This week's questions will all center and revolve around creating, manipulating, and using Python's dictionaries data type.

1. The advent of the telegraph machine heralded a new age in communications, but it required a method of encoding our everyday English into a system that could be transmitted over the wires. Electricity can transmit information effectively through switching on or off, but that requires a binary system of conveying information. While ASCII codes could work, for just English use the system devised by Samuel Morse is more compact, and consists of either long or short pulses. The combination of long and short pulses, together with the pauses in between, make up what is commonly known as Morse Code today. The lowercase English letters are each represented by a combination of short or long pulses (commonly denoted *dots* or *dashes*), shown in Figure 1 below:

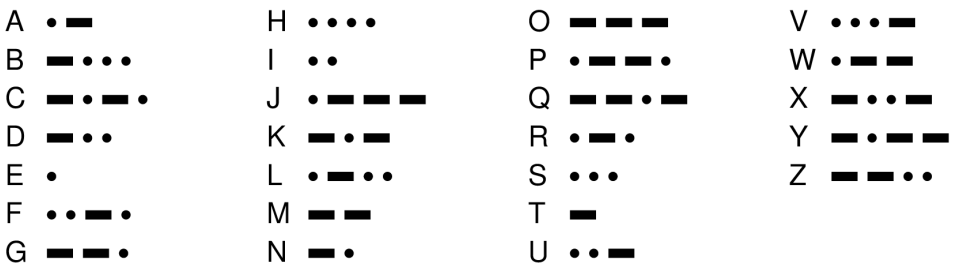

Figure 1: The English letter and their Morse Code equivalents, where a dot (.) represents a short pulse and a dash (-) represents a long pulse.

Because each English letter maps to one combination of dots and dashes, a dictionary is a great way to store this information! One has already been defined in `MoreCodeDictionary.py`, and all you need to do is import the constant `LETTERS_TO_MORSE` for usage. Your task in this problem is to write a program that prompts the user for an english sentence, and then converts that sentence into Morse code and prints it to the screen before prompting for another message. Each sequence of dots and dashes representing a letter should be separated by a space in your final printing, else you would have no way of knowing where one letter ends and the next begins. Any punctuation symbols should be skipped.

A sample run of this program, with content taken from messages between the Titanic and the Carpathia in 1912, is shown below:

```
› python Prob1.py

> SOS TITANIC
... --- ... - .. - .- -. .. -.-.
> WE ARE SINKING FAST
.-- . .- .-. . ... .. -. -.- .. -. --. ..-. .- ... -
>
```

2. **Large Language Models:**

   The biggest story in computing over the last two years has been in generative AI. Whether with ChatGPT, Midjourney, Stable Diffusion, Bard, or others, large language models have taken the computing world by storm.

   Although the software underlying something like ChatGPT is much more complex, the core technology is based on a **large language model** (LLM) that scans some large amount of text (typically called a *corpus*) and then uses the text in that corpus to create sentences in which new words are chosen based on the frequency with which they appear in the context set by the words already generated. A program like ChatGPT picks out some important words from the query, rearranges them so that they resemble the beginning of a plausible answer, and then fleshes out that answer by choosing words that are likely to follow the previously chosen words. Real applications use contexts that include several words as well as semantic information that may come even earlier, but you can get a sense of how such programs operate by building a chatbot-style generator that uses only the immediately preceding word to guide its choice of the next word.

   This process is most easily illustrated by example. Suppose that the text used for the model is the following excerpt from Shakespeare's *Macbeth*:

   > Tomorrow, and tomorrow, and tomorrow
   > Creeps in this petty pace from day to day
   > To the last syllable of recorded time;

   In this simplified implementation, the language model is a dictionary in which each key is a word, and the corresponding value is a list of all the words that, at some point, followed that key. Given the above three lines, the model would thus be the following dictionary:

```
{
    "and": ["tomorrow", "tomorrow"],
    "creeps": ["in"],
    "day": ["to", "to"],
    "from": ["day"],
    "in": ["this"],
    "last": ["syllable"],
    "of": ["recorded"],
    "pace": ["from"],
    "petty": ["pace"],
    "recorded": ["time"],
    "syllable": ["of"],
    "the": ["last"],
    "this": ["petty"],
    "time": [],
    "to": ["day", "the"],
    "tomorrow": ["and", "and", "creeps"]
}
```

This dictionary shows, for example, that the word `"and"` appears twice in the text and in both cases is followed by the word `"tomorrow"`. The word `"tomorrow"` appears three times, twice followed by the word `"and"`, and once followed by the word `"creeps"`.

Your goal in this problem is to write a function called `create_model(text)`, which takes in a string and returns the dictionary representing the language model. Because of the need to parse through the string word by word, you should take advantage of the `TokenScanner` library introduced in class and in Chapter 12 to help you accomplish this task. You should be able to use your function to generate a model similar to the above example.

3. Now that you can create an LLM model, it is time to generate some text from it! Here you should write a function

```
def generate_text(model, start, max_words):
```

which will take in a *model* in the form of a dictionary, a *start*ing word in the form of a string, and a *max_words* in the form of an integer.

Beginning with your starting word, you should look up that word in the model dictionary to see what other words commonly come after in. Then randomly choose one of those words to append to your sentence. Make that word your new current word, and then repeat the process. Your should stop generating text when either you reach the maximum number of desired words, or you reach a word that has no possibilities in the model for what text might come after it.

With the simple MacBeth model, there is not a ton of room for flexibility, and thus given the starting word of `"tomorrow"`, you could either get:

- the original text
- the original text but with "and tomorrow" repeated any number of times, and/or
- the original text but with "to day" repeated any number of times

One sample output might look like:

```
〉python generate_text(macbeth, "tomorrow", 200)

tomorrow and tomorrow and tomorrow creeps in this petty
pace from day to day to day to day to day to the last
syllable of recorded time
```

There are other sources of text included in the repository as well in case you want to try your luck at generating a larger corpus! These are distributed as simple text files, so if you want to use them you'll need to first read them into a string that your `create_model` function can work with.