COM661 Full Stack Strategies and Development

BE01. Python Text, Files and Data Structures

Aims

- To introduce simple string processing via the Python interpreter
- To demonstrate splitting and joining strings
- To demonstrate case manipulation of character strings
- To introduce the basic options for opening text files
- To demonstrate interating across each line of a text file
- To demonstrate writing values to a text file
- To introduce complex Python data structures
- To measuse the time performance of Python code
- To present the serialization of complex Python data structures

Table of Contents

1.1 Text Processing

Python is an object-oriented programming language that is suitable for a wide range of software development tasks. In recent years, it has become popular for web applications development, due to its strong support for integration with other languages and tools, and its extensive libraries.

Python is a simple language to learn, with relatively few keywords and data types – but it is extremely powerful and flexible. Python code is also among the most readable of all high-level languages.

Note: If you are installing Python on your own computer, please use the latest version (currently **3.13**) to ensure compatibility with the material in this section. There is still a large legacy community that uses the older Python 2.7, but official support for this will soon be withdrawn. Version 3.x has some important differences from earlier versions.

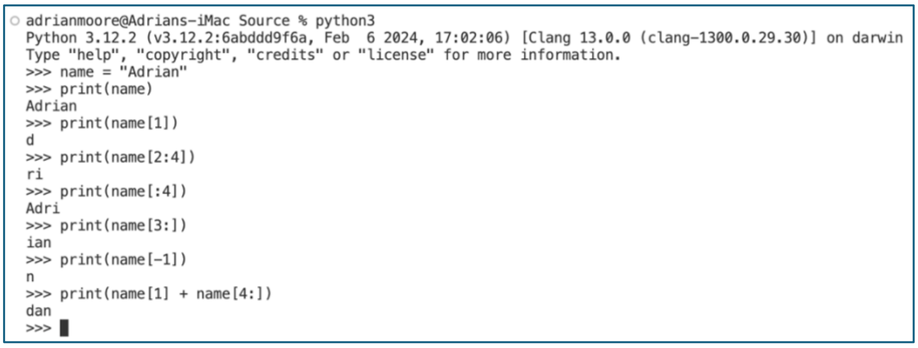
String values in Python are enclosed in either single or double quote characters and are concatenated by the **+** operator. Python also supports a powerful **slice** operator that can be applied to strings. The slice operator is invoked by **`string[start:end]`** which extracts all characters from position **`start`** to position **`end-1`**. If we omit either **`start`** or **`end`**, then the slice is assumed to be either from the beginning or to the end of the string. A negative value denotes character positions counting from the end of the string so that **`string[-1]`** refers to the last character, **`string[-2]`** refers to the next-to-last character, and so on.

Do it now! Try the string slice manipulations presented in Figure 1.1 by entering them directly to the Python interpreter. Experiment with other examples of your own until you are comfortable with the operation of the string slice operator.

Note: To run Python code from the command prompt, you need to open a terminal window and launch the Python interpreter via the command **`python`** (or **`python3`** depending on your installation). A terminal window is most easily obtained by opening it from the Visual Studio Code menu.

Alternatively, on a Windows machine, open a terminal window (command prompt) by clicking the Windows “Start” button, entering “cmd” in the search box, and clicking “cmd.exe’ from the list of options provided.

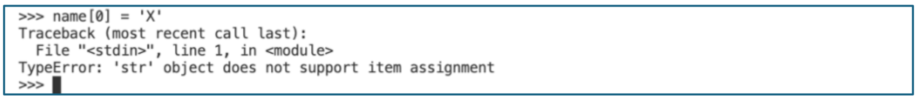
On a MacOS machine, you open the application called **Terminal** that will be pre-installed on the computer. You can find this by opening the Launchpad on the Dock (**Terminal** is most often in a group of applications called “Other”), or by searching for it in the Finder.



```
○ adrianmoore@Adrians-iMac Source % python3
  Python 3.12.2 (v3.12.2:6abddd9f6a, Feb  6 2024, 17:02:06) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
  Type "help", "copyright", "credits" or "license" for more information.
  >>> name = "Adrian"
  >>> print(name)
  Adrian
  >>> print(name[1])
  d
  >>> print(name[2:4])
  ri
  >>> print(name[:4])
  Adri
  >>> print(name[3:])
  ian
  >>> print(name[-1])
  n
  >>> print(name[1] + name[4:])
  dan
  >>> 
```

Figure 1.1 String Slice Operator

One very important concept in Python strings is that they are immutable – i.e., once created, the value of a string cannot be modified. We can prove this by attempting to change the first character of our **`name`** variable, as shown in Figure 1.2 below.



```
>>> name[0] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> 
```

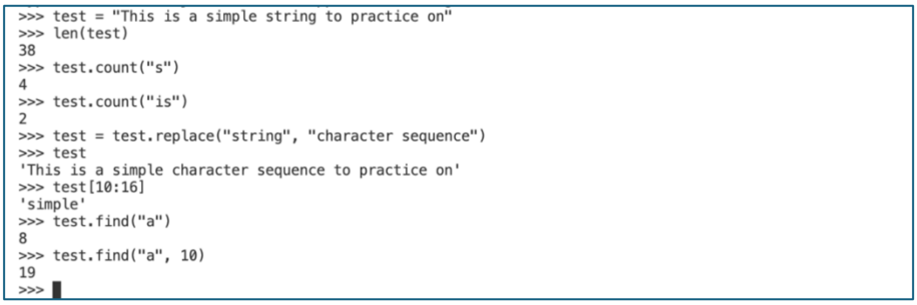
Figure 1.2 Immutable Strings

Do it now! Try modifying the value of the name variable, by issuing the command **`name[0]='X'`**. Observe the error message as displayed in Figure 1.2.

Try it now!	Given that strings are immutable, try and devise a command that has the same effect that you tried to achieve in the previous example. i.e., the value of **`name`** should be 'X' followed by the remaining characters from the original value of **`name`**.

Python provides a rich set of string manipulation facilities, some of the most useful of which are illustrated by the following set of examples.

Do it now!	Open the Python interpreter and enter the following commands shown in Figure 1.3. Verify the operation of the string manipulation functions described



```
>>> test = "This is a simple string to practice on"
>>> len(test)
38
>>> test.count("s")
4
>>> test.count("is")
2
>>> test = test.replace("string", "character sequence")
>>> test
'This is a simple character sequence to practice on'
>>> test[10:16]
'simple'
>>> test.find("a")
8
>>> test.find("a", 10)
19
>>> 
```

Figure 1.3 String Operations

Note that some operations are functions (e.g. **`len()`**) while others are methods (e.g. **`count()`**, **`replace()`**) – Typically, functions can be applied to other collections such as lists, while methods apply only to strings.

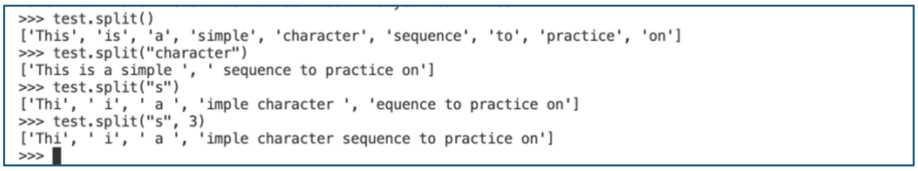
The **`len()`** function returns the number of characters (i.e. the length) in the string. Characters are numbered from 0 to length-1, with individual characters accessed by **`stringName[position]`**.

The **`count()`** method returns the number of non-overlapping occurrences of the sub-string that is passed as a parameter,

The **`find()`** method returns the first instance of the specified substring. Where a second parameter is supplied, it denotes the character position from which the search should begin.

