
django-classy-tags Documentation

Release 0.3.0

Jonas Obrist

August 28, 2016

1	Installation	3
2	Usage	5
2.1	A simple example	5
2.2	Defining options	5
2.3	Writing a block tag	6
2.4	Easy ‘as’ Tags	7
2.5	Inclusion Tags	7
2.6	Advanced Block Definition	8
3	Arguments in depth	9
3.1	Argument	9
3.2	KeywordArgument	9
3.3	IntegerArgument	9
3.4	ChoiceArgument	10
3.5	MultiValueArgument	10
3.6	MultiKeywordArgument	10
3.7	Flag	10
4	Reference	11
4.1	<code>classytags.arguments</code>	11
4.2	<code>classytags.blocks</code>	12
4.3	<code>classytags.core</code>	13
4.4	<code>classytags.exceptions</code>	14
4.5	<code>classytags.helpers</code>	14
4.6	<code>classytags.parser</code>	15
4.7	<code>classytags.utils</code>	16
4.8	<code>classytags.values</code>	17
5	Extending django-classy-tags	19
5.1	Creating a custom argument class	19
5.2	Custom argument parser	20
5.3	Example	20
6	Contribute to django-classy-tags	21
6.1	Review the documentation	21
6.2	Spread the word	21
6.3	Improve the test suite	21
6.4	Write code	21

7	Changelog	23
7.1	0.8.0	23
7.2	0.7.2	23
7.3	0.7.1	23
7.4	0.7.0	23
7.5	0.6.2	23
7.6	0.6.1	23
7.7	0.6.0	24
7.8	0.5.2	24
7.9	0.5.1	24
7.10	0.5.0	24
7.11	0.3.3	24
7.12	0.3.2	24
7.13	0.3.1	24
7.14	0.3.0	24
7.15	0.2.2	25
7.16	0.2.1	25
7.17	0.2.0	25
7.18	0.1.3	25
8	Indices and tables	27
	Python Module Index	29

django-classy-tags is an approach at making writing template tags in Django easier, shorter and more fun by providing an extensible argument parser which reduces most of the boiler plate code you usually have to write when coding custom template tags.

django-classy-tags does **no magic by design**. Thus you will not get automatic registering/loading of your tags like other solutions provide. You will not get automatic argument guessing from function signatures but rather you have to declare what arguments your tag accepts. There is also no magic in your template tag class either, it's just a subclass of `django.template.Node` which invokes a parser class to parse the arguments when it's initialized and resolves those arguments into keyword arguments in it's `render` method and calls it's `render_tag` method with those keyword arguments.

Contents:

Installation

To install `django-classy-tags`, please use `sudo pip install django-classy-tags`.

If you don't want to use `pip`, download the latest version from [pypi](#), unpack the tarball and run `sudo python setup.py install`.

`django-classy-tags` has no dependencies other than Django. Django 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9 and 1.10 are supported.

`django-classy-tags` supports Python 2.7, 3.3, 3.4 and 3.5.

2.1 A simple example

A very basic tag which takes no arguments and always returns 'hello world' would be:

```
from classytags.core import Tag
from django import template

register = template.Library()

class HelloWorld(Tag):
    name = 'hello_world'

    def render_tag(self, context):
        return 'hello world'

register.tag(HelloWorld)
```

Now let's explain this. To create a tag, you subclass `classytags.core.Tag` and define a `classytags.core.Tag.render_tag()` method which takes the context and any template tag options you define as arguments to the method. Since we did not define any options for this tag, it only takes context. The `classytags.core.Tag.render_tag()` method should always return a string.

`classytags.core.Tag.render_tag` on a tag class is what is used when registering the tag with a Django template tag library and also what will be used in the template.

2.2 Defining options

Defining options is done by setting the `classytags.core.Tag.options` attribute on your tag class to an instance of `classytags.core.Options`. The `Options` class takes any amount of argument objects or strings (called breakpoints) as initialization arguments.

Let's build a tag which takes a single argument and an optional 'as varname' argument:

```
from classytags.core import Tag, Options
from classytags.arguments import Argument
from django import template

register = template.Library()

class Hello(Tag):
```

```
name = 'hello'
options = Options(
    Argument('name'),
    'as',
    Argument('varname', required=False, resolve=False)
)

def render_tag(self, context, name, varname):
    output = 'hello %s' % name
    if varname:
        context[varname] = output
        return ''
    else:
        return output
```

```
register.tag(Hello)
```

In a template we could now do either `{% hello "world" %}` which would output `'hello world'` or `{% hello "world" as "varname" %}` which would output nothing but set the `{{ varname }}` template variable to `'hello world'`. You may also use `{% hello "world" as varname %}` to achieve the same result like the last example.

