HKN CS 61A Midterm 2 Review

Austin Le
Sherdil Niyaz
Joong Hwa Lee

Hello!

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 (hkn.eecs.berkeley.edu/tutor)

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. Some members of the course staff may be presenting, but this review is *still not* official.

Agenda

- Lists, Tuples, Dictionaries, Sequences
- Data Abstraction
- Nonlocal
- Object-Oriented Programming
- Inheritance
- Linked Lists
- Trees
- Orders of Growth

Follow along: **http://tinyurl.com/hkn-cs61a-mt2**

Iterables

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

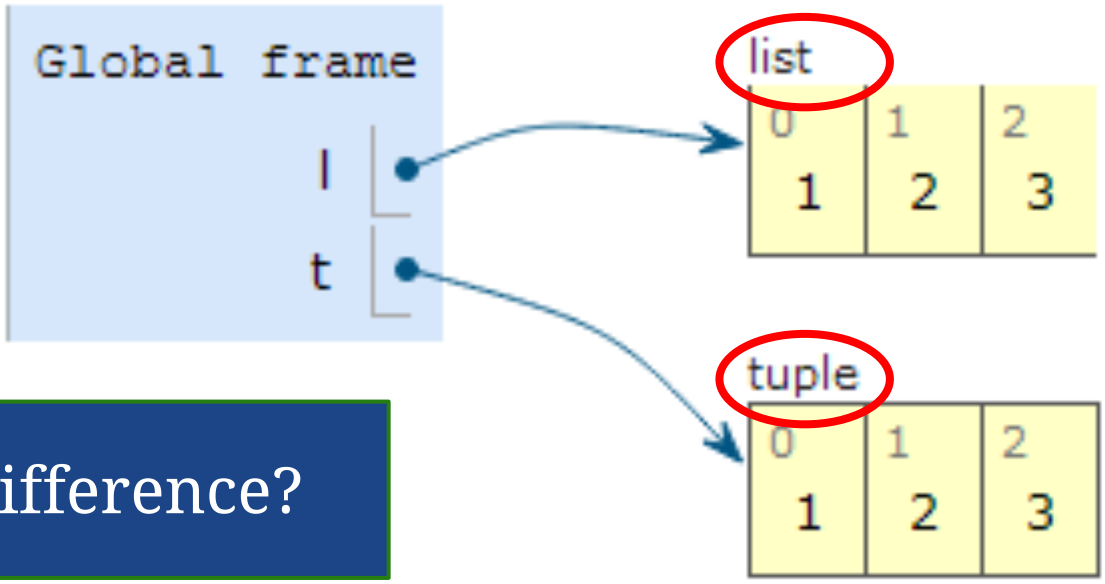
- *Tuples*: Sequences that are **immutable.** We ***cannot*** change the items in a tuple; we can only create new tuples.

- *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.

Box & Pointer Diagrams

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

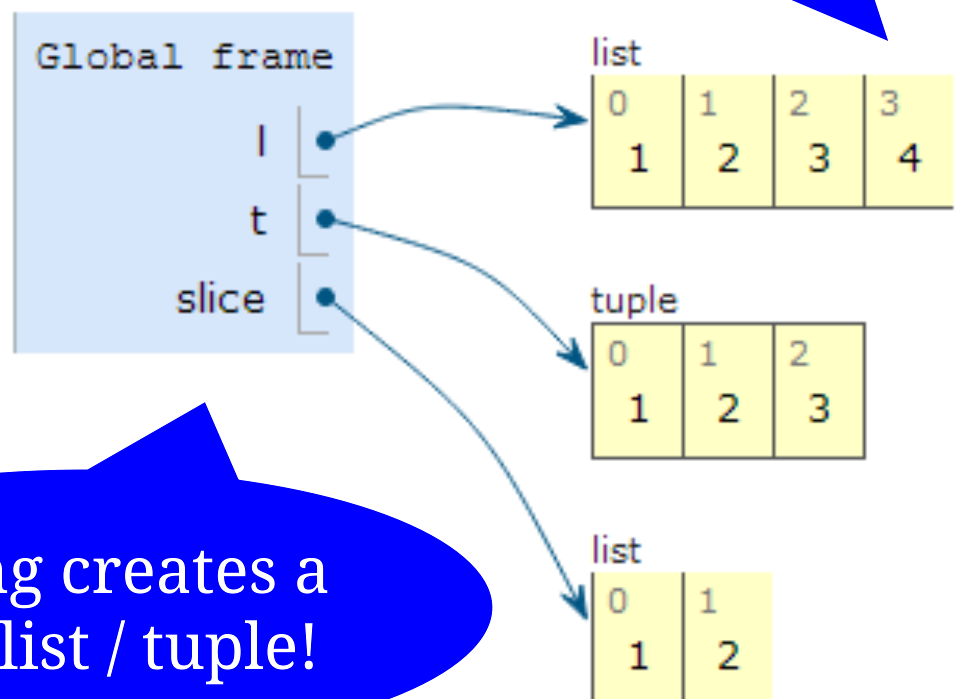


What's the difference?

Box & Pointer Diagrams

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

append() mutates the original list.



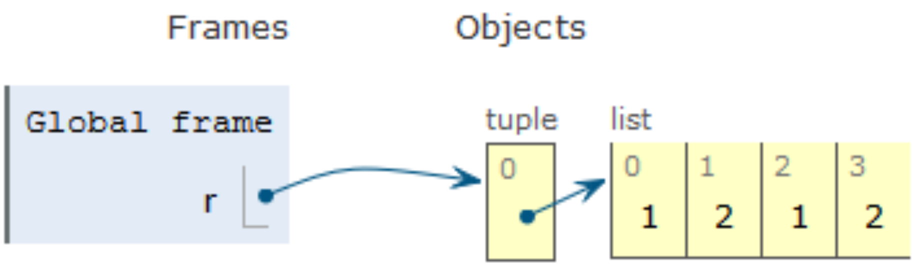
Slicing creates a new list / tuple!

Draw the Box & Pointer 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
```

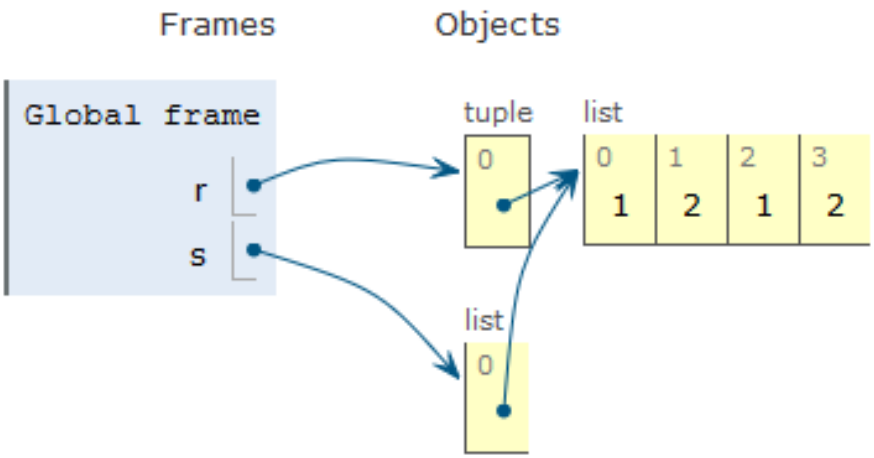
Draw the Box & Pointer Diagram!

