Jerry Cain
CS 106AX

Handout #17
October 29th, 2025

Assignment #5—Python Programs

The random sentence generator problem is credited to Mike Cleron (Google), and the problem statement was written by Julie Zelenski. The reassemble problem was also constructed by Julie Zelenski, who herself credits Rich Pattis (UC Irvine) and Owen Astrachan (Duke) for their initial work on a similar assignment.

Now that the first midterm is behind you, you're in a position to officially move on to Python and solve some more computationally interesting problems than those typically solved using JavaScript. This assignment has you complete two separate programs utilizing strings, arrays, dictionaries, and file processing. It's an opportunity to get used to PyCharm, since it'll be your development environment for much of the rest of the quarter.

These two problems are challenging, but as always, we'll be available to help you work through each of them should you want help.

Due: Thursday, November 6th at 5:00 pm

Problem I: Random Sentence Generator

Over the past four decades, computers have revolutionized student life. In addition to providing entertainment and distraction, computers have also facilitated all sorts of student work. One important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, extension requests, etc. with important sounding and somewhat grammatically correct random sequences.

Neglected, that is, until now.

The Random Sentence Generator is a marvelous piece of technology that creates random sentences from a **context-free grammar**. A grammar is a construct describing the various combinations of words that can be used to form valid sentences. There are profoundly useful grammars available to generate extension requests, Star Trek plots, James Bond manuscripts, "Dear John" letters, and more. You can even create your own grammar! Let's show you the value of this practical and wonderful tool:

- Extension Request Tactic #1: Wear down the TA's patience.

  I need an extension because I had to go to an alligator wrestling meet, and then, just when my mojo was getting back on its feet, I just didn't feel like working, and, well I'm a little embarrassed about this, but I had to practice for the Winter Olympics, and on top of that my roommate ate my disk, and right about then well, it's all a haze, and then my dorm burned down, and just then I had tons of midterms and tons of papers, and right about then I lost a lot of money on the four-square semi-finals, oh, and then I had recurring dreams about my notes, and just then I forgot how to write, and right about then my dog ate my dreams, and just then I had to practice for an intramural monster truck meet, oh, and then the bookstore was out of erasers, and on top of that my roommate ate my sense of purpose, and then get this, the programming language was inadequately abstract.

- Extension Request Tactic #2: Plead innocence.

I need an extension because I forgot it would require work and then I didn't know I was in this class.

- Extension Request Tactic #3: Honesty.
  I need an extension because I just didn't feel like working.

What is a grammar?

A grammar is a set of rules for some language, be it English, Java, C++, or something you just invent for fun.☺ If you continue to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce to you a particular kind of grammar called a context-free grammar (**CFG**).

Here is an example of a simple CFG for generating poems:

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```

According to this grammar, two syntactically valid poems are **`"The big yellow flowers sigh warily tonight."`** and **`"The slugs portend like waves tonight."`** Essentially, the strings in brackets (<>) are variables that expand according to the rules in the grammar.

More precisely, each string in brackets is known as a **nonterminal**. A nonterminal is a placeholder that will expand to another sequence of words when generating a poem. In contrast, a **terminal** is a normal word that is not changed to anything else when expanding the grammar. The word *terminal* is supposed to conjure up the image that it's something of a endpoint, and that no further expansion is possible.

A **definition** consists of a nonterminal and a list of possible **productions** (or **expansions**). There will always be at least one and potentially several productions for each nonterminal. A production is just a text string of words, some of which themselves may be non-terminals. A production can even be the empty string, which makes it possible for a nonterminal to evaporate into nothingness.

An entire definition is summarized within a grammar text file as:

```
<verb>                     ⇐ the first line names the nonterminal and is delimited by < and >
3                          ⇐ the second line is always the number of possible expansions
sigh <adverb>              ⇐ the third line is the first possible expansion
portend like <object>.     ⇐    followed by another expansion if there is a second one
die <adverb>               ⇐    followed by another expansion if there is a third one, etc.
                           ⇐ for readability, there's a blank line after each definition, including the last one
```

You always begin random sentence generation with the single non-terminal **`<start>`** as the working string, and iteratively search for the first nonterminal and replace it with any one of its possible expansions (which may and often will include its own nonterminals). Repeat the process over and over until all nonterminals are gone.

```
<start>
The <object> <verb> tonight.                    // expand <start>
The big yellow flowers <verb> tonight.          // expand <object>
The big yellow flowers sigh <adverb> tonight.   // expand <verb>
The big yellow flowers sigh warily tonight.     // expand <adverb>
```

Since we choose productions at random (investigate Python's **`choice`** function), a second generation will almost certainly produce a different sentence.

