
Stere Documentation

Release 0.7.0

Joshua Fehler

May 02, 2019

Contents:

1	Documentation	1
1.1	Getting Started	1
1.2	Pages	2
1.3	Fields	3
1.4	Splinter Integration	7
1.5	Appium Integration	11
1.6	Areas	12
1.7	Repeating	16
1.8	Workflows	18
1.9	Fields Returning Objects	18
1.10	Best Practices	19
2	Indices and tables	23

1.1 Getting Started

1.1.1 Requirements

Python \geq 3.6

1.1.2 Installation

Stere can be installed with pip using the following command:

```
pip install stere
```

1.1.3 Setup

Specifying the automation library

Using a *stere.ini* file, the automation library used can be specified. This determines which library specific Fields are loaded.

While only Splinter and Appium have custom Fields to take advantage of their specific capabilities, any automation library that implements an API similar to Selenium should be possible to connect to Stere.

splinter is used by default, and *appium* is supported. Any other value will be accepted, which will result in no specific Fields being loaded.

```
[stere]
library = appium
```

Stere.browser

Stere requires a browser (aka driver) to work with. This can be any class that ultimately drives automation. Pages, Fields, and Areas inherit their functionality from this object.

Here's an example with [Splinter](#):

```
from stere import Stere
from splinter import Browser

Stere.browser = Browser()
```

As long as the base Stere object has the browser set, the browser's functionality is passed down to everything else.

Stere.url_navigator

Optionally, an attribute called *url_navigator* can be provided a string that maps to the method in the browser that opens a page.

In Splinter's case, this is the *visit* method.

```
from stere import Stere
from splinter import Browser

Stere.browser = Browser()
Stere.url_navigator = 'visit'
```

This attribute is used by the *Page* class to make url navigation easier.

1.2 Pages

class stere.Page

Represents a single page in an application. The Page class is the base which all Page Objects should inherit from.

Inheriting from Page is not required for Fields or Areas to work.

All attribute calls that fail are then tried on the browser attribute. This allows classes inheriting from Page to act as a proxy to whichever browser/driver is being used.

Using Splinter's browser.url method as an example, the following methods are analogous:

```
>>> MyPage.url == MyPage.browser.url == browser.url
```

The choice of which syntax to use depends on how you want to write your test suite.

navigate ()

When the base Stere object has been given the *url_navigator* attribute, and a Page Object has a *page_url* attribute, the *navigate()* method can be called.

This method will call the method defined in *url_navigator*, with *page_url* as the first parameter.

Returns The instance where *navigate()* was called from.

Return type *Page*

Example

```
>>> from splinter import Browser
>>> from stere import Page
>>>
>>>
>>> class Home(Page):
>>>     def __init__(self):
>>>         self.page_url = 'https://en.wikipedia.org/'
>>>
>>>
>>> Stere.browser = Browser()
>>> Stere.url_navigator = 'visit'
>>>
>>> home_page = Home()
>>> home_page.navigate()
```

1.2.1 Using Page as a Context Manager

Page contains `__enter__()` and `__exit__()` methods. This allows any page to be used as a Context Manager.

Example:

```
from pages import Home

with Home() as p:
    p.login_button.click()
```

1.3 Fields

1.3.1 Field

class `stere.fields.Field`

includes ()

Will search every element found by the Field for a value property that matches the given value. If an element with a matching value is found, it's then returned.

Useful for when you have non-unique elements and know a value is in one of the elements, but don't know which one.

Parameters **value** (*str*) – A text string inside an element you want to find.

Returns element

Example

```
>>> class PetStore(Page):
>>>     def __init__(self):
>>>         self.inventory = Link('xpath', '//li[@class="inv"]')
>>>
```

(continues on next page)

(continued from previous page)

```
>>> pet_store = PetStore()
>>> pet_store.inventory_list.includes("Kittens").click()
```

before()

Called automatically before methods with the `@use_before` decorator are called.

Performer methods are decorated with `@use_before`.

By default it does nothing. Override this method if an action must be taken before a method is called.

In the following example, Dropdown has been subclassed to hover over the element before clicking.

