CPSLP Programming assignment

1 Introduction - Speech Synthesiser

Your task for this assignment is to create a **Speech Synthesiser**! So, your Python program will take text input from a user and convert it to a sound waveform containing intelligible speech. This will be a *very basic* waveform concatenation system, whereby the acoustic units are recordings of diphones. You will be provided with several files to help you do this:-

simpleaudio.py
: This is a version of the `simpleaudio.py` module that we have used in the lab sessions. The `Audio` class contained therein will allow you to save, load and play `.wav` files as well as perform some simple audio processing functions. You should **not** modify this file.

synth.py
: This is a skeleton structure for your program, with a few hints to get you going. Your task is to fill in the missing components to make it work. You are free to add any classes, methods or functions that you wish but you must **not** change the existing argparse arguments.

diphones/
: A folder containing `.wav` files for the diphone sounds. A diphone is a voice recording lasting from the middle of one speech sound to the middle of a second speech sound (i.e. the transition between two speech sounds). If you are unfamiliar with diphones, try listening to some of them to understand what this means!

examples/
: A folder containing example `.wav` files of how the synthesiser should sound.

2 Task 1 - Basic Synthesis

The primary task for this assignment is to design a program that takes an input phrase and synthesises it. The main steps in this procedure are as follows:-

- normalise the text (convert to lower case, remove punctuation, etc.) to give you a straightforward sequence of words
- expand the word sequence to a phone (or "speech sound") sequence – you should make use of `nltk.corpus.cmudict` to do this, which is a pronunciation lexicon provided as part of NLTK. It is already imported in the skeleton script for you, but you will need to find out yourself (e.g. by googling) what the `cmudict` object is, what it can do and so how to use it for your purposes.
- expand the phone sequence to the corresponding diphone sequence
- concatenate the necessary diphone wav files together in the right order to produce the required synthesised audio in an instance of the Audio class.

A user should be able to execute your program by running the `synth.py` script from the command line with arguments, e.g. the following should play "hello":-

```
python synth.py -p "hello"
```

If a word is not in the **cmudict** then you should print an informative warning to the user and exit the program.

You can listen to the examples `hello.wav` and `rose.wav` which were created as follows:-

```
python synth.py -o hello.wav "hello nice to meet you"
python synth.py -o rose.wav "A rose by any other name would smell as sweet"
```

If you execute the same commands with your program and the output sounds the same then it is likely you have a functioning basic synthesiser!

3 Task 2 - Extending the Functionality

Implement **at least two** of the following extensions:-

Extension A – Volume Control
Allow the user to set the volume argument (`--volume, -v`) to a value between 0 and 100 (minimum and maximum loudness respectively). You could for example use the `rescale` method from the `Audio` class to do this.

Extension B – Punctuation
If the input phrase contains a comma – insert 200ms of silence.
If it contains a period, colon, question mark or exclamation mark – insert 400ms of silence.
Strip all other punctuation.

Extension C – Spelling
Allow the user to set the spell argument (`--spell, -s`) that will synthesise the text as spelled out instead of read normally. Do this by converting a string into a sequence of letters, and then to an appropriate phone sequence to pronounce for each letter in its alphabetic form. [hint: cmudict contains entries for letter names]

Extension D – Emphasis markup
One way emphasis can be indicated on a word is by increased loudness, duration and some pitch (f0) accent. Implement a simplified version of this in your synthesiser. Allow the user to put curly braces around any 1 word in the input text, and increase the loudness (don't worry about duration or pitch) of that word noticeably compared to the rest of the utterance. So, for example:

```
"The {cat} sat on the mat."
-> the word 'cat' should sound louder than the rest of the utterance.
```

Extension E – Smoother Concatenation

Simply pasting together diphone audio waveforms one after the other can lead to audible glitches where the waveform “jumps” at join points. Implement a simple way to alleviate this by “cross-fading” between adjacent diphones using a 10 msec overlap. (To achieve a cross-fade, you need to lower the amplitude at the end of one diphone down to 0.0 over 10msec and then add in the signal from the start of the next diphone which is similarly tapered from 0.0 to normal amplitude over the same 10msec period). Note this will only mitigate one cause of “choppiness” in the synthetic speech and so can only do so much! Audio examples are provided for you to compare cross-faded concatenation with simple concatenation. [hint: you will probably find numpy very useful to implement this extension in a succinct and efficient way!]

