Project2: 20 Questions! as a Tree

This file should be submitted as `proj2.py`. Remember to use the good design principles described in class.
Start by downloading proj2 starter codes. You'll find a skeleton `proj2_skeleton.py` starter file plus file with support functions.
In this assignment, you'll implement a 20 Questions game. (It's called "20 Questions", but there can be as many questions as you like!) Your program will play 20 Questions with a user. It'll be up to the human playing the game to come up with a secret object, and then the program will try to guess what the human is thinking of. If the program guesses **wrong**, then it will ask the human for help learning about the secret object. That means your program will actually get **smarter** as you play more games!
Your program will store all of the questions (and answers) it knows about as a tree. There are two possible types of "nodes" in these trees:

- *Question* (also called "internal") nodes consist of tuples of three things: A question to ask; what to do if the answer to the question is "yes"; and what to do if the answer to the question is "no". The "what to do" part is simply another, smaller, tree.
- *Answer* (also called "leaf") nodes also consist of a 3-tuple (also called a "triple"), but the last parts of the triple are `None`. There is a string representing the answer, followed by two Nones.

A tree is made up of a set of tuples (of tuples (of tuples ...)). For example:

```
("Is it bigger than a breadbox?", ("an elephant", None, None),
("a mouse", None, None))
```

represents a tree with one question and two answers.

```
("Is it bigger than a breadbox?",
    ("Is it gray?",
        ("an elephant", None, None),
        ("a tiger", None, None)),
    ("a mouse", None, None))
```

represents a tree with two questions and three answers.
Getting familiar with the trees
The two trees above are already present in the starter file. Begin by experimenting with them a bit in iPython:

```
In [1]: run proj2.py


In [2]: smallTree
Out[2]:
('Is it bigger than a breadbox?',
 ('an elephant', None, None),
 ('a mouse', None, None))


In [3]: printTree(smallTree)
Is it bigger than a breadbox?
+-Yes: It is an elephant
`-No:  It is a mouse


In [4]: mediumTree
Out[4]:
('Is it bigger than a breadbox?',
 ('Is it gray?', ('an elephant', None, None), ('a tiger', Non
e, None)),
 ('a mouse', None, None))


In [5]: printTree(mediumTree)
Is it bigger than a breadbox?
+-Yes: Is it gray?
| +-Yes: It is an elephant
| `-No:  It is a tiger
`-No:  It is a mouse


In [6]: mediumTree[1]
Out[6]: ('Is it gray?', ('an elephant', None, None), ('a tige
r', None, None))
```

```
In [7]: printTree(mediumTree[1])
Is it gray?
+-Yes: It is an elephant
`-No:  It is a tiger
```

Note several things about the above interaction:

- The way Python prints trees isn't especially easy to read. We've provided a `printTree` function that formats things in a nicer fashion. You'll find `printTree` to be highly useful when you're debugging your program.
- Trees are "nested"; the "Is it gray?" question is a *subtree* has exactly the same "shape" as `smallTree`.
- You can access a subtree with subscripting notation, such as `mediumTree[1]`.
- When a human (or a computer!) plays 20 Questions, they will follow a "path" through the tree. For example, if the object is *not* bigger than a breadbox, there is no point in asking whether it is gray.

What to write

Here are the functions that your program should have. You're welcome to introduce other helper functions as needed.

`simplePlay(tree)` [15 points]

This function accepts a single argument, which is a tree (or a sub-part of a tree), and plays the game once by using the tree to guide its questions. It returns `True` if the computer guessed the answer. Here's an outline:

1. If the tree is a leaf, ask whether the object is the object named in the leaf. Return `True` or `False` appropriately.
2. If the tree is *not* a leaf, ask the question in the tree.
   - If the user answers "yes", call yourself recursively on the subtree that is the second element in the triple.
   - If the user answers "no", recur on the subtree that is the third element in the triple.

Try `simplePlay` out on `smallTree` and `mediumTree`. **Make sure** that it returns `True` or `False` depending on whether the computer guessed correctly. (For example, try it out while thinking of an elephant, and again while thinking of a car.)

Hint 1

`play(tree)` [40 points]

Just like `simplePlay`, this function accepts a single argument, which is a tree, and plays the game once by using the tree to guide its questions. However, instead of returning just `True` or `False`, `play` returns **a new tree** that is the result of playing the game on the original tree and learning from the answers.

- If the computer guessed an object correctly, the "new" tree that is returned will be an identical copy of the original tree.
- BUT, if the computer doesn't guess the object correctly, it will ask the user for the name of the object and a question that will distinguish it (see below for an example). In that case the new tree that is returned will be similar to the old one, but with the new object and an additional question inserted. This means that the computer will learn as you play!

