Lab 4: Text Processing

Text Processing

Human languages are stored as texts in computers, such as articles, news, novels, speeches, etc. Text processing is an important step to analyze the meaning of sentences or documents. Python can be used to process the huge amount of text data for the requirements in various textual data analysis. This lab shows some very preliminary text processing functions in Python. For a more comprehensive treatment on the topic, refer to the online Python documentation at:
https://docs.python.org/3/library/stdtypes.html#textseq and

https://docs.python.org/3/library/text.html

Background of Text Coding

Today's digital computers represent everything by binary numbers, i.e., 0 and 1. It's important to have global standards on how to represent information (such as numbers, texts, images, audio, video, etc.) in computers so that different computers can exchange information without misunderstanding. Below are some fundamental terminologies related to digitalized information:

Bit: A single 0 or 1 is called a bit.

Byte: Eight bits form a byte. There are 256 different bytes, from 00000000 to 11111111.

Word: Several bytes form a word (e.g., we may have 2-byte word, 4-byte word, or 8-byte word).

ASCII Code

In 1968, the ASCII (American Standard Code for Information Interchange) code was standardized. ASCII defined numeric codes from 0 to 127 for various characters, including ***digits*** 0 to 9, ***lowercase letters*** a to z, ***uppercase letters*** A to Z, and some widely used ***punctuation symbols***. In practice, an ASCII character occupies a single byte. Check out https://en.wikipedia.org/wiki/ASCII for the history and details of ASCII. However, ASCII code is designed for English characters only and cannot be used for other human languages.

Unicode

Unicode is a standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. The Unicode standard describes how characters are represented by different integers called **code points**. To be compatible with the popular ASCII, the first 128 code points (i.e., integers 0 to 127) in Unicode are the same as ASCII code. The latest version Unicode 11.0 contains more than 1 million code points that covers 146 modern and historic scripts, organized as a set of tables for different languages. But how to represent a code point (i.e., an integer ranged from 0 to more than 1million) by 0s and 1s? There are different solutions known as **character encodings**, such as ***UTF-8***, ***UTF-16***, and ***UTF-32***. **Python 3.x uses UTF-8 as the default encoding to represent characters and strings**.

UTF-8

UTF-8 is the dominant character encoding scheme of Unicode on the World Wide Web. It uses one byte for the first 128 code points (the same as ASCII), and **up to** 4 bytes for other characters. So any ASCII text is also a UTF-8 text, which is a nice feature because a software developed for UTF-8 can automatically process all existing ASCII-based documents. Chinese, Japanese and Korean characters are encoded in 3 bytes in UTF-8.

Summary: Unicode defines how to map a character to a code point (essentially an integer). Encoding schemes such as UTF-8, UTF-16, and UTF-32 define how to represent a code point by one or a few bytes for storage or communication purposes. In order to correctly process a plain text document, a software needs to know the exact encoding scheme.

Strings

In Python, a **string** is an **object** that stores a sequence of characters. Strings have the built-in data type **str** with many handy features. Notice that Python has no specific data type for characters (**chars**). A character is simply a string with length of 1. String literals can be enclosed by single, double or triple quotes, such as "Alice", 'Bob', etc. Python strings are **immutable** which means they cannot be changed after they are created.

Since a string consists a sequence of characters, characters in a string can be accessed using the standard [ ] syntax. Python uses zero-based indexing, so if **s** is 'hello', then **s[1]** is 'e'. If the index is out of bounds for the string, Python raises an error. The **len()** function returns the length of a string. The '+' operator can concatenate two strings.

```
name = "Alice"
type(name)
print(name[3])            #c
print(len(name))          #5
print(name + " Lee")      #Alice Lee
```

Basic Terminologies

Python follows the following rules of **character classification**:

whitespace = ' \t\n\r\v\f'

(Remark: Python whitespace includes space, tab, linefeed, return, vertical tab, and formfeed)

ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'

ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

ascii_letters = ascii_lowercase + ascii_uppercase

digits = '0123456789'

hexdigits = digits + 'abcdef' + 'ABCDEF'

octdigits = '01234567'

punctuation = r"""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""

printable = digits + ascii_letters + punctuation + whitespace

Remark: the above strings are defined in the **string** module of Python.

