>> cool = 'hilfinger'
>>> story = [cool[i:2*i] for i in range(6)]
>>> story
['', 'i', 'lf', 'fin', 'inge', 'nger']
>>> bro = map(len, story)
>>> bro

_____

>>> print(reduce(print, bro), list(bro))

_____
```

Map and Reduce

```
>>> from functools import reduce
>>> cool = 'hilfinger'
>>> story = [cool[i:2*i] for i in range(6)]
>>> story
['', 'i', 'lf', 'fin', 'inge', 'nger']
>>> bro = map(len, story)
>>> bro
<map object at 0xdeadbeef>
>>> print(reduce(print, bro), list(bro))

______
```

Map and Reduce

```
>>> story
['', 'i', 'lf', 'fin', 'inge', 'nger']
>>> bro
<map object at 0xdeadbeef>
>>> print(reduce(print, bro), list(bro))
0 1
None 2
None 3
None 4
None 4
None []
```

bonus

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)))


______
```

bonus

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)~~))
                           [2, 3]

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


_____
```

bonus

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)~~))
                           [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, filter(is_prime, fib)))


_____
```

bonus

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)~~))
                            [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, ~~filter(is_prime, fib)~~))
                            [2, 3]
```

bonus

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)~~))
                           [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)]
```

bonus

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)~~))
                           [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> list(map(get_fib, ~~filter(is_prime, fib)~~))
                           [2, 3]
# intermediate step [fib[2], fib[3]]
```

bonus

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

Dictionaries

Dictionaries store data by mapping keys to values. We can access this data with 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 we try to add a value for a key that already exists, this value overrides the previous value.)

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:

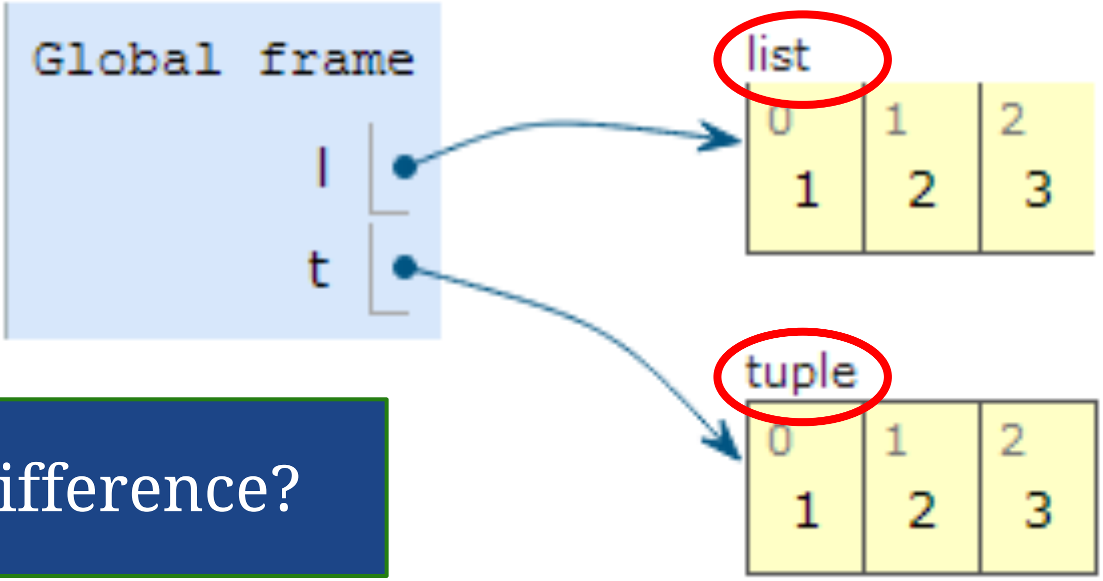
[(0, 0), (0, 1), (0, 2), **(0, 3)**, (1, 0), (1, 1), (1, 2), **(1, 3)**, (2, 0), (2, 1), (2, 2), **(2, 3)**]

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

Because dictionary keys are unique.

Lists/Tuples in Environment Diagrams

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

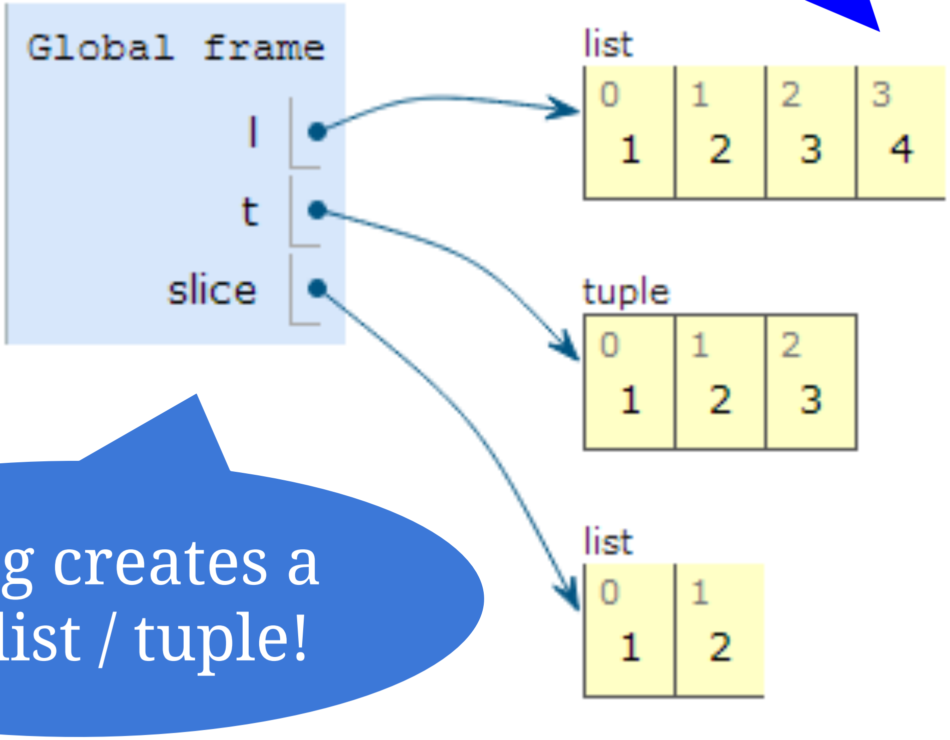


What's the difference?

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

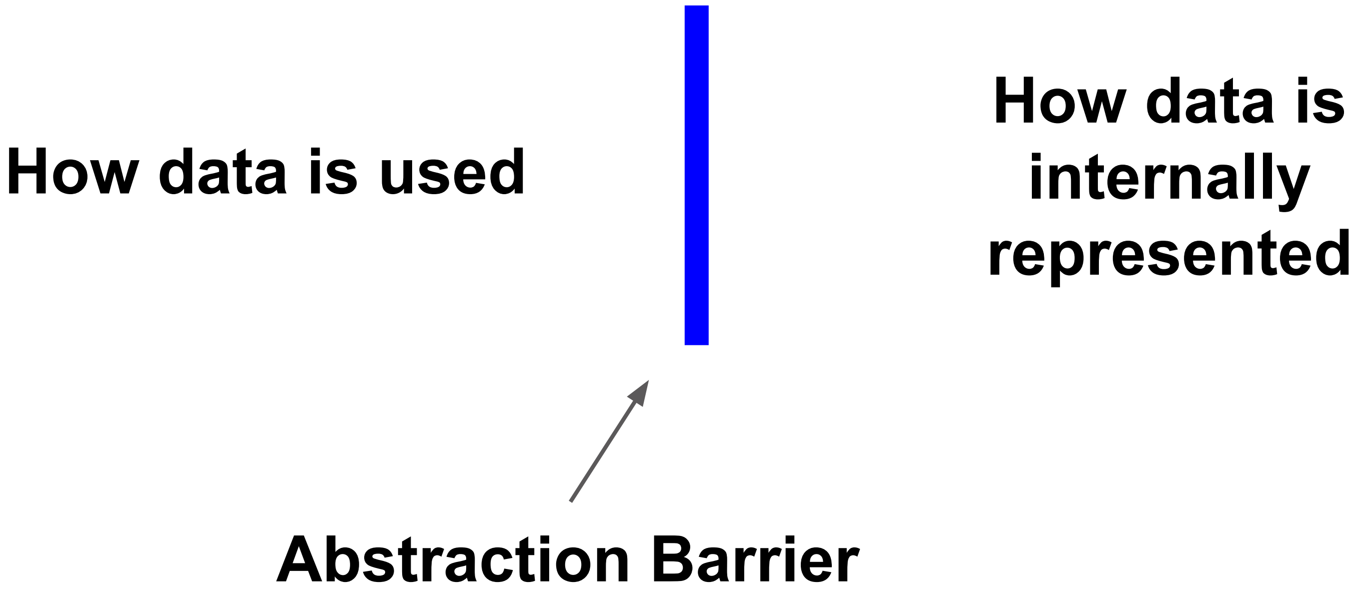
```
r = ([1, 2, 1, 2],)
s = list(r)
t = r
r[0][2] = t[0]
s[0] = r[0][1:]
s[0][1][2][3] = 4
```

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.

http://xkcd.com/676/

Data Abstraction



How data is used

How data is internally represented

Abstraction Barrier

Example: Points

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

Constructor - Builds an object of the abstract data type.