Here's an example of this function in action (user-typed input is shown in blue:

```
In [2]: newTree = play(smallTree)

Is it bigger than a breadbox? yes

Is it an elephant? yes

I got it!

In [3]: newTree

Out[3]: ('Is it bigger than a breadbox?', ('an elephant', Non
e, None), ('a mouse', None, None))

In [4]: newTree = play(tree)

Is it bigger than a breadbox? yes

Is it an elephant? no

Drats!  What was it? a car

What's a question that distinguishes between a car and an elep
hant? Does it have wheels?

And what's the answer for a car? yes

In [5]: newTree

Out[5]: ('Is it bigger than a breadbox?', ('Does it have wheel
s?', ('a car', None, None), ('an elephant', None, None)), ('a
mouse', None, None))
```

The `play` function should be recursive. Again, test it first with `smallTree` and then with `mediumTree`. But this time, also be sure to think of an object that *isn't* in the tree and add it to the tree. If you saved the output of `play` in `newTree`, you can print the result nicely like this:

```
In [6]: printTree(newTree)
Is it bigger than a breadbox?
+-Yes: Does it have wheels?
| +-Yes: It is a car
| `-No:  It is an elephant
`-No:  It is a mouse
```

Hint 2

Hint 3

Hint 4

Saving and restoring trees

As it stands, 20 Questions is a somewhat boring game because it always starts with the same tree. An improvement would be be able to save game trees so that we can share our games and reload them later. To do that, we'll need to create (write) and read *files*. A file is a place on your computer's hard drive or SSD that can hold arbitrary information and keep it pretty much forever; every document (and every program and every picture) on your computer lives inside a file. We keep track of files by giving them names like `proj2.py` , and organize them by collecting them into *folders* (also called "directories").

Opening and closing files

If you want to work with a file in a Python program, you must first *open* it. That's like double-clicking it in VScode—or like opening a notebook. When you open a Python file, you have to tell Python whether you're planning to read it or write it. You do so like this:

```
document = open("myfile.txt", "w")
```

Here, `"myfile.txt"` is the name you want to give to the file, and `"w"` indicates that you intend to write it (as you might guess, `"r"` is used for reading). Unlike opening a notebook, opening a file for writing will throw away anything that's currently there (and will create the file if it doesn't already exist). The `open` function returns a *file handle*, which is a way to refer to the file in the rest of your program. In this case we've used `document` as a mnemonic name for the file handle.

When you're done with the file, you must *close* it so that Python knows you're done with it:

```
document.close()
```

Reading and writing files

If you have opened a file for writing, there are several ways to put data into it. The simplest, which we'll use for this assignment, is our old friend `print`:

```
print("Here is some text", file = document)
```

The additional "file =" argument tells Python that you want the information to be saved in the given file (via its handle) rather than being displayed on the screen.

To read one line of data from a file, you can use `readline`:

```
line = document.readline()
```

As it happens, the line you read from the file will normally contain a "newline" character at the end; you can get rid of that with `strip`:

```
line = line.strip()
```

Here's a full example, at the ipython prompt, of creating and writing a file:

```
In [1]: document = open("test.txt", "w")


In [2]: print("Spish is spiced fish.", file = document)


In [3]: print("Penguins hate spam but they love spish.", file
= document)


In [4]: document.close()
```

At this point, if you open `test.txt` in Visual Studio, you should see two lines of text about the penguin diet.

Now, let's read the file back and print it out—again, just working at the ipython prompt:

```
In [5]: document = open("test.txt", "r")   # Note that we can
re-use "document"


In [6]: while True:
   ...:     line = document.readline()
   ...:     if line == "":
```

```
   ...:         break
   ...:     line = line.strip()
   ...:     print(line)
   ...:
Spish is spiced fish.
Penguins hate spam but they love spish.


In [7]: document.close()
```

Note a few things about reading files:

- Each call to `readline` gives you a *new* line from the file.
- You don't need to know how many lines are in the file, and so you can read and process an arbitrarily large file, one line at a time.
- When there are no more lines, `readline` returns an empty string. (It turns out that we won't need that feature in this assignment, though.)

Take a look at the IOdemo.py file for a few more examples of how to read and write files.

Saving the tree with `saveTree(tree, treeFile)` [30 points]

With that background, it's time to save our tree to a file so that the game will remember what it has learned!

