Programming Project 1
CSci 1913: Introduction to Algorithms,
Data Structures, and Program Development
October 9, 2019

0. Introduction.

A *grammar* is a set of rules for generating strings. In this project, you will write a Python program that generates random strings using a grammar. For example, a version of this program generated the following simple English sentences as Python strings.

```
the cat bit the boy .
the cat kissed the dog and the boy chased the boy .
the cat chased the dog and the girl bit the boy but the girl chased the cat .
the girl chased the dog .
the boy kissed the girl and the cat kissed the girl .
```

Perhaps such grammars could write children's books automatically. Seriously, however, more complex grammars have been used to generate random test inputs for programs, as a debugging aid.

1. Theory.

To write this program, you must have a way to generate random integers. You must also know what a rule is, what a grammar is, and how rules and grammars work.

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 sequence of pseudo-random integer terms. It works like this. Let N_0 be an integer called the *seed.* The seed is the first term of the sequence, and must be between 1 and $2^{31} - 2$, inclusive. Starting from the seed, later terms N_1, N_2, ... are produced 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 Python operator `%` returns the remainder after dividing one integer by another. You will always get the same sequence of terms from a given seed. For example, if you start with the seed 101, then you will get a pseudo-random sequence whose first few terms are 1697507, 612712738, 678900201, 695061696, 1738368639, 246698238, 1613847356, and 1214050682. You could use this sequence to test if your random number generator works correctly.

Terms in the sequence may be large, but you can make them smaller by using the `%` operator again. If N is a term from the sequence, and M is an integer greater than 0, then N `%` M gives you an integer between 0 and $M - 1$, inclusive. For example, if you need an integer between 0 and 9, then you would write N `%` 10.

Grammars. The easiest way to explain a grammar is to show an example. This is the grammar that generated the simple sentences about boys, cats, dogs, and girls that appeared in the introduction.

01	*Noun*	→	`'cat'`
02	*Noun*	→	`'boy'`
03	*Noun*	→	`'dog'`
04	*Noun*	→	`'girl'`
05	*Verb*	→	`'bit'`
06	*Verb*	→	`'chased'`
07	*Verb*	→	`'kissed'`
08	*Phrase*	→	`'the'` *Noun Verb* `'the'` *Noun*
09	*Story*	→	*Phrase*
10	*Story*	→	*Phrase* `'and'` *Story*
11	*Story*	→	*Phrase* `'but'` *Story*
12	*Start*	→	*Story* `'.'`

Each line is a *rule,* identified by a number, so this grammar has 12 rules. The names in italics, like *Noun, Verb,* and *Phrase,* are called *nonterminals.* The strings in quotes, like `'girl'`, `'the'` and `'.'`, are called *terminals.* In each rule, the arrow '→' means *may be replaced by*. As a result, rule 01 says that the nonterminal *Noun* may be replaced by the terminal `'cat'`. Similarly, rule 10 says that the nonterminal *Story* may be replaced by the nonterminal *Phrase*, followed by the terminal `'and'`, followed by the nonterminal *Story*.

To generate strings from the grammar, you play a little game. The game always begins with the nonterminal *Start.* The object of the game is to use the rules to get rid of the nonterminals, replacing them by terminals. If you can do that, then the terminals left over at the end are concatenated to produce a string generated by the grammar. For example, you begin with *Start* and use rule 12 to replace it, like this.

Story `'.'`

Now you have a new nonterminal, *Story*, to get rid of. According to rule 10, you can replace *Story* by *Phrase* `'and'` *Story*, so you get this.

Phrase `'and'` *Story* `'.'`

You can then use rule 09 to replace *Story* by *Phrase.*

Phrase `'and'` *Phrase* `'.'`

You use rule 08 to replace the first *Phrase*, so you get this.

`'the'` *Noun Verb* `'the'` *Noun* `'and'` *Phrase* `'.'`

According to rule 01, you can replace the first *Noun* by `'cat'`, and according to rule 02, you can replace the second *Noun* by `'boy'`.