Extension F – Text Normalisation for Dates

If the string contains dates in the form DD/MM, DD/MM/YY or DD/MM/YYYY, then convert them to word sequences, e.g. –

```
"John Lennon died on 8/12/80"
-> "john lennon died on december eighth nineteen eighty"
```

To keep things simple, assume your code will only need to handle years in the range 1900-1999. [hint: you may wish to make use of the built-in `datetime` and/or `re` modules to help you do this.]

4 Rules and Assessment

Your submission will comprise a single file of Python code and should abide by the following rules:-

- you are encouraged to discuss the assignment together but all submissions must be written individually and be your **own work**.
- the penalties for plagiarism are **potentially very harsh**. Googling how to use a particular object or syntax feature is to be expected, and finding information in that way is no problem at all. If you find a one-liner to achieve some neat “trick”, it’s also fine to include that, as long as you attribute it (e.g. provide the URL for the StackOverflow page you found it on in a code comment for example). Conversely, cutting and pasting whole sections of code, or even whole functions is definitely **not** in the spirit of the exercise and will be penalised.
- your submission may only use **numpy**, **nltk**, the provided files, and any packages that are **built-in** to Python. I must be able to run your code on my computer using just the one file and without installing anything else.
- you may **not** change any of the existing argparse arguments provided in `synth.py`.

The assignments will be graded out of 100 and will be assessed according to the following criteria:-

Task 1 (40 marks)

Your system:-

- is able to synthesise several test phrases that contain only words (no punctuation, numbers, etc.).
- is reasonably robust ("validation!") - for example it can handle out of vocabulary words or a missing "diphones" directory in an elegant and/or informative way rather than just falling over and breaking
- is able to play and/or save the output to a file

Be careful to note: just because your code runs and works as specified above, it does not mean you will necessarily get full marks! Many aspects of your code will be considered beyond simply whether it works or not:

- has the task clearly been understood with clear evidence of a sensible attempt to implement solution? (10)
- does it work entirely as specified? (5)
- is it efficient? (5)
- is the code sensible/logical? (10)
- is the code legible, sufficiently documented and "friendly" for others (5)
- is the code robust? (5)

Task 2 (40 marks)

You must implement at least 2 of the extensions in order to pass.
However, for a higher mark, you can implement more of the extensions or present particularly strong solutions. For example, 3 extensions implemented to a high standard could achieve the same grade as 4 extensions implemented less comprehensively. Also note that the extensions vary in terms of the effort required to implement them. Extension A is far simpler than Extension E, for example, and the marks awarded will naturally reflect this. Again, keep in mind the marking criteria: functionality, design, robustness, legibility.

Style and Design (20 marks)

Further marks will be awarded based on how well your code is designed and presented overall. Therefore, take care to use appropriate object-orientated design as well as suitable code formatting, naming and plenty of appropriate comments and docstrings. You can also earn extra marks by being 'pythonic' (i.e. keeping your code succinct, well-organised and easily-readable; good use of nice Python language features (e.g. comprehensions); etc.)

You will be required to submit your modified `synth.py` file at the end.

Submissions are marked anonymously. You **must** prepend your exam number (the 'B' number on your student card) to the file prior to submitting it, e.g. your submitted file should be in the format:

```
B123456_synth.py
```

In addition, do **not** include any identifying content (e.g. Matriculation number, name etc.) in the submitted file. Submit only one Python code file in the above format (and **not** a zip file for example).

Finally, if you should have any queries about the task, or questions more generally, then please do not hesitate to ask - post a question on the course forum!