Recursive Objects
(Revisit)
Lecture Quiz
Consider the following variable assignments. Which of the created variables can we call \texttt{\_\_str\_} on?

class \texttt{Lamb}:
    species\_name = "Lamb"
    scientific\_name = "Ovis aries"

    def \texttt{\_\_init\_}(self, name):
        self.name = name

lamb = Lamb("Fleecey")  # 80%
grade = 79.5  # 80%

colors = ["red", "orange", "yellow"]  # 72%

translations = {"one": "uno", "two": "dos"}  # 65%

is\_fluffy = True  # 60%
Consider the following variable assignments. Which of the created variables can we call \texttt{__str__} on?

```python
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def \texttt{__init__}(self, name):
        self.name = name

lamb = Lamb("Fleecey")
print(Lamb.__str__(lamb))  # 80%
grade = 79.5
print(float.__str__(grade))  # 80%
colors = ["red", "orange", "yellow"]
print(list.__str__(colors))  # 72%
translations = {"one": "uno", "two": "dos"}
print(dict.__str__(translations))  # 65%
is_fluffy = True
print(bool.__str__(is_fluffy))  # 60%
```

All of them! All of them inherit from \texttt{object} and thus have a \texttt{__str__} method defined. Run \texttt{type(var).__str__(var)} to prove it to yourself.
Imagine we have a class `Book` with a class variable `available_formats`, and an instance of that class named `bookywook`.

```python
class Book:
    available_formats = ["Kindle", "paperback"]

    def __init__(self, title):
        self.title = title

bookywook = Book("Where's Boo?")
```

Which line of code would retrieve the value of that attribute?

- `Book.__getattribute__(bookywook, "available_formats")`  # 61%
- `getattr(bookywook, "available_formats")`  # 60%
- `bookywook.available_formats`  # 51%
- `Book.available_formats`  # 20%
- `getattr(Book, "available_formats")`  # 19%
Attribute access

Imagine we have a class `Book` with a class variable `available_formats`, and an instance of that class named `bookywook`.

class Book:
    available_formats = ["Kindle", "paperback"]

    def __init__(self, title):
        self.title = title

bookywook = Book("Where's Boo?")

Which line of code would retrieve the value of that attribute?

```python
Book.__getattribute__(bookywook, "available_formats")  # 61%
getattr(bookywook, "available_formats")               # 60%
bookywook.available Formats                           # 51%
Book.available Formats                                 # 20%
getattr(Book, "available_formats")                    # 19%
```

All of them retrieve the value!
Trees
Tree concepts

- A tree has a **root label** and a list of **branches**
- Each **branch** is itself a **tree**
- A tree with **zero branches** is called a **leaf**
Trees: Data abstraction

This is what we've been using:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree(label, children)</td>
<td>Returns a tree with given LABEL at its root, whose children are CHILDREN</td>
</tr>
<tr>
<td>label(tree)</td>
<td>Returns the label of root node of TREE</td>
</tr>
<tr>
<td>branches(tree)</td>
<td>Returns the branches of TREE (each a tree).</td>
</tr>
<tr>
<td>is_leaf(tree)</td>
<td>Returns true if TREE is a leaf node.</td>
</tr>
</tbody>
</table>
Trees: Data abstraction

Using an implementation like this:

def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not branches(tree)

🤔 How could we represent trees as a Python class?
A Tree class

```python
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

🤔 What's different? What's the same?
# tree versus Tree

<table>
<thead>
<tr>
<th>tree</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t = \text{tree}(\text{label, branches=}[]) )</td>
<td>( t = \text{Tree}(\text{label, branches=}[]) )</td>
</tr>
<tr>
<td><code>branches(t)</code></td>
<td><code>t.branches</code></td>
</tr>
<tr>
<td><code>label(t)</code></td>
<td><code>t.label</code></td>
</tr>
<tr>
<td><code>is_leaf(t)</code></td>
<td><code>t.is_leaf()</code></td>
</tr>
</tbody>
</table>

```python
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n - 2)
        right = fib_tree(n - 1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```
```python
def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n - 2)
        right = fib_tree(n - 1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```
Doubling a Tree

Is the `Tree` object mutable or immutable?
Is `double(t)` destructive or non-destructive?

