Hi! Welcome to 61A Discussion :)

We will begin at **5:10**!

Attendance: **go.cs61a.org/ben-disc**

Slides: **cs61a.bencuan.me**

Announcements

- Ants is due today!
- HW5 due next Tuesday
  - Please fill out the midsemester survey!
- Midterm next Thursday
  - See logistics post on Piazza
  - No discussion next Thursday
  - Review lab next Tuesday

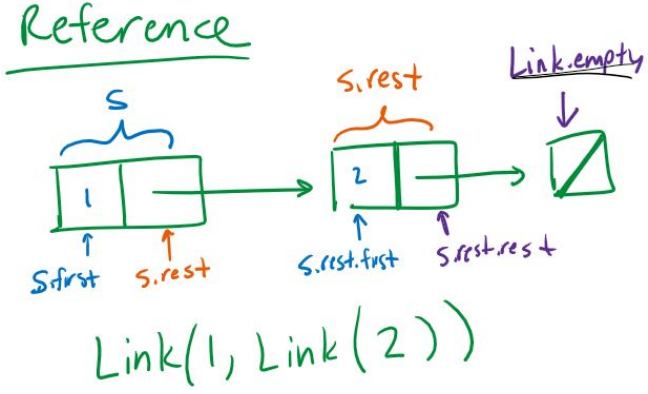
Agenda

- Attendance
- Linked Lists
- Iterators and Generators

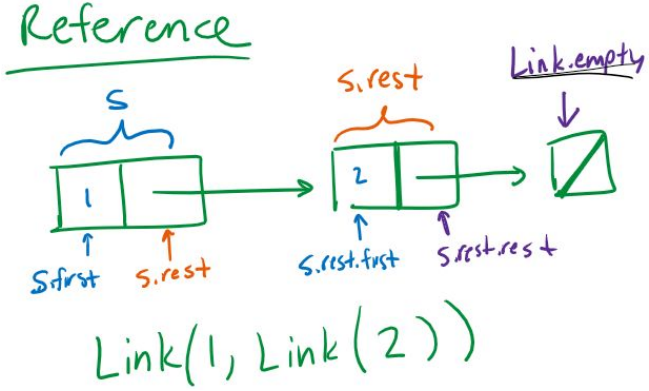
Linked Lists

Lab review (1)

- Basically a tree with only one branch (rest)
- Create with Link(first, rest)
- First is a label, **rest is always a link**
- Check empty: lnk is Link.empty



Lab review (2)




```
class Link:
    empty = …
    def __init__(self, first, rest=Link.empty):
        self.first = first
        self.rest = rest

s = Link(1, Link(2))
```

Quick recap on mutation:

Mutation = whenever you change the actual object
If you're simply rearranging an arrow in an environment diagram, this is **NOT** a mutation!

is it a mutation?

1. Which of the following are mutations? (assume t is a tree and s is a linked list)

- ○ t.label = 3
- ○ t = Tree(3, t.branches)
- ○ t.branches[0] = Tree(3)
- ○ s.rest = Link(s, s.rest)
- ○ s = s.rest

Linked List Demo (Q1)

```
>>> link = Link(1, Link(2, Link(3)))
>>> link.first
```

```
>>> link.rest.first
```

```
>>> link.rest.rest.rest is Link.empty
```

```
>>> link.rest = link.rest.rest
>>> link.rest.first
```

```
>>> link = Link(1)
>>> link.rest = link
>>> link.rest.rest.rest.rest.first
```

Linked List Demo (Q1)

```
>>> link = Link(2, Link(3, Link(4)))
>>> link2 = Link(1, link)
>>> link2.first
```

```
>>> link2.rest.first
```

Quick recap on mutation:

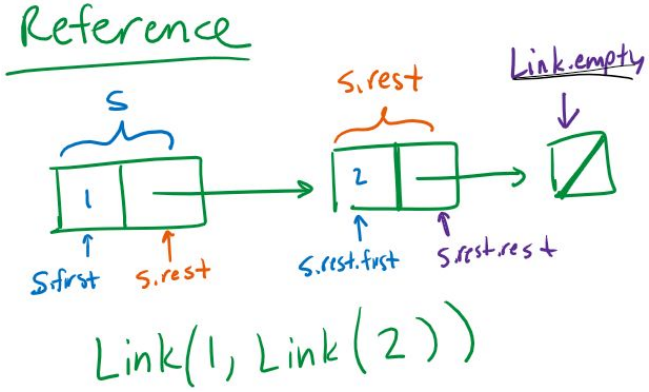
Mutation = whenever you change the actual object
If you're simply rearranging an arrow in an environment diagram, this is **NOT** a mutation!

is it a mutation?

