**Python track: lab 3: fun with cryptography**

This lab, you're going to be implementing a very rudimentary cryptography program in python. Your program will have to generate a random substitution code for all the letters in the range a-z, mapping them to other letters in a-z such that the letters can be translated uniquely back to the original ones. All other characters will be passed through unchanged. Your program will use this code to translate a text file (which you supply, though it should work correctly regardless of the contents of the file) and then translate it back into the original form. The point is not so much to do world-class cryptography, but rather to give you some more experience working with objects and modules in python.

We are supplying you with a [test script](http://courses.cms.caltech.edu/cs11/material/python/lab3/test_crypto) called test\_crypto. Note: this will only work on a Unix system, so please use a Unix system to develop this (or at least to test it).

You have to write two files:

1. **char\_translator.py**   This file will contain code to generate the random substitution code and to translate and untranslate individual characters based on that code. **NOTE:** python now contains the string methods string.maketrans() and string.translate() which pretty much do this for you. So don't use either of these methods, or the problem becomes totally trivial.
2. **crypto.py**   This file will do the actual translation and untranslation of files.

Here are the classes and methods that should be in each file. Feel free to add more classes or methods if it's convenient for you. In the file "char\_translator.py", you should have:

* A function called **random\_permute\_chars**, which generates and returns a string which is a random permutation of the characters in the range a-z (all lowercase). The python functions**random.shuffle** and **string.join** will be useful to you; see the python library documentation to learn how they work. To convert a string to a list, do this:
* l = map(lambda x:x, s)

where l is the list and s is the string. Or, more simply, you can just do this:

l = list(s)

Note that the permutation must have no duplicates and must include all the letters from a-z (by definition of permutation). Note also that python strings can be indexed as if they were lists, and you can use

for c in s:

# code using c

(where c is a character and s is a string) to iterate through the characters.

* A class called **CharTranslator**, with methods:
  + **\_\_init\_\_**, which takes a string which is a random permutation of characters in the range a-z as its only argument and stores it in the CharTranslator object. Make sure you check that the string is a valid permutation.
  + **translate\_char**, which translates a single character based on the permutation. In other words, to get the translation of a character, find the corresponding character in the permuted string and return that. You can leave uppercase characters alone if you like, or translate them (making sure that the result is also uppercase). All other characters should pass through unchanged.
  + **untranslate\_char**, which undoes the translation.
* An exception class called **InvalidTranslationString** which is raised if the translation string is invalid. This should be used in the **\_\_init\_\_** method for **CharTranslator**.

In the file "crypto.py", you should have a class called **Coder** which has these methods:

* **\_\_init\_\_**, which takes a CharTranslator object as its argument and stores it in the Coder object.
* **encode\_file**, which takes an input and output file name and encodes the input file, storing the output in the output file.
* **decode\_file**, which takes an input and output file name and decodes the input file, storing the output in the output file.

Once you've developed your code, run the file **test\_crypto** with some file name as its argument. If your modules are correct the program will encode the file, decode it, and test to see if the decoded version is the same as the encoded version. Try to make sure that your test file has all the letters from a-z and A-Z as well as some non-alphabetic characters. Program source code is a good input to use.

**Hints**

Here are a few things to keep in mind while you write this program.

* You may find the ord and chr functions to be useful. They are built-in functions that convert characters (strings of length 1) to ASCII codes and back. Use pydoc to find out more.
* Use functions and methods from the string and list modules (type "pydoc string" and "pydoc list" at the shell prompt to find out more about these functions and methods) to make your program more concise. In particular, the string "abcde...z" is just string.ascii\_lowercase. Also check out the join and find functions, and the index and sort methods on lists. Also note that you can do comparisons on strings and lists, not just on numbers.
* Some string functions can be used as methods *e.g.* the string.join() function, which can be written like this:
* >>> "".join(["foo", "bar", "baz"])
* 'foobarbaz'

This says to join the given list of strings with an empty string as the separator.

* Make sure you close all files that you open using the close() method of file objects.
* Assume that the input file can be arbitrarily large (e.g. one gigabyte). Because of this, you don't want to load the entire file into memory before you output anything, even if this would make your code a bit more concise or more clever-looking.
* To iterate through the lines in a file you can use "for line in file: ..." instead of an infinite loop with a break statement in the middle.
* Since the files you need to write are modules, not executable programs, you don't need the "#! /usr/bin/env python" line at the beginning and you don't need to do "chmod +x" on the files. However, the "test\_crypto" file I supply you with (note the lack of the ".py" suffix) **is** an executable file, so you should do a "chmod +x" on it after you download it.
* When an error occurs in a class method, you should **not** print an error message and call "sys.exit(1)". You have no idea what context the class instances will be used in, but normally they will be part of a larger program (as they are in this case), and exiting when one thing goes wrong may not be the right thing to do -- you just don't know. So instead, you should raise an exception, preferably with an error message that tells the code that called the method what went wrong. In some cases (*e.g.* opening a non-existent file) an exception will be thrown from a method you call that will contain the error message already; in that case, you don't have to catch the exception and re-throw a different one (or, even worse, the same one); just let it go through (this may come as a shock to Java programmers, but rest assured, it's fine).

