mwparserfromhell Documentation

Release 0.3.2

Ben Kurtovic

September 01, 2013

CONTENTS

`mwparserfromhell` (the *MediaWiki Parser from Hell*) is a Python package that provides an easy-to-use and outrageously powerful parser for MediaWiki wikicode. It supports Python 2 and Python 3.

Developed by Earwig with contributions from Σ, Legoktm, and others. Development occurs on GitHub.

CHAPTER

ONE

INSTALLATION

The easiest way to install the parser is through the Python Package Index, so you can install the latest release with `pip install mwparserfromhell` (get pip). Alternatively, get the latest development version:

```
git clone https://github.com/earwig/mwparserfromhell.git
cd mwparserfromhell
python setup.py install
```

If you get `error: Unable to find vcvarsall.bat` while installing, this is because Windows can't find the compiler for C extensions. Consult this StackOverflow question for help. You can also set `ext_modules` in `setup.py` to an empty list to prevent the extension from building.

You can run the comprehensive unit testing suite with `python setup.py test`.

CHAPTER

TWO

CONTENTS

2.1 Usage

Normal usage is rather straightforward (where `text` is page text):

```
>>> import mwparserfromhell
>>> wikicode = mwparserfromhell.parse(text)
```

`wikicode` is a `mwparserfromhell.Wikicode` object, which acts like an ordinary `unicode` object (or `str` in Python 3) with some extra methods. For example:

```
>>> text = "I has a template! {{foo|bar|baz|eggs=spam}} See it?"
>>> wikicode = mwparserfromhell.parse(text)
>>> print wikicode
I has a template! {{foo|bar|baz|eggs=spam}} See it?
>>> templates = wikicode.filter_templates()
>>> print templates
['{{foo|bar|baz|eggs=spam}}']
>>> template = templates[0]
>>> print template.name
foo
>>> print template.params
['bar', 'baz', 'eggs=spam']
>>> print template.get(1).value
bar
>>> print template.get("eggs").value
spam
```

Since nodes can contain other nodes, getting nested templates is trivial:

```
>>> text = "{{foo|{{bar}}={{baz|{{spam}}}}}}"
>>> mwparserfromhell.parse(text).filter_templates()
['{{foo|{{bar}}={{baz|{{spam}}}}}}', '{{bar}}', '{{baz|{{spam}}}}', '{{spam}}']
```

You can also pass *recursive=False* to `filter_templates()` and explore templates manually. This is possible because nodes can contain additional `Wikicode` objects:

```
>>> code = mwparserfromhell.parse("{{foo|this {{includes a|template}}}}")
>>> print code.filter_templates(recursive=False)
['{{foo|this {{includes a|template}}}}']
>>> foo = code.filter_templates(recursive=False)[0]
>>> print foo.get(1).value
this {{includes a|template}}
>>> print foo.get(1).value.filter_templates()[0]
```

```
{{includes a|template}}
>>> print foo.get(1).value.filter_templates()[0].get(1).value
template
```

Templates can be easily modified to add, remove, or alter params. `Wikicode` objects can be treated like lists, with `append()`, `insert()`, `remove()`, `replace()`, and more. They also have a `matches()` method for comparing page or template names, which takes care of capitalization and whitespace:

```
>>> text = "{{cleanup}} '''Foo''' is a [[bar]]. {{uncategorized}}"
>>> code = mwparserfromhell.parse(text)
>>> for template in code.filter_templates():
...     if template.name.matches("Cleanup") and not template.has("date"):
...         template.add("date", "July 2012")
...
>>> print code
{{cleanup|date=July 2012}} '''Foo''' is a [[bar]]. {{uncategorized}}
>>> code.replace("{{uncategorized}}", "{{bar-stub}}")
>>> print code
{{cleanup|date=July 2012}} '''Foo''' is a [[bar]]. {{bar-stub}}
>>> print code.filter_templates()
['{{cleanup|date=July 2012}}', '{{bar-stub}}']
```

You can then convert `code` back into a regular `unicode` object (for saving the page!) by calling `unicode()` on it:

```
>>> text = unicode(code)
>>> print text
{{cleanup|date=July 2012}} '''Foo''' is a [[bar]]. {{bar-stub}}
>>> text == code
True
```

(Likewise, use `str(code)` in Python 3.)

For more tips, check out `Wikicode's full method list` and the `list of Nodes`.

2.2 Integration

`mwparserfromhell` is used by and originally developed for EarwigBot; `Page` objects have a `parse()` method that essentially calls `mwparserfromhell.parse()` on `get()`.

If you're using Pywikipedia, your code might look like this:

```
import mwparserfromhell
import wikipedia as pywikibot
def parse(title):
    site = pywikibot.getSite()
    page = pywikibot.Page(site, title)
    text = page.get()
    return mwparserfromhell.parse(text)
```

If you're not using a library, you can parse templates in any page using the following code (via the API):

```
import json
import urllib
import mwparserfromhell
API_URL = "http://en.wikipedia.org/w/api.php"
def parse(title):
    raw = urllib.urlopen(API_URL, data).read()
    res = json.loads(raw)
```

```
    text = res["query"]["pages"].values()[0]["revisions"][0]["*"]
    return mwparserfromhell.parse(text)
```

2.3 Changelog

2.3.1 v0.3.2

Released September 1, 2013 (changes):

- Added support for Python 3.2 (along with current support for 3.3 and 2.7).
- Renamed `Template.remove()`'s first argument from *name* to *param*, which now accepts `Parameter` objects in addition to parameter name strings.

2.3.2 v0.3.1

Released August 29, 2013 (changes):

- Fixed a parser bug involving URLs nested inside other markup.
- Fixed some typos.

