pypcom Documentation

Release latest

Chris NeJame

May 09, 2020

User Guide:

1 Why Use PyPCOM **3**
- 1.1 Quick Example . . . 3
  - 1.1.1 "What about the fixture?" . . . 4
  - 1.1.2 "Couldn't I do that with a normal POM?" . . . 4

2 Installation **5**
- 2.1 Requirements . . . 5
- 2.2 Installing PyPCOM . . . 5

3 Components **7**
- 3.1 Defining . . . 7
  - 3.1.1 Adding Sub-Components . . . 7
  - 3.1.2 Adding Custom Methods . . . 8
- 3.2 Enterring Text . . . 8
  - 3.2.1 Advanced . . . 8
- 3.3 Waiting . . . 9
- 3.4 Sub-Components and *_find_from_parent* . . . 9
  - 3.4.1 Simple Example . . . 10
  - 3.4.2 Advanced Example . . . 11
- 3.5 Deferring Attribute Lookups (Or "How does it do that?") . . . 12
  - 3.5.1 Why Descriptors? . . . 12
  - 3.5.2 How does it support selenium methods/attributes like it does? . . . 12

4 Using PyPCOM in Automated Tests **15**
- 4.1 The Tests . . . 15
  - 4.1.1 Quick Example . . . 15

5 State **17**
- 5.1 How to Use . . . 17
- 5.2 How It Works . . . 18

6 ExpectedAttribute **19**
- 6.1 Defining Your Own . . . 19
- 6.2 Provided ExpectedAttribute Classes . . . 20

Python Module Index **23**

Index 25

PyPCOM is a component based Page Object Model meant to work with Selenium, that is designed to make Page Object development and maintanence significantly faster and easier, while at the same time making your structures easier to work with and much more extensible and reusable. It does this by allowing you to break up your pages into logical components with their own namespace and functionality, while still allowing you to treat every component as if it were an element.

Still not convinced? Check out the *Why Use PyPCOM* section for a more detailed explanation on what PyPCOM offers.

Ready to get started? You can jump to any of the sections in the user guide below.

CHAPTER 1

Why Use PyPCOM

This was designed to make writing page objects extremely fast and simple. Because Pages and Components are both classes, making templated structures to build off of and inherit from is incredibly easy.

It even allows you to shorten and simplify the locators you use for sub- components by letting you have the sub-component search within its parent- component only. That way you don't have to worry about having incredibly long, fragile, and hard-to-maintain locators.

Everything is based around classes and descriptors, so the components you make can be easily inherited from to extend their functionality or reused. This also makes writing fixtures and tests easier because you can focus more on the behavior and state of each components, and abstract the cumbersome details into the code of each component.

Debugging, abstraction, and documentation are all also easier because you now have the capability to deal with the different sections of a page individually and can compartmentalize the logic into different classes and methods.

Are you using a framework with templates to build your frontend, and/or have a lot of common strutures in the HTML? You can use this model to create component templates yourself, enabling faster, easier, and more manageable page object development.

1.1 Quick Example

Login pages are very common, and the functionality is almost always the same. You can almost always expect a username field, a password field, and a submit button. Submitting the form should be a straightforward task. So here's how PyPCOM can be used to set up this structure:

```
class Username(PageComponent):
    _find_from_parent = True
    _locator = (By.ID, "username")

class Password(PageComponent):
    _find_from_parent = True
    _locator = (By.ID, "password")

class LoginForm(PageComponent):
```

(continues on next page)

(continued from previous page)

```
    username = Username()
    password = Password()
    _locator = (By.ID, "loginForm")
    def fill_out(self, username, password, **kwargs):
        self.username = username
        self.password = password

class LoginPage(Page):
    form = LoginForm()
    def login(self, **credentials):
        self.form.fill_out(**credentials)
        self.form.submit()
```

Notice that both *fill_out* and *submit* are called on *self.form* inside *LoginPage*'s *login* method. This is possible because you are able to treat components both as components with custom defined methods, as well as `WebElement` objects. This is because PyPCOM defers attribute lookups to the `WebElement` object (assuming it can be found) if the component doesn't have the relevant attribute itself. However, if the `WebElement` object doesn't have the attribute, then it shows the component as not having that attribute.

