CPSLP Programming assignment

1 Introduction - Speech Synthesiser

Your task for this assignment is to create a speech synthesiser! Your commandline Python program will take text input from a user and convert it to a sound waveform of 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 in this module 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_args.py
: A module that will handle commandline parsing for you. You should **not** modify this file. You can safely ignore any commandline arguments in that module which are not specifically mentioned in the assignment description below.

synth.py
: This is a skeleton structure for your program, with hints to get you going. You will see there are some classes and functions there already to start you off, but your task is to add more code to make it work. You are free to add any classes, methods or functions that you wish but you **must not change** the existing 2 class names, the function prototypes (i.e. the function/method names and their parameters) or argparse arguments. There are also several comments to provide guidance - please do delete these to tidy up your code once you have progressed with your solution.

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. (Hint: you cannot rely upon the fact that all possible diphones have been given to you!)

examples/
: A folder containing example `.wav` files to compare how your synthesiser sounds with respect to a reference implementation.

test_synth.py
: A suite of unittests for testing your implementation. This set of tests does not aim to test the *correctness* of any results, but instead merely aims to ensure the *API* works in the way expected by the automatic marking code (i.e. checks your code won't break the automatic marking code), so **you must ensure they run successfully before submitting your code**. Any issues caused by your code failing to pass these pre-submission unittests will be your responsibility.

To set yourself up to work on this assignment most conveniently, just create a PyCharm project in this directory (i.e. the one created by unpacking the downloadable zipfile). Then simply add 2 configurations:

- A standard run configuration, which you should set up to run the `synth.py` script. Add initial commandline arguments "-p hello", so that when you click the green run button, PyCharm will execute the following for you automatically:

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

  That can be just to get you going; you can subsequently change the command line parameters in the PyCharm project configuration to suit whatever task you are working on at the time. Or, alternatively, just run your program manually yourself from the command line with whatever commandline arguments you like.

- A standard unittest configuration, pointing at the `test_synth.py` file. Clicking the green run button when this configuration is active will run the unittest suite provided for you. These will check you haven't changed the API against the instructions. The testsuite will also check which of the optional extension tasks it appears you have attempted. If you have attempted some task but the corresponding test still fails, then you need to investigate to ensure your implementation of the task is properly enabled. Tests for tasks you have not implemented should be expected to always fail. Feel free to add any of your own further tests to this testsuite to help you code solutions, though do be careful not to break the ones already there.

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:-

1. normalise the text (convert to lower case, remove punctuation, etc.) to give you a straightforward sequence of words that can be looked up in the pronunciation dictionary in the next step
2. 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 experiment yourself what the `cmudict` object is, what it can do and so how to use it for your purposes. Don't forget, utterances should always start and end with a silence phone! For example, saying the sentence "Hello!" requires a phone sequence of ['PAU', 'HH', 'AH', 'L', 'OW', 'PAU'] (where PAU is the label for the short pause or silence phone).
3. expand the phone sequence to the corresponding diphone sequence (e.g. to say "Hello!" we need ['PAU-HH', 'HH-AH', 'AH-L',...] etc.)
4. concatenate the corresponding diphone wav files together in the right order to produce the required synthesised audio in an instance of the Audio class. Your aim here is to create a single new waveform (e.g. one array of audio data in a single Audio class instance), and not just to quickly play one diphone sound after the other, for example.

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 your program should take appropriate action (e.g. raise an exception with an informative message for the user).

You can listen to the examples `hello.wav` and `rose.wav` in the `./examples` subdirectory, 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! You could also compare waveforms sample by sample, to ensure your synthesiser functions entirely like the reference implementation.

3 Task 2 - Extending the Functionality

Implement **at least two** of the following extensions (but note, the easy ones do not receive as much credit):-

Extension A – Volume Control [effort: 1 = easy]

Allow the user to set the volume argument (`--volume, -v`) to a value between 0 and 100 (minimum and maximum loudness respectively) to change the amplitude of the synthesised waveform before either playing and/or saving it.

Extension B – Speaking Backwards [effort: 2 = fairly easy]

Have fun by making the synthesiser speak backwards in the following three ways:

signal When the user gives the commandline option `--reverse signal`, switch the waveform signal for the whole synthetic utterance back to front.

words When the user gives the commandline option `--reverse words`, reverse the order of the words that will be synthesised (e.g. `"oh hi there" -> "there hi oh"`).

phones When the user gives the commandline option `--reverse phones`, reverse the order of the phones that will be spoken for the whole utterance.

Extension C – Pronunciation addenda [effort: 3 = medium]

Add the necessary code to enable the user to use the `--addpron` flag. The user would give this flag to specify a text file containing pronunciation "addenda". In short, these can be both pronunciations for words that are not present in the cmudict pronunciation lexicon in NLTK, or pronunciations that should override those which are already present in cmudict. A sample textfile `pron_addenda.txt` is included in the assignment zipfile to illustrate what the format of the pronunciation addenda is. You will see entries in that file for the words "drookit", "scunner" and "numpty". Those are local Scottish words which are not included in cmudict. However, when the user gives that file using the `--addpron` flag, your synthesiser should

become able to pronounce sentences containing those words! Since the user is supplying this data, you might like to consider ways the user's file could be wrong or broken, and so design your program to be robust to that.