```
def x(point):
    return point[0]
```

Selector - Extracts relevant information from the object.

```
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 with the following constructors and selectors:

`make_segment(start, end)`

Constructs a line segment between points at `start` and `end`.

`start(segment), end(segment)`

Returns the start and end points respectively.

`length(segment)`

Returns the distance between the segment's start and end points.

`consecutive(seg1, seg2)`

Returns `True` if `seg1`'s `end` is the same as `seg2`'s `start`, or `False` otherwise.

For reference, the data abstraction for points has the following constructors and selectors:

```
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 start(segment):
    return segment[0]
def end(segment):
    return segment[1]
```

Question 1, part A (Soln.)

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

Question 1, part A (Soln.)

```
def make_segment(start, end):
    return (start, end)
def start(segment):
    return segment[0]
def end(segment):
    return segment[1]
def length(segment):
    return dist(start(segment), end(segment))
def consecutive(seg1, seg2):
    return end(seg1) == start(seg2)
```

Question 1, part A (Soln.)

```
def make_segment(start, end):
    return (start, end)
def start(segment):
    return segment[0]
def end(segment):
    return segment[1]
def length(segment):
    return dist(start(segment), end(segment))
def consecutive(seg1, seg2):
    ~~return end(seg1) == start(seg2)~~
    return ((x(end(seg1)) == x(start(seg2))) and
            (y(end(seg1)) == y(start(seg2))))
```

Question 1, part B

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

```
# Assume path is a tuple of line segments.
def path_length(path):
    prev = path[0][1]
    ret = dist(path[0][0], prev)
    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 has broken some abstraction barriers in doing so. Rewrite this function so that it uses the line segment abstraction properly.

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

bonus

Question 2

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

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

bonus

Question 2

Use the iterator abstraction to implement the `reduce` higher order function.

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

bonus

Question 2 (Soln.)

Use the iterator abstraction to implement the reduce higher order function.

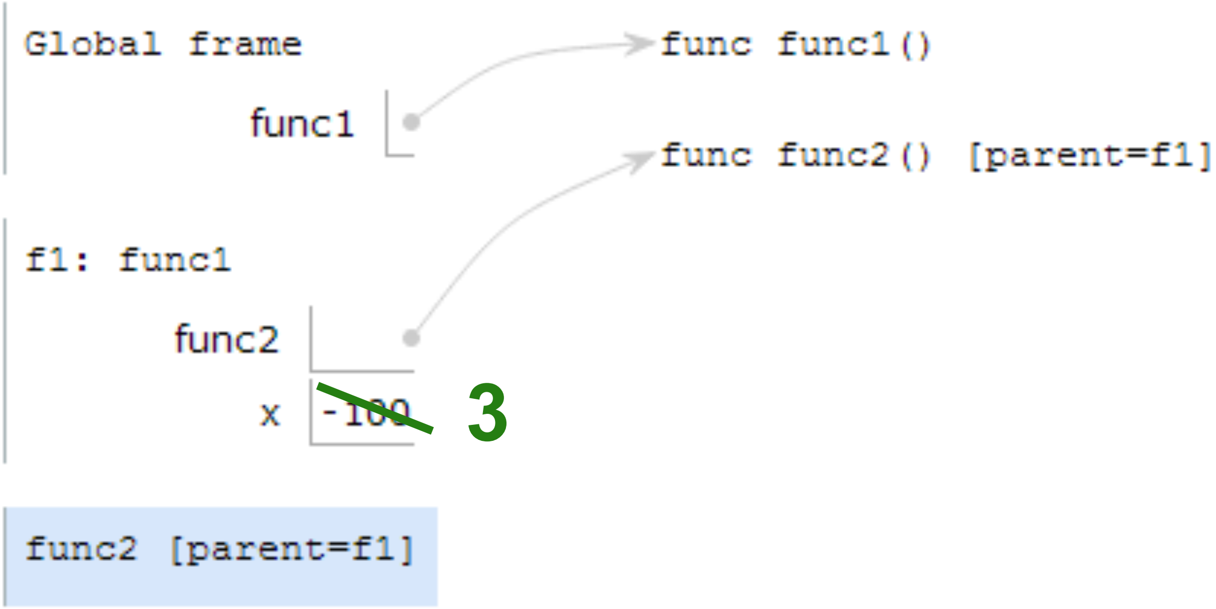
```
def reduce(combiner, iter, start):
    acc = start
    val = iter()
    while val != None:
        acc = combiner(acc, val)
        val = iter()
    return acc
```

You did not need to know how iter was implemented to actually use it!

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

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

Does This Work?

Yes!

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

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

Does This Work?

No.

x is in the global frame.

Draw the Environment Diagram

```
def k(b):
    def seven(up):
        b.extend(['<3','<3'])
        nonlocal b
        b = 5
        up[0][0] = 'cs61a'
        return up[0:2]
    return seven((b, 3, 6))

k(['cookies'])
```

Environment Diagram Notes

```
def k(b):
    def seven(up):
        b.extend(['<3','<3'])
        nonlocal b
        b = 5
        up[0][0] = 'cs61a'
        return up[0:2]
    return seven((b, 3, 6))

k(['cookies'])
```

Don't need nonlocal to mutate something!

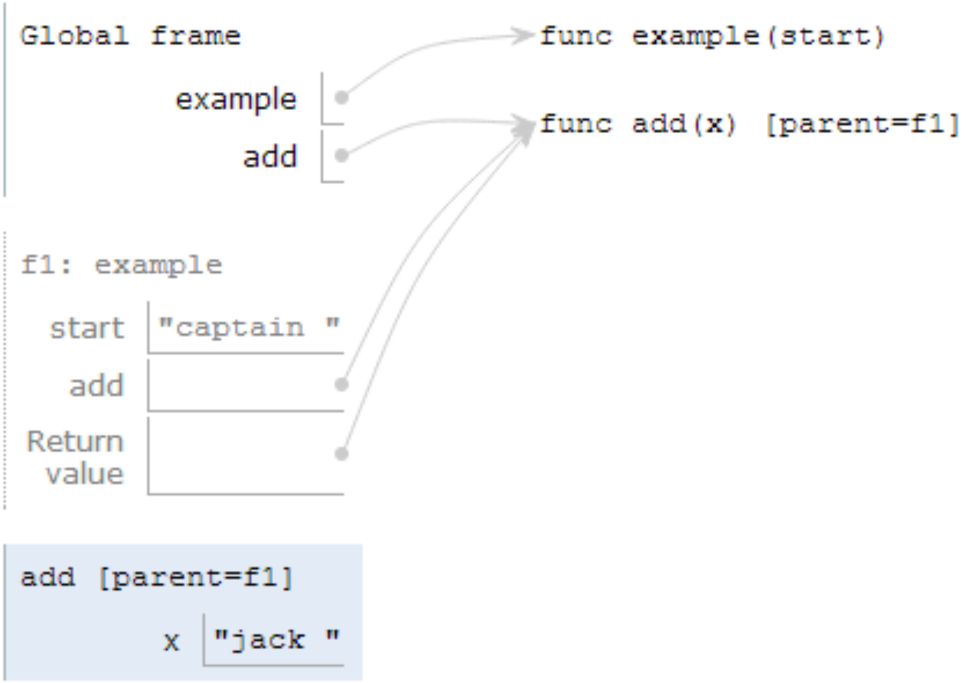
Need nonlocal to change what value a variable points to

Can't change what items a tuple contains, but you can mutate those items.

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
```