String methods in str class

The built-in **str** class has many methods for manipulating text. The commonly used methods include:

- **lower()**, **upper()**: returns the lowercase or uppercase version of the string. To know more about the functions, type **help(str.lower)** or **help(str.upper)** in the IPython console.

```
s = "HkbUCs"
print(s.upper())  #HKBUCS
print(s.lower())  #hkbucs
help(s.lower) #don't put () at the end of s.lower!
```

- **strip()**: returns a string with special char (as specified by the function parameter, or by default whitespaces) removed from the start and end.

```
s1 = "******HkbUCs******"
s2 = "      Computer    "
print(s1.strip('*'))      #HkbUCs
print(s2.strip())         #Computer
```

- **isalpha()**, **isdigit(),isspace(), isalnum(), islower(), isupper()**: tests if all the string chars are in the various character classes.
    - **isalpha():** Returns True if the string has at least 1 character and all characters are ***alphabetic***, and False otherwise
    - **isdigit()**: Returns True if the string contains only ***digits***, and False otherwise
    - **isspace()**: Returns True if the string contains only ***whitespace*** characters, and False otherwise
    - **isalnum()**: Returns True if the string has at least 1 character and all characters are ***alphanumeric***, and False otherwise
    - **islower()**: Returns True if the string has at least 1 cased character and all cased characters are in lowercase, and False otherwise
    - **isupper()**: Returns True if the string has at least 1 cased character and all cased characters are in uppercase, and False otherwise

```
s1 = "HkbUCs"
s2 = "1000"
s3 = "1000."
s3 = "           "
print(s1.isalpha())       #True
print(s2.isdigit())       #True
print(s3.isdigit())       #False
print(s4.isspace())       #True
```

- **startswith()**, **endswith()**: tests if the string starts or ends with the given other string.

```
s = "CS is fun!"
print(s.startswith("CS"))          #True
print(s.endswith("un!"))           #True
print(s.endswith("UN!"))           #Case sensitive, False
```

- **find()**: searches for the given other string within the original string, and returns the first index where it begins or -1 if not found.

```
s = "CS is fun!"
print(s.find("fun"))               #6
```

- **replace()**: returns a string where all occurrences of 'old' part have been replaced by 'new' part.

```
s = "CS is fun!"
t = s.replace("CS", "Computer Science")
print(t)    #Computer Science is fun!
```

- **split()**: returns a list of substrings separated by the given delimiter.

```
s = "CS is fun!"
words = s.split(' ')
print(words)                       #['CS', 'is', 'fun!']
```

- **join()**: opposite of split(), joins the elements in the given list together using the string as the delimiter.

```
s = ""
words = ['CS', 'is', 'fun', '!']
s = s.join(words)
print(s)                                   #CSisfun!
m = " ".join(words)
print(m)                                   #CS is fun!
```

The following example shows how to remove the punctuations from a sentence using **replace( )** and then split the sentence into a list of words using **split( )**:

```
myString = "He said, 'Hi! Stop! Stop!'"

import string # to access the string.punctuation

for c in string.punctuation:
    myString = myString.replace(c, '')

words = myString.split(' ') # split the sentence into words
print(words)
```

String Slices

The "slice" syntax is a handy way to refer to sub-parts of sequences (e.g., the substring of a string). The slice **s[start:end]** is the elements beginning at **start** and extending up to but **not** including **end**. For example, we have a string "Alice" stored in the variable *name*. With the slice syntax, we get:

- name[2:5] is "ice"; name[0:3] is "Ali"; name[2:] is "ice" too.

String Formatting (A Review of Lab1A – Example 2.2)

The string formatting operator % takes a printf-type format string on the left **(%d** for int, **%s** for string, **%f** for float), and the matching values in a tuple on the right: **format % values**

For example:

```
text = "%s! You have a meeting with %s at %d:%d pm." % ("Good Morning",
"Alice", 3, 30)
print(text)
# Good Morning! You have a meeting with Alice at 3:30 pm.
```

Alternatively, we can use the **format()** function to format a string. With this function, we do not need to specify the data types in the format string. We replace the %d, %s, %f with a set of braces.