```
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
```



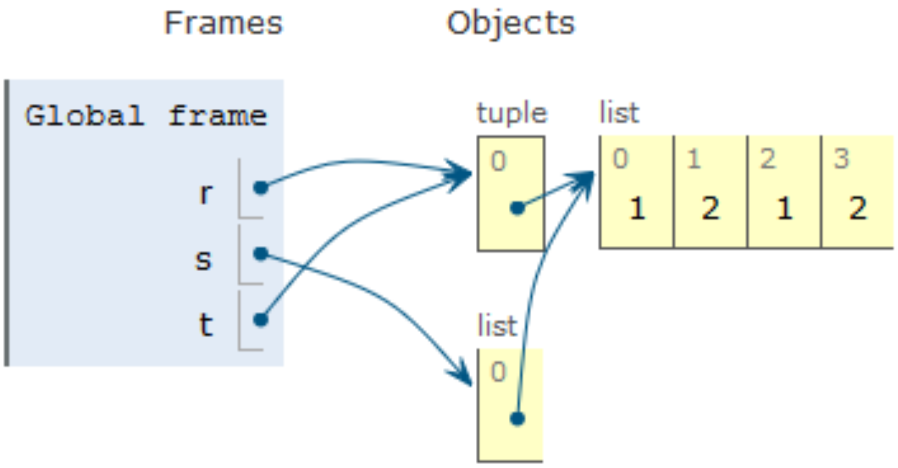
Draw the Box & Pointer Diagram!

```
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
```



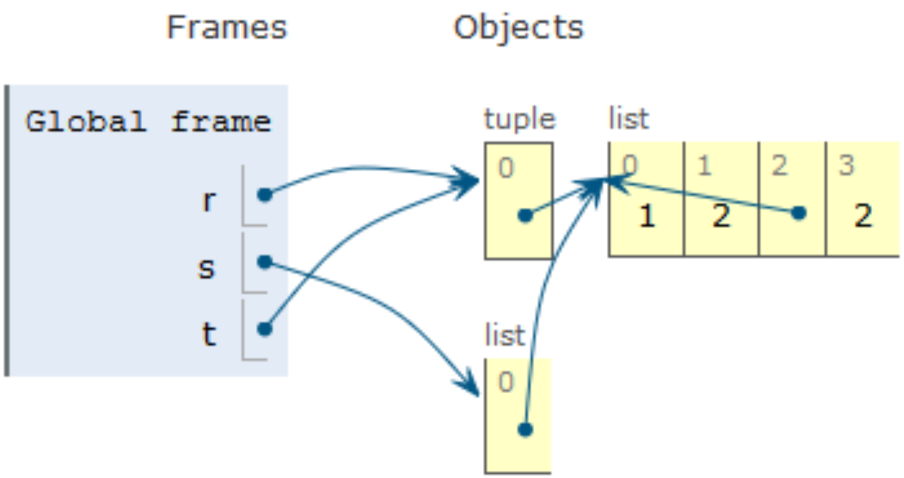
Draw the Box & Pointer Diagram!

```
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
```



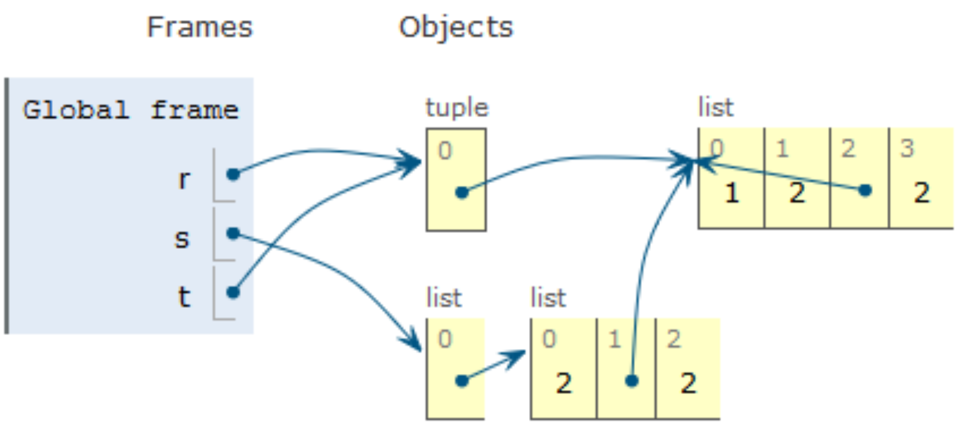
Draw the Box & Pointer Diagram!

```
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
```



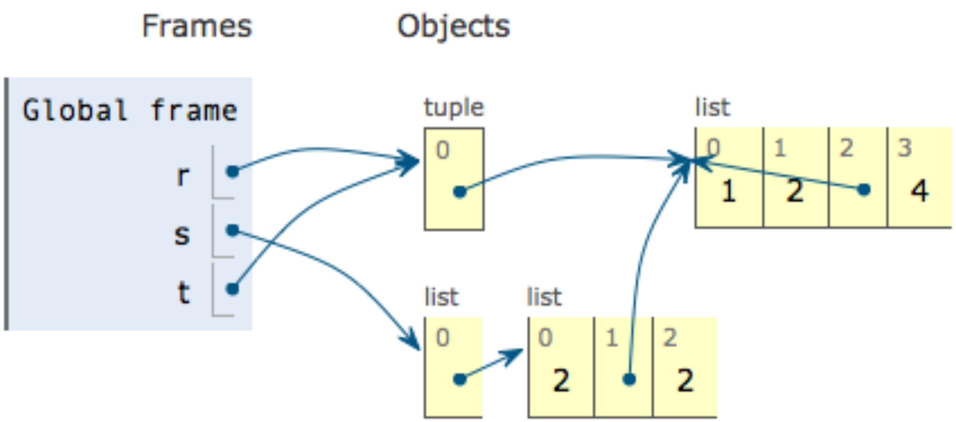
Draw the Box & Pointer Diagram!

```
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
```



Draw the Box & Pointer 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
```



Lists : Scope

```
lst = [1, 2, 3, 4, 5]
def add_five(to_be_changed):
   for item in to_be_changed:
      item += 5

>> add_five(lst)
>> lst
```

What would be the result?

Lists : Scope

```
lst = [1, 2, 3, 4, 5]
def add_five(to_be_changed):
   for item in to_be_changed:
      item += 5


>> add_five(lst)
>> lst
```

What would be the result?

