Lab 0

The purpose of this lab is to familiarize you with this term's lab system and to diagnose your programming ability and facility with Python. 6.034 uses Python for all of its labs, and you will be called on to understand the functioning of large systems, as well as to write significant pieces of code yourself.

While coding is not, in itself, a focus of this class, artificial intelligence is a hard subject full of subtleties. As such, it is important that you be able to focus on the problems you are solving, rather than the mechanical code necessary to implement the solution.

Python resources

Some resources to help you knock the rust off of your Python:

- Any of the many good Python handbooks out there, such as:
  - Dive Into Python, for experienced programmers
  - O'Reilly's Learning Python
  - Think Python, for beginning programmers
- The standard Python documentation, at [1] (the Library Reference and the Language Reference are particularly useful, if you know what you're looking for)

Python

There are a number of versions of Python available. This course will use standard Python ("CPython") from http://www.python.org/. If you are running Python on your own computer, you should download and install Python 2.5 or Python 2.6 from http://www.python.org/download/ . All the lab code will require at least version 2.3.

Mac OS X comes with Python 2.3 pre-installed, but the version you can download from python.org has better support for external libraries and a better version of IDLE.

Answering questions

The main file of this lab is called `lab0.py`. Open that file in IDLE. The file contains a lot of incomplete statements that you need to fill in with your solutions.

The first thing to fill in is a multiple choice question. The answer should be extremely easy. Many labs will begin with some simple multiple choice questions to make sure you're on the right track.

Run the tester

Every lab comes with a file called `tester.py`. This file checks your answers to the lab. For problems that ask you to provide a function, the tester will test your function with several different inputs and see if the output is correct. For multiple choice questions, the tester will tell you if your answer was right. Yes, that means that you never need to submit wrong answers to multiple choice questions.

The tester has two modes: "offline" mode (the default), and "online" or "submit" mode. The former runs some basic, self-contained internal tests on your code. It can be run as many times as you would like.

The latter runs some more tests, some of which may be randomly generated, and uploads your code to the 6.034 grader for grading.

You can run the online tester as many times as you want. If your code fails a test, you can submit it and try again. Because you always have the opportunity to fix your bugs, you can only get a 5 on a problem set if it passes all the tests. If your code fails a test, your grade will be 4 or below.

Using IDLE

If you are using IDLE, or if you do not have easy access to a command line (as on Windows), IDLE can run the tester.

Open the `tester.py` file and run it using Run Module or F5. This will run the offline tests for you. When the offline tests pass (or when you're up against a deadline, or when you have questions for the staff) you can

```
 test_online()
```

to submit your code and run the online tests.

In fact, it will run the submission and online test just as soon as you pass the offline tests, saving you a few keystrokes.

You should run the tester (and submit!) early and often. Think of it as being like the "Check" button from 6.01. It makes sure you're not losing points unnecessarily. Submitting your code makes it easy for the staff to look at it and help you.

Using the command line

If you realize just how much emacs and/or the command line rock, then you can open your operating system's Terminal or Command Prompt, and `cd` to the directory containing the files for Lab 0. Then, run:

```
python tester.py
```

to run the offline tester, and

```
python tester.py submit
```

to submit your code and run the online tester.

You should run the tester (and submit!) early and often. Think of it as being like the "Check" button from 6.01. It makes sure you're not losing points unnecessarily. Submitting your code makes it easy for the staff to look at it and help you.

Python programming

Now it's time to write some Python.

Warm-up stretch

Write the following functions:

- `cube(n)`, which takes in a number and returns its cube. For example, cube(3) => 27.
- `factorial(n)`, which takes in a non-negative integer *n* and returns *n!*, which is the product of the integers from 1 to *n*. (0! = 1 by definition.)

    We suggest that you should write your functions so that they raise nice clean errors instead of dying messily when the input is invalid. For example, it would be nice if factorial rejected negative inputs right away; otherwise, you might loop forever. You can signal an error like this:
    `raise Exception, "factorial: input must not be negative"`
    Error handling doesn't affect your lab grade, but on later problems it might save you some angst when you're trying to track down a bug.

- `count_pattern(pattern lst)`, which counts the number of times a certain pattern of symbols appears in a list, including overlaps. So `count_pattern( ('a', 'b'), ('a', 'b', 'c', 'e', 'b', 'a', 'b', 'f'))` should return 2, and `count_pattern(('a', 'b', 'a'), ('g', 'a', 'b', 'a', 'b', 'a', 'b', 'a'))` should return 3.

