CPSC231 Tutorial 6-1

FANG WEI

Introduction

The problem with tabs and spaces:

Python is a fairly unusual language in that it uses indentation as part of it's syntax. But tabs and space characters can complicate things.

A single tab may look identical to a sequence of spaces when the program is printed or displayed on the screen, but the Python interpreter may see the two as very different levels of indentation. This issue can lead to very difficult-to-find bugs in programs.

What we need are tools to 1) allow us to actually see the tabs and spaces as printable characters, and 2) tools to detect and intelligently convert tabs to spaces or spaces to tabs. The second problem is compounded because tabs can be "equivalent to" any number of spaces (typically 2, 3, 4, or 8 spaces are "equivalent to" a tab).

Full Assignment 3

Write a multi-functional UNIX-style utility program to help visualize and fix problems with tabs and spaces in Python programs. This program will process python program files in the following ways:

1. Change tabs in the indenting to spaces
2. Change spaces in the indenting to tabs
3. Substitute spaces, tabs, and newlines for printable characters, maintaining formatting
4. Undo 3.

Obviously, these 4 functions will not always be used at the same time. Therefore, we will use command-line qualifiers to allow the user to specify the subset of functionalities he/she wants. We do this in UNIX-eze, where we use a minus sign to introduce a short qualifier. If our program is called "tabs" and [] represents optional:

tabs [+t] [-t] [-T<integer>] [+v] [-v] [-help]

Qualifier +t -t

+t : replaces prefix sequences of spaces of length T with a single tab

-t : replaces prefix tabs with sequences of T spaces

```
# . stands for space and —— stands for tab
% python3 tabs.py +t
        pass            # input: 8-space indent (which is 2 default 4-space tabs)
————    pass    # output: 2 tabs (8 spaces each on the terminal)
    pass            # input: 6-space indent
——  pass            # output: a tab and 2 spaces
  —pass             # input: 2 spaces and a tab
——  pass            # output: 1 tab
```

```
% python3 tabs.py -t
————————pass    #input: 2 tabs
        pass    #output: 8 spaces
————  pass      #input: 1 tab and 2 spaces
      pass      #output:6 spaces
  —pass         #input:2 spaces and a tab
    pass        #output: 4spaces
```

Note that in these examples, the terminal has an 8-space tab (usually the default for terminal programs), whereas the tabs program has a default 4-space tab.
. and — are displayed by the editor, not printable

Qualifier +v -v

+v : changes all spaces, tabs, and newlines to printable (visible) characters

-v : undoes the effects of +v

```
% python3 tabs.py +t +v
        pass
» » pass¶
    pass
» ··pass¶
  pass
» pass¶
% python3 tabs.py -t +v
  pass      # input: 2 tabs (8 spaces each on the terminal)
········pass¶
  pass        # input: a tab and 2 spaces
··pass¶
  pass          # input: 2 spaces and a tab
····pass¶
%
```

Qualifier -T<integer> -help

-T<integer>: the <integer> defines the space-to-tab ratio, T (default=4)

```
1  % python3 tabs.py +t +v -T8
2          pass# input: 8-space indent (now default 8-space tabs)
3  » pass¶# output: 1 tabs
4        pass# input: 6-space indent
5  ····pass¶# output: 6-space indent  (6<8)
6      pass# input: 2 spaces and a tab
7  » pass¶# output: 1 tab
```

-help -prints out this help text

E.g.: python3 tabs.py –help

+t and -t are incompatible

+v and -v are incompatible

Requirements

General Requirements

Lots of them on the webpage. Read it carefully. If we have time left, we can go through them together.

Non-functional Requirements

Your program must be written in Python 3 and function properly on the UNIX machines in the CPSC undergrad lab.

This assignment is about functions and program decomposition, so besides yourmain() function, and the getInput() function (given below), it is expected you will create AT LEAST 4 other functions (at least one function for each of the -v, +v, -t, and +t functionalities). (The instructors' solution makes use of 8 additional functions, which is not to say that yours should necessarily use 8 -- it could use more or less.)

