CS1210 Computer Science I: Fundamentals
Homework 1: Handling Strings
Due Monday, October 2 at 11:59PM
(a partial first draft is due Friday, September 22 at 11:59PM)

Introduction

We've just completed our review of the most common Python datatypes, and you've been exposed to some simple operations, functions and methods for manipulating these datatypes. In this assignment, we're going to develop some code that relies greatly on the string datatype as well as all sorts of iteration. First, a few general points.

(1) This is a challenging project, and you have been given two weeks to work on it. **If you wait to begin, you will almost surely fail to complete it.** The best strategy for success is to work on the project a little bit every day. To help incentivize you to do so, we will provide preliminary feedback to a partial draft you will upload by the draft due date shown at the top of this page (more on this below).

(2) **The work you hand in should be only your own; you are not to work with or discuss your work with any other student.** Sharing your code or referring to code produced by others is a violation of the student honor code and will be dealt with accordingly.

(3) **Help is always available from the TAs or the instructor during their posted office hours.** You may also post general questions on the discussion board (although you should never post your Python code). **I have opened a discussion board topic specifically for HW1.**

Background

In this assignment we will be processing text. With this handout, you will find a file containing the entire text of *The Wind in the Willows*, a children's novel published in 1908. At some point during the course of this assignment, I will provide you additional texts for you to test your code on; updated versions of this handout may also be distributed as needed. You should think of this project as building tools to read in, manipulate, and analyze these texts.

The rest of these instructions outline the functions that you should implement, describing their input/output behaviors. As usual, you should start by completing the *hawkid*() function so that we may properly credit you for your work. **Test** *hawkid*() **to ensure it in fact returns your own hawkid as the only element in a single element tuple.** As you work on each function, test your work on the document provided to make sure your code functions as expected. Feel free to upload versions of your code as you go; we only grade the last version uploaded (although we do provide preliminary feedback on a draft version; see below), so this practice allows you to "lock in" working partial solutions prior to the deadline. Finally, some general guidance.

(1) You will be graded on both the **correctness** and the **quality** of your code, including the quality of your **comments**!

(2) As usual, respect the function signatures provided.

(3) Be careful with iteration; **always choose the most appropriate form of iteration (comprehension, while, or for)** as the function mandates. Poorly selected iterative forms may be graded down, even if they work!

(4) Finally, to incentivize getting an early start, you should **upload an initial version of your homework by midnight Friday, September 22** (that's one week from the start of the assignment). We will use the autograder to provide feedback on the first two functions, *getBook*()

and *cleanup*(), only. We reserve the right to deduct points from the final homework grade for students who do not meet this preliminary milestone.

def getBook(file):

This function should open the file named *file*, and return the contents of the file formatted as a single string. During processing, you should (1) remove any blank lines and, (2) remove any lines consisting entirely of CAPITALIZED WORDS. To understand why this is the case, inspect the *wind.txt* sample file provided. Notice that the frontspiece (title, index and so on) consists of ALL CAPS, and each CHAPTER TITLE also appears on a line in ALL CAPS.

def cleanup(text):

This function should take as input a string such as might be returned by *getBook*() and return a new string with the following modifications to the input:

Remove possessives, *i.e.*, "’s" at the end of a word;
Remove parenthesis, commas, colons, semicolons, hyphens and quotes (both single and double); and
Replace ’!’ and ’?’ with ’.’

A condition of this function is that it should be easy to change or extend the substitutions made. In other words, a function that steps through each of these substitutions in an open-coded fashion will not get full credit; write your function so that the substitutions can be modified or extended without having to significantly alter the code. Here’s a hint: if your code for this function is more than a few lines long, you’re probably not doing it right.

def extractWords(text):

This function should take as input a string such as might be returned by *cleanup*() and return an ordered list of words from the input string. The words returned should all be lowercase, and should contain only characters, no punctuation.

def extractSentences(text):

This function returns a list of sentences, where each sentence consists of a string terminated by a ’.’.

def countSyllables(word):

This function takes as input a string representing a word (such as one of the words in the output from *extractWords*(), and returns an integer representing the number of syllables in that word. One problem is that the definition of syllable is unclear. As it turns out, syllables are amazingly difficult to define in English!

For the purpose of this assignment, we will define a syllable as follows. First, we strip any trailing ’s’ or ’e’ from the word (the final ’e’ in English is often, but not always, silent). Next, we scan the word from beginning to end, counting each transition between a consonant and a vowel, where vowels are defined as the letters ’a’, ’e’, ’i’, ’o’ and ’u’. So, for example, if the word is "creeps," we strip the trailing ’s’ to get "creep" and count one leading vowel (the ’e’ following the ’r’), or a single syllable. Thus:

```
>>> countSyllables('creeps')
1
>>> countSyllables('devotion')
3
>>> countSyllables('cry')
1
```

The last example hints at the special status of the letter 'y', which is considered a vowel when it follows a non-vowel, but considered a non-vowel when it follows a vowel. So, for example:

```
>>> countSyllables('coyote')
2
```

Here, the 'y is a non-vowel so the two 'o's correspond to 2 transitions, or 2 syllables (don't forget we stripped the trailing 'e'). And while that's not really right ('coyote' has 3 syllables, because the final 'e' is not silent here), it does properly recognize that the 'y' is acting as a consonant.

