ScreenPy Documentation

Release 0.2.0

Perry Goy

Feb 18, 2020

Contents:

ScreenPy provides a solid, SOLID base for writing maintainable test suites following the Screenplay Pattern, popularized by Antony Marcano. It also provides nice test logging through Allure and support for BDD-style natural language test case writing.

CHAPTER 1

Installation

To install ScreenPy, run the following command, preferably in a virtual environment:

```
pip3 install screenpy
```

This will also install the `screenpy-quickstart` script and the following dependencies:

1. Selenium
2. PyHamcrest
3. Allure's Pytest plugin
4. Pytest

CHAPTER 2

Quickstart

2.1 screenpy-quickstart

To quickly set up a Screenplay Pattern test suite using ScreenPy, `cd` to the folder you will use for your suite and run this command:

```
screenpy-quickstart
```

This will set up `user_interface`, `questions`, `tasks`, and `features` directories and fill them with a simple test. For an explanation of each of these directories, see the *File Hierarchy* page!

CHAPTER 3

File Hierarchy

The key to a good Screenplay Pattern suite is understanding how the files all fit together. The hierarchy described herein is one example of how the files can be organized and named. If your team feels strongly that there are better conventions to follow, renaming the files will not break any of ScreenPy's functionality.

Here is an example hierarchy:

- suite_root
  - features # this is where the actual test files will live
    - feature1.py
    - ...
  - questions # questions your actors will ask about the site
    - question1.py
    - ...
  - tasks # groups of actions your actors can perform, with descriptive names
    - task1.py
    - ...
  - user_interface # files containing locators and/or URLs for each page
    - page1.py
    - ...
  - requirements.txt # where you list screenpy!

3.1 Features

The feature films! The story arcs! The whole point of the suite! These are the features of your application that you are testing; this is where all the actual test files go.

3.2 Questions

Things your actor asks about the application, to perform a thrilling turnabout (test fail) or a cathartic confirmation (test pass) upon finding the answer. These files are where you will access elements on the page to figure out if your test has passed.

For more information, see the *Questions* page!

3.3 Tasks

Tasks are descriptive ways to group one or more actions that your actors will do. A common task is a `Login` task, which will contain the actions necessary to log in. There may be many tasks your actors will need to do in your suite.

For more information, see the *Tasks* section!

3.4 User Interface

These files collect all the locators (built using the *`Target`* class) and maybe URLs for the pages of your application. These probably will not be super interesting files; they're kind of like the blocking notes for the screenplay.

CHAPTER 4

Included in ScreenPy

ScreenPy comes with a lot of the base tools you will need to get started, which should cover the most common use cases. You'll be set up in time to make curtain call!

4.1 Actors

Actors are the do-ers in the screenplay. Actors set the scene, perform their hearts out, and then make dramatic assertions that will either see a happy ending or a tragic failure.

More seriously, the actors represent the users of your application, doing the things you'd expect them to do on it (or things you might not expect them to do). Screenplay Pattern focuses entirely on what your users hope to do on your site, so your test cases will focus on what the actors do, which includes gaining *Abilities*, performing *Actions*, and asking *Questions*.

4.1.1 Using Actors

To instantiate a new actor, just give it a name:

```
from screenpy.actor import Actor, AnActor

Perry = AnActor.named("Perry")
```

Without any abilities, your actor will be woefully unprepared to begin their performance. To give your actor an ability, you can do something like:

```
from selenium.webdriver import Firefox
from screenpy.abilities import BrowseTheWeb

Perry.can(BrowseTheWeb.using(Firefox()))

# For convenience, you can also do the same like this
Perry = AnActor.named("Perry").who_can(BrowseTheWeb.using(Firefox()))
```

Now, Perry is able to attempt any actions that require the ability to BrowseTheWeb. Attempting actions looks like this:

```
from screenpy import Target
from screenpy.actions import Click

EXAMPLE_LINK = Target.the("example link").located_by("//a")
Perry.attempts_to(Click.the(EXAMPLE_LINK))
```

You'll notice we had to make a quick *Target* there. We'll get to *Targets* later, but a quick summary is that they're how you tell the actors where to perform the action.

In the above example, the action knows what ability it requires, and it will ask the actor to find its matching ability to perform the action. If the actor does not have that ability, the actor will raise an *UnableToPerformError*.