Functions

Define a function

Format:

```
def <function name>():
    body
```

Example:

```
def displayInstructions():
    print ("Displaying instructions on how to use the
      program")
```

Call a function

Format:

```
<function name>()
```

Example:

```
displayInstructions()
```

Hints (1)

This section is meant to help you with your program. You CAN cut-and-paste the code given here into your code without citing it without fear of penalty. Do not cut-and-paste any other code without citing though!

Defining your program

Since this is a fairly complex program, your TA may use a testing harness to verify that your program will run correctly. Therefore you must make it possible for another program to include and run your program, so all your code that you normally run when you invoke your program from the command line should be embedded within a main program function. In addition, you want to leave calling the main program to any test harness, but still call your program if the file in invoked directly from the command line. To do that, use the following paradigm:

```
1  # imports go here
2
3  # global constants go here
4
5  # all your function definitions go here
6
7  def main(): # Or start()
8      # Body of main() function goes here
9  main()
```

Hints(2)

Command-line input

You will need to gather the user-specified qualifiers from the command line. To do that you need to use sys.argv from the system library. To import from the system library use the following import statement:

import sys # needed to collect command-line arguments (sys.argv)

sys.argv is a sequence containing the command line arguments. The first element of sys.argv is the program name, and the remainder are the "words" (arguments) the user typed after the program name. These do NOT include expressions (such as redirections ["<", ">", and ">>"] and pipes ["|"]) interpreted by the command-line processor. Thus, the paradigm for reading command lines is as follows:

```
1  firstArg = True
2  for arg in sys.argv:
3      if firstArg: # the first argument is always the program name, so ignore it
4          firstArg = False
5      elif (arg=="-t"):
6          ...
7      ...
8      else: #if we got here, then we didn't recognize the argument
9          ...
```

Hints(3)

Input

The standard Python input() function does not handle EOF (End-Of-File) (the ctrl-D character) gracefully: If you type a ctrl-D in response to an input(), it will throw an exception (i.e., a run-time error and 'crash') and your program will terminate. The problem is easily solved, but we have not covered that yet. So use the following function in lieu of the standard input() function:

```
1  def getInput():
2      """This function works exactly like input() (with no arguments), except that,
3      instead of throwing an exception when it encounters EOF (End-Of-File),
4      it will return an EOF character (chr(4)).
5      Returns: a line of input or EOF if EOFError occurs during input.
6      """
7      try:
8          ret = input()
9      except EOFError:
10         ret = EOF
11     return ret
```

You might not understand this code since it includes concepts that we haven't covered yet (such as exceptions). That's okay for now -- just make sure you understand how to use it. Also note that unlike Python's input() it does not take any parameters: That's OK because this program *might* be taking it's input from a file, we *should not* prompt for each line of input..

Hints(4)

Substitutions for tab, space and newline:

The "visible" forms of the newline, tab, and space characters require the use of the chr() function. This function takes an integer as a parameter (which should be an ASCII code) and returns the character for that code. For example chr(65) would return capital letter 'A' because the ASCII code for this character is 65. You can refer to the program [chr_example.py] to get a feel for how this function works.

There are substitution characters for tab, space, and newline, as specified in requirement D1, D2 and D3. The getInput() function also requires an EOF character. These can be defined as constants as follows:

```
1  EOF = chr(4)          # A standard End-Of-File character (ascii value 4)
2  TAB_CHAR = chr(187)    # A ">>" character (as a single character) in the extended ascii set
3                         #   Used to make a tab character visible.
4  SPACE_CHAR = chr(183)  # A raised dot character in the extended ascii set
5                         #   Used to make a space character visible
6  NEWLINE_CHAR = chr(182) # A backwards P character in the extended ascii set
7                         #   Used to make a newline character visible
```

Using pre-written Python libraries

For this assignment, you can use any of the Python built-in functions. You may also use any of the build-in string methods; you might find

<str>.replace(),

<str>.lstrip(),

<str>.expandtabs(),

<str>.upper(),

'slicing' [<start> : <end>] operator particularly useful.

