TIE Documentation

Release 0.0.1

Raphi Gaziano

August 18, 2013

CONTENTS

Note: The tie library is still a work in progress, and as such the api described here is still subject to change. I'll do my best to keep changes to a minimum after its first release, and more importantly to reflect those changes in the documentation if and as they happen.

Should you happen to notice any missing, outdated or plain wrong information, feel free to contact me through github or via email.

CHAPTER

ONE

DEPENDENCIES

- None! (Besides Python, obviously. TIE has been tested with python 2.7 & 3.2, under both Linux & Windows.)

CHAPTER

TWO

USER GUIDE

2.1 Introduction

2.1.1 What is TIE ?

The TIE library provides a set of classes and utilities to facilitate the definition of very simple, personal template languages.

The library provides a basic substitution engine based on regular expressions, which doesn't recognize any particular syntax by itself; it's up to the user to provide it with his own tag patterns as well as their optional, custom behaviour.

TIE also provides simple tools to ease the definition of those custom tags, and aims to allow for easy customisation or extension (either by providing callbacks or by subclassing the provided types).

2.1.2 Do I need it ?

Maybe. Maybe not.

If you need a full-fledged template engine, with lots of features and good performances, then TIE is probably not for you. You'll be far better off using an already established template language - Quite a lot of them are already part of the python ecosystem and have more than proven themselves. Trying to emulate one of those with TIE *might* be possible, but will surely prove very cumbersome and inneficient.

Note: If you're looking for a lightweight, but more featured template engine, I'd like to recommend pyratemp. I like the author's "simple is better" philosophy, and his thoughts on template engines have been a nice source of inspiration for TIE.

On the other hand, TIE might still be overkill if your requirements are very simple. Python's batteries include a very nice and quite powerful string formatting syntax, and also provides a Template class for slightly more complex operations. Those built-in features might be more than sufficient for what you have in mind. (See http://docs.python.org/2/library/string.html for more info on python's string operations.)

TIE aims to step in when python's built-in tools might be enough for the job, but become too unwieldy to handle the task in a straight-forward way.

2.1.3 Installation

You can install TIE by simply using pip (this is the recomanded way):

```
pip install tie
```

If you must, you can also use easy_install:

```
easy_install tie
```

Alternativeley, you could also clone the project's repository and run the setup script:

```
git clone https://github.com/raphigaziano/TIE
cd TIE/
python setup.py install
```

2.1.4 Getting Started

For most basic uses, rendering a template with TIE involves 3 simple steps:

- Register your tag patterns
- Wrap your template(s) text in (a) Template object(s)
- Pass your templates the data they need to render them

A naive exemple could look like this:

```
>>> import tie
>>> # Register a tag pattern
>>> tie.tag.register("name")
>>> # Instanciate a Template object
>>> my_template = tie.Template("Hello, name!")
>>> # Render it!
>>> res = my_template(name="raphi")
>>> print(res)
Hello, raphi!
>>> res = my_template(name="Darth Vader, lord of the sith")
>>> print(res)
Hello, Darth Vader, lord of the sith!
```

Note: For testing purposes, I'm using the python 3 print function here, but this should work just as well with the python 2.x syntax. Adjust the code accordingly, or add a `from __future__ import print_function` statement before running this code.

While this exemple is way too simple to be useful, the basic process it illustrates should be able to handle a lot of common situations.

Head on to the *TIE tutorial* to start using TIE the right way !

2.2 Tutorial

This tutorial aims to explain how to use TIE to create a simple, yet useful template language. This will cover the basics of tag definition and registration, as well as the creation of a few custom tags with specialized behaviour, and how to easily manage them.

2.2.1 Part I - Simple substitution tags

The naive way's shortcomings

The *exemple* shown in the overview might demonstrate how simple using TIE can be, but defining tags in this way is a pretty bad idea.