Now that our actor has performed an action, they are ready to perform a test. Tests are performed with *Questions*, like so:

```
from screenpy.questions import Text
from screenpy.resolutions import ReadsExactly

THE_WELCOME_MESSAGE = Target.the("welcome_message").located_by("span.welcome")
Perry.should_see_the((Text.of(THE_WELCOME_MESSAGE), ReadsExactly("Welcome!"))
```

That's the whole flow! Your actor is now ready to exit:

```
Perry.exits_stage_right()
```

In summary, actors:

- Are created by naming them using the *named()* class method.
- Are granted *Abilities* using the *who_can()* or *can()* class methods.
- Perform *Actions* using their granted *Abilities*.
- Ask *Questions* about the state of the application under test.
- Exit gracefully, with a flourish.

4.1.2 Actor Class

class screenpy.actor.**Actor**(*name: str*)

Represents an actor, holding their name and abilities. Actors are the performers of your screenplay, they represent your users as they go about their business on your product.

An actor is meant to be instantiated using its static *named()* method. A typical invocation might look like:

Perry = Actor.named("Perry")

This will create the actor, ready to take on their first role.

ability_to(*ability: Any*) → Any

Syntactic sugar for *uses_ability_to()*.

attempts_to(**actions*) → None

Performs a list of actions, one after the other.

Parameters **actions** – the list of actions to perform.

can(**abilities*) → screenpy.actor.Actor

Syntactic sugar for *who_can()*.

exit() → None
The actor forgets all of their abilities, ready to assume a new role when their next cue calls them.

exit_stage_left() → None
Syntactic sugar for *exit()*.

exit_stage_right() → None
Syntactic sugar for *exit()*.

static named(*name: str*) → screenpy.actor.Actor
Names this actor, logs their entrance, and returns the instance.

Parameters **name** – the name of this new Actor.

Returns *Actor*

perform(*action: Any*) → None
Performs the given action.

Parameters **action** – the *Actions* to perform.

should_see(**tests*) → None
Syntactic sugar for *should_see_the()*.

should_see_that(**tests*) → None
Syntactic sugar for *should_see_the()*.

should_see_the(**tests*) → None
Asks a series of questions, asserting that the expected answer resolves.

Parameters **tests** – tuples of a *Questions* and a *Resolutions*.

Raises AssertionError – If the question's actual answer does not match the expected answer from the *Resolutions*.

uses_ability_to(*ability: Any*) → Any
Finds the ability referenced and returns it, if the actor is able to do it.

Parameters **ability** – the ability to retrieve.

Returns The requested ability.

Raises |UnableToPerformError| – the actor doesn't possess the ability.

was_able_to(**actions*) → None
Syntactic sugar for *attempts_to()*.

who_can(**abilities*) → screenpy.actor.Actor
Adds an ability to this actor.

Parameters **abilities** – The abilities this actor can do.

Returns *Actor*

4.2 Abilities

Abilities allow your *Actor* to **do** things. Actors will leverage their abilities to perform actions that require those abilities.

4.2.1 Using Abilities

To give an actor an ability, pass it in using the actor's *who_can()* or *can()* methods:

```
from screenpy import Actor, AnActor
from screenpy.abilities import BrowseTheWeb


# Add abilities on instantiation
Perry = AnActor.named("Perry").who_can(BrowseTheWeb.using_firefox())

# Or add abilities later
Perry = AnActor.named("Perry")
Perry.can(BrowseTheWeb.using_safari())
```

Granting an ability to an actor allows them to perform any *Actions* or ask any *Questions* that require that ability. If an action or a question require an ability that the actor does not have, the actor will raise an *UnableToPerformError*.

4.2.2 Writing New Abilities

There may be other abilities your actors need to possess in order to test your application. You are encouraged to write your own! The only prescribed method for an ability is the `forget` method, which will complete any cleanup required. For an example, see the *forget()* method of the BrowseTheWeb ability. A base class for Abilities is provided for convenience: `screenpy.abilities.base_ability.BaseAbility`

4.2.3 Included Abilities

BrowseTheWeb

class screenpy.abilities.browse_the_web.**BrowseTheWeb**(*browser: selenium.webdriver.remote.webdriver.WebDriver*)

The ability to browse the web with a web browser. This ability is meant to be instantiated with its *using()* static method, which takes in the WebDriver to use, or one of its other "using" methods. A typical invocation looks like:

BrowseTheWeb.using(selenium.webdriver.Firefox())

BrowseTheWeb.using_firefox()

This will create the ability that can be passed in to an actor's *who_can()* method.

find(*locator: Union[Target, Tuple[selenium.webdriver.common.by.By, str]]*) → selenium.webdriver.remote.webelement.WebElement

Syntactic sugar for *to_find()*.

find_all(*target: Union[Target, Tuple[selenium.webdriver.common.by.By, str]]*) → selenium.webdriver.remote.webelement.WebElement

Syntactic sugar for *to_find_all()*.

forget() → None

Asks the actor to forget how to BrowseTheWeb. This quits the connected browser.