2.3.3 v0.3

Released August 24, 2013 (changes):

- Added complete support for HTML `Tags`, including forms like `<ref>foo</ref>`, `<ref name="bar"/>`, and wiki-markup tags like bold (`'''`), italics (`''`), and lists (`*`, `#`, `;` and `:`).
- Added support for `ExternalLinks` (`http://example.com/` and `[http://example.com/ Example]`).
- `Wikicode`'s `filter()` methods are now passed *recursive=True* by default instead of *False*. **This is a breaking change if you rely on any filter() methods being non-recursive by default.**
- Added a `matches()` method to `Wikicode` for page/template name comparisons.
- The *obj* param of `Wikicode.insert_before()`, `insert_after()`, `replace()`, and `remove()` now accepts `Wikicode` objects and strings representing parts of wikitext, instead of just nodes. These methods also make all possible substitutions instead of just one.
- Renamed `Template.has_param()` to `has()` for consistency with `Template`'s other methods; `has_param()` is now an alias.
- The C tokenizer extension now works on Python 3 in addition to Python 2.7.
- Various bugfixes, internal changes, and cleanup.

2.3.4 v0.2

Released June 20, 2013 (changes):

- The parser now fully supports Python 3 in addition to Python 2.7.
- Added a C tokenizer extension that is significantly faster than its Python equivalent. It is enabled by default (if available) and can be toggled by setting `mwparserfromhell.parser.use_c` to a boolean value.

- Added a complete set of unit tests covering parsing and wikicode manipulation.
- Renamed `filter_links()` to `filter_wikilinks()` (applies to `ifilter()` as well).
- Added filter methods for `Arguments`, `Comments`, `Headings`, and `HTMLEntities`.
- Added *before* param to `Template.add()`; renamed *force_nonconformity* to *preserve_spacing*.
- Added *include_lead* param to `Wikicode.get_sections()`.
- Removed *flat* param from `get_sections()`.
- Removed *force_no_field* param from `Template.remove()`.
- Added support for Travis CI.
- Added note about Windows build issue in the README.
- The tokenizer will limit itself to a realistic recursion depth to prevent errors and unreasonably long parse times.
- Fixed how some nodes' attribute setters handle input.
- Fixed multiple bugs in the tokenizer's handling of invalid markup.
- Fixed bugs in the implementation of `SmartList` and `StringMixIn`.
- Fixed some broken example code in the README; other copyedits.
- Other bugfixes and code cleanup.

2.3.5 v0.1.1

Released September 21, 2012 (changes):

- Added support for `Comments` (`<!-- foo -->`) and `Wikilinks` (`[[foo]]`).
- Added corresponding `ifilter_links()` and `filter_links()` methods to `Wikicode`.
- Fixed a bug when parsing incomplete templates.
- Fixed `strip_code()` to affect the contents of headings.
- Various copyedits in documentation and comments.

2.3.6 v0.1

Released August 23, 2012:

- Initial release.

2.4 mwparserfromhell

2.4.1 mwparserfromhell Package

`mwparserfromhell` Package

mwparserfromhell (the MediaWiki Parser from Hell) is a Python package that provides an easy-to-use and outrageously powerful parser for MediaWiki wikicode.

compat Module

Implements support for both Python 2 and Python 3 by defining common types in terms of their Python 2/3 variants. For example, `str` is set to `unicode` on Python 2 but `str` on Python 3; likewise, `bytes` is `str` on 2 but `bytes` on 3. These types are meant to be imported directly from within the parser's modules.

smart_list Module

This module contains the `SmartList` type, as well as its `_ListProxy` child, which together implement a list whose sublists reflect changes made to the main list, and vice-versa.

class mwparserfromhell.smart_list.**SmartList**(*iterable=None*)

Bases: mwparserfromhell.smart_list._SliceNormalizerMixIn, list

Implements the `list` interface with special handling of sublists.

When a sublist is created (by `list[i:j]`), any changes made to this list (such as the addition, removal, or replacement of elements) will be reflected in the sublist, or vice-versa, to the greatest degree possible. This is implemented by having sublists - instances of the `_ListProxy` type - dynamically determine their elements by storing their slice info and retrieving that slice from the parent. Methods that change the size of the list also change the slice info. For example:

```
>>> parent = SmartList([0, 1, 2, 3])
>>> parent
[0, 1, 2, 3]
>>> child = parent[2:]
>>> child
[2, 3]
>>> child.append(4)
>>> child
[2, 3, 4]
>>> parent
[0, 1, 2, 3, 4]
```

append(*item*)

L.append(object) – append object to end

extend(*item*)

L.extend(iterable) – extend list by appending elements from the iterable

insert(*index*, *item*)

L.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove(*item*)

L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse()

L.reverse() – reverse *IN PLACE*

sort(*cmp=None*, *key=None*, *reverse=None*)

L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

class mwparserfromhell.smart_list.**_ListProxy**(*parent*, *sliceinfo*)

Bases: mwparserfromhell.smart_list._SliceNormalizerMixIn, list

Implement the `list` interface by getting elements from a parent.

This is created by a `SmartList` object when slicing. It does not actually store the list at any time; instead, whenever the list is needed, it builds it dynamically using the `_render()` method.

append(*item*)
L.append(object) – append object to end

count(*value*) → integer – return number of occurrences of value

extend(*item*)
L.extend(iterable) – extend list by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert(*index*, *item*)
L.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.

remove(*item*)
L.remove(value) – remove first occurrence of value. Raises ValueError if the value is not present.

reverse()
L.reverse() – reverse *IN PLACE*

sort(*cmp=None*, *key=None*, *reverse=None*)
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

`string_mixin` Module

This module contains the `StringMixIn` type, which implements the interface for the `unicode` type (`str` on py3k) in a dynamic manner.

class `mwparserfromhell.string_mixin.`**StringMixIn**
Implement the interface for `unicode`/`str` in a dynamic manner.