Example

```
>>> from stere.fields import Dropdown
>>>
>>> class CSSDropdown(Dropdown):
>>>     def before(self):
>>>         self.element.mouse_over()
```

after()

Called automatically before methods with the `@use_after` decorator are called.

Performer methods are decorated with `@use_after`.

By default it does nothing. Override this method if an action must be taken after the method has been called.

value_contains()

Check if the value of the Field contains an expected value.

Parameters

- **expected** (*str*) – The expected value of the Field
- **wait_time** (*int*) – The number of seconds to wait

Returns True if the value was found, else False

Return type bool

Example

```
>>> class PetStore(Page):
>>>     def __init__(self):
>>>         self.price = Link('xpath', '//li[@class="price"]')
>>>
>>> pet_store = PetStore()
>>> assert pet_store.price.value_contains("19.19", wait_time=6)
```

value_equals()

Check if the value of the Field equals an expected value.

Parameters

- **expected** (*str*) – The expected value of the Field
- **wait_time** (*int*) – The number of seconds to wait

Returns True if the value was found, else False

Return type `bool`

Example

```
>>> class PetStore(Page):
>>>     def __init__(self):
>>>         self.price = Link('xpath', '//li[@class="price"]')
>>>
>>> pet_store = PetStore()
>>> assert pet_store.price.value_equals("$19.19", wait_time=6)
```

1.3.2 Root

`class stere.fields.Root`

1.3.3 Text

`class stere.fields.Text`

1.3.4 Performer method

A Field can have a single method be designated as a performer. This method will be called when the Field is inside an Area and that Area's `perform()` method is called.

For example, Input's performer is the `fill()` method, and Button's performer is the `click()` method. Given the following Area:

```
search = Area(
    query=Input('id', 'xsearch'),
    submit=Button('id', 'xsubmit'),
)
```

and the following script:

```
search.perform('Orange')
```

When `search.perform('Orange')` is called, `query.fill('Orange')` is called, followed by `submit.click()`.

See the documentation for [Area](#) for more details.

1.3.5 Calling the performer method explicitly

The performer method is available as `Field.perform()`. Calling it will run the performer method, but they are not aliases.

No matter what the return value of the performer method is, the return value from calling `Field.perform()` will always be the `Field.returns` attribute.

Using the splinter Button Field as an example, the only difference between `Button.click()` and `Button.perform()` is that `perform` will return the object set in the `Field.returns` attribute. See [Returning Objects](#) for more details.

1.3.6 Calling the performer method implicitly

When a page instance is called directly, the *perform()* method will be executed.

The following code will produce the same results:

```
button = Button()
button.perform()
```

```
button = Button()
button()
```

1.3.7 Subclassing Field

Field can be subclassed to suit your own requirements.

If the `__init__()` method is overwritten, make sure to call `super()` before your own code.

If your class needs specific behaviour when interacting with Areas, it must be wrapped with the `@stere_performer` decorator to specify a performer method.

When creating a new type of Field, the `stere_performer` class decorator should be used to assign a performer method.

1.3.8 Field Decorators

`decorators.stere_performer()`

Wrapper for classes that contain a method which should be used by `Area.perform()`.

Associating a method with `perform` allows the class to be fully used by Area objects.

Parameters

- **method_name** (*str*) – The name of the method to perform
- **consumes_args** (*bool*) – True if the method takes an argument, else False

In the following example, when `Philosophers().diogenes_area.perform()` is called, `DiogenesButton.philosophize()` is called.

Example

```
>>> @stere_performer('philosophize', consumes_arg=False)
>>> class DiogenesButton(Field):
>>>     def philosophize(self):
>>>         print("As a matter of self-preservation, ")
>>>         print("a man needs good friends or ardent enemies, ")
>>>         print("for the former instruct him and the latter ")
>>>         print("take him to task.")
>>>
>>>
>>> class Philosophers(Page):
>>>     def __init__(self):
>>>         self.diogenes_area = Area(
>>>             quote_button=DiogenesButton('id', 'idDio'),
>>>             next_button=Button('id', 'idNext'),
>>>         )
```

(continues on next page)

(continued from previous page)

```
>>>
>>>
>>> Philosophers().diogenes_area.perform()
```

`decorators.use_before()`

When added to a method in a Field, the Field's `before()` method will be called before the decorated method is called.

