COMP202 Assignment 2 - First Half

First half due: Oct 17 at 23:59; **Second half due**: Oct 24 at 23:59
This is an individual assignment.

A Braille Translator

In this assignment, we will be writing a Braille translator. Braille is a writing system for blind people, created by blind inventor Louis Braille. It was developed in the early 19th century, for representing French text. His system has since been adapted for writing many other languages.

Because English Braille is a bit more complex from a computational perspective, and because Montréal is a bilingual city, we'll start by writing a French Braille translator.

Computers represent text through *character encodings*: a table associates each character with a unique binary number. ASCII, for instance, is an early character encoding system which is used to represent the English alphabet and its common punctuation and special characters. Unicode was later developed to support all the common writing systems of the world, including Braille.

Braille also uses a binary system to represent characters: whether a dot is raised or not. Indeed, it's generally seen as the earliest binary character encoding system — predating ASCII by 139 years!

Instructions

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, which allows the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while he or she is grading, then that increases the chance of them giving out partial marks. :)

Up to 30% can be removed for bad indentation of your code as well as omitting comments, or poor coding structure.

To get full marks, you must:

- Follow all directions below
  - In particular, make sure that all function and variable names are **spelled exactly** as described in this document. Else a 50% penalty will be applied.
- Make sure that your code runs.
  - Code with errors will receive a very low mark.
- Write your name and student ID as a comment in all .py files you hand in
- Name your variables and helper functions appropriately
  - The purpose of each variable should be obvious from the name
- Comment your work
  - A comment every line is not needed, but there should be enough comments to fully understand your program

Errata and Frequently Asked Questions

On MyCourses we have a discussion forum titled **Assignment 2**. A thread will be pinned in the forum with any errata and frequently asked questions. If you are stuck on the assignment, start by checking that thread.

What To Submit

This assignment will be submitted in two stages. The first half (Parts 1-3) are due Oct 17. The second half (Parts 4-5) are due Oct 24.
For both submissions: put all your files in a folder called Assignment2. Zip the folder (DO NOT RAR it) and submit it in MyCourses. If you do not know how to zip files, please ask any search engine or friends. Google will be your best friend with this, and a lot of different little problems as well.

Inside your zipped folder, there must be the following files. **Do not submit any other files.** Any deviation from these requirements may lead to lost marks.

1. `helpers.py` - submit Oct 17 **and** Oct 24
2. `char_to_braille.py` - submit Oct 17 **and** Oct 24
3. `to_unicode.py` - submit Oct 17 **and** Oct 24
4. `text_to_braille.py` - submit Oct 24
5. `english_braille.py` - submit Oct 24
6. `README.txt` - submit Oct 17 and Oct 24 In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit.

   This file is also where you should make note of anybody you talked to about the assignment. Remember this is an individual assignment, but you can talk to other students using the **Gilligan's Island Rule**: you can't take any notes/writing/code out of the discussion, and afterwards you must do something inane like watch television for at least 30 minutes.

   If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.

Learning Braille

For this assignment, you'll need to learn some basic Braille. To start, watch this video on the basics of English Braille: `https://www.youtube.com/watch?v=sqQ3gdE7ks0`

In Braille, characters are represented by *cells*. A *cell* generally represents one letter. A cell contains six (or sometimes eight) positions for dots, in two columns. At each position, there could be a raised (shaded) dot, or left flat (empty).

French Braille

All Braille systems are organized into *decades*: ordered groups of 10 letters. Each position in the *decade* has an associated *pattern*. See the YouTube video for more on the patterns and a mnemonic for memorizing them.

English Braille has two and a half *decades* for its letters, French has four. When Braille was developed in the 19th century, the letter 'w' was not commonly used in French. This is why 'w' is not where you'd expect it in the alphabet in either system.

The 40 letters make up a table of four rows and ten columns. We will be counting from zero for the rows and columns, since Python and other programming languages work this way.