To use this class, inherit from it and override the `__unicode__()` method (same on py3k) to return the string representation of the object. The various string methods will operate on the value of `__unicode__()` instead of the immutable `self` like the regular `str` type.

capitalize() → unicode
Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.

center(*width*[, *fillchar*]) → unicode
Return S centered in a Unicode string of length width. Padding is done using the specified fill character (default is a space)

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in Unicode string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode([*encoding*[, *errors*]]) → string or unicode
Decodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that is able to handle UnicodeDecodeErrors.

encode([*encoding*[, *errors*]]) → string or unicode
Encodes S using the codec registered for encoding. encoding defaults to the default encoding. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a

UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs([*tabsize*]) → unicode
Return a copy of S where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → unicode
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Like S.find() but raise ValueError when the substring is not found.

isalnum() → bool
Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

isalpha() → bool
Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

isdecimal() → bool
Return True if there are only decimal characters in S, False otherwise.

isdigit() → bool
Return True if all characters in S are digits and there is at least one character in S, False otherwise.

islower() → bool
Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

isnumeric() → bool
Return True if there are only numeric characters in S, False otherwise.

isspace() → bool
Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

istitle() → bool
Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper() → bool
Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

join(*iterable*) → unicode
Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

ljust(*width*[, *fillchar*]) → int
Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

lower() → unicode
Return a copy of the string S converted to lowercase.

lstrip([*chars*]) → unicode
Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

partition(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.

replace(*old*, *new*[, *count*]) → unicode
Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Like S.rfind() but raise ValueError when the substring is not found.

rjust(*width*[, *fillchar*]) → unicode
Return S right-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).

rpartition(*sep) -> (head*, *sep*, *tail*)
Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and S.

rsplit([*sep*[, *maxsplit*]]) → list of strings
Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.

rstrip([*chars*]) → unicode
Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

split([*sep*[, *maxsplit*]]) → list of strings
Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

splitlines(*keepends=False*) → list of strings
Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip([*chars*]) → unicode
Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None,

remove characters in chars instead. If chars is a str, it will be converted to unicode before stripping

swapcase() → unicode
Return a copy of S with uppercase characters converted to lowercase and vice versa.

title() → unicode
Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

translate(*table*) → unicode
Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, Unicode strings or None. Unmapped characters are left untouched. Characters mapped to None are deleted.

upper() → unicode
Return a copy of S converted to uppercase.

zfill(*width*) → unicode
Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

`definitions` Module

Contains data about certain markup, like HTML tags and external links.

mwparserfromhell.definitions.**get_html_tag**(*markup*)
Return the HTML tag associated with the given wiki-markup.

mwparserfromhell.definitions.**is_parsable**(*tag*)
Return if the given *tag*'s contents should be passed to the parser.

mwparserfromhell.definitions.**is_visible**(*tag*)
Return whether or not the given *tag* contains visible text.

mwparserfromhell.definitions.**is_single**(*tag*)
Return whether or not the given *tag* can exist without a close tag.

mwparserfromhell.definitions.**is_single_only**(*tag*)
Return whether or not the given *tag* must exist without a close tag.

mwparserfromhell.definitions.**is_scheme**(*scheme*, *slashes=True*, *reverse=False*)
Return whether *scheme* is valid for external links.

`utils` Module

This module contains accessory functions that wrap around existing ones to provide additional functionality.

mwparserfromhell.utils.**parse_anything**(*value*, *context=0*)
Return a `Wikicode` for *value*, allowing multiple types.

This differs from `Parser.parse()` in that we accept more than just a string to be parsed. Unicode objects (strings in py3k), strings (bytes in py3k), integers (converted to strings), `None`, existing `Node` or `Wikicode` objects, as well as an iterable of these types, are supported. This is used to parse input on-the-fly by various methods of `Wikicode` and others like `Template`, such as `wikicode.insert()` or setting `template.name`.

If given, *context* will be passed as a starting context to the parser. This is helpful when this function is used inside node attribute setters. For example, `ExternalLink`'s `url` setter sets *context* to `contexts.EXT_LINK_URI` to prevent the URL itself from becoming an `ExternalLink`.

wikicode Module

class mwparserfromhell.wikicode.**Wikicode**(*nodes*)
Bases: mwparserfromhell.string_mixin.StringMixIn

A Wikicode is a container for nodes that operates like a string.

Additionally, it contains methods that can be used to extract data from or modify the nodes, implemented in an interface similar to a list. For example, index() can get the index of a node in the list, and insert() can add a new node at that index. The filter() series of functions is very useful for extracting and iterating over, for example, all of the templates in the object.

append(*value*)
Insert *value* at the end of the list of nodes.

value can be anything parasable by parse_anything().

filter(*recursive=True*, *matches=None*, *flags=50*, *forcetype=None*)
Return a list of nodes within our list matching certain conditions.

This is equivalent to calling list() on ifilter().

filter_arguments(***kw*)
Iterate over arguments.

This is equivalent to filter() with *forcetype* set to Argument.

filter_comments(***kw*)
Iterate over comments.

This is equivalent to filter() with *forcetype* set to Comment.

filter_external_links(***kw*)
Iterate over external_links.

This is equivalent to filter() with *forcetype* set to ExternalLink.

filter_headings(***kw*)
Iterate over headings.

This is equivalent to filter() with *forcetype* set to Heading.

filter_html_entities(***kw*)
Iterate over html_entities.

This is equivalent to filter() with *forcetype* set to HTMLEntity.

filter_tags(***kw*)
Iterate over tags.

This is equivalent to filter() with *forcetype* set to Tag.

filter_templates(***kw*)
Iterate over templates.

This is equivalent to filter() with *forcetype* set to Template.

filter_text(***kw*)
Iterate over text.

This is equivalent to filter() with *forcetype* set to Text.

filter_wikilinks(***kw*)
Iterate over wikilinks.

This is equivalent to filter() with *forcetype* set to Wikilink.

get(*index*)
Return the *index*th node within the list of nodes.

get_sections(*levels=None*, *matches=None*, *flags=50*, *include_lead=None*, *include_headings=True*)
Return a list of sections within the page.

Sections are returned as `Wikicode` objects with a shared node list (implemented using `SmartList`) so that changes to sections are reflected in the parent Wikicode object.