Write a function called `saveTree(tree, treeFile)` that accepts a tree and a the handle of a file that is open for writing, and saves the tree in that file. The examples below demonstrate the format for the file using the two trees that we constructed above.

Why does `saveTree` expect a file handle?

It turns out that `saveTree` will want to be recursive. If `saveTree` opened the file itself, then the recursion would re-open the file and clobber it. So instead `saveTree` will *append* to an already-open file.

An example

First, we open a file in "w" (for "write") mode; then we save the tree to that file, and finally we close the file. The `close` is necessary or your data might not appear, or might be corrupted!

```
In [6]: treeFile = open("tree1.txt", "w")
In [7]: saveTree(smallTree, treeFile)
In [8]: treeFile.close()                           # Absolutely n
ecessary!
```

When we look at the contents of `tree1.txt` (for example, by opening it in Visual Studio), we see this:

```
Internal node
Is it bigger than a breadbox?
Leaf
an elephant
Leaf
a mouse
```

Now, let's try it with the `newTree` example:

```
In [9]: treeFile = open("tree2.txt", "w")          # We can re-us
e treeFile here
In [10]: saveTree(newTree, "tree2.txt")
In [11]: treeFile.close()                          # Absolutely n
ecessary!
```

The `tree2.txt` file looks like:

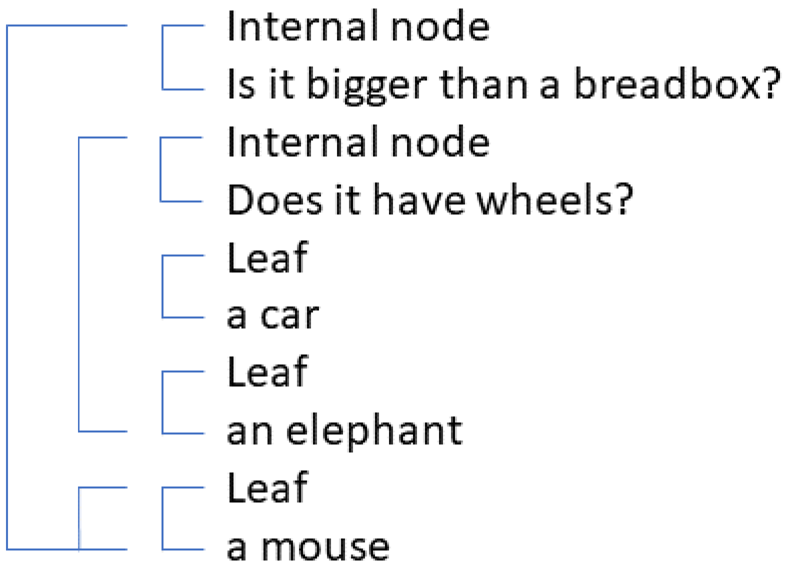
```
Internal node
Is it bigger than a breadbox?
Internal node
Does it have wheels?
Leaf
a car
Leaf
an elephant
Leaf
a mouse
```

Notice that the saved file is in the following format:

- An internal node (a question) starts with the line `Internal Node`.
  - After the `Internal Node` line is a line with the question itself.
  - Following the question, there are two groups of lines that represent the left and right children of the internal node.

    - Note that these groups of lines can be arbitrarily large!
    - As a result, figuring out what the original tree looked like is a bit tricky for humans—but it's exactly what the computer needs!.
- A leaf node (the answer) starts with the line `Leaf` and is followed by a line with the answer.

Here's an annotated version of the file above, showing how the various components are grouped:



treenodes.png

Here's the secret to making `saveTree` (and you) happy: it's recursive!
Hint 5

Optional but highly recommended: Loading the tree with `loadTree(treeFile)` (10 extra-credit points)

This next part of the problem is optional, but *very highly recommended* if you have time. It's only about 10-15 lines of code, it will help you achieve a higher level of mastery of this material, and you'll get 10 more points for it! **BUT**, if you're running out of time and don't do this, we'll understand! In that case, move on to the last part of the problem, which is the `main()` function below.
The `loadTree(treeFile)` function accepts a file that has already been opened for reading. It uses `readline()` to read one line at a time from the file, builds the tree described by that file, and returns it so that we can play that tree! Here are two examples, following on the running examples above.

```
In [12]: treeFile = open("tree1.txt", "r")   # open up the file for reading
In [13]: tree1 = loadTree(treeFile)         # read the file into a list
In [14]: treeFile.close()                    # always close your file afterwards
In [15]: tree1
Out[15]: (('Is it bigger than a breadbox?', ('an elephant', None, None), ('a mouse', None, None))
In [16]: treeFile = open("tree2.txt", "r")   # open up the file for reading
In [17]: tree2 = loadTree(treeFile)         # read the file into a list
In [18]: treeFile.close()                    # always close your file afterwards
In [19]: tree2
Out[19]: ('Is it bigger than a breadbox?', ('Does it have wheels?', ('a car', None, None), ('an elephant', None, None)), ('a mouse', None, None))
```

Again, `loadTree` is recursive.