1.1.1 "What about the fixture?"

The fixture would actually be quite easy to create, and it could be made to be parameterizable. Assuming you're using pytest, it would just look like this:

```
@pytest.fixture
def login(page, credentials):
    page.login(**credentials)
```

And with that, you have something that allows you to provide whatever credentials you want. You can even change the page fixture to give you a different page object that can handle whatever custom loggic is needed. The *login* fixture can even be parametrized indirectly for other purposes.

1.1.2 "Couldn't I do that with a normal POM?"

Yes, but, because of the compartmentalized structure of this approach, if you need to adjust how you log in because of a different login form (e.g. maybe there's an additional field, or the locators are different), you can still use almost everything the *LoginPage* class by inheriting from it and just replacing the *form* component.

It may not seem like much of a benefit at this small of a scale, but for more complex pages, it can save you a lot of effort and time.

CHAPTER 2

Installation

2.1 Requirements

PyPCOM is designed to work with Selenium, and as a result, depends on it for some things. Installing PyPCOM will automatically install Selenium for Python as well, if you don't already have it, so you shouldn't have to do anything extra to get it working.

2.2 Installing PyPCOM

Just like with most other Python packages, you can install it from PyPI using pip:

```
pip install pypcom
```

You can also install it from the source:

```
python setup.py install
```

CHAPTER 3

Components

These are the heart of soul of this framework.

At a very basic level, components are just classes that have a locator for a specific element, and get used as descriptors in pages or other components. When referenced, they can be treated as though you are referencing the `WebElement` they're associated with. That means you can reference their *.text* property or *.is_displayed()* on them. You can even do this for a component that has sub-components of its own.

But they offer much more convenience than that. You can read below on how they work and how you can get the most out of them.

3.1 Defining

Defining a component is ver straightforward. All you need to do is make a class that inherits from `PageComponent` (or `PC` for something shorter), and give it a locator that can be passed to *.find_element()*. It would look something like this:

```
class MyComponent(PC):
    _locator = (By.ID, "my-id")
```

3.1.1 Adding Sub-Components

If you want to add sub-components to it, it's identical to adding a component to the page class. You just need to add it like a decorator:

```
class MySubComponent(PC):
    _locator = (By.ID, "my-other-id")

class MyComponent(PC):
    _locator = (By.ID, "my-id")
    my_sub_component = MySubComponent()
```

To put *MyComponent* in a page class, you can then just add it in like you normally would:

```
class MyPage(Page):
    my_component = MyComponent()
```

You can then reference the normal `WebElement` attributes/methods and the sub-component like this:

```
page.my_component.is_displayed()
page.my_component.text
page.my_component.my_sub_component.is_displayed()
page.my_component.my_sub_component.text
```

3.1.2 Adding Custom Methods

Adding in custom methods is just as easy:

```
class MySubComponent(PC):
    _locator = (By.ID, "my-other-id")

    def do_more_things(self):
        # do more stuff
        pass

class MyComponent(PC):
    _locator = (By.ID, "my-id")
    my_sub_component = MySubComponent()

    def do_something(self):
        # do stuff
        pass
```

They can now be used just like the normal attributes and methods provided by `WebElement` and `PageComponent`. This also doesn't change how it works normally, so long as you don't add any sub-components or custom methods that would interfere with the normal `WebElement` or `PageComponent` methods/attributes.

3.2 Enterring Text

Enterring text is easy. For a given component, the locator just needs to point to the actual *input* element, and then you can invoke `send_keys()` through the = operator like this:

```
page.my_form.my_input = "something"
```

3.2.1 Advanced

If you need to change how this behavior works, you can override the *__set__* method in your component. Just make sure you look at how it works normally, so you basically duplicate it, and only modify the part where it invokes `send_keys()` to make sure it continues working as it needs to.

This approach will likely change in the future to provide a more convenient hook to override, but any additional hook will not break a custom *__set__* implementation if it copies the current one.

3.3 Waiting

Waiting is simple, too. You can either call `PageComponent.wait_until()` or `PageComponent.wait_until_not()` on the component you want to perform the wait on, and pass it a string for the condition you want to wait for. The three available conditions are *"present"*, *"visible"*, and *"clickable"*. More will be added soon, along with a system for passing custom condition callables.