1. Which of the following are mutations? (assume t is a tree and s is a linked list)

○ t.label = 3	**Yes**
○ t = Tree(3, t.branches)	**No**
○ t.branches[0] = Tree(3)	**Yes**
○ s.rest = Link(s, s.rest)	**Yes**
○ s = s.rest	**No**

Linked List Coding



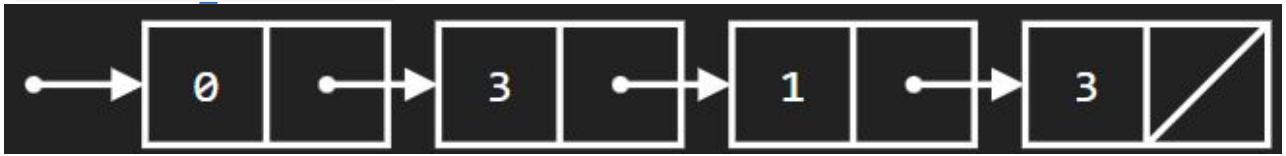

```
def build_link(s):
    result = Link.empty
    while s is not Link.empty:
        new_value = do stuff with link.first
        result = Link(new_value, result)
        s = s.rest
    return result
```

Some code writing (Q2)

Q2: Remove All

Implement a function `remove_all` that takes a `Link`, and a `value`, and remove any linked list node containing that value. You can assume the list already has at least one node containing `value` and the first element is never removed. Notice that you are not returning anything, so you should mutate the list.


```
>>> remove_all(l1, 2)
```



Hints:

- This is a **mutative function:** remember what counts as a mutation!
- It is usually easier to change rest instead of first
- The iterative and recursive versions are similar in complexity