Example

```
>>> class TransformingButton(Field):
>>>     def before(self):
>>>         print('Autobots! Transform and...')
>>>
>>>     @use_before
>>>     def roll_out(self):
>>>         print('roll out!')
>>>
>>> tf = TransformingButton()
>>> tf.roll_out()
>>>
>>> "Autobots! Transform and..."
>>> "roll out!"
```

`decorators.use_after()`

When added to a method in a Field, the Field's `after()` method will be called after the decorated method is called.

Example

```
>>> class TransformingButton(Field):
>>>     def after(self):
>>>         print('rise up!')
>>>
>>>     @use_after
>>>     def transform_and(self):
>>>         print('Decepticons, transform and...')
>>>
>>> tf = TransformingButton()
>>> tf.transform_and()
>>>
>>> 'Decepticons, transform and...'
>>> "rise up!"
```

1.4 Splinter Integration

Stere contains Fields designed specifically for when Splinter is connected. Each implements a specific performer method.

All Fields designed for Splinter also inherit the following convenience methods:

`SplinterBase.is_present()`
Checks if an element is present in the DOM.

Takes the same arguments as Splinter's *is_element_present_by_xpath*

`SplinterBase.is_not_present()`

Checks if an element is not present in the DOM.

Takes the same arguments as Splinter's *is_element_not_present_by_xpath*

`SplinterBase.is_visible()`

Checks if an element is present in the DOM and visible.

Parameters `wait_time` (*int*) – The number of seconds to wait

`SplinterBase.is_not_visible()`

Checks if an element is present in the DOM but not visible.

Parameters `wait_time` (*int*) – The number of seconds to wait

Example:

```
class Inventory(Page):
    def __init__(self):
        self.price = Link('css', '.priceLink')

assert Inventory().price.is_present(wait_time=6)
```

1.4.1 Fields

Button

class `stere.fields.Button`

Convenience Class on top of Field, it implements *click()* as its performer.

`Button.click()`

Uses Splinter's click method.

Example

```
>>> purchase = Button('id', 'buy_button')
>>> purchase.click()
```

Checkbox

class `stere.fields.Checkbox`

By default, the Checkbox field works against HTML inputs with type="checkbox".

Can be initialized with the *default_checked* argument. If True, the Field assumes the checkbox's default state is checked.

It implements *opposite()* as its performer.

`Checkbox.set_to()`

Set a checkbox to the desired state.

Parameters `state` (*bool*) – True for check, False for uncheck

Example

```
>>> confirm = Checkbox('id', 'selectme')
>>> confirm.set_to(True)
```

Checkbox.**toggle()**

If the checkbox is checked, uncheck it. If the checkbox is unchecked, check it.

```
>>> confirm = Checkbox('id', 'selectme')
>>> confirm.toggle()
```

Checkbox.**opposite()**

Switches the checkbox to the opposite of its default state. Uses the *default_checked* attribute to decide this.

```
>>> confirm = Checkbox('id', 'selectme')
>>> confirm.opposite()
```

Dropdown

class stere.fields.**Dropdown**

By default, the Dropdown field works against HTML Dropdowns. However, it's possible to extend Dropdown to work with whatever implementation of a CSS Dropdown you need.

It implements *select()* as its performer.

The *option* argument can be provided to override the default implementation. This argument expects a Field. The Field should be the individual options in the dropdown you wish to target.

```
self.languages = Dropdown('id', 'langDrop', option=Button('xpath', '/h4/a/strong
↪'))
```

Dropdown.**options()**

Searches for all the elements that are an option in the dropdown.

Returns list

Dropdown.**select()**

Searches for an option by its html content, then clicks the one that matches.

Parameters *value* (*str*) – The option value to select.

Raises *ValueError* – The provided value could not be found in the dropdown.

Input

class stere.fields.**Input**

A simple wrapper over Field, it implements *fill()* as its performer.