Expression depth

One way to measure the complexity of a mathematical expression is the depth of the expression describing it in Python lists. Write a program that finds the depth of an expression.

For example:

- `depth('x') => 0`
- `depth(('expt', 'x', 2)) => 1`
- `depth(('+', ('expt', 'x', 2), ('expt', 'y', 2))) => 2`
- `depth(('/', ('expt', 'x', 5), ('expt', ('-', ('expt', 'x', 2), 1), ('/', 5, 2)))) => 4`

Note that you can use the built-in Python "isinstance()" function to determine whether a variable points to a list of some sort. "isinstance()" takes two arguments: the variable to test, and the type (or list of types) to compare it to. For example:

```
>>> x = [1, 2, 3]
>>> y = "hi!"
>>> isinstance(x, (list, tuple))
True
>>> isinstance(y, (list, tuple))
False
```
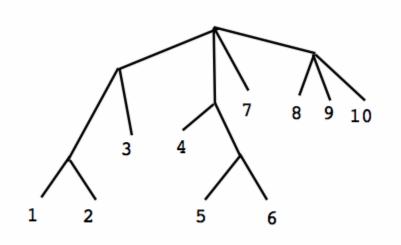
Tree reference

Figure 1: Example Tree

Your job is to write a procedure that is analogous to list referencing, but for trees. This "tree_ref" procedure will take a tree and an index, and return the part of the tree (a leaf or a subtree) at that index. For trees, indices will have to be lists of integers. Consider the tree in Figure 1, represented by this Python tuple: `(((1, 2), 3), (4, (5, 6)), 7, (8, 9, 10))`

To select the element 9 out of it, we'd normally need to do something like `tree[3][1]`. Instead, we'd prefer to do `tree_ref(tree, (3, 1))` (note that we're using zero-based indexing, as in list-ref, and that the indices come in top-down order; so an index of (3, 1) means you should take the fourth branch of the main tree, and then the second branch of that subtree). As another example, the element 6 could be selected by `tree_ref(tree, (1, 1, 1))`.

Note that it's okay for the result to be a subtree, rather than a leaf. So `tree_ref(tree, (0,))` should return `((1, 2), 3)`.

Symbolic algebra

Throughout the semester, you will need to understand, manipulate, and extend complex algorithms implemented in Python. You may also want to write more functions than we provide in the skeleton file for a lab.

In this problem, you will complete a simple computer algebra system that reduces nested expressions made of sums and products into a **single sum of products**. For example, it turns the expression `(2 * (x + 1) * (y + 3))` into `((2 * x * y) + (2 * x * 3) + (2 * 1 * y) + (2 * 1 * 3))`. You could choose to simplify further, such as to ((2 * x * y) + (6 * x) + (2 * y) + 6)), but it is not necessary.

This procedure would be one small part of a symbolic math system, such as the integrator presented in Monday's lecture. You may want to keep in mind the principle of reducing a complex problem to a simpler one.

An algebraic expression can be simplified in this way by repeatedly applying the associative law and the distributive law.

Associative law
      $((a + b) + c) = (a + (b + c)) = (a + b + c)$
      $((a * b) * c) = (a * (b * c)) = (a * b * c)$
Distributive law
      $((a + b) * (c + d)) = ((a * c) + (a * d) + (b * c) + (b * d))$

The code for this part of the lab is in `algebra.py`. It defines an abstract `Expression` class, `Sum` and `Product` expressions, and a method called `Expression.simplify()`. This method starts by applying the associative law for you, but it will fail to perform the distributive law. For that it delegates to a function called `do_multiply` that you need to write. Read the documentation in the code for more details.

This exercise is meant to get you familiar with Python and using it to solve an interesting problem. It is intended to be algorithmically straightforward. You should try to reason out the logic that you need for this function on your own. If you're having trouble expressing that logic in Python, though, don't hesitate to ask a TA.

Some hints for solving the problem:

- How do you use recursion to make sure that your result is thoroughly simplified?
- In which case should you *not* recursively call `simplify()`?

Survey

We are always working to improve the class. Most labs will have at least one survey question at the end to help us with this. Your answers to these questions are purely informational, and will have no impact on your grade.

Please fill in the answers to the following questions in the definitions at the end of lab0.py:

- When did you take 6.01?
- How many hours did 6.01 projects take you?
- How well do you feel you learned the material in 6.01?
- How many hours did this lab take you?

When you're done

Remember to run the tester! The tester will automatically upload your code to the 6.034 server for grading and collection.

6.034 Artificial Intelligence

Fall 2010