|  |  |
| --- | --- |
|  | Chapter 9 |
| Hangman |

Topics Covered In This Chapter:

* Multi-line Strings
* Methods
* Lists
* The append() and reverse() list methods
* The lower(), upper(), split(), startswith(), and endswith() string methods
* The in and not in operators
* The range() and list() functions
* del statements
* for loops
* elif statements

This chapter’s game introduces many new concepts, but don’t worry. You’ll experiment with these programming concepts in the interactive shell first. You’ll learn about methods, which are functions attached to values. You’ll also learn about a new type of loop called a for loop and a new data type called a list. Once you understand these concepts, it will be much easier to program Hangman.

## Source Code of Hangman

This chapter’s game is a bit longer than the previous games, but much of it is the ASCII art for the hangman pictures. Enter the following into the file editor and save it as hangman.py.

hangman.py

1. import random

2. HANGMANPICS = ['''

3.

4. +---+

5. | |

6. |

7. |

8. |

9. |

10. =========''', '''

11.

12. +---+

13. | |

14. O |

15. |

16. |

17. |

18. =========''', '''

19.

20. +---+

21. | |

22. O |

23. | |

24. |

25. |

26. =========''', '''

27.

28. +---+

29. | |

30. O |

31. /| |

32. |

33. |

34. =========''', '''

35.

36. +---+

37. | |

38. O |

39. /|\ |

40. |

41. |

42. =========''', '''

43.

44. +---+

45. | |

46. O |

47. /|\ |

48. / |

49. |

50. =========''', '''

51.

52. +---+

53. | |

54. O |

55. /|\ |

56. / \ |

57. |

58. =========''']

59. words = 'ant baboon badger bat bear beaver camel cat clam cobra cougar coyote crow deer dog donkey duck eagle ferret fox frog goat goose hawk lion lizard llama mole monkey moose mouse mule newt otter owl panda parrot pigeon python rabbit ram rat raven rhino salmon seal shark sheep skunk sloth snake spider stork swan tiger toad trout turkey turtle weasel whale wolf wombat zebra'.split()

60.

61. def getRandomWord(wordList):

62. # This function returns a random string from the passed list of strings.

63. wordIndex = random.randint(0, len(wordList) - 1)

64. return wordList[wordIndex]

65.

66. def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):

67. print(HANGMANPICS[len(missedLetters)])

68. print()

69.

70. print('Missed letters:', end=' ')

71. for letter in missedLetters:

72. print(letter, end=' ')

73. print()

74.

75. blanks = '\_' \* len(secretWord)

76.

77. for i in range(len(secretWord)): # replace blanks with correctly guessed letters

78. if secretWord[i] in correctLetters:

79. blanks = blanks[:i] + secretWord[i] + blanks[i+1:]

80.

81. for letter in blanks: # show the secret word with spaces in between each letter

82. print(letter, end=' ')

83. print()

84.

85. def getGuess(alreadyGuessed):

86. # Returns the letter the player entered. This function makes sure the player entered a single letter, and not something else.

87. while True:

88. print('Guess a letter.')

89. guess = input()

90. guess = guess.lower()

91. if len(guess) != 1:

92. print('Please enter a single letter.')

93. elif guess in alreadyGuessed:

94. print('You have already guessed that letter. Choose again.')

95. elif guess not in 'abcdefghijklmnopqrstuvwxyz':

96. print('Please enter a LETTER.')

97. else:

98. return guess

99.

100. def playAgain():

101. # This function returns True if the player wants to play again, otherwise it returns False.

102. print('Do you want to play again? (yes or no)')

103. return input().lower().startswith('y')

104.

105.

106. print('H A N G M A N')

107. missedLetters = ''

108. correctLetters = ''

109. secretWord = getRandomWord(words)

110. gameIsDone = False

111.

112. while True:

113. displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)

114.

115. # Let the player type in a letter.

116. guess = getGuess(missedLetters + correctLetters)

117.

118. if guess in secretWord:

119. correctLetters = correctLetters + guess

120.

121. # Check if the player has won

122. foundAllLetters = True

123. for i in range(len(secretWord)):

124. if secretWord[i] not in correctLetters:

125. foundAllLetters = False

126. break

127. if foundAllLetters:

128. print('Yes! The secret word is "' + secretWord + '"! You have won!')

129. gameIsDone = True

130. else:

131. missedLetters = missedLetters + guess

132.

133. # Check if player has guessed too many times and lost

134. if len(missedLetters) == len(HANGMANPICS) - 1:

135. displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)

136. print('You have run out of guesses!\nAfter ' + str(len(missedLetters)) + ' missed guesses and ' + str(len(correctLetters)) + ' correct guesses, the word was "' + secretWord + '"')

137. gameIsDone = True

138.

139. # Ask the player if they want to play again (but only if the game is done).

140. if gameIsDone:

141. if playAgain():

142. missedLetters = ''

143. correctLetters = ''

144. gameIsDone = False

145. secretWord = getRandomWord(words)

146. else:

147. break

How the Code Works

1. import random

The Hangman program randomly selected a secret word from a list of secret words. The random module will provide this ability, so line 1 imports it.

2. HANGMANPICS = ['''

3.

4. +---+

5. | |

6. |

7. |

8. |

9. |

10. =========''', '''

...the rest of the code is too big to show here...

This one assignment statement stretches over lines 2 to 58 in the source code. To help you understand what this code means, let’s learn about multi-line strings.