You will find this definition of syllable works pretty well for simple words, but fails for more complex words; English is a complex language with many orthographic bloodlines, so it may be unreasonable to expect a simple definition of syllable! Consider, for example:

```
>>> countSyllables('consumes')
3
>>> countSyllables('splashes')
2
```

Here, it is tempting to treat the trailing -es as something else to strip, but that would cause 'splashes' to have only a single syllable. Clearly, our solution fails under some conditions; but I would argue it is close enough for our intended use.

def ars(text):

Next, we turn our attention to computing a variety of readability indexes. Readability indexes have been used since the early 1900's to determine if the language used in a book or manual is too hard for a particular audience. At that time, of course, most of the population didn't have a high school degree, so employers and the military were concerned that their instructions or manuals might be too difficult to read. Today, these indexes are largely used to rate books by difficulty for younger readers.

The Automated Readability Score, or ARS, like all the indexes here, is based on a sample of the text (we'll be using the text in its entirety).

```
http://www.readabilityformulas.com/automated-readability-index.php
```

The ARS is based on two computed paramters; the average number of characters per word (cpw) and the average number of words per sentence (wps). The formula is:

$$ARS = 4.71 * cpw + 0.5 * wps - 21.43$$

were the weights are fixed as shown. Texts with longer words or sentences have a greater ARS; the value of the ARS is supposed to approximate the US grade level. Thus a text with an ARS of 12 corresponds roughly to high school senior reading level.

def fki(text):

The Flesch-Kincaid Index, or FKI, is also based on the average number of words per sentence (wps), but instead of characters per word (cpw) like the ARS, it uses syllables per word (spw).

```
http://www.readabilityformulas.com/flesch-grade-level-readability-formula.php
```

The formula is:

$$FKI = 0.39 * wps + 11.8 * spw - 15.59$$

As with the ARS, a greater value indicates a harder text. This is the scale used by the US military; like with the ARS, the value should approximate the intended US grade level. Of course, as the FKI was developed in the 1940's, it was intended to be calculated by people who had no trouble counting syllables without relying on an algorithm to do so.

def cli(text):

The Coleman-Liau Index, or CLI, also approximates the US grade level, but it is a more recent index, developed to take advantage of computers.

```
http://www.readabilityformulas.com/coleman-liau-readability-formula.php
```

The CLI thus uses average number of characters per 100 words (cphw) and average number of sentences per 100 words (sphw), and thus avoids the difficulties encountered with counting syllables by computer.

$$CLI = 0.0588 * cphw - 0.296 * sphw - 15.8$$

Testing Your Code

I have provided a function, *evalBook*(), that you can use to manage the process of evaluating a book. Feel free to comment out readability indexes you haven't yet tried to use.

I've also provided three texts for you to play with. The first, 'test.txt', is a simple passage taken from the readbility formulas website listed above. The output my solution produces is:

```
>>> evalBook('test.txt')
Evaluating TEST.TXT:
  10.59 Automated Readability Score
  10.17 Flesch-Kincaid Index
   7.28 Coleman-Liau Index
```

The second, 'wind.txt', is the complete text to *The Wind in the Willows* by Kenneth Grahame. My output:

```
>>> evalBook('wind.txt')
Evaluating WIND.TXT:
   7.47 Automated Readability Score
   7.63 Flesch-Kincaid Index
   7.23 Coleman-Liau Index
```

as befits a book intended for young adults. Finally, 'iliad.txt', is an English translation of Homer's *Iliad*. My output:

```
>>> evalBook('iliad.txt')
Evaluating ILIAD.TXT:
  12.36 Automated Readability Score
  10.50 Flesch-Kincaid Index
   9.46 Coleman-Liau Index
```

which I think, correctly, establishes the relative complexity of the language used.

