Natural Language Processing:

Assignment 2: Finite State Adventures
Census Nomenclature & Number Translation

Jordan Boyd-Graber

Out: **25. August 2014**
Due: **19. September 2014**

Introduction

This assignment is on finite state automata and will require you to create automata in python. There are a total of three problems, and we have provided some code to get you started, as well as a few utilities that will make it easier for you to debug and test your code.

We've provided an a FST implementation from NLTK. However, the documentation is less than perfect. Therefore, in this section, we will give you a brief introduction to everything that you will need to understand how to build finite state transducers in Python.

To start building FSTs, you need to first import the `fst` module into your program's namespace. Then, you need to instantiate an `FST` object. Once you have such an object, you can start adding states and arcs to it. Listing 1 shows how to build a very simple finite state transducer—one that removes all vowels from any given word.

Feel free to try out the example to see how it works on some of your own input. There are a few points worth mentioning:

1. The Python `string` module comes with a few built-in strings that you might be able to use in this assignment for purposes of iteration as used in the example on line 23. These are:

   - `string.letters` : All letters, upppercased and lowercased
   - `string.ascii_lowercase` : All lowercased letters
   - `string.ascii_uppercase` : All uppercased letters

Listing 1: A 1-state transducer that deletes vowels

```
# import the fst module
import fst

# import the string module
import string

# Define a list of all vowels for convenience
vowels = ['a', 'e', 'i', 'o', 'u']

# Instantiate an FST object with some name
f = fst.FST('devowelizer')

# All we need is a single state ...
f.add_state('1')

# and this same state is the initial and the final state
f.initial_state = '1'
f.set_final('1')

# Now, we need to add an arc for each letter; if the letter is a vowel
# then then transition outputs nothing but otherwise it outputs the same
# letter that it consumed.
for letter in string.ascii_lowercase:
    if letter in vowels:
        f.add_arc('1', '1', (letter), ())
    else:
        f.add_arc('1', '1', (letter), (letter))

# Evaluate it on some example words
print ''.join(f.transduce(['v', 'o', 'w', 'e', 'l']))
print ''.join(f.transduce('e x c e p t i o n'.split()))
print ''.join(f.transduce('c o n s o n a n t'.split()))
```

2. States can be added to an FST object by using its `add_state()` method. This method takes a single argument: a unique string identifier for the state. Our example has only one state (line 13). Furthermore, there can only be **one** initial state and this is indicated by assigning the state identifier to the FST object's `initial_state` field (line 17). However, there may be multiple final states in an FST. In fact, it is almost always necessary to have multiple final states when working with transducers. All final states may be so indicated by using the FST object's `set_final()` method (line 18).

3. Arcs can be added between the states of an FST object by using its `add_arc()` method. This method takes the following arguments (in order): the starting state, the ending state, the input symbol and, finally, the output symbol. If you wish to use single characters as input or output symbols, you must enclose them in parentheses (lines 25 and 27).

   However, if you wish to use entire words as input or output symbols, you must enclose the word in **square brackets** (not in parentheses). For example, if you wish to add an arc that takes the string *ten* as input and returns the number string *10* when going from state 1 to 2, you should use:

   ```
   f.add_arc('1', '2', ['ten'], ['10'])
   ```

   ϵ's may be indicated by an empty set of parentheses or square brackets, depending on the context. (line 25).

4. An FST object can be evaluated against any input string by using its `transduce()` method. Here's how:

   (a) If your transducer uses **characters** as input/output symbols, then the input to `transduce()` must be a **list of characters**. You may either directly input a list of characters (line 30) or you may convert a string to a list of characters by spacing out its characters and calling its `split()` method (lines 31 and 32).

   (b) If your transducer uses **words** as input/output symbols, then the input to `transduce()` should be a **list of words**. Again, you can either explicitly use a list of words or call the split method on a string of words separated by whitespace. For example, say your FST maps from strings like *ten* and *twenty* to number strings *10*

and *20*, then to evaluate it on the input string *ten twenty*, you should use either:

```
f.transduce('ten twenty'.split())
```

OR

```
f.transduce(['ten', 'twenty'])
```

Provided Utilities

To make it easier for you to solve the two programming problems below, we have provided 3 handy utilities in the included python file `fsmutils.py`. These utility functions will help you to test each transducer that you build and compose multiple transducers together.

The first is `composechars()`, which allows you to compose any number of transducers (that use single characters as input strings) and evaluate it on any input string[1]. For example, if you have created three transducers `f1`, `f2` and `f3` and you wish to evaluate their composition on the input string `S`, then you should use the following code:

```
from fsmutils import composechars
output = composechars(S, f1, f2, f3)
```

The above function call computes (`f3` ∘ `f2` ∘ `f1`)(`S`). i.e., it will first apply transducer `f1` to the given input `S`, use the output of this transduction as input to transducer `f2` and so on and so forth. It will raise a generic exception if one or more input transducers do not work correctly. Note that since all transducers for this function use single characters as the input symbols, `S` must be a list of characters.

The second utility function is `composewords()` which allows you to compose transducers that use words as input symbols, instead of single characters. The usage is similar to `composechars()` but the input string `S` must be a list of words in order to be used with this function.

[1]Note that this function only performs composition in a practical sense and does not actually create a single composed transducer. However, for this assignment, the former is more than sufficient.