2.3 Writing a block tag

You can write tags which wrap a block (odelist) in the template. An example for this kind of tag is Django's built-in `{% with %}` tag.

To write the `{% with %}` tag from Django using `django-classy-tags` you would do:

```
from classytags.core import Tag, Options
from classytags.arguments import Argument
from django import template

register = template.Library()

class With(Tag):
    name = 'with'
    options = Options(
        Argument('variable'),
        'as',
        Argument('varname', resolve=False),
        blocks=[('endwith', 'odelist')],
    )

    def render_tag(self, context, variable, varname, nodelist):
        context.push()
        context[varname] = variable
        output = nodelist.render(context)
        context.pop()
        return output

register.tag(With)
```

2.3.1 Working with multiple blocks

If you're working with multiple, optional blocks, the nodelist is always credited to the leftmost block name.

For example the Django `for` tag accepts an optional `empty` block. Let's take following classytag options:

```
options = Options(
    CommaSeperatableMultiValueArgument('loopvars'),
    'in',
    arguments.Argument('values'),
    blocks=[('empty', 'pre_empty'), ('endfor', 'post_empty')],
)
```

If you use it with `{% for x in y %}hello{% empty %}world{% endfor %}` the `pre_empty` argument to your `classytags.arguments.Argument.render_tag()` would hold a nodelist containing `hello`, `post_empty` would contain `world`. Now if you have `{% for x in y%}{{ hello }}{% endfor %}`, `pre_empty` remains the same, but `post_empty` is an empty nodelist.

2.4 Easy 'as' Tags

There is a helper class for tags which store their output (optionally) in the context. This class is in `classytags.helpers.AsTag` and instead of defining a `render_tag` method, you define a `classytags.helpers.AsTag.get_value()` method which returns the value you want to either display or be put into the context.

Here is a small example:

```
from classytags.core import Options
from classytags.arguments import Argument
from classytags.helpers import AsTag
from django import template

register = template.Library()

class Dummy(AsTag):
    options = Options(
        'as',
        Argument('varname', resolve=False, required=False),
    )

    def get_value(self, context):
        return 'dummy'

register.tag(Dummy)
```

Now if you do `{% dummy %}` in your templates, it will output 'dummy' there. If you use `{% dummy as myvar %}` 'dummy' will be stored into the `myvar` context variable.

2.5 Inclusion Tags

A helper class for inclusion tags (template tags which render a template) is provided at `classytags.helpers.InclusionTag`. Instead of the usual `render_tag` method it provides two methods `classytags.helpers.InclusionTag.get_template()` which by default returns the attribute `classytags.helpers.InclusionTag.template` and defines the template to use for rendering. The method `classytags.helpers.InclusionTag.get_context()` should return a dictionary holding the

content to use for rendering the template. Both those methods get the context and the arguments of the tag passed as arguments.

A very simple example would be:

```
from classytags.core import Options
from classytags.arguments import Argument
from classytags.helpers import InclusionTag
from django import template

register = template.Library()

class Dummy(InclusionTag):
    template = 'dummy.html'

    def get_context(self, context):
        return {'varname': 'dummy'}

register.tag(Dummy)
```

With the following template for *dummy.html*:

```
varname: {{ varname }}
```

This would always render as `varname: dummy`.

2.6 Advanced Block Definition

Sometimes you might want to allow your blocktag to be terminated by a variable end tag to make templates more readable. This is for example done in the `block` tag in Django, where you can do `{% block myblock %}...{% endblock %}` as well as `{% block myblock %}...{% endblock myblock %}`. To do so in `classytags`, you have to use advanced block definitions using the `classytags.blocks.BlockDefinition` class together with the `classytags.blocks.VariableBlockName` class.

An example for a tag with the same signature as Django's `block` tag:

```
class Block(Tag):
    options = Options(
        Argument('name', resolve=False),
        blocks=[
            BlockDefinition('odelist', VariableBlockName('endblock %(value)s', 'name'), 'endblock')
        ]
    )
```

Arguments in depth

Examples given here are in tabular form. The non-standard options are a list of options to the argument class which are not the default value. The input is the single token in your template tag that gets consumed by this argument class. The output is the value you get in your `render_tag` method for the keyword of this argument.