## Multi-line Strings

So far all strings have been on one line and had one quote character at the start and end. However, if you use three quotes at the start and end then the string can go across several lines:

>>> fizz = '''Dear Alice,

I will return to Carol's house at the end of the month. I will see you then.

Your friend,

Bob'''

>>> print(fizz)

Dear Alice,

I will return to Carol's house at the end of the month. I will see you then.

Your friend,

Bob

These are multi-line strings. In a multi-line string, the newline characters are included as part of the string. You don’t have to use the \n escape character, or escape quotes as long as you don’t use three of them together. This makes the code easier to read for large amounts of text.

## Constant Variables

The HANGMANPICS variable’s name is in all capitals. This is the programming convention for constant variables. Constants are variables meant to have values that never changes from their first assignment statement. Although you can change the value in HANGMANPICS just like any other variable, the all-caps name reminds you to not do so. Since the HANGMANPICS variable never needs to change, it’s marked as a constant.

Like all conventions, you don’t have to follow it. But following this convention makes it easier for other programmers to read your code. They’ll know that HANGMANPICS will always have the value it was assigned on line 2.

## Lists

A list value can contain several other values inside it. Try entering this into the interactive shell:.

>>> spam = ['Life', 'The Universe', 'Everything', 42]

>>> spam

['Life', 'The Universe', 'Everything', 42]

This list value in spam contains four values. When typing the list value into your code, it begins with a [ square bracket and ends with a ] square bracket. This is like how strings begin and end with a quote character.

Commas separate the individual values inside of a list. These values are also called items.

Indexes

Try entering animals = ['aardvark', 'anteater', 'antelope', 'albert'] into the interactive shell to store a list in the variable animals. The square brackets are also used to access an item inside a list. Try entering animals[0], animals[1], animals[2], and animals[3] into the interactive shell to see how they evaluate:

>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']

>>> animals[0]

'aardvark'

>>> animals[1]

'anteater'

>>> animals[2]

'antelope'

>>> animals[3]

'albert'

The number between the square brackets is the index. In Python, the index of the first item in a list is 0. The second item is at index 1, the third item is at index 2, and so on. Because the indexes begin at 0, not 1, we say that Python lists are zero-indexed.

Lists are good for storing several values without using a variable for each one. Otherwise, the code would look like this:

>>> animals1 = 'aardvark'

>>> animals2 = 'anteater'

>>> animals3 = 'antelope'

>>> animals4 = 'albert'

This code would be hard to manage if you have hundreds or thousands of strings. But a list can easily contain any number of values. Using the square brackets, you can treat items in the list just like any other value. Try entering animals[0] + animals[2] into the interactive shell:

>>> animals[0] + animals[2]

'aardvarkantelope'

The evaluation looks like this:

animals[0] + animals[2]

▼

'aardvark' + animals[2]

▼

'aardvark' + 'antelope'

▼

'aardvarkantelope'

IndexError

If you try accessing an index that is too large, you’ll get an IndexError that will crash your program. Try entering the following into the interactive shell:

>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']

>>> animals[9999]

Traceback (most recent call last):

File "", line 1, in

animals[9999]

IndexError: list index out of range

Changing the Values of List Items with Index Assignment

You can also use the square brackets to change the value of an item in a list. Try entering the following into the interactive shell:

>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']

>>> animals[1] = 'ANTEATER'

>>> animals

['aardvark', 'ANTEATER', 'antelope', 'albert']

The new 'ANTEATER' string overwrites the second item in the animals list. So animals[1] will evaluate to the list’s second item in expressions, but you can also use it on the left side of an assignment statement to assign a value as the list’s second item.

List Concatenation

You can join lists into one list with the + operator, just like you can join strings. Joining lists with the + operator is list concatenation. Try entering the following into the interactive shell:

>>> [1, 2, 3, 4] + ['apples', 'oranges'] + ['Alice', 'Bob']

[1, 2, 3, 4, 'apples', 'oranges', 'Alice', 'Bob']

['apples'] + ['oranges'] will evaluate to ['apples', 'oranges']. But ['apples'] + 'oranges' will result in an error. You cannot add a list value and string value instead of two list values. If you want to add non-list values to a list, use the append() method (described later).

The in Operator

The in operator can tell you if a value is in a list or not. Expressions that use the in operator return a Boolean value: True if the value is in the list and False if it isn’t. Try entering the following into the interactive shell:

>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']

>>> 'antelope' in animals

True

The expression 'antelope' in animals returns True because the string 'antelope' is one of the values in the animals list. It is located at index 2.

But if you type the expression 'ant' in animals, this will return False because the string 'ant' doesn’t exist in the list.

>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']

>>> 'antelope' in animals

True

>>> 'ant' in animals

False

The in operator also works for strings. It checks if one string exists in another. Try entering the following into the interactive shell:

>>> 'hello' in 'Alice said hello to Bob.'

True

Deleting Items from Lists with del Statements

A del statement will delete an item at a certain index from a list. Try entering the following into the interactive shell:

>>> spam = [2, 4, 6, 8, 10]

>>> del spam[1]

>>> spam

[2, 6, 8, 10]

Notice that when you deleted the item at index 1, the item that used to be at index 2 became the new value at index 1. The item that used to be at index 3 moved to be the new value at index 2. Everything above the deleted item moved down one index.