Global frame

example

add

func example(start)

func add(x) [parent=f1]

f1: example

start "captain "

add

Return value

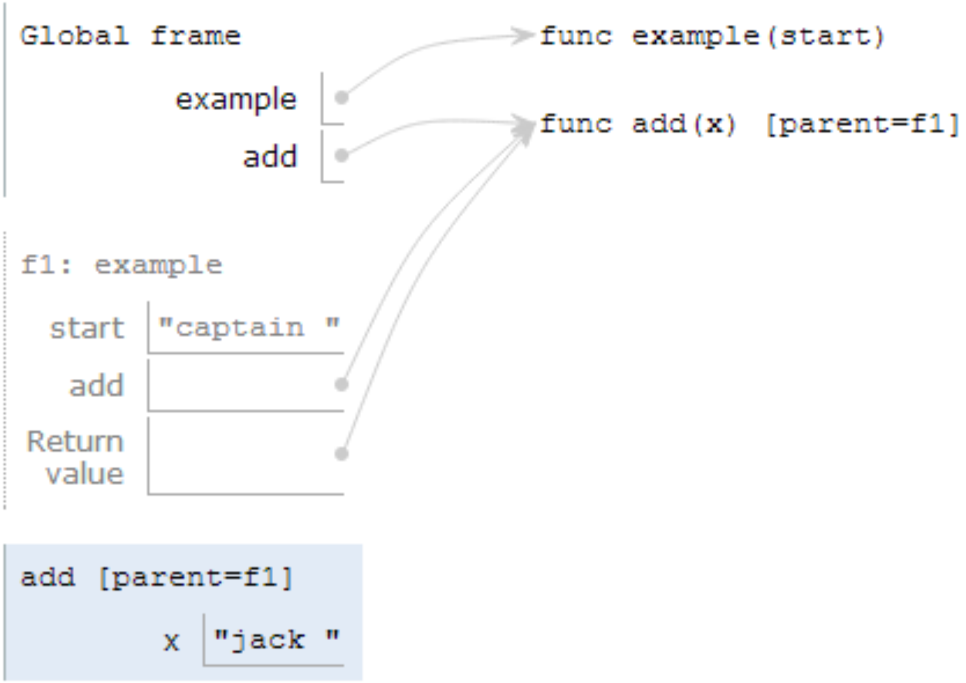
add [parent=f1]

x "jack "

```
>>> add = example(‘captain ’)
>>> add(‘jack ’)

>>> add(‘sparrow ’)
```

Nonlocal in Functions

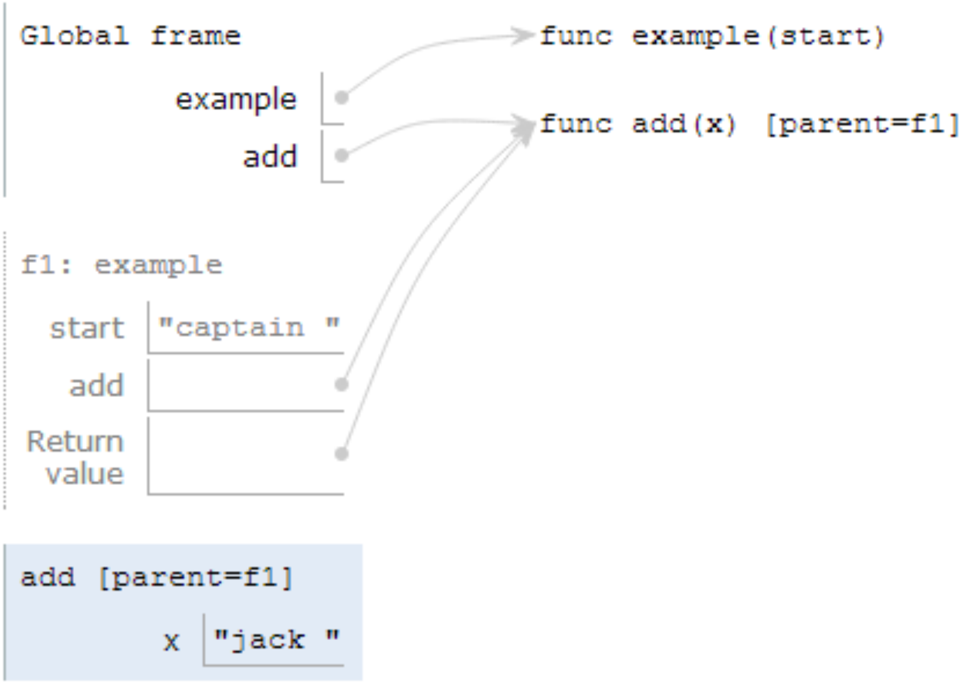
```
def example(start):
    def add(x):
        nonlocal start
        start += x
        return start
    return add
```



```
>>> add = example(‘captain ’)
>>> add(‘jack ’)
‘captain jack ’
>>> add(‘sparrow ’)
```

Nonlocal in Functions

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



```
>>> add = example(‘captain ’)
>>> add(‘jack ’)
‘captain jack ’
>>> add(‘sparrow ’)
‘captain jack sparrow’
```

Nonlocal in Functions - Side Question

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

>>> example(‘captain ’)(‘jack ’)

>>> example(‘captain ’)(‘sparrow ’)
```

Nonlocal in Functions - Side Question

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

Each call to example (‘captain ’) makes a different frame!

```
>>> example(‘captain ’)(‘jack ’)
‘captain jack ’
>>> example(‘captain ’)(‘sparrow ’)
‘captain sparrow ’
```

Nonlocal: Domo Population

```python
def domo_population(n, capacity):
    """Ronny really likes Domos, so she buys n to start with.
    They multiply at the rate given by a function at every
    timestep. However, if the function does not increase the
    number of domos, use the most recent one that did (start
    by doubling). When the number is greater than or equal to
    the capacity of her home, she gives 9/10 away to her friends.
    >>> timestep = domo_population(5, 40)
    >>> timestep(lambda x: x - 10)
    10
    >>> timestep(lambda x: x * 4)
    40
    >>> timestep(lambda x: x * 3)
    4
    """
```

Nonlocal: Domo Population (Soln.)

```python
def domo_population(n, capacity):
    increase = lambda x: x * 2
    def timestep(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 timestep
```

bonus

Nonlocal: Generate Lists

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

bonus

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

bonus

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.

bonus

~~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 really need nonlocal.

bonus

Nonlocal?: Generate Fibonacci List

```
def make_fib():
    """Generates a list of Fibonacci numbers.
    >>> fib = make_fib()
    >>> fib()
    [0]
    >>> fib()
    [0, 1]
    >>> fib()
    [0, 1, 1]
    >>> fib()
    [0, 1, 1, 2]
    """
```

bonus

~~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
```

HOF on sequences AND nonlocal?!

Write running_average. It returns a function that, when mapped over a sequence seq, replaces each index i in seq with the sum of all elements of seq up to and including index i.

```
def running_average():
    """
    >>> list(map(running_average(), [1, 3, 8, 2]))
    [1.0, 2.0, 4.0, 3.5]
    """
```

HOF on sequences AND nonlocal?!

Solution:

```
def running_average():
    """
    >>> list(map(running_average(), [1, 3, 8, 2]))
    [1.0, 2.0, 4.0, 3.5]
    """
    total, index = 0, 0
    def average(n):
        nonlocal total, index
        index += 1
        total += n
        return total / index
    return average
```

Nonlocal Summary

`x = 2`

• No nonlocal statement • 'x' **is not** bound locally	Create a new binding from name 'x' to object 2 in the first frame of the current environment.
• No nonlocal statement • 'x' **is** bound locally	Re-bind name 'x' to object 2 in the first frame of the current environment.
• nonlocal x • 'x' **is** bound in a non-local frame	Re-bind name 'x' to object 2 in the first non-local frame of the current environment in which it is bound.
• nonlocal x • 'x' **is not** bound in a non-local frame	`SyntaxError: no binding for nonlocal 'x' found`
• nonlocal x • 'x' **is** bound in a non-local frame • 'x' also bound locally	`SyntaxError: name 'x' is parameter and nonlocal`

bonus

Object-Oriented Programming: Review

- *Class*: Template for the objects of that type
- *Object*: Instance of a class
  - `__init__` is a constructor
  - All methods take `self` as the first parameter
    - When calling a method on an instance, you do not need to pass in `self`.
    - You do when calling it on a method on a class (examples later!)
- *Instance variables*: Fields of a specific instance of a class
  - For example, every `Person` will have a different name.
  - This is the **has-a** relationship.
- *Class variables* are the same for *every* instance of a class
  - For example, a count of the total number of `Persons` will not change from `Person` to `Person`.