Here's a quick example of its usage:

```
page.component.wait_until("visible", timeout=5)
```

It accepts strings that correspond to the normal expected conditions you've seen. But you can also reference expected conditions you've defined yourself and attached to the `PageComponent` in its *_expected_condition* attribute. Here's an example of how it can be set up:

```
def custom_visible_condition(component):
    def callable(driver):
        return component.is_displayed()
    return callable

class MyComponent(PC):
    _locator = (...)
    _expected_conditions = {
        "custom_visible": custom_visible_condition,
    }
```

and here's how you'd use it:

```
page.my_component.wait_until("custom_visible")
```

You can also pass in the callable directly, like this:

```
page.my_component.wait_until(custom_visible_condition)
```

If you need to, you can provide additional keyword arguments for more flexible logic. Of course, you'll have to make sure you can handle it properly within the callable. For example, if you have some more advanced component structures and need to perform a query that goes beyond normal selenium logic, you could implement a *query* method (with whatever name you want, of course) and provide the necessary query details at the time the wait is executed. This might be how your callable looks:

```
def custom_query_condition(component, **query_details):
    def callable(driver):
        return component.query(**query_details)
    return callable
```

Then you could add it to the *_expected_conditions* dict attribute of that component, maybe as "complex_component_present", and invoke it like this:

```
page.my_component.wait_until("complex_component_present", **query_details)
```

3.4 Sub-Components and *_find_from_parent*

Often, you will find yourself with long and convoluted selectors, simply because the element you want to find is in some heavily nested node, and you have to repeat parts of your selector in many sub-components.

PyPCOM offers a solution to this that lets you simply search for a sub-component's associated `WebElement` within its parent component's `WebElement` by calling `find_element()` on that instead of the driver. This allows you to give the sub-component a locator that is relative to its parent component's `WebElement`, so you don't have to keep repeating the common parts of the locator, and can instead create a simpler, cleaner, and more appropriate locator than you might not have been able to otherwise.

To use it, all you have to do is set *_find_from_parent* to *True* in the class definition of the sub-component. The parent components don't need to be aware of this, so long as they have a *_locator* of their own.

3.4.1 Simple Example

Let's say you have the following collection of elements somewhere in your page:

```
<div class='some-area'>
    <div class='content-section'>
        <img src='iamges/myImage.png' />
        <p class='content'>Some text content.</p>
        <a href='something.html'>Some Link</a>
    </div>
</div>
```

To reliably find these elements, you might havee to use a involving references to both parent elements. For example:

```
class MyImage(PC):
    _locator = (By.CSS_SELECTOR, "div.some-area div.content-section img")

class SomeContent(PC):
    _locator = (By.CSS_SELECTOR, "div.some-area div.content-section p")

class SomethingLink(PC):
    _locator = (By.CSS_SELECTOR, "div.some-area div.content-section a")

class SomeContentSection(PC):
    _locator = (By.CSS_SELECTOR, "div.some-area div.content-section")
    my_image = MyImage()
    some_content = SomeContent()
    something_link = SomethingLink()

class SomeArea(PC):
    _locator = (By.CSS_SELECTOR, "div.some-area")
    some_content_section = SomeContentSection()
```

If you had to do that for several elements throughout all of your pages, that would get tedious very quickly and would involve a lott of repeating yourself. Not to mention, this would also make all those locators fragile, and if they break, it would take quite a while to fix each one.

Using *_find_from_parent* cuts out all that repetition and compartmentalizes your locator logic:

```
class MyImage(PC):
    _find_from_parent = True
    _locator = (By.TAG_NAME, "img")

class SomeContent(PC):
    _find_from_parent = True
    _locator = (By.TAG_NAME, "p")

class SomethingLink(PC):
```

(continues on next page)

(continued from previous page)

```
    _find_from_parent = True
    _locator = (By.TAG_NAME, "a")

class SomeContentSection(PC):
    _find_from_parent = True
    _locator = (By.CSS_SELECTOR, "div.content-section")
    my_image = MyImage()
    some_content = SomeContent()
    something_link = SomethingLink()

class SomeArea(PC):
    _locator = (By.CSS_SELECTOR, "div.some-area")
    some_content_section = SomeContentSection()
```