Extension D – Synthesising a File [effort: 4 = more challenging]

Add appropriate code to enable the user to give the `--fromfile` flag. The user can use this flag to specify a text file containing text to synthesise – and in this case, your program should open the file with the given filename and synthesise whatever is contained there. Note, however, your program should process the text *sentence by sentence* and interact with other commandline arguments correctly. For example, your program should synthesise the first sentence and then play it if the `--play` flag is given, then synthesise the second, and then play that one, and so on. To separate one sentence from the next, ensure you insert 400msec of pure silence between utterances (e.g. wherever an end-of-sentence mark is found, such as full stop or exclamation mark etc.) Anticipate the user will provide standard text, so you cannot assume there will be just one sentence per line. If the `--outfile` option is given by the user, then each of the synthesised waveforms must be concatenated to a single waveform (including the extra 400ms silences between utterances) prior to saving it at the end.

Extension E – Smoother Concatenation [effort: 4 = more challenging]

Simply pasting together diphone audio waveforms one after the other can lead to audible glitches where the waveform "jumps" at boundaries between diphones. 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 gradually reduce the amplitude at the end of one diphone waveform over a 10msec window and then overlap and add in the signal from the start of the next diphone which is similarly tapered up to normal amplitude over the same 10msec window period. Use a Hann window for scaling and overlapping the audio signals at all concatenation points. Note this will only mitigate one cause of "choppiness" in the synthetic speech and so can only do so much to make it sound better! Audio examples are provided for you to compare cross-faded concatenation with simple concatenation. Compare sizes of those files with ones produced by your code. to make sure you're not losing or gaining any samples. [hint: you will probably find numpy very useful to implement this extension in a succinct and efficient way!]

4 Rules and Assessment

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

- all submissions must be written individually and be your **own work**.
- the University's penalties for plagiarism are **potentially harsh**. Googling how to use a particular Python 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). In contrast, cutting and pasting whole sections of code, or even whole functions/classes is **not** in the spirit of the exercise and will be penalised. Note, Python code plagiarism detection software can spot copied code even if the names

and formatting are changed. So, please, do just put into practice everything you've learned this semester and come up with your own code.

- your submission may only use **numpy**, **nltk**, the provided files, and any packages that are **built in** to Python's Standard Library. A marker must be able to run your code on their computer using just the one file you submit and without installing anything else.
- you may **not** change any of the existing argparse arguments provided in `synth_args.py` – when you view that file you will see argparse has already been set up to take all the correct command-line arguments for you.

The assignments will be graded out of 100 and will be assessed using a combination of automatic testing and human markers according to the following criteria:-

Task 1 (total 30 marks) -

Ensure your system:-

- is able to synthesise test phrases paying attention only to full words (ignoring abbreviations, punctuation or numbers for example).
- is reasonably robust - for example it can handle out of vocabulary words, or malformed input, or a missing "diphones" directory, or even missing diphones, in an elegant and/or informative way rather than just falling over or breaking
- is able to play and/or save the output to a file

IMPORTANT: just because your code might run and work as specified above, it does not mean it will necessarily get full marks. Many aspects of your code will be considered beyond simply whether it works or not, including for example:

- has the task clearly been understood with clear evidence of a sensible attempt to implement a working solution?
- does it work entirely as specified?
- is it efficient?
- is the code well-structured, sensible and with good clear logic?
- is the code legible, sufficiently documented and "friendly" for others
- is the code robust?
- is the code written in a succinct and pythonic way?

Task 2 (max total 30 marks) -

Implement **at least two of the extensions**. Note that the extensions clearly vary in terms of the effort required to implement them. Extension A is far simpler than Extension D or E, for example. The marks available reflect the effort required for each of the extension tasks (indicative marks: easy (2); fairly easy (4); medium (7); challenging (10)). To aim for a higher grade, you can choose to implement more of the extensions, taking care to present strong solutions (three extensions implemented to a high standard could achieve the same grade as 4 extensions implemented less well, for example). Again, keep in mind key characteristics such as functionality, design, maintainability, efficiency, and robustness.

General Criteria (total 40 marks) -

Further marks will be awarded based on how well your code is designed and presented overall. Therefore, take care to use appropriate modular design as well as suitable code formatting, naming and appropriate comments and docstrings. You can also earn more credit 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.). The specific criteria used (with their respective mark allocation): Design (10); Readibility (10); Pythonicity (10); Documentation (5); Robustness (5)).

You will be required to submit your modified `synth.py` file as arranged by the Teaching Offices through Learn once you have finished.

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 name 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.

To summarise: submit only one Python code file, in plain text format (and not a zip file for example), with a filename which has your exam number at the start.

Finally, if you should have any queries about the task, or questions more generally, then please do not hesitate to ask!