COM661 Full Stack Strategies and Development

Practical A1: Python Text Processing

Aims

- To introduce simple string processing via the Python interpreter
- To demonstrate splitting and joining strings
- To demonstrate manipulation of the case 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 the file pointer as a cursor within a text file

Contents

A1.1 TEXT PROCESSING 2
A1.2 FILE HANDLING 8
A1.3 CASE STUDY – A HANGMAN GAME 11
A1.4 FURTHER INFORMATION 15

A1.1 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 **version 3.9** 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 There is still a large legacy community that uses the older Python 2.7, but official support for this will soon be withdrawn.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.

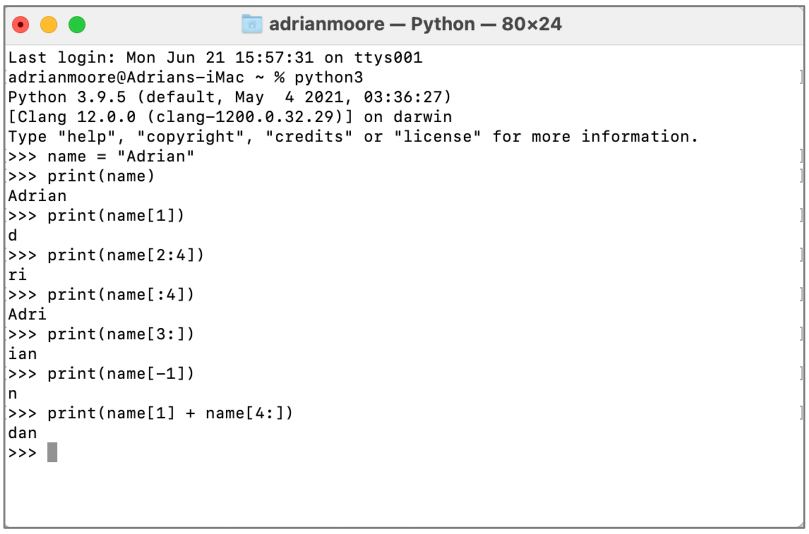
String values in Python are enclosed in either single or double quote characters and are concatenated by the **+** operator. Python also supports a powerful **slice** operator that can be applied to strings. The slice operator is invoked by **`string[start:end]`** which extracts all characters from position **`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 A1.1 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).

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.

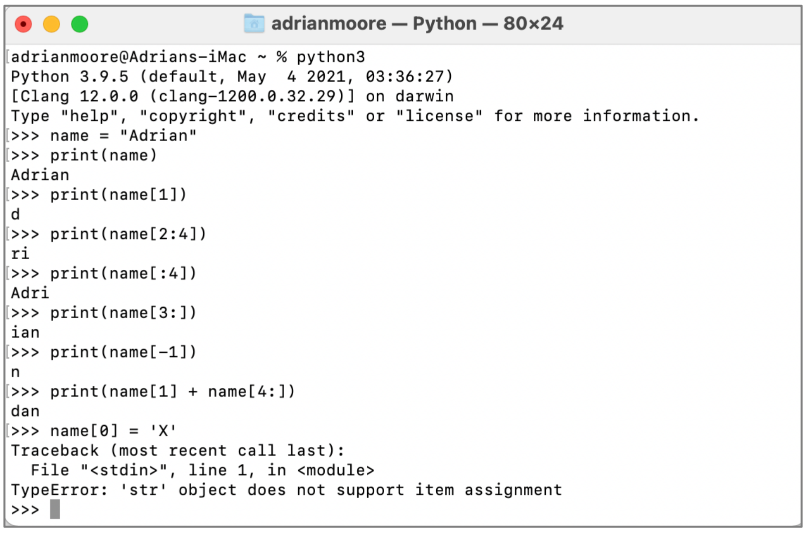
On a MacOS machine, you open the application called Terminal that will be pre-installed on the computer. You can find this by opening the Launchpad on the Dock (Terminal is most often in a group of applications called “Other”), or by searching for it in the Finder.



```
Last login: Mon Jun 21 15:57:31 on ttys001
adrianmoore@Adrians-iMac ~ % python3
Python 3.9.5 (default, May  4 2021, 03:36:27)
[Clang 12.0.0 (clang-1200.0.32.29)] 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 A1.1 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 A1.2 below.



