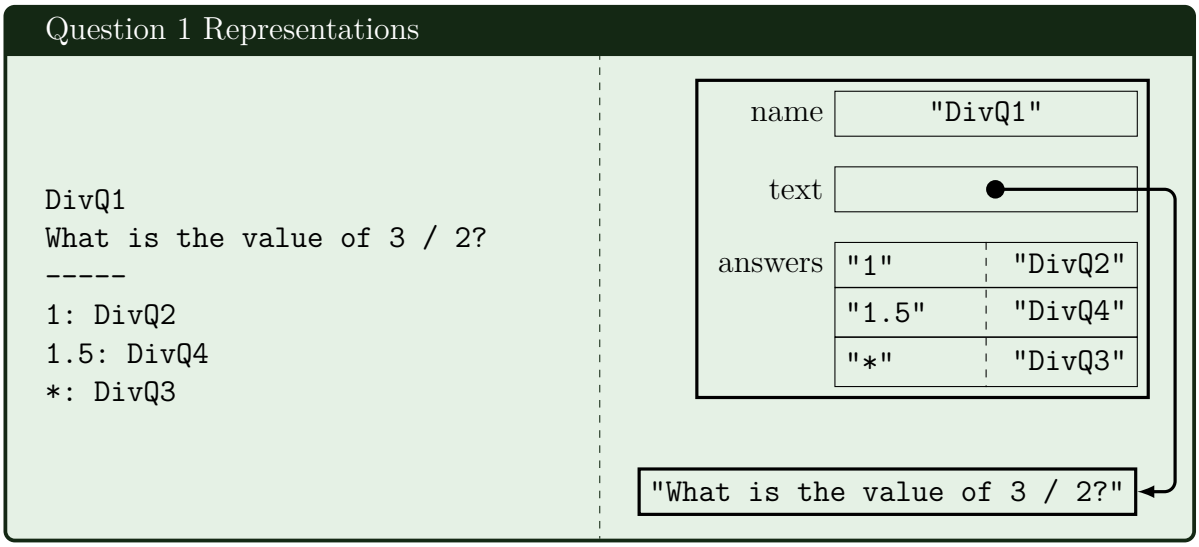
Before you write the code for the Adventure game, it is important to understand the teaching machine program presented in Chapter 12, which has a nearly identical structure. This section includes two problems that reinforce your knowledge of the Teaching Machine and begin the process of converting from the Teaching Machine to the Adventure model.

1. **Diagramming the internal data structure**

   In class and in the text you saw how the internal data structure of the teaching machine program was constructed. Shown below is the human-readable text and corresponding data structure for the first question to serve as a reminder.



   Your goal in this problem is to sketch out the corresponding data structure for the `AdvRoom` class populated with the contents of the first room, shown below:

```
OutsideBuilding
Outside Building
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
-----
WEST: EndOfRoad
UP: EndOfRoad
NORTH: InsideBuilding
IN: InsideBuilding
SOUTH: Valley
DOWN: Valley
```

   Pay particular attention to what data types are being used to store different items, and come up with reasonable names for the different attributes.

2. **Adding messages to the teaching machine**

   As written, the `TeachingMachine.py` program provides no feedback when the user gives an incorrect answer. Here you will look into possible ways of implementing such feedback, which will also be directly applicable to the Adventure project.

   (a) Devise a strategy that would let the course designer specify an optional message along with each response. How would you need to adjust your data-file and corresponding internal representation?

   (b) While there are many strategies that you could have come up with, one that is easy to implement and mimics what Will Crowther did with Adventure is to allow questions in the Teaching Machine to have a "FORCED" answer. If a question has such an answer, then the user is **not** prompted for a response, and instead the user is taken immediately on to the new question after the text is displayed. An example of a snippet of the data file might thus look like:

```
DivQ1
What is the value of 3 / 2?
-----
1: DivMsg
1.5: DivQ4
*: DivQ3

DivMsg
The / operator always produces floats.
-----
FORCED: DivQ2

DivQ2
What is the value of 9 / 3?
-----
3: DivMsg
3.0: DivQ4
*: DivQ3
```

   The `FORCED` entry in the question named `DivMsg` tells the Teaching Machine to continue automatically to `DivQ2` without asking the user for an answer, as could be seen in the following run transcript:

```
〉python TeachingMachine.py

Course name: Python
What is the value of 3 / 2?
> 1
The / operator produces floats.
What is the value of 9 / 3?
> 3
The / operator produces floats.
What is the value of 9 / 3?
>
```

Making the necessary changes in the `TMCourse` class is not difficult, you need only to check if a question has a `FORCED` answer before prompting the user for input, and set the next question accordingly. An example course is provided in the template file for you to test your program against.

3. **Large Language Models:**

   The biggest story in computing over the last year 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.

Included in the starting materials is also a function to use the produced model to generate text, in case you want to see what that might look like and generate. You could also grab a text file of the entire works of Shakespeare here if you'd like to try creating a model from that!