OOP: Example #1

```
class Person(object):
    num_people = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.num_people += 1
    def has_birthday(self):
        self.age = self.age + 1
        return self.age
    def greet(self):
        return "Hi, I'm " + self.name
```

```
>>> p = Person('Paul Hilfinger',
1293842) # This calls __init__.
>>> p.greet()
"Hi, I'm Paul Hilfinger"
>>> p.has_birthday()
1293843
>>> Person.has_birthday()
has_birthday() missing 1 required
argument: 'self'
>>> Person.has_birthday(p)
1293844
>>> Person.num_people
1
>>> p.num_people
1
```

OOP: Example #2

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
```

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                      (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
_____
>>> t.if_sunny
_____
>>> t.photosynthesize
_____
>>> t.photo_fn
_____
>>> t.photosynthesize()
_____
>>> t
_____
>>> t.leaves
_____
>>> t
_____
```

OOP: Example #2

```python
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                     (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
```

```
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
_____
>>> t.photosynthesize
_____
>>> t.photo_fn
_____
>>> t.photosynthesize()
_____
>>> t
_____
>>> t.leaves
_____
>>> t
_____
```

OOP: Example #2

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
```

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                    (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
Error!
>>> t.photosynthesize
_____
>>> t.photo_fn
_____
>>> t.photosynthesize()
_____
>>> t
_____
>>> t.leaves
_____
>>> t
_____
```

OOP: Example #2

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                      (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
Error!
>>> t.photosynthesize
<bound method at ...>
>>> t.photo_fn
_____
>>> t.photosynthesize()
_____
>>> t
_____
>>> t.leaves
_____
>>> t
_____
```

OOP: Example #2

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                    (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
Error!
>>> t.photosynthesize
<bound method at ...>
>>> t.photo_fn
<function <lambda> at ...>
>>> t.photosynthesize()
_____
>>> t
_____
>>> t.leaves
_____
>>> t
_____
```

OOP: Example #2

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
```

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                   (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
Error!
>>> t.photosynthesize
<bound method at ...>
>>> t.photo_fn
<function <lambda> at ...>
>>> t.photosynthesize()
>>> t
_____
>>> t.leaves
_____
>>> t
_____
```

OOP: Example #2

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
```

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                      (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
Error!
>>> t.photosynthesize
<bound method at ...>
>>> t.photo_fn
<function <lambda> at ...>
>>> t.photosynthesize()
>>> t
Tree<10, 15.0>
>>> t.leaves
_____
>>> t
_____
```

OOP: Example #2

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                      (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
Error!
>>> t.photosynthesize
<bound method at ...>
>>> t.photo_fn
<function <lambda> at ...>
>>> t.photosynthesize()
>>> t
Tree<10, 15.0>
>>> t.leaves
11
>>> t
_____
```

OOP: Example #2

```
sunny = True
class Tree:
  energy_for_leaf = 10
  def __init__(self, leaves, if_sunny=1.5, not_sunny=0.5):
    self._leaves, self.energy = leaves, 0
    self.photo_fn = lambda leaves, sunny: leaves * \
                      (if_sunny if sunny else not_sunny)
  def photosynthesize(self):
    self.energy += self.photo_fn(self.leaves, sunny)
  @property
  def leaves(self):
    self.grow_leaves()
    return self._leaves
  def grow_leaves(self):
    while self.energy > self.energy_for_leaf:
      self._leaves += 1
      self.energy -= self.energy_for_leaf
  def __repr__(self):
    return 'Tree<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> t = Tree(10)
>>> t # repr example
Tree<10, 0>
>>> Tree.energy_for_leaf
10
>>> t.energy_for_leaf
10
>>> t.if_sunny
Error!
>>> t.photosynthesize
<bound method at ...>
>>> t.photo_fn
<function <lambda> at ...>
>>> t.photosynthesize()
>>> t
Tree<10, 15.0>
>>> t.leaves
11
>>> t
Tree<11, 5.0>
```

bonus

Inheritance: Review

- The child class is a conceptual extension of the base class -- the **is-a** relationship.
  - For example, a Dog **is a** kind of Mammal.
- Since the child class is an extension of the base class, we can often reuse code!

Inheritance: Example

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def has_birthday(self):
        self.age = self.age + 1
        return self.age
    def greet(self):
        return "Hi, I'm " + self.name
```

```
class Fireman(object):
    def __init__(self, name, age, fid):
        self.name = name
        self.age = age
        self.fid = fid
    def has_birthday(self):
        self.age = self.age + 1
        return self.age
    def greet(self):
        return "Hi, I'm " + self.name
    def put_out_fire(self):
        print(‘PUTTING OUT FIRE!’)
```

Inheritance: Example

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def has_birthday(self):
        self.age = self.age + 1
        return self.age
    def greet(self):
        return "Hi, I'm " + self.name
```

```
class Fireman(object):
    def __init__(self, name, age, fid):
        self.name = name
        self.age = age
        self.fid = fid
    def has_birthday(self):
        self.age = self.age + 1
        return self.age
    def greet(self):
        return "Hi, I'm " + self.name
    def put_out_fire(self):
        print(‘PUTTING OUT FIRE!’)
```

WRONG

WRONG

WRONG

Inheritance: Better Example

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def has_birthday(self):
        self.age = self.age + 1
        return self.age
    def greet(self):
        return "Hi, I'm " + self.name
```

```
class Fireman(Person):
    def __init__(self, name, age, fid):
        Person.__init__(self, name,
age)
        self.fid = fid
    def put_out_fire(self):
        print(‘PUTTING OUT FIRE!’)
```

```
>>> f = Fireman('John DeNero', 1230981,
1)
>>> f.name
'John DeNero'
>>> f.has_birthday()
1230982
>>> f.put_out_fire()
PUTTING OUT FIRE!
```

Immortal Professors

Using the definition of the `Person` class from before, define a `Professor` class that extends `Person`. The `Professor` class should contain a list of classes that the `Professor` is teaching.

Since `Professors` are immortal, however, when you call `has_birthday()` on a `Professor`, *nothing should change*.

Professors should also have a method, `is_teaching_class(class_name)` that returns `True` if the `Professor` is teaching `class_name`, and `False` otherwise.

Immortal Professors

```
class Person(object):
    num_people = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.num_people += 1
    def has_birthday(self):
        self.age = self.age + 1
        return self.age
    def greet(self):
        return "Hi, I'm " + self.name
```

```
class Professor(Person):
    def __init__(self, name, age, classes):
        Person.__init__(self, name, age)
        self.classes = classes
    def has_birthday(self):
        print(‘Professors are immortal!’)
    def is_teaching_class(self, class_name):
        return class_name in self.classes
```

Fruits

bonus

```
class Fruit(object):
    def __init__(self, ripe, shape):
        self.ripe = ripe
        self.shape = shape
    def is_ripe(self):
        if type(self) != Fruit:
            print(“Can’t tell!”)
        else:
            print(self.ripe)
```

```
>>> f = Fruit(True, ‘cube’)
>>> f.is_ripe()
True
```

```
class Orange(Fruit):
    def __init__(self, shape, color):
        Fruit.init(self, False, shape)
        self.color = color
    def is_ripe(self):
        if isinstance(self, Fruit):
            Fruit.is_ripe(self)
        if self.color.lower() == ‘orange’:
            print(True)
        else:
            print(False)
```

Fruits

bonus

```
class Fruit(object):
    def __init__(self, ripe, shape):
        self.ripe = ripe
        self.shape = shape
    def is_ripe(self):
        if type(self) != Fruit:
            print(“Can’t tell!”)
        else:
            print(self.ripe)

>>> o = Orange(‘round’, ‘yellow’)
>>> o.is_ripe()
Can’t tell!
False
>>> o.shape = ‘orange’
>>> o.is_ripe()
Can’t tell!
False
```

```
class Orange(Fruit):
    def __init__(self, shape, color):
        Fruit.init(self, False, shape)
        self.color = color
    def is_ripe(self):
        if isinstance(self, Fruit):
            Fruit.is_ripe(self)
        if self.color.lower() == ‘orange’:
            print(True)
        else:
            print(False)
```

Jedi

```
class Jedi(object):
    def __init__(self, name, lightsaber_color, ls_power):
        self.name = name
        self.ls_color = lightsaber_color
        self.ls_power = ls_power
    def lightsaber_duel(self, other_jedi):
        if self.ls_power > other_jedi.ls_power:
            print(self.name + ‘ defeated ’ + other_jedi.name)
        elif self.ls_power == other_jedi.ls_power:
            print(‘Tie!’)
        else:
            print(self.name + ‘ has fallen to ’ + other_jedi.name)
```

DarkJedi

```
class DarkJedi(Jedi):
    def __init__(self, name, lightsaber_color, ls_power, evil_power):
        self.name = name
        self.lightsaber_color
        self.ls_power = self.ls_power
        self.evil_power = evil_power
    def use_power(self):
        print(self.evil_power)
    def lightsaber_duel(self, other_jedi):
        "*** YOUR CODE HERE ***"
```

DarkJedi

```
class DarkJedi(Jedi):
    def __init__(self, name, lightsaber_color, ls_power, evil_power):
        Jedi.__init__(self, name, lightsaber_color, ls_power)
        self.evil_power = evil_power
    def use_power(self):
        print(self.evil_power)
    def lightsaber_duel(self, other_jedi):
        Jedi.lightsaber_duel(self, other_jedi)
```

DarkJedi

```
class DarkJedi(Jedi):
    def __init__(self, name, lightsaber_color, ls_power, evil_power):
        Jedi.__init__(self, name, lightsaber_color, ls_power)
        self.evil_power = evil_power
    def use_power(self):
        print(self.evil_power)
    ~~def lightsaber_duel(self, other_jedi):~~
        ~~Jedi.lightsaber_duel(self, other_jedi)~~
```