```
adrianmoore@Adrians-iMac ~ % python3
Python 3.9.5 (default, May  4 2021, 03:36:27)
[Clang 12.0.0 (clang-1200.0.32.29)] 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
>>> name[0] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> 
```

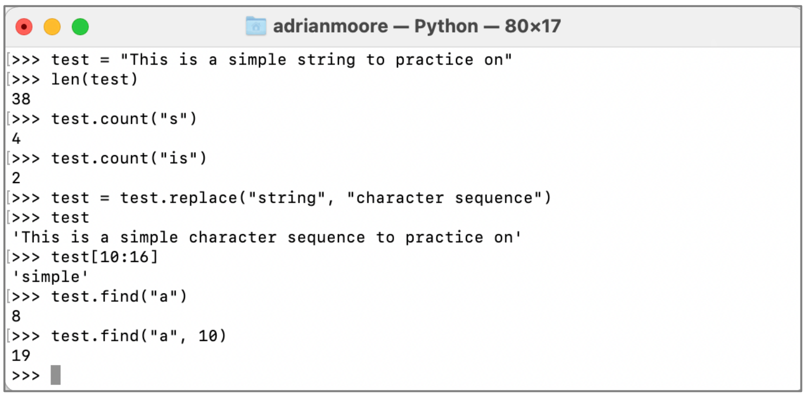
Figure A1.2 Immutable Strings

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 A1.2.

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 A1.3. 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 A1.3 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.

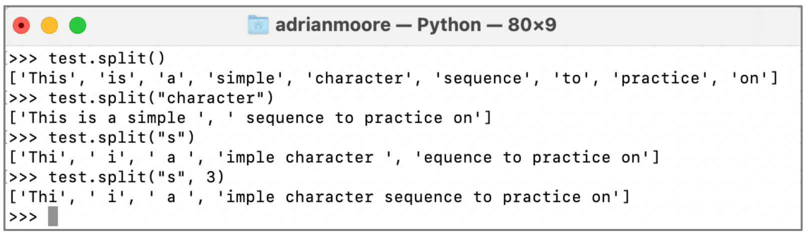
The **`len()`** function returns the number of characters (i.e. the length) in the string. Characters are numbered from 0 to length-1, with individual characters accessed by **`stringName[position]`**.

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.

In the previous practical, 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 A1.4 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 A1.4 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.

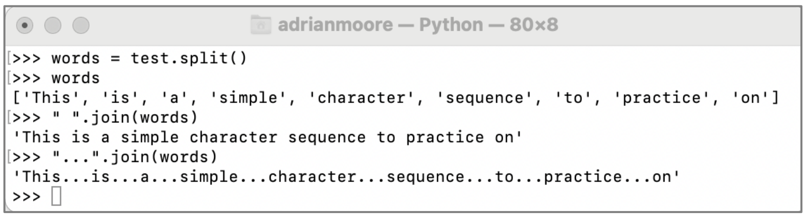
and returns a list of substrings of the original value.

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.

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 A1.5 below into the Python interpreter and verify the operation of the string **`join()`** method.

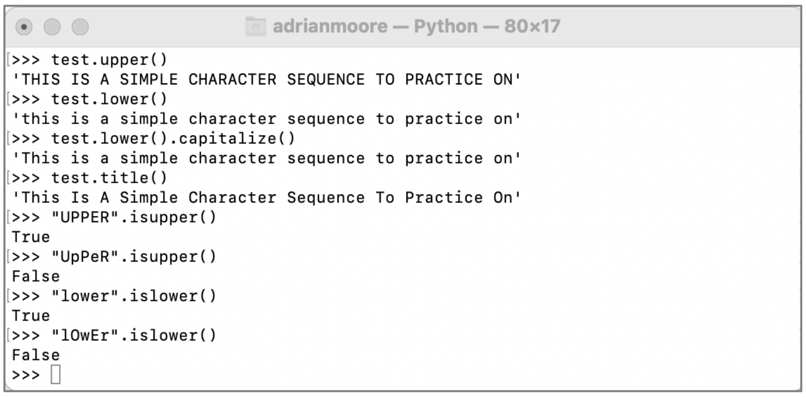


```
>>> words = test.split()
>>> words
['This', 'is', 'a', 'simple', 'character', 'sequence', 'to', 'practice', 'on']
>>> " ".join(words)
'This is a simple character sequence to practice on'
>>> "...".join(words)
'This...is...a...simple...character...sequence...to...practice...on'
>>> 
```

Figure A1.5 Joining strings

The **`join()`** method takes as a parameter a list of string elements, 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 A1.6 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
>>> 
```

