Programming Project 1
CSCI 1913: Introduction to Algorithms, Data Structures, and Program Development

Last revised October 2, 2018

0. Introduction.

For this project, you will write a Python class called `Mumble` that generates random strings which look something like English nonsense words. For example, a version of `Mumble` produced nonsense words like `adrisater`, `apongom`, `blarore`, `heldowiff`, `heroqugupaser`, `hinotherund`, `lorinies`, `mersorerent`, `tenerans`, and `whedly`. Perhaps science fiction writers could use a program like `Mumble` to choose names for their characters, or for mysterious alien planets. More seriously, it is said that the names of some corporations were chosen by similar, but more sophisticated, random word generators.

1. Theory.

To generate random strings, we need an algorithm that can generate random integers. No algorithm can generate truly random integers, but it can generate *pseudo-random* integers that seem random, even though they're not. Python has its own random number generators, but for this project, you must implement your own.

The *Park-Miller Algorithm* (named for its inventors) is a simple way to generate a series of pseudo-random integers. It works like this. Let N_0 be an integer called the *seed*. The seed is the first term of the series, and must be between 1 and $2^{31} - 2$, inclusive. Starting from N_0, later terms N_1, N_2, ... are computed by the following equation.

$$N_{k+1} = (7^5 N_k) \,\%\, (2^{31} - 1)$$

Here 7^5 is 16807, and 2^{31} is 2147483648. The operator `%` returns the remainder after dividing one integer by another. We always get the same series of integers for a given seed. For example, if we start with the seed 101, then we get a series whose first few terms are 1697507, 612712738, 678900201, 695061696, 1738368639, 246698238, 1613847356, and 1214050682.

Some of these integers are large, but we can make them smaller by using the `%` operator again. If N_j is a term from the Park-Miller series, then $N_j \,\%\, M$ gives us an integer between 0 and $M - 1$, inclusive. For example, if we need a pseudo-random integer from 0 to 9, then we write $N_j \,\%\, 10$.

Now that we know how to compute random integers, we can discuss how to choose the letters of an English word at random. As a simplifying assumption, we'll consider only words made from lower case letters, represented in Python as strings `'a'` through `'z'`.

Some letters are unlikely to follow other letters in English words. For example, `'m'` is likely to be followed by `'a'` or `'e'`. It is less likely to be followed by `'n'`, and it is very unlikely to be followed by `'q'`. If we choose letters purely at random, then we risk making strings like `'mnq'` that don't look like English words at all. To produce words that look like English, we must consider how likely it is for one letter to be followed by another. To do that, we'll associate each letter with a Python tuple called a *chooser*. The following example shows how choosers work. The chooser for the letter `'m'` looks like this.

```
( 61649,
  ' ', 13202, 'a',  9255, 'b',  1086, 'c',    36, 'e', 15100,
  'f',   133, 'g',     1, 'i',  4822, 'k',     6, 'l',   162,
  'm',  1383, 'n',   227, 'o',  6641, 'p',  3430, 'r',    81,
  's',  1918, 't',    59, 'u',  1645, 'w',     6, 'y',  2456 )
```

This chooser says that in a large sample of English words, `'m'` was followed by `'a'` 9255 times, and that `'m'` was followed by `'b'` 1086 times, etc. The blank `' '` means that `'m'` appeared at the end of a word 13202 times. Since the letter `'q'` does not appear in the chooser, it means `'m'` was never followed by `'q'` at all. The first element of the chooser, 61649, is always the sum of the letter counts in the rest of the chooser, so that 61649 =

13202 + 9255 + 1086 + 36 ⋯ + 2465.

Now suppose we're generating random words, one character at a time, and that we've just generated a partial word whose last letter is `'m'`. To compute the next letter of the word, we obtain the chooser for `'m'` and pick a number *R* at random from 0 to 61649 − 1. (We showed how to do that when we discussed the Park-Miller algorithm.) Suppose we choose *R* = 38678. Moving left to right, we subtract the chooser's letter counts from *R* until it becomes less than or equal to 0.

R −= 13202 # So *R* becomes 25476.
R −= 9255 # So *R* becomes 16221.
R −= 1086 # So *R* becomes 15135.
R −= 36 # So *R* becomes 15099.
R −= 15100 # So *R* becomes −1.

Note that *R* became less than or equal to 0 after we subtracted the letter count for `'e'`. That means we can stop subtracting letter counts from *R*, and that the letter `'m'` should be followed by `'e'` in the partial word we're generating. Letters with large counts are more likely to make *R* become less than or equal to 0, and are therefore more likely to be chosen.

The file **`choosers.py`** contains Python code that constructs a dictionary called `choosers`. Its keys are lower case letter strings, and its values are the choosers for those letters. For example, `choosers['m']` returns the chooser from the example. Also, `choosers[' ']`, with a blank as a key, returns a chooser that can be used to compute the first letter of a random word.