Sith Lord

```
class SithLord(DarkJedi):
    """
    >>> lord_vader = SithLord(‘Anakin’, red)
    >>> lord_vader.name
    ‘Darth Anakin’     # oops?
    >>> lord_vader.evil_power
    ‘force lightning’
    >>> palpatine = SithLord(‘Sidious’, red)
    >>> evil_guy = DarkJedi(‘Random Guy’, purple, 6, ‘noxious fumes’)
    >>> lord_vader.lightsaber_fight(evil_guy)
    Sith Lords always win!
    >>> lord_vader.lightsaber_fight(palpatine)
    Tie!
    """
```

Sith Lord - Cross out the incorrect lines

```
class SithLord(DarkJedi):
    def __init__(self, name, ls_color):
        DarkJedi.__init__(self, name, ls_color, ‘force lightning’)
        DarkJedi.__init__(self, name, ‘red’, ls_color, ‘force lightning’)
        DarkJedi.__init__(self, ‘Darth ’ + name, ls_color, 11, ‘force lightning’)
    def lightsaber_duel(self, other_jedi):
        if isinstance(other_jedi, DarkJedi):
        if type(other_jedi) != SithLord:
        if type(other_jedi) == SithLord:
        if type(other_jedi) == DarkJedi:
            print(‘Sith Lords always win!’)
        else:
            DarkJedi.lightsaber_duel(self, other_jedi)
```

Sith Lord - Solution

```
class SithLord(DarkJedi):
    def __init__(self, name, ls_color):
        ~~DarkJedi.__init__(self, name, ls_color, ‘force lightning’)~~
        ~~DarkJedi.__init__(self, name, ‘red’, ls_color, ‘force lightning’)~~
        DarkJedi.__init__(self, ‘Darth ’ + name, ls_color, 11, ‘force lightning’)
    def lightsaber_duel(self, other_jedi):
        ~~if isinstance(other_jedi, DarkJedi):~~
        if type(other_jedi) != SithLord:
        ~~if type(other_jedi) == SithLord:~~
        ~~if type(other_jedi) == DarkJedi:~~
            print(‘Sith Lords always win!’)
        else:
            DarkJedi.lightsaber_duel(self, other_jedi)
```

Sith Lord - Solution

```
class SithLord(DarkJedi):
    def __init__(self, name, ls_color):
        ~~DarkJedi.__init__(self, name, ls_color, ‘force lightning’)~~
        ~~DarkJedi.__init__(self, name, ‘red’, ls_color, ‘force lightning’)~~
        DarkJedi.__init__(self, ‘Darth ’ + name, ls_color, 11, ‘force lightning’)
    def lightsaber_duel(self, other_jedi):
        ~~if isinstance(other_jedi, DarkJedi):~~
        ~~if type(other_jedi) != SithLord:~~
        ~~if type(other_jedi) == SithLord:~~
        ~~if type(other_jedi) == DarkJedi:~~
        if not isinstance(other_jedi, SithLord):
            print(‘Sith Lords always win!’)
        else:
            DarkJedi.lightsaber_duel(self, other_jedi)
```

Classes

It is 2001 and you are a college student at Cal. You decide to create **Facepalm**, an application for the Palm Pilot that maintains information about different people in your address book. **Facepalm** will have a **profile** for each person. You decide to write a class called **Profile** that simulates a **Facepalm** profile. It stores a person's **name**, the person's **institution**, and a **list of profiles** of the person's friends. It also has the `add_friend(profile)` method, which adds the given profile to the list of friends’ profiles, if that profile is not already present. Your co-founder started, but we all know you’re the better programmer!

```
class Profile(object):
    def __init__(self, name, inst):
        pass
```

Classes - Solution

```
class Profile(object):
    def __init__(self, name, inst):
        self.name = name
        self.inst = inst
        self.friends = []
    def add_friend(self, profile):
        if profile not in self.friends:
             self.friends.append(profile)
```

More Classes

You aren’t exactly raking in the money that you were expecting from the app. To try to get some revenue, you decide that profiles will be restricted by default. A restricted profile can only add 100 friends, beyond which they are not able to add more friends. You then offer **`PaidProfile`**`s`, which lift this restriction.

Modify `Profile.add_friend` to implement this restriction. Also define another class `PaidProfile` to mimic the `Profile` class, except in the behavior of the `add_friend` method.

More Classes - Solution

```
class Profile(object):
    ...
    def add_friend(self, profile):
        if profile not in self.friends:
            if len(self.friends) < 100:
                self.friends.append(profile)
            else:
                print(“You have 100 friends, please upgrade!”)

class PaidProfile(Profile):
    def add_friend(self, profile):
        if profile not in self.friends:
            self.friends.append(profile)
```

Sum of Human Knowledge

What happened in Paris on October 5, 1582?

Sum of Human Knowledge

What happened in Paris on October 5, 1582?

It never happened.

To sync the calendar year with the solar year, Paris moved from the Julian Calendar to the Gregorian Calendar, where century years could only be leap years every 400 years.

This meant that October 4, 1582 was followed by October 15, 1582.

(Source: http://www.findingdulcinea.com/news/on-this-day/September-October-08/On-this-Day--In-1582--Oct--5-Did-Not-Exist-.html)

bonus

Recursive Lists (Data Abstraction)

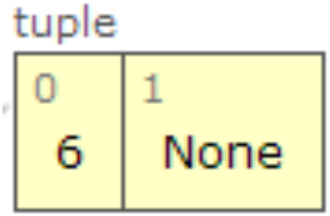
```
empty_rlist = None

def rlist(first, rest):
    return (first, rest)

def first(rlist):
    return rlist[0]

def rest(rlist):
    return rlist[1]
```

Don’t break the abstraction barriers! Use the variables and functions highlighted in blue.

bonus

Recursive Lists (Data Abstraction)

```
empty_rlist = None

def rlist(first, rest):
    return (first, rest)

def first(rlist):
    return rlist[0]

def rest(rlist):
    return rlist[1]
```



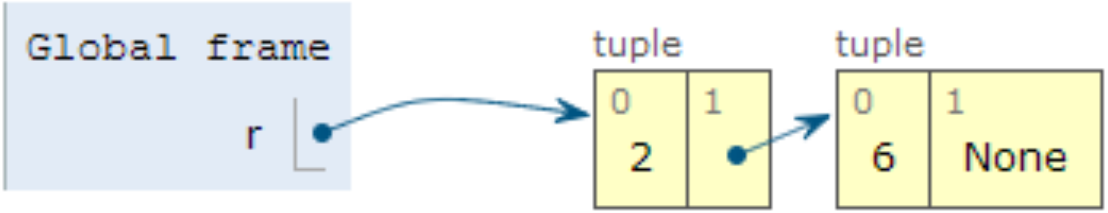
```
rlist(6, empty_rlist)
```

What if we want to add a 2 to the front?

```
rlist(2, rlist(6, empty_rlist))
```

bonus

Recursive Lists (Data Abstraction)



```
r = rlist(2, rlist(6, empty_rlist))
```

How do we change the 2 to a 3?

We want to keep: `rlist(6, empty_rlist)`

```
r = rlist(3, rest(r))
```

bonus

Recursive Lists (Data Abstraction)

Write reduce:

```
def reduce(rlst, combiner, default):
    """
    >>> r = rlist(1, rlist(2, empty_rlist))
    >>> reduce(r, lambda x, y: x + y, 0)
    3
    """
```

bonus

Recursive Lists (Data Abstraction)

Write reduce:

```
def reduce(rlst, combiner, default):
    if rlst == empty_rlist:
        return default
    return combiner(first(rlst),
                    reduce(rest(rlst))
```

bonus

Recursive Lists (Data Abstraction)

Write `filter`:

```
def filter(rlst, pred):
    """
    >>> r = rlist(1, rlist(2, empty_rlist))
    >>> filter(r, lambda x: x % 2 == 0)
    (2, None)
    """
```

Note that you should never represent an rlist like this. This is: `rlist(2, empty_rlist)`

bonus

Recursive Lists (Data Abstraction)

Write `filter`:

```
def filter(rlst, pred):
    if rlst == empty_rlist:
        return empty_rlist
    if pred(first(rlst)):
        return rlist(first(rlst),
                     filter(rest(rlst), pred))
    return filter(rest(rlst), pred)
```

bonus

Recursive Lists (Data Abstraction)

```
def reverse(r):
    """
    >>> r = rlist(1, rlist(2, empty_rlist))
    >>> reverse(r)
    (2, (1, None))
    """
```

This is:
`rlist(2, rlist(1, empty_rlist))`

bonus

Recursive Lists (Data Abstraction)

```
def reverse(r):
    if r == empty_rlist:
        return empty_rlist
    last_item = get_last(r)
    all_but_last = remove_last(r)
    return rlist(last_item, reverse(all_but_last))
```

```
def get_last(r):
  if r == empty_rlist:
    return empty_rlist
  if rest(r) == empty_rlist:
    return first(r)
  return get_last(rest(r))
```

```
def remove_last(r):
  if r == empty_rlist:
    return empty_rlist
  if rest(r) == empty_rlist:
    return empty_rlist
  return rlist(first(r),
               remove_last(rest(r)))
```

