gTTS Documentation

Pierre-Nick Durette

May 15, 2018

Documentation

gTTS (*Google Text-to-Speech*), a Python library and CLI tool to interface with Google Translate's text-to-speech API. Writes spoken mp3 data to a file, a file-like object (bytestring) for further audio manipulation, or stdout. It features flexible pre-processing and tokenizing, as well as automatic retrieval of supported languages.

CHAPTER 1

Installation

```
pip install gTTS
```

1.1 Command-line (`gtts-cli`)

After installing the package, the `gtts-cli` tool becomes available:

```
$ gtts-cli
```

1.1.1 gtts-cli

Read <text> to mp3 format using Google Translate's Text-to-Speech API (set <text> or –file <file> to - for standard input)

```
gtts-cli [OPTIONS] <text>
```

Options

-f, --file `<file>`
Read from <file> instead of <text>.

-o, --output `<output>`
Write to <file> instead of stdout.

-s, --slow
Read more slowly.

-l, --lang `<lang>`
IETF language tag. Language to speak in. List documented tags with –all. [default: en]

--nocheck
Disable strict IETF language tag checking. Allow undocumented tags.

--all
Print all documented available IETF language tags and exit.

--debug
Show debug information.

--version
Show the version and exit.

Arguments

<text>
Optional argument

1.1.2 Examples

List available languages:

```
$ gtts-cli --all
```

Read 'hello' to `hello.mp3`:

```
$ gtts-cli 'hello' --output hello.mp3
```

Read 'bonjour' in French to `bonjour.mp3`:

```
$ gtts-cli 'bonjour' --lang fr --output bonjour.mp3
```

Read 'slow' slowly to `slow.mp3`:

```
$ gtts-cli 'slow' --slow --output slow.mp3
```

Read 'hello' to `stdout`:

```
$ gtts-cli 'hello'
```

Read `stdin` to `hello.mp3` via `<text>` or `<file>`:

```
$ echo -n 'hello' | gtts-cli - --output hello.mp3
$ echo -n 'hello' | gtts-cli --file - --output hello.mp3
```

Read 'no check' to `nocheck.mp3` without language checking:

```
$ gtts-cli 'no check' --lang zh --nocheck --ouput nocheck.mp3
```

Note: Using `--nocheck` can make the command *slightly* faster. It exists however to force a `<lang>` language tag that might not be documented but would work with the API, such as for specific regional sub-tags of documented tags (examples for 'en': 'en-gb', 'en-au', etc.).

1.1.3 Playing sound directly

You can pipe the output of `gtts-cli` into any media player that supports `stdin`. For example, using the `play` command from SoX:

```
$ gtts-cli 'hello' | play -t mp3 -
```

1.2 Module (`gtts`)

1.2.1 gTTS (`gtts.gTTS`)

class gtts.tts.**gTTS**(*text, lang='en', slow=False, lang_check=True, pre_processor_funcs=[<function tone_marks>, <function end_of_line>, <function abbreviations>, <function word_sub>], tokenizer_func=<bound method Tokenizer.run of re.compile('(?<=\?).|(?<=\!).|(?<=\).|(?<!\.[a-z])\. |(?<!\.[a-z])\, |\|\|\/|\»|\/|\|\. . . |\|\|\:|\¿|\|\¡|\(|\;|\)|\—', re.IGNORECASE) from: [<function tone_marks>, <function period_comma>, <function other_punctuation>]>*)

gTTS – Google Text-to-Speech.

An interface to Google Translate's Text-to-Speech API.

