Discussion 05:
List Mutation, Nonlocal, and Iterators/Generators

Jennifer Tsui
jentsui.github.io

Announcements

- HW 05 due next Thursday (10/4)
- Guerrilla section this Saturday (9/29) from 12-2PM in Soda 271/273
  - covers sequences, data abstraction, **trees**
- Extra practice problems: links.cs61a.org/extra

Identity

lst1 = [1,2]
lst2 = [1,2]

lst1 and lst2 have the same elements, but they point to different list objects!

Checking if 2 lists are the same

lst1 = [1,2]

lst2 = [1,2]

lst1 == lst2	lst1 is lst2
- Checks for equality - Do they have the same elements?	- Checks for identity - Do they point to the same list object?

Checking if 2 lists are the same

lst1 = [1,2]

lst2 = [1,2]

lst1 == lst2	lst1 is lst2
- Checks for equality - Do they have the same elements?	- Checks for identity - Do they point to the same list object?
True	**False**

Try

lst1 = [1, 2, 3]

lst2 = lst1

Question 1: lst1 == lst2

Question 2: lst1 is lst2

Try

lst1 = [1, 2, 3]

lst2 = lst1

Question 1: lst1 == lst2

True

Question 2: lst1 is lst2

True

Note: lst2 is not a copy of lst1. If lst1 changes, so will lst2!

Refresher - Copying a List

lst1 = lst2 DOES NOT copy a list

Instead, we can do:

lst1 = [1,2]

lst2 = lst1[:]

or lst2 = [elem for elem in lst1]

(These operations create a new list)

Try

```
a = [1, 2]
b = [3, a]
c = b[:]
```

Try

a = [1, 2]

b = [3, a]

c = b[:]

List splicing creates a **shallow copy**

- c[1] refers to the list that **a** refers to
- we don't make a copy of the nested list [1, 2]
- we copy what is literally in each box (in this case, an arrow to the list that **a** refers to)

List Operations

Create a New List

We covered this already!

- list splicing
- list comprehension

Mutate original list

Mutating a List

lst1 = [1,2]

lst2 = lst1

lst1[0] = 5

What is lst1?

What is lst2?

lst1 and lst2 are both [5, 2]

Mutating a List

Operation	What it does	Return Value	Practice
append(elem)	Adds 1 box to end of list, puts elem in that box	None	a = [1,2] b = [3,4] c = a.append(b)

Draw box and pointer diagrams!

Mutating a List

Operation	What it does	Return Value	Practice
extend(sequence)	Iterates through elements of sequence, adding each element to end of list	None	a = [1,2] b = [3,4] a.extend(b) a.extend(5)

Draw box and pointer diagrams!

Mutating a List

Operation	What it does	Return Value	Practice
pop(index) pop()	Removes element at provided index. If no index specified, removes last element	The removed item	a = [1, 2] b = [3, a] x = b.pop()

Draw box and pointer diagrams!

Mutating a List

Operation	What it does	Return Value	Practice
remove(elem)	Removes first occurrence of elem. Errors if elem does not exist.	None	a = [1, 2, 1] a.remove(1) a == [2, 1]
insert(index, elem)	Inserts elem at the provided index. (Adds a new value rather than replacing existing value)	None	a = [1, 2, 1] a.insert(1, 5) a == [1, 5, 2, 1]

Adding Lists

lst1 = lst1 + lst2

- This creates a new list

```
a = [1, 2]
b = a
a = a + [3]
```

After this, b is [1, 2]

lst1 += lst2

- This mutates lst1

```
a = [1, 2]
b = a
a += [3]
```

After this, b is [1, 2, 3]

Summary: Operations on Lists

Create a new List:

- List splicing
- List comprehensions
- lst1 = lst1 + lst2

Mutate the original list:

- lst[0] = 5
- Append
- Extend
- Pop
- Remove
- Insert
- lst1 += lst2

Do Q1.1 on pg. 2 (Draw box-and-pointer diagram!)

```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst1 is lst2


>>> lst2.extend([5, 6])
>>> lst1[4]


>>> lst1.append([-1, 0, 1])
>>> -1 in lst2


>>> lst2[5]
```

```
>>> lst3 = lst2[:]
>>> lst3.insert(3, lst2.pop(3))
>>> len(lst1)


>>> lst1[4] is lst3[6]


>>> lst3[lst2[4][1]]


>>> lst1[:3] is lst2[:3]


>>> lst1[:3] == lst3[:3]
```

Nonlocal

What we already know...