Each section contains all of its subsections. If *levels* is given, it should be a iterable of integers; only sections whose heading levels are within it will be returned. If *matches* is given, it should be a regex to be matched against the titles of section headings; only sections whose headings match the regex will be included. *flags* can be used to override the default regex flags (see `ifilter()`) if *matches* is used.

If *include_lead* is `True`, the first, lead section (without a heading) will be included in the list; `False` will not include it; the default will include it only if no specific *levels* were given. If *include_headings* is `True`, the section's beginning `Heading` object will be included; otherwise, this is skipped.

get_tree()
Return a hierarchical tree representation of the object.

The representation is a string makes the most sense printed. It is built by calling `_get_tree()` on the `Wikicode` object and its children recursively. The end result may look something like the following:

```
>>> text = "Lorem ipsum {{foo|bar|{{baz}}|spam=eggs}}"
>>> print mwparserfromhell.parse(text).get_tree()
Lorem ipsum
{{
      foo
    | 1
    = bar
    | 2
    = {{
            baz
      }}
    | spam
    = eggs
}}
```

ifilter(*recursive=True*, *matches=None*, *flags=50*, *forcetype=None*)
Iterate over nodes in our list matching certain conditions.

If *recursive* is `True`, we will iterate over our children and all descendants of our children, otherwise just our immediate children. If *matches* is given, we will only yield the nodes that match the given regular expression (with `re.search()`). The default flags used are `re.IGNORECASE`, `re.DOTALL`, and `re.UNICODE`, but custom flags can be specified by passing *flags*. If *forcetype* is given, only nodes that are instances of this type are yielded.

ifilter_arguments(***kw*)
Iterate over arguments.

This is equivalent to `ifilter()` with *forcetype* set to `Argument`.

ifilter_comments(***kw*)
Iterate over comments.

This is equivalent to `ifilter()` with *forcetype* set to `Comment`.

ifilter_external_links(***kw*)
Iterate over external_links.

This is equivalent to `ifilter()` with *forcetype* set to `ExternalLink`.

ifilter_headings(***kw*)
Iterate over headings.

This is equivalent to `ifilter()` with *forcetype* set to `Heading`.

ifilter_html_entities(***kw*)
Iterate over html_entities.

This is equivalent to `ifilter()` with *forcetype* set to `HTMLEntity`.

ifilter_tags(***kw*)
Iterate over tags.

This is equivalent to `ifilter()` with *forcetype* set to `Tag`.

ifilter_templates(***kw*)
Iterate over templates.

This is equivalent to `ifilter()` with *forcetype* set to `Template`.

ifilter_text(***kw*)
Iterate over text.

This is equivalent to `ifilter()` with *forcetype* set to `Text`.

ifilter_wikilinks(***kw*)
Iterate over wikilinks.

This is equivalent to `ifilter()` with *forcetype* set to `Wikilink`.

index(*obj*, *recursive=False*)
Return the index of *obj* in the list of nodes.

Raises `ValueError` if *obj* is not found. If *recursive* is `True`, we will look in all nodes of ours and their descendants, and return the index of our direct descendant node within *our* list of nodes. Otherwise, the lookup is done only on direct descendants.

insert(*index*, *value*)
Insert *value* at *index* in the list of nodes.

value can be anything parasable by `parse_anything()`, which includes strings or other `Wikicode` or `Node` objects.

insert_after(*obj*, *value*, *recursive=True*)
Insert *value* immediately after *obj* in the list of nodes.

obj can be either a string, a `Node`, or other `Wikicode` object (as created by `get_sections()`, for example). *value* can be anything parasable by `parse_anything()`. If *recursive* is `True`, we will try to find *obj* within our child nodes even if it is not a direct descendant of this `Wikicode` object. If *obj* is not found, `ValueError` is raised.

insert_before(*obj*, *value*, *recursive=True*)
Insert *value* immediately before *obj* in the list of nodes.

obj can be either a string, a `Node`, or other `Wikicode` object (as created by `get_sections()`, for example). *value* can be anything parasable by `parse_anything()`. If *recursive* is `True`, we will try to find *obj* within our child nodes even if it is not a direct descendant of this `Wikicode` object. If *obj* is not found, `ValueError` is raised.

matches(*other*)
Do a loose equivalency test suitable for comparing page names.

other can be any string-like object, including `Wikicode`. This operation is symmetric; both sides are adjusted. Specifically, whitespace and markup is stripped and the first letter's case is normalized. Typical usage is `if template.name.matches("stub"): ...`.

nodes
A list of `Node` objects.

This is the internal data actually stored within a `Wikicode` object.

remove(*obj*, *recursive=True*)
Remove *obj* from the list of nodes.

obj can be either a string, a `Node`, or other `Wikicode` object (as created by `get_sections()`, for example). If *recursive* is `True`, we will try to find *obj* within our child nodes even if it is not a direct descendant of this `Wikicode` object. If *obj* is not found, `ValueError` is raised.

replace(*obj*, *value*, *recursive=True*)
Replace *obj* with *value* in the list of nodes.

obj can be either a string, a `Node`, or other `Wikicode` object (as created by `get_sections()`, for example). *value* can be anything parasable by `parse_anything()`. If *recursive* is `True`, we will try to find *obj* within our child nodes even if it is not a direct descendant of this `Wikicode` object. If *obj* is not found, `ValueError` is raised.

set(*index*, *value*)
Set the `Node` at *index* to *value*.

Raises `IndexError` if *index* is out of range, or `ValueError` if *value* cannot be coerced into one `Node`. To insert multiple nodes at an index, use `get()` with either `remove()` and `insert()` or `replace()`.

strip_code(*normalize=True*, *collapse=True*)
Return a rendered string without unprintable code such as templates.

The way a node is stripped is handled by the `__showtree__()` method of `Node` objects, which generally return a subset of their nodes or `None`. For example, templates and tags are removed completely, links are stripped to just their display part, headings are stripped to just their title. If *normalize* is `True`, various things may be done to strip code further, such as converting HTML entities like `&Sigma;`, `&#931;`, and `&#x3a3;` to Σ. If *collapse* is `True`, we will try to remove excess whitespace as well (three or more newlines are converted to two, for example).