Recursive Lists (OOP)

```
class Rlist:
    class EmptyRlist:
        def __len__(self):
            return 0

    empty = EmptyRlist()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

Recursive Lists (OOP)

```
class Rlist:
    class EmptyRlist:
        def __len__(self):
            return 0
    empty = EmptyRlist()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

Make an Rlist with a 2 in it?

```
Rlist(2)
```

An Rlist with 1 then 2 in it?

```
Rlist(1, Rlist(2))
```

Recursive Lists (OOP)

```
r = Rlist(1, Rlist(2, Rlist(3)))
```

How do we retrieve the 1?

```
r.first
```

Retrieve the 2?

```
r.rest.first
```

Change the 2 to a 5?

```
r.rest.first = 5
```

Recursive Lists (OOP)

Define a procedure `skip_consecutives` that, given an Rlist of numbers, removes the consecutive duplicates with mutation.

```
def skip_consecutives(r):
    """
    >>> r = Rlist(1, Rlist(1, Rlist(3,
                  Rlist(2, Rlist(1)))))
    >>> skip_consecutives(r)
    # r is now Rlist(1, Rlist(3, Rlist(2, Rlist(1))))
    """
```

Recursive Lists (OOP)

Define a procedure `skip_consecutives` that, given an Rlist of numbers, removes the consecutive duplicates with mutation.

```python
def skip_consecutives(r):
    if r is Rlist.empty:
        return
    current = r.rest
    while current is not Rlist.empty \
          and r.first == current.first:
        r.rest = r.rest.rest
        current = r.rest
    skip_consecutives(r.rest)
```

Recursive Lists (OOP)

```
def reverse(r):
    """
    >>> r = Rlist(1, Rlist(2))
    >>> reverse(r)
    # r is now Rlist(2, Rlist(1))
    """
```

Recursive Lists (OOP)

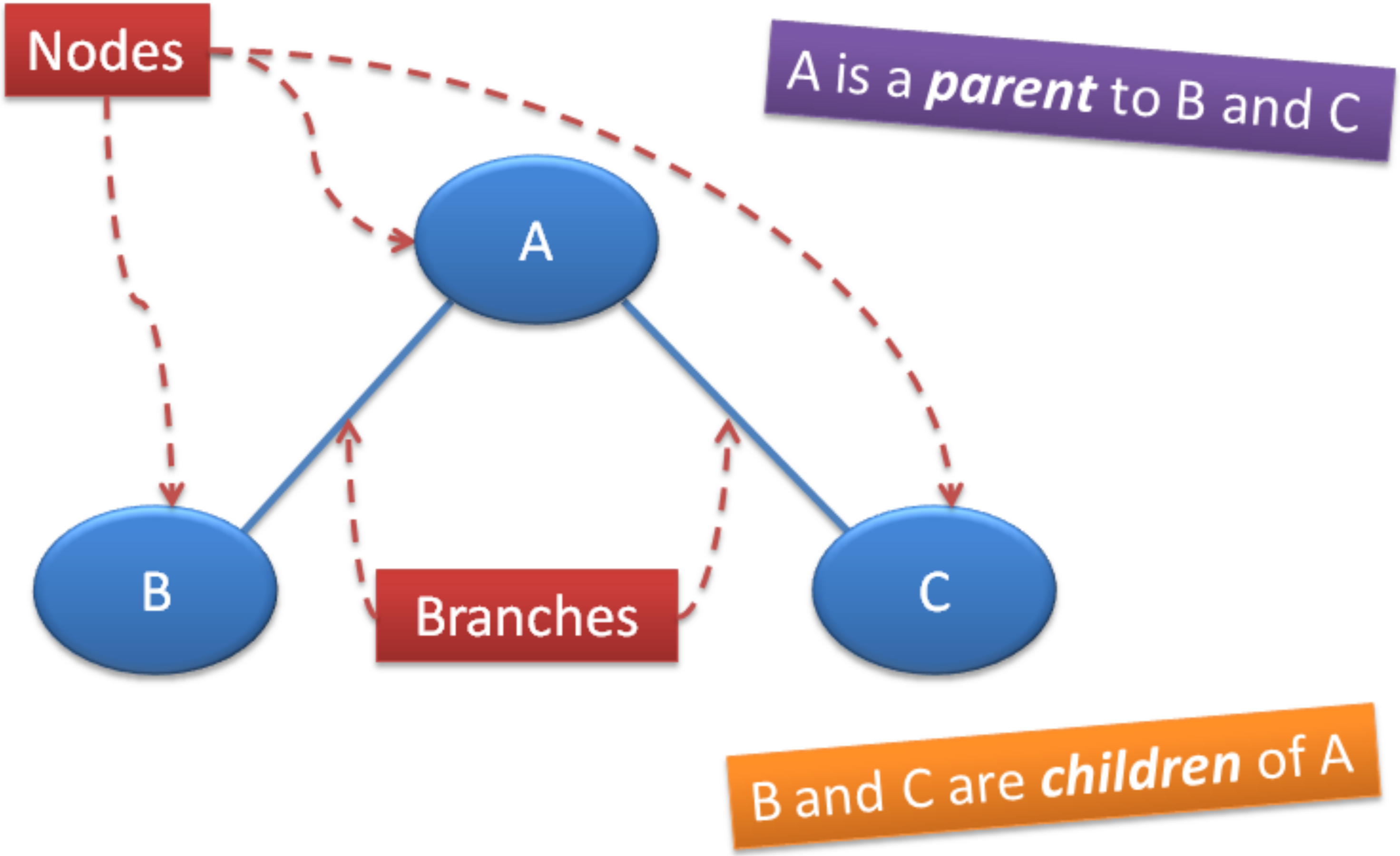
```
def reverse(r):
    if r is Rlist.empty:
        return Rlist.empty
    last_item = get_last(r)
    all_but_last = remove_last(r)
    return Rlist(last_item, reverse(all_but_last))
```

```
def get_last(r):
  if r is Rlist.empty:
    return Rlist.empty
  if rest(r) is Rlist.empty:
    return first(r)
  return get_last(rest(r))
```

```
def remove_last(r):
  if r is Rlist.empty:
    return Rlist.empty
  if rest(r) is Rlist.empty:
    return Rlist.empty
  return Rlist(first(r),
               remove_last(rest(r)))
```