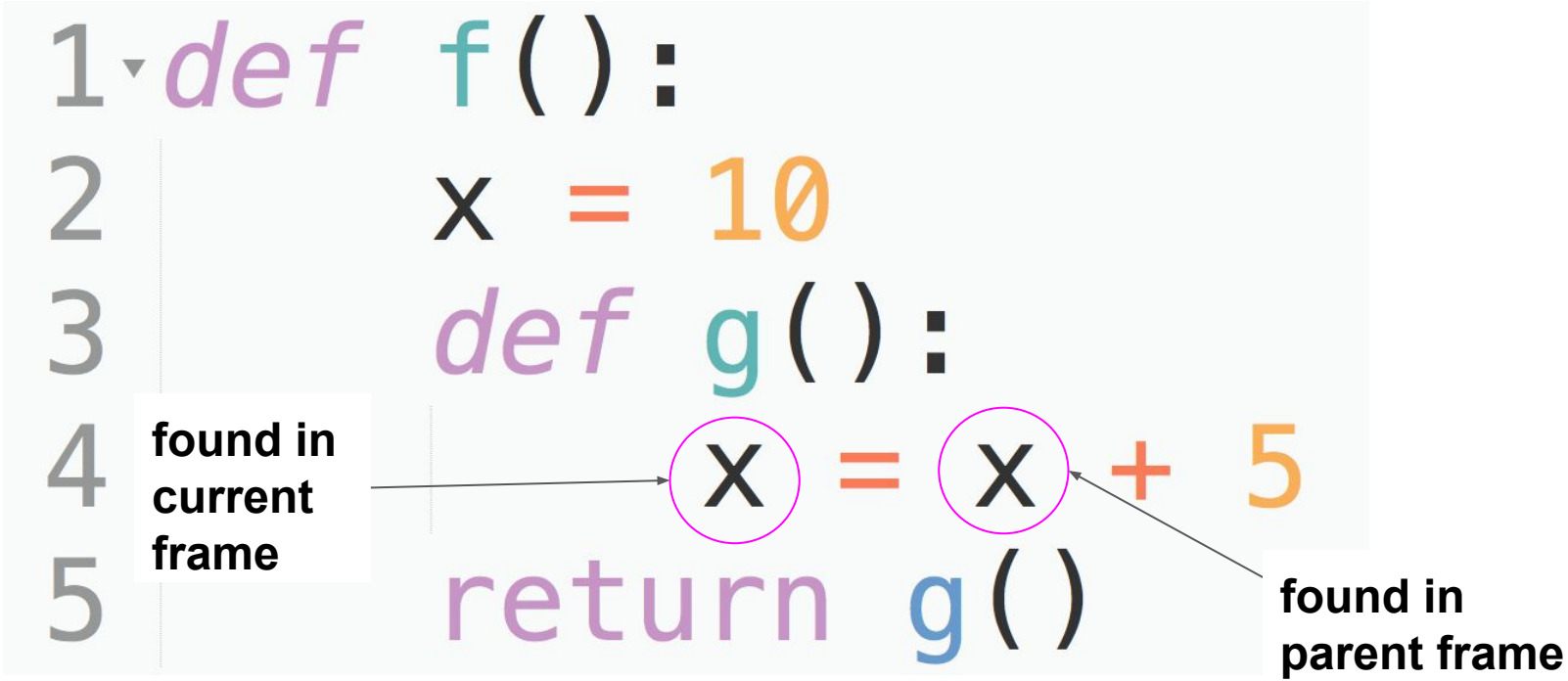
Things we can do:

- Access a name in the current frame.
- Access a name in a parent frame if it doesn't exist in the current frame.
- Add a new name to the current frame.
- Change a binding in the current frame.

Normally, assignment statement changes value in current frame

```
1 def f():
2         x = 10
3         def g():
4             x = 5
5         return g()
```

Common error: local variable referenced before assignment



Nonlocal

By default,

- you *can* **access** variables in parent frames.
- you *cannot* **modify** variables in parent frames.