The *default_value* argument can be provided, which will be used if *fill()* is called with no arguments.

```
self.quantity = Dropdown('id', 'qty', default_value='555')
```

Input.**fill()**

Uses Splinter's fill method.

Parameters *value* (*str*) – The text to enter into the input.

Example

```
>>> first_name = Input('id', 'fillme')
>>> first_name.fill('Joseph')
```

Fills the element with value.

Link

class stere.fields.**Link**

A simple wrapper over Field, it implements *click()* as its performer.

Link.click()

Uses Splinter's click method.

Example

```
>>> login = Link('id', 'loginLink')
>>> login.click()
```

Clicks the element.

1.4.2 Locator Strategies

These represent the way a locator can be searched for.

By default, the strategies available with Splinter are:

- css
- xpath
- tag
- name
- text
- id
- value

These strategies can be overridden with a custom strategy (ie: You can create a custom css strategy with different behaviour).

1.4.3 Custom Locator Strategies

Custom strategies can be defined using the *@strategy* decorator on top of a Class.

Any class can be decorated with *@strategy*, as long as the *_find_all* and *_find_all_in_parent* methods are implemented.

In the following example, the 'data-test-id' strategy is defined. It wraps Splinter's *find_by_xpath* method to simplify the locator required on the Page Object.

```

from stere.strategy import strategy

@strategy('data-test-id')
class FindByDataTestId():
    def _find_all(self):
        """Find from page root."""
        return self.browser.find_by_xpath(f'//*[@data-test-id="{self.locator}"]')

    def _find_all_in_parent(self):
        """Find from inside parent element."""
        return self.parent_locator.find_by_xpath(f'//*[@data-test-id="{self.locator}"]')

```

With this implemented, Fields can now be defined like so:

```
my_button = Button('data-test-id', 'MyButton')
```

Support for data-* attributes is also available via the `add_data_star_strategy` function:

```

from stere.strategy import add_data_star_strategy

add_data_star_strategy('data-test-id')

```

This will automatically add the desired data-* attribute to the valid Splinter strategies.

1.5 Appium Integration

Stere contains Fields designed specifically for when Appium is connected. Each implements a specific performer method.

1.5.1 Fields

Button

class `stere.fields.Button`

Convenience Class on top of Field, it implements `click()` as its performer.

`Button.click()`

Uses Appium's click method.

Example:

```

>>> purchase = Button('id', 'buy_button')
>>> purchase.click()

```

Input

class `stere.fields.Input`

A simple wrapper over Field, it implements `send_keys()` as its performer.

The `default_value` argument can be provided, which will be used if `send_keys()` is called with no arguments.

```
self.quantity = Dropdown('id', 'qty', default_value='555')
```

`Input.send_keys()`

Uses Appium's fill method.

Parameters `value` (*str*) – The text to enter into the input.

Example:

```
>>> first_name = Input('id', 'fillme')
>>> first_name.send_keys('Joseph')
```

Fills the element with value.

1.5.2 Locator Strategies

These represent the way a locator will be searched for.

By default, the strategies available are:

- `accessibility_id`
- `android_uiautomator`
- `ios_class_chain`
- `ios_predicate`
- `ios_uiautomation`

These strategies can be overridden with a custom strategy (ie: You can create a custom `accessibility_id` strategy with different behaviour).

1.6 Areas

Areas represent groupings of Fields on a Page.

The following Area objects are available:

- `Area`: A non-hierarchical, unique group of Fields.
- `RepeatingArea`: A hierarchical, non-unique group of Areas. They require a Root Field.

class `stere.areas.Area`

A collection of unique fields.

The Area object takes any number of Fields as arguments. Each Field must be unique on the Page and only present in one Area.

Example:

```
>>> from stere.areas import Area
>>> from stere.fields import Button
>>>
>>> class Album(Page):
>>>     def __init__(self):
>>>         self.tracks = Area(
>>>             first_track=Button('xpath', '//my_xpath_string'),
>>>             second_track=Button('xpath', '//my_xpath_string'),
```

(continues on next page)

(continued from previous page)

```

>>>         third_track=Button('xpath', '//my_xpath_string'),
>>>     )
>>>
>>> def test_stuff():
>>>     album = Album()
>>>     album.tracks.third_track.click()

```

perform()

For every Field in an Area, “do the right thing” by calling the Field’s perform() method.