The final utility function is `trace()`. Given any single transducer `f` and a string `S`, this function will print the entire path taken through `f` when using `S` as the input. This can prove extremely invaluable for debugging any transducer. It may be used as follows:

```
from fsmutils import trace
trace(f, S)
```

Problem 1: Soundex (35 points)

Background: The Soundex algorithm is a phonetic algorithm commonly used by libraries and the Census Bureau to represent people's names as they are pronounced in English. It has the advantage that name variations with minor spelling differences will map to the same representation, as long as they have the same pronunciation in English. Here is how the algorithm works:

Step 1: Retain the first letter of the name. This may be uppercased or lowercased.

Step 2: Remove all **non-initial** occurrences of the following letters: `a, e, h, i, o, u, w, y`. (To clarify, this step removes all occurrences of the given characters *except* when they occur in the first position.)

Step 3: Replace the remaining letters (except the first) with numbers:

- `b, f, p, v` $\rightarrow 1$
- `c, g, j, k, q, s, x, z` $\rightarrow 2$
- `d, t` $\rightarrow 3$
- `l` $\rightarrow 4$
- `m, n` $\rightarrow 5$
- `r` $\rightarrow 6$

If two or more letters from the same number group were adjacent in the *original* name, then *only* replace the first of those letters with the corresponding number and ignore the others.

Step 4: If there are more than 3 digits in the resulting output, then drop the extra ones.

Step 5: If there are less than 3 digits, then pad at the end with the required number of trailing zeros.

The final output of applying Soundex algorithm to any input string should be of the form `Letter Digit Digit Digit`. Table 1 shows the output of the Soundex algorithm for some example names.

Input	Output
Jurafsky	J612
Jarovski	J612
Resnik	R252
Reznick	R252
Euler	E460
Peterson	P362

Table 1: Example outputs for the Soundex algorithm.

Construct an FST that implements the Soundex algorithm. Obviously, it is non-trivial to implement a single transducer for the entire algorithm. Therefore, the strategy we will adopt is a bottom-up one: implement multiple transducers, each performing a simpler task, and then compose them together to get the final output. One possibility is to partition the algorithm across three transducers:

1. **Transducer 1**: Performs steps 1-3 of the algorithm, i.e, retaining the first letter, removing letters and replacing letters with numbers.
2. **Transducer 2**: Performs step 4 of the algorithm, i.e., truncating extra digits.
3. **Transducer 3**: Performs step 5 of the algorithm, i.e., padding with zeros if required.

Note that each of these three transducers will have characters as input/output symbols.

To make things easier for you, we have provided the file `soundex.py` which is where you will write your code. It already imports all needed modules and functions (including `fsmutils.py`). It also creates three transducer

objects—as dictated by the bottom-up strategy outlined above—such that all you should have to do is to figure out the states and arcs required by each transducer. It also contains code that allows you to input a single name on the command line to get the output.
Notes:

(i) Your grade will only be evaluated on the command line. You do not have to factor the code in the same way as the unit tests; however, the unit tests will no longer work.

(ii) While we have provided you with sample unit tests containing some names, it might be very useful to test your code on other names. For comparison purposes, you may use one of the many Soundex calculators available online.

Problem 2 (55 points)

Background: In the French language, Arabic numerals that we use in everyday can be spelled out just like they can in English. For example, the numeral 175 is written as `one hundred seventy five` in English and *cent soixante quinze* in French.

French is interesting becuase they have a mixture of a deximal (base 10) and vegesimal (base 20) system, created by committee to placate two different regions of Franch that used different systems. Thus, for example, you have:

Arabic	French	English Gloss
4	quatre	four
6	six	six
16	seize	sixteen
20	vingt	twenty
26	vingt six	twenty six
56	cinquante six	fifty six
66	soixante six	sixty six
76	soixante seize	sixty sixteen
80	quatre vingt	four twenty
86	quatre vingt six	four twenty six
96	quatre ving seize	four twenty sixteen
996	neuf cent quatre ving seize	nine hundred four twenty sixteen

To summarize:

- Between 70–79 and 90–99, French numbers use a vegesimal base system. For everything else, it is base 10.
- Numbers congruent to 1 mod 10 have an "and". For example, 21 is *vingt et un* ("twenty and one")
- Numbers larger than 100 are written as x hundred. For example, 600 becomes *six cent* ("six hundred").

You may want to consult an online reference, such as `https://www.udemy.com/blog/french-numbers-1-1000/`.

Construct an FST in NLTK that can translate any given Arabic numeral into its corresponding French string. For the sake of convenience, you will only be given integer input less than 1000.

Notes:

(i) Just like for the Soundex problem, we have provided a file called `french_count.py` with boilerplate code. You should add your code to this file, having the `french_count` function return a transducer that produces the correct output.

(ii) You should not type any French string into this file. All necessary French strings are in a dictionary called `kFRENCH_TRANS`. Adding additional French strings will cause you to lose points.

English Morphology (10 points)

English is one of the least interesting languages morphologically, but it's a good warm up (Chinese, Vietnamese are even less interesting). If you take a look at `tests.py`, you can see some of the words we're going to working with: pack, ice, frolic, pace, ace, traffic, lilac, lick. Our goal is to be able to generate things like "spruced" (from "spruce+d") and "picnicking" (from "picnic+ed") using regular expressions (which are magically tranformed into finite state machines[2]).

The shell for this part of the assignment is in `morphology.py`. Essentially all you need to do is implement an additional regular expression rule that will correctly handle c/ck alternations. Before doing this, you should be able to execute `tests.py` to see the tests failing.

[2]How this happens is outside the scope of the course.