	0	1	2	3	4	5	6	7	8	9
0	⠁ a / 1	⠃ b / 2	⠉ c / 3	⠙ d / 4	⠑ e / 5	⠋ f / 6	⠛ g / 7	⠓ h / 8	⠊ i / 9	⠚ j / 0
1	⠅ k	⠇ l	⠍ m	⠝ n	⠕ o	⠏ p	⠟ q	⠗ r	⠎ s	⠞ t
2	⠥ u	⠧ v	⠭ x	⠽ y	⠵ z	⠯ ç	⠿ é	⠷ à	⠮ è	⠾ ù
3	⠡ â	⠣ ê	⠩ î	⠹ ô	⠱ û	⠫ ë	⠻ ï	⠳ ü	⠪ œ	⠺ w

Decade 0 is also used to represent the digits 1-9 and 0, in that order. So 1 is the same as 'a', etc.

Finally, there is another decade to represent the most common punctuation, where we take the *decade*-pattern and shift it all downward. Note the most common punctuation in French Braille is different from English Braille (in the video).

	0	1	2	3	4	5	6	7	8	9
4	⠂ ,	⠆ ;	⠒ :	⠲ .	⠢ ?	⠖ !	⠶ ”	⠦ (	⠔ *	⠴)

Finally, for this assignment, there are six more important characters in French to know about which do not follow the *decade* pattern. Note the capital symbol in French Braille is different from the English Braille one in the video.

⠀	⠤	⠄	⠶	⠨	⠼
space	hyphen/dash	apostrophe	guillemets	capital	number

1 Warmup [0 points]

For us to be converting text to Braille, we'll need a few helper functions. Download `helpers.py`, put your name at the top, and complete these functions:

1. `is_in_decade`: given two strings, determine if the first one is contained in the second one, and that the first string is a single character. This will be useful in the future where we are checking if a character is part of a given *decade*.
2. `is_irregular`: given a string, determine if it's a single character that is one of the irregular characters listed in the pre-defined global variable `IRREGULAR_CHARS`. These "irregular" characters are called as such because they don't follow regular patterns like the other Braille characters we'll see. This will only return True if it's a single character, so "- -" would return False.
3. `is_digit`: given a string, determine if it's a single character representing a digit (0-9). We provide the global variable `DIGITS`.
4. `is_punctuation`: given a string, determine if it's a single character that is one of the punctuation marks that follow the regular Braille patterns. These have been pre-defined for you in the global variable `PUNCTUATION`.

Then have a look at the function `is_letter` we provided. It tests whether a string is a single character that is one of the regular French Braille letters. You should not edit this function. It is provided for you to call this function elsewhere.

Then complete:

5. `is_known_character`: given a string, determine if it's a single character representing one of the letters, digits or punctuation supported by our Braille translator.
6. `is_capitalized`: return whether a string is a single capitalized letter supported by our Braille translator.

2 Single Characters to French Braille [30 points]

In this assignment we'll be converting four types of characters to Braille: letters (A-Z & a-z), digits (0-9), regular punctuation, and irregular characters. You may want to review page 3 for this.

Download `char_to_braille.py` and put your name at the top. Complete this function:

1. `convert_irregular` [5 points]: given a character that is one of the irregular characters, return it in French Braille. We'll be using `o` to represent a raised dot and `.` to represent an empty position, and putting a new line in between each row of the Braille cell. A hyphen hence becomes:

   ```
   ..
   ..
   oo
   ```

   Return None if the input is not an irregular character. Note: there are multiple characters that correspond to apostrophe and hyphen (see docstring).

Then take a moment to note the function `decade_pattern`, which gives the pattern associated with each column of the Braille *decades*. A "pattern" is the set of what dots are raised in the top two rows of the 0th decade. You should not edit this function. It is provided for you to call elsewhere.

Then, complete:

2. `convert_digit` [5 points]: given a string representing a digit, convert it to Braille. Return None if the string is not a digit.
3. `convert_punctuation` [5 points]: given a string representing one of the regular forms of punctuation, convert it to French Braille. Return None if the string is not one of the standard punctuation in `PUNCTUATION`.
4. `decade_ending` [5 points]: return the associated bottom row for a given *decade*, from the table on page 3. (0: '..', 1: 'o.', 2: 'oo', 3: '.o')

Then read the docstrings of these two functions that we give you. Do not edit these functions; they are provided for you to call elsewhere.