You can type del spam[1] again and again to keep deleting items from the list:

>>> spam = [2, 4, 6, 8, 10]

>>> del spam[1]

>>> spam

[2, 6, 8, 10]

>>> del spam[1]

>>> spam

[2, 8, 10]

>>> del spam[1]

>>> spam

[2, 10]

The del statement is a statement, not a function or an operator. It doesn’t have parentheses or evaluate to a return value.

Lists of Lists

Lists can contain other values, including other lists. Let’s say you have a list of groceries, a list of chores, and a list of your favorite pies. You can put all three lists into another list. Try entering the following into the interactive shell:

>>> groceries = ['eggs', 'milk', 'soup', 'apples', 'bread']

>>> chores = ['clean', 'mow the lawn', 'go grocery shopping']

>>> favoritePies = ['apple', 'frumbleberry']

>>> listOfLists = [groceries, chores, favoritePies]

>>> listOfLists

[['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow the lawn', 'go grocery shopping'], ['apple', 'frumbleberry']]

To get an item inside the list of lists, you would use two sets of square brackets like this: listOfLists[1][2] which would evaluate to the string 'go grocery shopping'.

This is because listOfLists[1][2] evaluates to ['clean', 'mow the lawn', 'go grocery shopping'][2]. That finally evaluates to 'go grocery shopping':

listOfLists[1][2]

▼

[['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow the lawn', 'go grocery shopping'], ['apple', 'frumbleberry']][1][2]

▼

['clean', 'mow the lawn', 'go grocery shopping'][2]

▼

'go grocery shopping'

Figure 9-1 is another example of a list of lists, along with some of the indexes that point to the items. The arrows point to indexes of the inner lists themselves. The image is also flipped on its side to make it easier to read.

## Methods

Methods are functions attached to a value. For example, all string values have a lower() method, which returns a copy of the string value in lowercase. You can call it like 'Hello'.lower(), which returns 'hello'. You cannot call lower() by itself and you do not pass a string argument to lower() (as in lower('Hello')). You must attach the method call to a specific string value using a period. The next section describes string methods further.

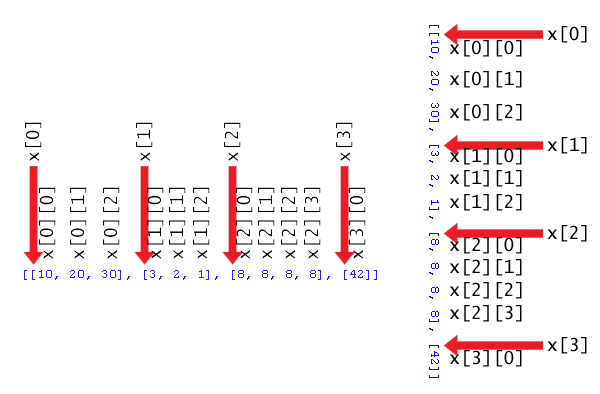


Figure 9-1: The indexes of a list of lists.

## The lower() and upper() String Methods

Try entering 'Hello world!'.lower() into the interactive shell to see an example of this method:

>>> 'Hello world!'.lower()

'hello world!'

There is also an upper() method for strings, which returns a string with all the characters in uppercase. Try entering 'Hello world!'.upper() into the interactive shell:

>>> 'Hello world!'.upper()

'HELLO WORLD! '

Because the upper() method returns a string, you can call a method on that string also. Try entering 'Hello world!'.upper().lower() into the interactive shell:

>>> 'Hello world!'.upper().lower()

'hello world!'

'Hello world!'.upper() evaluates to the string 'HELLO WORLD!', and then string's lower() method is called. This returns the string 'hello world!', which is the final value in the evaluation.

'Hello world!'.upper().lower()

▼

'HELLO WORLD!'.lower()

▼

'hello world!'

The order is important. 'Hello world!'.lower().upper() isn’t the same as 'Hello world!'.upper().lower():

>>> 'Hello world!'.lower().upper()

'HELLO WORLD!'

That evaluation looks like this:

'Hello world!'.lower().upper()

▼

'hello world!'.upper()

▼

'HELLO WORLD!'

If a string is stored in a variable, you can call a string method on that variable. Look at this example:

>>> spam = 'Hello world!'

>>> spam.upper()

'HELLO WORLD!'

This does not change the value in spam. The spam variable will still contain 'Hello world!'.

Note that the integer and float data types don’t have any methods.

## The reverse() and append() List Methods

The list data type also has methods. The reverse() method will reverse the order of the items in the list. Try entering spam = [1, 2, 3, 4, 5, 6, 'meow', 'woof'], and then spam.reverse() to reverse the list. Then enter spam to view the contents of the variable.

>>> spam = [1, 2, 3, 4, 5, 6, 'meow', 'woof']

>>> spam.reverse()

>>> spam

['woof', 'meow', 6, 5, 4, 3, 2, 1]

The most common list method you’ll use is append(). This method will add the value you pass as an argument to the end of the list. Try entering the following into the interactive shell:

>>> eggs = []

>>> eggs.append('hovercraft')

>>> eggs

['hovercraft']

>>> eggs.append('eels')

>>> eggs

['hovercraft', 'eels']

>>> eggs.append(42)

>>> eggs

['hovercraft', 'eels', 42]

These methods do change the lists they are called on. They don’t return a new list. We say that these methods change the list in-place.

## The split() List Method

Line 59 is a long line of code, but it is really just a simple assignment statement. This line also uses the split() method, which is a method for the string data type like the lower() and upper() methods.

