Back to the Hangman Game

By putting together all you've learned so far, you can create the Hangman game presented at the beginning of the chapter. This program is much longer than anything you've seen, but don't be intimidated by its size. The code isn't much more complex than that of the other projects you've worked through. The biggest part of the program is just my modest ASCII art, the eight versions of the stick figured being hanged. The real meat of the program is not much more than a screenful of code.

Setting Up the Program

First things first. As always, I started with opening comments, explaining the program. Next, I imported the `random` module. I'll need the module to pick a random word from a sequence.

```
# Hangman Game
#
# The classic game of Hangman. The computer picks a random word
# and the player tries to guess it, one letter at a time. If the player
# can't guess the word in time, the little stick figure gets hanged.
#
# Michael Dawson

# imports
import random
```

Creating Constants

Though there are several screenfuls of code in this next section, I only create three constants in all that programming. First, I created the biggest tuple you've seen. It's really just a sequence of eight elements, but each element is a triple-quoted string that spans 12 lines.

Each string is a representation of the gallows where the stick figure is being hanged. Each subsequent string shows a more complete figure. Each time the player guesses incorrectly, the next string is displayed. By the eighth entry, the image is complete and the figure is a goner. If this final string is displayed, the player has lost and the game is over. I assigned this tuple to `HANGMAN`, a variable name in all caps, because I'll be using it as a constant.

```
# constants
HANGMAN = (
"""
  ----

 |     |
 |
 |
 |
 |
 |
 |
 |
----------

""",
"""

  ----
 |     |
 |     O
 |
 |
 |
 |
 |
 |
----------

""",
"""

  ----
 |     |
 |     O
 |    -+-
 |
 |
 |
 |
 |
----------

""",
"""

  ----
 |     |
 |     O
 |   /-+-
 |
 |
 |
 |
 |
----------

""",
"""

  ----
 |     |
 |     O
 |   /-+-/
 |
 |
 |
 |
 |
----------

""",
"""

  ----

 |     |
 |     O
 |   /-+/
 |     |
 |
 |
 |
 |
----------

""",
"""

  ----
 |     |
 |     O
 |   /-+/
 |     |
 |     |
 |    |
 |    |
 |
----------

""",
"""

  ----
 |     |
 |     O
 |   /-+/
 |     |
 |     |
 |    | |
 |    | |
 |
----------

""")
```

Next, I created a constant to represent the maximum number of wrong guesses a player can make before the game is over:

```
MAX_WRONG = len(HANGMAN) - 1
```

The maximum number of wrong guesses is one less than the length of `HANGMAN`. This is because the first image, of the empty gallows, is displayed even before the player makes a first guess. So although there are eight images in `HANGMAN`, the player only gets seven wrong guesses before the game is over.

Finally, I created a tuple containing all of the possible words that the computer can pick from for the player to guess. Feel free to modify the program and make up your own list.

```
WORDS = ("OVERUSED", "CLAM", "GUAM", "PUCK", "TAFFETA")
```

Initializing the Variables

Next, I initialized the variables. I used the `random.choice()` function to pick a random word from the list of possible words. I assigned this secret word to the variable `word`.

```
# initialize variables
word = random.choice(WORDS)   # the word to be guessed
```

I created another string, `so_far`, to represent what the player has guessed so far in the game. The string starts out as just a series of dashes, one for each letter in the word. When the player correctly guesses a letter, the dashes in the positions of that letter are replaced with the letter itself.

```
so_far = "-" * len(word)   # one dash for each letter in word to be guessed
```

I created `wrong` and assigned it the number 0. `wrong` keeps track of the number of wrong guesses the player makes.

```
wrong = 0                  # number of wrong guesses player has made
```

I created an empty list, `used`, to contain all the letters the player has guessed:

```
used = []                  # letters already guessed
```

Creating the Main Loop

I created a loop that continues until either the player has guessed too many wrong letters or the player has guessed all the letters in the word:

```
print "Welcome to Hangman. Good luck!"

while (wrong < MAX_WRONG) and (so_far != word):
    print HANGMAN[wrong]
    print "\nYou've used the following letters:\n", used
    print "\nSo far, the word is:\n", so_far
```

Notice that I put both conditions in parentheses. When using just one logical operator (like I did here), using parentheses has no real effect. The computer doesn't care. But I think that the parentheses help separate the conditions and make the program easier for humans to read, so I used them.

Next, I print the current stick figure, based on the number of wrong guesses the player has made. The more wrong guesses the player has made, the closer the stick figure is to being done in. After that, I display the list of letters that the player has used in this game. And then I show what the partially guessed word looks like so far.

Getting the Player's Guess

I get the player's guess and convert it to uppercase so that it can be found in the secret word (which is in all caps). After that, I make sure that the player hasn't already used this letter. If the player has already guessed this letter, then I make the player enter a new character until the player enters one he or she hasn't used yet. Once the player enters a valid guess, I convert the guess to uppercase and add it to the list of used letters.

```
    guess = raw_input("\n\nEnter your guess: ")
    guess = guess.upper()

    while (guess in used):
        print "You've already guessed the letter:", guess
        guess = raw_input("Enter your guess: ")
        guess = guess.upper()

    used.append(guess)
```

Checking the Guess

Next, I check to see if the guess is in the secret word. If it is, I let the player know. Then I go about creating a new version of `so_far` to include this new letter in all the places where the letter is in the secret word.

```
    if (guess in word):

       print "\nYes!", guess, "is in the word!"


       # create a new so_far to include guess
       new = ""
       for i in range(len(word)):
           if guess == word[i]:
               new += guess
```

```
            else:
                new += so_far[i]
        so_far = new
```

If the player's guess isn't in the word, then I let the player know and increase the number of wrong guesses by one.

```
    else:
        print "\nSorry,", guess, "isn't in the word."
        wrong += 1
```

Ending the Game

At this point, the game is over. If the number of wrong guesses has reached the maximum, the player has lost. In that case, I print the final image of the stick figure. Otherwise, I congratulate the player. In either case, I let the player know what the secret word was.

```
if (wrong == MAX_WRONG):
    print HANGMAN[wrong]
    print "\nYou've been hanged!"
else:
    print "\nYou guessed it!"

print "\nThe word was", word

raw_input("\n\nPress the enter key to exit.")
```

Summary

In this chapter, you learned all about lists and dictionaries, two new types. You learned that lists are mutable sequences. You saw how to add, delete, sort, and even reverse those elements. But even with all that lists offer, you learned that there are some cases where the less flexible tuple is actually the better (or required) choice. You also learned about shared references that can occur with mutable types and saw how to avoid them when necessary. You saw how to create and use nested sequences to work with even more interesting information, like a high score list. You learned how to create and modify dictionaries that let you work with pairs of data, too.

Challenges

1. Create a program that prints a list of words in random order. The program should print all the words and not repeat any.
2. Write a Character Creator program for a role-playing game. The player should be given a pool of 30 points to spend on four attributes: Strength, Health, Wisdom, and Dexterity. The player should be able to spend points from the pool on any attribute and should also be able to take points from an attribute and put them back into the pool.
3. Write a Who's Your Daddy? program that lets the user enter the name of a male and produces the name of his father. (You can use celebrities, fictional characters, or even historical figures for fun.) Allow the user to add, replace, and delete son-father pairs. The program should also allow the user to get a list of all sons, or fathers, or son-father pairs.
4. Improve the Who's Your Daddy program by adding a choice that lets the user enter a name and get back a grandfather. Your program should still only use one dictionary of son-father pairs. Make sure to include several generations in your dictionary so that a match can be found.