Parameters

- **text** (*string*) – The text to be read.
- **lang** (*string, optional*) – The language (IETF language tag) to read the text in. Defaults to 'en'.
- **slow** (*bool, optional*) – Reads text more slowly. Defaults to `False`.
- **lang_check** (*bool, optional*) – Strictly enforce an existing `lang`, to catch a language error early. If set to `True`, a `ValueError` is raised if `lang` doesn't exist. Default is `True`.
- **pre_processor_funcs** (*list*) – A list of zero or more functions that are called to transform (pre-process) text before tokenizing. Those functions must take a string and return a string. Defaults to:

```
[
    pre_processors.tone_marks,
    pre_processors.end_of_line,
    pre_processors.abbreviations,
    pre_processors.word_sub
]
```

- **tokenizer_func** (*callable*) – A function that takes in a string and returns a list of string (tokens). Defaults to:

```
Tokenizer([
    tokenizer_cases.tone_marks,
    tokenizer_cases.period_comma,
    tokenizer_cases.other_punctuation
]).run
```

See also:

Pre-processing and tokenizing

Raises

- `AssertionError` – When `text` is `None` or empty; when there's nothing left to speak after pre-precessing, tokenizing and cleaning.
- `ValueError` – When `lang_check` is `True` and `lang` is not supported.
- `RuntimeError` – When `lang_check` is `True` but there's an error loading the languages dictionnary.

save(*savefile*)
Do the TTS API request and write result to file.

Parameters **savefile** (*string*) – The path and file name to save the `mp3` to.

Raises *gTTSError* – When there's an error with the API request.

write_to_fp(*fp*)
Do the TTS API request and write bytes to a file-like object.

Parameters **fp** (*file object*) – Any file-like object to write the `mp3` to.

Raises

- *gTTSError* – When there's an error with the API request.
- `TypeError` – When `fp` is a file-like object that takes bytes.

exception gtts.tts.**gTTSError**(*msg=None*, ***kwargs*)
Exception that uses context to present a meaningful error message

infer_msg(*tts*, *rsp*)
Attempt to guess what went wrong by using known information (e.g. http response) and observed behaviour

1.2.2 Languages (`gtts.lang`)

Note: The easiest way to get a list of available language is to print them with `gtts-cli --all`

gtts.lang.**tts_langs**()
Languages Google Text-to-Speech supports.

Returns

A dictionnary of the type *{ '<lang>': '<name>'}*

Where <*lang*> is an IETF language tag such as *en* or *pt-br*, and <*name*> is the full English name of the language, such as *English* or *Portuguese (Brazil)*.

Return type dict

The dictionnary returned combines languages from two origins:

- Languages fetched automatically from Google Translate
- Languages that are undocumented variations that were observed to work and present different dialects or accents.

1.2.3 Examples

Write 'hello' in English to `hello.mp3`:

```
>>> from gtts import gTTS
>>> tts = gTTS('hello', lang='en')
>>> tts.save('hello.mp3')
```

Write 'hello bonjour' in English then French to `hello_bonjour.mp3`:

```
>>> from gtts import gTTS
>>> tts_en = gTTs('hello', lang='en')
>>> tts_fr = gTTS('bonjour', lang='fr')
>>>
>>> with open('hello_bonjour.mp3', 'wb') as f:
...     tts_en.write_to_fp(f)
...     tts_fr.write_to_fp(p)
```

1.2.4 Playing sound directly

There's quite a few libraries that do this. Write 'hello' to a file-like object do further manipulation::

```
>>> from gtts import gTTS
>>> from io import BytesIO
>>>
>>> mp3_fp = BytesIO()
>>> tts = gTTS('hello', 'en')
>>> tts.write_to_fp(mp3_fp)
>>>
>>> # Load `audio_fp` as an mp3 file in
>>> # the audio library of your choice
```

1.2.5 Logging

`gtts` does logging using the standard Python logging module. The following loggers are available:

`gtts.tts` Logger used for the `gTTS` class

`gtts.lang` Logger used for the `lang` module (language fetching)

`gtts` Upstream logger for all of the above

1.3 Pre-processing and tokenizing

The *gtts.tokenizer* module powers the default pre-processing and tokenizing features of gTTS and provides tools to easily expand them. *gtts.tts.gTTS* takes two arguments pre_processor_funcs (list of functions) and tokenizer_func (function). See: *Pre-processing*, *Tokenizing*.