Indeed, consider what would happen if you used another template string:

```
>>> my_template = tie.Template("Hello, my name is name!")
>>> my_template(name="raphi")
'Hello, my raphi is raphi!'
```

Your *name* tag matched *all* occurences of the word "name" in your template, which is probably *not* what you wanted!

And, let's face it, this was to be expected. *name* is an awfull tag pattern - In order for TIE to detect your placeholders more intelligently, they need to contain some specific tokens that will help differentiate them from regular words. In order to be more flexible, TIE requires you to include those tokens in your patterns yourself - But this also means that you should think carefully about them to avoid that kind of confusion.

Let's decide that our tags should be surrounded by '%' characters to be detected.

```
>>> tie.tag.register("%name%")                              # Register our new tag pattern
>>> my_template = tie.Template("Hello, my name is %name%!") # Use it in our template
>>> my_template(name="raphi")                               # Does it work ???
'Hello, my name is !'
```

Well shoot. Our tag apparently matched but it got replaced with a blank string instead of our custom data!

This is because when you call your `Template` object to render it, it receives your arguments as a dictionnary (this is the normal python behaviour for keyword arguments). TIE's default behaviour is then to replace each detected tag with a matching value from this dictionnary. If it can't find it, it raises a warning and returns a blank string.

> **Warning:** The behaviour described above might change in future versions.

This means that in our case, our `name` argument and our *%name%* tag don't match, which explains why the above code didn't work.

But... *%name%* is not a valid python identifier, is it ?

```
>>> my_template(%name%="raphi")
Traceback (most recent call last):
    ...
    my_template(%name%="raphi")
                ^
SyntaxError: invalid syntax
```

Nope, it isn't.

So, we need to define special tokens to identify our template tags, but we can't use non-alphanumerical characters besides the underscore ? Well, this sucks. And I thought this library claimed to be "flexible" ?

Don't worry. We just need to improve our tag just a little more.

Note: Experienced Python users might be thinking of building the arguments dictionary themselves and sending it with the splat operator, like this:

```
my_template(**{"%name%": "raphi"}) # **Don't do this!**
```

This will work, but is ugly as hell. Experienced or not, Python users shouldn't have to write such ugly code.

Regular expressions to the rescue !

If you've never heard of regular expressions, then things might start to get a bit hairy. I'll try to explain how the first few ones we'll use in this tutorial work, but you'll need to learn more about them to use TIE efficiently. I suggest reading this howto from the python's documentation to get started. Also, while you shouldn't need to use it directly, reading the standard library's `re` module's reference might help you as well.

One of the neat things about regular expressions is that they allow you to capture specific parts, or "groups", of the matching string. If you define one such group in your pattern, TIE will try to match your context variables against it, instead of using the whole tag.

The simplest way to define a group is simply to surround in with parenthesis. (You can also use another syntax to assign names to your groups. While this can come in handy, there's no real need to do so in our situation, so we'll settle for an anonymous group for the sake of readability.)

Let's try this:

```
>>> tie.tag.register("%(name)%")
>>> my_template = tie.Template("Hello, my name is %name%!")
>>> my_template(name="raphi")
'Hello, my name is raphi!'
```

Hurrah! This lib might not be so useless after all!

While you can get more fancy, this is really all you have to understand to start using TIE. As long as you include appropriate tokens [1] in your patterns, and remember to define a group that can match the variables names you'll be using in your code, you're ready to start defining a simple template language using arbitrary tags.

But, as far as regular expressions go, *%(name)%* is about as simple as it gets. If you've ever used regexes, then you know that they can be far more powerful (and far less readable ;)) than this.

Let's see if we can tweak our tag further...

---

[1] What's an appropriate token? Well, it all depends on the context in which you plan to use your template tags. If generating html documents, surrounding your tags with angle brackets (<>) might not be the best idea...

Just take some time to think about it and use some common sense. Typical patterns could look like the ones we're defining in this tutorial (*%my_tag%*), or like the ones used by the django and Jinja2 template engines (*{{ my_tag }}*).