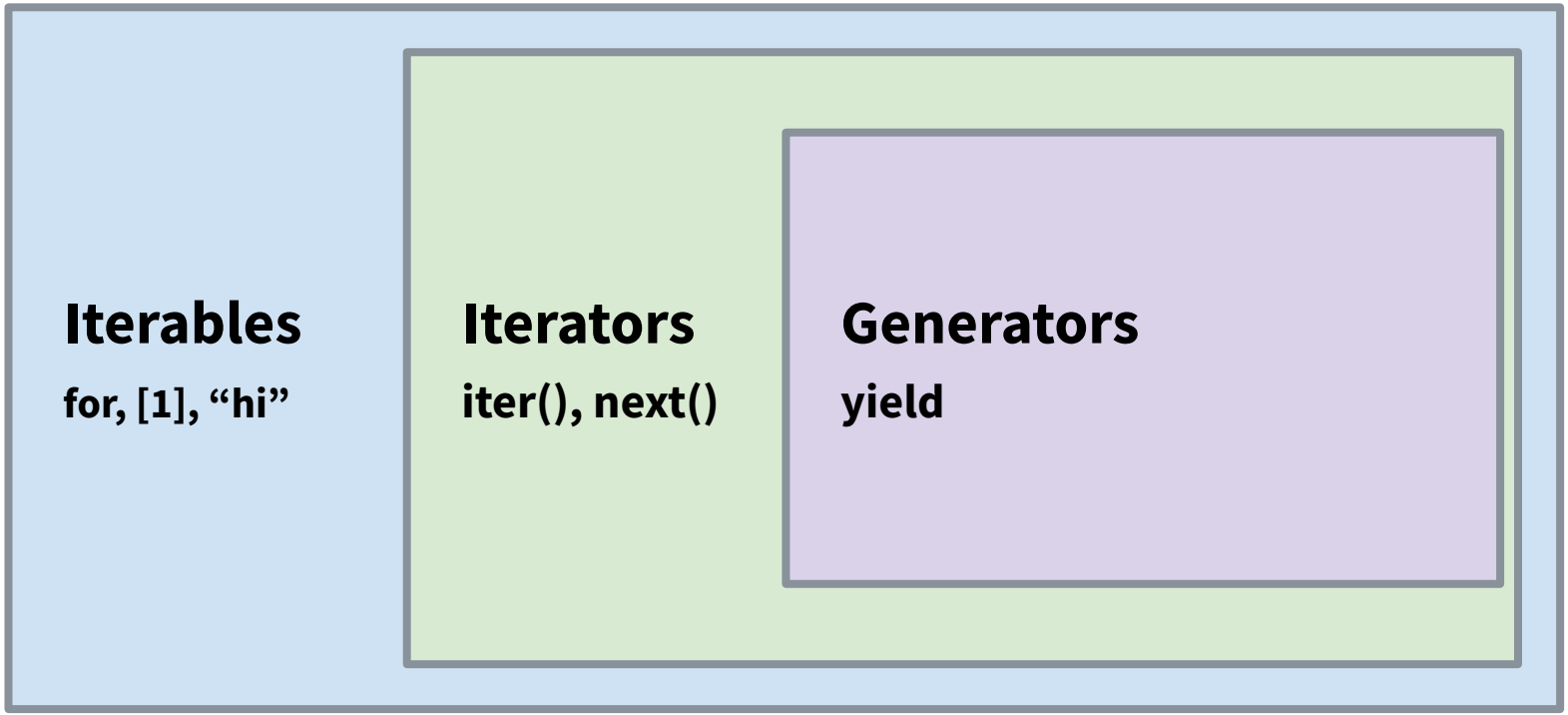
Q2: Remove All

Implement a function `remove_all` that takes a `Link`, and a `value`, and remove any linked list node containing that value. You can assume the list already has at least one node containing `value` and the first element is never removed. Notice that you are not returning anything, so you should mutate the list.

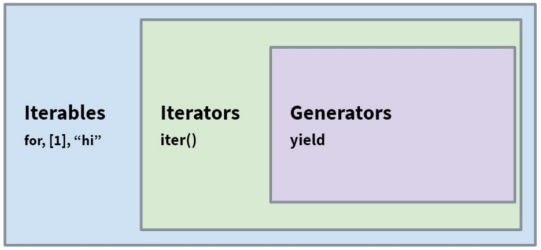
```
def remove_all(link, value):


    return None
```

Iterators and Generators

Type Comparison Chart



More specific comparisons



- **Iter*ables*:** anything that can be used in a for loop
  - Lists, strings, dictionaries, iterators, generators, ranges…
- **Iter*ators*:** a specific object type created using `iter()` that you can call `next()` on
- **Generators:** created with the `yield` statement, runs to next yield every `next(gen)` call

Anatomy of an Iterator

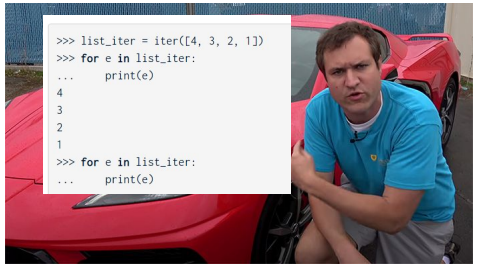
```
iterator = iter(iterable)
try:
    while True:
        elem = next(iterator)
        # do something
except StopIteration:
    pass
```

<- `iterable` can be a list, string, other iterator, etc.

<- It's safe to do `while true` here since StopIteration is guaranteed to occur!

Calling `next` is like doing `k+=1` in a while loop: **do it only once!!**

Quirks and Features



- **Iterators can only be run through once!**
  - Doing a for loop a second time will do nothing
- **next() can ONLY be called on iterators, not iterables!**
  - Calling next([1,2,3]) will error because lists aren't iterators.
- **For loops automatically call next() and end on StopIteration**
  - **Do not call next() in a for loop!** Use while loops instead

Iterables are super useful!!