59. words = 'ant baboon badger bat bear beaver camel cat clam cobra cougar coyote crow deer dog donkey duck eagle ferret fox frog goat goose hawk lion lizard llama mole monkey moose mouse mule newt otter owl panda parrot pigeon python rabbit ram rat raven rhino salmon seal shark sheep skunk sloth snake spider stork swan tiger toad trout turkey turtle weasel whale wolf wombat zebra'.split()

This assignment statement has just one long string, full of words separated by spaces. And at the end of the string is a split() method call. The split() method evaluates to a list with each word in the string as a single list item. The “split” occurs wherever a space occurs in the string.

It is easier to type the code using split(). If you created it as a list to begin with, you would have to type: ['ant', 'baboon', 'badger',... and so on, with quotes and commas for every word.

For example, try entering the following into the interactive shell:

>>> sentence = input()

My very energetic mother just served us nachos.

>>> sentence.split()

['My', 'very', 'energetic', 'mother', 'just', 'served', 'us', 'nachos.']

The result is a list of nine strings, one string for each of the words in the original string. The spaces are not included in any of the items in the list.

You can also add your own words to the string on line 59, or remove any you don’t want to be in the game. Just make sure that spaces separate the words.

How the Code Works

Line 61 defines the getRandomWord() function. A list argument will be passed for its wordList parameter. This function will return a single secret word from the list in wordList.

61. def getRandomWord(wordList):

62. # This function returns a random string from the passed list of strings.

63. wordIndex = random.randint(0, len(wordList) - 1)

64. return wordList[wordIndex]

Line 63 stores a random index for this list in the wordIndex variable. You do this by calling randint() with two arguments. The first argument is 0 (for the first possible index) and the second argument is the value that the expression len(wordList) - 1 evaluates to (for the last possible index in a wordList).

List indexes start at 0, not 1. If you have a list of three items, the index of the first item is 0, the index of the second item is 1, and the index of the third item is 2. The length of this list is 3, but the index 3 would be after the last index. This is why line 63 subtracts 1 from the length. The code on line 63 will work no matter what the size of wordList is. Now you can add or remove strings to wordList if you like.

The wordIndex variable will be set to a random index for the list passed as the wordlist parameter. Line 64 will return the element in wordList at the integer index stored in wordIndex.

Let’s pretend ['apple', 'orange', grape'] was passed as the argument to getRandomWord() and that randint(0, 2) returned the integer 2. That would mean that line 64 would evaluate to return wordList[2], and then evaluate to return 'grape'. This is how the getRandomWord() returns a random string in the wordList list.

So the input to getRandomWord() is a list of strings, and the return value output is a randomly selected string from that list. This will be useful for the Hangman game to select a secret word for the player to guess.

Displaying the Board to the Player

Next, you need a function to print the hangman board on the screen. It will also display how many letters the player has correctly (and incorrectly) guessed.

66. def displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord):

67. print(HANGMANPICS[len(missedLetters)])

68. print()

This code defines a new function named displayBoard(). This function has four parameters:

* HANGMANPICS - A list of multi-line strings that will display the board as ASCII art. (The global HANGMANPICS variable will be passed for this parameter.)
* missedLetters - A string of the letters the player has guessed that are not in the secret word.
* correctLetters - A string of the letters the player has guessed that are in the secret word.
* secretWord – A string of the secret word that the player is trying to guess.

The first print() function call will display the board. HANGMANPICS will be a list of strings for each possible board. HANGMANPICS[0] shows an empty gallows, HANGMANPICS[1] shows the head (when the player misses one letter), HANGMANPICS[2] shows a head and body (when the player misses two letters), and so on until HANGMANPICS[6] which shows the full hangman.

The number of letters in missedLetters will reflect how many incorrect guesses the player has made. Call len(missedLetters) to find out this number. So, if missedLetters is 'aetr' then len('aetr') will return 4. Printing HANGMANPICS[4] will display the appropriate hangman board for 4 misses. This is what HANGMANPICS[len(missedLetters)] on line 67 evaluates to.

70. print('Missed letters:', end=' ')

71. for letter in missedLetters:

72. print(letter, end=' ')

73. print()

Line 70 prints the string 'Missed letters:' with a space character at the end instead of a newline. Remember that the keyword argument end=' ' uses only one = sign (like =), not two (like ==).

Line 71 is a new type of loop, called a for loop. A for loop often uses the range() function. Both are explained in the next two sections.

## The range() and list() Functions

When called with one argument, range() will return a range object of integers from 0 up to (but not including) the argument. This range object can be converted to the more familiar list data type with the list() function. Try entering list(range(10)) into the interactive shell:

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list('Hello')

['H', 'e', 'l', 'l', 'o']

The list() function is similar to the str() or int() functions. It takes the value it is passed and returns a list. It’s easy to generate huge lists with the range() function. Try entering in list(range(10000)) into the interactive shell:

>>> list(range(10000))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...

*...skipped for brevity...*

...9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]

The list is so huge, that it won’t even all fit onto the screen. But you can store the list into a variable:

>>> spam = list(range(10000))

If you pass two integer arguments to range(), the range object it returns is from the first integer argument up to (but not including) the second integer argument. Try entering list(range(10, 20)) into the interactive shell:

>>> list(range(10, 20))

[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

The range() function is often used in for loops, which are much like the while loops you’ve already seen.

## for Loops

The for loop is useful for looping over a list of values. This is different from the while loop, which loops as long as a certain condition is True. A for statement begins with the for keyword, followed by a new variable name, followed by the in keyword, followed by an iterable value, and ending with a colon.