Python 3 Cheat Sheet

Official Python documentation on http://docs.python.org/py3k

Base Types

integer, float, boolean, string

```
  int  783    0    -192
float  9.23   0.0  -1.7e-6
 bool  True   False
  str  "One\nTwo"   'I\'m'
```

10^{-6} (for e-6); new line (for \n); ' escaped (for \')

multiline: `"""X\tY\tZ 1\t2\t3"""` — tab char (for \t)

str: immutable, ordered sequence of chars

Container Types

- ordered sequence, fast index access, repeatable values

```
 list  [1,5,9]  ["x",11,8.9]  ["word"]   []
tuple  (1,5,9)  11,"y",7.4    ("word",)  ()
```

tuple: *immutable*; `11,"y",7.4`: expression with just comas

`str` as an ordered sequence of chars

- no *a priori* order, unique key, fast key access ; keys = base types or tuples

```
dict  {"key":"value"}                                {}
      {1:"one",3:"three",2:"two",3.14:"π"}
set   {"key1","key2"}     {1,9,3,0}     set()
```

dict: dictionary, *key/value associations*

Identifiers

for variables, functions, modules, classes… names

`a..zA..Z_` followed by `a..zA..Z_0..9`

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

☺ `a toto x7 y_max BigOne`

☹ ~~`8y`~~ ~~`and`~~

Variables assignment

`x = 1.2+8+sin(0)` — x: variable name (identifier); `1.2+8+sin(0)`: value or computed expression

`y,z,r = 9.2,-7.6,"bad"` — `y,z,r`: variables names; `9.2,-7.6,"bad"`: container with several values (here a tuple)

`x+=3` increment

`x-=2` decrement

`x=None` « undefined » constant value

Conversions

`type(expression)`

```
int("15")                                   can specify integer number base in 2nd parameter
int(15.56)                                  truncate decimal part (round(15.56) for rounded integer)
float("-11.24e8")
str(78.3)                                   and for litteral representation → repr("Text")
```

see other side for string formating allowing finer control

`bool` → use comparators (with ==, !=, <, >, …), logical boolean result

```
list("abc")                               → ['a','b','c']      use each element from sequence
dict([(3,"three"),(1,"one")])             → {1:'one',3:'three'}
set(["one","two"])                        → {'one','two'}      use each element from sequence
":".join(['toto','12','pswd'])            → 'toto:12:pswd'
"words with  spaces".split()              → ['words','with','spaces']
"1,4,8,2".split(",")                      → ['1','4','8','2']
```

`":"`: joining string; `['toto','12','pswd']`: sequence of strings; `","`: splitting string

Sequences indexing

for lists, tuples, strings, …

negative index	-6	-5	-4	-3	-2	-1	
positive index	0	1	2	3	4	5	
`lst=[`	`11,`	`67,`	`"abc",`	`3.14,`	`42,`	`1968]`	
positive slice	0	1	2	3	4	5	6
negative slice	-6	-5	-4	-3	-2	-1	

`len(lst)` → 6

individual access to items via `[index]`

```
lst[1]→67        lst[0]→11     first one
lst[-2]→42       lst[-1]→1968  last one
```

access to sub-sequences via `[start slice:end slice:step]`

```
lst[:-1]→[11,67,"abc",3.14,42]       lst[1:3]→[67,"abc"]
lst[1:-1]→[67,"abc",3.14,42]         lst[-3:-1]→[3.14,42]
lst[::2]→[11,"abc",42]               lst[:3]→[11,67,"abc"]
lst[:]→[11,67,"abc",3.14,42,1968]    lst[4:]→[42,1968]
```

Missing slice indication → from start / up to end.

On mutable sequences, usable to remove `del lst[3:5]` *and to modify with assignment* `lst[1:4]=['hop',9]`

Boolean Logic

Comparators: `< > <= >= == !=` ($\leq$ $\geq$ $=$ $\neq$)

`a and b` logical and — *both simultaneously*

`a or b` logical or — *one or other or both*

`not a` logical not

`True` true constant value

`False` false constant value

Statements Blocks

```
parent statement:
    statements block 1…
    ⁝
    parent statement:
        statements block 2…
        ⁝
next statement after block 1
```

indentation !

Conditional Statement

statements block executed only if a condition is true

```
if logical expression:
    statements block
```

can go with several elif, elif... and only one final else, example :

```python
if x==42:
    # block if logical expression x==42 is true
    print("real truth")
elif x>0:
    # else block if logical expression x>0 is true
    print("be positive")
elif bFinished:
    # else block if boolean variable bFinished is true
    print("how, finished")
else:
    # else block for other cases
    print("when it's not")
```

Maths

floating point numbers… approximated values! *angles in radians*

Operators: `+ - * / // % **` (× ÷; `//` integer ÷; `%` ÷ remainder; `**` a^b)

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
```

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
acos(0.5)→1.0471…
sqrt(81)→9.0      √
log(e**2)→2.0
```

etc. (cf doc)

Conditional loop statement

statements block executed as long as condition is true

```
while logical expression:
    statements block
```

```
s = 0
i = 1
```
} *initializations* ***before*** *the loop*

condition with at least one variable value (here ***i****)*

```
while i <= 100:
    # statement executed as long as i ≤ 100
    s = s + i**2
    i = i + 1
```
} ☝ *make condition variable change*

$$s=\sum_{i=1}^{i=100} i^2$$

```
print("sum:",s)
```
} *computed result after the loop*