- `map(f, iterable)` - Creates an iterable over `f(x)` for `x` in `iterable`. In some cases, computing a list of the values in this iterable will give us the same result as [`func(x)` for `x` in `iterable`]. However, it's important to keep in mind that iterators can potentially have infinite values because they are evaluated lazily, while lists cannot have infinite elements.
- `filter(f, iterable)` - Creates iterator over `x` for each `x` in `iterable` if `f(x)`
- `zip(iterables*)` - Creates an iterable over co-indexed tuples with elements from each of the `iterables`
- `reversed(iterable)` - Creates iterator over all the elements in the input iterable in reverse order
- `list(iterable)` - Creates a list containing all the elements in the input `iterable`
- `tuple(iterable)` - Creates a tuple containing all the elements in the input `iterable`
- `sorted(iterable)` - Creates a sorted list containing all the elements in the input `iterable`
- `reduce(f, iterable)` - Must be imported with `functools`. Apply function of two arguments `f` cumulatively to the items of `iterable`, from left to right, so as to reduce the sequence to a single value.

Try some WWPD... (Q3)

What would Python display?

```
>>> s = [[1, 2, 3, 4]]
>>> i = iter(s)
>>> j = iter(next(i))
>>> next(j)
```

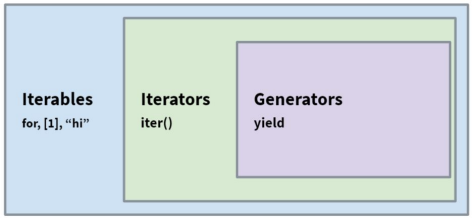
```
>>> s.append(5)
>>> next(i)
```

```
>>> next(j)
```

```
>>> list(j)
```

```
>>> next(i)
```

Anatomy of a Generator



pythontutor


```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

<- defined the same way as a normal function!

<- `return` replaced with a `yield`. Pauses here until next() is called again

```
>>> next(c)
Beginning countdown!
5
```

```
>>> next(c)
4
>>> next(c)
3
```

```
>>> next(c)
2
>>> next(c)
1
```

```
>>> next(c)
0
>>> next(c)
Blastoff!
StopIteration
```

Quirks and Features

- **Do not call generators like functions, this will not work!!** (do `next(gen)`, not `gen()`)
- Generators can be **infinite** (so don't call list(gen) or put generators into a list comprehension)
- Creating multiple instances of the same generator: each one is tracked separately

```
>>> c1, c2 = countdown(5), countdown(5)
>>> c1 is c2
False
>>> next(c1)
5
>>> next(c2)
5
```

Q4: Filter-Iter

Implement a generator function called `filter_iter(iterable, fn)` that only yields elements of `iterable` for which `fn` returns True.

```
def filter_iter(iterable, fn):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even)) # a list of the values yielded from the call to
        filter_iter
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter_iter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter_iter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    "*** YOUR CODE HERE ***"
```

Hints:

- **Iterable,** not iterator! Think for loops

A quick note on yield from

- It's not as bad as you think!!
- Basically just calling yield a bunch of times

```
def gen1():
    yield 1
    yield 2
    yield 3
```

```
def gen2():
    yield from [1, 2, 3]
```

```
(BONUS)
def gen3():
    yield from gen2()
```

Q6: Primes Generator

Write a function `primes_gen` that takes a single argument `n` and yields all prime numbers less than or equal to `n` in decreasing order. Assume `n >= 1`. You may use the `is_prime` function included below, which we implemented in Discussion 3.

Optional Challenge: Now rewrite the generator so that it also prints the primes in *ascending order*.

```
def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    if _____________________________________:
        return

    if _____________________________________:

        yield _____________________________________

    yield from _____________________________________
```

Hints:
- Don't worry about how is_prime works, just use it!
- Base case? recursive case?

(How do we make the problem simpler for yield from?)

Q5: Infinite Hailstone

Write a generator function that outputs the hailstone sequence starting at number `n`. After reaching the end of the hailstone sequence, the generator should yield the value 1 infinitely.

Here's a quick reminder of how the hailstone sequence is defined:

1. Pick a positive integer `n` as the start.
2. If `n` is even, divide it by 2.
3. If `n` is odd, multiply it by 3 and add 1.
4. Continue this process until `n` is 1.

Write this generator function recursively. If you're stuck, you can first try writing it iteratively and then seeing how you can turn that implementation into a recursive one.

Hint: Since `hailstone` returns a generator, you can `yield from` a call to `hailstone`!