`[1, 2, 3, 4, 5]` because only the local variable `item` is modified, not the actual elements themselves in `lst`

Lists : Scope

```
lst = [1, 2, 3, 4, 5]
def add_five(to_be_changed):
    for i in range(0, len(to_be_changed)):
        to_be_changed[i] += 5

>> add_five(lst)
>> lst
```

What would be the result?

`[6, 7, 8, 9, 10]` yay!

List Comprehensions Example

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

______
>>> [w[i:] for w in words for i in [1, 2]]

______
```

List Comprehensions Example

```
>>> words = "We love CS61A!".split()
>>> words
['We', 'love', 'CS61A!']
>>> [len(w) for w in words]
[2, 4, 6]
>>> [w[i:] for w in words for i in [1, 2]]
______
```

List Comprehensions Example

```
>>> words = "We love CS61A!".split()
>>> words
['We', 'love', 'CS61A!']
>>> [len(w) for w in words]
[2, 4, 6]
>>> [w[i:] for w in words for i in [1, 2]]
['e', '', 'ove', 've', 's61A!', '61A!']
```

Dictionary Comprehensions

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
```

______

Dictionary Comprehensions

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}
```

Dictionary Comprehensions

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?

What would we do if we wanted to have all the values above in the dictionary?

Dictionary Comprehensions

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.

What would we do if we wanted to have all the values above in the dictionary? **The value for each key would be a list.**

Sequences

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

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

Output - Sequence of the same length as the input.

Example:

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

Sequences

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.

Example:

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

Sequences

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
```

Sequences

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?

Sequences

```
>>> 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
```

Sequences

```
>>> 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
```

Sequences

```
>>> 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 to get `123456`.

Sequences

```
>>> cool = 'denero'
>>> story = [cool[i:2*i] for i in range(6)]
>>> story

______
>>> bro = apply_to_all(len, story)
>>> bro

______
```

Sequences

```
>>> cool = 'denero'
>>> story = [cool[i:2*i] for i in range(6)]
>>> story
['', 'e', 'ne', 'ero', 'ro', 'o']
>>> bro = apply_to_all(len, story)
>>> bro
_____
```

Sequences

```
>>> cool = 'denero'
>>> story = [cool[i:2*i] for i in range(6)]
>>> story
['', 'e', 'ne', 'ero', 'ro', 'o']
>>> bro = apply_to_all(len, story)
>>> bro
[0, 1, 2, 3, 2, 1]
```

Sequences

keep_if - 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 returns True or False, and any iterable sequence (list, tuple, etc.).

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

For example:

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

Sequences

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> apply_to_all(is_prime, keep_if(is_prime,
fib))
_____
>>> get_fib = lambda x: fib[x]
>>> apply_to_all(get_fib, keep_if(is_prime,
fib))
_____
```

Sequences

\>>> primes = [2, 3, 5, 7, 11]

\>>> fib = [0, 1, 1, 2, 3]

\>>> is_prime = lambda x: x in primes

\>>> apply_to_all(is_prime, ~~keep_if(is_prime, fib)~~ [2, 3])

______

\>>> get_fib = lambda x: fib[x]

\>>> apply_to_all(get_fib, keep_if(is_prime, fib))

______

Sequences

```
>>> primes = [2, 3, 5, 7, 11]
>>> fib = [0, 1, 1, 2, 3]
>>> is_prime = lambda x: x in primes
>>> apply_to_all(is_prime, keep_if(is_prime, fib)))                        [2, 3]
[True, True]
>>> get_fib = lambda x: fib[x]
>>> apply_to_all(get_fib, keep_if(is_prime, fib))
_____
```

Sequences

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

_____
```

Sequences

\>>> primes = [2, 3, 5, 7, 11]

\>>> fib = [0, 1, 1, 2, 3]

\>>> is_prime = lambda x: x in primes

\>>> apply_to_all(is_prime, ~~keep_if(is_prime, fib)~~)) [2, 3]

[True, True]

\>>> get_fib = lambda x: fib[x]

\>>> apply_to_all(get_fib, ~~keep_if(is_prime, fib)~~) [2, 3]

intermediate step [get_fib(2), get_fib(3)]

Sequences

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

Sequences

\>>> primes = [2, 3, 5, 7, 11]

\>>> fib = [0, 1, 1, 2, 3]

\>>> is_prime = lambda x: x in primes

\>>> apply_to_all(is_prime, ~~keep_if(is_prime, fib)~~)) [2, 3]

[True, True]

\>>> get_fib = lambda x: fib[x]

\>>> apply_to_all(get_fib, ~~keep_if(is_prime, fib)~~) [2, 3]

intermediate step [fib[2], fib[3]]

[1, 2]

Data Abstraction

http://xkcd.com/676/

Data Abstraction

How data is used

How data is internally represented

Abstraction Barrier

Data Abstraction 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)
```

Constructor - Builds an object of the abstract data type.

Selector - Extracts relevant information from the object.

Write these functions to complete the segment data abstraction!

`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)
```

Write these functions to complete the segment data abstraction! (sol'n)

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

Write these functions to complete the segment data abstraction! (sol’n)

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

Write these functions to complete the segment data abstraction! (sol'n)

```
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))
```

Write these functions to complete the segment data abstraction! (sol'n)

```
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)
```

Write these functions to complete the segment data abstraction! (sol'n)

```
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))))
```

Fix this!

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][0]
    ret = dist(prev, path[0][1])
    for (s, cur) in path[1:]:
        if s != prev:
            return None
        else:
            ret += dist(s, cur)
        prev = cur
    return ret
```

Fix this! (sol'n)

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.