3.4.2 Advanced Example

This also allows for making complex, generic component structures that can be re-used in several places. Let's say you have a common structure for your form control elements in all your forms where each field has an <*input*> element and a <*label*> bundled inside its own <*div*>. It would look something like this:

```
<div class='form-field'>
    <label for='first-name'>First Name:</label>
    <input id='first-name' name='first-name' />
</div>
<div class='form-field'>
    <label for='last-name'>Last Name:</label>
    <input id='last-name' name='last-name' />
</div>
```

This would be tedious to have to define a label and input component for every field in your site. But you could create a generic structure like this that you could reuse:

```
class Label(PC):
    _find_from_parent = True
    _locator = (By.TAG_NAME, "label")

class Input(PC):
    _find_from_parent = True
    _locator = (By.TAG_NAME, "input")

class FormField(PC):
    label = Label()
    input = Input()

    def __set__(self, instance, value):
        self._parent = instance
        self.driver = self._parent.driver
        self.input = value
```

With that, you could just inherit from *FormField* to make a new class for each field, and it would even let you assign a value to the input by setting the field component itself (i.e. *page.form.my_field = "something"*). You could even get a little fancy with the locator to make sure you always find the right field <*div*>:

```
class FirstNameField(FormField):
    _locator = (
        By.XPATH,
        (
            "//div[contains(concat(' ', @class, ' '), ' form-field ')]"
            "[input[@id='first-name']]"
        ),
    )

class LastNameField(FormField):
    _locator = (
        By.XPATH,
        (
            "//div[contains(concat(' ', @class, ' '), ' form-field ')]"
            "[input[@id='last-name']]"
        ),
    )
```

That XPATH would locate a * *' that both has a single class of *form-field*, and also contains an *<input>* with the desired *id*. It won't find the *<input>* itself; it just finds the right * * that contains it. But that's intended. This way we know we found the element that contains only that *<input>* and its *<label>*, and we can let the *FormField* class hold all the common logic.

3.5 Deferring Attribute Lookups (Or "How does it do that?")

3.5.1 Why Descriptors?

PyPCOM works using descriptors for the components, but the only things it really uses that for are making sure a reference to the *driver* and each component's parent component/page is accessible, and to allow for convenient value setting.

PyPCOM needs to make sure that, before it does anything, as a component is referenced (either through *__get__* or *__set__*), it grabs the reference to the *driver* from the managing instance, storing a reference to both the driver and the instance in the component itself so that they can be referenced later on. For example, if you were to reference something like:

```
page.some_component.another_component = "some text"
```

some_component would be referenced through *__get__* and get a reference to the driver from *page*. It would also store a reference to *page* as its parent. *another_component* would then be referenced through *__set__* and get a reference to the driver from *some_component*. It would also store a reference to *some_component* as its parent.

Descriptors also means classes will be used, so you can define custom behavior, inherit behavior from other components, and re-use components as much as you want.

3.5.2 How does it support selenium methods/attributes like it does?

PyPCOM relies on the default attribute lookup behavior of objects in Python. If a class instance, or the class itself does not have a certain attribute defined, then Python calls the object's *__getattr__* method (assuming it has one defined).

For components, when you reference an attribute of them, if the component instance has no such attribute, and neither does its class, then the component instance attempts to find its associated `WebElement` and get the attribute from there. If the `WebElement` doesn't have that attribute, then PyPCOM will tell you that the component doesn't have

the attribute. If the component doesn't have a *_locator* defined, or the `WebElement` can't be located, PyPCOM will raise an appropriate error.

Because there is a finite, established set of `WebElement` attributes, PyPCOM assumes that you must be looking for a component's attributes if it can't find them on the `WebElement`. As a result, when it can't find an attribute, the error it raises will tell you that the component was the one without the attribute. This does not mean that it didn't try to find the attribute on the `WebElement`

CHAPTER 4

Using PyPCOM in Automated Tests

PyPCOM was built around writing end-to-end/UI tests for websites, and, as a result, comes with several features to make that process easier. While it will work within any Python testing framework, it was built with pytest in mind, so examples and code snippets will be written assuming your test framework is pytest.

4.1 The Tests

PyPCOM comes with two handy classes to assist with writing tests: `State` and `ExpectedAttribute`. Using them, you can check the state of a component against several things at once with only a single assert statement. If you're using pytest, it will also automatically provide an organized message of all the problems it found in the failure message.