Earlier, we noted that strings are **immutable** – once created, their value cannot be changed. This has a consequence for methods such as **`replace()`** where we have to return the method result to a new string object. By giving the new string the same variable name as the original, we cause the old string called **`test`** to be replaced by the new one.

Do it now!	Enter the commands shown in Figure 1.4 below into the Python interpreter and verify the operation of the string **`split()`** method.



```
>>> test.split()
['This', 'is', 'a', 'simple', 'character', 'sequence', 'to', 'practice', 'on']
>>> test.split("character")
['This is a simple ', ' sequence to practice on']
>>> test.split("s")
['Thi', ' i', ' a ', 'imple character ', 'equence to practice on']
>>> test.split("s", 3)
['Thi', ' i', ' a ', 'imple character sequence to practice on']
>>> 
```

Figure 1.4 Splitting Strings

The **`split()`** method optionally takes 2 parameters

- the *delimiter* to be used as the split character
- the *maximum number* of split operations to be performed

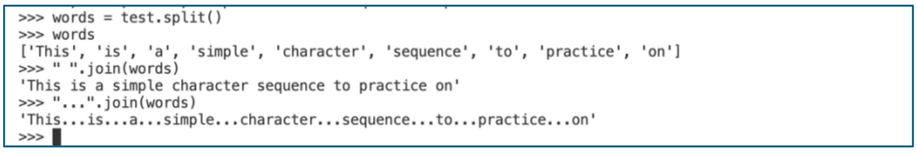
and returns a list of substrings of the original value.

The delimiter defaults to any sequence of whitespace characters. In the first example above, we demonstrate this by splitting the string into individual words, each retuned as a separate element in a list. We will examine list structures in more detail later in this Practical.

Note in the second example how the delimiter value does not form part of the returned list. Splitting on the instance of the word ‘character’ results in a list consisting of 2 elements – all text up to the delimiter word, and all text following the delimiter.

The third and fourth examples demonstrate the effect of the maximum-number-of-splits parameter. By default, the **`split()`** method will generate as many substrings as possible, but when a second parameter is provided, we stop splitting after the requisite number of splits have taken place. Note that the number of returned elements will be one greater than the parameter value – in the example above we can see how 3 split operations result in 4 elements being generated.

Do it now!	Enter the commands shown in Figure 1.5 below into the Python interpreter and verify the operation of the string **`join()`** method.

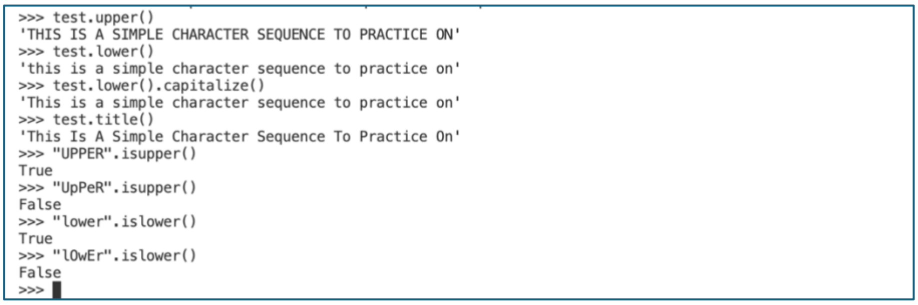


```
>>> words = test.split()
>>> words
['This', 'is', 'a', 'simple', 'character', 'sequence', 'to', 'practice', 'on']
>>> " ".join(words)
'This is a simple character sequence to practice on'
>>> "...".join(words)
'This...is...a...simple...character...sequence...to...practice...on'
>>> 
```

Figure 1.5 Joining strings

The **`join()`** method takes a list of string elements as a parameter and is applied to another string element that will be used as a divider in the joined string. For example, in the first **`join()`** operation above, we apply the **`split()`** method to a string consisting of a space character, resulting in a string object containing all of the words in the list separated by a space. In the second example, we illustrate a different string being used as the divider.

Do it now! Enter the commands shown in Figure 1.6 below into the Python interpreter and verify the operation of the case manipulation methods.



```
>>> test.upper()
'THIS IS A SIMPLE CHARACTER SEQUENCE TO PRACTICE ON'
>>> test.lower()
'this is a simple character sequence to practice on'
>>> test.lower().capitalize()
'This is a simple character sequence to practice on'
>>> test.title()
'This Is A Simple Character Sequence To Practice On'
>>> "UPPER".isupper()
True
>>> "UpPeR".isupper()
False
>>> "lower".islower()
True
>>> "lOwEr".islower()
False
>>> 
```

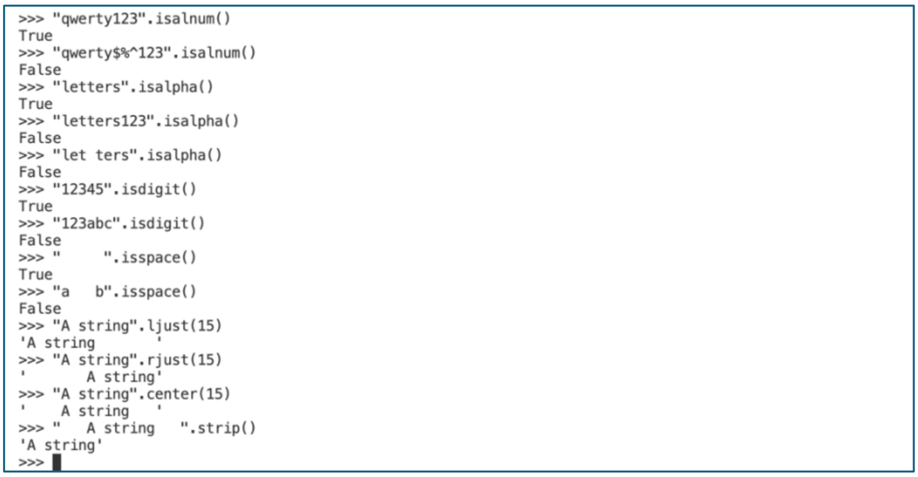
Figure 1.6 Case Manipulation

Python provides a wide range of methods that manipulate the case of string objects. Here we demonstrate **`upper()`** and **`lower()`** which return uppercase and lowercase versions of strings respectively, as well as **`capitalize()`** and **`title()`**, which capitalize the first letter of the string and the first letter of each word respectively.

In addition, there is a set of methods that test the case of string objects. Here, we demonstrate **`isupper()`** and **`islower()`**, which test for uppercase and lowercase strings respectively.

Note also the format of the Python Boolean constants **`True`** and **`False`**.

Do it now! Enter the final set of string manipulation commands shown in Figure 1.7 below into the Python interpreter and verify their operation.



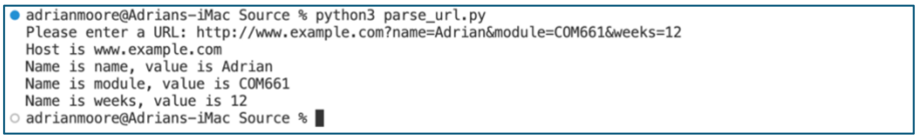
```
>>> "qwerty123".isalnum()
True
>>> "qwerty$%^123".isalnum()
False
>>> "letters".isalpha()
True
>>> "letters123".isalpha()
False
>>> "let ters".isalpha()
False
>>> "12345".isdigit()
True
>>> "123abc".isdigit()
False
>>> "     ".isspace()
True
>>> "a   b".isspace()
False
>>> "A string".ljust(15)
'A string       '
>>> "A string".rjust(15)
'       A string'
>>> "A string".center(15)
'    A string   '
>>> "   A string   ".strip()
'A string'
>>> █
```

Figure A1.7 Testing String Composition

Python provides functions that allow us to test the composition of a string. This can be very useful if we want to ensure that (for example) only digits are accepted for a particular input. Here we demonstrate tests for alpha-numeric (**`isalnum()`**), letters only (**`isalpha()`**), numeric only (**`isdigit()`**) and whitespace (**`isspace()`**).