**Python track: lab 4: fun with cellular automata**

This lab, you're going to be implementing a general-purpose program for computing one-dimensional cellular automata in python. We will use this program next week to generate interesting graphical patterns.

**One-dimensional cellular automata**

Most programmers have either written or at least played with programs that simulate the game of "Life". Life is a particular kind of two-dimensional cellular automaton. In this assignment we'll be looking at one-dimensional cellular automata (1dCA). (Note that "automata" is the plural of the word "automaton".) You'll see that 1dCAs can also generate lots of interesting patterns, including both periodic and "chaotic" patterns. In fact, some 1dCAs are actually universal models of computation all by themselves, but we won't go into that here.

A 1dCA is composed of the following elements:

1. An array of **cells**, each of which has a **state** (represented by a number between 0 and the maximum number of states minus 1).
2. A **neighborhood radius**, which states how many cells in either direction affect the next state of any given cell. The cell's next state depends only on the states of these cells as well as its own current state.
3. A **state transition table** that determines which state to set a given cell to, given the previous state of the cell and the states of the cells in its neighborhood.

Some implications of this are:

1. The 1dCA is entirely deterministic.
2. The next state of the cells in the 1dCA depends only on the current states of the cells.

We will add one extra rule to simplify our program: the next state of a given cell depends only on the **sum** of the states of the cells in its neighborhood and its own state.

There is always the question of what to do for the edges of the automaton, where not all the cells in the neighborhood may be available. Two obvious choices are:

1. assume that the unavailable (missing) cells have states = 0, or
2. apply a "wraparound" rule whereby   cell[-1] = cell[nstates-1] and cell[nstates] = cell[0]. Make sure you wrap around on both sides if you do this.

You are free to use either rule in your program, or to implement both (user-selectable, of course ;-)).

**Description of the program**

You are to write a python module called **automaton.py** which implements a 1dCA with the following characteristics:

1. The above updating rules are implemented.
2. An arbitrary sized cell array can be specified.
3. An arbitrary number of states can be specified.
4. An arbitrary number of nearest neighbors can be specified.
5. An arbitrary update rule (state transition table) can be specified.

Note that the size of the state transition table is a function of the number of states and the number of neighbors.

Your module should include a class called **cellular\_automaton\_1d**. You should implement (at least!) these methods for your class (you may choose different names if you like):

1. **\_\_init\_\_**, of course.

Use this to specify values (including default values) for the number of states, number of nearest neighbors, and the state transition table (which should default to None, meaning it will be generated randomly). You should also specify the size of the cell array, though you don't need to specify a default size.

The format of the state transition table is an array of integers where:

* + the index (position) in the array represents the sum of the state of a cell and the states of its nearest neighbors
  + the contents of the array at that index represent the next state of the cell

Note that the states are represented as positive integers starting from 0.

**NOTE: Check for errors!** You should raise an exception if any of the arguments to the constructor are invalid (*e.g.* some arguments that should be integers aren't integers, some arguments are out of range (*e.g.* a negative number of nearest neighbors) the state transition table is too small for the number of states and the number of neighbors, the contents of the table are invalid, etc.). **Check for every possible error**, not just for the ones that come to you off the top of your head.

1. **generate\_random\_table**, which will generate a valid random state transition table for the given number of neighbors and states.
2. **random\_initialize**, which will randomly initialize each cell in the cell array to a valid state.
3. **update**, which will use the current values of the cell array in order to generate the next set of values. You will need a temporary array for this. Make sure you don't change the original array until the new array is completely computed.
4. **get\_states**, which will return the cell array (ideally, a copy of the cell array).
5. **dump**, which will print the cell array to stdout. If there are no more than 10 states, each cell should print as a single digit in the range [0-9] (with no spaces between the digits); otherwise, just print the array directly using python's "print" statement. To print a single digit without a newline or a space after it, you can use the sys.stdout.write() function.

**Testing your class**

At the end of your module (outside of the class) put this code:

if \_\_name\_\_ == "\_\_main\_\_":

try:

# Arguments here to the constructor are:

# size of cell array, number of states, number of neighbors (on each

# side), the state transition table.

a = cellular\_automaton\_1d(70, 2, 1, [0, 1, 1, 0])

a.random\_initialize()

a.dump()

for i in range(100):

a.update()

a.dump()

except ... # whatever can be thrown in the try block

Since your constructor can throw exceptions, you should put it in a try/except block as shown, catch all exceptions it raises, and print out a usage message where appropriate. **Do not use a catch-all exception handler.**

Test your class by typing

% python automaton.py

(where % is the prompt). This should generate an interesting pattern of 1s and 0s. (If the pattern isn't interesting, you probably have a bug.) If you want to see the pattern more clearly, do this:

% python automaton.py | tr "01" ".\*"

This will show a nice fractal-like pattern. Try other arguments to the constructor and see what kinds of patterns you generate.

**NOTE**: If the pattern displayed by the program doesn't look cool, then you've absolutely certainly made a mistake. Fix it before submitting it! It's surprising how many students have trivial mistakes in their lab 4 code.