A `State` is used to compare against the component, while multiple `ExpectedAttribute` objects are passed to the `State` when it's instantiated. The `ExpectedAttribute` objects are responsible for knowing how to check the values off the component and reporting any problems. The `State` is just there to manage the comparison of each `ExpectedAttribute` against the component and generate a failure message for pytest to show in the failure output.

4.1.1 Quick Example

Assuming you have all the page objects and fixtures defined, writing a test can be as easy as this (assume this is a method inside a test class):

```
def test_some_component(self, page):
    assert page.some_component == State(
        IsDisplayed(),
        Text("My Text"),
        TagName("p"),
    )
```

If the element was found to be displayed, but had different text and wasn't a <*p*> tag, you would see a failure that looks something like this:

```
Comparing SomeComponent State:
    Text: "Something else" != "My Text"
    TagName: "div" != "p"
```

The *IsDisplayed*, *Text*, and *TagName* classes you see all inherit from `ExpectedAttribute`. PyPCOM comes with several baked in (see *Provided ExpectedAttribute Classes* for a full list), but the system is designed to easily extended so you can add your own. For more detail on that, check out *State* or *ExpectedAttribute*.

CHAPTER 5

State

`State` is used to bundle up multiple `ExpectedAttribute` objects so that they can all be checked against a `PageComponent` with a single *assert* statement. The intent is to achieve the following:

- **Maintain the best practice of only using a single *assert* statement per test** test function/method, while still allowing the test to check multiple things at once.
- Speed up and simplify writing complex tests for a page's components.
- Make the code for tests more readable/writable and compact.
- Provide concise, readable output containing all problems should the test fail.

5.1 How to Use

Assuming you have all the page objects and fixtures defined, writing a test can be as easy as this (assume this is a method inside a test class):

```
def test_some_component(self, page):
    assert page.some_component == State(
        IsDisplayed(),
        Text("My Text"),
        TagName("p"),
    )
```

If the element was found to be displayed, but had different text and wasn't a <*p*> tag, you would see a failure that looks something like this:

```
Comparing SomeComponent State:
    Text: "Something else" != "My Text"
    TagName: "div" != "p"
```

The *`IsDisplayed`*, *`Text`*, and *`TagName`* classes you see all inherit from `ExpectedAttribute`. PyPCOM comes with several baked in (see *Provided ExpectedAttribute Classes* for a full list), but the system is designed to easily extended so you can add your own. For more detail on that, check out *State* or *ExpectedAttribute*.

5.2 How It Works

When you first make the `State` object, you pass it one or more `ExpectedAttribute` objects, which it holds onto. The `ExpectedAttribute` objects are responsible for knowing how to check the `PageComponent` for any problems, and storing them for later reference. The `State` object runs through all the `ExpectedAttribute` objects in it's `__eq__()` method, and once all the `ExpectedAttribute` objects are checked against the `PageComponent` object, the `State` object checks their results. If it sees that any problems were found, the comparison will just evaluate to `False`.

For most testing frameworks, that's as far as it will go. But if you're using **pytest_**, then when it comes time for it to print out the failure report, the `State` object will be used to generate a more readable failure report message by having it compile the list of problems reported by each `ExpectedAttribute` object. It's able to do this after the tests have long since been evaluated, because each of the `ExpectedAttribute` objects hold onto their findings, and the `State` object keeps a reference to each of them.

CHAPTER 6

ExpectedAttribute

The `ExpectedAttribute` objects serve as a means of compartmentalizing the logic for both how to check for something specific against a `PageComponent`, and how to summarize any problems found. You can pass any number of `ExpectedAttribute` objects to a `State` object when you create it.

PyPCOM offers plenty of `ExpectedAttribute` classes out of the box, which can be found below. But the system is designed to be customizable so you can inherit from the `ExpectedAttribute` class and define your own checks along with how to report them.

6.1 Defining Your Own

In order to define your own `ExpectedAttribute`, all you need to do is make a class that inherits from `ExpectedAttribute`, and then give it an `__init__` and `compare` method.

The `__init__` method can be used to accept any expected values you want the object to check for.