Finally, we demonstrate methods that pad out a string to a specified number of characters. **`ljust()`** pads the string by adding space characters to the end of the string, **`rjust()`** pads by adding spaces to the beginning of the string, while **`center()`** adds extra spaces equally to both left and right. These are useful for formatting output in neat columns. The **`strip()`** method preforms the opposite function and removes extraneous whitespace from a string.

Try it now!	Write the Python script *parse_url.py* that reads from the keyboard a string value representing a full URL including a query string and extracts & prints its component parts. For example, the URL http://www.example.com?name=Adrian&module=COM661&weeks=12 should produce output such as that illustrated in Figure 1.8 below. **Note:** The query string is the part of the URL that follows the **`?`** character and contains a number of **`name=value`** pairs separated by the ampersand (**`&`**) and representing additional data to be passed to the webpage. **Hint** – your code should be generalized to work for any URL and any number of query string parameters



```
● adrianmoore@Adrians-iMac Source % python3 parse_url.py
  Please enter a URL: http://www.example.com?name=Adrian&module=COM661&weeks=12
  Host is www.example.com
  Name is name, value is Adrian
  Name is module, value is COM661
  Name is weeks, value is 12
○ adrianmoore@Adrians-iMac Source % ▋
```

Figure 1.8 URL Parser

1.2 File Handling

Up to now, application input and output has been to/from the standard console using the **`input()`** and **`print()`** functions. In this section we will examine the use of text files to store application data.

Files are opened the by **`open()`** function, that takes parameters specifying the location and name of the file and the type of access required and returns a file object which is used as the subject for further file activity.

The most common access modes are

Mode	Example	Description
r	f = open(“data.txt”, ”**r**”)	Open file for reading only. The file pointer is placed at the beginning of the file. This is the default mode
w	f = open(“data.txt”, ”**w**”)	Open file for writing only. If the file already exists, overwrite it with the new version.
a	f = open(“data.txt”, ”**a**”)	Open file for appending. File pointer is placed at the end of the file. Create file if not already existing

Examine the application ***longest_word.py***, that opens the text file ***words.txt***, which contains the full set of allowable English language words in crossword puzzles, organised one word per line in the file. The application reads each word from the file and determines the length of the longest word. It also creates an output file ***biggest.txt*** and writes each word with the longest length to this file.

Do it now!	Run the application ***longest_word.py*** and verify its operation. Examine the output file ***biggest.txt*** that is generated.

File: longest_word.py

```
fin = open("words.txt", "r")
biggest, num_of_words = 0, 0
for line in fin:
   word = line.strip()
   num_of_words = num_of_words+1
   if len(word) > biggest:
      biggest = len(word)

print("The longest of the {} words contains {} \
       characters".format(num_of_words, biggest))

fin.seek(0)
fout = open("biggest.txt", "w")
for line in fin:
      word = line.strip()
      if len(word) == biggest:
            output = word + "\n"
            fout.write(output)
            print(word)
fin.close()
fout.close()
```

The application is built around two file objects **`fin`** (file in) and **`fout`** (file out). **`fin`** is created by opening ***words.txt*** for reading in the first line of the code fragment.

Note in the second line, how we can create and initialise multiple variables on the same line. Here we will use **`biggest`** to store the length of the longest word, and **`num_of_words`** to count the total number of words in the file.

The **`for`**…**`in`** loop demonstrates a very useful technique for reading text files, by iterating across each line of the file. In the body of the loop, we use **`strip()`** to remove the newline character at the end of the line and compare the length of the word to the greatest length discovered so far – updating the value of **`biggest`** if required.

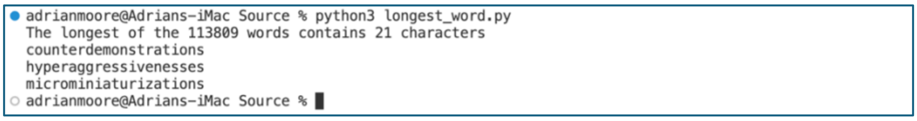
Once all words have been read, the **`for`** loop terminates and we report the total number of words read and the length of the longest word.

Now we need to traverse the input file again, this time comparing the length of each word to the longest length previously discovered and adding the word it to the output file if it is one of the group of longest words.

To re-read the input file from the beginning, we could simply close and re-open it, but instead we move the file pointer by invoking the **`seek()`** method. The parameter of **`seek()`** is an offset in bytes from the beginning of the file, hence **`seek(0)`** will indicate the top (beginning) of the file. The output file is opened for writing – causing the file to be created if it does not already exist.

The second **`for`**…**`in`** loop again iterates across each line in the input file. This time we compare the length of the current word against the biggest length found, and if they match, we **`write()`** the word (plus newline character) to the output file. Once the **for** loop terminates, we **`close()`** both files.

Running the application should result in output such as that illustrated in Figure 1.9 below.



```
adrianmoore@Adrians-iMac Source % python3 longest_word.py
The longest of the 113809 words contains 21 characters
counterdemonstrations
hyperaggressivenesses
microminiaturizations
adrianmoore@Adrians-iMac Source % █
```

Figure 1.9 File Processing

Try it now!	Write the Python application ***check_for_letters.py*** that prompts the user to enter 3 letters from the keyboard. The application should then generate an output file ***letters.txt*** that contains all words from ***words.txt*** that contain the 3 letters in the order in which they were supplied by the user. The letters do not have to appear consecutively in the word – for example letters ‘p’, ‘c’ and ‘o’ would result in the word ‘application’ being accepted.

1.3 Case Study – A Hangman Game

As a slightly more complex example of an interactive text processing application, we will develop a version of the game Hangman, where the object is to guess a word by suggesting letters that appear in the word. (Source: https://en.wikipedia.org/wiki/Hangman_(game))

The first task for the application is to have the computer supply a word for the user to guess. In our version, we will prompt the user to supply the number of letters that should be in the word, and the application selects a random word from ***words.txt*** with that number of letters.

File: hangman.py

```
import random

number_of_letters = int(input("Enter word length: "))
words = get_words(number_of_letters)
print("There are {} words with {} letters". \
             format(len(words), number_of_letters))

guess_word = words[random.randint(0, len(words)-1)]

print("Guessing the word: {}".format(guess_word))
```

Here, we call the function **`get_words()`** (which we will develop in the next stage), passing to it the number of letters required. This function returns a list of words from which we select one at random. The selected word is assigned to the variable **`guess_word`**.

File: hangman.py

```
import random

def get_words(number_of_letters):
   words_found = []
   fin = open("words.txt", "r")
   for line in fin:
      word = line.strip()
      if len(word) == number_of_letters:
         words_found.append(word)
   fin.close()
   return words_found

...
```

The function **`get_words()`** opens the file ***words.txt*** and reads each word in turn. If the length of a word matches the number supplied by the user, then that word is added to the **`words_found`** list by the **`append()`** method. Once all words have been tested, the **`words_found`** list is returned by the function.