Context for all examples: `{'name': 'classytags'}`

3.1 Argument

Table 3.1: Examples

Non-standard options	Input	Output
	name	'classytags'
	'name'	'name'
resolve=False	name	'name'
default='myvalue'		'myvalue'

3.2 KeywordArgument

Table 3.2: Examples

Non-standard options	Input	Output
	key=name	{'key': 'classytags'}
	key='name'	{'key': 'name'}
defaultkey='mykey'	name	{'defaultkey': 'classytags'}
defaultkey='mykey'		{'mykey': None}
resolve=False	key=name	{'key': 'name'}
splitter='->'	key=name->'value'	{'key=name': 'value'}

3.3 IntegerArgument

Table 3.3: Examples

Non-standard options	Input	Output
	'1'	1
	name	0

3.4 ChoiceArgument

Table 3.4: Examples

Non-standard options	Input	Output
<code>choices=['choice']</code>	<code>name</code>	<code>'choice'</code>
<code>choices=['choice', 'classytags']</code>	<code>name</code>	<code>'classytags'</code>

3.5 MultiValueArgument

Table 3.5: Examples

Non-standard options	Input	Output
		<code>[]</code>
	<code>name 'is' 'awesome'</code>	<code>['classytags', 'is', 'awesome']</code>

3.6 MultiKeywordArgument

Table 3.6: Examples

Non-standard options	Input	Output
		<code>{}</code>
	<code>name=name awesome='yes'</code>	<code>{'name': 'classytags', 'awesome': 'yes'}</code>
<code>splitter=':', resolve=False</code>	<code>hello:world</code>	<code>{'hello': 'world'}</code>

3.7 Flag

Table 3.7: Examples

Non-standard options	Input	Output
<code>true_values=['true', 'yes']</code>	<code>name</code>	<code>False</code>
<code>true_values=['true', 'yes']</code>	<code>'YES'</code>	<code>True</code>
<code>true_values=['true', 'yes'], case_sensitive=True</code>	<code>'YES'</code>	<code>False</code>

4.1 classytags.arguments

This module contains standard argument types.

class `classytags.arguments.Argument` (*name*[, *default=None*][, *required=True*], [*resolve=True*])

A basic single value argument with *name* as it's name.

default is used if *required* is False and this argument is not given.

If *resolve* is False, the argument will not try to resolve it's contents against the context. This is especially useful for 'as varname' arguments. Note that quotation marks around the argument will be removed if there are any.

value_class

The class to be used to wrap the value in. Defaults to `classytags.values.StringValue`.

get_default ()

Returns the default value for this argument

parse (*parser*, *token*, *tagname*, *kwargs*)

Parses a single *token* into *kwargs*. Should return True if it consumed this token or False if it didn't.

parse_token (*parser*, *token*)

Parses a single *token* using *parser* into an object which is can be resolved against a context. Usually this is a template variable, a filter expression or a `classytags.utils.TemplateConstant`.

class `classytags.arguments.Argument` (*name*[, *default=None*][, *required=True*], [*resolve=True*])

Same as `classytags.arguments.Argument` but with `classytags.values.StrictStringValue` as *value_class*.

class `classytags.arguments.KeywordArgument` (*name*[, *default=None*][, *required=True*] [, *resolve=True*][, *defaultkey=None*][, *splitter='='*])

An argument that allows key=value notation.

defaultkey is used as key if no key is given or the default value should be used.

splitter is used to split the key value pair.

wrapper_class

Class to use to wrap the key value pair in. Defaults to `classytags.values.DictValue`.

class `classytags.arguments.IntegerArgument`

Same as `classytags.arguments.Argument` but with `classytags.values.IntegerValue` as *value_class*.

class `classytags.arguments.ChoiceArgument` (*name*, *choices* [, *default=None*] [, *required=True*] [, *resolve=True*])

An argument which validates it's input against predefined choices.

class `classytags.arguments.MultiValueArgument` (*name* [, *default=NULL*] [, *required=True*] [, *max_values=None*] [, *resolve=True*])

An argument which accepts a variable amount of values. The maximum amount of accepted values can be controlled with the *max_values* argument which defaults to `None`, meaning there is no maximum amount of values.

default is an empty list if *required* is `False`.

resolve has the same effects as in `classytags.arguments.Argument` however applies to all values of this argument.

The default value for *value_class* is \$ `classytags.values.ListValue`.

sequence_class

Class to be used to build the sequence. Defaults to `classytags.utils.ResolvableList`.

class `classytags.arguments.MultiKeywordArgument` (*name* [, *default=None*] [, *required=True*] [, *resolve=True*] [, *max_values=None*] [, *splitter='='*])

Similar to `classytags.arguments.KeywordArgument` but allows multiple key value pairs to be given. The will be merged into one dictionary.