`'the' 'cat'` *Verb* `'the' 'boy' 'and'` *Phrase* `'.'`

And according to rule 06, you can replace *Verb* by `'chased'`.

`'the' 'cat' 'chased' 'the' 'boy' 'and'` *Phrase* `'.'`

Continuing the game, you use rule 08 again to replace *Phrase.*

`'the' 'cat' 'chased' 'the' 'boy' 'and' 'the'` *Noun Verb* `'the'` *Noun* `'.'`

You use rule 02 to replace the first *Noun* by `'boy'`, and use rule 01 to replace the second *Noun* by `'cat'`. Finally, you use rule 07 to replace *Verb* by `'kissed'`.

```
'the' 'cat' 'chased' 'the' 'boy' 'and' 'the' 'boy' 'kissed' 'the' 'cat' '.'
```

At this point, you've eliminated all the nonterminals, so you've won the game. If you concatenate all the strings together, separated by blanks, then you get a string generated by the grammar, like this:

```
'the cat chased the boy and the boy kissed the cat .'
```

By the way, there are many different kinds of grammars, each with different kinds of rules. The grammars used for this project are called *context-free grammars,* in which each rule has a single nonterminal on the left side of the arrow '→'.

2. Implementation.

The grammar game described in the previous section is so simple that a short Python program can play it. You must implement such a program for this project. Your program must use three Python classes, called `Random`, `Rule`, and `Grammar`. They are all separate classes: none of them extends any of the others.

The class `Random`. The first class must be called `Random`, and it must implement the Park-Miller algorithm. 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)`

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

`next(self)`

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

`choose(self, limit)`

> (5 points.) Call `next` to obtain a random integer. Then compute a new integer between 0 and `limit` − 1 from it, as described in the previous section. Return the new integer.

For efficiency, your `Random` class must not compute big numbers like 7^5 and 2^{31} over and over again. Either compute them only once, and store them in variables, or else write them as constants, so you don't have to compute them at all.

All the methods in `Random` must be very short, just one to three lines each. If your methods are longer than that, then you do not understand what you are doing, so you should ask for help.

The class `Rule`. The second class must be called `Rule`, and its instances must represent a rule from a grammar. It must have the following methods. They must have the same parameters as described here, and they must work as described here.

`__init__(self, left, right)`

> (5 points.) Each instance of `Rule` must have three variables. The variable `self.left` must be initialized to the string `left`. The variable `self.right` must be initialized to the tuple of strings `right`. The variable `self.count` must be initialized to the integer 1: it is used by the class `Grammar` to help choose rules.

`__repr__(self)`

> (5 points.) Return a string of the form `'`$C\ L$ `->` $R_1\ R_2 \cdots R_n$`'`, where C is `self.count`, L is `self.left`, and $R_1\ R_2 \cdots R_n$ are the elements of `self.right`. For example, if you create a rule by calling the constructor:

```
Rule('Story', ('Phrase', 'and', 'Story'))
```

then its `__repr__` method must return the string:

```
'1 Story -> Phrase and Story'
```

In other words, the string must look something like a rule.

Python calls the method `__repr__` automatically when it prints an instance of the class `Rule`. This method is not used by the rest of this program, but it is helpful for debugging. If the `__repr__` method was not defined, then Python would print an instance of `Rule` as a string of gibberish, and you would not be able to tell what rule it was. The variable `self.left` is used only by `__repr__`.

The class `Grammar`. The third class must be called `Grammar`, and it must implement a grammar using rules like the ones described in the previous section. 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)`

(5 points.) Initialize an instance of `Grammar`. It must make an instance of the random number generator `Random` that uses `seed`. It must also make a dictionary that stores rules. The dictionary must be initially empty.

`rule(self, left, right)`

(5 points.) Add a new rule to the grammar. Here `left` is a string. It represents the nonterminal on the left side of the rule. Also, `right` is a tuple whose elements are strings. They represent the terminals and nonterminals on the right side of the rule.