Fields that require an argument can either be given sequentially or with keywords.

Parameters

- **args** – Arguments that will sequentially be sent to Fields in this Area.
- **kwargs** – Arguments that will be sent specifically to the Field with a matching name.

Example

Given the following Page Object:

```

>>> from stere.areas import Area
>>> from stere.fields import Button, Input
>>>
>>> class Login():
>>>     def __init__(self):
>>>         self.form = Area(
>>>             username=Input('id', 'app-user'),
>>>             password=Input('id', 'app-pwd'),
>>>             submit=Button('id', 'app-submit')
>>>         )

```

Any of the following styles are valid:

```

>>> def test_login():
>>>     login = Login()
>>>     login.my_area.perform('Sven', 'Hoek')

```

```

>>> def test_login():
>>>     login = Login()
>>>     login.my_area.perform(username='Sven', password='Hoek')

```

```

>>> def test_login():
>>>     login = Login()
>>>     login.my_area.perform('Sven', password='Hoek')

```

workflow()

Sets the current workflow for an Area.

Designed for chaining before a call to perform().

Parameters **value** (*str*) – The name of the workflow to set.

Returns The calling Area

Return type *Area*

Example:

```
>>> my_area.workflow('Foobar').perform()
```

class stere.areas.RepeatingArea

Represents multiple identical Areas on a page.

A root argument is required, which is expected to be a non-unique Field on the page.

A collection of Areas are built from every instance of the root that is found. Every other Field provided in the arguments is populated inside each Area.

In the following example, there's a table with 15 rows. Each row has two cells. The sixth row in the table should have an item with the name "Banana" and a price of "\$7.00"

```
>>> from stere.areas import RepeatingArea
>>> from stere.fields import Root, Link, Text
>>>
>>> class Inventory(Page):
>>>     def __init__(self):
>>>         self.inventory_items = RepeatingArea(
>>>             root=Root('xpath', '//table/tr'),
>>>             name=Link('xpath', './td[1]'),
>>>             price=Text('xpath', './td[2]'),
>>>         )
```

```
>>> inventory = Inventory()
>>> assert 15 == len(inventory.areas)
>>> assert "Banana" == inventory.areas[5].name
>>> assert "$7.00" == inventory.areas[5].price
```

areas

Find all instances of the root, then return a list of Areas: one for each root.

Returns list-like collection of every Area that was found.

Return type *Areas*

Example

```
>>> def test_stuff():
>>>     listings = MyPage().my_repeating_area.areas
>>>     listings[0].my_input.fill('Hello world')
```

area_with()

Searches the RepeatingArea for a single Area where the Field's value matches the expected value and then returns the entire Area object.

Parameters

- **field_name** (*str*) – The name of the Field object.
- **field_value** (*str*) – The value of the Field object.

Returns The Area object that matches the search.

Return type *Area*

Example

```

>>> class Inventory(Page):
>>>     def __init__(self):
>>>         self.items = RepeatingArea(
>>>             root=Root('xpath', '//my_xpath_string'),
>>>             description=Text('xpath', '//my_xpath_string')
>>>         )
>>>
>>> def test_stuff():
>>>     inventory = Inventory()
>>>     found_area = inventory.items.area_with(
>>>         "description", "Bananas")

```

class stere.areas.Areas

Searchable collection of Areas.

Behaves like a list.

containing()

Searches for Areas where the Field's value matches the expected value and then returns an Areas object with all matches.

Parameters

- **field_name** (*str*) – The name of the Field object.
- **field_value** (*str*) – The value of the Field object.