Subpackages

nodes Package

nodes Package This package contains `Wikicode` "nodes", which represent a single unit of wikitext, such as a Template, an HTML tag, a Heading, or plain text. The node "tree" is far from flat, as most types can contain additional `Wikicode` types within them - and with that, more nodes. For example, the name of a `Template` is a `Wikicode` object that can contain text or more templates.

class `mwparserfromhell.nodes.`**Node**
Represents the base Node type, demonstrating the methods to override.

`__unicode__()` must be overridden. It should return a `unicode` or (`str` in py3k) representation of the node. If the node contains `Wikicode` objects inside of it, `__iternodes__()` should be overridden to yield tuples of (`wikicode`, `node_in_wikicode`) for each node in each wikicode, as well as the node itself (`None`, `self`). If the node is printable, `__strip__()` should be overridden to return the printable version of the node - it does not have to be a string, but something that can be converted to a string with `str()`. Finally, `__showtree__()` can be overridden to build a nice tree representation of the node, if desired, for `get_tree()`.

argument Module

class mwparserfromhell.nodes.argument.**Argument**(*name*, *default=None*)

Bases: mwparserfromhell.nodes.Node

Represents a template argument substitution, like {{{foo}}}.

default

The default value to substitute if none is passed.

This will be None if the argument wasn't defined with one. The MediaWiki parser handles this by rendering the argument itself in the result, complete braces. To have the argument render as nothing, set default to "" ({{{arg}}} vs. {{{arg|}}}).

name

The name of the argument to substitute.

comment Module

class mwparserfromhell.nodes.comment.**Comment**(*contents*)

Bases: mwparserfromhell.nodes.Node

Represents a hidden HTML comment, like <!-- foobar -->.

contents

The hidden text contained between <!-- and -->.

external_link Module

class mwparserfromhell.nodes.external_link.**ExternalLink**(*url*, *title=None*, *brackets=True*)

Bases: mwparserfromhell.nodes.Node

Represents an external link, like [http://example.com/ Example].

brackets

Whether to enclose the URL in brackets or display it straight.

title

The link title (if given), as a Wikicode object.

url

The URL of the link target, as a Wikicode object.

heading Module

class mwparserfromhell.nodes.heading.**Heading**(*title*, *level*)

Bases: mwparserfromhell.nodes.Node

Represents a section heading in wikicode, like == Foo ==.

level

The heading level, as an integer between 1 and 6, inclusive.

title

The title of the heading, as a Wikicode object.

html_entity Module

class mwparserfromhell.nodes.html_entity.**HTMLEntity**(*value*, *named=None*, *hexadecimal=False*, *hex_char=u'x'*)

Bases: mwparserfromhell.nodes.Node

Represents an HTML entity, like , either named or unnamed.

hex_char
If the value is hexadecimal, this is the letter denoting that.

For example, the hex_char of `"&#x1234;"` is `"x"`, whereas the hex_char of `"&#X1234;"` is `"X"`. Lowercase and uppercase `x` are the only values supported.

hexadecimal
If unnamed, this is whether the value is hexadecimal or decimal.

named
Whether the entity is a string name for a codepoint or an integer.

For example, `&Sigma;`, `&#931;`, and `&#x3a3;` refer to the same character, but only the first is "named", while the others are integer representations of the codepoint.

normalize()
Return the unicode character represented by the HTML entity.

value
The string value of the HTML entity.

tag Module

class mwparserfromhell.nodes.tag.**Tag**(*tag*, *contents=None*, *attrs=None*, *wiki_markup=None*, *self_closing=False*, *invalid=False*, *implicit=False*, *padding=u''*, *closing_tag=None*)
Bases: mwparserfromhell.nodes.Node

Represents an HTML-style tag in wikicode, like `<ref>`.

add(*name*, *value=None*, *quoted=True*, *pad_first=u' '*, *pad_before_eq=u''*, *pad_after_eq=u''*)
Add an attribute with the given *name* and *value*.

name and *value* can be anything parasable by `utils.parse_anything()`; *value* can be omitted if the attribute is valueless. *quoted* is a bool telling whether to wrap the *value* in double quotes (this is recommended). *pad_first*, *pad_before_eq*, and *pad_after_eq* are whitespace used as padding before the name, before the equal sign (or after the name if no value), and after the equal sign (ignored if no value), respectively.

attributes
The list of attributes affecting the tag.

Each attribute is an instance of `Attribute`.

closing_tag
The closing tag, as a `Wikicode` object.

This will usually equal `tag`, unless there is additional spacing, comments, or the like.

contents
The contents of the tag, as a `Wikicode` object.

get(*name*)
Get the attribute with the given *name*.

The returned object is a `Attribute` instance. Raises `ValueError` if no attribute has this name. Since multiple attributes can have the same name, we'll return the last match, since all but the last are ignored by the MediaWiki parser.

has(*name*)
Return whether any attribute in the tag has the given *name*.

Note that a tag may have multiple attributes with the same name, but only the last one is read by the MediaWiki parser.

implicit
Whether the tag is implicitly self-closing, with no ending slash.

This is only possible for specific "single" tags like `<br>` and `<li>`. See `definitions.is_single()`. This field only has an effect if `self_closing` is also `True`.

invalid
Whether the tag starts with a backslash after the opening bracket.

This makes the tag look like a lone close tag. It is technically invalid and is only parsable Wikicode when the tag itself is single-only, like `<br>` and `<img>`. See `definitions.is_single_only()`.

padding
Spacing to insert before the first closing `>`.

remove(*name*)
Remove all attributes with the given *name*.

self_closing
Whether the tag is self-closing with no content (like `<br/>`).

tag
The tag itself, as a `Wikicode` object.

wiki_markup
The wikified version of a tag to show instead of HTML.