An iterable is a value of the list, range, or string data types. There are also other data types that are considered iterables which will be introduced later.

Each time the program execution iterates through the loop the new variable in the for statement is assigned the value of the next item in the list.

>>> for i in range(5):

... print('i is set to ' + str(i))

...

i is set to 0

i is set to 1

i is set to 2

i is set to 3

i is set to 4

The range object returned by range(5) is equivalent to the list [0, 1, 2, 3, 4] in a for statement. The first time the execution goes through the code in the for-block, the variable i will be set to 0. On the next iteration, i will be set to 1, and so on.

The for statement automatically converts the range object returned by range() into a list, so there’s no need for list(range(5)) in the for statement. Just use range(5).

Lists and strings are also iterable data types. You can use them in for statements. Try entering the following into the interactive shell:

>>> for thing in ['cats', 'pasta', 'programming', 'spam']:

... print('I really like ' + thing)

...

I really like cats

I really like pasta

I really like programming

I really like spam

>>> for i in 'Hello':

... print(i)

...

H

e

l

l

o

A while Loop Equivalent of a for Loop

The for loop is similar to the while loop, but when you only need to iterate over items in a list, using a for loop is much less code to type. This is a while loop that acts the same as the previous for loop by adding extra code:

>>> iterableVal = ['cats', 'pasta', 'programming', 'spam']

>>> index = 0

>>> while (index < len(iterableVal)):

... thing = iterableVal[index]

... print('I really like ' + thing)

... index = index + 1

...

I really like cats

I really like pasta

I really like programming

I really like spam

But using the for statement automatically does this extra code and makes programming easier since you have less to type.

The rest of the displayBoard() function displays the missed letters and creates the string of the secret word with all the not yet guessed letters as blanks.

70. print('Missed letters:', end=' ')

71. for letter in missedLetters:

72. print(letter, end=' ')

73. print()

The for loop on line 71 will iterate over each character in the missedLetters string and print them on the screen. Remember that the end=' ' will replace the newline character that is printed after the string with a single space character.

For example, if missedLetters was 'ajtw' this for loop would display a j t w.

## Slicing

List slicing creates a new list value with a subset of another list’s items. In code, specify two indexes (the beginning and end) with a colon in the square brackets after a list. For example, try entering the following into the interactive shell:

>>> spam = ['apples', 'bananas', 'carrots', 'dates']

>>> spam[1:3]

['bananas', 'carrots']

The expression spam[1:3] evaluates to a list with items from index 1 up to (but not including) index 3 in spam.

If you leave out the first index, Python will automatically think you want index 0 for the first index:

>>> spam = ['apples', 'bananas', 'carrots', 'dates']

>>> spam[:2]

['apples', 'bananas']

If you leave out the second index, Python will automatically think you want the rest of the list:

>>> spam = ['apples', 'bananas', 'carrots', 'dates']

>>> spam[2:]

['carrots', 'dates']

Slicing is a simple way to get a subset of the items in a list. You use slices with strings in the same way you use them with lists. Each character in the string is like an item in the list. Try entering the following into the interactive shell:

>>> myName = 'Zophie the Fat Cat'

>>> myName[4:12]

'ie the F'

>>> myName[:10]

'Zophie the'

>>> myName[7:]

'the Fat Cat'

The next part of the code in Hangman uses slicing.

Displaying the Secret Word with Blanks

Now you want code to print the secret word, but with blank lines for the letters that have not been guessed. You can use the \_ character (called the underscore character) for this. First create a string with nothing but one underscore for each letter in the secret word. Then replace the blanks for each letter in correctLetters.

So if the secret word was 'otter' then the blanked out string would be '\_\_\_\_\_' (five \_ characters). If correctLetters was the string 'rt' you would change the string to '\_tt\_r'. Line 75 to 79 is the code that does that.

75. blanks = '\_' \* len(secretWord)

Line 75 creates the blanks variable full of \_ underscores using string replication. Remember that the \* operator can also be used on a string and an integer, so the expression '\_' \* 5 evaluates to '\_\_\_\_\_'. This will make sure that blanks has the same number of underscores as secretWord has letters.

77. for i in range(len(secretWord)): # replace blanks with correctly guessed letters

78. if secretWord[i] in correctLetters:

79. blanks = blanks[:i] + secretWord[i] + blanks[i+1:]

Line 77 has a for loop to go through each letter in secretWord and replace the underscore with the actual letter if it exists in correctLetters.

For example, pretend the value of secretWord is 'otter' and the value in correctLetters is 'tr'. You would want the string '\_tt\_r' displayed to the player. Let’s figure out how to create this string.

Line 77’s len(secretWord) call would return 5. The range(len(secretWord)) call becomes range(5), which makes the for loop iterate over 0, 1, 2, 3, and 4.

Because the value of i will take on each value in [0, 1, 2, 3, 4], the code in the for loop is the same as this:

if secretWord[0] in correctLetters:

blanks = blanks[:0] + secretWord[0] + blanks[1:]

if secretWord[1] in correctLetters:

blanks = blanks[:1] + secretWord[1] + blanks[2:]

if secretWord[2] in correctLetters:

blanks = blanks[:2] + secretWord[2] + blanks[3:]

if secretWord[3] in correctLetters:

blanks = blanks[:3] + secretWord[3] + blanks[4:]

if secretWord[4] in correctLetters:

blanks = blanks[:4] + secretWord[4] + blanks[5:]

If you are confused as to what the value of something like secretWord[0] or blanks[3:] is, then look at Figure 9-2. It shows the value of the secretWord and blanks variables, and the index for each letter in the string.