1.3.1 Definitions

Pre-processor: Function that takes text and returns text. Its goal is to modify text (for example correcting pronounciation), and/or to prepare text for proper tokenization (for example enuring spacing after certain characters).

Tokenizer: Function that takes text and returns it split into a list of *tokens* (strings). In the gTTS context, its goal is to cut the text into smaller segments that do not exceed the maximum character size allowed for each TTS API request, while making the speech sound natural and continuous. It does so by splitting text where speech would naturaly pause (for example on ".") while handling where it should not (for example on "10.5" or "U.S.A."). Such rules are called *tokenizer cases*, which it takes a list of.

Tokenizer case: Function that defines one of the specific cases used by *gtts.tokenizer.core.Tokenizer*. More specefically, it returns a regex object that describes what to look for for a particular case. *gtts.tokenizer.core.Tokenizer* then creates its main *regex* pattern by joining all *tokenizer cases* with "|".

1.3.2 Pre-processing

You can pass a list of any function to *gtts.tts.gTTS*'s pre_processor_funcs attribute to act as pre-processor (as long as it takes a string and returns a string).

By default, *gtts.tts.gTTS* takes a list of the following pre-processors, applied in order:

```
[
    pre_processors.tone_marks,
    pre_processors.end_of_line,
    pre_processors.abbreviations,
    pre_processors.word_sub
]
```

gtts.tokenizer.pre_processors.**abbreviations**(*text*)

Remove periods after an abbreviation from a list of known abbrevations that can be spoken the same without that period. This prevents having to handle tokenization of that period.

Note: Could potentially remove the ending period of a sentence.

Note: Abbreviations that Google Translate can't pronounce without (or even with) a period should be added as a word substitution with a `PreProcessorSub` pre-processor. Ex.: 'Esq.', 'Esquire'.

`gtts.tokenizer.pre_processors.`**`end_of_line`**(*text*)
Re-form words cut by end-of-line hyphens.

Remove "<hyphen><newline>".

`gtts.tokenizer.pre_processors.`**`tone_marks`**(*text*)
Add a space after tone-modifying punctuation.

Because the *tone_marks* tokenizer case will split after a tone-modidfying punctuation mark, make sure there's whitespace after.

`gtts.tokenizer.pre_processors.`**`word_sub`**(*text*)
Word-for-word substitutions.

Customizing & Examples

This module provides two classes to help build pre-processors:

- *gtts.tokenizer.core.PreProcessorRegex* (for *regex*-based replacing, as would `re.sub` use)
- *gtts.tokenizer.core.PreProcessorSub* (for word-for-word replacements).

The `run(text)` method of those objects returns the processed text.

Speech corrections (word substitution)

The default substitutions are defined by the *gtts.tokenizer.symbols.SUB_PAIRS* list. Add a custom one by appending to it:

```
>>> from gtts.tokenizer import pre_processors
>>> import gtts.tokenizer.symbols
>>>
>>> gtts.tokenizer.symbols.SUB_PAIRS.append(
...     ('sub.', 'submarine')
... )
>>> test_text = "Have you seen the Queen's new sub.?"
>>> pre_processors.word_sub(test_text)
"Have you seen the Queen's new submarine?"
```

Abbreviations

The default abbreviations are defined by the *gtts.tokenizer.symbols.ABBREVIATIONS* list. Add a custom one to it to add a new abbreviation to remove the period from. *Note: the default list already includes an extensive list of English abbreviations that Google Translate will read even without the period.*

See *gtts.tokenizer.pre_processors* for more examples.

1.3.3 Tokenizing

You can pass any function to *gtts.tts.gTTS*'s `tokenizer_func` attribute to act as tokenizer (as long as it takes a string and returns a list of strings).