If set to a value, this will be displayed instead of the brackets. For example, set to `''` to replace `<i>` or `----` to replace `<hr>`.

`template` Module

class `mwparserfromhell.nodes.template.`**Template**(*name*, *params=None*)
Bases: `mwparserfromhell.nodes.Node`

Represents a template in wikicode, like `{{foo}}`.

add(*name*, *value*, *showkey=None*, *before=None*, *preserve_spacing=True*)
Add a parameter to the template with a given *name* and *value*.

name and *value* can be anything parasable by `utils.parse_anything()`; pipes and equal signs are automatically escaped from *value* when appropriate.

If *showkey* is given, this will determine whether or not to show the parameter's name (e.g., `{{foo|bar}}`'s parameter has a name of `"1"` but it is hidden); otherwise, we'll make a safe and intelligent guess.

If *name* is already a parameter in the template, we'll replace its value while keeping the same whitespace around it. We will also try to guess the dominant spacing convention when adding a new parameter using `_get_spacing_conventions()`.

If *before* is given (either a `Parameter` object or a name), then we will place the parameter immediately before this one. Otherwise, it will be added at the end. If *before* is a name and exists multiple times in the template, we will place it before the last occurance. If *before* is not in the template, `ValueError` is raised. The argument is ignored if the new parameter already exists.

If *preserve_spacing* is `False`, we will avoid preserving spacing conventions when changing the value of an existing parameter or when adding a new one.

get(*name*)
Get the parameter whose name is *name*.

The returned object is a `Parameter` instance. Raises `ValueError` if no parameter has this name. Since multiple parameters can have the same name, we'll return the last match, since the last parameter is the only one read by the MediaWiki parser.

has(*name*, *ignore_empty=True*)
Return `True` if any parameter in the template is named *name*.

With *ignore_empty*, `False` will be returned even if the template contains a parameter with the name *name*, if the parameter's value is empty. Note that a template may have multiple parameters with the same name, but only the last one is read by the MediaWiki parser.

has_param(*name*, *ignore_empty=True*)
Alias for `has()`.

name
The name of the template, as a `Wikicode` object.

params
The list of parameters contained within the template.

remove(*param*, *keep_field=False*)
Remove a parameter from the template, identified by *param*.

If *param* is a `Parameter` object, it will be matched exactly, otherwise it will be treated like the *name* argument to `has()` and `get()`.

If *keep_field* is `True`, we will keep the parameter's name, but blank its value. Otherwise, we will remove the parameter completely *unless* other parameters are dependent on it (e.g. removing `bar` from `{{foo|bar|baz}}` is unsafe because `{{foo|baz}}` is not what we expected, so `{{foo||baz}}` will be produced instead).

If the parameter shows up multiple times in the template and *param* is not a `Parameter` object, we will remove all instances of it (and keep only one if *keep_field* is `True` - the first instance if none have dependents, otherwise the one with dependents will be kept).

`text` Module

class `mwparserfromhell.nodes.text.`**Text**(*value*)
Bases: `mwparserfromhell.nodes.Node`

Represents ordinary, unformatted text with no special properties.

value
The actual text itself.

`wikilink` Module

class `mwparserfromhell.nodes.wikilink.`**Wikilink**(*title*, *text=None*)
Bases: `mwparserfromhell.nodes.Node`

Represents an internal wikilink, like `[[Foo|Bar]]`.

text
The text to display (if any), as a `Wikicode` object.

title
The title of the linked page, as a `Wikicode` object.

Subpackages

extras Package

extras Package This package contains objects used by Nodes, but are not nodes themselves. This includes the parameters of Templates or the attributes of HTML tags.

attribute Module

class mwparserfromhell.nodes.extras.attribute.**Attribute**(*name*, *value=None*, *quoted=True*, *pad_first=u' '*, *pad_before_eq=u''*, *pad_after_eq=u''*)

Bases: mwparserfromhell.string_mixin.StringMixIn

Represents an attribute of an HTML tag.

This is used by Tag objects. For example, the tag <ref name="foo"> contains an Attribute whose name is "name" and whose value is "foo".

name
The name of the attribute as a Wikicode object.

pad_after_eq
Spacing to insert right after the equal sign.

pad_before_eq
Spacing to insert right before the equal sign.

pad_first
Spacing to insert right before the attribute.

quoted
Whether the attribute's value is quoted with double quotes.

value
The value of the attribute as a Wikicode object.

parameter Module

class mwparserfromhell.nodes.extras.parameter.**Parameter**(*name*, *value*, *showkey=True*)

Bases: mwparserfromhell.string_mixin.StringMixIn

Represents a paramater of a template.

For example, the template {{foo|bar|spam=eggs}} contains two Parameters: one whose name is "1", value is "bar", and showkey is False, and one whose name is "spam", value is "eggs", and showkey is True.

name
The name of the parameter as a Wikicode object.

showkey
Whether to show the parameter's key (i.e., its "name").

value
The value of the parameter as a Wikicode object.

parser Package

parser Package This package contains the actual wikicode parser, split up into two main modules: the tokenizer and the builder. This module joins them together under one interface.

class mwparserfromhell.parser.**Parser**
Represents a parser for wikicode.

Actual parsing is a two-step process: first, the text is split up into a series of tokens by the Tokenizer, and then the tokens are converted into trees of Wikicode objects and Nodes by the Builder.

parse(*text*, *context=0*)
Parse *text*, returning a Wikicode object tree.

builder Module

class mwparserfromhell.parser.builder.**Builder**
Combines a sequence of tokens into a tree of Wikicode objects.

To use, pass a list of Tokens to the build() method. The list will be exhausted as it is parsed and a Wikicode object will be returned.

_handle_argument()
Handle a case where an argument is at the head of the tokens.

_handle_attribute(*start*)
Handle a case where a tag attribute is at the head of the tokens.

_handle_comment()
Handle a case where an HTML comment is at the head of the tokens.

_handle_entity()
Handle a case where an HTML entity is at the head of the tokens.