The `compare` method will be passed a reference to the `PageComponent` when it gets called by the `State` object during the actual comparison. In that method, there's three ways you can track problems you find, all of which are equally recommended (so use whichever you find most appealing):

1. You can manually add problems you find using `add_problem()`
2. Use a standard `assert` statement, with an failure message attached (e.g. `assert True is False, "True is not False"`)
3. Raise an `AssertionError` manually with a provided failure message (e.g. `raise AssertionError("some failure message")`)

If you choose the first option, it's recommended that you stick with `AssertionError``s, or at least provide an object with a ``.message` attribute so that `ExpectedAttribute` can find the problem's message in the way it normally does.

Here's a quick example of a custom `ExpectedAttribute`, which is provided already in PyPCOM (with the doc-strings removed, however):

```
class IsDisplayed(ExpectedAttribute):
    _msg = {
        True: "Element is not displayed when it should be",
        False: "Element is displayed when it shouldn't be",
    }
    def __init__(self, expected=True):
        self._expected = bool(expected)
    def compare(self, other):
        assert other.is_displayed() is self._expected, self._msg[self._expected]
```

6.2 Provided ExpectedAttribute Classes

class pypcom.state.expected_attribute.**Href**(*expected=True*)
Checks if the component has a certain href value or not.

Args: expected (bool): What href value the element should have.

compare(*other*)
Check the element's href.

class pypcom.state.expected_attribute.**IsDisplayed**(*expected=True*)
Checks if the component is currently displayed on the page or not.

Args: expected (bool): Whether or not the element should be displayed.

compare(*other*)
Check if the element is displayed or not.

class pypcom.state.expected_attribute.**IsEnabled**(*expected=True*)
Checks if the component is currently enabled on the page or not.

Args: expected (bool): Whether or not the element should be enabled.

compare(*other*)
Check if the element is enabled or not.

class pypcom.state.expected_attribute.**IsPresent**(*expected=True*)
Checks if the component is currently present on the page or not.

Args: expected (bool): Whether or not the element should be present.

compare(*other*)
Check if the element is present or not.

The output will be one of the following:

```
IsPresent: Element is not present when it should be
IsPresent: Element is present when it shouldn't be
```

class pypcom.state.expected_attribute.**Placeholder**(*expected=True*)
Checks if the component has a certain placeholder value or not.

Args: expected (bool): What placeholder the element should have.

compare(*other*)
Check the element's placeholder.

class pypcom.state.expected_attribute.**TagName**(*expected=True*)
Checks if the component has a certain tag name or not.

Args: expected (bool): What tag name the element should have.

compare(*other*)
Check the element's tag name.

class pypcom.state.expected_attribute.**Text**(*expected=True*)
Checks if the component has certain text or not.

Args: expected (bool): What text the element should have.

compare(*other*)
Check the element's text.

class pypcom.state.expected_attribute.**Type**(*expected=True*)
Checks if the component is of a certain type or not.

Args: expected (bool): What type the element should be.

compare(*other*)
Check the element's type.

Python Module Index

p

pypcom.state.expected_attribute, 20

Index

C

compare() (*pypcom.state.expected_attribute.Href method*), 20
compare() (*pypcom.state.expected_attribute.IsDisplayed method*), 20
compare() (*pypcom.state.expected_attribute.IsEnabled method*), 20
compare() (*pypcom.state.expected_attribute.IsPresent method*), 20
compare() (*pypcom.state.expected_attribute.Placeholder method*), 20
compare() (*pypcom.state.expected_attribute.TagName method*), 21
compare() (*pypcom.state.expected_attribute.Text method*), 21
compare() (*pypcom.state.expected_attribute.Type method*), 21

H

Href (*class in pypcom.state.expected_attribute*), 20

I

IsDisplayed (*class in pypcom.state.expected_attribute*), 20
IsEnabled (*class in pypcom.state.expected_attribute*), 20
IsPresent (*class in pypcom.state.expected_attribute*), 20

P

Placeholder (*class in pypcom.state.expected_attribute*), 20
pypcom.state.expected_attribute (*module*), 20

T

TagName (*class in pypcom.state.expected_attribute*), 20
Text (*class in pypcom.state.expected_attribute*), 21
Type (*class in pypcom.state.expected_attribute*), 21