By default, `gTTS` takes the *gtts.tokenizer.core.Tokenizer*'s *gtts.tokenizer.core.Tokenizer.run()*, initialized with default *tokenizer cases*:

```
Tokenizer([
    tokenizer_cases.tone_marks,
    tokenizer_cases.period_comma,
    tokenizer_cases.other_punctuation
]).run
```

The available *tokenizer cases* are as follows:

`gtts.tokenizer.tokenizer_cases.`**`legacy_all_punctuation`**`()`
Match all punctuation.

Use as only tokenizer case to mimic gTTS 1.x tokenization.

`gtts.tokenizer.tokenizer_cases.`**`other_punctuation`**`()`
Match other punctuation.

Match other punctuation to split on; punctuation that naturally inserts a break in speech.

`gtts.tokenizer.tokenizer_cases.`**`period_comma`**`()`
Period and comma case.

Match if not preceded by ".<letter>" and only if followed by space. Won't cut in the middle/after dotted abbreviations; won't cut numbers.

Note: Won't match if a dotted abbreviation ends a sentence.

Note: Won't match the end of a sentence if not followed by a space.

`gtts.tokenizer.tokenizer_cases.`**`tone_marks`**`()`
Keep tone-modifying punctuation by matching following character.

Assumes the *tone_marks* pre-processor was run for cases where there might not be any space after a tone-modifying punctuation mark.

Customizing & Examples

A *tokenizer case* is a function that returns a compiled *regex* object to be used in a `re.split()` context.

gtts.tokenizer.core.Tokenizer takes a list of *tokenizer cases* and joins their pattern with "|" in one single pattern.

This module provides a class to help build tokenizer cases: *gtts.tokenizer.core.RegexBuilder*. See *gtts.tokenizer.core.RegexBuilder* and *gtts.tokenizer.tokenizer_cases* for examples.

Using a 3rd-party tokenizer

Even though *gtts.tokenizer.core.Tokenizer* works well in this context, there are way more advanced tokenizers and tokenzing techniques. As long as you can restrict the lenght of output tokens, you can use any tokenizer

you'd like, such as the ones in NLTK.

1.3.4 gtts.tokenizer module reference (`gtts.tokenizer`)

class gtts.tokenizer.core.**RegexBuilder**(*pattern_args*, *pattern_func*, *flags=0*)

Builds regex using arguments passed into a pattern template.

Builds a regex object for which the pattern is made from an argument passed into a template. If more than one argument is passed (iterable), each pattern is joined by "|" (regex alternation 'or') to create a single pattern.

Parameters

- **pattern_args** (*iteratable*) – String element(s) to be each passed to `pattern_func` to create a regex pattern. Each element is `re.escape`'d before being passed.
- **pattern_func** (*callable*) – A 'template' function that should take a string and return a string. It should take an element of `pattern_args` and return a valid regex pattern group string.
- **flags** – `re` flag(s) to compile with the regex.

Example

To create a simple regex that matches on the characters "a", "b", or "c", followed by a period:

```
>>> rb = RegexBuilder('abc', lambda x: "{}\.".format(x))
```

Looking at `rb.regex` we get the following compiled regex:

```
>>> print(rb.regex)
'a\.|b\.|c\.'
```

The above is fairly simple, but this class can help in writing more complex repetitive regex, making them more readable and easier to create by using existing data structures.

Example

To match the character following the words "lorem", "ipsum", "meili" or "koda":

```
>>> words = ['lorem', 'ipsum', 'meili', 'koda']
>>> rb = RegexBuilder(words, lambda x: "(?<={}).".format(x))
```

Looking at `rb.regex` we get the following compiled regex:

```
>>> print(rb.regex)
'(?<=lorem).|(?<=ipsum).|(?<=meili).|(?<=koda).'
```

class gtts.tokenizer.core.**PreProcessorRegex**(*search_args*, *search_func*, *repl*, *flags=0*)

Regex-based substitution text pre-processor.