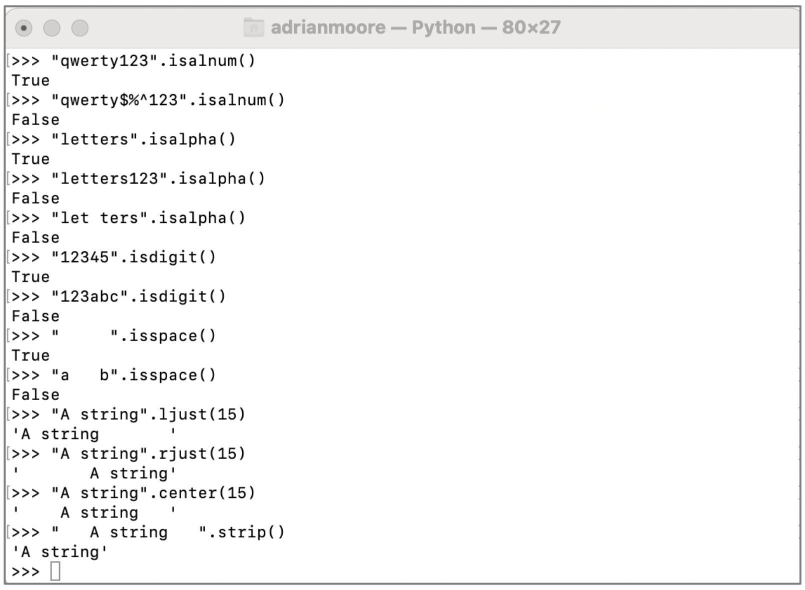
Figure A1.6 Case Manipulation

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 A1.7 below into the Python interpreter and verify their operation.



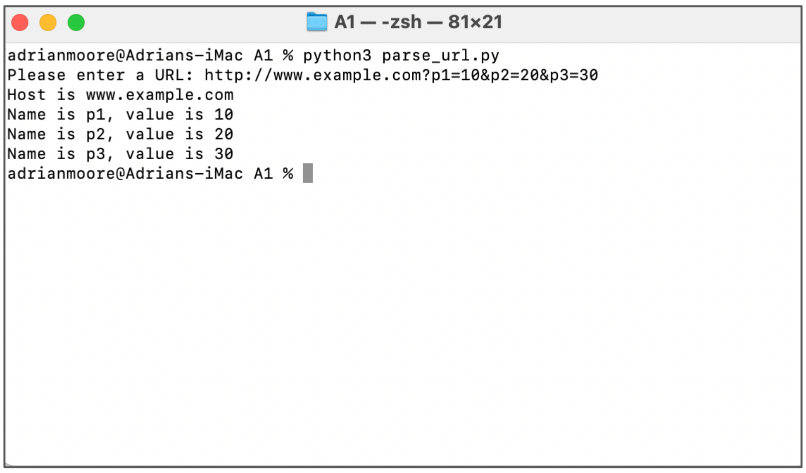
```
>>> "qwerty123".isalnum()
True
>>> "qwerty$%^123".isalnum()
False
>>> "letters".isalpha()
True
>>> "letters123".isalpha()
False
>>> "let ters".isalpha()
False
>>> "12345".isdigit()
True
>>> "123abc".isdigit()
False
>>> "     ".isspace()
True
>>> "a   b".isspace()
False
>>> "A string".ljust(15)
'A 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 querystring and extracts & prints its component parts. For example, the URL http://www.example.com?p1=10&p2=20&p3=30 should produce output such as that illustrated in Figure A1.8 below. **Hint** – your code should be generalized to work for any URL and any number of querystring parameters



```
adrianmoore@Adrians-iMac A1 % python3 parse_url.py
Please enter a URL: http://www.example.com?p1=10&p2=20&p3=30
Host is www.example.com
Name is p1, value is 10
Name is p2, value is 20
Name is p3, value is 30
adrianmoore@Adrians-iMac A1 %
```

Figure A1.8 URL Parser

A1.2 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: A1/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.

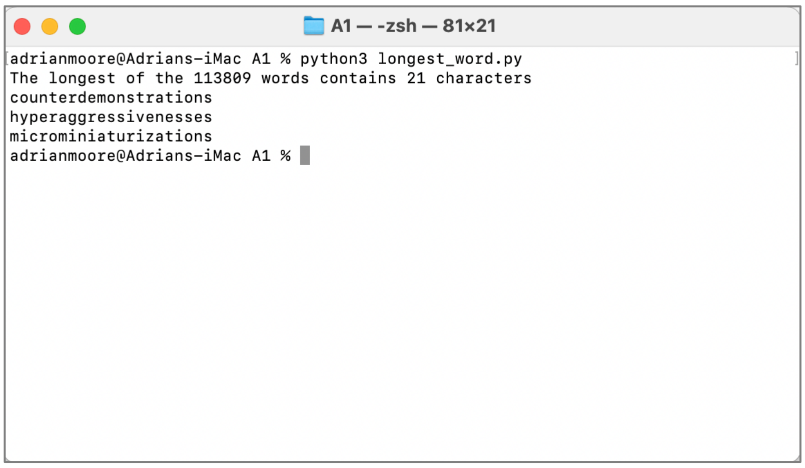
Once all words have been read, the **`for`** loop terminates and we report the total number of words read and the length of the longest word.

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 similar to that illustrated in Figure A1.9 below.