Note: It's possible to use the `re` module's flags in your tags' regexes. To do so, you'll have to instanciate your `Tag` objects explicitely and pass them to the `register` function, instead of simply passing the regex string, like so:

```
import re
import tie

tie.tag.register(
   tie.Tag("^my_awesome_regex$", flags=re.FOO | re.BAR)
)
```

More generic tags

So, now that we know how to define better template tags, let's register another one:

```
>>> tie.tag.register(    # Notice that you can pass an arbitrary number
...     "%(name)%",      # of patterns to register them all at once
...     "%(age)%"
... )
>>> my_template = tie.Template("Hello, my name is %name% and I'm %age% years old!")
>>> my_template(name="raphi", age=26)
"Hello, my name is raphi and I'm 26 years old!"
```

Yup, still works. And as a bonus, you might have noticed that we passed the `age` argument as an integer value, and not as a string. TIE is just smart enough to call the `__str__` method of the objects it's asked to process in order to display them. Keep that in mind if you plan on sending custom objects to your templates.

We still have to register a new pattern for every tag we want to support. This is perfectly fine if you want to allow only a limited set of template tags - sometimes you need tight control over what can or can't go in your templates, and explicitely defining each tag in this way will help you manage what's going on.

But still, wouldn't it be nice if we could let TIE match any arbitrary argument we might send it ? Get rid of the *%name%* and *%age%* tags and instead, have some kind of generic *%<var>%* tag that would match whatever context argument happened to be referenced between those two percent signs ?

Remember. While the ones we've used so far didn't look like much, our tag patterns are still regular expressions. Knowing this, and assuming you've read up a bit on those, the solution becomes trivial:

```
>>> tie.tag.register("%(\w+)%")
>>> my_template = tie.Template("Hello, my name is %name% and I'm %age% years old!")
>>> my_template(name="raphi", age=26)
"Hello, my name is raphi and I'm 26 years old!"
```

The `\w` special sequence will match any alphanumeric character (that is, any upper-or-lowercase letter, number, or underscore). The + indicates that the preceding pattern should appear at least once, and can be repeated several times. So in effect, this regular expression will match any single word not containing fancy characters and surrounded by percent signs. And since underscores are allowed, any valid variable name should match!

Once again, if you need to get up to date about regular expressions, I recommend starting with the guide from the official Python documentation.

2.2.2 Part II - Managing your templates

While it's allright to define your template strings directly in your code for very simple use cases such as the ones we've covered so far, real world applications should enforce a better separation of concerns and store their templates in external files. Think MVC: Your presentation layer (which most templating systems will be be a part of) should always be kept cleanly separated from the rest of your code.

Also, we're dealing with very short and simple templates here. Real world applications will probably need much larger templates, and jugling with all those multiline strings in the middle of your code will surely prove annoying and difficult, which is another reason why you should just store them in external text files.

While you can certainly manage these external files yourself, TIE provides some handy shortcuts to help you keep things nice and tidy.

Let's have a look at those and start using some best-practices before diving in any further.

Using external template files

Fire up your favourite editor and start designing a simple template. I'll use a pretty minimal one, and save it as *test_template.txt*:

```
Hello, world!
My name is %name%,
and I'm %age% years old!
Yay!
```

Now, back to your python code.

You could use the python builtin `open()` function to read your new template file and pass its contents to the `Template`'s constructor, but the class provides a handy factory method to handle this for you:

```
>>> my_template = tie.Template.from_file("test_template.txt")
>>> res = my_template(name="Eddie", age=21)
>>> print(res)
Hello, world!
My name is Eddie,
and I'm 21 years old!
Yay!
```

Just provide a valid path to your template and it will take care of instanciating itself from its contents, allowing you to avoid some clutter and focus on more important stuff.