Runs a series of regex substitutions (`re.sub`) from each `regex` of a *`gtts.tokenizer.core.RegexBuilder`* with an extra `repl` replacement parameter.

Parameters

- **search_args** (*iteratable*) – String element(s) to be each passed to `search_func` to create a regex pattern. Each element is `re.escape`'d before being passed.
- **search_func** (*callable*) – A 'template' function that should take a string and return a string. It should take an element of `search_args` and return a valid regex search pattern string.
- **repl** (*string*) – The common replacement passed to the `sub` method for each `regex`. Can be a raw string (the case of a regex backreference, for example)
- **flags** – `re` flag(s) to compile with each *regex*.

Example

Add "!" after the words "lorem" or "ipsum", while ignoring case:

```
>>> import re
>>> words = ['lorem', 'ipsum']
>>> pp = PreProcessorRegex(words,
...                        lambda x: "({})".format(x), r'\1!',
...                        re.IGNORECASE)
```

In this case, the regex is a group and the replacement uses its backreference `\1` (as a raw string). Looking at `pp` we get the following list of search/replacement pairs:

```
>>> print(pp)
(re.compile('(lorem)', re.IGNORECASE), repl='{!'),
(re.compile('(ipsum)', re.IGNORECASE), repl='{!')
```

It can then be run on any string of text:

```
>>> pp.run("LOREM ipSuM")
"LOREM! ipSuM!"
```

See *gtts.tokenizer.pre_processors* for more examples.

run(*text*)
Run each regex substitution on `text`.

Parameters **text** (*string*) – the input text.

Returns text after all substitutions have been sequentially applied.

Return type string

class gtts.tokenizer.core.**PreProcessorSub**(*sub_pairs*, *ignore_case=True*)
Simple substitution text preprocessor.

Performs string-for-string substitution from list a find/replace pairs. It abstracts *gtts.tokenizer.core.PreProcessorRegex* with a default simple substitution regex.

Parameters

- **sub_pairs** (*list*) – A list of tuples of the style `(<search str>, <replace str>)`
- **ignore_case** (*bool*) – Ignore case during search. Defaults to `True`.

Example

Replace all occurences of “Mac” to “PC” and “Firefox” to “Chrome”:

```
>>> sub_pairs = [('Mac', 'PC'), ('Firefox', 'Chrome')]
>>> pp = PreProcessorSub(sub_pairs)
```

Looking at the `pp`, we get the following list of search (regex)/replacement pairs:

```
>>> print(pp)
(re.compile('Mac', re.IGNORECASE), repl='PC'),
(re.compile('Firefox', re.IGNORECASE), repl='Chrome')
```

It can then be run on any string of text:

```
>>> pp.run("I use firefox on my mac")
"I use Chrome on my PC"
```

See *gtts.tokenizer.pre_processors* for more examples.

run(*text*)

Run each substitution on `text`.

Parameters **text** (*string*) – the input text.

Returns text after all substitutions have been sequentially applied.

Return type string

class gtts.tokenizer.core.**Tokenizer**(*regex_funcs*, *flags=2*)

An extensible but simple generic rule-based tokenizer.

A generic and simple string tokenizer that takes a list of functions (called *tokenizer cases*) returning `regex` objects and joins them by “|” (regex alternation ‘or’) to create a single regex to use with the standard `regex.split()` function.

`regex_funcs` is a list of any function that can return a `regex` (from `re.compile()`) object, such as a *gtts.tokenizer.core.RegexBuilder* instance (and its `regex` attribute).

See the *gtts.tokenizer.tokenizer_cases* module for examples.

Parameters

- **regex_funcs** (*list*) – List of compiled `regex` objects. Each functions’s pattern will be joined into a single pattern and compiled.
- **flags** – `re` flag(s) to compile with the final regex. Defaults to `re.IGNORECASE`

Note: When the `regex` objects obtained from `regex_funcs` are joined, their individual `re` flags are ignored in favour of `flags`.