Now that we have the word for the user to guess, we can move to the main game loop. In this version of the game, we give the user 6 lives (corresponding to head, torso, 2 arms and 2 legs in the classic pencil-and-paper version). For as long as the user has any lives remaining, they suggest a letter that might be found in the word. If the letter is present, the computer reveals all instances of the letter – otherwise the user is informed that the letter is not present, and he loses a life.

File: hangman.py

```
...

lives = 6
guess_string = "_" * number_of_letters
while lives > 0:
    this_letter = input("Guess a letter: ")
    if this_letter in guess_word:
        guess_string = \
             replace_all(guess_string, guess_word, this_letter)
        if guess_string == guess_word:
            print("You guessed the word!")
            break
    else:
        lives = lives - 1
        print("Letter not found lives remaining: {}". \
               format(lives))
    print(guess_string)
```

Note the **`if`**…**`in`** operator that provides an easy test for the presence of a character in a string. This avoids the need to manually iterate across the string in a loop.

The variable **`guess_string`** is used as the visual representation of the user's progress. Initially it is set to a string of "_ _ _ _ _ _ *",* with one underscore character for each letter in the word. If the letter suggested by the user is contained in the word, the function **`replace_all()`** replaces the appropriate underscores with the actual letter. For example, if the word is "apple", and the user's first guess is 'p', the value of **`guess_string`** will change from "_ _ _ _ _*"* to *"*_ p p _ _*"*. The function **`replace_all()`** is presented below.

File: hangman.py

```
def replace_all(guess, word, letter):
   for pos in range(len(guess)):
      if word[pos] == letter:
         guess = guess[:pos] + letter + guess[pos+1:]
   return guess

...
```

The function **`replace_all()`** takes 3 parameters

- **`guess`** – the current state of the **`guess_string`** (e.g. “_ p p _ _”)
- **`word`** – the word being guessed by the user
- **`letter`** – the letter suggested by the user

and returns the updated version of the **`guess_string`**. The function operates by iterating across each letter in the word, testing for the letter being the same as that provided by the user. Where a match is found, a new version of the **`guess_string`** is created by taking the characters before and after the current position and appending them either side of the suggested letter. For example, with a word “apple”, a **`guess_string`** value ‘_ p p _ _’ and letter “l”, the function would concatenate the strings “_ p p”, “l” and “_”.

The final stage of the application adds the name and score of a successful player to the roll of honour maintained in the file ***hangmanScores.txt***.

File: hangman.py

```
      ...

      if lives > 0:
         player = input("Enter player name: ")
         fout = open("hangman_scores.txt", "a")
         score = lives * number_of_letters
         new_score_text = "{}, {}\n".format(player, score)
         fout.write(new_score_text)
         fout.close()
```

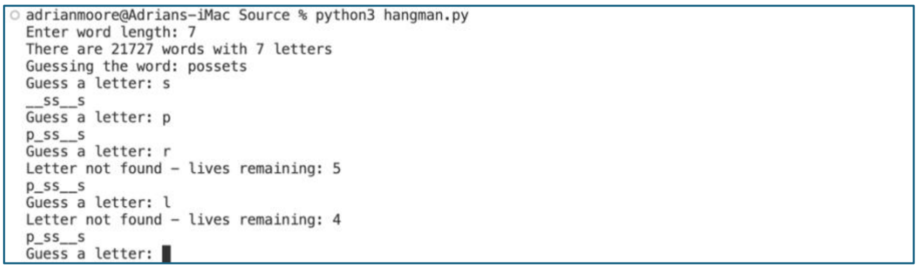
If the player reaches the end of the game with some lives intact (i.e., they have successfully guessed the word), the application prompts them to enter their name. The results file ***hangman_scores.txt*** is opened in “append” mode, and the player’s name and score are added to the file.

Note that we provide an incentive for users to attempt longer words by defining the score as the number of lives remaining multiplied by the number of letters in the word.

All of the code presented in this section is available in the file ***hangman.py***.

Do it now!

Execute the application ***hangman.py*** and see its operation. Comment out the line that prints the word to be guessed to get a "true" game. Perform multiple runs and see how the file ***hangman_scores.txt*** records details of all successful games.



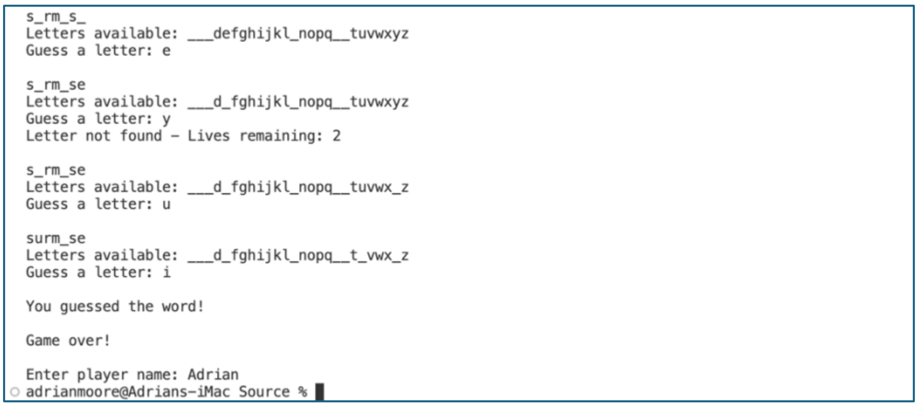
```
○ adrianmoore@Adrians-iMac Source % python3 hangman.py
  Enter word length: 7
  There are 21727 words with 7 letters
  Guessing the word: possets
  Guess a letter: s
  __ss__s
  Guess a letter: p
  p_ss__s
  Guess a letter: r
  Letter not found - lives remaining: 5
  p_ss__s
  Guess a letter: l
  Letter not found - lives remaining: 4
  p_ss__s
  Guess a letter: █
```

Figure 1.10 Hangman

Try it now!

Further develop the hangman game to make the following improvements:

1) The game should keep track of which letters are still available for selection and should display this information to the user before each turn. See Figure 1.11 below for an example of the output required.
2) When a player runs out of lives, the game should tell them the word they failed to guess
3) The scores in the file should be cumulative – i.e., if a player's name is already in the file, then the new score should be added to his existing score. The file should maintain one line per player – rather than one line per game played.



```
  s_rm_s_
  Letters available: ___defghijkl_nopq__tuvwxyz
  Guess a letter: e

  s_rm_se
  Letters available: ___d_fghijkl_nopq__tuvwxyz
  Guess a letter: y
  Letter not found - Lives remaining: 2

  s_rm_se
  Letters available: ___d_fghijkl_nopq__tuvwx_z
  Guess a letter: u

  surm_se
  Letters available: ___d_fghijkl_nopq__t_vwx_z
  Guess a letter: i

  You guessed the word!

  Game over!

  Enter player name: Adrian
○ adrianmoore@Adrians-iMac Source % █
```

Figure 1.11 Hangman Enhanced

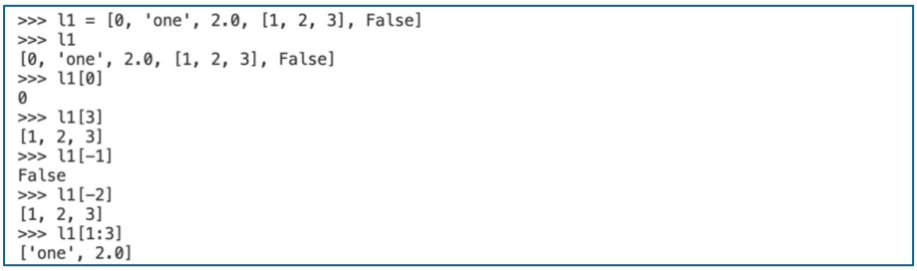
1.4 Python Data Structures

1.4.1 Lists

We have previously seen the Python list structure as the output from the string **`split()`** method, where the substrings are presented as a collection enclosed within square brackets and separated by commas such as

```
[ ‘this’, ‘is’, ‘a’, ‘list’ ]
```