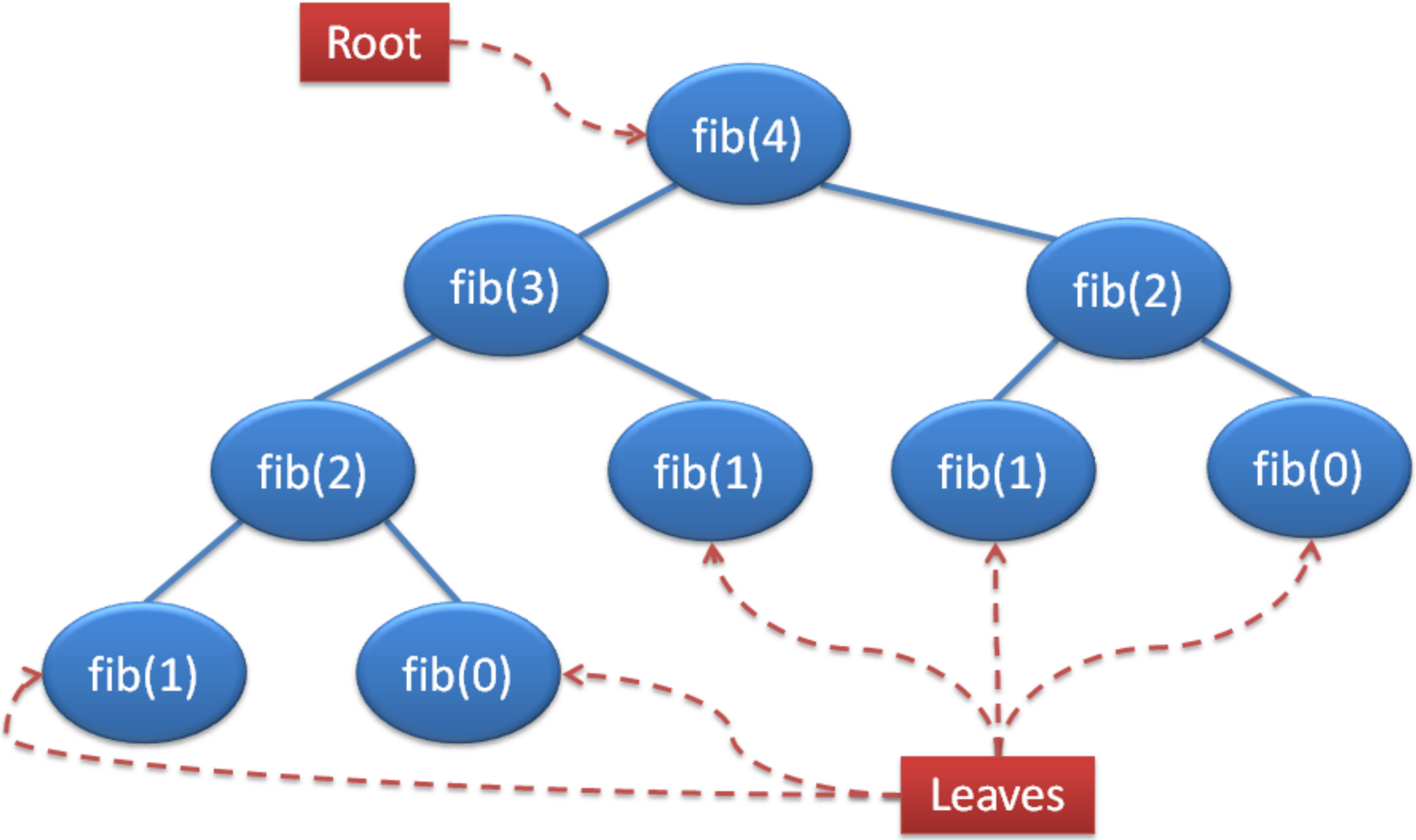
bonus

Trees: Review



bonus

Trees: Review



bonus

Trees: Review

```
class Tree(object):
    def __init__(self, label, *children):
        self._label = label
        self._children = list(children)

    @property
    def label(self):
        return self._label


    def __iter__(self):
        """An iterator over my children."""
        return iter(self._children)


    def __getitem__(self, k):
        """My kth child."""
        return self._children[k]
```

bonus

Trees: Review

```
class Tree(object):
    ...
    @property
    def is_leaf(self):
        return self.arity == 0

    @property
    def arity(self):
        """The number of my children."""
        return len(self._children)
```

bonus

Trees: Review

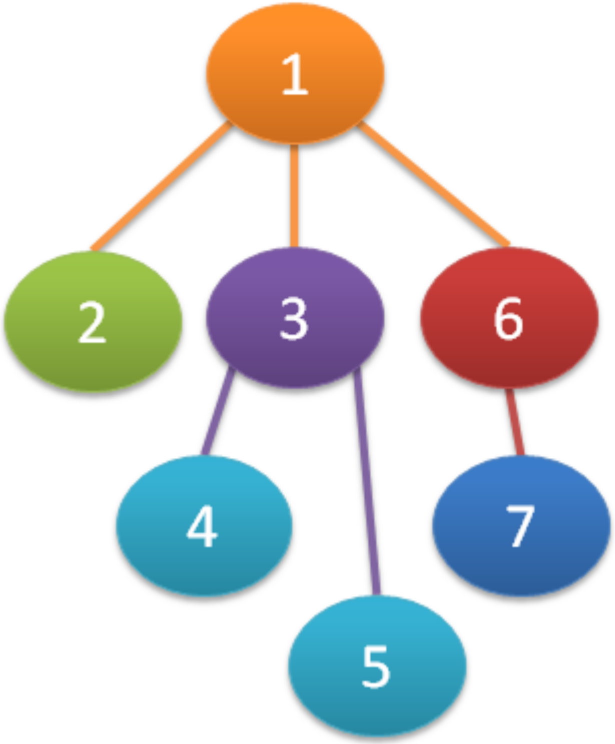
```
Tree(1,
    Tree(2),
    Tree(3,
            Tree(4),
            Tree(5)),
    Tree(6,
            Tree(7)))
```

Root

No children

Children
(each is a Tree!)

Only one child



bonus

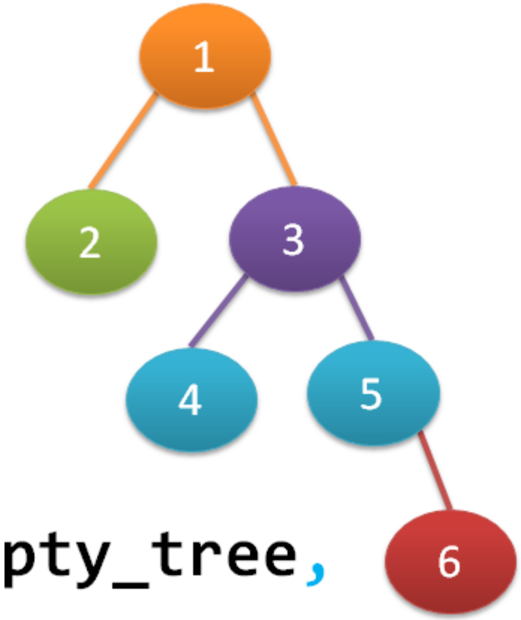
Trees: Review

```
class BinTree(Tree):
    def __init__(self, label, left=None, right=None):
        Tree.__init__(self, label,
                      left or BinTree.empty_tree,
                      right or BinTree.empty_tree)

    @property
    def left(self):
        return self[0]   # or self.__getitem__(0)

    @property
    def right(self):
        return self[1]


# Make BinTree.empty_tree be an arbitrary node with no children
BinTree.empty_tree = BinTree(None)
```

bonus

Trees: Review

```
BinTree(1,
        BinTree(2),
        BinTree(3,
                BinTree(4)
                BinTree(5,
                        BinTree.empty_tree,
                        BinTree(6))))
```



bonus

Trees: Review

Notice that trees are also *recursively defined*.

A tree is made from other trees –

these trees are its *subtrees*.

Thus, a general strategy to write functions that operate on tree problems is *recursively*:

Apply the function on the subtrees and

combine the results in a relevant way.

Trees

Write a function `tree_equals` that takes in two `BinTrees` that contain integers and returns `True` if the binary trees have the same ‘shape’ and the corresponding nodes have the same values.

```
def tree_equals(t1, t2):
```

Trees

```
def tree_equals(t1, t2):
    if t1 is BinTree.empty_tree and \
       t2 is BinTree.empty_tree:
        return True
    if t1 is BinTree.empty_tree or \
       t2 is BinTree.empty_tree:
        return False
    left_equals = tree_equals(t1.left, t2.left)
    right_equals = tree_equals(t1.right, t2.right)
    return t1.label == t2.label and \
           left_equals and right_equals
```

Trees

Write the function `prod_tree`, which takes a `BinTree` of numbers and returns the product of all the numbers in the `BinTree`.

```
>>> t = BinTree(1, None, BinTree(2,
                                 BinTree(3),
                                 BinTree(4)))
>>> prod_tree(t)
24
```

Trees

Write the function `prod_tree`, which takes a `BinTree` of numbers and returns the product of all the numbers in the `BinTree`.

```
def prod_tree(t):
    if t is BinTree.empty_tree:
        return 1
    return t.label * prod_tree(t.left) *
\
                     prod_tree(t.right)
```

Trees

Write the function `prod_tree`, which takes a `Tree` of numbers and returns the product of all the numbers in the `Tree`.

```
>>> t = Tree(1, Tree(2), Tree(3, Tree(4,
                                      Tree(5),
                                      Tree(6))))
>>> prod_tree(t)
720
```

Trees

Write the function prod_tree, which takes a Tree of numbers and returns the product of all the numbers in the Tree.

```
def prod_tree(t):
    result = t.label
    for child in t:
        result *= prod_tree(child)
    return result
```

Trees

Write the function `prod_tree`, which takes a `Tree` of numbers and returns the product of all the numbers in the `Tree`.

```
from functools import reduce
from operator import mul
def prod_tree(t):
    return reduce(mul,
                  map(prod_tree, t),
                  t.label)
```

Trees

Write the function `sum_filter_tree`, which takes a `BinTree` of numbers and returns the sum of all the numbers in the `BinTree` that satisfy the given predicate.

```
>>> t = BinTree(1, BinTree(2, BinTree(3), BinTree(4)))
>>> is_even = lambda x: x % 2 == 0
>>> sum_filter_tree(t, is_even)
6
```

Trees

Write the function `sum_filter_tree`, which takes a `BinTree` of numbers and returns the sum of all the numbers in the `BinTree` that satisfy the given predicate.

```
def sum_filter_tree(t, pred):
    if t is BinTree.empty_tree:
        return 0
    subtree_sums = sum_filter_tree(t.left, pred) + \
                   sum_filter_tree(t.right, pred)
    if pred(t.label):
        return t.label + subtree_sums
    return subtree_sums
```

Trees

Write the function `sum_filter_tree`, which takes a `Tree` of numbers and returns the sum of all the numbers in the `Tree` that satisfy the given predicate.

```
>>> t = Tree(1, Tree(2), Tree(3, Tree(4, Tree(5), Tree(6))))
>>> is_even = lambda x: x % 2 == 0
>>> sum_filter_tree(t, is_even)
6
```

Trees

Write the function `sum_filter_tree`, which takes a `Tree` of numbers and returns the sum of all the numbers in the `Tree` that satisfy the given predicate.

