11. Software Design for NLP

11.1 Object-Oriented Programming in Python

Object-Oriented Programming is a programming paradigm in which complex structures and processes are decomposed into modules, each encapsulating a single data type and the legal operations on that type.

11.1.1 Data Classes: Trees in NLTK

An important data type in language processing is the syntactic tree. Here we will review the parts of the NLTK code which defines the **`Tree`** class.

The first line of a class definition is the **`class`** keyword followed by the class name, in this case **`Tree`**. This class is derived from Python's built-in **`list`** class, permitting us to use standard list operations to access the children of a tree node.

```
>>> class Tree(list):
```

Next we define the initializer, also known as the *constructor*. It has a special name, starting and ending with double underscores; Python knows to call this function when you asks for a new tree object by writing **`t = Tree(node, children)`**. The constructor's first argument is special, and is standardly called **`self`**, giving us a way to refer to the current object from inside the definition. This constructor calls the list initializer (similar to calling **`self = list(children)`**), then defines the **`node`** property of a tree.

... def __init__(self, node, children): ... list.__init__(self, children) ... self.node = node

Next we define another special function that Python knows to call when we index a Tree. The first case is the simplest, when the index is an integer, e.g. **`t[2]`**, we just ask for the list item in the obvious way. The other cases are for handling slices, like **`t[1:2]`**, or **`t[:]`**.

... def __getitem__(self, index): ... if isinstance(index, int): ... return list.__getitem__(self, index) ... else: ... if len(index) == 0: ... return self ... elif len(index) == 1: ... return self[int(index[0])] ... else: ... return self[int(index[0])][index[1:]] ...

This method was for accessing a child node. Similar methods are provided for setting and deleting a child (using **`__setitem__`**) and **`__delitem__`**).

Two other special member functions are **`__repr__()`** and **`__str__()`**. The **`__repr__()`** function produces a string representation of the object, one which can be executed to re-create the object, and is accessed from the interpreter simply by typing the name of the object and pressing 'enter'. The **`__str__()`** function produces a human-readable version of the object; here we call a pretty-printing function we have defined called **`pp()`**.

... def __repr__(self): ... childstr = ' '.join([repr(c) for c in self]) ... return '(%s: %s)' % (self.node, childstr) ... def __str__(self): ... return self.pp()

Next we define some member functions that do other standard operations on trees. First, for accessing the leaves:

... def leaves(self): ... leaves = [] ... for child in self: ... if isinstance(child, Tree): ... leaves.extend(child.leaves()) ... else: ... leaves.append(child) ... return leaves

Next, for computing the height:

... def height(self): ... max_child_height = 0 ... for child in self: ... if isinstance(child, Tree): ... max_child_height = max(max_child_height, child.height()) ... else: ... max_child_height = max(max_child_height, 1) ... return 1 + max_child_height

And finally, for enumerating all the subtrees (optionally filtered):

... def subtrees(self, filter=None): ... if not filter or filter(self): ... yield self ... for child in self: ... if isinstance(child, Tree): ... for subtree in child.subtrees(filter): ... yield subtree

11.1.2 Processing Classes: N-gram Taggers in NLTK

This section will discuss the **`tag.ngram`** module.

11.2 Algorithm Design

An *algorithm* is a "recipe" for solving a problem. For example, to multiply 16 by 12 we might use any of the following methods:

1. Add 16 to itself 12 times over
2. Perform "long multiplication", starting with the least-significant digits of both numbers
3. Look up a multiplication table
4. Repeatedly halve the first number and double the second, 16*12 = 8*24 = 4*48 = 2*96 = 192
5. Do 10*12 to get 120, then add 6*12

Each of these methods is a different algorithm, and requires different amounts of computation time and different amounts of intermediate information to store. A similar situation holds for many other superficially simple tasks, such as sorting a list of words. Now, as we saw above, Python provides a built-in function **`sort()`** that performs this task efficiently. However, NLTK-Lite also provides several algorithms for sorting lists, to illustrate the variety of possible methods. To illustrate the difference in efficiency, we will create a list of 1000 numbers, randomize the list, then sort it, counting the number of list manipulations required.

```
>>> from random import shuffle
>>> a = range(1000)                    # [0,1,2,...999]
>>> shuffle(a)                         # randomize
```

Now we can try a simple sort method called *bubble sort*, which scans through the list many times, exchanging adjacent items if they are out of order. It sorts the list **a** in-place, and returns the number of times it modified the list:

```
>>> from nltk_lite.misc import sort
>>> sort.bubble(a)
250918
```

We can try the same task using various sorting algorithms. Evidently *merge sort* is much better than bubble sort, and *quicksort* is better still.

```
>>> shuffle(a); sort.merge(a)
6175
>>> shuffle(a); sort.quick(a)
2378
```

Readers are encouraged to look at **nltk_lite.misc.sort** to see how these different methods work. The collection of NLTK-Lite modules exemplify a variety of algorithm design techniques, including brute-force, divide-and-conquer, dynamic programming, and greedy search. Readers who would like a systematic introduction to algorithm design should consult the resources mentioned at the end of this tutorial.

11.2.1 Exercises

1. Consider again the problem of hyphenation across linebreaks. Suppose that you have successfully written a tokenizer that returns a list of strings, where some strings may contain a hyphen followed by a newline character, e.g. **long-\nterm**. Write a function which iterates over the tokens in a list, removing the newline character from each, in each of the following ways:

   a) Use doubly-nested for loops. The outer loop will iterate over each token in the list, while the inner loop will iterate over each character of a string.

   b) Replace the inner loop with a call to **re.sub()**

   c) Finally, replace the outer loop with call to the **map()** function, to apply this substitution to each token.

   d) Discuss the clarity (or otherwise) of each of these approaches.

2. Develop a simple extractive summarization tool, which prints the sentences of a document which contain the highest total word frequency. Use **FreqDist** to count word frequencies, and use **sum** to sum the frequencies of the words in each sentence. Rank the sentences according to their score. Finally, print the *n* highest-scoring sentences in document order. Carefully review the design of your program, especially your approach to this double sorting. Make sure the program is written as clearly as possible.

11.3 Further Reading

David Harel (2004). *Algorithmics: The Spirit of Computing* (Third Edition), Addison Wesley.

Anany Levitin (2004). *The Design and Analysis of Algorithms*, Addison Wesley.