Raises `TypeError` – When an element of `regex_funcs` is not a function, or a function that does not return a compiled `regex` object.

Warning: Joined `regex` patterns can easily interfere with one another in unexpected ways. It is recommanded that each tokenizer case operate on distinct or non-overlapping chracters/sets of characters (For

example, a tokenizer case for the period (".") should also handle not matching/cutting on decimals, instead of making that a seperate tokenizer case).

Example

A tokenizer with a two simple case (*Note: these are bad cases to tokenize on, this is simply a usage example*):

```
>>> import re, RegexBuilder
>>>
>>> def case1():
...     return re.compile("\,")
>>>
>>> def case2():
...     return RegexBuilder('abc', lambda x: "{}\.".format(x)).regex
>>>
>>> t = Tokenizer([case1, case2])
```

Looking at `case1().pattern`, we get:

```
>>> print(case1().pattern)
'\,'
```

Looking at `case2().pattern`, we get:

```
>>> print(case2().pattern)
'a\.|b\.|c\.'
```

Finally, looking at `t`, we get them combined:

```
>>> print(t)
're.compile('\,|a\.|b\.|c\.', re.IGNORECASE) from: [<function case1 at␣
↪0x10bbcdd08>, <function case2 at 0x10b5c5e18>]'
```

It can then be run on any string of text:

```
>>> t.run("Hello, my name is Linda a. Call me Lin, b. I'm your friend")
['Hello', ' my name is Linda ', ' Call me Lin', ' ', " I'm your friend"]
```

run(*text*)

Tokenize *text*.

Parameters **text** (*string*) – the input text to tokenize.

Returns A list of strings (token) split according to the tokenizer cases.

Return type list

```
symbols.ABBREVIATIONS = ['dr', 'jr', 'mr', 'mrs', 'ms', 'msgr', 'prof', 'sr', 'st']
symbols.SUB_PAIRS = [('M.', 'Monsieur')]
symbols.ALL_PUNC = '?!.,¡()[]¿...;:—-\n'
symbols.TONE_MARKS = '?!'
```

1.4 License

1.5 Contributing

1.5.1 Reporting Issues

On the Github issues page. Thanks!

1.5.2 Submitting Patches

1. **Fork**. Follow PEP 8!
2. **Write/Update tests** (see below).
3. **Document**. Docstrings follow the Google Python Style Guide (docs by Sphinx). You can 'test' documentation:

```
$ pip install .[docs]
$ cd docs && make html # generated in docs/_build/html/
```

4. **Open Pull Request**. To the `master` branch.
5. **Changelog**. This project uses towncrier for managing the changelog. Please consider creating one or more 'news fragment' in the `/news/` directory and adding them to your PR, in the style of `<issue_or_pr_number>.<type>` where 'type' is one of: 'feature', 'bugfix', 'doc', 'removal' or 'misc'.

   See towncrier (New Fragments) for more details. Example:

```
$ echo 'Fixed a thing!' > gtts/news/1234.bugfix
```

Note:

Please don't hesitate to contribute! While good tests, docs and structure are encouraged, I do welcome great ideas over absolute comformity to the above!
Thanks!

1.5.3 Testing

Testing is done with the `unittest` framework.
As a rule, the file `./tests/test_<module>.py` file tests the `<module>` module.

To run all tests (testing only language 'en' and generating an html coverage report in `gtts/htmlcov/`):

```
$ pip install .[tests]
$ TEST_LANGS=en pytest -v -s gtts/ --cov=gtts --cov-report=html
```

1.6 Changelog

1.6.1 2.0.0 (2018-04-30)

(#108)

