Computer Science 1MD3
Tutorial 7: Classes and Avanced Data Types

Aaron Simpson

A new concept that you haven't yet been introduced to is that of a recursive data type. You have already been exposed to recursive functions, where the function itself appears in its own definition. Recursive data types are much the same, one of the elements defined in each linked list actually contains a linked list in itself. This will make more sense as you see its implementation in practice.

Definition of Linked Lists

```
class Node:
        def __init__(self, cargo=None, next=None):
                self.cargo = cargo
                self.next = next

        def __str__(self):
                return str(self.cargo)
```

As with most new concepts there is some syntax involved in this definition that you haven't been exposed to yet. Rather than explaining how the new elements specifically work, it may make more sense to see what they do in the context of this problem. The first unfamiliar part is the *class* notation. This has the same shape as when defining a function (not that it does not need the *def* keyword to begin). Part of the reason that the definition for linked lists is recursive is because we are not defining the linked list class as a whole, instead we are defining how a single node in that list works, and referring to other nodes in that definition. On the next line you will see the familiar *def* again with an oddly shaped new word *init* with double underscores on either side of it. You might guess that this is shaped like a function definition, and you'd be right. The reason for the special notation is that whenever you define a special method called *__init__* inside a new class, this method is called every time the class is instantiated (as in, whenever you make a new variable of the class's type). This makes the *__init__* method very useful for preparing a class for its components to be manipulated. As you can see the *__init__* method takes 3 arguments, *self*, *cargo*, and *next*. The use of *=None* after *cargo* and *next* are what's called “default values”. This means that if no *next* parameter is given, its value is automatically set to *None* when the class is initialized. The same is true for the *cargo* parameter if no value is given for it. The first parameter of a method is basically always the special *self* parameter, which Python uses for if the object needs to perform an action on itself. The way this works is rather obscure and you do not need to worry about it at the moment. This means however that when instantiating an empty node, you can do so with empty braces. Doing so would look like this:

```
nameformynode = Node()
```

This would create a new *Node* object called n*ameformynode*. Its *cargo* and *next* parameters would automatically be set to *None* as it is initiated.

The next method defined is the *__str__* method. This is a special case as well in that it teaches the class what to do if some other function is trying to treat the class as a string (i.e., if something tried to print this class). The entire method consists of the line *return str(self.cargo)*. This means, "If a function wants some part of you to work with as a string, convert your *cargo* to a string, and give the function that". Let's say we instantiated a new object like this:

```
newnode = Node("stuff")
```

This would make another new node called *newnode* containing the string *"stuff"* in its *cargo* container. Now, if we were trying to display this node's contents in the interpreter the first thing we might try to do is enter its name.

```
newnode
```

All this will do is produce a message which looks like an error (though it isn't really), telling you there's an instance of the Node class at some memory address. This makes sense when you think about it, as the object *newnode* itself is an entity composed of multiple parts and not something that can be displayed by conventional means. If we were to use the *print* function however:

```
print newnode
```

...then the interpreter knows to use the *__str__* method we defined earlier to figure out exactly what it is that should be printed. In this case, it is the *cargo* converted to a string, which should result in *stuff* being printed to the screen.

Use of the lists

Now that the function of classes should be clearer, we can start to see how linked lists actually utilize this. The cleverness of this data type as well as its recursive element are both seen in the *next* component. The intended use of this element is to point to the next element in the list, linking the objects together. To do this, you actually set the *next* value to equal another node object. For example try entering the following code in the interpreter:

```
>>> node1 = Node("This is the first")
>>> node2 = Node("This is the second")
>>> node1.next = node2
>>> print node1
>>> print node1.next
```

If we wanted, we could give a node its *cargo* and *next* parameters both at once, like we can give multiple parameters to any function.

```
>>> node1 = Node("This is the first", node2)
```

This would result in the same *node1* as in the previous example. We cannot, however, replace the first line in that example with this one, as *node2* doesn't exist yet when *node1* is first being defined.

Trees

If you understand the concept of linked lists properly, trees are a very small logical step after that. One way of having them function is to represent them as a linked list, but with more than one link available. For example:

```
class TreeNode:
        def __init__(self, cargo=None, left=None, right=None):
                self.cargo = cargo
                self.left = left
                self.right = right

        def __str__(self):
                return str(self.cargo)
```

The *left* and *right* components replace *next* here, allowing each node to form a path to more than one location. Each of these paths can be seen as a "branch" on the tree. The implementation of the cargo remains the same as in the linked list.

Any remaining time in this tutorial can be spent reviewing the basics of these classes with your TA. These data types are highly testable material so try to make a point of memorizing their idioms to save you time later on.