`nonlocal` statements allow you to **modify** a name in a **parent** frame instead of creating a new binding in the current frame.

- cannot modify variables in current frame
- cannot create bindings in parent frames

```
def foo():
    x = 10
    def bar():
        nonlocal x
        x = 13
    bar()
    return x

foo()
```

This `nonlocal` *statement tells Python: "Don't create a new local variable* x*; modify the one in the parent frame instead!"*

Stepper: Q2.1, pg. 5

```
1  def stepper(num):
2      def step():
3          nonlocal num
4          num = num + 1
5          return num
6      return step

7  s = stepper(3)
8  s()
9  s()
```

Nonlocal name lookup rules:

1) Look for the name in the current frame's parent frame first.
2) If it's not found, continue looking at parent frames.

Nonlocal variable assignment rules:

1) Use *nonlocal name lookup rules* to find the binding.
2) Replace the old binding of the name with the new value.

Edge Cases

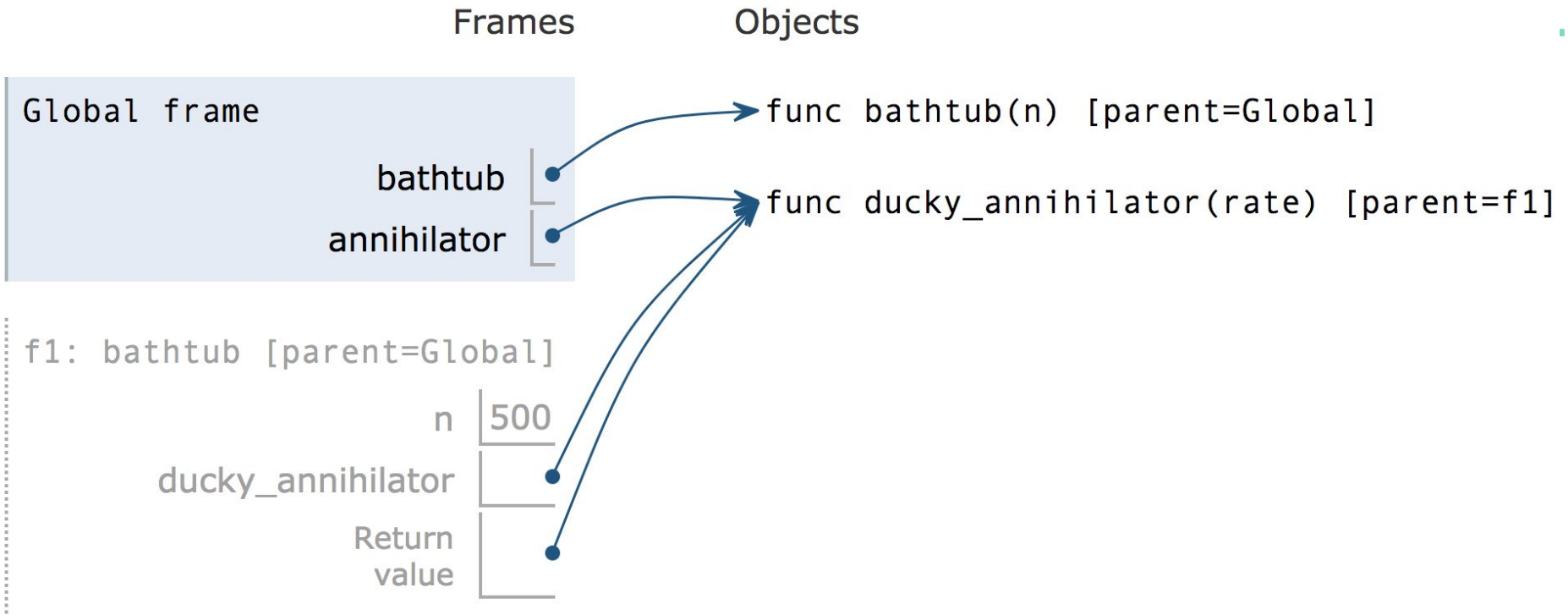
- We don’t look at variables in the global frame

```
x = 10
def f():
    nonlocal x
    x = 11
f()
=> No binding for nonlocal 'x' found
```