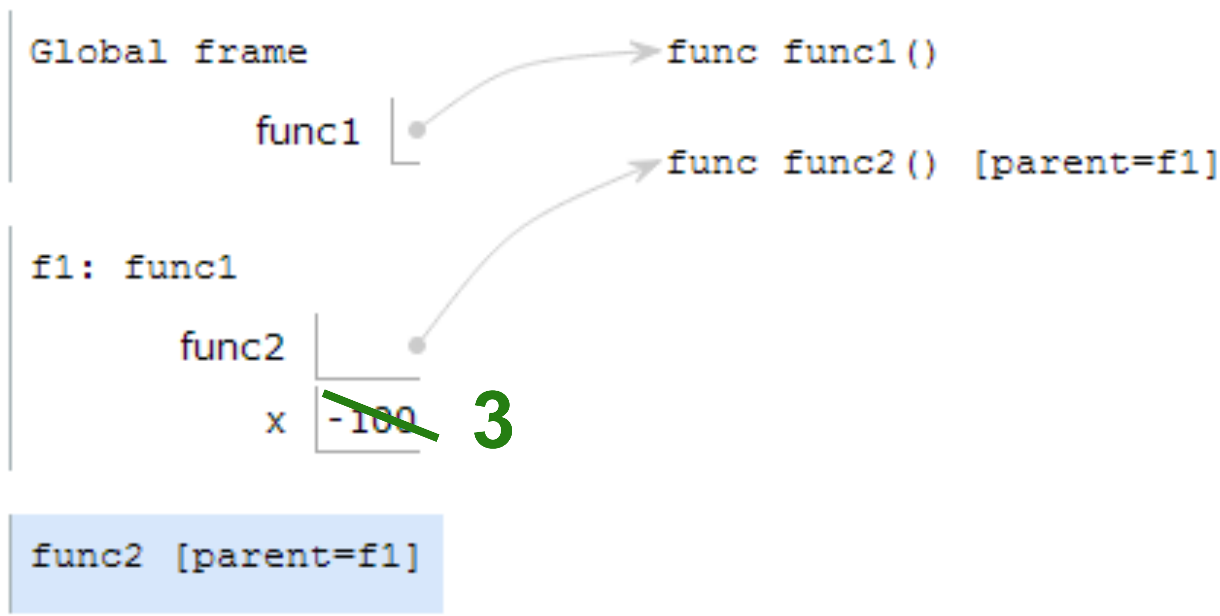
```
# Assume path is a tuple of line segments.
def path_length(path):
    prev = path[0]~~[0]~~
    ret = ~~dist(prev, path[0][1])~~ length(prev)
    ~~for (s, cur) in path[1:]:~~ for cur in path[1:]:
        ~~if s != prev:~~ if not consecutive(prev, cur):
            return None
        else:
    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

Nonlocal

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.

x is a local variable (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

Don't need nonlocal to mutate something!

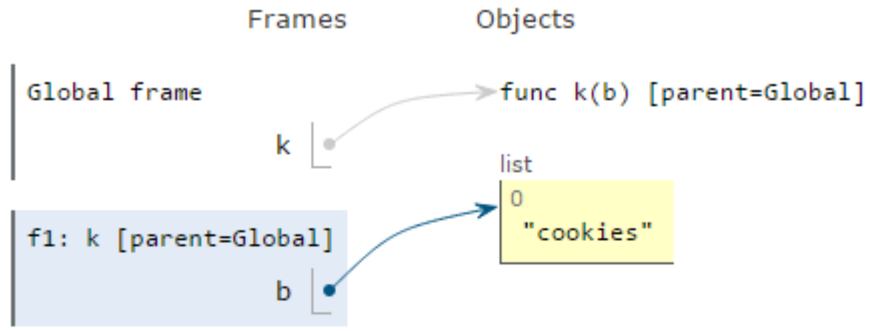
```
def k(b):
    def seven(up):
        b.extend(['<3','<3'])
        nonlocal b
        b = 5
        up[0][0] = 'cs61a'
        return up[0:2]
    return seven((b, 3, 6))

k(['cookies'])
```

Need nonlocal to change what value a variable points to

Slicing creates a new list with the same values.

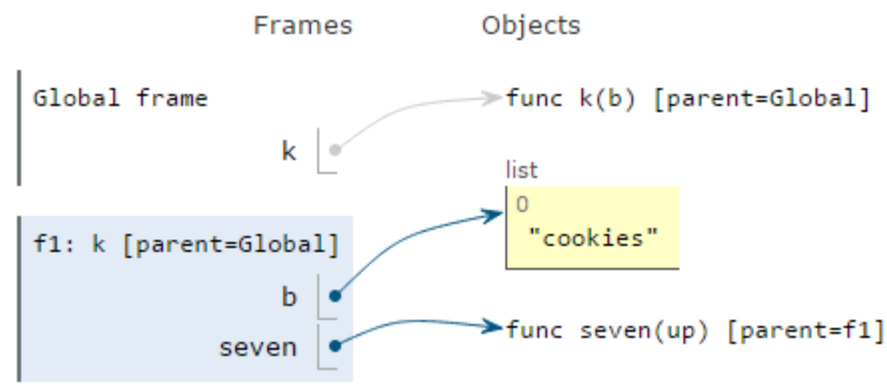
Draw the Environment Diagram

```
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'])
```



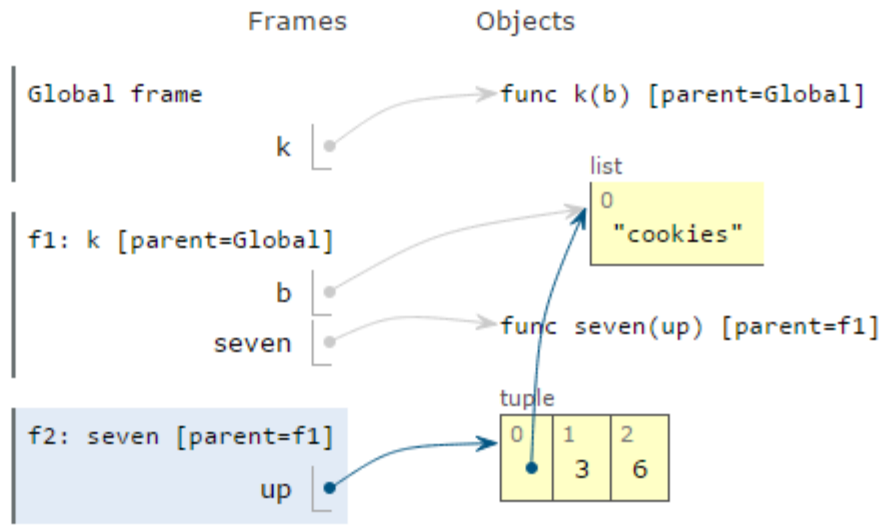
Draw the Environment Diagram

```
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'])
```



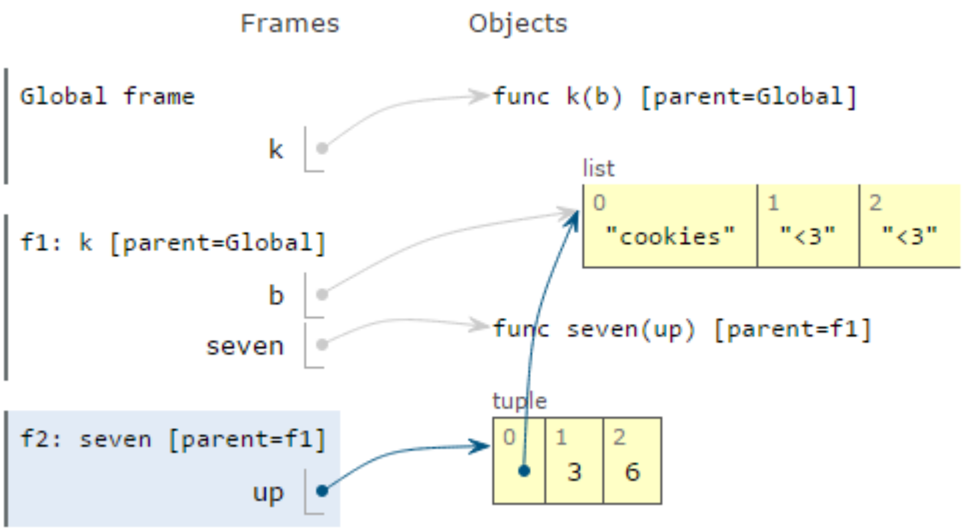
Draw the Environment Diagram