- `letter_row` given a letter, returns which of the four letter *decades* it belongs to (zero-indexed). See the table on page 3.
- `letter_col` given a letter, returns its position (zero-indexed) within its *decade*. See the table on page 3.

Then complete:

5. `convert_letter` [5 points]: given a letter (upper or lower case), convert it to French Braille. Return None if it is not a letter in French Braille.
6. `char_to_braille` [5 points]: given a character, convert it to French Braille if we know how. If it's a letter we don't support, return the letter unchanged.

3 Converting Braille Representations

Text can be represented, or *encoded* in many different ways. Our goal in this part of the assignment is to convert from how we have been representing braille characters so far into Unicode, which is how we represent text on most of the Internet.

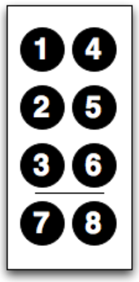
3.1 Different representations

In this assignment, we will use five ways to represent a letter such as ⠝ (n).

1. The **o-string representation**. This is what we've been using so far, where we represent a raised dot with an o and a lowered with a ., and new line characters to indicate new rows of the braille cell. So ⠝ is:

   ```
   oo
   .o
   o.
   ```

2. The **raised-position representation**. Every position in a braille *cell* has a standardized number. The positions are numbered as such:

   We can then list how many of the positions have a raised dot. For ⠝ that would be 1, 4, 5, and 3; or, the string '1453'.

   Traditionally, braille uses six dots, which is what we have been working with so far. However for academic/mathematical notation an eight-dot system is often used to expand the range of possible symbols.

3. The **binary representation**. Based on the position numbers above, if we have a raised dot we place a 1; if lowered, 0. So '1453' becomes '10111000'.

   For this we will always use an 8-bit representation. This is because Unicode represents 8-dot braille cells rather than the tradional 6-dot cells. To turn a 6-dot cell into an 8-dot cell, add two unraised dots to the end.

4. The **hex representation**. This is a hexadecimal number that corresponds to the Unicode representation (see below), such as 2813_{16} for h (⠓).

5. The **Unicode representation**. Unicode is the most popular way of encoding text on computers today. It also supports 8-dot Braille. Each character in Unicode has an associated hexadecimal number, whether it's 'n' (0144_{16}), 'न' (0928_{16}) or the heart emoji (2764_{16}).

For us to go from o-string to Unicode, we will convert in multiple stages:
o-string → raised-pos → binary → hex → Unicode

3.2 Code to Complete [20 points]

Download the file `to_unicode.py` and put your name at the top. Complete these functions:

1. `raisedpos_to_binary` [5 points]: convert a string in raised-position format to binary format (see previous page). Do not worry about handling strings that are not in raised-position format.

2. `binary_to_hex` [5 points]: convert a binary representation of a Braille character to its unique hex number. See the docstring for how.
   Note: this will require the built-in function `hex`, which takes in a base-10 number and converts it to a string in hexadecimal that starts with '0x'. For example, `hex(26)` returns the string '0x1a' rather than just '1a'.

3. `is_ostring` [5 points]: return whether the input string is formatted like a Braille o-string. Both 6-dot and 8-dot o-strings should return True.

4. `ostring_to_unicode` [5 points]: if we have an o-string, using the functions in the to_unicode module, convert from o-string to Unicode. Otherwise, return the string unchanged. Remember we wrote some functions in this module for you to use.

4 Text to French Braille [20 points]

So far we've been converting single characters to Braille. It is time for us to start converting words and whole sentences!

Download the file `text_to_braille.py` and put your name at the top.

Create a folder `tests/` in the directory you have your Python files. Download to this folder: `test1.txt` through `test6.txt` and `expected1.txt` through `expected6.txt`.

