Reflection - MP2

mp2-fib.py

In approaching the first part of MP2, I knew my first step would be to format the contents of my function header according to an if-statement in which I would include three conditionals: one for outputting 0 with an input of zero; one for outputting the integer 1 with an input of n when n equals either 1 or 2; and one for computing the nth fibonacci number when that number is an integer greater than 2, which covers integers in the sequence whose value exceeds 1. The last condition of the if-statement seemed to be the most difficult to compose because of the formatting logic between each of the two variables meant to identify the two integers preceding the third number in the fibonacci sequence. I then knew that I would have to identify these two variables, which, being the first and second number in the series, would have to equal 1. The difficult step seem, however, to be in producing a for-statement in which these two variables could be re-identified according to the input represented by n. To do so, I declared that for x in the range between the third number in the Fibonacci series to the nth input, the number of which I realized would only be cover by the range function when added with 1. Within the for-statement, I then established a third variable c to identify the addition of both a and b, which respectively represent a variable for (n - 1) and for (n - 2) when called upon. In subsequently returning c outside of the for-statement, I then formatted an assert statement for the nth input of 1, 2, 3, as well as 100 — all of which proved to be successful when computed.

mp2-bigrams.py

Generally speaking, I had less difficulty with the second part of MP2, if only because I've previously worked on pulling bigrams from strings and lists of textual content, but I had never done so without the NLTK module, so I nonetheless found writing this code to be helpful and informative in its own right. To begin, I knew I would have to establish a variable for an empty list, identified as *empty_list* in the first line of the function. I then wrote a for-statement in which for *x* in the range of length of the list, *sequence*, append *empty_list* by formatting one at a time the inputted list to recursively pair the first token with its subsequent token until the function runs through the full list — and then to return *empty_list* with these appended changes to show for it. My final step involved composing three assert statements to test, on the one hand, if the program would successfully push back against an empty list and a list with only one token; and, on other hand, if the the program would also format bigrams out of a conventional list of text-based tokens, as it is intended to do.