Returns A new Areas object with matching results

Return type *Areas*

Example

```

>>> class Inventory(Page):
>>>     def __init__(self):
>>>         self.items = RepeatingArea(
>>>             root=Root('xpath', '//my_xpath_string'),
>>>             description=Text('xpath', '//my_xpath_string')
>>>         )
>>>
>>> def test_stuff():
>>>     # Ensure 10 items have a price of $9.99
>>>     inventory = Inventory()
>>>     found_areas = inventory.items.areas.containing(
>>>         "price", "$9.99")
>>>     assert 10 == len(found_areas)

```

contain()

Check if a Field in any Area contains a specific value.

Parameters

- **field_name** (*str*) – The name of the Field object.
- **field_value** (*str*) – The value of the Field object.

Returns True if matching value found, else False

Return type *bool*

Example

```
>>> class Inventory(Page):
>>>     def __init__(self):
>>>         self.items = RepeatingArea(
>>>             root=Root('xpath', '//div[@id='inventory']'),
>>>             description=Text('xpath', './td[1]')
>>>         )
>>>
>>> def test_stuff():
>>>     inventory = Inventory()
>>>     assert inventory.items.areas.contain(
>>>         "description", "Bananas")
```

1.6.1 Reusing Areas

Sometimes an identical Area may be present on multiple pages. Areas do not need to be created inside a page object, they can be created outside and then called from inside a page.

```
header = Area(
    ...
)

class Items(Page):
    def __init__(self, *args, **kwargs):
        self.header = header
```

1.6.2 Subclassing Areas

If an Area appears on many pages and requires many custom methods, it may be better to subclass the Area instead of embedding the methods in the Page Object:

```
class Header(Area):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def my_custom_method(self, *args, **kwargs):
        ...

class Main(Page):
    def __init__(self, *args, **kwargs):
        self.header = Header()

class Other(Page):
    def __init__(self, *args, **kwargs):
        self.header = Header()
```

1.7 Repeating

```
class stere.areas.Repeating
```

Parameters

- **root** (`Field`) – A non-unique root to search for.
- **repeater** (`Repeating`) – An object that inherits from `Repeating`.

Represents abstract non-unique collections that repeat, based on a common root.

This can be used to identify anything that not only appears multiple times, but also contains things which appear multiple times.

Repeating are inherently confusing and should only be used if something appears multiple times, contains something else that appears multiple times, and is truly non-unique with no other way to target it.

The last object in a chain of `Repeating` must be a `RepeatingArea`. This is because ultimately, there must be an end to the number of things that repeat.

Example

A page where multiple “project” containers appear, each with a table of items.

```
>>> from stere.areas import Repeating, RepeatingArea
>>> from stere.fields import Root, Link, Text
>>>
>>> projects = Repeating(
>>>     root=Root('css', '.projectContainer'),
>>>     repeater=RepeatingArea(
>>>         root=Root('xpath', '//table/tr'),
>>>         description=Link('xpath', './td[1]'),
>>>         cost=Text('xpath', './td[2]'),
>>>     )
>>> )
>>>
>>> assert 2 == len(projects.children)
>>> first_project = projects.children[0]
>>> assert first_project.areas.contains(
>>>     'description', 'Solar Panels')
>>>
>>> second_project = projects.children[1]
>>> assert second_project.areas.contains(
>>>     'description', 'Self-Driving Cars')
```

new_container()

Must return an object to contain results from `Repeater.children()`

By default a list is returned.

Returns list

children()

Find all instances of the root, then return a collection containing children built from those roots.

The type of collection is determined by the `Repeating.new_container()` method.

Returns list-like collection of every repeater that was found.

1.8 Workflows

When working with an Area that has multiple possible routes, there may be Fields which you do not want the `.perform()` method to call under certain circumstances.

Take the following example Page Object:

```
class AddSomething(Page):
    def __init__(self):
        self.form = Area(
            item_name=Input('id', 'itemName'),
            item_quantity=Input('id', 'itemQty'),
            save=Button('id', 'saveButton'),
            cancel=Button('id', 'cancelButton')
        )
```

Calling `AddSomething().form.perform()` would cause the save button and then the cancel button to be acted on.