Arguments are the same as for `classytags.arguments.KeywordArgument` and `classytags.arguments.MultiValueArgument`, except that *default_key* is not accepted and *default* should be a dictionary or `None`.

class `classytags.arguments.Flag` (*name* [, *default=NULL*] [, *true_values=None*] [, *false_values=None*] [, *case_sensitive=False*])

A boolean flag. Either *true_values* or *false_values* must be provided.

If *default* is not given, this argument is required.

true_values and *false_values* must be either a list or a tuple of strings. If both *true_values* and *false_values* are given, any value not in those sequences will raise a `classytag.exceptions.InvalidFlag` exception.

case_sensitive defaults to `False` and controls whether the values are matched case sensitive or not.

4.2 classytags.blocks

This module contains classes for *Advanced Block Definition*.

class `classytags.blocks.BlockDefintion` (*alias*, **names*)

A block definition with the given alias and a sequence of names. The members of the names sequence must either be strings, `classytags.blocks.VariableBlockName` instances or other objects implementing at least a `collect()` method compatible with the one of `classytags.blocks.VariableBlockName`.

alias

The alias for this definition to be used in the tag's kwargs.

names

Sequence of strings or block name definitions.

validate (*options*)

Validates this definition against an instance of `classytags.core.Options` by calling the `validate()` on all it's *names* if such a method is available.

collect (*parser*)

Returns a sequence of strings to be used in the `parse_until` statement. This is a sequence of strings that this block accepts to be handled. The `parser` argument is an instance of `classytags.parser.Parser`.

class `classytags.blocks.VariableBlockName` (*template, argname*)

A block name definition to be used in `classytags.blocks.BlockDefinition` to implement block names that depend on the (unresolved) value of an argument. The `template` argument to this class should be a string with the `value` string substitution placeholder. For example: `'end_my_block %(value)s'`. The `argname` argument is the name of the argument from which the value should be extracted.

validate (*options*)

Validates that the given `argname` is actually available on the tag.

collect (*parser*)

Returns the template substituted with the value extracted from the tag.

4.3 classytags.core

This module contains the core objects to create tags.

class `classytags.core.Options` (**options, **kwargs*)

Holds the options of a tag. `options` should be a sequence of `classytags.arguments.Argument` subclasses or strings (for breakpoints). You can give them keyword argument `blocks` to define a list of blocks to parse until. You can specify a custom argument parser by providing the keyword argument `parser_class`.

all_argument_names

A list of all argument names in this tag options. Used by `classytags.blocks.VariableBlockName` to validate its definition.

get_parser_class ()

Returns `classytags.parser.Parser` or a subclass of it.

bootstrap ()

An internal method to bootstrap the arguments. Returns an instance of `classytags.utils.StructuredOptions`.

parse (*parser, token*):

An internal method to parse the template tag. Returns a tuple (`arguments, blocks`).

class `classytags.core.TagMeta`

The metaclass of `classytags.core.Tag` which ensures the tag has a `name` attribute by setting one based on the class name if none is provided.

class `classytags.core.Tag` (*parser, token*)

The `Tag` class is nothing other than a subclass of `django.template.Node` which handles argument parsing in its `__init__()` method rather than an external function. In a normal use case you should only override `name`, `options` and `render_tag()`.

Note: When registering your template tag, register the class object, *not* an instance of it.

name

The name of this tag (for use in templates). This attribute is optional and if not provided, the un-camelcase class name will be used instead. So `MyTag` becomes `my_tag`.

options

An instance of `classytags.core.Options` which holds the options of this tag.

`__init__(parser, token):`

Warning: This is an internal method. It is only documented here for those who would like to extend django-classy-tags.

This is where the arguments to this tag get parsed. It's the equivalent to a *compile function* in Django's standard templating system. This method does nothing else but assing the `kwargs` and `blocks` attributes to the output of `options.parse()` with the given *parser* and *token*.

`render(context)`

Warning: This is an internal method. It is only documented here for those who would like to extend django-classy-tags.

This method resolves the arguments to this tag against the context and then calls `render_tag()` with the context and those arguments and returns the return value of that method.

`render_tag(context[, **kwargs])`

The method used to render this tag for a given context. *kwargs* is a dictionary of the (already resolved) options of this tag as well as the blocks (as nodelists) this tag parses until if any are given. This method should return a string.

4.4 classytags.exceptions

This module contains the custom exceptions used by django-classy-tags.

exception `classytags.exceptions.BaseError`

The base class for all custom excpetions, should never be raised directly.

exception `classytags.exceptions.ArgumentRequiredError(argument, tagname)`

Gets raised if an option of a tag is required but not provided.

exception `classytags.exceptions.InvalidFlag(argname, actual_value, allowed_values, tagname)`

Gets raised if a given value for a flag option is neither in *true_values* nor *false_values*.

exception `classytags.exceptions.BreakpointExpected(tagname, breakpoints, got)`

Gets raised if a breakpoint was expected, but another argument was found.

exception `classytags.exceptions.TooManyArguments(tagname, extra)`

Gets raised if too many arguments are provided for a tag.