For example:

```
text = "{0}! You have a meeting with {1} at {2}:{3} pm.".format(
"Good Morning", "Alice", 3, 30)
print(text)
# Good Morning! You have a meeting with Alice at 3:30 pm.
```

We can see that the result is the same as using % operator. Both formatters are great, there is no problem if you use either the % operator or the **format()** function.

More examples of the **format()** function:

```
'{0}, {1}, {2}'.format('a', 'b', 'c')   #a, b, c
'{}, {}, {}'.format('a', 'b', 'c')      #a, b, c
'{2}, {1}, {0}'.format('a', 'b', 'c')   #c, b, a
'{0}, {1}, {0}'.format('a', 'b', 'c')   #a, b, a
'{0}, {1}, {0}'.format('a', 'b')        #a, b, a
```

Exercise 1

Write a Python function that accepts a sentence and returns the first longest word. Assume the words in the string are separated by whitespaces.

Example:

Input = "Hello World Computer Science is fun"

Expected Output = “Computer”

Exercise 2

Write a Python program that accepts a string and *stepsize* which is an integer between 0 and 25. The function is able to create a Caesar encryption which transfers a character to the one which is *stepsize* away.

Caesar encryption example with *stepsize* of 3:

ABCDEFGHIJKLMNOPQRSTUVWXYZ
DEFGHIJKLMNOPQRSTUVWXYZABC
transforms “HELLO” to “KHOOR”

File I/O

Read a File

In Python, before accessing a file, we need to use built-in function ***open()*** to open it which returns a **file object**. ***open()*** usually takes two string parameters: ***filename*** and ***mode***. After opening a file successfully, you can read its data by ***read( )*** or ***readline( )*** functions through the file object. After accessing the file, use ***close( )*** to close the file. Study the following examples to learn how to use these functions.

Download the file demo.txt and put it in your python directory. Type the following statements:

```
file1 = open("demo.txt", "r") # file1 is a file object
content = file1.read() # read all data from file1 as a string
file1.close() # close file1. Now we cannot access demo.txt through file1
print(content)
```

There are four different modes for opening a file:

- “r”: opens a file for reading (default)
- “w”: opens a file for writing, truncating the file first (i.e., **the old data will be lost!**)
- “x”: creates a new file and opens it for writing
- “a”: opens a file for appending, creates a new file if the file does not exist.

By default, **read( )** will read data till the end of the file if you do not specify the range. We can also control how many bytes of data to read by providing an integer argument, i.e., **read(size)**:

```
file1 = open("demo.txt", "r") # open the file
s = file1.read(100) # read 100 bytes
print(s)
print()
s = file1.read(100) # read another 100 bytes
print(s)
file1.close()
```

For text file, we can use **readline( )** function read a line, e.g.,

```
file1 = open("demo.txt", "r")
s = file1.readline() # read a line from demo.txt, including the last '\n'
print(s)
print()
s = file1.readline() # read another line from demo.txt
print(s)
file1.close()
```

The following example shows how to read all lines from a file separately:

```
file1 = open("demo.txt", "r")
s = file1.readline()
while s != '': # when reach the end of the file, s should become empty ''
    print(s)
    s = file1.readline()
file1.close()
```

Another way to read lines from a file is through **readlines( )**, which save all lines in a file as a list of string. This is the most commonly used way to process a text file.

```
file1 = open("demo.txt", "r")
lines = file1.readlines()
file1.close()
print(len(lines)) # check the number of lines in demo.txt
for s in lines: # iterate the list of lines
    print(s)
```

We can also use **write( )** function to write data into a file opened with “w” or “x” or “a”:

```
file1 = open("demo.txt", "a") # "a" means we can append data at the end
file1.write("Now we have something new!")
file1.close()
```

Try the following example, to find out what is the difference between mode ***a*** and mode ***w***:

```
file1 = open("demo.txt", "w")
file1.write("Now we have something new!")
file1.close()
```

Exercise 3

Write a function **copy(x, y)** that can copy a file **x** into another file **y**. Use demo.txt as the input to verify your function.

Hint: use functions open( ), read( ), write( ), and close( ).