In these sorts of cases, Workflows can be used to manage which Fields are called.

```
class AddSomething(Page):
    def __init__(self):
        self.form = Area(
            item_name=Input('id', 'itemName', workflows=["success", "failure"]),
            item_quantity=Input('id', 'itemQty', workflows=["success", "failure"]),
            save=Button('id', 'saveButton', workflows=["success"]),
            cancel=Button('id', 'cancelButton', workflows=["failure"])
        )
```

Calling `AddSomething().form.workflow("success").perform()` will ensure that only Fields with a matching workflow are called.

1.9 Fields Returning Objects

Fields take an optional *returns* argument. This can be any object. When the Field's `perform()` method is called, this object will be returned.

This can be used to return another Page Object.

```
class Navigation(Page):
    def __init__(self):
        self.goto_settings = Button('id', 'settingsLink', returns=NextPage())
```

```
def test_navigation():
    page = Navigation()
    next_page = page.goto_settings.perform()
```

1.9.1 Fields inside an Area

When a Field is inside an Area and has the `returns` argument set, only the object for the last Field in the Area will be returned when `Area.perform()` is called.

```
class Address(Page):
    def __init__(self):
        self.form = Area(
            address=Input('id', 'formAddress'),
            city=Input('id', 'formCity', returns=FooPage()),
            submit=Button('id', 'formsubmit', returns=NextPage()),
        )
```

```
def test_address_form():
    page = Address()
    next_page = page.form.perform()
```

1.10 Best Practices

A highly opinionated guide. Ignore at your own peril.

1.10.1 Favour adding methods to Fields and Areas over Page Objects

If a new method is acting on a specific Field, subclass the Field and add the method there instead of adding the method to the Page Object.

Wrong:

```
class Inventory(Page):
    def __init__(self):
        self.medals = Field('id', 'medals')

    def count_the_medals(self):
        return len(self.medals.find())

def test_you_got_the_medals():
    inventory = Inventory()
    assert 3 == inventory.count_the_medals()
```

Right:

```
class Medals(Field):
    def count(self):
        return len(self.find())

class Inventory(Page):
    def __init__(self):
        self.medals = Medals('id', 'medals')

def test_you_got_the_medals():
    inventory = Inventory()
    assert 3 == inventory.medals.count()
```

Explanation:

Even if a Field or Area initially appears on only one page, subclassing will lead to code that is more easily reused and/or moved.

In this example, `inventory.count_the_medals()` may look easier to read than `inventory.medals.count()`. However, creating methods with long names and specific verbiage makes your Page Objects less predictable and more prone to inconsistency.

1.10.2 Favour page composition over inheritance

When building Page Objects for something with many reused pieces (such as a settings menu) don't build an abstract base Page Object. Build each component separately and call them in Page Objects that reflect the application.

Inheritance:

```
class BaseSettings(Page):
    def __init__(self):
        self.settings_menu = Area(...)

class SpecificSettings(BaseSettings):
    def __init__(self):
        super().__init__()
```

Composition:

```
from .another_module import settings_menu

class SpecificSettings(Page):
    def __init__(self):
        self.menu = settings_menu
```

Explanation:

Doing so maintains the benefits of reusing code, but prevents the creation of Page Objects that don't reflect actual pages in an application.

Creating abstract Page Objects to inherit from can make it confusing as to what Fields are available on a page.

1.10.3 Naming Fields

Describing the Field VS Describing the Field's Action

When naming a field instance, the choice is usually between a description of the field or a description of what the field does:

Describing the Field:

```
class Navigation(Page):
    def __init__(self):
        self.settings_button = Button('id', 'settingsLink')
```

Describing the Action:

```
class Navigation(Page):
    def __init__(self):
        self.goto_settings = Button('id', 'settingsLink')
```

At the outset, either option can seem appropriate. Consider the usage inside a test:


```
nav_page = Navigation()
nav_page.settings_button.click()
```

VS

```
nav_page = Navigation()
nav_page.goto_settings.click()
```

However, consider what happens when a Field returns a Page:

```
class Navigation(Page):
    def __init__(self):
        self.settings_page = Button('id', 'settingsLink', returns=NextPage())
```

```
class Navigation(Page):
    def __init__(self):
        self.goto_settings = Button('id', 'settingsLink', returns=NextPage())
```

```
nav_page = Navigation()
settings_page = nav_page.settings_button.perform()
```

```
nav_page = Navigation()
settings_page = nav_page.goto_settings.perform()
```

Or, calling the perform method implicitly:

```
nav_page = Navigation()
settings_page = nav_page.settings_button()
```

```
nav_page = Navigation()
settings_page = nav_page.goto_settings()
```

In the end, naming Fields will depend on what they do and how your tests use them.