4.5 classytags.helpers

This modules contains helper classes to make building template tags even easier.

class `classytags.helpers.AsTag`

A helper tag base class to build 'as varname' tags. Note that the option class still has to contain the 'as varname' information. This tag will use the last argument in the options class to set the value into the context.

This class implements the method `classytags.helpers.AsTag.get_value()` which gets the context and all arguments except for the varname argument as arguments. It should always return the value this tag comes up with, the class then takes care of either putting the value into the context or returns it if the varname argument is not provided.

Note: You should not override the `render_tag()` method of this class.

get_value_for_context(context, **kwargs):

New in version 0.5.

Should return the value of this tag if used in the ‘as varname’ form. By default this method just calls `get_value` and returns that.

You may want to use this method if you want to suppress exceptions in the ‘as varname’ case.

get_value(context, **kwargs)

Should return the value of this tag. The context setting is done in the `classytags.core.Tag.render_tag()` method of this class.

class `classytags.helpers.InclusionTag`

A helper class for writing inclusion tags (template tags which render a template).

Note: You should not override the `render_tag()` method of this class.

template

The template to use if `get_template()` is not overridden.

push_context

New in version 0.5.2.

By default, this is `False`. If it’s set to `True` the context will be pushed before rendering the included template, preventing context pollution.

get_template(context, **kwargs)

This method should return a template (path) for this context and arguments. By default returns the value of `template`.

get_context(context, **kwargs)

Should return the context (as a dictionary or an instance of `django.template.Context` or a subclass of it) to use to render the template. By default returns an empty dictionary.

4.6 classytags.parser

The default argument parser lies here.

class `classytags.parser.Parser(options)`

The default argument parser class. It get’s initialized with an instance of `classytags.utils.StructuredOptions`.

options

The `classytags.utils.StructuredOptions` instance given when the parser was instantiated.

parser

The (template) parser used to parse this tag.

bits

The split tokens.

tagname

Name of this tag.

kwargs

The data extracted from the bits.

blocks

A dictionary holding the block nodelists.

arguments

The arguments in the current breakpoint scope.

current_argument

The current argument if any.

todo

Remaining bits. Used for more helpful exception messages.

parse (*parser, token*)

Parses a token stream. This is called when your template tag is parsed.

handle_bit (*bit*)

Handle the current bit (token).

handle_next_breakpoint (*bit*)

The current bit is the next breakpoint. Make sure the current scope can be finished successfully and shift to the next one.

handle_breakpoints (*bit*)

The current bit is a future breakpoint, try to close all breakpoint scopes before that breakpoint and shift to it.

handle_argument (*bit*)

The current bit is an argument. Handle it and contribute to *kwargs*.

parse_blocks ()

Parses the blocks this tag wants to parse until if any are provided.

finish ()

After all bits have been parsed, finish all remaining breakpoint scopes.

check_required ()

A helper method to check if there's any required arguments left in the current breakpoint scope. Raises a *classytags.exceptions.ArgumentRequiredError* if one is found and contributes all optional arguments to *kwargs*.

4.7 classytags.utils

Utility classes and methods for django-classy-tags.

class `classytags.utils.NULL`

A pseudo type.

class `classytags.utils.TemplateConstant` (*value*)

A constant pseudo template variable which always returns it's initial value when resolved.

literal

Used by the `classytags.blocks.VariableBlockName` to generate it's final name.

class `classytags.utils.StructuredOptions` (*options, breakpoints*)

A helper class to organize options.

options

The arguments in this options.

breakpoints

A *copy* of the breakpoints in this options

blocks

A *copy* of the list of tuples (blockname, alias) of blocks of this tag.

current_breakpoint

The current breakpoint.

next_breakpoint

The next breakpoint (if there is any).

shift_breakpoint ()

Shift to the next breakpoint and update *current_breakpoint* and *next_breakpoint*.

get_arguments ()

Returns a copy of the arguments in the current breakpoint scope.

class `classytags.utils.ResolvableList (item)`

A subclass of list which resolves all it's items against a context when it's resolve method gets called.

`classytags.utils.get_default_name (name)`

Turns 'CamelCase' into 'camel_case'.