<str>.replace()

```
#str.replace(old, new[, max])
#old -- This is old substring to be replaced.
#new -- This is new substring, which would replace old substring.
#max -- If this optional argument max is given, only the first count occurrences are replaced.
str = "this is string example....wow!!! this is really string"
print (str.replace("is", "was"))
print (str.replace("is", "was", 3))
"""
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
"""
```

<str>.lstrip()

```
"""
str.lstrip([chars])
This method returns a copy of the string in which all chars
have been stripped from the beginning of the string
(default whitespace characters).
"""
str = "     this is string example....wow!!!     "
print (str.lstrip());
str = "88888888this is string example....wow!!!8888888"
print (str.lstrip('8'));
#output
"""
this is string example....wow!!!
this is string example....wow!!!8888888
"""
```

<str>.expandtabs(),

```
"""
str.expandtabs(tabsize=8)
This method returns a copy of the string in which tab characters
i.e., '\t' have been expanded using spaces.
tabsize -- This specifies the number of characters to be replaced for a tab character '\t'.
"""
str = "this is\tstring example....wow!!!"
print ("Original string: " + str)
print ("Defualt exapanded tab: " +  str.expandtabs())
print ("Double exapanded tab: " +  str.expandtabs(16))
#output
"""
Original string: this is        string example....wow!!!
Defualt exapanded tab: this is string example....wow!!!
Double exapanded tab: this is         string example....wow!!!
"""
```

<str>.upper(),

```
"""
str.upper()
This method returns a copy of the string in
which all case-based characters have been uppercased.
"""
str = "this is string example....wow!!!"
print ("str.capitalize() : ", str.upper())
#output
"""
THIS IS STRING EXAMPLE....WOW!!!
"""
```

'slicing' [<start> : <end>]

```
1  var="hello world"
2  #str[start:end]
3  print (var[0])
4  print (var[1:5]) #not including end
5  print (var[2:]) # from 2 to the end
6  print (var[:5]) # from beginning to 5-1
7  #output
8  """
9  h
10 ello
11 llo world
12 hello
13 """
```

Run your program

The program will take it's input from standard input (i.e., the user), and output it's results to standard output (i.e., the screen). That will allow us to type in python commands from the console and get the results back immediately after we finish typing a line. The program continues to read input until it encounters EOF (End-Of-File), which a a ctrl-D character on UNIX, Linux, and Mac (and, I believe, a ctrl-Z-return sequence in Windows).

But this kind of program is most useful when it can read a file and place it's output in a different file. So typically, this program will be run with redirected ("<") input from a file, and possibly redirected (">") output to a file. These two command-line operators temporarily redirect input (output) so that it appears you typed in a file (or the output goes to a file, respectively).

For example:

```
$ python3 tabs.py +v +t -T4 < A3.py > temp.out        # 4 spaces becomes 1 tab (results in temp.out)
```

will change prefix space sequences of length 4 to tabs in the source file of this program, and save it in temp.out. Then, if we do:

```
$ python3 tabs.py -v -t -T4 < temp.out > temp2.out    # 1 tab becomes 4 spaces (results in temp2.out)
```

we should find (assuming the original file was properly indented by 4s with spaces only) that:

```
$ diff tabs.py temp2.out
```

finds no differences (diff is a UNIX utility that can be used to determine if there are any 'differences' between files).

Submitting your work:

1. Assignments (the source code/'dot-py' file) must be electronically submitted according to the assignment submission requirements using D2L.

2. As a reminder, you are not allowed to work in groups for this class. Copying the work of another student will be regarded as academic misconduct (cheating). For additional details about what is and is not okay for this class please refer to the notes on misconduct for this course.

3. Before you submit your assignment , go through the checklist of items to be used in marking online.