```
adrianmoore@Adrians-iMac A1 % python3 longest_word.py
The longest of the 113809 words contains 21 characters
counterdemonstrations
hyperaggressivenesses
microminiaturizations
adrianmoore@Adrians-iMac A1 % 
```

Figure A1.9 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.

A1.3 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: A1/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: A1/hangman.py

```
def get_words(number_of_letters):
   words_found = []
   fin = open("words.txt")
   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: A1/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: A1/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: A1/hangman.py

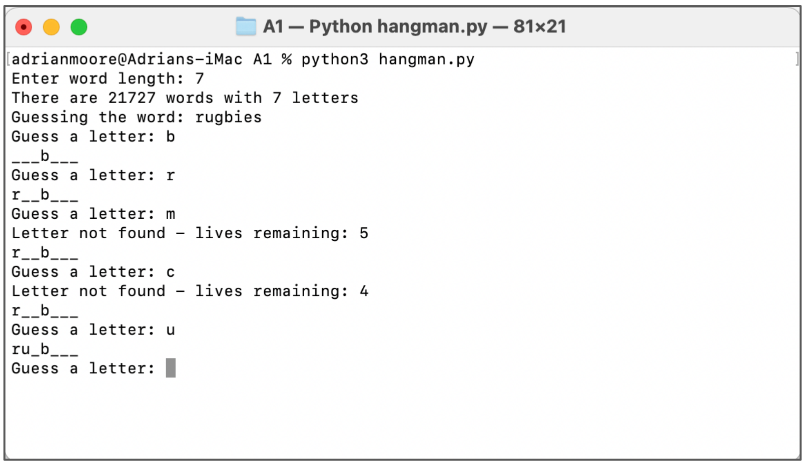
```
...

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*.

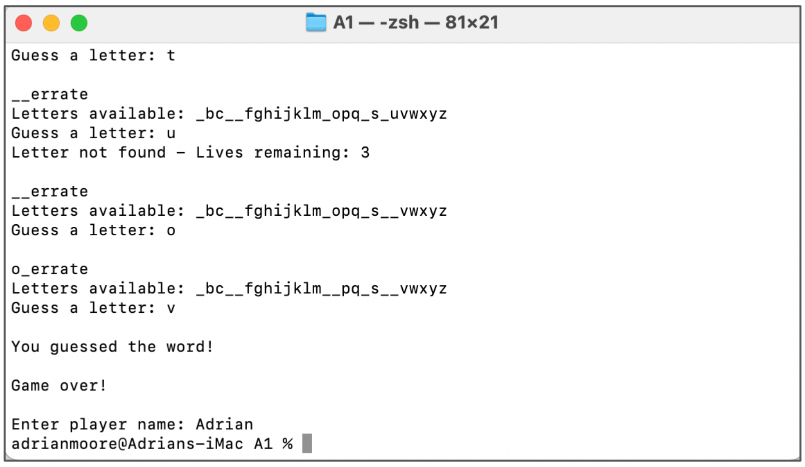
Do it now!	Execute the application *hangman.py* and see its operation. Comment out the line that prints the word to be guessed to get a "true" game. Perform multiple runs and see how the *hangman_scores.txt* file records details of all successful games.



```
adrianmoore@Adrians-iMac A1 % python3 hangman.py
Enter word length: 7
There are 21727 words with 7 letters
Guessing the word: rugbies
Guess a letter: b
___b___
Guess a letter: r
r__b___
Guess a letter: m
Letter not found - lives remaining: 5
r__b___
Guess a letter: c
Letter not found - lives remaining: 4
r__b___
Guess a letter: u
ru_b___
Guess a letter: 
```

Figure A1.10 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 A1.5 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.



```
Guess a letter: t

__errate
Letters available: _bc__fghijklm_opq_s_uvwxyz
Guess a letter: u
Letter not found - Lives remaining: 3

__errate
Letters available: _bc__fghijklm_opq_s__vwxyz
Guess a letter: o

o_errate
Letters available: _bc__fghijklm__pq_s__vwxyz
Guess a letter: v

You guessed the word!

Game over!

Enter player name: Adrian
adrianmoore@Adrians-iMac A1 %
```

Figure A1.11 Hangman Enhanced

A1.4 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://my.safaribooksonline.com/book/programming/python/0596001673/files/pythoncook-chp-4-sect-8
  Processing every word in a file

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