Now that we know how choosers work, we can describe an algorithm that can generate random words with letter frequencies resembling those of English words. We choose the first letter of the word using `choosers[' ']`, as described in the example. Then we repeatedly choose the next letter of the word in the same way, but using the chooser for its most recently chosen (last) letter. We continue, building the word character by character, until the word is long enough, or until we choose a blank `' '`, at which point the word is complete.

By the way, letter counts are taken from an English translation of Leo Tolstoy's novel ***War and Peace,*** available for free from the **Project Gutenberg** web site. This is one of the longest published novels ever written, with over a thousand pages, and more than a half million words. We wrote a throwaway program (not included here) that read the text and constructed Python code for the dictionary `choosers`.

2. Implementation.

To implement the random word generating program, you must write two Python classes. The first class must be called `Random`, and it must implement the Park-Miller algorithm discussed in section 1. The class `Random` must have the following methods. They must have the same parameters as described here, and they must work as described here.

`__init__(self, seed)`

(5 points.) Initialize an instance of `Random` so it generates the series of random integers that follow `seed`. You may assume that `seed` is in the proper range for the Park-Miller algorithm to work.

`next(self)`

(5 points.) Generate the next random integer in the series, and return it.

`choose(self, max)`

(5 points.) Use `next` to generate a random integer that is greater than or equal to 0, but less than `max`. You may assume that `max` is an integer greater than 0.

Recall that when one method calls another method from the same class, the call must be prefixed by `self`. For example, to have `choose` call `next`, you must write `self.next()`. If you write simply `next()`, then Python tries to call a function `next` that is defined outside the class.

Also, your `Random` class must not compute big numbers like 7^5 and 2^{31} over and over again. You might compute them once and store them in variables. You might also write them as constants, so you don't have to compute them at all. The methods of the class `Random` must be short and simple. Each one can be written in only one or two lines. If you find yourself wanting to write longer methods, then you do not understand how `Random` works, and you should ask for help.

The second class you must write is called `Mumble`. It must have the following methods. They must have the same parameters as described here, and they must work as described here.

`__init__(self, seed, choosers)`

> (5 points.) Initialize an instance of `Mumble`. Make a new random number generator whose seed is `seed` so the other methods can use it. You may assume `seed` is in the proper range for the Park-Miller algorithm to work. Let `make` use the dictionary `choosers` when it makes new random words.

`make(self, max)`

> (10 points.) Using the random number generator and dictionary from `__init__`, make a new random word that has at most `max` letters. Use the algorithm described in section 1. You must raise an exception if `max` ≤ 0.

`test(self, count, min, max)`

> (10 points.) Print `count` random words, one per line, by calling `make`. Each word must have at least `min` letters, and at most `max` letters. You must raise an exeption if $1 \le$ `min` $\le$ `max` is not true.

Do not use inheritance when you write `Random` and `Mumble`. The class `Random` is not a subclass of the class `Mumble` or vice versa, This is because the two classes do different things. Also, the methods `make` and `test` are more complex than those of `Random`, and should probably have 10–20 lines of code each. If you find yourself wanting to write more code than this, then you should ask for help.

3. Example.

Here's an example of how `Mumble` works. We first create an instance of `Mumble`, with random seed 101, using the dictionary `choosers`. Then we call the `test` method. The arguments to `test` ask for 50 random words, whose minimum length is 5 letters, and whose maximum length is 10 letters.

```
mumbler = Mumble(101, choosers)
mumbler.test(50, 5, 10)
```

Here's the resulting output.

```
rolld
ndelousce
hithevoudi
inenct
aprely
tespederst
stcle
omond
cessef
endild
swnpil
bumyere
serotthe
```

```
vesis
candr
adrisater
imenisswin
shour
hivis
mersoreren
tmend
saidd
imithavofo
thithass
whedly
sedant
somas
apongom
blarore
olourt
ananqunty
hasal
healy
pachee
astot
asife
tiscure
stsof
berche
hinotherun
sstor
heldowiff
tillle
ardet
wapit
tenerans
ithioonglr
alule
tathitenqu
lorinies
```

Of course some of these words look more like English than others. Your program may produce different words from these and still be correct.

4. Deliverables.

Unlike the lab assignments, you must work on this project individually, without a partner. Although you can discuss this project in a general way with others, YOUR CODE MUST BE WRITTEN ENTIRELY BY YOURSELF, ALONE! See the syllabus for details. The project is worth 40 points, and you must submit it to Moodle in about three weeks, at **11:55 PM** on **October 19, 2018**.

Also unlike the laboratory assignments, the TA's will determine your grade by reading your Python code in detail, awarding partial credit where possible. As a result, you must submit Python source code for the classes `Random` and `Mumble` together in one `.py` file. If you have questions about how or where to turn in your work, then please contact your lab TA's.