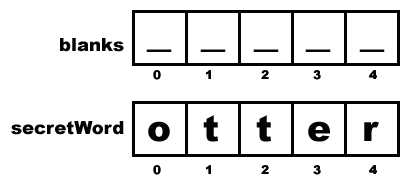


Figure 9-2: The indexes of the blanks and secretWord strings.

If you replace the list slices and the list indexes with the values that they represent, the loop code would be the same as this:

if 'o' in 'tr': # False

blanks = '' + 'o' + '\_\_\_\_' # This line is skipped.

if 't' in 'tr': # True

blanks = '\_' + 't' + '\_\_\_' # This line is executed.

if 't' in 'tr': # True

blanks = '\_t' + 't' + '\_\_' # This line is executed.

if 'e' in 'tr': # False

blanks = '\_tt' + 'e' + '\_' # This line is skipped.

if 'r' in 'tr': # True

blanks = '\_tt\_' + 'r' + '' # This line is executed.

# blanks now has the value '\_tt\_r'

The above code examples all do the same thing when secretWord is 'otter' and correctLetters is 'tr'. The next few lines of code print the new value of blanks with spaces between each letter.

81. for letter in blanks: # show the secret word with spaces in between each letter

82. print(letter, end=' ')

83. print()

Get the Player’s Guess

The getGuess() function will be called so that the player can enter a letter to guess. The function returns the letter the player guessed as a string. Further, getGuess() will make sure that the player types a valid letter before returning from the function.

85. def getGuess(alreadyGuessed):

86. # Returns the letter the player entered. This function makes sure the player entered a single letter, and not something else.

A string of the letters the player has guessed is passed as the argument for the alreadyGuessed parameter. Then the getGuess() function asks the player to guess a single letter. This single letter will be getGuess()’s return value.

87. while True:

88. print('Guess a letter.')

89. guess = input()

90. guess = guess.lower()

Line 87’s while loop will keep asking the player for a letter until they enter text that is:

1. A single letter.
2. A letter they have not guessed previously.

The condition for the while loop is simply the Boolean value True. That means the only way execution will ever leave this loop is by executing a break statement (which leaves the loop) or a return statement (which leaves not just the loop but the entire function).

The code inside the loop asks the player to enter a letter, which is stored in the variable guess. If the player entered a capitalized letter, it will be overwritten with a to lowercase letter on line 90.

## elif (“Else If”) Statements

The next part of the Hangman program uses elif statements. You can think of elif “else if” statements as saying “If this is true, do this. Or else if this next condition is true, do that. Or else if none of them are true, do this last thing.”

Take a look at the following code:

if catName == 'Fuzzball':

print('Your cat is fuzzy.')

elif catName == 'Spots':

print('Your cat is spotted.')

else:

print('Your cat is not fuzzy or spotted.')

If the catName variable is equal to the string 'Fuzzball', then the if statement’s condition is True and the if-block tells the user that their cat is fuzzy. However, if this condition is False, then Python tries the elif (“else if”) statement’s condition next. If catName is 'Spots', then the 'Your cat is spotted.' string is printed to the screen. If both are False, then the code tells the user their cat isn’t fuzzy or spotted.

You can have as many elif statements as you want:

if catName == 'Fuzzball':

print('Your cat is fuzzy.')

elif catName == 'Spots':

print('Your cat is spotted.')

elif catName == 'Chubs':

print('Your cat is chubby.')

elif catName == 'Puff':

print('Your cat is puffy.')

else:

print('Your cat is neither fuzzy nor spotted nor chubby nor puffy.')

When one of the elif conditions is True, its code is executed and then execution jumps to the first line past the else-block. So one and only one of the blocks in the if-elif-else statements will be executed. You can also leave off the else-block if you don’t need one, and just have if-elif statements.

Making Sure the Player Entered a Valid Guess

91. if len(guess) != 1:

92. print('Please enter a single letter.')

93. elif guess in alreadyGuessed:

94. print('You have already guessed that letter. Choose again.')

95. elif guess not in 'abcdefghijklmnopqrstuvwxyz':

96. print('Please enter a LETTER.')

97. else:

98. return guess

The guess variable contains player’s letter guess. The program needs to make sure they typed in a valid guess: one and only one lowercase letter. If they didn't, the execution should loop back and ask them for a letter again.

Line 91’s condition checks if guess is not one character long. Line 93’s condition checks if guess already exists inside the alreadyGuessed variable. Line 95’s condition checks if guess is not a lowercase letter.

If all of these conditions are False, then the else statement’s block executes and getGuess() returns the value in guess on line 98.

Remember, only one of the blocks in if-elif-else statements will be executed.

Asking the Player to Play Again

100. def playAgain():

101. # This function returns True if the player wants to play again, otherwise it returns False.

102. print('Do you want to play again? (yes or no)')

103. return input().lower().startswith('y')

The playAgain() function has just a print() function call and a return statement. The return statement has an expression that looks complicated, but you can break it down. Here’s a step by step look at how Python evaluates this expression if the user types in YES.

input().lower().startswith('y')

▼

'YES'.lower().startswith('y')

▼

'yes'.startswith('y')

▼

True

The point of the playAgain() function is to let the player type in yes or no to tell the program if they want to play another round of Hangman. The player should be able to type YES, yes, Y, or anything else that begins with a “Y” in order to mean “yes”. If the player types in YES, then the return value of input() is the string 'YES'. And 'YES'.lower() returns the lowercase version of the attached string. So the return value of 'YES'.lower() is 'yes'.