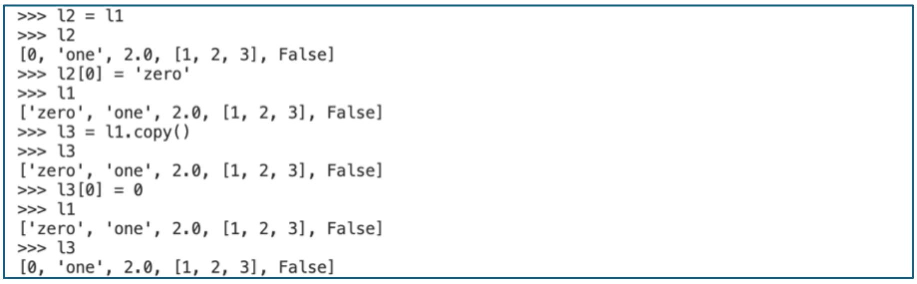
In general, lists are ordered collections where each element has an index position beginning from zero, elements can be of mixed types including lists, and duplicates are allowed. Figure 1.12 below illustrates the definition and use of lists in Python. Note that the slice operator previously used in string manipulation can also be applied to lists.



```
>>> l1 = [0, 'one', 2.0, [1, 2, 3], False]
>>> l1
[0, 'one', 2.0, [1, 2, 3], False]
>>> l1[0]
0
>>> l1[3]
[1, 2, 3]
>>> l1[-1]
False
>>> l1[-2]
[1, 2, 3]
>>> l1[1:3]
['one', 2.0]
```

Figure 1.12 Defining and Using Lists

Note also that assigning a list to a new variable simply creates a new reference to the original list. If the copy is intended to create a new list that can be manipulated separately, then the list **`copy()`** method should be used, as shown in Figure 1.13.

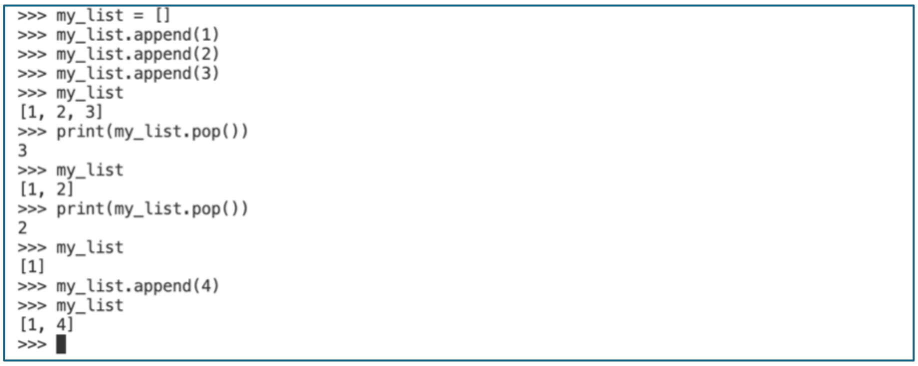


```
>>> l2 = l1
>>> l2
[0, 'one', 2.0, [1, 2, 3], False]
>>> l2[0] = 'zero'
>>> l1
['zero', 'one', 2.0, [1, 2, 3], False]
>>> l3 = l1.copy()
>>> l3
['zero', 'one', 2.0, [1, 2, 3], False]
>>> l3[0] = 0
>>> l1
['zero', 'one', 2.0, [1, 2, 3], False]
>>> l3
[0, 'one', 2.0, [1, 2, 3], False]
```

Figure 1.13 Copying Lists

Do it now!	Enter the commands shown in Figure 1.12 and 1.13 into the Python interpreter to ensure that you understand their operation. Try different values of your own to check your understanding of the manipulation of lists and their individual elements.

Python also provides a pair of methods that allow us to manipulate lists without using their index values. The **`append()`** and **`pop()`** methods operate at the end of the list (i.e. the highest index position), such that **`append()`** adds an element to the end of the list and **`pop()`** removes and returns the element at the end of the list. These methods are illustrated by Figure 1.14 below which defines an empty list **`my_list`** before appending three items. We then see that the first item popped is the last to be added, before popping a further item and appending a final element.



```
>>> my_list = []
>>> my_list.append(1)
>>> my_list.append(2)
>>> my_list.append(3)
>>> my_list
[1, 2, 3]
>>> print(my_list.pop())
3
>>> my_list
[1, 2]
>>> print(my_list.pop())
2
>>> my_list
[1]
>>> my_list.append(4)
>>> my_list
[1, 4]
>>> █
```

*Figure 1.14 Using **`append()`** and **`pop()`***

As a slightly more complex example, we will consider a situation where we want to maintain multiple lists in a single structure. Here, we create a word storage structure as a list of lists where each sub-list will be used to store words that begin with successive letters of the alphabet; such that element [0] is a list of words that begin with the letter ‘a’, element [1] is a list of words that begin with the letter ‘b’, and so on, as illustrated below.

```
[
   [ ‘apple’, ‘archer’, …, ],
   [‘ball’, ‘boy’, ‘butterfly’, …, ]
   …
   [‘zebra’, ‘zoo’, …]
]
```

The words to be stored in the structure will be a set of 50 words randomly chosen from the file of allowable words in crossword puzzles and the contents of the structure will be displayed when the application runs.

The application is presented in Figure 1.15 below. First, we open the file containing the full set of allowable words and read them into a list **`all_words`**. Next, we initialize **`words_list`** to an empty list and then use a **`for`** loop to append 26 empty lists into **`words_list`**. This will result in **`words_list`** being a list of empty lists in the form

```
[ [], [], [], [], …, [] ]
```

Note that there is no way in Python to declare an empty list to store a specified number of elements in a single operation. Note also the use of the dummy variable _ in the definition of the **`for`** loop

```
for _ in range(26):
```

This is used when we deploy a **`for`** loop in such a way that we have no need to access the loop counter variable. Had we used a regular loop counter variable here, such as

```
for i in range(26):
```

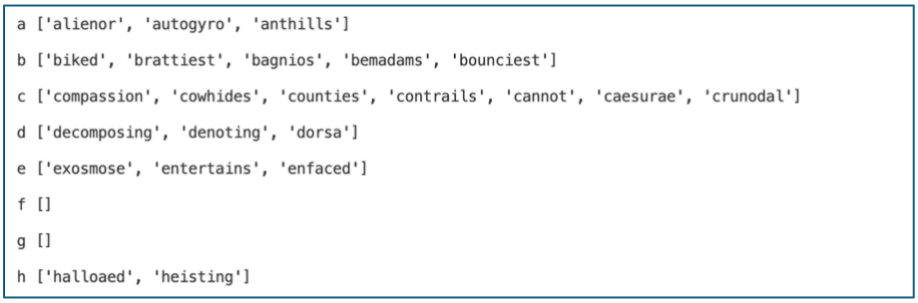
then some Python environments will generate a warning that the variable **`i`** is unused and so the dummy variable is a way of avoiding this.

Having initialized the **`words_list`**, we then choose 50 random entries from **`all_words`**. For each word selected, we store it in the **`words_list`** entry corresponding to its initial letter, such that all selected words beginning with ‘a’ will be in **`words_list[0]`**, all selected words beginning with ‘b’ will be in **`words_list[1]`**, and so on.

Finally, we print each letter of the alphabet together with its corresponding element from **`words_list`**.