Find the value of `left` in the dictionary. If there is no such value, then let the value of `left` in the dictionary be a tuple whose only element is `Rule(left, right)`. However, if there is such a value, then it will be a tuple. Add `Rule(left, right)` to the end of that tuple.

`generate(self)`

(5 points.) Generate a string. If there is a rule with the left side `'Start'` in the dictionary, then call `generating` with the tuple `('Start',)` as its argument, and return the result. If there is no such rule, then raise an exception, because you cannot generate strings without a rule for `'Start'`.

`generating(self, strings)`

(10 points.) This method, along with `select`, does most of the work for this program. The parameter `strings` is a tuple whose elements are strings. The strings represent terminals and nonterminals.

Initialize a variable called `result` to `''`, the empty string. It holds the result that will be returned by this method. Then use a loop to visit each string in `strings`.

If the visited string is not a key in the dictionary, then it is a terminal. Add it to the end of `result`, followed by a blank `' '`.

If the visited string is a key in the dictionary, then it is a nonterminal. Call `select` on the string to obtain a tuple of strings. Then call `generating` recursively on that tuple of strings, to obtain a new string. Add the new string to the end of `result`, without a blank at the end.

Continue in this way until all the strings in `strings` have been visited. At that point, `result` will be a string generated by the grammar. Return `result`.

`select(self, left)`

(10 points.) This method, along with `generating`, does most of the work for this program. It chooses a rule at random whose left side is the string `left`. This happens in several steps:

1. Set the variable `rules` to be the tuple of all rules with `left` on their left sides (from the dictionary). Set the variable `total` to the sum of all the `count` variables in `rules`.

2. Set the variable `index` to an integer chosen at random. It must be greater than or equal to 0, but less than `total`.

3. Visit each rule in `rules`, one at a time. Subtract the rule's `count` variable from `index`. Continue in this way until `index` becomes less than or equal to 0. Set the variable `chosen` to the rule that was being visited when this occurred. (As a result, rules with large `count` variables are more likely to be chosen than rules with small `count` variables.)

4. Add 1 to the `count` variables of all rules in `rules`, other than `chosen`. (This makes it more likely that a rule other than `chosen` will be selected later—giving a wider range of random strings.)

Finally, return the `right` variable of `chosen`, a tuple of strings.

You may want to write helper functions or methods for the class `Grammar`, especially for `Grammar`'s method `select`.

3. Example.

Here is an grammar `G` that you can use to test your program. It has the rules of the example grammar that was described earlier.

```
G = Grammar(101)
G.rule('Noun',   ('cat',))                                # 01
G.rule('Noun',   ('boy',))                                # 02
G.rule('Noun',   ('dog',))                                # 03
G.rule('Noun',   ('girl',))                               # 04
G.rule('Verb',   ('bit',))                                # 05
G.rule('Verb',   ('chased',))                             # 06
G.rule('Verb',   ('kissed',))                             # 07
G.rule('Phrase', ('the', 'Noun', 'Verb', 'the', 'Noun'))  # 08
G.rule('Story',  ('Phrase',))                             # 09
G.rule('Story',  ('Phrase', 'and', 'Story'))              # 10
G.rule('Story',  ('Phrase', 'but', 'Story'))              # 11
G.rule('Start',  ('Story', '.'))                          # 12
```

In this example, `G`'s random number generator is initialized with the seed 101. If you call `G.generate()` five times, then you should get the same five sentences that appear in the introduction. You may want to run additional examples to make sure that your program works correctly.

4. Deliverables.

This project is worth 60 points. It will be due in two weeks, at **11:55 PM** on **Wednesday, October 23, 2019**. You must turn in Python source code for the classes `Random`, `Rule`, and `Grammar`. You must also turn in enough tests and output to convince yourself and the TA's that your program works correctly. Turn in only one file, and append output to the bottom of the file in comments. If you do not know how or where to turn it in, then please contact your lab TA.