Your program should prompt the user to open a grammar file using the file chooser (understood to be in the **`grammars`** subdirectory), read in the grammar, and generate three random sentences. Once the three sentences have been generated, your program should just end.

You may assume all grammar files are well formed, and you needn't worry about word wrap as you print super-long sentences.

Here's a sample run from my own solution when run in PyCharm and select **`news.g`** using the file chooser. You're free to replicate my formatting, though I'm less concerned about formatting and more concerned with thoughtful file parsing, data structure design, and clean sentence-generation algorithms.

```
Random sentence #1:
-------------------
The New York Times reports: Scientists working at the golden-ratio-
esque quinoa Collider in in Geneva have uncovered evidence that the
psychic powers of mongoose are not as very extreme as previously
postulated. Thankfully, this leads to the logical conclusion that we
should all be extremely confident that the Bulls will win the NBA
championship.

Random sentence #2:
-------------------
The New York Times reports: Scientists working at the immeasurable
cheese fry Collider in in Geneva have uncovered evidence that the
```

```
psychic powers of mongoose are not as all-encompassingly extreme as
previously postulated. This makes us very unsure about the fate of
the universe.

Random sentence #3:
-------------------
The New York Times reports: MIT researchers have decided without any
evidence that God exists, (specifically Zeus), but He mostly just
sits in his room playing Call of Duty 4. This makes us totally
confident that the multiverse turns out only to exist in laundry
rooms across Middle America.
```

Problem II: Reassemble

You love the course textbook *Introduction to JavaScript Programming* so much that you own 10 copies of it, but ahead of yesterday's midterm your evil roommate took a shredder to all ten copies and, well, **shredded** them. Overcoming your sorrow, you attempt to reassemble the fragments to recreate the originals. Little did you know that solving this problem would prep you for a career in computational biology and international espionage!

You are to write a program to read an input file of text fragments and reassemble them. The fragments were created by duplicating a text document many times over and chopping each copy into pieces. The process of reconstruction finds overlapping sequences and aligns them to match.

This technique is used for genome shotgun sequencing in the Human Genome Project. A DNA strand is cloned many times and randomly cut into chunks of manageable sizes that can be sequenced. The reassembly process aligns those sequenced fragments to reconstruct the original. Nova had a fascinating show on Cracking the Code of Life about the race to decode the human genome. In addition to biology, a number of intelligence-gathering tasks involve reassembly. The carpet weavers piecing together photos in the movie Argo are based in truth, archivists are reassembling Stasi trash, and the team "All Your Shreds are Belong to U.S." (great name!) won the DARPA-sponsored shredder challenge. All of these tasks are based on a reassembly process that finds overlap and merges into a unified whole, just as you will do for this problem.

The program reads a collection of fragments and then processes them in a series of rounds. Each round examines all possible pairings and for each pair considers all possible alignments. This identifies the pair with the longest overlap. Those two fragments are then aligned at the point of maximal overlap and merged into a superstring that has the two original fragments as substrings. Each round decreases the count of fragments by one. The process repeats until it reaches a single result.