File: store_words_list.py

```
import random

all_words = []
fin = open("words.txt")
for line in fin:
    word = line.strip()
    all_words.append(word)
fin.close()

words_list = []
for _ in range(26):
    words_list.append([])

for _ in range(50):
    random_word = all_words[random.randint(0,len(all_words)-1)]
    words_list[ord(random_word[0]) -  \
                ord('a') ].append(random_word)

letters = "abcdefghijklmnopqrstuvwxyz"
for i in range(26):
    print(letters[i], words_list[i], '\n')
```

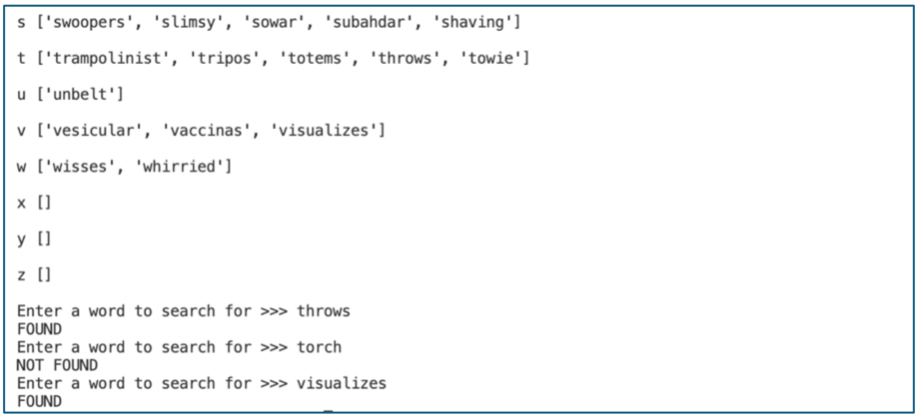


```
a ['alienor', 'autogyro', 'anthills']

b ['biked', 'brattiest', 'bagnios', 'bemadams', 'bounciest']

c ['compassion', 'cowhides', 'counties', 'contrails', 'cannot', 'caesurae', 'crunodal']

d ['decomposing', 'denoting', 'dorsa']

e ['exosmose', 'entertains', 'enfaced']

f []

g []

h ['halloaed', 'heisting']
```

Figure 1.15 Lists of Randomly Selected Words

Do it now!	Run the file ***store_words_list.py*** and verify that you receive output similar to that shown in Figure 1.15 above. You should be able to observe that multiple runs of the code generate varying output and that each of the randomly selected words is stored in the appropriate element of the list.

Try it now!	Add code to ***store_words_list.py*** so that the user is given 3 opportunities to enter a search word from the console. For each word entered, the list is searched and the result "FOUND" or "NOT FOUND" is displayed as shown in Figure 1.16 below.



```
s ['swoopers', 'slimsy', 'sowar', 'subahdar', 'shaving']

t ['trampolinist', 'tripos', 'totems', 'throws', 'towie']

u ['unbelt']

v ['vesicular', 'vaccinas', 'visualizes']

w ['wisses', 'whirried']

x []

y []

z []

Enter a word to search for >>> throws
FOUND
Enter a word to search for >>> torch
NOT FOUND
Enter a word to search for >>> visualizes
FOUND
```

Figure 1.16 Searching the Lists of Words

1.4.2 Dictionaries

A dictionary is a data structure that can store any number of Python objects. Dictionaries consist of pairs of keys and their corresponding values.

Do it now! Open the Python command line interpreter and verify the following sequence of Python dictionary manipulation commands.

First, we will create a new dictionary. Note that dictionaries are enclosed by { } brackets.

```
>>> modules = {“COM668”: “Project”, “COM661”: “Full Stack
Strategies and Development”, “COM682”: “Cloud Native
Development”}
```

This creates a dictionary containing 3 items, each consisting of a module code as a key and a module title as a value. To retrieve the value associated with any key, we can refer to is as…

```
>>> modules[“COM668”]
‘Project’
```

If we wish to add a new item to the dictionary, we simply assign a value to its key, e.g.

```
>>> modules[“COM662”] = “Data Analytics”
>>> modules
{‘COM668’: ‘Project’, ‘COM661’: ’Full Stack Strategies and
Development’, ‘COM682’: ‘Cloud Native Development’, ‘COM662’:
‘Data Analytics’}
```

Note: In versions of Python before 3.7, the order of elements in a dictionary was entirely arbitrary and not governed by the order in which they are added. From v3.7 on, dictionary order is preserved, but it is not considered good practice to depend on it.

Each key in a dictionary can correspond to only one value. If a second value for any key is presented, then it over-writes the previous entry.

```
>>> modules[“COM682”] = “Cloud Development”
>>> modules
{‘COM668’: ‘Project’, ‘COM661’: ’Full Stack Strategies and
Development’, ‘COM682’: ‘Cloud Development’, ‘COM662’: ‘Data
Analytics’}
```

Elements can be removed from a dictionary by the `del` command.

```
>>> del modules[“COM662”]
>>> modules
{‘COM668’: ‘Project’, ‘COM661’: ’Full Stack Strategies and
Development’, ‘COM682’: ‘Cloud Development’ }
```

We can also test for the presence of a key using the powerful `in` operator.

```
>>> ‘COM662’ in modules
False
>>> ‘COM661’ in modules
True
```

Dictionaries are often a preferred solution to lists for complex data structures given their flexibility. Just as a list element can be of any type including a list, so the value of a dictionary can be of any type, including lists and dictionaries. We will demonstrate this be revisiting our previous word storage application and re-factoring it to use a dictionary instead of a list of lists.

Note in the code box below (and throughout the module material), the … notation at the top of the listing refers to code that is unchanged from the previous version. In this case, we import the `random` module and generate the `all_words` list as before. In the remaining code, the modifications from the previous version are highlighted in bold text.

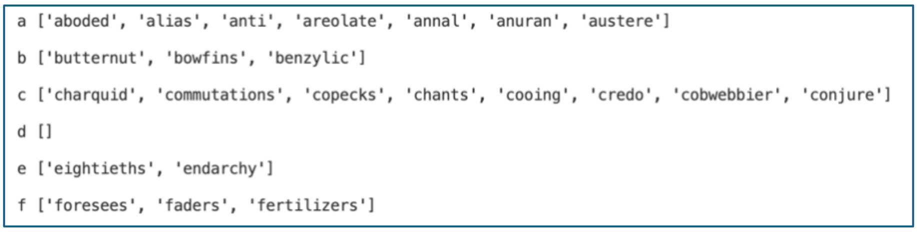
This time, we define **`word_dict`** as an empty dictionary and use the **`for`** loop to add 26 elements, where the keys are successive letters of the alphabet (from 'a' to 'z') and the values are empty lists.

Then, for each of the 50 randomly selected words, we append that word to the appropriate dictionary entry according to its initial letter, so that words beginning in 'a' are appended to the list at dictionary entry 'a' and so on.

Finally, we traverse the characters of the **`letters`** string, printing out for each the letter and the corresponding dictionary value.