But there’s the second method call, startswith('y'). This function returns True if the associated string begins with the string parameter between the parentheses, and False if it doesn’t. The return value of 'yes'.startswith('y') is True.

Now you have evaluated this expression! What it does is let the player type in a response, lowercases the response, checks if it begins with the letter 'y', then returns True if it does and False if it doesn’t.

On a side note, there’s also an endswith(someString) string method that will return True if the string ends with the string in someString and False if it doesn’t. endswith() is sort of like the opposite of startswith().

Review of the Hangman Functions

That’s all the functions we are creating for this game! Let’s review them:

* getRandomWord(wordList) will take a list of strings passed to it, and return one string from it. That is how a word is chosen for the player to guess.
* displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord) will show the current state of the board, including how much of the secret word the player has guessed so far and the wrong letters the player has guessed. This function needs four parameters passed to work correctly. HANGMANPICS is a list of strings that hold the ASCII art for each possible hangman board. correctLetters and missedLetters are strings made up of the letters that the player has guessed that are in and not in the secret word, respectively. And secretWord is the secret word the player is trying to guess. This function has no return value.
* getGuess(alreadyGuessed) takes a string of letters the player has already guessed and will keep asking the player for a letter that isn’t in alreadyGuessed.) This function returns the string of the valid letter the player guessed.
* playAgain() is a function that asks if the player wants to play another round of Hangman. This function returns True if the player does and False if the player doesn’t.

After the functions is the code for the main part of the program at line 106. Everything previous was just function definitions and a large assignment statement for HANGMANPICS.

Setting Up the Variables

106. print('H A N G M A N')

107. missedLetters = ''

108. correctLetters = ''

109. secretWord = getRandomWord(words)

110. gameIsDone = False

Line 106 is the first print() call that executes when the game is run. It displays the title of the game. Next is assigning blank strings for missedLetters and correctLetters since the player hasn’t guessed any missed or correct letters yet.

The getRandomWord(words) call will evaluate to a randomly selects word from the words list.

Line 110 sets gameIsDone to False. The code will set gameIsDone to True when it wants to signal that the game is over and should ask the player if they want to play again.

Displaying the Board to the Player

112. while True:

113. displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)

The while loop’s condition is always True, which means it will loop forever until a break statement is encountered. (This happens later on line 147.)

Line 113 calls the displayBoard() function, passing it the list of hangman ASCII art pictures and the three variables set on lines 107, 108, and 109. Based on how many letters the player has correctly guessed and missed, this function displays the appropriate hangman board to the player.

Letting the Player Enter Their Guess

115. # Let the player type in a letter.

116. guess = getGuess(missedLetters + correctLetters)

The getGuess() function needs all the letters in missedLetters and correctLetters combined, so line 116 concatenates the strings in these variables and passes the result as the argument. This argument is needed by getGuess() because the function has to check if the player types in a letter that they have already guessed.

Checking if the Letter is in the Secret Word

118. if guess in secretWord:

119. correctLetters = correctLetters + guess

If the guess string exists in secretWord, then concatenate guess to the end of the correctLetters string. This string will be the new value of correctLetters.

Checking if the Player has Won

121. # Check if the player has won

122. foundAllLetters = True

123. for i in range(len(secretWord)):

124. if secretWord[i] not in correctLetters:

125. foundAllLetters = False

126. break

How can the program know if the player has guessed every letter in the secret word? Well, correctLetters has each letter that the player correctly guessed and secretWord is the secret word itself. But you can’t just check if correctLetters == secretWord because consider this case: if secretWord was the string 'otter' and correctLetters was the string 'orte', then correctLetters == secretWord would be False even though the player has guessed each letter in the secret word.

The only way you can be sure the player won is to iterate over each letter in secretWord and see if it exists in correctLetters. If, and only if, every letter in secretWord exists in correctLetters will the player have won.

If you find a letter in secretWord that doesn’t exist in correctLetters, you know that the player has **not** guessed all the letters. The new variable foundAllLetters is set to True on line 122 before the loop begins. The loop starts out assuming that all the letters in the secret word have been found. But the loop’s code on line 125 will change foundAllLetters to False the first time it finds a letter in secretWord that isn’t in correctLetters.

127. if foundAllLetters:

128. print('Yes! The secret word is "' + secretWord + '"! You have won!')

129. gameIsDone = True

If all letters in the secret word have been found, the player is told they have won and gameIsDone is set to True.

When the Player Guesses Incorrectly

130. else:

131. missedLetters = missedLetters + guess

This is the start of the else-block. Remember, the code in this block will execute if the condition was False. But which condition? To find out, point your finger at the start of the else keyword and move it straight up like in Figure 9-3. You’ll see that the else keyword's indentation is the same as the if keyword's indentation on line 118.

