CS61A Lecture 13

Thursday, September 26th, 2019

Announcements

- Cats checkpoint deadline on Monday.
- Homework 4 is due in 2 weeks' time.
- Flatten problem will not graded on Homework 3.

Box-and-Pointer Notation

What does a list look like in an environment diagram? We use a box-and-pointer notation, because of the closure property of data type.

A method for combining data values satisifies the closure property if the result of combination can itself be combined using the same method.

- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else).

Lists are represented as a row of index-labaled adjacent boxe, one per element. Each box either contains a primitive value or points to a compound value.

We use an arrow to point to lists, just like to a function name.

Slicing

Slicing is a method of cutting up lists into only the elements we want.

```
>>> odds = [1,3,5]
>>> odds
[1,3,5]
```

Previously, we used list comprehension to do this:

```
>>> [odds[i] for i in range(0,2)]
[1,3]
```

Now, we can just:

```
>>> odds[0:2]
[1,3]
```

Slicing takes three arguments, `start`, `end` and `step`. You can "skip" ahead to end or step, depending on what you want, by just not putting anything between colons:

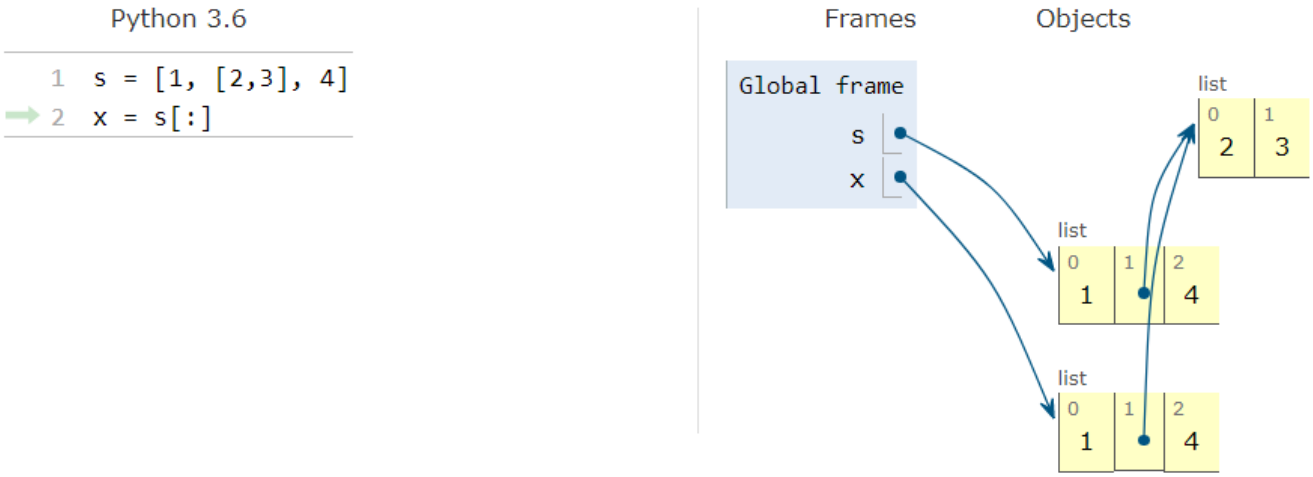
```
>>> odds[:2]
[1,3]
```

You can access the last element with just -1:

```
>>> odds[-1]
[5]
```

Every time you slice, you get a new list. You do not change the original list, which we will talk about in Monday's lecture.

You only create a **shallow copy** of the original list, which means if your list contains another list, it will only copy the *pointer* to the nested list, not the entire list with copied nested lists.



Container Processing

Python is by some measures, the world's most popular programming language, which wasn't true when we started teaching 61A in Python. So we just guessed well!

Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value.

```
>>> sum(range(5))
10
>>> sum(range(5),20)
30
```

`sum` takes the arguments `iterable` and `[, start]`.

```
>>> max(range(6))
5
```

`max` and `min` not only take integer and float arguments, but also a `key` function. You will get back the value within the arguments passed in that is the largest once the key function has been called. For example:

```
>>> max([2,3,5,4], key=lambda x: -x) #-2 is the largest value among all negative values
2
```

You can also use the `all` function to return True if `bool(x)` is True for all values x in the iterable. The function `bool` takes a value and tells you whether it is True or False, and `all` tells you whether all values passed in were true.

The `any` function works similarly, but only needs one value to be True to return True.

> **Calling `max` on lists**
>
> ```
> >>> max([1,2],[3,4])
> [3,4]
> >>> max([3,4].[3])
> [3,4]
> ```

Trees

Wikipedia describes a tree best:

In computer science, a tree is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

Tree Abstraction

We can describe a tree in two ways, with both a **recrusive description** and a **relative description**.

Recursive	**Relative**
A tree has a root and a list of branches	Each location in a tree is called a node
Each branch is a tree	Each node has a label value
A tree with zero branches is called a leaf	One node can be the parent/child of another

There is no built-in tree class in Python, because we can abstract it away. Today, we will use lists to do so:

We need a constructor:

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

Technically, the code we are using in this class is a bit more complicated:

```
def tree(label, branches):
    for branch in branches:
```

```
        assert is_tree(branch), "branches are trees"
    return [label] + list(branches)
```

We will also need selectors:

```
def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:0]
```

How do we check if something is a tree?

```
def is_tree(tree):
if type(tree) != list or len(tree) < 1:
    return False
for branch in branches(tree):
    if not is_tree(branch):
        return False
return True
```

And how to know if something is a leaf?

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

> **Violating and not violating abstraction barriers**
> This does not violate the barrier:
>
> ```
> >>> t = tree(4, [tree(2), tree(3)])
> >>> branches(t)[0]
> [2]
> >>> label(branches(t)[0])
> 2
> ```
>
> This does however:
>
> ```
> >>> branches(t)[0][0]
> 2
> ```

Tree Processing

There are two main uses of tree recursion, one is a problem like count change, and the other is to do things with tree-structured data.

Processing a leaf is often the base case of a tree processing function. The recursive case typically makes a recursive call on each branch, then aggregates. For example,

```
def count_leaves(t):
```

```
"""Count the leaves of a tree."""
if is_leaf(t):
    return 1
else:
    branch_counts = [count_leaves(b) for b in branches(t)]
    return sum(branch_counts)
```

This is tree recursive because there could be multiple branches of `b`, calling `count_leaves` multiple times.

Creating Trees

A function that creates a tree from another tree is also typically recursive. So let's say you have `fib_tree`, and you want to increment every leaf by 1.

```
def increment_leaves(t):
    if is_leaf(t):
        return tree(label(t)+1)
    else:
        bs = [increment_leaves(b) for b in branches(t)]
        return tree(label(t),bs)
```

Or you want to increment every element by 1! This is how:

```
def increment(t):
    return tree(label(t)+1, [increment(b) for b in branches(t)])
```

Where's the base case? Well, if you call incrmeent on a leaf, then it will just return an empty list! There's the implicit base case hiding in the function.