☝ *be careful of inifinite loops !*

Loop control

break *immediate exit*

continue *next iteration*

Iterative loop statement

statements block executed for each item of a container or iterator

```
for variable in sequence:
    statements block
```

Go over sequence's **values**

```
s = "Some text"
cnt = 0
```
} *initializations* ***before*** *the loop*

loop variable, value managed by **for** *statement*

```
for c in s:
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Count number of ***e*** *in the string*

loop on dict/set = loop on sequence of keys

use slices to go over a subset of the sequence

Go over sequence's **index**

- modify item at index
- access items around index (before/after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Limit values greater than 15, memorization of lost values.

Go simultaneously over sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

Display / Input

```
print("v=",3,"cm :",x,",",y+4)
```

items to display: litteral values, variables, expressions

print options:

- `sep=" "` (items separator, default space)
- `end="\n"` (end of print, default new line)
- `file=f` (print to file, default standard output)

```
s = input("Instructions:")
```

☝ **input** always returns a **string**, convert it to required type (cf boxed *Conversions* on on ther side).

Operations on containers

Note: For dictionaries and set, these operations use ***keys****.*

`len(c)` → items count

`min(c)` `max(c)` `sum(c)`

`sorted(c)` → sorted copy

`val in c` → boolean, membersihp operator `in` (absence `not in`)

`enumerate(c)` → *iterator* on (index,value)

Special for ***sequence containeurs*** *(lists, tuples, strings) :*

`reversed(c)` → reverse *iterator* `c*5` → duplicate `c+c2` → concatenate

`c.index(val)` → position `c.count(val)` → events count

Generator of int sequences

frequently used in **for** *iterative loops*

default 0 — not included

range(*[start,]stop [,step]*)

`range(5)` → 0 1 2 3 4

`range(3,8)` → 3 4 5 6 7

`range(2,12,3)` → 2 5 8 11

range returns a « generator », converts it to list to see the values, example:

```
print(list(range(4)))
```

Operations on lists

☝ modify original list

`lst.append(item)`	add item at end
`lst.extend(seq)`	add sequence of items at end
`lst.insert(idx,val)`	insert item at index
`lst.remove(val)`	remove first item with value
`lst.pop(idx)`	remove item at index and return its value
`lst.sort()` `lst.reverse()`	sort / reverse list *in place*

Function definition

function name (identifier) — named parameters

```
def fctname(p_x,p_y,p_z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

result value of the call. if no computed result to return: `return None`

☝ parameters and all of this bloc only exist *in* the block and *during* the function call (*"black box"*)

Operations on dictionaries

`d[key]=value` `d.clear()`

`d[key]`→*value* `del d[clé]`

`d.update(d2)` } *update/add associations*

`d.keys()`
`d.values()`
`d.items()` } *views on keys, values associations*

`d.pop(clé)`

Operations on sets

Operators:

`|` → union (vertical bar char)

`&` → intersection

`-` `^` → difference/symetric diff

`<` `<=` `>` `>=` → inclusion relations

`s.update(s2)`

`s.add(key)` `s.remove(key)`

`s.discard(key)`

Function call

```
r = fctname(3,i+2,2*i)
```

one argument per parameter

retrieve returned result (if necessary)

Files

storing data on disk, and reading it back

```
f = open("fil.txt","w",encoding="utf8")
```

file variable for operations	name of file on disk (+path…)	opening mode	encoding of chars for text files:
		`'r'` read	`utf8` `ascii`
		`'w'` write	`latin1` …
		`'a'` append…	

cf functions in modules `os` and `os.path`

writing

```
f.write("hello")
```

☝ *text file → read /write only* ***strings****, convert from/to required type.*

reading

```
s = f.read(4)
```

empty string if end of file; if char count not specified, read whole file

read next line

```
s = f.readline()
```

```
f.close()
```
☝ don't forget to close file after use

Pythonic automatic close : `with open(…) as f:`

very common: iterative loop reading lines of a text file

```
for line in f :
    # line processing block
```

Strings formatting

formating directives — values to format

```
"model {} {} {}".format(x,y,r) → str
"{selection:formating!conversion}"
```

- **Selection** :
  `2`
  `x`
  `0.nom`
  `4[key]`
  `0[2]`

Examples:

```
"{:+2.3f}".format(45.7273)
→'+45.727'
"{1:>10s}".format(8,"toto")
→'      toto'
"{!r}".format("I'm")
→'"I\'m"'
```

- **Formating** :

fillchar alignment sign minwidth.precision~maxwidth type

`< > ^ =` `+ - space` `0` at start for filling with 0

integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…

float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default), `%` percent

string : `s` …

- **Conversion** : `s` (readable text) or `r` (litteral representation)