```
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'])
```



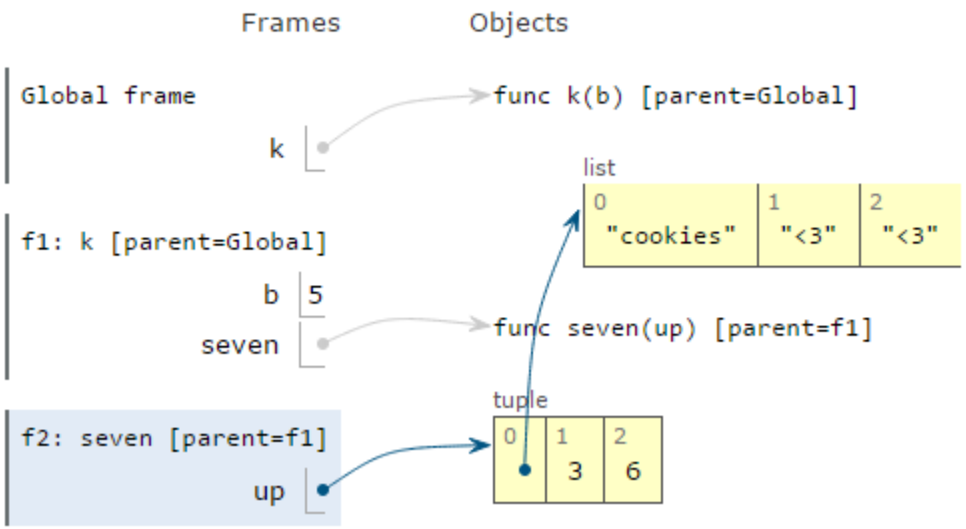
Draw the Environment Diagram

```
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'])
```



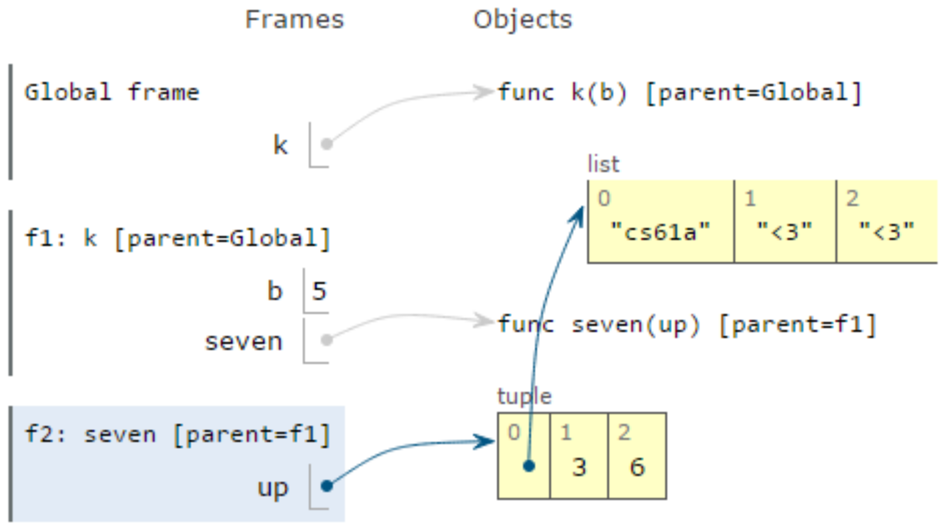
Draw the Environment Diagram

```
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'])
```



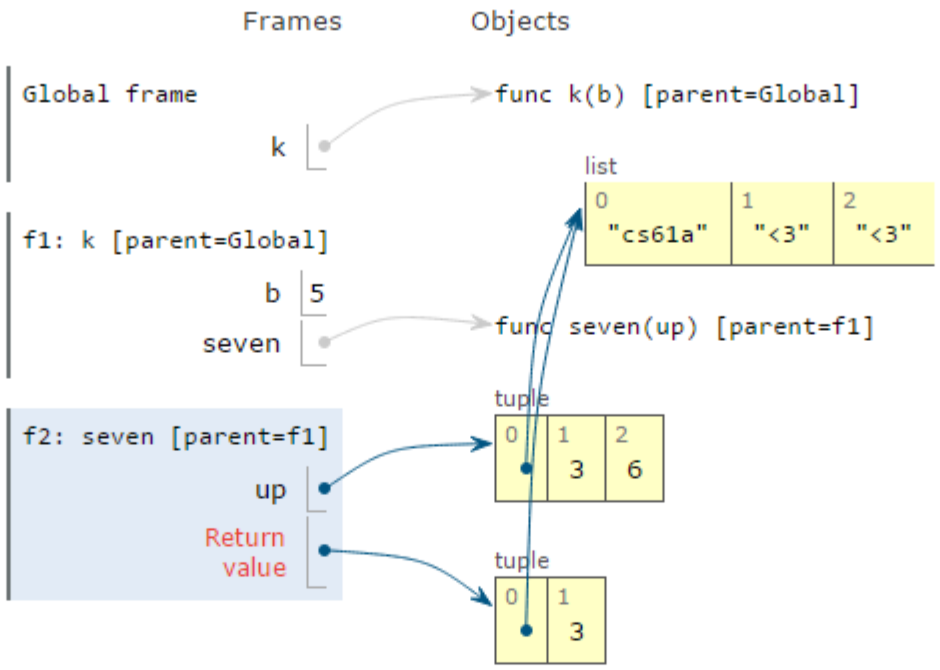
Draw the Environment Diagram

```
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'])
```



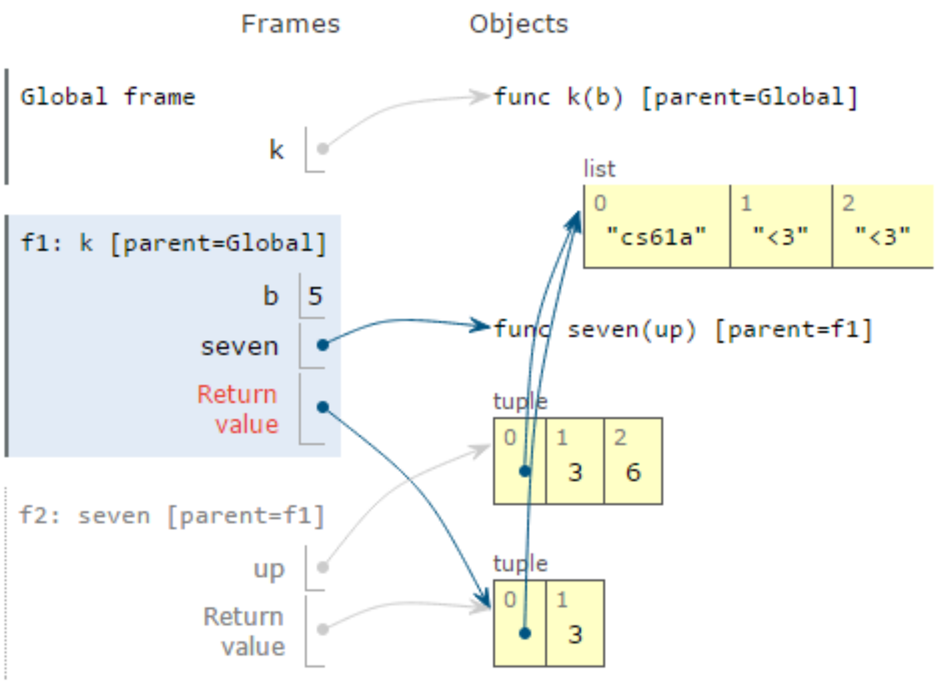
Draw the Environment Diagram