Register your templates to a manager

Since template managers are a nice, but rather optional feature, they haven't been implemented yet.

I do plan to add them soon, so check back in a while for them!

2.2.3 Part III - Custom Tag behaviour

Coming soon!

2.3 HOWTOs

Contents:

CHAPTER

THREE

API REFERENCE

3.1 TIE Exceptions

Here are the specific exceptions TIE Might throw at you.

> **Warning:** Only a few of those are currently used.
> Error handling is still quite minimal, so this hierarchy should **not** be considered stable.

Exceptions and warnings for the tie library.

exception `tie.exceptions.`**`TIEError`**
TIE lib's main Exception class.

exception `tie.exceptions.`**`TagError`**
Tag related Errors

exception `tie.exceptions.`**`InvalidTagError`**
Invalid value to register a Tag object

exception `tie.exceptions.`**`TemplateError`**
Template related Errors

exception `tie.exceptions.`**`ContextWarning`**
Context variables Warning

3.2 Tag module

The tie.tag module exposes the functions needed to manage your tag patterns, as well as the base classes needed to customize their behaviour or the way they will be managed in your application.

3.2.1 Module's Top level Classes & Utilities

`tie.tag.`**`register`**(**tag_list*)
Register a sequence of tags.

Call this function and pass it an arbitrary number of tags to register them with TIE.

Since your Tag list is needed by various internal parts of the TIE library, you **must** use this function in order for them to have any effect. TIE stores them in a default `TagManager` instance, which you can replace if you need it to behave differently (See below).

Each *tag* parameter should be either a string of the tag's regular expression (or an already compiled regex object), or an instance of `Tag` (or of any class inheriting from it). Instanciating your Tag objects manually allows you to adjust their behaviour, either by tweaking their default parameters or by using a custom subclass:

```
tie.tag.register(
    "sometagpattern",
    tie.Tag("anothertag", processor=FOO),
    MyCustomTagSubclass("taggytagtag"),
    ...
)
```

Internally, this function simply hands each of its parameters to the current `TagManager` and lets it handle the registration process. Actual error checking is done in the `Tag` 's constructor.

Note: The actual arguments expected by *register* might vary if you decide to use a different `TagManager`. For instance, a `PriorityTagManager` will expect tuples of (tag, priority). *register* will simply pass each item it receives to the current manager; see their documentation, as well as the one for any custom Tag class you might use, to know for certain how you should register your tag patterns.

---

class `tie.tag.`**`Tag`**(*pattern*, *flags=0*, *processor=tie.processors.sub*)

The Tag class is TIE's central component.

It's a somewhat boosted regular expression object, which knows how to match itself against a given template, and modify each occurence of its pattern within the template's text using its internal *processor* function.

TIE takes care of managing and handling its registered Tag object, but instanciating them manually allows one to change their default behaviour by providing a custom callback as the *processor* argument. (Default processors callbacks are defined in the `<tie.processors>` module.)

If further customisation is needed, feel free to override its public methods in a subclass.

Parameters:

- ***pattern*: Regular expression used for tag matching.** This can be either a normal string or an already compiled regular expression.
- ***flags*: `re` module's flags for pattern compilation.** Pass them just as you would when using the `re.compile()` function.
- ***processor*: Tag processing callback.** Processor function should accept a `match` object as their first parameter, and a dictionnary of keyword arguments containing the context variables available for processing.

  For more information about tag processors, see this HOWTO on tags customization (once its there, that is...)

Todo

Link to custom tags guide

`Tag.`**`match`**(*template*)

Find all matches in `template` and return them as an generator object.

`Tag.`**`process`**(*template*, ***context*)

Scan the template string for all occurences of the Tag and process those using the instance's own processor function. Context args will be passed to the processor function. Return a dictionnary mapping the matched tags from the template with their corresponding values.

Note: For convenience, the Tag class is imported into TIE's global namespace, so you can just `import tie.Tag`.