First, complete this function in `text_to_braille`:

1. `new_filename` [5 points]: given a filename with extension (e.g "file.txt") append a provided string (e.g. 'braille') before the filename extension (e.g. "file_braille.txt"). You can assume the given filename will always have an extension (".csv") and at most one period in the whole filename.

Then scroll up within `text_to_braille` and complete:

2. `is_all_caps` [5 points]: return whether a string is comprised of only capitalized letters that are supported by our Braille translator, and if there are at least two such letters.

3. `word_to_braille` [10 points]: convert a word into o-string French Braille. Note:
   (a) To separate the o-string Braille cells in a word, we put two new lines in between them.
   (b) Each digit should have ⠼ added before it. French Braille does this differently than what is done in English Braille (recall that YouTube video).
   (c) Each capital letter should have ⠨ added before them, unless:
   (d) If a word is in all capital letters, add ⠨⠨ before the first letter and then don't add ⠨ before each letter. Only do this when the word has more than letter (so the word 'I' gets only one ⠨).

4.1 What this module does

You'll see there are numerous other functions in this file. You should not edit them. They are all functions we provide to you. Together, they:

1. Open a file such as `test1.txt`, and split it into a series of paragraphs, which are each split into a series of words
2. Using your `word_to_braille` it converts each word to Braille, word by word
3. And then saves the entire translated text to the specified new filename, e.g. `test1_braille.txt`
4. In the doctests, it compares `test1_braille.txt` to `expected1.txt`, `test2_braille.txt` to `expected2.txt` and `test3_braille.txt` to `expected3.txt`. By using files we can compare larger texts than the strings we manually enter. For example, `expected3.txt` is the first chapter of Candide's "Voltaire".

5 English Braille

Now that we can translate text into French Braille, let's build on this to make an English Braille translator! There are some important differences between English and French Braille. These are the ones we expect your code to handle. The blue text indicates what our starter code already does for you.

1. To indicate capital letters, English Braille uses ⠠ rather than ⠨. Similarly, English Braille uses ⠠⠠ to indicate all-caps.
2. English Braille also uses ⠼ to indicate a number is starting. But rather than put ⠼ in front of *every* digit, English Braille puts ⠼ only in front of the first digit, and then puts ⠰ at the end of the number.
3. Punctuation:
   (a) Parentheses: both open and close are represented by ⠶, instead of having ⠦ to open and ⠴ to close.
   (b) Quotations: are opened by ⠦ and closed by ⠴, instead of having ⠶ to represent both.
   (c) ? is represented by ⠦ instead of ⠢ (Note: ⠦ can *also* represent an open quotation mark)
4. English Braille has contractions: common words or letter sequences that are represented by a single Braille cell. To simplify the assignment we will not account for *all* the common contractions in English Braille. These are the ones your code should translate:
   (a) Four common whole-word contractions, which replace ç through ù in English Braille:

English word	'and'	'for'	'the'	'with'
English Braille	⠯	⠿	⠮	⠾
Value in French Braille	ç	é	à	ù

   (b) The two-letter contractions which replace the last *decade* of French letters:

English letter combination	'ch'	'gh'	'sh'	'th'	'wh'	'ed'	'er'	'ou'	'ow'
English Braille	⠡	⠣	⠩	⠹	⠱	⠫	⠻	⠳	⠪
Value in French Braille	â	ê	î	ô	û	ë	ï	ü	œ

5.1 Notes on Capitalization

- The word 'AND' should come out as ⠠⠯ rather than ⠠⠠⠯. This is because once it is in Braille it is a single letter and the ⠠⠠ is for when we have more than one letter.
- The word 'And' would also come out as ⠠⠯.
- But "aNd" would come out as ⠁⠠⠝⠙ — mixing case within a word stops contracting from happening.

5.2 Code to Complete

5.2.1 Starting Notes

Download the file `english_braille.py` and put your name at the top.