```
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'])
```



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'])
```



Nonlocal: Domo Population

John Denero really likes Domos, so he 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 function that did. The starter function is `lambda x: x * 2`. When the number is greater than or equal to the capacity of his home, he gives 9/10 away to his beloved students. (It requires 1 timestep to give away 9/10 of the domos.)

```
def domo_population(n, capacity):
    """
    >>> 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.)

```
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
```

Object-Oriented Programming

OOP: Person

```
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('John Denero', 8341)
# This calls __init__.
>>> p.greet()
"Hi, I'm John Denero"
>>> p.has_birthday()
8342
>>> Person.has_birthday()
has_birthday() missing 1 required
argument: 'self'
>>> Person.has_birthday(p)
8343
>>> Person.num_people
1
>>> p.num_people
1
```

OOP: Plant

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Plant.energy_for_leaf
10
>>> p.energy_for_leaf
_____
>>> p.if_sunny
_____
>>> p.photosynthesize
_____
>>> p.photo_fn
_____
>>> p.photosynthesize()
_____
>>> p
_____
>>> p.leaves
_____
>>> p
_____
```

OOP: Plant

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Plant.energy_for_leaf
10
>>> p.energy_for_leaf
10
>>> p.if_sunny
_____
>>> p.photosynthesize
_____
>>> p.photo_fn
_____
>>> p.photosynthesize()
_____
>>> p
_____
>>> p.leaves
_____
>>> p
_____
```

OOP: Plant

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Plant.energy_for_leaf
10
>>> p.energy_for_leaf
10
>>> p.if_sunny
Error!
>>> p.photosynthesize
_____
>>> p.photo_fn
_____
>>> p.photosynthesize()
_____
>>> p
_____
>>> p.leaves
_____
>>> p
_____
```

OOP: Plant

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Plant.energy_for_leaf
10
>>> p.energy_for_leaf
10
>>> p.if_sunny
Error!
>>> p.photosynthesize
<bound method at ...>
>>> p.photo_fn
______
>>> p.photosynthesize()
______
>>> p
______
>>> p.leaves
______
>>> p
______
```

OOP: Plant

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

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

OOP: Plant

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

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

OOP: Plant

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

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

OOP: Plant

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

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

OOP: Plant

```
>>> p = Plant(10)
>>> p # repr example
Plant<10, 0>
```

```
sunny = True
class Plant:
  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 'Plant<{}, {}>'.format(self._leaves, self.energy)
```

```
>>> Plant.energy_for_leaf
10
>>> p.energy_for_leaf
10
>>> p.if_sunny
Error!
>>> p.photosynthesize
<bound method at ...>
>>> p.photo_fn
<function <lambda> at ...>
>>> p.photosynthesize()
>>> p
Plant<10, 15.0>
>>> p.leaves
11
>>> p
Plant<11, 5.0>
```

Inheritance

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!’)
```

How can we use the concept of inheritance to improve our Fireman class?

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', 8341, 1)
>>> f.name
'John DeNero'
>>> f.has_birthday()
8342
>>> f.put_out_fire()
PUTTING OUT FIRE!
```

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 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):
        ...

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

DarkJedi

```
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):
        ...

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)
```

Do we need this?

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)
```

Do we need this?

Nope! It is inherited from Jedi.

Facepalm

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 `profile` is not already present.

Facepalm - Solution

```
class Profile(object):
    def __init__(self, name, inst):
        "*** YOUR CODE HERE ***"
    def add_friend(self, profile):
        "*** YOUR CODE HERE ***"
```

Facepalm - 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)
```

Facepalm … with profit

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. If a person tries to add more friends when they have 100 already, you should tell them to upgrade to **`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.

Facepalm ... with profit

```
class Profile(object):
    def __init__(self, name, inst):
        self.name = name
        self.inst = inst
        self.friends = []
    def add_friend(self, profile):
        "*** YOUR CODE HERE ***"

class PaidProfile(Profile):
    "*** YOUR CODE HERE ***"
```

Facepalm ... with profit - 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)
```

Next Topic: Linked Lists

Linked Lists

```
class Link:
        """A linked list with a first element and the rest."""
        empty = ()
        def __init__(self, first, rest=empty):
            assert rest is Link.empty or isinstance(rest, Link)
            self.first = first
            self.rest = rest
        def __getitem__(self, i):
            if i == 0:
                return self.first
            else:
                return self.rest[i-1]
        def __len__(self):
            return 1 + len(self.rest)
```

Linked Lists

```python
class Link:
        """A linked list with a first element and the rest."""
        empty = ()
        def __init__(self, first, rest=empty):
            assert rest is Link.empty or isinstance(rest, Link)
            self.first = first
            self.rest = rest
        def __getitem__(self, i):
            if i == 0:
                return self.first
            else:
                return self.rest[i-1]
        def __len__(self):
            return 1 + len(self.rest)
```

Make an Linked List with a 2 in it?

`Link(2)`

A Linked List with 1 then 2 in it?

`Link(1, Link(2))`

Linked Lists

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

How do we retrieve the 1?

```
r.first
```

Retrieve the 2?

```
r.rest.first
```

Linked Lists

Write `reduce`:

```
def reduce(lst, combiner, default):
    """
    >>> r = Link(1, Link(2, empty))
    >>> reduce(r, lambda x, y: x + y, 0)
    3
    """
```

Linked Lists

Write `reduce`:

```
def reduce(lst, combiner, default):
    if lst is Link.empty:
        return default
    return combiner(lst.first,
                    reduce(lst.rest, combiner, default))
```

Linked Lists

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

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

Linked Lists

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

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

Linked Lists: Challenge Question

You have a linked list (the object-based version). What is the most efficient way to find the middle element?

Linked Lists: Challenge Question

You have a linked list (the object-based version). What is the most efficient way to find the middle element?

Answer: Keep two pointers. Iterate one pointer two nodes at a time, but iterate the other only one node at a time. When the first node hits the end of the list, the second hits the middle element. (This is a popular interview question!)