File: store_words_dictionary.py

```
      ...

      letters = "abcdefghijklmnopqrstuvwxyz"
      words_dict = {}
      for i in range(26):
          words_dict[letters[i]] = []

      for _ in range(50):
          random_word=all_words[random.randint(0,len(all_words)-1) ]
          words_dict[random_word[0]].append(random_word)

      for letter in letters:
          print(letter, words_dict[letter], '\n')
```

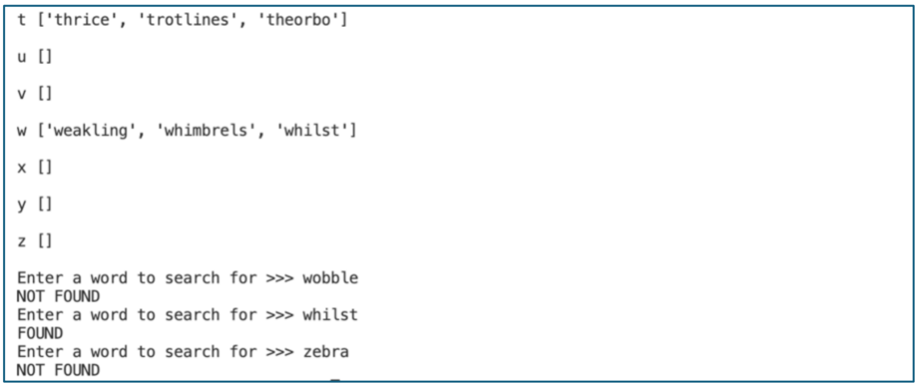


```
a ['aboded', 'alias', 'anti', 'areolate', 'annal', 'anuran', 'austere']

b ['butternut', 'bowfins', 'benzylic']

c ['charquid', 'commutations', 'copecks', 'chants', 'cooing', 'credo', 'cobwebbier', 'conjure']

d []

e ['eightieths', 'endarchy']

f ['foresees', 'faders', 'fertilizers']
```

Figure 1.17 Dictionary of Randomly Selected Words

Do it now!	Run the file ***store_words_dictionary.py*** and verify that you receive output similar to that shown in Figure 1.17 above. You should be able to observe that multiple runs of the code again generate varying output and that each of the randomly selected words is stored in the appropriate dictionary entry.

Try it now!	Repeat the previous exercise by adding code to ***store_words_dictionary.py*** so that the user is given 3 opportunities to enter a search word from the console. For each word entered, the dictionary is searched and the result "FOUND" or "NOT FOUND" is displayed as shown in Figure 1.18 below.



```
t ['thrice', 'trotlines', 'theorbo']

u []

v []

w ['weakling', 'whimbrels', 'whilst']

x []

y []

z []

Enter a word to search for >>> wobble
NOT FOUND
Enter a word to search for >>> whilst
FOUND
Enter a word to search for >>> zebra
NOT FOUND
```

Figure 1.18 Searching the Dictionary of Words

1.4.3 Measuring Performance

The main performance advantage offered by dictionaries over lists is the direct access to each individual element via the dictionary key. Although the **`in`** keyword can be used in the same way on list and dictionary structures, the fact that it operates on list **elements** but on dictionary **keys** makes a significant difference when using complex data structures.

For example, if a list of modules is organised as

```
    modules = [ 'COM661', 'COM668', 'COM682']
```

then the statement

```
    if 'COM668' in modules:
```

can check the entire list in a single operation. Likewise, if the **`modules`** structure is organised as a dictionary with the module code as the key and the module coordinator as the value, such as

```
    modules = { 'COM661' : 'Adrian',
                'COM668' : 'George',
                'COM682' : 'Zeeshan' }
```

then the same `if` statement can confirm the presence or otherwise of a given module in the collection. However, if the list structure is to contain all of information as the dictionary (i.e. both the module code and coordinator), then the list definition would be such as

```
modules = [  [‘COM661’, ‘Adrian’],
             [‘COM668’, ‘George’],
             [‘COM682’, ‘Zeeshan’]
          ]
```

Now the `if` statement would fail to pick up a matching module code since no list entry is an exact match for a module code and every element would need to be checked to verify whether the desired module code is present, such as

```
for module in modules:
   if module[0] == ‘COM668’:
      return True
return False
```

The effect of this can be clearly seen by an example that models the storage of keywords and URLs in a search engine. Following a web scraping exercise, a search engine maintains a collection of keywords that have been scraped from webpages along with the URLs on which the keywords can be found. If a user enters a keyword in a search operation, the search engine responds with the list of URLs where the word is found. When stored as a list, the structure might be

```
contents = [
              [ ‘word1’, [ ‘URL 1’, ‘URL 2’, ‘URL 3’ ] ],
              [ ‘word2, [ ‘URL 1’, ‘URL 4’, ‘URL 5’ ] ],
              [ ‘word3’, [ ‘URL 2’, ‘URL 5’, ‘URL 6’ ] ],
              ...
           ]
```

representing that `word1` is found on pages at `URL 1`, `URL 2` and `URL 3`; `word2` is found on pages at `URL 1`, `URL 4` and `URL 5`, and so on. Now, to check whether the structure contains a given `search_term`, the entire structure would need to be checked one element at a time, such as

```
for entry in contents:
   if entry[0] == search_term:
      return entry[1]
return []
```

In the worst case, where the search term is not stored in the collection, then every element would need to be compared. This is demonstrated by the example ***timed_list_search.py*** which constructs a collection of between 1000 and 32000 unique keywords and measures the time required to search the collection for a term that is not present.

First, we provide the function **`time_execution()`** that takes a reference to a piece of Python code as a parameter and returns the time required to execute this code. Execution time is calculated by taking a system clock reading by the **`time.perf_counter()`** function both before and after executing the code and subtracting the start time from the finish time to obtain the execution time in nanoseconds.

The function **`add_word()`** accepts the current state of the collection as a parameter, together with the word to be added to the collection and the URL at which the word has been found. The function checks whether the collection already contains an entry for the word and if so, adds the URL to the list for that entry. If the word is one that has not previously been seen, then a new collection entry is added for that word, with the list of URLs initialised to the one passed as a parameter.

File: timed_list_search.py

```
import time

def time_execution(code):
    start = time.perf_counter()
    eval(code)
    run_time = time.perf_counter() - start
    return run_time

def add_word(this_collection, keyword, url):
    for entry in this_collection:
        if entry[0] == keyword:
            entry[1].append(url)
            return
    this_collection.append([keyword,[url]])
```

Next, we provide functions to construct and search the collection. The function **`make_collection()`** accepts a large list of words as a parameter as well as an integer that states the size of collection required. The function creates a new collection entry for each word, specifying that the word has been found at a dummy URL. The collection of the specified size is then returned from the function. The function **`lookup()`** takes as parameters the word to be searched for and the current collection being used. The function checks each collection entry in turn, returning the corresponding list of URLs if a match is found or an empty list if there is no match.

File: timed_list_search.py

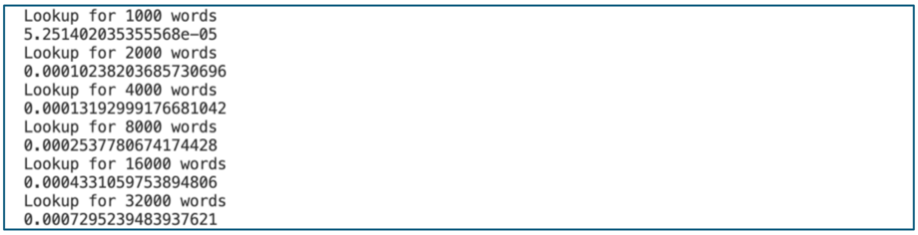
```
...

def make_collection(all_words, size):
      this_collection = []
      for i in range(size):
            add_word(this_collection, all_words[i], "dummyURL")
      return this_collection

def lookup(keyword, collection):
      for e in collection:
            if e[0] == keyword:
                  return e[1]
      return []
```

Finally, we read the complete list of words from the previous example into a list and compare execution times for a range of collection sizes from 1000 to 32,000 elements. For each size, we create a collection of that size before measuring the execution time of the worst-case scenario search – that when the word being searched does not exist within the collection meaning that every collection element must be examined.