4.8 classytags.values

class `classytags.values.StringValue (var)`

errors

A dictionary holding error messages which can be caused by this value class. Defaults to an empty dictionary.

value_on_error

The value to use when the validation of a input value fails in non-debug mode. Defaults to an empty string.

var

The variable wrapped by this value instance.

resolve (context)

Resolve *var* against *context* and validate it by calling the *clean ()* method with the resolved value.

clean (value)

Validates and/or cleans a resolved value. This method should always return something. If validation fails, the *error ()* helper method should be used to properly handle debug modes.

error (value, category)

Handles an error in *category* caused by *value*. In debug mode this will cause a `django.template.TemplateSyntaxError` to be raised, otherwise a `TemplateSyntaxWarning` is called and *value_on_error* is returned. The message to be used for both the exception and the warning will be constructed by the message in *errors* if *category* is in it. The value can be used as a named string formatting parameter.

class `classytags.values.StrictStringValue (var)`

Same as *StringValue* but enforces that the value passed to it is a string (instance of `basestring`).

class `classytags.values.IntegerValue (var)`

Subclass of *StringValue*.

clean (*value*)

Tries to convert the value to an integer.

class `classytags.values.ListValue` (*value*)

Subclass of *StringValue* and `list`.

Appends the initial value to itself in initialization.

resolve (*context*)

Resolves all items in itself against *context* and calls `clean()` with the list of resolved values.

class `classytags.values.DictValue` (*dict*)

Subclass of *StringValue* and `dict`.

resolve (*context*)

Resolves all *values* against *context* and calls `clean()` with the resolved dictionary.

Extending django-classy-tags

You can extend django-classy-tags by writing your own subclasses of `classytags.arguments.Argument` which behave to your needs. If that does not cover your needs, you may also subclass `classytags.core.Options` and set a custom argument parser, which should subclass `classytags.parser.Parser`.

5.1 Creating a custom argument class

The most important method in this class for customization is `classytags.arguments.Argument.parse()`, so let's have a closer look at it. It takes exactly four arguments, which are as follows:

- *parser*: An instance of `django.template.Parser`.
- *token*: The current token as a string.
- *tagname*: The name of the tag being handled.
- *kwargs*: The dictionary of already parsed arguments.

The parse method must return a boolean value:

- If your method returns `True`, it means it has successfully handled the provided token. Your method has to add content to *kwargs* itself. The parser does not do that! When you return `True`, the next token will also try to get parsed by this argument's parse method.
- If your method returns `False`, it means it has not handled this token and the next argument class in the stack should be used to handle this token. Usually you would return `False` when your argument's name is already in *kwargs*. Obviously this only applies to single-value arguments.

So let's look at the standard `classytags.arguments.Argument.parse()`:

```
def parse(self, parser, token, tagname, kwargs):
    """
    Parse a token.
    """
    if self.name in kwargs:
        return False
    else:
        kwargs[self.name] = self.parse_token(parser, token)
        return True
```

First it checks if the name is already in *kwargs*. If so, return `False` and let the next argument handle this token. Otherwise do some checking if we should resolve this token or not and add it to *kwargs*. Finally return `True`.

You might notice the `classytags.arguments.Argument.parse_token()` method used there. This method is responsible for turning an token into a template variable, a filter expression or any other object which allows to be resolved against a context. The one in `classytags.arguments.Argument` looks like this:

```
def parse_token(self, parser, token):
    if self.resolve:
        return parser.compile_filter(token)
    else:
        return TemplateConstant(token)
```

5.1.1 Cleaning arguments

If all you want to do is clean arguments or enforce a certain type, you can just change the `classytags.arguments.Argument.value_class` of your subclass of `classytags.arguments.Argument` to a subclass of `classytags.values.StringValue` which implements a `clean` method in which you can check the type and/or cast a type on the value. For further information on value classes, see `classytags.values`.

5.2 Custom argument parser

The argument parser was written with extensibility in mind. All important steps are split into individual methods which can be overwritten. For information about those methods, please refer to the reference about `classytags.parser.Parser`.

To use a custom parser, provide it as the `parser_class` keyword argument to `classytags.core.Options`.

Note: Each time your tag gets parsed, a new instance of the parser class gets created. This makes it safe to use `self`.

5.3 Example

Let's make an argument which, when resolved, returns a template.

First we need a helper class which, after resolving loads the template specified by the value:

```
from django.template.loader import get_template

class TemplateResolver(object):
    def __init__(self, real):
        self.real = real

    def resolve(self, context):
        value = self.real.resolve(context)
        return get_template(value)
```

Now for the real argument:

```
from classytags.arguments import Argument

class TemplateArgument(Argument):
    def parse_token(self, parser, token):
        real = super(TemplateArgument, self).parse_token(parser, token)
        return TemplateResolver(real)
```

Contribute to django-classy-tags

As with (hopefully) any open source project, contribution is very welcome.