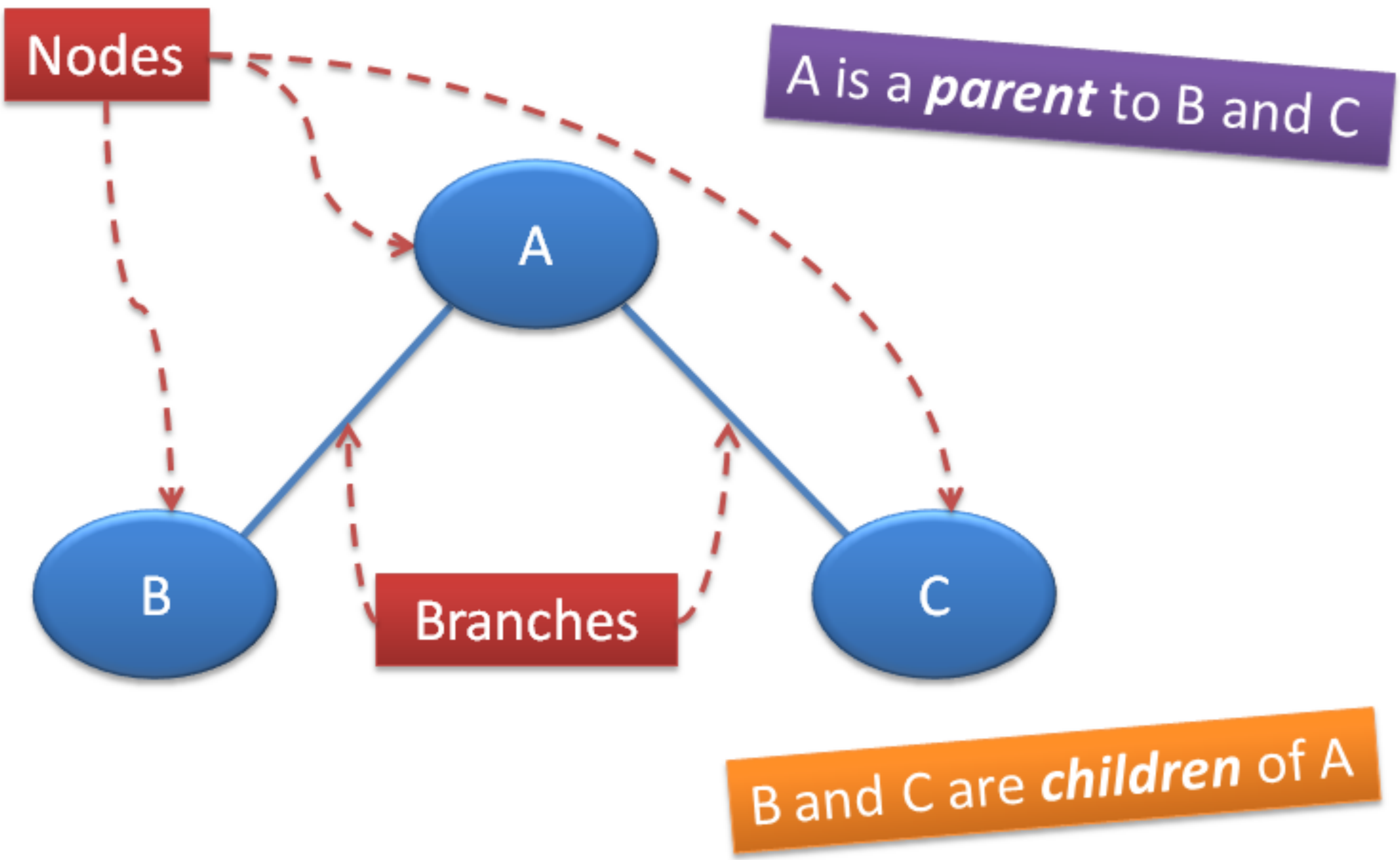
Linke ... tion

You
ve ... nd
the

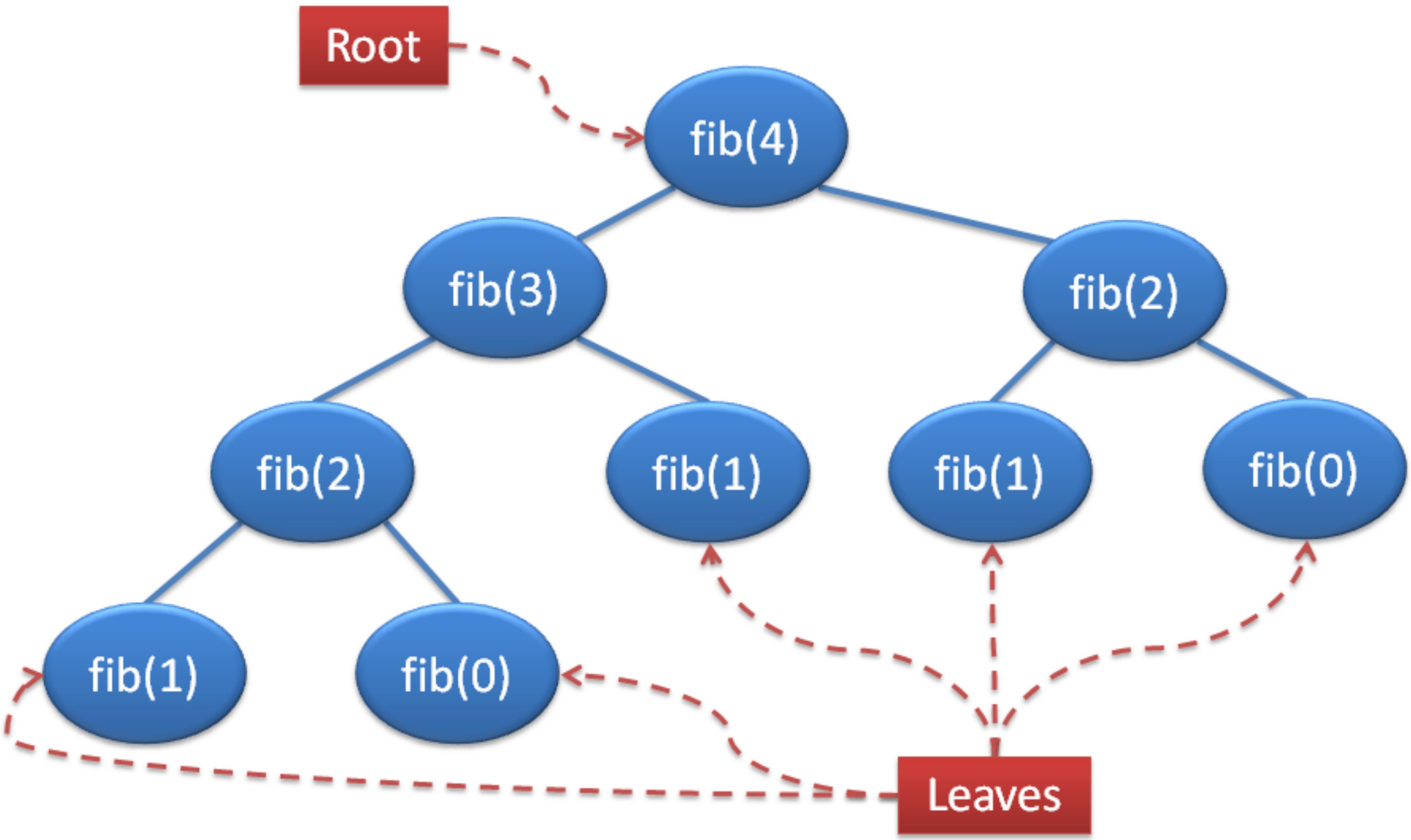
Ans ... pointer
two ... er only
one ... ode hits the
end ... the middle
elem