_handle_external_link(*token*)
Handle when an external link is at the head of the tokens.

_handle_heading(*token*)
Handle a case where a heading is at the head of the tokens.

_handle_parameter(*default*)
Handle a case where a parameter is at the head of the tokens.

default is the value to use if no parameter name is defined.

_handle_tag(*token*)
Handle a case where a tag is at the head of the tokens.

_handle_template()
Handle a case where a template is at the head of the tokens.

_handle_token(*token*)
Handle a single token.

_handle_wikilink()
Handle a case where a wikilink is at the head of the tokens.

_pop(*wrap=True*)
Pop the current node list off of the stack.

If *wrap* is True, we will call _wrap() on the list.

_push()
Push a new node list onto the stack.

_wrap(*nodes*)
Properly wrap a list of nodes in a Wikicode object.

_write(*item*)
Append a node to the current node list.

build(*tokenlist*)
Build a Wikicode object from a list tokens and return it.

contexts Module This module contains various "context" definitions, which are essentially flags set during the tokenization process, either on the current parse stack (local contexts) or affecting all stacks (global contexts). They represent the context the tokenizer is in, such as inside a template's name definition, or inside a level-two heading. This is used to determine what tokens are valid at the current point and also if the current parsing route is invalid.

The tokenizer stores context as an integer, with these definitions bitwise OR'd to set them, AND'd to check if they're set, and XOR'd to unset them. The advantage of this is that contexts can have sub-contexts (as `FOO == 0b11` will cover `BAR == 0b10` and `BAZ == 0b01`).

Local (stack-specific) contexts:

- TEMPLATE
  - TEMPLATE_NAME
  - TEMPLATE_PARAM_KEY
  - TEMPLATE_PARAM_VALUE
- ARGUMENT
  - ARGUMENT_NAME
  - ARGUMENT_DEFAULT
- WIKILINK
  - WIKILINK_TITLE
  - WIKILINK_TEXT
- EXT_LINK
  - EXT_LINK_URI
  - EXT_LINK_TITLE
  - EXT_LINK_BRACKETS
- HEADING
  - HEADING_LEVEL_1
  - HEADING_LEVEL_2
  - HEADING_LEVEL_3
  - HEADING_LEVEL_4
  - HEADING_LEVEL_5
  - HEADING_LEVEL_6
- TAG
  - TAG_OPEN
  - TAG_ATTR
  - TAG_BODY
  - TAG_CLOSE

- STYLE
  - STYLE_ITALICS
  - STYLE_BOLD
  - STYLE_PASS_AGAIN
  - STYLE_SECOND_PASS
- DL_TERM
- SAFETY_CHECK
  - HAS_TEXT
  - FAIL_ON_TEXT
  - FAIL_NEXT
  - FAIL_ON_LBRACE
  - FAIL_ON_RBRACE
  - FAIL_ON_EQUALS

Global contexts:

- GL_HEADING

Aggregate contexts:

- FAIL
- UNSAFE
- DOUBLE
- INVALID_LINK

tokenizer Module

class mwparserfromhell.parser.tokenizer.**Tokenizer**

Creates a list of tokens from a string of wikicode.

END = <object object at 0x7fbcd7d36770>

MARKERS = [u'{', u'}', u'[', u']', u'<', u'>', u'|', u'=', u'&', u'"', u'#', u'*', u';', u':', u'/', u'-', u'\n', <object object at 0x7

MAX_CYCLES = 100000

MAX_DEPTH = 40

START = <object object at 0x7fbcd7d36760>

USES_C = False

_can_recurse()

Return whether or not our max recursion depth has been exceeded.

_context

The current token context.

_emit(*token*)

Write a token to the end of the current token stack.

_emit_all(*tokenlist*)

Write a series of tokens to the current stack at once.

_emit_first(*token*)
Write a token to the beginning of the current token stack.

_emit_style_tag(*tag*, *markup*, *body*)
Write the body of a tag and the tokens that should surround it.

_emit_text(*text*)
Write text to the current textbuffer.

_emit_text_then_stack(*text*)
Pop the current stack, write *text*, and then write the stack.

_fail_route()
Fail the current tokenization route.

Discards the current stack/context/textbuffer and raises `BadRoute`.

_handle_argument_end()
Handle the end of an argument at the head of the string.

_handle_argument_separator()
Handle the separator between an argument's name and default.

_handle_blacklisted_tag()
Handle the body of an HTML tag that is parser-blacklisted.

_handle_dl_term()
Handle the term in a description list (`foo` in `;foo:bar`).

_handle_end()
Handle the end of the stream of wikitext.

_handle_free_link_text(*punct*, *tail*, *this*)
Handle text in a free ext link, including trailing punctuation.

_handle_heading_end()
Handle the end of a section heading at the head of the string.

_handle_hr()
Handle a wiki-style horizontal rule (`----`) in the string.

_handle_invalid_tag_start()
Handle the (possible) start of an implicitly closing single tag.

_handle_list()
Handle a wiki-style list (`#`, `*`, `;`, `:`).

_handle_list_marker()
Handle a list marker at the head (`#`, `*`, `;`, `:`).

_handle_single_only_tag_end()
Handle the end of an implicitly closing single-only HTML tag.

_handle_single_tag_end()
Handle the stream end when inside a single-supporting HTML tag.

_handle_tag_close_close()
Handle the ending of a closing tag (`</foo>`).

_handle_tag_close_open(*data*, *token*)
Handle the closing of a open tag (`<foo>`).

_handle_tag_data(*data*, *text*)
Handle all sorts of *text* data inside of an HTML open tag.

_handle_tag_open_close()
Handle the opening of a closing tag (`</foo>`).

_handle_tag_space(*data*, *text*)
Handle whitespace (*text*) inside of an HTML open tag.

_handle_tag_text(*text*)
Handle regular *text* inside of an HTML open tag.

_handle_template_end()
Handle the end of a template at the head of the string.

_handle_template_param()
Handle a template parameter at the head of the string.