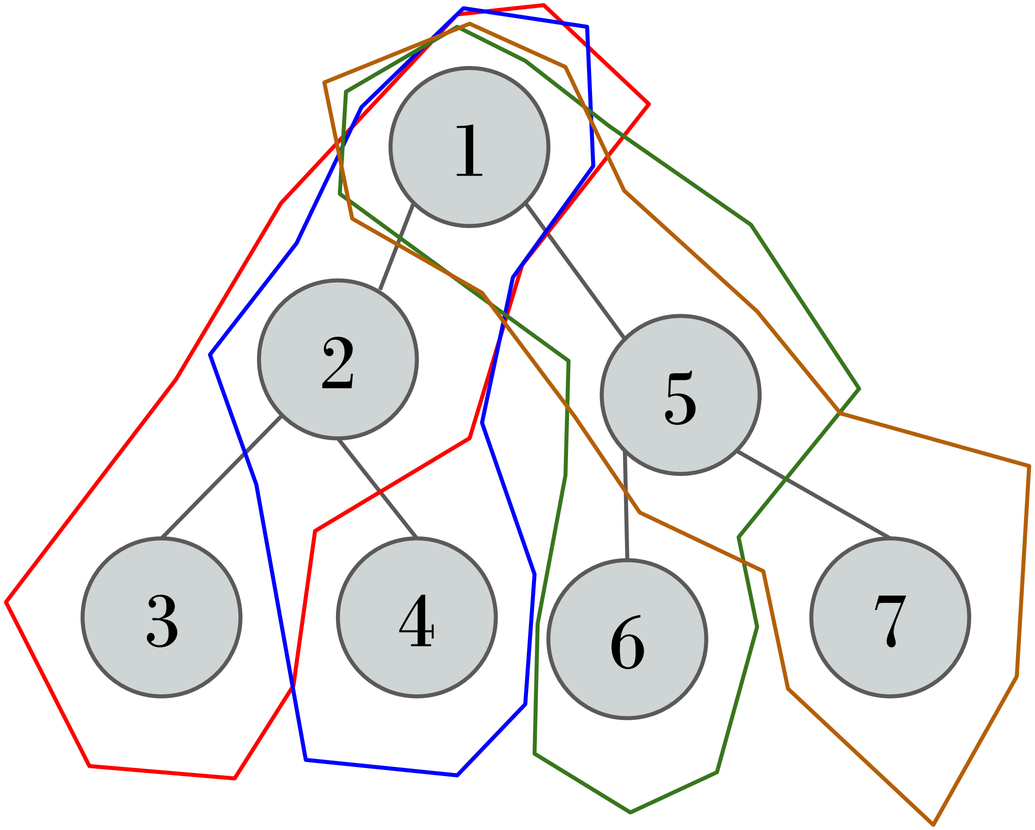
```
def sum_filter_tree(t, pred):
    result = 0
    if pred(t.label):
        result += t.label
    for child in t:
        result += sum_filter_tree(child, pred)
    return result
```

Trees

Write a function `all_paths` that takes in a `BinTree` and returns a list of tuples, where each nested tuple is a path from the root to a leaf.

```
>>> all_paths(t)
[(1, 2, 3), (1, 2, 4), (1, 5, 6), (1, 5, 7)]
```

Trees



```
>>> all_paths(t)
[(1, 2, 3), (1, 2, 4), (1, 5, 6), (1, 5, 7)]
```

Trees

```
def all_paths(t):
    if t is BinTree.empty_tree:
        return []
    if t.is_leaf:
        return [(t.label,)]

    paths_in_left = all_paths(t.left)
    paths_in_right = all_paths(t.right)
    result = []
    for path in paths_in_left + paths_in_right:
        result.append((t.label,) + path)
    return result
```

Orders of Growth

Keep in mind what happens as n grows large.

What is the order of growth for this function?

```
def func(n):
   for i in range(n):
      print(i)
   return n
```

Orders of Growth

Keep in mind what happens as n grows large.

What is the order of growth for this function?

```
def func(n):
    for i in range(n):
        print(i)
    return n
```

O(n)

Orders of Growth

Keep in mind what happens as n grows large.

What is the order of growth for this function?

```
def func(n):
    for i in range(n // 2):
        print(i)
    return n
```

Orders of Growth

Keep in mind what happens as n grows large.

What is the order of growth for this function?

```
def func(n):
    for i in range(n // 2):
        print(i)
    return n
```

O(n)

Orders of Growth

What is the order of growth for this function?

```
def func1(n):
    return n + func1(n - 1) if n > 1 else n

def func2(n):
    if n <= 1:
        return 5
    sum = 0
    for i in range(n):
        sum += func1(n)
    return sum + func2(n - 1)
```

Orders of Growth

What is the order of growth for this function?

```
def func1(n):
    return n + func1(n - 1) if n > 1 else n

def func2(n):
    if n <= 1:
        return 5
    sum = 0
    for i in range(n):
        sum += func1(n)
    return sum + func2(n - 1)
```

O(n^3)

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    if n <= 1:
        return n
    return func(n - 1) + func(n - 2)
```

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    if n <= 1:
        return n
    return func(n - 1) + func(n - 2)
```

$O(2^n)$

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    if n <= 1:
        return n
    return 1 + func(n // 2)
```

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    if n <= 1:
        return n
    return 1 + func(n // 2)
```

O(log n)

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    if n <= 1:
        return 1
    if n <= 50:
        return func(n - 1) + func(n - 2)
    elif n > 50:
        return func(50) + func(49)
```

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    if n <= 1:
        return 1
    if n <= 50:
        return func(n - 1) + func(n - 2)
    elif n > 50:
        return func(50) + func(49)
```

O(1)

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    lst = []
    for i in range(n):
        lst.append(i)
        # Order of growth of ‘append’ is O(1) in the length of the list.
    if n <= 1:
        return 1
    if n <= 50:
        return func(n - 1) + func(n - 2)
    elif n > 50:
        return func(50) + func(49)
```

Orders of Growth

What is the order of growth for this function?

```
def func(n):
    lst = []
    for i in range(n):
        lst.append(i)
        # Order of growth of ‘append’ is O(1) in the length of the list.
    if n <= 1:
        return 1
    if n <= 50:
        return func(n - 1) + func(n - 2)
    elif n > 50:
        return func(50) + func(49)
```

O(n)

Orders of Growth

```
def foo(x, y):
    if x == 0:
        return 1
    if y > 0:
        return foo(x, y - 1)
    return 1 + foo(x // 2, y)
```

```
def baz(z):
    return abs(z)
```

What is the order of growth in time for `foo(x, baz(y))` with respect to x?

What is the order of growth in time for `foo(x, baz(y))` with respect to y?

Orders of Growth

```
def foo(x, y):
    if x == 0:
        return 1
    if y > 0:
        return foo(x, y - 1)
    return 1 + foo(x // 2, y)
```

```
def baz(z):
    return abs(z)
```

What is the order of growth in time for `foo(x, baz(y))` with respect to x?

O(log x)

What is the order of growth in time for `foo(x, baz(y))` with respect to y?

Orders of Growth

```
def foo(x, y):
    if x == 0:
        return 1
    if y > 0:
        return foo(x, y - 1)
    return 1 + foo(x // 2, y)
```

```
def baz(z):
    return abs(z)
```

What is the order of growth in time for `foo(x, baz(y))` with respect to x?

O(log x)

What is the order of growth in time for `foo(x, baz(y))` with respect to y?

O(y)

Orders of Growth

```
def foo(x, y):
    if x == 0:
        return 1
    if y > 0:
        return foo(x, y - 1)
    return 1 + foo(x // 2, y)
```

```
def baz(z):
    return abs(z)
```

What is the order of growth in time for `foo(x, baz(y))` with respect to x?

O(log x)

What is the order of growth in time for `foo(x, baz(y))` with respect to y?

O(y)

What is the order of growth in time for `foo(x, baz(y))` with respect to x *and* y?

Orders of Growth

```
def foo(x, y):
    if x == 0:
        return 1
    if y > 0:
        return foo(x, y - 1)
    return 1 + foo(x // 2, y)
```

```
def baz(z):
    return abs(z)
```

What is the order of growth in time for `foo(x, baz(y))` with respect to x?

O(log x)

What is the order of growth in time for `foo(x, baz(y))` with respect to y?

O(y)

What is the order of growth in time for `foo(x, baz(y))` with respect to x *and* y?

O(y + log(x))