Next Topic: Trees

Trees: Review



Trees: Review



Trees: Review

```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = branches

    def __repr__(self):
        if self.branches:
            return 'Tree({0}, {1})'.format(self.entry, repr(self.branches))
        else:
            return 'Tree({0})'.format(repr(self.entry))

    def is_leaf(self):
        return not self.branches
```

Trees: Review

```
class BinTree(Tree):
    empty = Tree(None)
    empty.is_empty = True
    def __init__(self, entry, left=empty, right=empty):
        for branch in (left, right):
            assert isinstance(branch, BinTree) or branch.is_empty
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

    @property
    def left(self):
        return self.branches[0]

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

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 `john_finder` that takes in a tree and returns whether it contains the string “DeNero”:

```
>>> john_finder(Tree(“DeNero”, (Tree(“Hilfinger”)))
True
>>>john_finder(Tree(“#420blazeit_6969”))
False

def john_finder(t):
“***YOUR CODE HERE***”
```

Trees

```
def john_finder(t):
    if t.entry == “DeNero”:
        return True
    for b in t.branches:
        if john_finder(b):
            return True
    return False
```

Trees (Binary)

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):
    “***YOUR CODE HERE***”
```

Trees (Binary)

```
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
    return t1.entry == t2.entry and \
           tree_equals(t1.left, t2.left) and \
           tree_equals(t1.right, t2.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`.

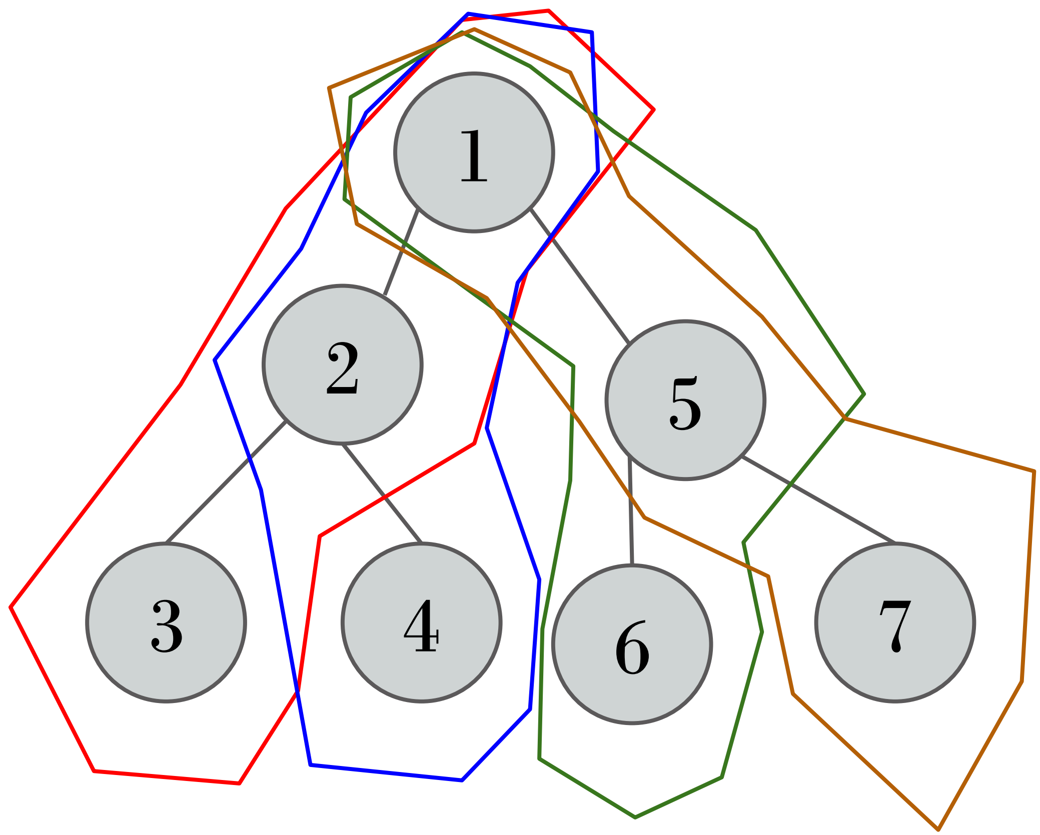
```
def prod_tree(t):
    result = t.entry
    for branch in t.branches:
        result *= prod_tree(branch)
    return result
```

Trees (Binary: HARD!)

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 (Binary: HARD!)




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

Trees (Binary: HARD!)

```
def all_paths(t):
    if t is BinTree.empty_tree:
        return []
    if t.branches == ():
        return [(t.entry,)]

    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.entry,) + path)
    return result
```

Next Topic: Orders of Growth

Orders of Growth: Review

Way of expressing how long a function/program takes to execute in terms of the size of its input as it grows very large (given as a variable, usually n).

Big ϴ Notation: Throw away constants in front of variable:

$25n^2$ ---> $\Theta(n^2)$

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
```

Θ(n)

Orders of Growth

What is the order of growth for this function?

```
def denero(denero):
    denero = 5 * denero
    john = denero ** 2
    while (john > 0):
        print (“Announcements!”)
        john = john - 1
```

Orders of Growth

What is the order of growth for this function?

```
def denero(denero):
    denero = 5 * denero
    john = denero ** 2
    while (john > 0):
        print (“Announcements!”)
        john = john - 1
```

$\Theta(\text{denero}^2)$

Orders of Growth

What is the order of growth for this function?

```
def doge(n):
    if n <= 1:
        print (“Wow”)
        return n
    return doge(n - 1) + doge(n - 2)
```

Orders of Growth

What is the order of growth for this function?

```
def doge(n):
    if n <= 1:
        print ("Wow")
        return n
    return doge(n - 1) + doge(n - 2)
```

$\Theta(2^n)$

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)
```

Θ(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)
```

$\Theta(1)$

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)
```

Θ(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?

ϴ(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?

ϴ(log x)

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

ϴ(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?

Θ(log x)

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

Θ(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?

ϴ(log x)

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

ϴ(y)

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

ϴ(y + log(x))

Feedback

We would like your feedback on this review session, so that we can improve for future review sessions. This is **completely optional**.

If you would like to provide suggestions, complaints or comments, please fill out the paper feedback form.

Thanks for coming, and best of luck on your midterm!