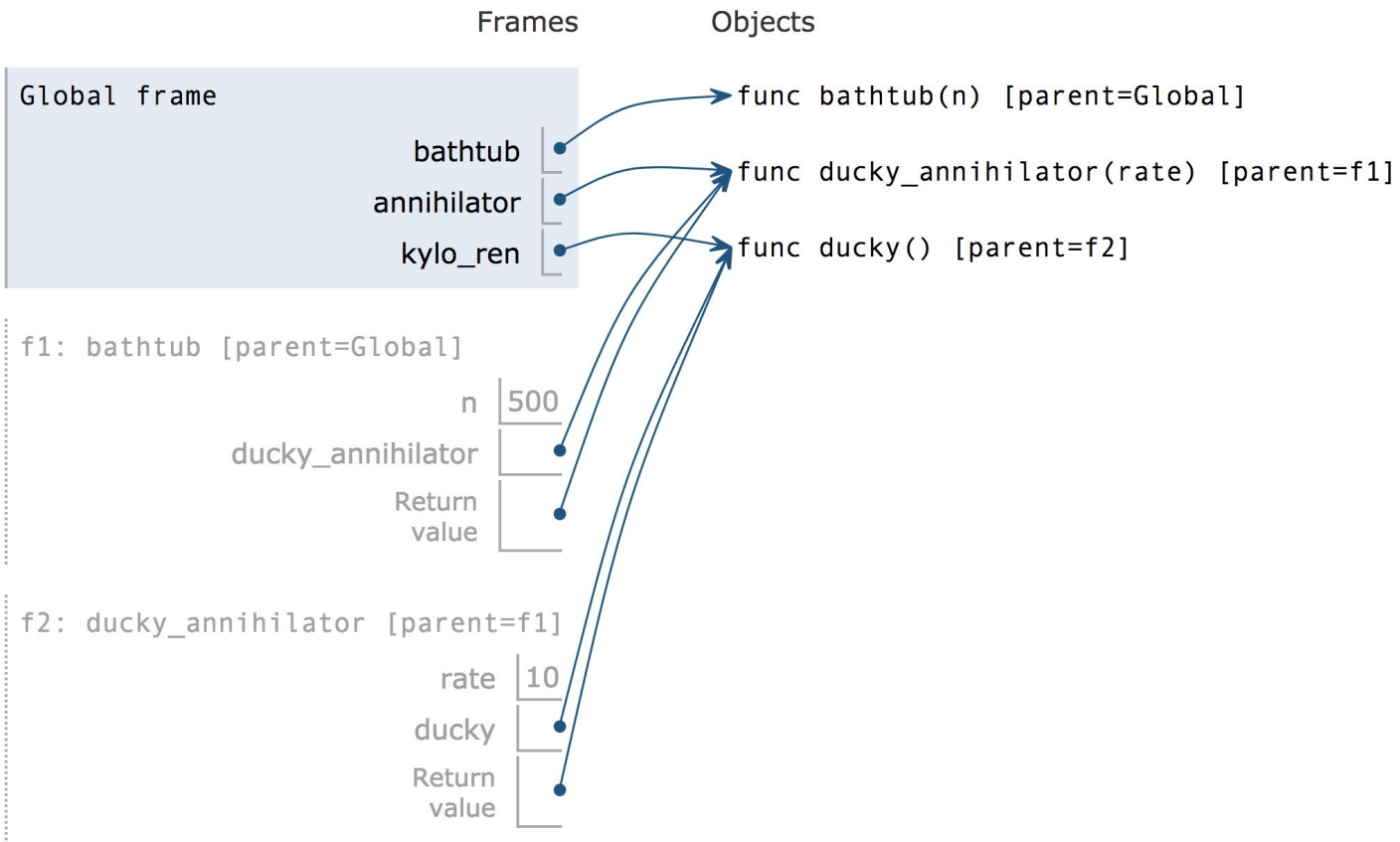
Bathtub: Q2.4 pg. 6

```
def bathtub(n):
    """
    >>> annihilator = bathtub(500) # the force awakens...
    >>> kylo_ren = annihilator(10)
    >>> kylo_ren()
    490 rubber duckies left
    >>> rey = annihilator(-20)
    >>> rey()
    510 rubber duckies left
    >>> kylo_ren()
    500 rubber duckies left
    """
    def ducky_annihilator(rate):
        def ducky():









        return ducky
    return ducky_annihilator
```