An actor who is exiting will forget all their abilities.

to_find(*target: Union[Target, Tuple[selenium.webdriver.common.by.By, str]]*) → selenium.webdriver.remote.webelement.WebElement

Locates a single element on the page using the given locator.

Parameters **target** – the *Target* or tuple describing the element.

Returns WebElement

Raises |BrowsingError|

to_find_all(*target: Union[Target, Tuple[selenium.webdriver.common.by.By, str]]*) → List[selenium.webdriver.remote.webelement.WebElement]
Locates many elements on the page using the given locator.

Parameters **target** – the *Target* or tuple describing the elements.

Returns List[WebElement]

to_get(*url: str*) → screenpy.abilities.browse_the_web.BrowseTheWeb
Uses the connected browser to visit the specified URL.

This action supports using the BASE_URL environment variable to set a base URL. If you set BASE_URL, the url passed in to this function will be appended to the end of it. For example, if you have *BASE_URL=http://localhost*, then to_get("/home") will send your browser to "http://localhost/home".

If BASE_URL isn't set, then the passed-in url is assumed to be a fully qualified URL.

Parameters **url** – the URL to visit.

Returns *BrowseTheWeb*

to_switch_to(*target: Target*) → None
Switches the browser context to the target.

Parameters **target** – the *Target* or tuple describing the element to switch to.

to_switch_to_alert() → selenium.webdriver.common.alert.Alert
Switches to an alert and returns it.

Returns Alert

Raises |BrowsingError| – no alert was present to switch to.

to_switch_to_default() → None
Switches the browser context back to the default frame.

to_visit(*url: str*) → screenpy.abilities.browse_the_web.BrowseTheWeb
Syntactic sugar for *to_get()*.

to_wait_for(*target: Union[Target, Tuple[selenium.webdriver.common.by.By, str]], timeout: int = 20, cond: Callable = <class 'selenium.webdriver.support.expected_conditions.visibility_of_element_located'>*)
Waits for the element to fulfill the given condition.

Parameters

- **target** – the tuple or *Target* describing the element.
- **timeout** – how many seconds to wait before raising a TimeoutException. Default is 20.
- **cond** – the condition to wait for. Default is visibility_of_element_located.

Raises |BrowsingError| – the target did not satisfy the condition in time.

static using(*browser: selenium.webdriver.remote.webdriver.WebDriver*) → screenpy.abilities.browse_the_web.BrowseTheWeb
Specifies the driver to use to browse the web. This can be any WebDriver instance, even a remote one.

Parameters **browser** – the webdriver instance to use.

Returns *BrowseTheWeb*

static using_android() → screenpy.abilities.browse_the_web.BrowseTheWeb
Creates an uses a default Remote driver instance to connect to a running Appium server and open Chrome on Android. Use this if you don't need to set anything up for your test browser.

Note that Appium requires non-trivial setup to be able to connect to Android emulators. See the Appium documentation to get started: http://appium.io/docs/en/writing-running-appium/running-tests/

Environment Variables:

APPIUM_HUB_URL: the URL to look for the Appium server. Default is "http://localhost:4723/wd/hub"

ANDROID_DEVICE_VERSION: the version of the device to put in the desired capabilities. Default is "10.0"

ANDROID_DEVICE_NAME: the device name to request in the desired capabilities. Default is "Android Emulator"

Returns *BrowseTheWeb*

static using_chrome() → screenpy.abilities.browse_the_web.BrowseTheWeb
Creates and uses a default Chrome Selenium webdriver instance. Use this if you don't need to set anything up for your test browser.

Returns *BrowseTheWeb*

static using_firefox() → screenpy.abilities.browse_the_web.BrowseTheWeb
Creates and uses a default Firefox Selenium webdriver instance. Use this if you don't need to set anything up for your test browser.

Returns *BrowseTheWeb*

static using_ios() → screenpy.abilities.browse_the_web.BrowseTheWeb
Creates an uses a default Remote driver instance to connect to a running Appium server and open Safari on iOS. Use this if you don't need to set anything up for your test browser.

Note that Appium requires non-trivial setup to be able to connect to iPhone simulators. See the Appium documentation to get started: http://appium.io/docs/en/writing-running-appium/running-tests/

Environment Variables:

APPIUM_HUB_URL: the URL to look for the Appium server. Default is "http://localhost:4723/wd/hub"

IOS_DEVICE_VERSION: the version of the device to put in the desired capabilities. Default is "13.1"

IOS_DEVICE_NAME: the device name to request in the desired capabilities. Default is "iPhone Simulator"

Returns *BrowseTheWeb*

static using_safari() → screenpy.abilities.browse_the_web.BrowseTheWeb
Creates and uses a default Safari Selenium webdriver instance. Use this if you don't need to set anything up for your test browser.