Features

- The `gtts` module
  - New logger ("gtts") replaces all occurrences of `print()`
  - Languages list is now obtained automatically (`gtts.lang`) (#91, #94, #106)
  - Added a curated list of language sub-tags that have been observed to provide different dialects or accents (e.g. "en-gb", "fr-ca")
  - New `gTTS()` parameter `lang_check` to disable language checking.
  - `gTTS()` now delegates the `text` tokenizing to the API request methods (i.e. `write_to_fp()`, `save()`), allowing `gTTS` instances to be modified/reused
  - Rewrote tokenizing and added pre-processing (see below)
  - New `gTTS()` parameters `pre_processor_funcs` and `tokenizer_func` to configure pre-processing and tokenizing (or use a 3rd party tokenizer)
  - Error handling:
    - Added new exception `gTTSError` raised on API request errors. It attempts to guess what went wrong based on known information and observed behaviour (#60, #106)
    - `gTTS.write_to_fp()` and `gTTS.save()` also raise `gTTSError` on *gtts_token* error
    - `gTTS.write_to_fp()` raises `TypeError` when `fp` is not a file-like object or one that doesn't take bytes
    - `gTTS()` raises `ValueError` on unsupported languages (and `lang_check` is `True`)
    - More fine-grained error handling throughout (e.g. *request failed* vs. *request successful with a bad response*)
- Tokenizer (and new pre-processors):
  - Rewrote and greatly expanded tokenizer (`gtts.tokenizer`)
  - Smarter token 'cleaning' that will remove tokens that only contain characters that can't be spoken (i.e. punctuation and whitespace)

  - Decoupled token minimizing from tokenizing, making the latter usable in other contexts
  - New flexible speech-centric text pre-processing
  - New flexible full-featured regex-based tokenizer (`gtts.tokenizer.core.Tokenizer`)
  - New `RegexBuilder`, `PreProcessorRegex` and `PreProcessorSub` classes to make writing regex-powered text *pre-processors* and *tokenizer cases* easier
  - Pre-processors:
    * Re-form words cut by end-of-line hyphens
    * Remove periods after a (customizable) list of known abbreviations (e.g. "jr", "sr", "dr") that can be spoken the same without a period
    * Perform speech corrections by doing word-for-word replacements from a (customizable) list of tuples
  - Tokenizing:
    * Keep punctuation that modify the inflection of speech (e.g. "?", "!")
    * Don't split in the middle of numbers (e.g. "10.5", "20,000,000") (#101)
    * Don't split on "dotted" abbreviations and accronyms (e.g. "U.S.A")
    * Added Chinese comma ("、"), ellipsis ("…") to punctuation list to tokenize on (#86)

- The `gtts-cli` command-line tool
  - Rewrote cli as first-class citizen module (`gtts.cli`), powered by Click
  - Windows support using *setuptool*'s *entry_points*
  - Better support for Unicode I/O in Python 2
  - All arguments are now pre-validated
  - New `--nocheck` flag to skip language pre-checking
  - New `--all` flag to list all available languages
  - Either the `--file` option or the `<text>` argument can be set to "-" to read from `stdin`
  - The `--debug` flag uses logging and doesn't pollute `stdout` anymore

Bugfixes

- `_minimize()`: Fixed an infinite recursion loop that would occur when a token started with the miminizing delimiter (i.e. a space) (#86)
- `_minimize()`: Handle the case where a token of more than 100 characters did not contain a space (e.g. in Chinese).
- Fixed an issue that fused multiline text together if the total number of characters was less than 100
- Fixed `gtts-cli` Unicode errors in Python 2.7 (famous last words) (#78, #93, #96)

Deprecations and Removals

- Dropped Python 3.3 support
- Removed `debug` parameter of `gTTS` (in favour of logger)
- `gtts-cli`: Changed long option name of `-o` to `--output` instead of `--destination`

- `gTTS()` will raise a `ValueError` rather than an `AssertionError` on unsupported language

Improved Documentation

- Rewrote all documentation files as reStructuredText
- Comprehensive documentation writen for Sphinx, published to http://gtts.readthedocs.io
- Changelog built with towncrier

Misc

- Major test re-work
- Language tests can read a `TEST_LANGS` enviromment variable so not all language tests are run every time.
- Added AppVeyor CI for Windows
- PEP 8 compliance