3.2.2 Managers

TIE uses an internal manager object to keep track of every registered tag. It will use a basic `TagManager` instance by default, which should be able to handle the simplest use cases, so that you don't have to worry about those if you don't need to.

It also provides a few specialized managers with commonly needed special behaviour. If you need tighter control on how your tags should be stored and handled, you can also define and use your own `TagManager` subclass.

The `tie.tag` module exposes the two following functions to set or access the current manager:

`tie.tag.`**`set_manager`**`(`*manager*`)`
Set the global TagManager to `manager`. Only useful to setup a custom manager.

`tie.tag.`**`get_manager`**`()`
Return the global TagManager

Note: Since the `register` function appends the tags it receives to the current manager, it should only be called after setting any custom one.

Warning: Unlike Template Managers, which are completely optionnal, most of TIE's internal objects *require* a global TagManager instance to be set in order to be able to perform their tasks. While it is possible to bypass calling the `get_manager` function when using a non-default manager if you also tweak these objects, doing so will probably bypass most of TIE's convenience as well.

TIE comes with the following managers:

class `tie.tag.`**`TagManager`**
A basic `Tag` container to keep track of registered tags. TIE will use this manager by default. You can iterate over it to retrieve individual tags – Those will be yielded in the order of their insertion:

```
>>> tie.tag.register('pattern2',
...                  'pattern1',
...                  'pattern3'
... )
>>> manager = tie.tag.get_manager()
>>> for tag in manager:
...     print(tag)
...
<Tag 'pattern2'>
<Tag 'pattern1'>
<Tag 'pattern3'>
```

Tags are stored in a simple list, in a "private" `_tag_list` attribute. Subclasses will probably need to override this attribute in order to use other data structures.

`TagManager.`**`add`**`(`*tag*`)`
Register a new tag. Override this method to accomodate a different internal data strucutre.

This method is called by the `register` function.

`TagManager.`**`clear`**`()`
Clear the internal tag list. Override to accomodate a different internal data strucutre.

TagManager.**__iter__**()
Yield contained tags. Override to accomodate a different internal data strucutre.

TagManager.**_check_tag**(*tag*, *cls=tie.tag.Tag*)
Static method.

Internal checking method, called before inserting any tag to the manager's tag list. It simply passes its `tag` parameter to the `cls` constructor if `tag` is not already an instance (or subclass) of it – This is what allows you to pass either regular strings or `Tag` instances to the `register` function.

Actual error handling is left to the called constructor.

You might need to override this method if you're using fancier Tag objects. If not, you should probably still remember to call it before inserting your tags when redefining the `add` method.

Moar specialized managers provided by TIE are listed below:

class tie.tag.**PriorityTagManager**
Bases: tie.tag.TagManager

TagManager that keeps a priority value along its tags and yields them in that order.

Tags with the lowest priority value will be yielded first:

```
>>> tie.tag.set_manager(tie.tag.PriorityTagManager())
>>> tie.tag.register(
...     ('sometag', 2),
...     ('othertag', 0),
...     ('taggytag', 1),
... )
>>> manager = tie.tag.get_manager()
>>> for tag in manager:
...     print(tag)
...
<Tag 'othertag'>
<Tag 'taggytag'>
<Tag 'sometag'>
```

add(*tag*)
Register a new tag. `tag` should be a tupple `(tag, priority)`. If not, `priority` will default to 0.

clear()
Clear the internal tag list.

3.3 Template module

The tie.template module stores all the classes needed to represent your template objects.

Note: Only the base Template class is defined in here for now.

TemplateManager basic classes will be defined here as well in a future release.

---

class tie.template.**Template**(*tmpl*, *renderer=renderers.default_renderer*)
Template objects represent your template strings and provide ways to handle them easily.

It uses a callback function (defaulting to `renderers.default_renderer`) to handle the actual rendering, most of its public methods being convenience wrappers around it. Passing it another function on instantiation will allow you to alter this default processing, but the default function should be fine for most cases.