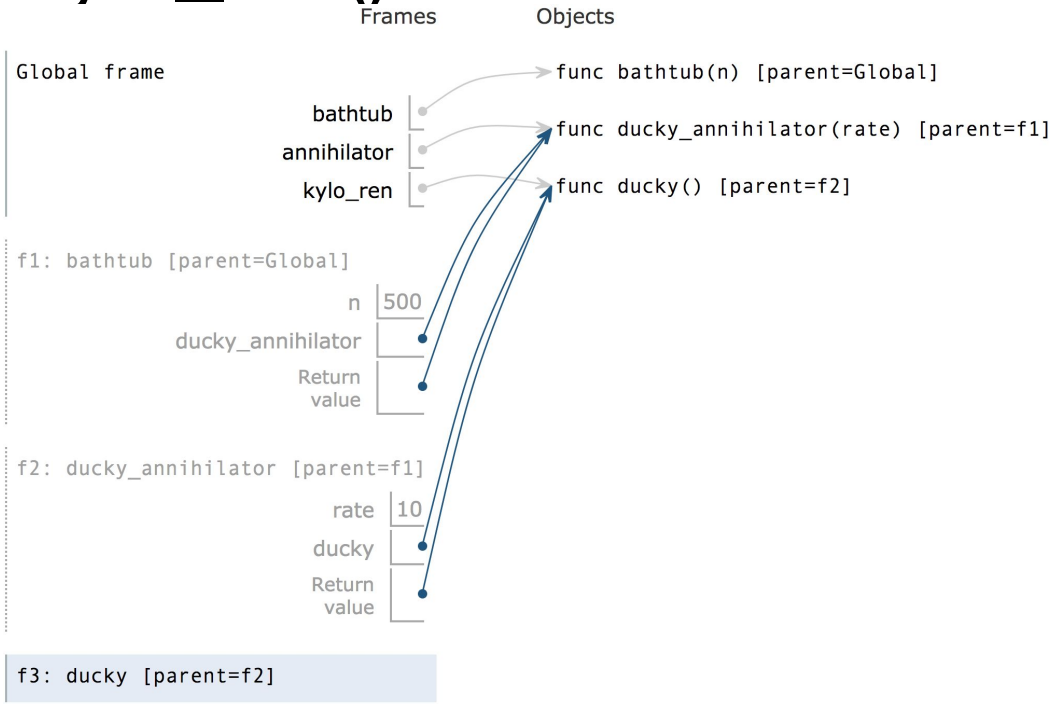
annihilator = bathtub(500)



kylo_ren = annihilator(10)



kylo_ren()



We're in frame f3

We want to print out **490 rubber duckies left**

490 is 500 (which is n) minus 10 (which is rate)