1.6.2 1.2.2 (2017-08-15)

Misc

- Update LICENCE, add to manifest (#77)

1.6.3 1.2.1 (2017-08-02)

Features

- Add Unicode punctuation to the tokenizer (such as for Chinese and Japanese) (#75)

Bugfixes

- Fix > 100 characters non-ASCII split, `unicode()` for Python 2 (#71, #73, #75)

1.6.4 1.2.0 (2017-04-15)

Features

- Option for slower read speed (`slow=True` for `gTTS()`, `--slow` for `gtts-cli`) (#40, #41, #64, #67)
- System proxy settings are passed transparently to all http requests (#45, #68)
- Silence SSL warnings from urllib3 (#69)

Bugfixes

- The text to read is now cut in proper chunks in Python 2 unicode. This broke reading for many languages such as Russian.
- Disabled SSL verify on http requests to accommodate certain firewalls and proxies.
- Better Python 2/3 support in general (#9, #48, #68)

Deprecations and Removals

- 'pt-br' : 'Portuguese (Brazil)' (it was the same as 'pt' and not Brazilian) (#69)

1.6.5 1.1.8 (2017-01-15)

Features

- Added `stdin` support via the '-' `text` argument to `gtts-cli` (#56)

1.6.6 1.1.7 (2016-12-14)

Features

- Added utf-8 support to `gtts-cli` (#52)

1.6.7 1.1.6 (2016-07-20)

Features

- Added 'bn' : 'Bengali' (#39, #44)

Deprecations and Removals

- 'ht' : 'Haitian Creole' (removed by Google) (#43)

1.6.8 1.1.5 (2016-05-13)

Bugfixes

- Fixed HTTP 403s by updating the client argument to reflect new API usage (#32, #33)

1.6.9 1.1.4 (2016-02-22)

Features

- Spun-off token calculation to gTTS-Token (#23, #29)

1.6.10 1.1.3 (2016-01-24)

Bugfixes

- `gtts-cli` works with Python 3 (#20)
- Better support for non-ASCII characters (#21, #22)

Misc

- Moved out gTTS token to its own module (#19)

1.6.11 1.1.2 (2016-01-13)

Features

- Added gTTS token (tk url parameter) calculation (#14, #15, #17)

1.6.12 1.0.7 (2015-10-07)

Features

- Added `stdout` support to `gtts-cli`, text now an argument rather than an option (#10)

1.6.13 1.0.6 (2015-07-30)

Features

- Raise an exception on bad HTTP response (4xx or 5xx) (#8)

Bugfixes

- Added `client=t` parameter for the api HTTP request (#8)

1.6.14 1.0.5 (2015-07-15)

Features

- `write_to_fp()` to write to a file-like object (#6)

1.6.15 1.0.4 (2015-05-11)

Features

- Added Languages: *zh-yue* : 'Chinese (Cantonese)', *en-uk* : 'English (United Kingdom)', *pt-br* : 'Portuguese (Brazil)', *es-es* : 'Spanish (Spain)', *es-us* : 'Spanish (United StateS)', *zh-cn* : 'Chinese (Mandarin/China)', *zh-tw* : 'Chinese (Mandarin/Taiwan)' (#4)

Bugfixes

- `gtts-cli` print version and pretty printed available languages, language codes are now case insensitive (#4)

1.6.16 1.0.3 (2014-11-21)

Features

- Added Languages: 'en-us' : 'English (United States)', 'en-au' : 'English (Australia)' (#3)

1.6.17 1.0.2 (2014-05-15)

Features

- Python 3 support

1.6.18 1.0.1 (2014-05-15)

Misc

- SemVer versioning, CI changes

1.6.19 1.0 (2014-05-08)

Features

- Initial release

CHAPTER 2

Misc

- genindex
- modindex

Python Module Index

g

Index

Symbols

A

E

G

I

L

O

P

R

S

T

W