Hint 6

Hint 7

Putting it together with the `main()` function [15 points]

Finally, write a `main()` function that has the following behavior:

- Prints a welcome message
- Asks the user if they would like to load a tree from a file (optional, but bonus points as indicated above)
- If the user didn't want to load from a file, the initial tree should be: `("Is it bigger than a breadbox?", ("an elephant", None, None), ("a mouse", None, None))`
- Plays the game

- Asks the user if they would like to play again, in which case we play again with the new tree
- When the user is done playing, asks the user if they would like to save the file, in which case the user is queried for a file name and the file is saved

Here's an example, with the user input in blue:

```
Welcome to 20 Questions!
Would you like to load a tree from a file? yes
What's the name of the file? tree2.txt
Is it bigger than a breadbox? yes
Does it have wheels? no
Is it an elephant? yes
I got it!
Would you like to play again? yes
Is it bigger than a breadbox? yes
Does it have wheels? no
Is it an elephant? no
Drats!  What was it? a hippo
What's a question that distinguishes between a hippo and an el
ephant? Does it have tusks?
And what's the answer for a hippo? no
Would you like to play again? yes
Is it bigger than a breadbox? yes
Does it have wheels? no
Does it have tusks? no
Is it a hippo? yes
I got it!
Would you like to play again? no
Would you like to save this tree for later? yes
Please enter a file name: tree3.txt
Thank you!  The file has been saved.
Bye!
```

Finishing and Submitting

First, starting from the initial tree with just one question ("Is it bigger than a breadbox") and two animals ("an elephant" and "a mouse"), play several times to build up a game that has a more interesting game tree.

Then, save that tree to a file called `proj2_tree.txt`.

If you implemented the "load" feature, you can play the game using your classmates' trees! Have fun!

Then, submit your code, proj2.py and your `proj2_tree.txt` file in the submission system.

Proj2 hint 1

For this function and for the next step, you may find it useful to write some helper functions:

`isLeaf(tree)`
Returns `True` if the tree is a leaf and `False` if it is an internal node. (You might prefer to name this function `isAnswer`, or even write its inverse, isQuestion.)

`yes(prompt)`
Uses the `prompt` to ask the user a yes/no question, and returns `True` if the answer is yes, `False` if it is no. Our version insists on a proper answer (including convenient and fun options like "y", "yup", and "sure").

`playLeaf(tree)`Plays a leaf node by suggesting an answer and deciding whether it was correct. (A reasonable alternate name could be `playAnswer`, or you're of course welcome to invent your own name.)

We chose not to have a separate `playInternal(tree)` function, but that would be a perfectly sensible option.

Proj2 Hint 2

The secret to happiness here is to realize that when `play` calls itself recursively, it must use the result of the recursion to *construct* the tree that it will return. That's true regardless of what the recursive call returns and regardless of whether the computer got the right answer!

Proj2 Hint 3

We strongly recommend that you write a separate `playLeaf` (or `playAnswer`) function that handles the "hard" case that arises when the computer gets things wrong. `playLeaf` should query the user for the necessary information and return an appropriate subtree.

Proj2 Hint 4

Realize that `playLeaf` might return a leaf (if the computer got it right) or a brand-new internal (question) node. The new question node must have *two* answer nodes embedded inside it.

Proj2 Hint 5

It's important to not overthink this one. `saveTree` needs to decide whether it's producing an internal node or a leaf. If it's a leaf, just write the two lines and you're done. If it's an internal node, write the "Internal Node" line and the question, and then make *two* recursive calls. Trust that the recursive calls will do the right thing!

Proj2 Hint 6

`loadTree` is somewhat symmetrical with `saveTree`. Start by reading a line to find out whether you're creating a leaf or an internal node. If you're making a leaf, you just need to return a triple such as `('an elephant', None, None)`. For a leaf, that *is* the tree you're building!

Proj2 Hint 7

If you're making an internal node, you need to read the question and then return a triple such as ('Is it bigger than a breadbox?', *tree1*, *tree2*), where *tree1* and *tree2* come from recursive calls to `loadTree`. Again, don't overthink things; trust the recursion!