```python
def double(t):
    """Doubles every label in T, mutating T."
    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> double(t)
    >>> t
    Tree(2, [Tree(6, [Tree(10)]), Tree(14)])
    """
    t.label = t.label * 2
    if not t.is_leaf():
        for b in t.branches:
            double(b)
```
Doubling a Tree

```python
def double(t):
    """Doubles every label in T, mutating T.
    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> double(t)
    >>> t
    Tree(2, [Tree(6, [Tree(10)]), Tree(14)])
    """
    t.label = t.label * 2
    if not t.is_leaf():
        for b in t.branches:
            double(b)
```

Is the `Tree` object mutable or immutable? Mutable!
Is `double(t)` destructive or non-destructive?
Doubling a Tree

```python
def double(t):
    """Doubles every label in T, mutating T.""
    t.label = t.label * 2
    if not t.is_leaf():
        for b in t.branches:
            double(b)
```

Is the `Tree` object mutable or immutable?Mutable!
Is `double(t)` destructive or non-destructive? Destructive!
A fancier Tree

This is what assignments actually use:

class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
    def is_leaf(self):
        return not self.branches
    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1}).format(self.label, branch_str)
    def __str__(self):
        return '\n'.join(self.indented())
    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append(' ' + line)
        return [str(self.label)] + lines

It's built in to code.cs61a.org, and remember, you can

\text{draw()} \quad \text{any tree/Tree.}
Linked lists
Why do we need a new list?

Python lists are implemented as a "dynamic array", which isn't optimal for all use cases.

-inserting an element is slow, especially near front of list:

<table>
<thead>
<tr>
<th>&quot;A&quot;</th>
<th>&quot;B&quot;</th>
<th>&quot;C&quot;</th>
<th>&quot;D&quot;</th>
<th>&quot;E&quot;</th>
<th>&quot;F&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3300</td>
<td>3301</td>
<td>3302</td>
<td>3303</td>
<td>3304</td>
<td>3305</td>
</tr>
</tbody>
</table>

What should we insert?

-value: Z  @ index: 3  Insert
Why do we need a new list?

Python lists are implemented as a "dynamic array", which isn't optimal for all use cases.

😭 Inserting an element is slow, especially near front of list:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>&quot;B&quot;</td>
<td>&quot;C&quot;</td>
<td>&quot;D&quot;</td>
<td>&quot;E&quot;</td>
<td>&quot;F&quot;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3300</td>
<td>3301</td>
<td>3302</td>
<td>3303</td>
<td>3304</td>
<td>3305</td>
</tr>
</tbody>
</table>

What should we insert?

value: Z  @ index: 3  ✋ Insert

😭 Plus inserting too many elements can require re-creating the entire list in memory, if it exceeds the pre-allocated memory.
Linked lists

A linked list is a chain of objects where each object holds a **value** and a **reference to the next link**. The list ends when the final reference is empty.

What should we insert?

value: Z  @ index: 5  Insert
A linked list is a chain of objects where each object holds a **value** and a **reference to the next link**. The list ends when the final reference is empty.

What should we insert?

value: Z @ index: 5

Linked lists require more space but provide faster insertion.
A Link class

```python
class Link:
    empty = ()

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

How would we use that?
A Link class

class Link:
    empty = ()

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

How would we use that?

ll = Link("A", Link("B", Link("C")))

Try in PythonTutor
A fancier LinkedList

It's built-in to code.cs61a.org and you can `draw()` any Link.
Creating linked lists
Creating a range

Similar to \[x \text{ for } x \text{ in } \text{range}(3, 6)\]

```python
def range_link(start, end):
    """Return a Link containing consecutive integers from START to END, not including END.
    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
```

Try in PythonTutor
Creating a range

Similar to \[x \text{ for } x \text{ in } \text{range}(3, 6)\]

def range_link(start, end):
    """Return a Link containing consecutive integers from START to END, not including END."
    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    if start >= end:
        return Link.empty
    return Link(start, range_link(start + 1, end))

Try in PythonTutor
Mapping a linked list

Similar to \([f(x) \text{ for } x \text{ in } \text{lst}]\)

```python
def map_link(f, ll):
    """Return a Link that contains f(x) for each x in Link LL.
    """
    >>> square = lambda x: x * x
    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
```