Greedy Match and Merge

Consider a collection with the four fragments shown below (extra spaces were inserted between letters for clarity):

```
a l l   i s   w e l l
```

```
e l l   t h a t   e n
h a t   e n d
t   e n d s   w e l l
```

On the first round, the longest overlap is a six-character overlap between the second and third fragments:

```
e l l   t h a t   e n
          h a t   e n d
```

These two fragments merge to the result below. This merged fragment is added to the collection, and the two fragments from which it came are discarded.

```
e l l   t h a t   e n d
```

On the next round, the three remaining fragments are now:

```
a l l   i s   w e l l
e l l   t h a t   e n d
t   e n d s   w e l l
```

The new second and third fragments overlap more than any pair involving the first, and that overlap looks like this:

```
e l l   t h a t   e n d
              t   e n d s   w e l l
```

These fragments are removed and replaced with their merged result:

```
e l l   t h a t   e n d s   w e l l
```

The last round merges the two remaining fragments:

```
a l l   i s   w e l l
                e l l   t h a t   e n d s   w e l l
```

This is the final result:

```
a l l   i s   w e l l   t h a t   e n d s   w e l l
```

Even though it didn’t present itself in this example, a match is also possible when one fragment is completely contained within another. Consider:

```
s   w e l l   t h a
      e l l
```

The second fragment is entirely contained within the first. When these two fragments are merged, the result is the longer of the two strings, and the length of the overlap is the length of the smaller of the two strings.

Ties are broken arbitrarily. If more than one pair have the same max-length overlap, either can be selected as the winner for that round. If a pair has two equally good alignments (e.g. **`abxy`** and **`xyab`** can merge to either **`abxyab`** or **`xyabxy`**), either can be chosen. If a pair has no overlap, they are merged by concatenation, with your choice of order (e.g. **`ab`** and **`xy`** can merge to either **`abxy`** or **`xyab`**). Either option is considered a correct reassembly.

Here is my own solution's output when I load **`preamble_frags`**:

```
We the People of the United States, in Order to form a more perfect
Union, establish Justice, insure domestic Tranquility, provide for the
common defence, promote the general Welfare, and secure the Blessings of
Liberty to ourselves and our Posterity, do ordain and establish this
Constitution for the United States of America.
```

Yes, that's the Preamble to the Constitution, and that's what you'd expect to be synthesized on behalf of a data file called **`preamble_frags`**. Note that the reassembled document is printed, but nothing else is.

A little computer science theory aside. Finding the optimal reassembly is equivalent to the *shortest common superstring*, a classic problem in theoretical computer science. Given a set of strings S, find the shortest string that contains all the strings in S as substrings. The shortest superstring program is NP-hard, which means no efficient, polynomial-time solution is believed to exist. We are not asking your program to find an optimal result--- this difficult problem is much too computationally expensive. Instead, we approximate using a *greedy strategy* that finds a local maximum and optimistically pursues this locally optimal choice in the hopes that it will lead to the globally optimal result. This will find a common superstring, but it is not guaranteed to be the shortest.

Implementation Details

The program is invoked without any additional arguments, which means you use PyCharm's run button to execute your program.

We provide a collection of input files within a folder called reassemble-files, and each file within it is well formed according to the specific format. The content of each valid data file looks like this:

```
        {fragment 1}{fragment 2}{fragment 3} ... {fragment N}
```

Each fragment is a sequence of characters wrapped in curly braces. In between fragments (outside the braces), there can be arbitrary amounts of whitespace (tab, space, newline). The body of a fragment can contain any character whatsoever, save for the **`{`** and **`}`** characters. A fragment is guaranteed to contain at least one character (which might be a space or a newline character.) All input files are small enough that they can be loaded into a single string using the **`read`** and **`splitlines`** methods discussed in lecture.

Reading in data files is tricky, since newline characters aren't used to delimit fragments. That decision was intentional, since newline characters might very well be **inside** a fragment, as you'll see if you open up any of the larger data files like **`gettysburg_frags`** and **`dream_frags`**.

Finally, make it a point to ignore the larger data files until you're convinced your solution is working well with all the smaller ones. Every single one of you can get this program to operate correctly, and we'll work with you to ensure you do. An equally fun part of the reassemble problem is getting the program to work—and work **quickly**—on the larger data files. That, however, is more about optimization and the search for opportunities to avoid very time-consuming operations that you know won't produce anything meaningful. If you're struggling to figure out how to make a fully functional program run much more quickly, then ask me, Avi, and your section leader for some hints and we'll provide a few. We do expect your program to reassemble **`dream_frags`** in under five minutes, but it can be optimized to run much more quickly than that!

Thanks, and good luck with your first Python assignment!