To help you get started:

- We provide the function `two_letter_contractions`. This function takes English text, and replaces the two letter contractions with the associated French accented letter. For example, 'ch' becomes 'â'. Then 'â' goes through our French Braille translator, and we get ⠡.
- We also provide `whole_word_contractions`. This function takes English text, and replaces the whole-word contractions with the associated French accented letter. So for example, 'for' becomes 'é'. Then when this 'é' goes through our French Braille translator, we get ⠿.

In this Part, we employ a common strategy in computer science: *reduction.* Often times in CS we "reduce" a new problem to an already-solved one. For example, we took the problem of creating an English Braille translator, and now we figure outhow to use our existing French Braille translator to do the bulk of the work.

We can do this through figuring out how to intelligently change the input and output of the French Braille translator. In CS speak, we say we "reduce" English Braille translation to French Braille translation.

5.2.2 What to Complete [30 points]

To complete:

1. `convert_contractions` [5 points]. This function takes English text, and replaces *all* the contractions with the associated French accented letter. So, for example 'wither' becomes ⠾⠻.
2. `convert_quotes` [5 points]. This function replaces all straight-quotation marks with rounded open/closed (" ") quotation marks, so that we can differentiate them when converting to English Braille.
   - *Tip:* The " and " are examples of *sentinel values.* A sentinel value is a technique when programming to assign a special or "dummy" value in order do some intermediate processing. You may want to use more sentinel values in the rest of this assignment!
3. `english_text_to_braille` [20 points]. Convert a string of English text into English Braille represented in Unicode. To do this:
   - You must create and call at least two helper functions. This is a big problem and we want to see you practice decomposing a big problem into smaller problems.
   - Your helper functions must have doctests!
   - You will want to be calling functions from previous modules; recall we wrote functions for you to use.
   - There are multiple correct ways to complete this function.

We give you some code to start you off with in `english_text_to_braille`:

(a) We call `convert_contractions`, and then we call `text_to_braille`

(b) Then we replace the French capitalization symbol with the English one.

You should add code before, in between, and after the provided lines of code in `english_text_to_braille`. What else should you change about the text before it is run through the French Braille translator? What should you change about what comes out of the French Braille translator?

6 Closing Notes

6.1 Learning More

If you're curious about the history of Braille and computing, you can learn more from:

- This 99 Percent Invisible podcast episode "The Universal Page": `https://99percentinvisible.org/episode/the-universal-page/` on the history of Braille.
- The book *Code: The Hidden Language of Computer Hardware and Software* by Charles Petzold traces the history of computer code, starting from Braille and Morse code and their influences upon modern computing.
- The book *Mapping Access: Universal Design and the Politics of Disability* by Aimi Hamraie looks at the intertwined histories of technology and disability, and how many computer technologies we use today were created by/for disabled people. Examples include the mouse of your computer, and the touchscreen on your phone/tablet/etc.

And if you'd like to learn more about character encodings, and why you might see a French name like "François" often turned into "Fran?ois" on a computer, you can learn more from:

- Joel Splosky's blog post: "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)" `https://www.joelonsoftware.com/2003/10/08/`
- Aditya Mukerjee's magazine article: " I Can Text You A Pile of Poo, But I Can't Write My Name" `https://modelviewculture.com/pieces/i-can-text-you-a-pile-of-poo-but-i-cant-write-` Discusses the politics of character encodings: as the title indicates, we have Unicode support for smiling-poo emojis, but at the time of press, Bengali was not supported. This is despite Bengali being the seventh most common native language in the world. A big theme in the history of character encodings is how US English was established as the default to the continuing detriment of other languages.

If you read through all of the starter code in `text_to_braille` you might have noticed that, to make the code work the same on Windows/Mac/Linux I had to in one place remove '\n' and '\r'. This is because different operating systems represent a new line differently. Linux uses '\n', Windows uses '\r', and Mac uses '\r\n'.

What's the difference? It goes back to the day of typewriters: '\r' meant moving the typewriter's position to the far-right (ending a line of text), whereas '\n' meant shifting the typewriter down a line. Today, both can mean a new line, but on a typewriter they were different! This is one example of how the history of technology continues to affect programming today.