Try in PythonTutor
Mapping a linked list

Similar to $[f(x) \text{ for } x \text{ in } lst]$
Filtering a linked list

Similar to \[x \text{ for } x \text{ in } \text{lst} \text{ if } f(x)\]

def filter_link(f, ll):
    """Return a Link that contains only the elements x of Link LL for which f(x) is a true value."
    >>> is_odd = lambda x: x % 2 == 1
    >>> filter_link(is_odd, range_link(3, 6))
    Link(3, Link(5))
    """
Filtering a linked list

Similar to `[x for x in lst if f(x)]`

```python
def filter_link(f, ll):
    """Return a Link that contains only the elements x of Link LL
    for which f(x) is a true value.
    >>> is_odd = lambda x: x % 2 == 1
    >>> filter_link(is_odd, range_link(3, 6))
    Link(3, Link(5))
    """
    if ll is Link.empty:
        return Link.empty
    elif f(ll.first):
        return Link(ll.first, filter_link(f, ll.rest))
    return filter_link(f, ll.rest)

Try in PythonTutor
Mutating linked lists
Linked lists can change

Attribute assignments can change `first` and `rest` attributes of a `Link`.

```python
s = Link("A", Link("B", Link("C")))
s.first = "Hi"
s.rest.first = "Hola"
s.rest.rest.first = "Oi"
```

Try in PythonTutor
Beware infinite lists

The rest of a linked list can contain the linked list as a sub-list.

```python
s = Link("A", Link("B", Link("C")))
t = s.rest
t.rest = s

s.first
s.rest.rest.rest.rest.rest.first
```
def `insert_front`(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST, returning new linked list."

    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST, returning new linked list."

    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """

    return Link(new_val, linked_list)
Adding to an ordered linked list

```python
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST."
    if new_val < ordered_list.first:
        elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        elif new_val > ordered_list.first:
            return ordered_list
```
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5))))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        original_first = ordered_list.first
        ordered_list.first = new_val
        ordered_list.rest = Link(original_first, ordered_list.rest)
    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        return ordered_list
    elif new_val > ordered_list.first:
        return ordered_list
    return ordered_list
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        original_first = ordered_list.first
        ordered_list.first = new_val
        ordered_list.rest = Link(original_first, ordered_list.rest)
    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        ordered_list.rest = Link(new_val)
    elif new_val > ordered_list.first:
        return ordered_list
Adding to an ordered linked list

```python
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        original_first = ordered_list.first
        ordered_list.first = new_val
        ordered_list.rest = Link(original_first, ordered_list.rest)
    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        ordered_list.rest = Link(new_val)
    elif new_val > ordered_list.first:
        add(ordered_list.rest, new_val)
    return ordered_list
```
Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

<table>
<thead>
<tr>
<th>Version</th>
<th>10,000 runs</th>
<th>100,000 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Try it yourself on your local machine (Legit Python!): warandpeace.py
Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

<table>
<thead>
<tr>
<th>Version</th>
<th>10,000 runs</th>
<th>100,000 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python list</td>
<td>2.6 seconds</td>
<td>37 seconds</td>
</tr>
<tr>
<td>Link</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Try it yourself on your local machine (Legit Python!): warandpeace.py
Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

<table>
<thead>
<tr>
<th>Version</th>
<th>10,000 runs</th>
<th>100,000 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python list</td>
<td>2.6 seconds</td>
<td>37 seconds</td>
</tr>
<tr>
<td>Link</td>
<td>0.01 seconds</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Try it yourself on your local machine (Legit Python!): warandpeace.py
Recursive objects

Why are Tree and Link considered recursive objects?
Recursive objects

Why are **Tree** and **Link** considered recursive objects?

Each type of object contains references to the same type of object.

- An instance of **Tree** can contain additional instances of **Tree**, in the **branches** variable.
- An instance of **Link** can contain an additional instance of **Link**, in the **rest** variable.

Both classes lend themselves to recursive algorithms. Generally:

- For **Tree**: The base case is when **is_leaf()** is true; the recursive call is on the **branches**.
- For **Link**: The base case is when the rest is **empty**; the recursive call is on the **rest**.