6.1 Review the documentation

A great way to contribute to django-classy-tags, which requires absolutely no coding skills and can be done by anyone who knows English, is to read this documentation and inform me about any mistakes in the documentation. Also any suggestion on how to write the documentation clearer and easier to understand would be greatly appreciated.

If you use django-classy-tags yourself as a developer and found certain things in the documentation to be unclear to you or you would like to have more in depth documentation on some features, please also tell me so I can make it easier for people to use this project.

6.2 Spread the word

Tell people about this project! The more people use it the higher the chance the quality of this project will increase.

6.3 Improve the test suite

There is a handy little test framework located in the testdata folder. It allows automated testing of django-classy-tags implementation against builtin Django tags. Have a look at the existing tags and feel free to add more!

Also adding good old unit tests is a great way to help too.

6.3.1 Running the test suite

From the root folder of your django-classy-tags checkout, run `./runtests.sh`.

6.4 Write code

Want a feature implemented or a bug eliminated? The quickest way to get there is to write the code yourself and get it back to me.

Please note that your code must have *full* test coverage and *full* documentation coverage, otherwise it won't be pulled into the main repository.

Changelog

7.1 0.8.0

- Removed Python 2.6 support. Supported versions are now 2.7, 3.3, 3.4 and 3.5.
- Added support for Django 1.10.

7.2 0.7.2

- Fixed regression introduced in 0.7.1 breaking Django 1.9 and higher.

7.3 0.7.1

- Prepare support for Django 1.10. Please note that Django 1.10 is **not** supported by this release, as Django 1.10 is not released yet.

7.4 0.7.0

- Added support for Django 1.9
- Added support for Python 3.5
- Added a nice `__repr__` to `classytags.core.Options`
- Added ability to combine `classytags.core.Options` instances using the add operator.

7.5 0.6.2

- Fixed Django 1.8 support

7.6 0.6.1

- Packaging fixes

7.7 0.6.0

- Added support for Django 1.8
- Dropped support for Django 1.2

7.8 0.5.2

- Added `classytags.helpers.InclusionTag.push_context`.

7.9 0.5.1

- Fixed `classytags.helpers.InclusionTag.template` being required.

7.10 0.5.0

- Added Python 3 support
- Added *Advanced Block Definition*.
- Added Arguments in depth documentation.

7.11 0.3.3

- Fixed issues with `classytags.helpers.InclusionTag``.

7.12 0.3.2

- Fixed issue in `classytags.arguments.MultiKeywordArgument` and `classytags.arguments.KeywordArgument` and their behavior when given a default value.

7.13 0.3.1

- Fixed `classytags.arguments.MultiKeywordArgument` and `classytags.arguments.KeywordArgument` not returning sane defaults.
- Added `child_nodelist` attribute on tag instances as well as setting the child nodelists as attributes onto the instance during initialization for compatibility with applications that require these attributes to be set.

7.14 0.3.0

- Added `classytags.arguments.KeywordArgument`
- Added `classytags.arguments.MultiKeywordArgument`

- Added `classytags.arguments.ChoiceArgument`
- Added ability to override the parser class in the initialization of the `classytags.core.Options` class, to make the usage of custom parsers easier.
- Added `classytags.values.DictValue`
- Added `classytags.values.ChoiceValue`

7.15 0.2.2

- Fixed issue in `classytags.helpers.AsTag` when trying to extract the variable to store the value in, but no argument is given.
- Fixed `classytags.helpers.InclusionTag` not validating the `template` attribute on initialization.

7.16 0.2.1

- Fixed version in documentation not matching release version.

7.17 0.2.0

- Added ability to have typed arguments.
- Added `classytags.arguments.IntegerArgument`
- Added more graceful failing in non-debug mode by using warnings instead of exceptions.

7.18 0.1.3

- Added `classytags.helpers.InclusionTag`

Indices and tables

- `genindex`
- `modindex`
- `search`

C

- `classytags.arguments`, 11
- `classytags.blocks`, 12
- `classytags.core`, 13
- `classytags.exceptions`, 14
- `classytags.helpers`, 14
- `classytags.parser`, 15
- `classytags.utils`, 16
- `classytags.values`, 17

A

alias (classytags.blocks.BlockDefintion attribute), 12
 all_argument_names (classytags.core.Options attribute), 13
 Argument (class in classytags.arguments), 11
 ArgumentRequiredError, 14
 arguments (classytags.parser.Parser attribute), 16
 AsTag (class in classytags.helpers), 14

