pyhamcrest_toolbox Documentation

Release 0.3.0

Timofey Danshin

Jan 13, 2019

Contents

CHAPTER 1

Documentation

1.1 MulticomponentMatcher

class pyhamcrest_toolbox.multicomponent.**MulticomponentMatcher**(**args*, ***kwargs*)

The class that is the base for writing multicomponent matchers, i.e. the ones that have the form of:

```
assert_that(someting, matches_someting(x).and_something_else(y))
```

In your subclass, all you **have** to do is write your `and_something_else()` methods that register your matcher plugins. **NOTE** that you have to return `self` from such methods, as (a) that is required for chaining, and (b) the final instance returned by the chain must be a matcher.

You can write your main matching logic in the traditional way, but you can also register a matcher plugin from your `__init__` method.

The MatcherPlugin entities are still matchers and can be used outside of MulticomponentMatchers. They can be reused in several MulticomponentMatchers, or they can even be grouped into mixins and plugged in as bunches.

Here's an example of what a subclass of the `MulticomponentMatcher` could look like:

```
class GrailMatcher(MulticomponentMatcher):
    def __init__(self, is_holy):
        self.register(GrailHolynessMatcher(is_holy))

    def with_width(width):
        self.register(GrailWidthMatcher(wrap_matcher(width)))
        return self

    def with_height(height):
        self.register(GrailHeightMatcher(wrap_matcher(height)))
        return self
```

And this is all it takes to write your multicomponent matcher. All the descriptions and mismatch descriptions will be build automatically from the plugins.

1.1.1 register

`MulticomponentMatcher.`**`register`**`(`*plugin*`)`

Call this method to register your plugins to your matcher, either from your additional matcher methods (`with_something` or `and_someting`) or from the `__init__` method. **NOTE** that you **must** return `self` from those additional matcher methods.

Parameters **`plugin`** – Instances of `MatcherPluginMixin`

Returns

1.1.2 MatcherPlugin

`class` `pyhamcrest_toolbox.multicomponent.`**`MatcherPlugin`**`(`**args*, ***kwargs*`)`

This is the class to extend when you create your matcher plugins for a multicomponent matcher. Instead of overriding the usual `BaseMatcher` methods, you need to override the ones below.

The original standard `BaseMatcher` methods are replaced with these ones because they are overridden in the `MatcherPlugin` class to work with the `MulticomponentMatcher`.

1.1.3 component_matches

`MatcherPlugin.`**`component_matches`**`(`*item*`)`

Return `True` if it matches, `False` otherwise.

1.1.4 describe_to

`MatcherPlugin.`**`describe_to`**`(`*description*`)`

Generates a description of the object.

The description may be part of a description of a larger object of which this is just a component, so it should be worded appropriately.

Parameters **`description`** – The description to be built or appended to.

The same as `desribe_to` in `Matcher` (`hamcrest.core.selfdescribing.SelfDescribing.describe_to()`). Add the description of the object you expect to the description provided.

1.1.5 describe_component_mismatch

`MatcherPlugin.`**`describe_component_mismatch`**`(`*item*, *mismatch_description*`)`

Basically, the same as `hamcrest.core.matcher.Matcher.describe_mismatch()`.

Here is an example of what a `GrailHolynessMatcher` might look like:

1.2 Wrappers

In order to get going with the toolbox easily, it is important to be able to use your existing matchers with it. To that end, we have created a wrapper that turns your existing matchers to plugins.

1.2.1 MatcherPluginWrapper

class pyhamcrest_toolbox.wrapper_base.**MatcherPluginWrapper**(**args*, ***kwargs*)

This class allows turning good old matchers into matcher plugins that can be used in MulticomponentMatchers.

1.2.2 matcher_class

MatcherPluginWrapper.**matcher_class = <class 'hamcrest.core.core.isequal.IsEqual'>**

Set this variable to the class of the matcher you want to wrap. If you don't, it IsEqual will be used (which is returned by the *equal_to*) function. Note, that you have to specify a class, not a function.

1.2.3 description_prefix

MatcherPluginWrapper.**description_prefix = ''**

The prefix that will be added to the description of the wrapped matcher

1.2.4 mismatch_description_prefix

MatcherPluginWrapper.**mismatch_description_prefix = ''**

The prefix to be added to the mismatch_description

1.2.5 convert_item

MatcherPluginWrapper.**convert_item**(*item*)

Convert the item that the MulticomponentMatcher deals with into the item that your component plugin is responsible for.

The multicomponent matcher will pass the item that it has received, and it is up to your matcher plugin to get the object that it works with from that object. By default, returns the item itself.

downloads 159/month

CHAPTER 2

PyHamcrest Toolbox

PyHamcrest is a great thing for writing reusable tests, but sometimes writing your own matchers may be a bit of a chore. This project aims at making this task easy, fast and even fun (for want of a better word).

