# 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.

## 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 "|".

## 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*) [source]

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*) [source]

Re-form words cut by end-of-line hyphens.

Remove "<hyphen><newline>".

**`gtts.tokenizer.pre_processors.tone_marks`**(*text*) [source]

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*) [source]

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.

## 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.colon()`** [source]

Colon case.

Match a colon ":" only if not preceeded by a digit. Mainly to prevent a cut in the middle of time notations e.g. 10:01

**`gtts.tokenizer.tokenizer_cases.legacy_all_punctuation()`** [source]

Match all punctuation.

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

**`gtts.tokenizer.tokenizer_cases.other_punctuation()`** [source]

Match other punctuation.

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

**`gtts.tokenizer.tokenizer_cases.period_comma()`** [source]

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()` [source]

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.

## Minimizing

The Google Translate text-to-speech API accepts a maximum of **100 characters**.

If after tokenization any of the tokens is larger than 100 characters, it will be split in two:

- On the last space character that is closest to, but before the 100th character;
- Between the 100th and 101st characters if there's no space.

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

*class* `gtts.tokenizer.core.RegexBuilder`(*pattern_args, pattern_func, flags=0*) [source]

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*) [source]

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='\1!'),
(re.compile('(ipsum)', re.IGNORECASE), repl='\1!')
```

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*) [source]

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*) [source]

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*) [source]

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=<RegexFlag.IGNORECASE: 2>*) [source]

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*) [source]

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**= *[('Esq.', 'Esquire')]*

**symbols.ALL_PUNC**= *'?!？！.,¡()[]¿…‥،;:—。，、：\n'*

**symbols.TONE_MARKS**= *'?!？！'*