Computer Science I
Hiding Information

CSCI-141
Lecture 2 of 2

09/11/2017

1 Problem Statement

We have a series of text files that contain information we wish to hide. We want to remove a name or other possible identifying words from reports that will be distributed publicly.

To simplify matters we assume that a file has no more than one word per line. Suppose we have a file (called `words1.txt`) that contains the following text. Spaces are displayed explicitly.

```
␣word1
␣␣word2
␣␣␣word3
␣␣word4
␣word5
word6

word7
```

How can we read in this file? How can we hide the word of interest?

2 Solution Design and Analysis

2.1 Hiding Algorithm

Let's start with the hiding algorithm. We simply need to examine every word in a file, and if it matches the word we wish to hide, we will hide it by displaying three dashes (e.g. 'Maria' becomes '`---`').

```
def hide( file, hidden_word):
    for  current_word in file:
        if current_word == hidden_word:
            print( "---")
        else:
            print( current_word)
```

2.2 File Objects

This `file` parameter is a new type of thing, and it refers to a file opened for reading the contents. When we open a file, we get a reference to the opened file, ready for use in reading the contents.

The Python function, `open` takes a string representing the name of a file, opens that file, and returns a reference to the now-opened file. If we save the reference in a variable, then we can use the variable in a `for` loop, and close it afterwards. The `file` parameter in the code above is known as a *file object*.

2.3 Processing File Content

With a reference to an opened file, we can use a "*for line in file* loop pattern" to process the file content. We supply the file object as the subject of the `for` loop, and each cycle of the loop reads the next line in the file. When the loop sees the end of the file, the `for` loop terminates, and execution continues at the next line of code after the loop.

After processing the file content, it is good practice to close the file object. If the file object variable is `fd`, then the code to close the file is `fd.close()`.

```
fd = open( "words1.txt")
for line in fd:
   print( line)
fd.close()
```

We see the following.

```
word1

␣␣word2

␣␣␣word3

␣␣word4

␣word5

word6



word7
```

Notice all the spaces from the original *and* the additional blank lines. That's because `print` generates a **newline** which produces a new line of output, and also displays the newline character stored at the end of each line in the file.

While we could make the output look nicer by telling `print` to not generate a newline by using `print( line, end='')`, this approach has consequences for the solution to this problem. Even if there were no spaces next to a word in the file, there would still be a newline at the end, and that would make the text in the file a different string from the string we are seeking.

```
for line in open( "words1.txt"):   # open and process the file content
   print( line == 'word1')
```

We see the following:

```
False
False
False
False
False
False
False
False
```

We need a new feature of Python strings: `strip`. This function creates and returns a new string with all whitespace, including newlines, removed from the beginning and end of the string.

We invoke this function the same way we called `turtle` functions; we write `st.strip()`, where *st* denotes a string we want to strip.

```
for line in open( "words1.txt"):
   print( line.strip() == "word1")
```

We see the following, as expected.

```
True
False
False
False
False
False
False
False
```

2.3.1 Alternative Mechanism: with Statement

An alternative to using the open-close pairs of functions is the `with` block statement. The fragment below shows how to use this.

```
with open( file_name) as fd:
    hide( fd, hidden_word)
```

The `with` block opens the file and assigns the `fd` variable to the open file. The indented block of code then processes the open file object, and, when the block finishes executing, the `with` statement closes the file.

2.4 Program Solution

Putting these ideas together, we can write the Python function `hide` and a helper function that takes a file name rather than a file. A helper function is useful for two reasons here:

- The `hide` function will more closely resemble the hiding algorithm.

- When writing functions that operate on a specific kind of object, it is good practice to pass that kind of object as an argument rather than some other representation of that kind of object.

```
def hide( file, hidden_word):
    """
    hide: File String -> NoneType (IO: Display modified file)
    """
    for current_word in  file:
       if current_word.strip() == hidden_word:
           print( '---')
       else:
           print( current_word )

def hide_using_file_name( file_name, hidden_word):
    """
    hide_using_file_name: String String -> NoneType
    Effect: Display the file with modifications.
    """
    fd = open( file_name)
    hide( fd, hidden_word)
    fd.close()
```

The rest of the code in `hiding.py` is a main/test function that processes several files containing example words.

3 Linear Search and Time Complexity

Although we originally framed this as a problem of hiding a word, we can re-frame it as *searching* for every instance of a particular word.

Linear search is the process of visiting each element in a sequence in order (e.g. each character in a string), stopping when either we find what we seek, or we have visited every element in the sequence and not found a match.

What is the *time complexity* of hiding and linear search of a file?

We would need to count the number of times we execute the loop to read an element of the file and see whether the word of interest is there. Since Python's `for` loop processes a file line-by-line, the loop body executes once for each line of the file, and all the operations inside it repeat for each and every word in the file.

So, what would make the `hide` function take a really really long time? That would be a file with a huge number of lines. The N for the hiding problem is thus the number of lines in the file. We would state that the complexity of our hiding program is linear time over N, where N is the number of lines in the file.

4 Testing

In the approach shown in `hiding.py`, we run the function `hide_using_file_name` several times: once when the word being sought is not there; once when it occurs one time; and once when it occurs more than one time. Verification that the output is correct must be done manually.