B

BaseError, 14
 bits (classytags.parser.Parser attribute), 15
 BlockDefintion (class in classytags.blocks), 12
 blocks (classytags.parser.Parser attribute), 16
 blocks (classytags.utils.StructuredOptions attribute), 17
 bootstrap() (classytags.core.Options method), 13
 BreakpointExpected, 14
 breakpoints (classytags.utils.StructuredOptions attribute), 16

C

check_required() (classytags.parser.Parser method), 16
 ChoiceArgument (class in classytags.arguments), 11
 classytags.arguments (module), 11
 classytags.blocks (module), 12
 classytags.core (module), 13
 classytags.exceptions (module), 14
 classytags.helpers (module), 14
 classytags.parser (module), 15
 classytags.utils (module), 16
 classytags.values (module), 17
 clean() (classytags.values.IntegerValue method), 17
 clean() (classytags.values.StringValue method), 17
 collect() (classytags.blocks.BlockDefintion method), 12
 collect() (classytags.blocks.VariableBlockName method), 13
 current_argument (classytags.parser.Parser attribute), 16
 current_breakpoint (classytags.utils.StructuredOptions attribute), 17

D

DictValue (class in classytags.values), 18

E

error() (classytags.values.StringValue method), 17
 errors (classytags.values.StringValue attribute), 17

F

finish() (classytags.parser.Parser method), 16
 Flag (class in classytags.arguments), 12

G

get_arguments() (classytags.utils.StructuredOptions method), 17
 get_context() (classytags.helpers.InclusionTag method), 15
 get_default() (classytags.arguments.Argument method), 11
 get_default_name() (in module classytags.utils), 17
 get_parser_class() (classytags.core.Options method), 13
 get_template() (classytags.helpers.InclusionTag method), 15
 get_value() (classytags.helpers.AsTag method), 15

H

handle_argument() (classytags.parser.Parser method), 16
 handle_bit() (classytags.parser.Parser method), 16
 handle_breakpoints() (classytags.parser.Parser method), 16
 handle_next_breakpoint() (classytags.parser.Parser method), 16

I

InclusionTag (class in classytags.helpers), 15
 IntegerArgument (class in classytags.arguments), 11
 IntegerValue (class in classytags.values), 17
 InvalidFlag, 14

K

KeywordArgument (class in classytags.arguments), 11

kwargs (classytags.parser.Parser attribute), 15

L

ListValue (class in classytags.values), 18

literal (classytags.utils.TemplateConstant attribute), 16

M

MultiKeywordArgument (class in classytags.arguments), 12

MultiValueArgument (class in classytags.arguments), 12

N

name (classytags.core.Tag attribute), 13

names (classytags.blocks.BlockDefintion attribute), 12

next_breakpoint (classytags.utils.StructuredOptions attribute), 17

NULL (class in classytags.utils), 16

O

Options (class in classytags.core), 13

options (classytags.core.Tag attribute), 13

options (classytags.parser.Parser attribute), 15

options (classytags.utils.StructuredOptions attribute), 16

P

parse() (classytags.arguments.Argument method), 11

parse() (classytags.parser.Parser method), 16

parse_blocks() (classytags.parser.Parser method), 16

parse_token() (classytags.arguments.Argument method), 11

Parser (class in classytags.parser), 15

parser (classytags.parser.Parser attribute), 15

push_context (classytags.helpers.InclusionTag attribute), 15

R

render() (classytags.core.Tag method), 14

render_tag() (classytags.core.Tag method), 14

ResolvableList (class in classytags.utils), 17

resolve() (classytags.values.DictValue method), 18

resolve() (classytags.values.ListValue method), 18

resolve() (classytags.values.StringValue method), 17

S

sequence_class (classytags.arguments.MultiValueArgument attribute), 12

shift_breakpoint() (classytags.utils.StructuredOptions method), 17

StrictStringValue (class in classytags.values), 17

StringValue (class in classytags.values), 17

StructuredOptions (class in classytags.utils), 16

T

Tag (class in classytags.core), 13

TagMeta (class in classytags.core), 13

tagname (classytags.parser.Parser attribute), 15

template (classytags.helpers.InclusionTag attribute), 15

TemplateConstant (class in classytags.utils), 16

todo (classytags.parser.Parser attribute), 16

TooManyArguments, 14

V

validate() (classytags.blocks.BlockDefintion method), 12

validate() (classytags.blocks.VariableBlockName method), 13

value_class (classytags.arguments.Argument attribute), 11

value_on_error (classytags.values.StringValue attribute), 17

var (classytags.values.StringValue attribute), 17

VariableBlockName (class in classytags.blocks), 13

W

wrapper_class (classytags.arguments.KeywordArgument attribute), 11