1.10.4 Single blank line when changing page object

Wrong:

```
def test_the_widgets():
    knicknacks = Knicknacks()
    knicknacks.menu.gadgets.click()
    knicknacks.gadgets.click()
    gadgets = Gadgets()
    gadgets.navigate()

    gadgets.add_widgets.click()
    gadgets.add_sprocket.click()
```

Right:

```
def test_the_widgets():
    knicknacks = Knicknacks()
    knicknacks.menu.gadgets.click()
    knicknacks.gadgets.click()
```

(continues on next page)

(continued from previous page)

```
gadgets = Gadgets()
gadgets.navigate()
gadgets.add_widgets.click()
gadgets.add_sprocket.click()
```

Explanation:

Changing pages usually indicates a navigation action. Using a consistent line break style visually helps to indicate the steps of a test.

CHAPTER 2

Indices and tables

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

A

after() (*stere.fields.Field* method), 4
 Area (class in *stere.areas*), 12
 area_with() (*stere.areas.RepeatingArea* method), 14
 Areas (class in *stere.areas*), 15
 areas (*stere.areas.RepeatingArea* attribute), 14

B

before() (*stere.fields.Field* method), 4

C

children() (*stere.areas.Repeating* method), 17
 click() (*stere.fields.appium.button.stere.fields.Button.Button* method), 11
 click() (*stere.fields.stere.fields.Button.Button* method), 8
 click() (*stere.fields.stere.fields.Link.Link* method), 10
 contain() (*stere.areas.Areas* method), 15
 containing() (*stere.areas.Areas* method), 15

F

Field (class in *stere.fields*), 3
 fill() (*stere.fields.stere.fields.Input.Input* method), 9

I

includes() (*stere.fields.Field* method), 3
 is_not_present() (*stere.strategy.splinter.SplinterBase* method), 8
 is_not_visible() (*stere.strategy.splinter.SplinterBase* method), 8
 is_present() (*stere.strategy.splinter.SplinterBase* method), 7
 is_visible() (*stere.strategy.splinter.SplinterBase* method), 8

N

navigate() (*stere.Page* method), 2
 new_container() (*stere.areas.Repeating* method), 17

O

opposite() (*stere.fields.stere.fields.Checkbox.Checkbox* method), 9
 options() (*stere.fields.stere.fields.Dropdown.Dropdown* method), 9

P

Page (class in *stere*), 2
 perform() (*stere.areas.Area* method), 13

R

Repeating (class in *stere.areas*), 16
 RepeatingArea (class in *stere.areas*), 14
 Root (class in *stere.fields*), 5

S

select() (*stere.fields.stere.fields.Dropdown.Dropdown* method), 9
 send_keys() (*stere.fields.appium.input.stere.fields.Input.Input* method), 12
 set_to() (*stere.fields.stere.fields.Checkbox.Checkbox* method), 8
 stere.fields.Button (built-in class), 8, 11
 stere.fields.Checkbox (built-in class), 8
 stere.fields.Dropdown (built-in class), 9
 stere.fields.Input (built-in class), 9, 11
 stere.fields.Link (built-in class), 10
 stere_performer() (*stere.fields.decorators* method), 6

T

Text (class in *stere.fields*), 5
 toggle() (*stere.fields.stere.fields.Checkbox.Checkbox* method), 9

U

use_after() (*stere.fields.decorators* method), 7
 use_before() (*stere.fields.decorators* method), 7

V

`value_contains()` (*stere.fields.Field method*), [4](#)

`value_equals()` (*stere.fields.Field method*), [4](#)

W

`workflow()` (*stere.areas.Area method*), [13](#)