CS61A Lecture 20

Monday, October 14th, 2019

Announcements

- Office hours today from 3:30 to 6:30 in Soda 275.
- Homework/project party from 6:30 to 8:00 in Cory 241.
- Homework 4 and hog composition extended to today.
- Midterm on Thursday 10/24 from 8 to 10pm.

Compositional Data

Objects that as part of them, have other objects. They could even be the same object again, which makes these structures inherently recursive.

One example is a linked list. A linked list is either empty, for a first value and the rest of the linked list.

```
[3,-]->[4,-]->[5,-]->[]
```

A `Link` in the code only consists of the value itself, and a pointer to another `Link`.

It is not inherently built into Python itself, or any other general programming language, but it is usually manually coded by programmers and customize them to the use case they are required.

To build the above `Link`, we use the following code:

```
Link(3,Link(4,Link(5,Link.empty)))
```

You can represent the empty "end" to the `Link` in any way you want. It could be the value `None`, or an empty tuple, or whatever reasonable definition. By giving `Link.empty` a name, we can change it with abstraction later on.

This is used so often in computer science there is a convention on how to draw it, which is the empty list that is usually just a slash. We can input this into our code so that the `Link.empty` is an optional argument.

Here's how to do it:

```
class Link:
    empty = ()
    def __init__(self,first,rest=empty):
        assert rest is Link.empty or instance(rest,Link)
        self.first = first
        self.rest = rest
```

Now we can access our data with a combination of `first` and `rest`?

```
>>> s = Link(3,Link(4,Link(5)))
>>> s.first
3
>>> s.rest.first
4
```

> **Why is `empty` not `Link.empty`?**
> This is a quirk of Python's execution rules, such that the `def` statement is executed before the class is even built, so the only way to refer to the `empty` name is through `name`, at least in this `def` statement, for which it is in scope.

The reason why linked lists are used is for efficiency, which we will discuss on Wednesday. Imagine you wanted to represent the numbers 2, 3, 4 and 5. With linked lists, the 3, 4 and 5 are already built as the 2, 3, 4 and 5 is being built. As a result, you can share memory between the two implementations.

```
    def __repr__(self):
        if self.rest:
            rest_repr = ', '+ repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.rest) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Let's say we want more than just to be able to get `first` and `rest`. How can do that?

Property Methods

In some cases, we want the value of instance attributes to be computed on demand. For example:

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s
Link(3,Link(6,Link(5)))
```

We can do this with the property decorator `@property`.

First, we build a bound method:

```
def second(self):
    return self.rest.first
```

But this method is a bound method, so we change it by adding the decorator:

```
@property
def second(self):
    return self.rest.first
```

For the same reason, the @.setter function changes the behavior of an assignment statement:

```
@second.setter
def second(self,value):
    return self.rest.first = value
```

Tree Class

To make sure you're ready for this week's labs and discussion, we will represent Trees using both the data abstraction rules from before, or, more importantly, using the Python object system.

> **Review**
>
> 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 |

A Tree has a label and a list of branches; each branch is a Tree.

```
class Tree:
    def __init__(self,label,branches=[])
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

The important idea we are learning now is that everything we could do with OOP, we could've done just with functions too.

The nice thing about the object definition is that in the data abstraction definition, we had to invent how we were going to represent the data, and how to return what the user wanted.

```
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(label.left,[left,right])
```

And remember that with the `repr` and `str` functions, we no longer have to build all-new functions like `print_tree`, which you may have seen before.

We might still need to define our own functions, for example, `leaves`:

```
def leaves(t):
    if t.is_leaf():
        return 0
    else:
        return
```