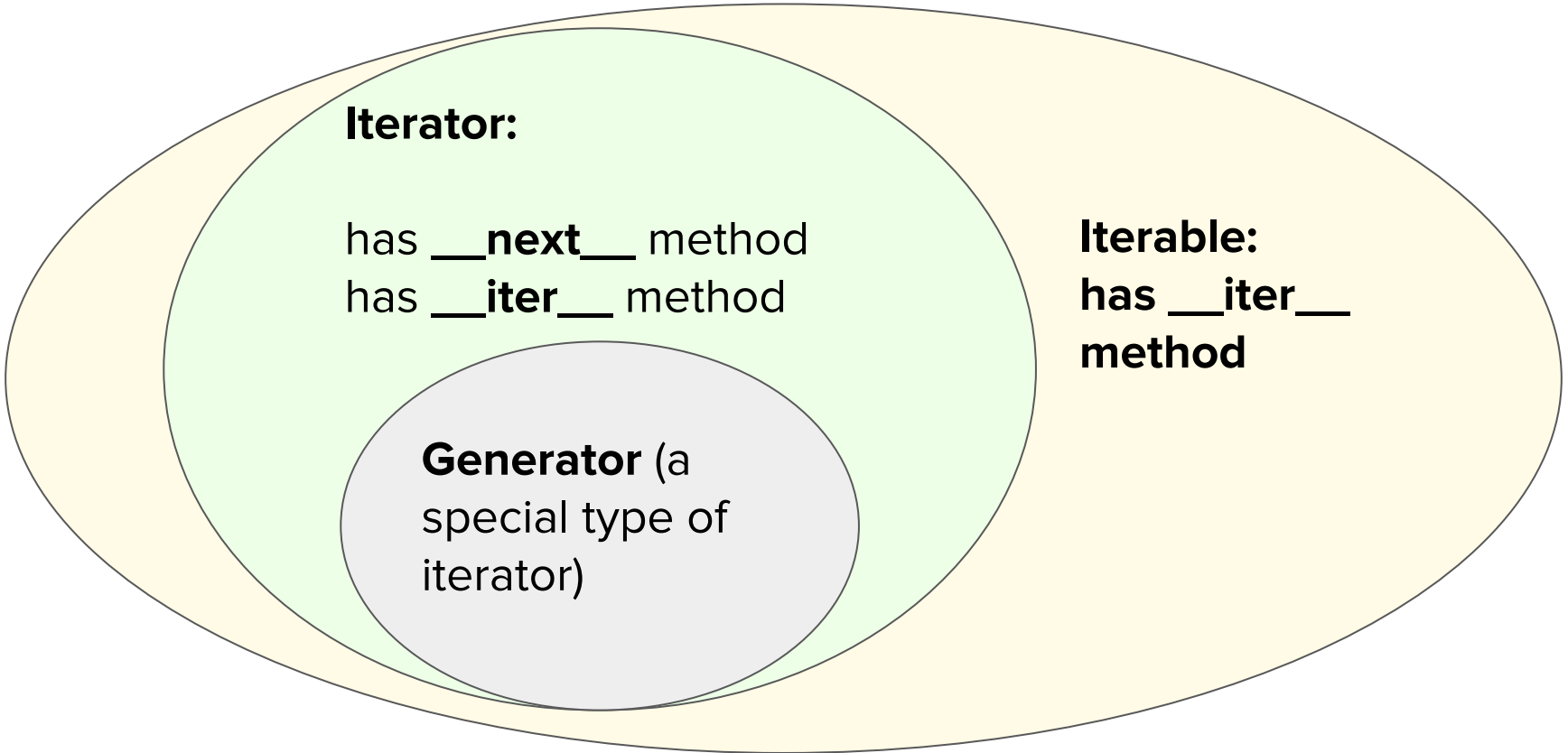
Attendance:
links.cs61a.org/jen
magic word:

Iterators and Generators

Definitions

- **Iterable**: has __iter__ method
  - Things like lists, dictionaries, ranges
  - Calling iter on an iterable returns a **new iterator**
- **Iterator:** has __iter__ method and __next__ method
  - Think of iterable as a book, and iterator as a bookmark
  - 99% of the time, calling iter on an iterator (that's not an iterable) returns **itself** (not a new iterator)
  - Will raise **StopIteration** exception when we run out elements
- **Generator function**: A function that when called, produces a generator
  - Important: We **DO NOT step through body of function!!!!**
- **Generator:** a special type of iterator
  - a function that contains "yield" or "yield from"
  - produced when we call a generator function

Definitions



For loop

```
counts = [1, 2, 3]

for i in counts:
    print(i)
```

```
items = iter(counts)
while True:
    try:
        i = next(items)
        print(i)
    except StopIteration:
        break #Exit the while loop
```

Q3.1 page 8

3.1 What would Python display? If a StopIteration Exception occurs, write `StopIteration`, and if another error occurs, write `Error`.

```
>>> lst = [6, 1, "a"]
>>> next(lst)


>>> lst_iter = iter(lst)
>>> next(lst_iter)


>>> next(lst_iter)


>>> next(iter(lst))


>>> [x for x in lst_iter]
```

Remember:
Typically, calling iter on an iterable gives a **new iterator (start over from the beginning),** but calling iter on an iterator returns **itself (start where we left off)**

Generators (demo)

```
def f():
    print(1)
    yield 'hi'
    print('here')
    yield from [1, 2]
    return 'hello!'
```