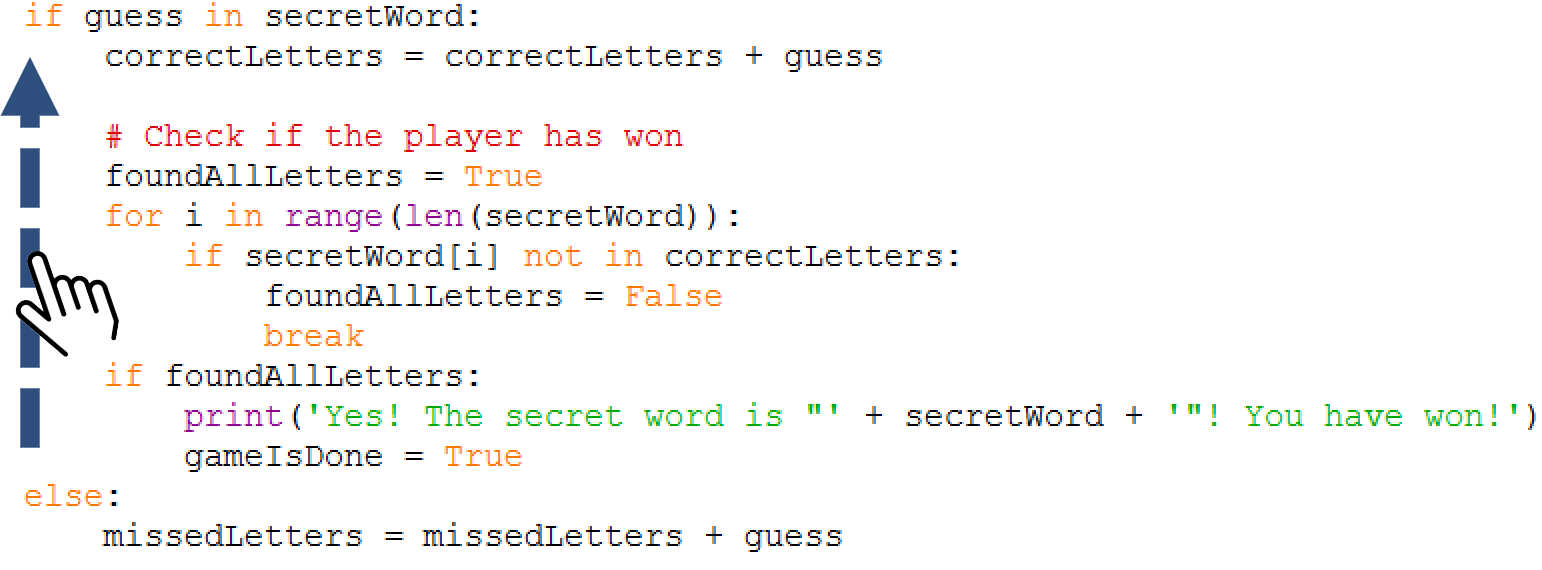


Figure 9-3: The else statement is matched with the if statement at the same indentation.

So if the condition on line 118 (guess in secretWord) was False, then the execution moves into this else-block.

Wrongly guessed letters are concatenated to the missedLetters string on line 131. This is like what line 119 did for letters the player guessed correctly.

133. # Check if player has guessed too many times and lost

134. if len(missedLetters) == len(HANGMANPICS) - 1:

135. displayBoard(HANGMANPICS, missedLetters, correctLetters, secretWord)

136. print('You have run out of guesses!\nAfter ' + str(len(missedLetters)) + ' missed guesses and ' + str(len(correctLetters)) + ' correct guesses, the word was "' + secretWord + '"')

137. gameIsDone = True

Each time the player guesses wrong, the code concatenates the wrong letter to the string in missedLetters. So the length of missedLetters (or, in code, len(missedLetters)) is also the number of wrong guesses.

The HANGMANPICS list has 7 ASCII art strings. So when len(missedLetters) equals 6, you know the player has lost because the hangman picture will be finished. Remember, HANGMANPICS[0] is the first item in the list, and HANGMANPICS[6] is the last one.

So, when the length of the missedLetters string is equal to len(HANGMANPICS) - 1 (that is, 6), the player has run out of guesses. Line 136 prints the secret word and line 137 sets the gameIsDone variable is set to True.

139. # Ask the player if they want to play again (but only if the game is done).

140. if gameIsDone:

141. if playAgain():

142. missedLetters = ''

143. correctLetters = ''

144. gameIsDone = False

145. secretWord = getRandomWord(words)

If the player won or lost after guessing their letter, the game should ask the player if they want to play again. The playAgain() function handles getting a yes or no from the player, so it is called on line 141.

If the player does want to play again, the values in missedLetters and correctLetters must be reset to blank strings, gameIsDone to False, and a new secret word stored in secretWord. This way when the execution loops back to the beginning of the while loop on line 112, the board will be back to a fresh game.

146. else:

147. break

If the player did not type in something that began with “y” when asked if they wanted to play again, then line 141’s condition would be False and the else-block executes. The break statement causes the execution to jump to the first instruction after the loop. But because there are no more instructions after the loop, the program terminates.

Summary

This has been a long chapter, and you’ve been introduced to several new concepts. But Hangman has been our most advanced game yet. As your games get more and more complex, it’ll be a good idea to sketch out a flow chart on paper of what happens in your program.

Lists are values that can contain other values. Methods are functions specific to a data type. Lists have append() and reverse() methods. Strings have lower(), upper(), split(), startswith(), and endswith() methods. You’ll learn about many more data types and methods in the rest of this book.

The for loop is a loop that iterates over the items in a list, unlike a while loop which iterates as long as a condition is True. The elif statement lets you add an “or else if” clause to the middle of your if-else statements. The del statement can delete variables or items inside lists.