To reiterate the obvious, a test should always do all the checks it has to do, and even if some of them fail, the rest should still be run. That way you will get a better idea of what's gone wrong with your code, and you will waste less time fixing the first of the checks only to find the the second one is still failing, and that means that you should have fixed the first one in a different way.

So, instead of this:

```python
def test_the_holy_grail():
    the_holy_grail = seek_the_holy_grail()
    assert_that(the_holy_grail.is_holy(), is_(True))
    assert_that(the_holy_grail.depth, greater_than(5))
    assert_that(the_holy_grail.width, greater_than(6))
    assert_that(the_holy_grail.height, greater_than(7))
```

this should be written as:

```python
def test_the_holy_grail():
    the_holy_grail = seek_the_holy_grail()
    assert_that(
        the_holy_grail,
        is_holy()
            .with_depth(greater_than(5))
            .with_width(greater_than(6))
            .with_height(greater_than(7))
    )
```

or:

```python
def test_the_holy_grail():
    the_holy_grail = seek_the_holy_grail()
    assert_that(
```

(continues on next page)

(continued from previous page)

```
        the_holy_grail,
        grail(holy=True, width=5))
```

Both these examples, however, require writing your own matchers. With this toolbox, it is easy.

2.1 MulticomponentMatcher

The `MulticomponentMatcher` allows writing the chain-style matchers.

All you have to do is to write your `is_holy` matcher that inherits from the `MulticomponentMatcher` as the backbone. Then you write individual matchers for each of the holy grail properties enhancing them with the `MatcherPlugin`, and you register them with the `is_holy` matcher.

So, this is your `is_holy` matcher:

```
class IsHolyMatcher(MulticomponentMatcher):
    def __init__(self):
        super().__init__()

def is_holy():
    return IsHolyMatcher()
```

And that's it. You don't have to override the usual matcher methods. Everything will be done by the parent class. However, it doesn't do any matching yet, so we need to write the plugins. Let's start with the actual holiness:

```
class HolinessMatcher(MatcherPlugin):
    def __init__(is_holy=True):
        super().__init__()
        self.is_holy = is_holy

    def component_matches(self, item):
        # the item will be a grail
        return self.is_holy == item.is_holy()

    def describe_to(self, description):
        description.append_text(
            "A grail which is {}holy".format("" if self.is_holy else "not "))

    def describe_component_mismatch(self, item, mismatch_description):
        mismatch_description.append_text(
            "The grail was {}holy".format("" if item.is_holy() else "not "))
```

And then you register it with the main matcher:

```
class IsHolyMatcher(MulticomponentMatcher):
    def __init__(self, is_holy):
        super().__init__()
        self.register(HolynessMatcher(is_holy))

def holy(is_holy):
    return IsHolyMatcher(is_holy)
```

Of course, you could write that `HolinessMatcher` logic in your `IsHolyMatcher`, but if we have the power of plugins, then why not use it?

For now, we only have this bit: `assert_that(the_grail, is_holy())`, and not the `.with_width(...)` stuff. So let's write it. I won't go through the process of writing the plugin for the width as it is rather straightforward, but here's how you register it with the main matcher:

```
class IsHolyMatcher(MulticomponentMatcher):
    def __init__(self, is_holy):
        super().__init__()
        self.register(HolinessMatcher(is_holy))

    def with_width(self, value):
        return self.register(GrailWidthMatcher(value))

def holy(is_holy):
    return IsHolyMatcher(is_holy)
```

Now you can do the `is_holy().with_width(greater_than(5))` stuff. **Note that you have to return `self.register(...)` from the plugin registering methods**, as (a) you might want to chain them, and (b) the result of the chain still needs to be a matcher.

2.2 KwargMulticomponentMatcher

This matcher allows writing the kwarg-style matchers (as in the second example above), which are more pythonic, but look kind of unnatural when you want to match against another matcher instead of a plain value. I will show what I mean in a minute.

The general approach is the same as with the multicomponent matcher: you write matcher plugins for your components, and then you register them with your main matcher:

```
class GrailMatcher(KwargMulticomponentMatcher):
    def __init__(self, holy=None, width=None):
        self.register_for_kwarg(HolinessMatcher(holy), holy)
        self.register_for_kwarg(GrailWidthMatcher(width), width)
```

And then in your tests you do:

```
def test_correct_width_wrong_holiness(self, my_grail):
    assert_that(
        my_grail,
        grail(holy=True, width=4))
```

As I said before, this looks more pythonic, however, if you want to check your values against matchers, and not just plain values (like *width=4* here), your code starts looking a bit strange:

```
def test_correct_width_wrong_holiness(self, my_grail):
    assert_that(
        my_grail,
        grail(holy=True, width=greater_than(4)))
```

My recommendation is to use the chain-style matchers if you know that your main matcher might be used this way.

2.3 Demos

You can find the demos for both approaches in the *demo* folder of this repo. Clone it, install the requirements from *demo/requirements.txt*, and run *pytest demo/test_**

Index