_handle_template_param_value()
Handle a template parameter's value at the head of the string.

_handle_wikilink_end()
Handle the end of a wikilink at the head of the string.

_handle_wikilink_separator()
Handle the separator between a wikilink's title and its text.

_is_free_link_end(*this*, *next*)
Return whether the current head is the end of a free link.

_parse(*context=0*, *push=True*)
Parse the wikicode string, using *context* for when to stop.

_parse_argument()
Parse an argument at the head of the wikicode string.

_parse_bold()
Parse wiki-style bold.

_parse_bracketed_uri_scheme()
Parse the URI scheme of a bracket-enclosed external link.

_parse_comment()
Parse an HTML comment at the head of the wikicode string.

_parse_entity()
Parse an HTML entity at the head of the wikicode string.

_parse_external_link(*brackets*)
Parse an external link at the head of the wikicode string.

_parse_free_uri_scheme()
Parse the URI scheme of a free (no brackets) external link.

_parse_heading()
Parse a section heading at the head of the wikicode string.

_parse_italics()
Parse wiki-style italics.

_parse_italics_and_bold()
Parse wiki-style italics and bold together (i.e., five ticks).

_parse_style()
Parse wiki-style formatting (`''`/`'''` for italics/bold).

_parse_tag()
Parse an HTML tag at the head of the wikicode string.

_parse_template()
Parse a template at the head of the wikicode string.

_parse_template_or_argument()
Parse a template or argument at the head of the wikicode string.

_parse_wikilink()
Parse an internal wikilink at the head of the wikicode string.

_pop(*keep_context=False*)
Pop the current stack/context/textbuffer, returing the stack.

If *keep_context* is `True`, then we will replace the underlying stack's context with the current stack's.

_push(*context=0*)
Add a new token stack, context, and textbuffer to the list.

_push_tag_buffer(*data*)
Write a pending tag attribute from *data* to the stack.

_push_textbuffer()
Push the textbuffer onto the stack as a Text node and clear it.

_read(*delta=0*, *wrap=False*, *strict=False*)
Read the value at a relative point in the wikicode.

The value is read from `self._head` plus the value of *delta* (which can be negative). If *wrap* is `False`, we will not allow attempts to read from the end of the string if `self._head + delta` is negative. If *strict* is `True`, the route will be failed (with `_fail_route()`) if we try to read from past the end of the string; otherwise, `self.END` is returned. If we try to read from before the start of the string, `self.START` is returned.

_really_parse_entity()
Actually parse an HTML entity and ensure that it is valid.

_really_parse_external_link(*brackets*)
Really parse an external link.

_really_parse_tag()
Actually parse an HTML tag, starting with the open (`<foo>`).

_remove_uri_scheme_from_textbuffer(*scheme*)
Remove the URI scheme of a new external link from the textbuffer.

_stack
The current token stack.

_textbuffer
The current textbuffer.

_verify_safe(*this*)
Make sure we are not trying to write an invalid character.

regex = <_sre.SRE_Pattern object at 0x1d4f030>

tag_splitter = <_sre.SRE_Pattern object at 0x1d488d0>

tokenize(*text*, *context=0*)
Build a list of tokens from a string of wikicode and return it.

exception mwparserfromhell.parser.tokenizer.**BadRoute**(*context=0*)
Raised internally when the current tokenization route is invalid.

tokens Module This module contains the token definitions that are used as an intermediate parsing data type - they are stored in a flat list, with each token being identified by its type and optional attributes. The token list is generated in a syntactically valid form by the `Tokenizer`, and then converted into the :py:class‘~.Wikicode‘ tree by the `Builder`.

class mwparserfromhell.parser.tokens.**Token**(***kwargs*)
A token stores the semantic meaning of a unit of wikicode.

class mwparserfromhell.parser.tokens.**Text**(***kwargs*)

class mwparserfromhell.parser.tokens.**TemplateOpen**(***kwargs*)

class mwparserfromhell.parser.tokens.**TemplateParamSeparator**(***kwargs*)

class mwparserfromhell.parser.tokens.**TemplateParamEquals**(***kwargs*)

class mwparserfromhell.parser.tokens.**TemplateClose**(***kwargs*)

class mwparserfromhell.parser.tokens.**ArgumentOpen**(***kwargs*)

class mwparserfromhell.parser.tokens.**ArgumentSeparator**(***kwargs*)

class mwparserfromhell.parser.tokens.**ArgumentClose**(***kwargs*)

class mwparserfromhell.parser.tokens.**WikilinkOpen**(***kwargs*)

class mwparserfromhell.parser.tokens.**WikilinkSeparator**(***kwargs*)

class mwparserfromhell.parser.tokens.**WikilinkClose**(***kwargs*)

class mwparserfromhell.parser.tokens.**ExternalLinkOpen**(***kwargs*)

class mwparserfromhell.parser.tokens.**ExternalLinkSeparator**(***kwargs*)

class mwparserfromhell.parser.tokens.**ExternalLinkClose**(***kwargs*)

class mwparserfromhell.parser.tokens.**HTMLEntityStart**(***kwargs*)

class mwparserfromhell.parser.tokens.**HTMLEntityNumeric**(***kwargs*)

class mwparserfromhell.parser.tokens.**HTMLEntityHex**(***kwargs*)

class mwparserfromhell.parser.tokens.**HTMLEntityEnd**(***kwargs*)

class mwparserfromhell.parser.tokens.**HeadingStart**(***kwargs*)

class mwparserfromhell.parser.tokens.**HeadingEnd**(***kwargs*)

class mwparserfromhell.parser.tokens.**CommentStart**(***kwargs*)

class mwparserfromhell.parser.tokens.**CommentEnd**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagOpenOpen**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagAttrStart**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagAttrEquals**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagAttrQuote**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagCloseOpen**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagCloseSelfclose**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagOpenClose**(***kwargs*)

class mwparserfromhell.parser.tokens.**TagCloseClose**(***kwargs*)

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

m