Todo

LINK TO GUIDE ON CUSTOM TEMPLATES

Parameters:

- *tmpl*: Template string, containing your defined tags.
- *renderer*: Rendering callback.

`Template.`**`render`**`(**context)`
Process the template & return the result. context is the keyword dictionary of context variables to be injected into the processed template.

Override this method if you need some custom behiavour that can't be handled by a simple callback.

`Template.`**`__call__`**`(**context)`
Convenience alias for `Template.render()`.

This is what allows you to simply call your template objects directly:

```
>>> t = Template("Hello, %name%!")
>>> res = t(name="Santa")
>>> print(res)
'Hello, Santa!'
```

is equivalent to:

```
>>> t = Template("Hello, %name%!")
>>> res = t.render(name="Santa")
>>> print(res)
'Hello, Santa!'
```

`Template.`**`from_file`**`(tmpl_path, *args, **kwargs)`
Class method.

This alternative constructor allows you to instanciate a template object directly from an external file.

Simply pass it a valid file path instead of a template string, as well as any other argument required by the `Template` constructor, and a `Template` instance initialized with the specified file's contents will be returned.

Note: For convenience, the Template class is imported into TIE's global namespace, so you can just `import tie.Template`.

3.4 Processors Module

This module stores all the callback functions provided by TIE and used to process matching tags in a template.

Note: Only one function, the default `sub` processor, is defined for now.

`tie.processors.`**`sub`**`(match, **context)`
Default tag processor. Returns the appropriate value from **context for a matched tag.

This processor performs a simpe substitution on the processed tag. It will check for a value matching the current tag within the `context` dictionnary and return it, or an empty string if no match was found.

Warning: The "no match found" behaviour is still undefined. For now it simply raises a warning and return an empty value, so that the tag will simply be suppressed.

3.4.1 Custom processors:

Todo

Renderer "protocol" summary

3.5 Renderers Module

This module stores all the callback functions provided by TIE and used to process process a template as a whole.

Note: Only one function, the default `default_renderer` processor, is defined for now.

`tie.renderers.`**`default_renderer`**(*template*, ***context*)
Default template renderer. Process each registered Tag and returns the whole processed string.

3.5.1 Custom renderers:

Todo

Renderer "protocol" summary

3.6 Helpers Module

This module regroups various helper functions which you can use when defining your custom processors or renderers.

3.6.1 Regular Expresssions Helpers:

The following helpers are convenience shortcuts for manipulating regex matches when handling custom tags.

`tie.helpers.`**`get_single_group`**(*match*, *key=1*)
Return one and only one group from the passed match object. If no group was defined in the matched regexp, return the whole match.

Parameters:

- **`match`:** A re.match object to extract the group from.
- **`key`:** Optionnal key argument to get a specific group. This can be either a list index or a string to get a named group. (See the `re.MatchObject.group()` method's documentation if you don't know how to get a specific group from a match object). Defaults to 1, to return the first defined group in the matching regex.

CHAPTER

FOUR

INDICES AND TABLES

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

CHAPTER

FIVE

TODO LIST

Todo

Renderer “protocol” summary

(The *original entry* is located in /var/build/user_builds/tie/checkouts/latest/doc/source/api/processors.rst, line 23.)

Todo

Renderer “protocol” summary

(The *original entry* is located in /var/build/user_builds/tie/checkouts/latest/doc/source/api/renderers.rst, line 16.)

Todo

Link to custom tags guide

(The *original entry* is located in /var/build/user_builds/tie/checkouts/latest/doc/source/api/tag.rst, line 84.)

Todo

LINK TO GUIDE ON CUSTOM TEMPLATES

(The *original entry* is located in /var/build/user_builds/tie/checkouts/latest/doc/source/api/template.rst, line 26.)

PYTHON MODULE INDEX

t