File: timed_list_search.py

```
...

all_words = []
fin = open("words.txt")
for line in fin:
    word = line.strip()
    all_words.append(word)
fin.close()

collection = make_collection(all_words, 1000)
print("Lookup for 1000 words")
print(time_execution("lookup('xxx', collection)"))

# and also for 4000, 8000, 16000 and 32000
```

Running the program demonstrates that the execution time grows with the size of the collection, and it can be seen from the results in Figure 1.19 that execution time is roughly doubled as the collection size is doubled.



```
Lookup for 1000 words
5.251402035355568e-05
Lookup for 2000 words
0.00010238203685730696
Lookup for 4000 words
0.00013192999176681042
Lookup for 8000 words
0.0002537780674174428
Lookup for 16000 words
0.0004331059753894806
Lookup for 32000 words
0.0007295239483937621
```

Figure 1.19 Search time increases proportionally

Having observed the performance of a list-based approach, we can then re-factor the code to use a dictionary instead. This time, the collection structure will be as

```
contents = {
              'word1' : [ 'URL 1', 'URL 2', 'URL 3' ],
              'word2' : [ 'URL 1', 'URL 4', 'URL 5' ],
              'word3' : [ 'URL 2', 'URL 5', 'URL 6' ],
              ...
           }
```

where each keyword in the collection is a dictionary key and the corresponding value is the previous list of URLs where that word has been observed.

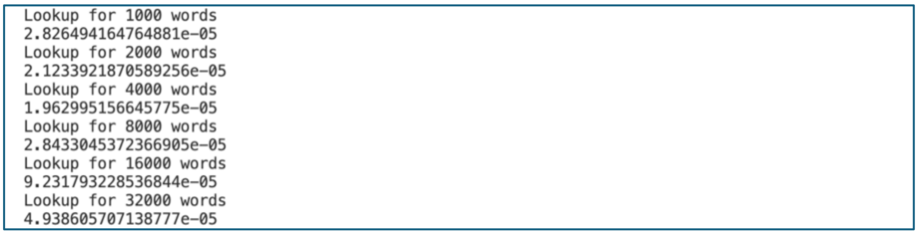
The first update is in the **`add_word()`** function, where rather than check each element for a matching keyword, we can now use the **`in`** operator to check all keys in a single operation. If a match is found, then the URL is added to the corresponding list, otherwise a new collection element is created with the keyword as the key and the list initialized to the URL.

`make_collection()`only requires that the collection be initialized to an empty dictionary rather than the previous list, while **`lookup()`** also makes use of the **`in`** operator to return the corresponding list if a match is found, or the empty list otherwise.

File: timed_dict_search.py

```
...

def add_word(this_collection, keyword, url):
    if keyword in this_collection:
        this_collection[keyword].append(url)
    else:
        this_collection[keyword] = [url]

def make_collection(all_words, size):
    this_collection = {}
    for i in range(size):
        add_word(this_collection, all_words[i], "dummyURL")
    return this_collection

def lookup(keyword, collection):
    if keyword in collection:
        return collection[keyword]
    else:
        return []

...
```

Now when we run the application, we see not only that overall execution time is much faster, but also that the individual search times are relatively constant – in other words the time required to search a dictionary is unaffected by an increase in dictionary size.



```
Lookup for 1000 words
2.826494164764881e-05
Lookup for 2000 words
2.1233921870589256e-05
Lookup for 4000 words
1.962995156645775e-05
Lookup for 8000 words
2.8433045372366905e-05
Lookup for 16000 words
9.231793228536844e-05
Lookup for 32000 words
4.938605707138777e-05
```

Figure 1.20 Search time remains constant

Do it now!

Run the files ***timed_list_search.py*** *and* ***timed_dict_search.py*** and verify that you receive output similar to that shown in Figures 1.19 and 1.20 above. Your execution times may not correspond exactly with those in the Figures, but you should be able to see that while the search time for a list implementation roughly doubles as the list size doubles, the search time for a dictionary is unaffected by dictionary size.

1.4.4 Serializing Data Structures

We have seen how lists and dictionaries can be combined to create complex data structures, but this effort is pointless without some way of storing our structure so that we don’t have to generate it from scratch every time we want to execute the application. In the search engine example, we could traverse our collection, writing the data to a file item by item – but this would lose the structure that our code has worked so hard to generate.

Fortunately, Python provides **Pickle** – a powerful module that serializes and de-serializes data structures so that they can be written to/read from a text file in a single operation. Consider the code below, which extends the ***store_words_dictionary.py*** example to use Pickle’s `dump()` method to write the data structure to a text file in a single operation. Note that we use “wb” as the file mode when opening the file for writing, respectively, rather than the more usual “w”. The additional “b” flag denotes that the information being written and read is a byte stream, rather than simple text.

File: store_words_dictionary_pickled.py

```
import random, pickle

...

fout = open("words_dictionary.txt", "wb")
pickle.dump(words_dict, fout)
fout.close()
```

Do it now!	Run the file ***store_words_dictionary_pickled.py*** and see how it creates the new file ***words_dictionary.txt*** that contains a representation of the dictionary structure.

We can demonstrate that the structure has been saved by reading it from the file into a new Python application. Examine the source of ***retrieve_words_dictionary_pickled.py*** which uses the Pickle `load()` method to read the list back into a Python variable. The contents of the variable can then be parsed and printed in the usual way, verifying that the dictionary has been successfully saved and then loaded.

File: retrieve_words_dictionary_pickled.py

```
import pickle

fin = open("words_dictionary.txt", "rb")
words_dict = pickle.load(fin)
fin.close()

letters = "abcdefghijklmnopqrstuvwxyz"
for letter in letters:
    print(letter, words_dict[letter], '\n')
```

Do it now! Run the file ***retrieve_words_dictionary_pickled.py*** and verify that the dictionary contents printed to the console are the same as those originally written.

Try it now! Revisit your Hangman application from Section 1.3 so that the high score table is maintained as a dictionary rather than a text file. Modify the application so that the dictionary of scores is automatically **`pickle`**d to a file after each update.

1.5 Further Information

- http://en.wikibooks.org/wiki/Python_Programming/Variables_and_Strings
  Python variables and strings

- http://docs.python.org/library/stdtypes.html#string-methods
  Python manual – string methods

- https://docs.python.org/3/library/stdtypes.html - printf-style-string-formatting
  Python manual – string formatting operations

- http://www.tutorialspoint.com/python3/python_strings.htm
  Python strings from Tutorialspoint

- https://www.tutorialspoint.com/python3/file_methods
  Python files I/O from Tutorialspoint

- https://www.thoughtco.com/analyze-a-file-with-python-2813717
  Ways to work with files in Python

- https://www.geeksforgeeks.org/python-program-to-read-file-word-by-word/
  Processing every word in a file

- http://docs.python.org/tutorial/inputoutput.html#reading-and-writing-files
  Python manual – reading and writing of files

- http://docs.python.org/library/pickle.html
  Python Pickle module

- http://wiki.python.org/moin/UsingPickle
  Using Pickle

- https://www.w3schools.com/python/python_lists.asp
  Python Lists

- http://www.tutorialspoint.com/python/python_dictionary.htm
  Python dictionary tutorial

- http://www.i-programmer.info/programming/python/3990-the-python-dictionary.html
  The Python dictionary: i-programmer tutorial

- http://www.greenteapress.com/thinkpython/html/book012.html
  Think Python! Dictionaries

- https://builtin.com/articles/timing-functions-python
  Timing Python Functions