Returns *BrowseTheWeb*

wait_for(*locator: Union[Target, Tuple[selenium.webdriver.common.by.By, str]], timeout: int = 20, cond: Callable = <class 'selenium.webdriver.support.expected_conditions.visibility_of_element_located'>*)
Syntactic sugar for *to_wait_for()*.

AuthenticateWith2FA

class screenpy.abilities.authenticate_with_2fa.**AuthenticateWith2FA**(*otp: pyotp.totp.TOTP*)

The ability to retrieve a one-time password from a two-factor authenticator. This ability is meant to be instantiated with its *using_secret()* method, which will take in the 2FA secret, or its *using()* static method, which takes in an instantiated PyOTP instance. A typical invocation looks like:

AuthenticateWith2FA.using_secret("KEEPITSECRETKEEPITSAFE")

AuthenticateWith2FA.using(pyotp_instance)

This will create the ability that can be passed in to an actor's *who_can()* method.

forget() → None
Cleans up the pyotp instance stored in this ability.

to_get_token() → str
Gets the current two-factor token to use as a one-time password.

Returns str

static using(*otp: pyotp.totp.TOTP*) → screenpy.abilities.authenticate_with_2fa.AuthenticateWith2FA
Uses an already-created TOTP instance to provide tokens.

Parameters **otp** (*pyotp.TOTP*) – an instance of a TOTP object.

Returns *AuthenticateWith2FA*

static using_secret(*secret: str*) → screenpy.abilities.authenticate_with_2fa.AuthenticateWith2FA
Creates a TOTP instance with the given secret.

Parameters **secret** – the secret given by the 2FA service. You may need to decode a QR code to get this secret.

Returns *AuthenticateWith2FA*

4.3 Targets

Targets are a way to encapsulate a human-readable string along with a CSS selector or xpath locator.

To instantiate a target, you might do something like this:

```
from screenpy import Target

EXAMPLE_ELEMENT1 = Target.the("first example element").located_by("//div")
EXAMPLE_ELEMENT2 = Target.the("second example element").located_by("span.example")
```

Let's break that down a little bit.

The class method *the()* expects a human-readable string to give the element a log-friendly name. That same class method returns the newly instantiated *Target* object, ready to have its *located_by()* method called.

The *located_by()* method takes in the actual locator, which can either be XPath or CSS Selector.

Targets are expected to be defined in your user_interface files, and can then be used in your *Actions*, your *Questions*, and your *Tasks*.

4.3.1 Target Class

class screenpy.target.**Target**(*desc: str*)

A class to contain information about an element. This class stores a nice human-readable string describing an element along with either an XPath or a CSS selector string. It is intended to be instantiated by calling its static *the()* method. A typical invocation might look like:

Target.the("header search bar").located_by("div.searchbar")

It can then be used in Questions, Actions or Tasks to access that element.

all_found_by(*the_actor: screenpy.actor.Actor*) → List[selenium.webdriver.remote.webelement.WebElement]

Gets a list of WebElement objects described by the stored locator.

Parameters **the_actor** (Actor) – The *Actor* who should look for these elements.

Returns list(WebElement)

found_by(*the_actor: screenpy.actor.Actor*) → selenium.webdriver.remote.webelement.WebElement

Gets the WebElement object representing the targeted element.

Parameters **the_actor** (Actor) – The *Actor* who should look for this element.

Returns WebElement

get_locator() → Tuple[selenium.webdriver.common.by.By, str]

Returns the stored locator as a (By, str) tuple.

Returns Tuple(By, str)

Raises |TargetingError| – if no locator was supplied to the target.

located(*locator: Tuple[selenium.webdriver.common.by.By, str]*) → screenpy.target.Target

Supplies an instantiated target with a locator. This locator is a tuple of the By strategy to use and the identifying string, e.g.

Target.the("signout link").located((By.LINK_TEXT, "Sign Out"))

Parameters **locator** – the (By, str) tuple to use to find the element.

Returns *Target*

located_by(*locator: str*) → screenpy.target.Target

Supplies an instantiated Target with a locator string, which is either a CSS selector or an XPATH string. The strategy will be determined before it is stored.

Parameters **locator** – the string to use as a locator for the element. Can be a CSS selector or an xpath string.

Returns *Target*

static the(*desc: str*) → screenpy.target.Target

Creates a Target with a description. This method call should be followed up with a call to *located_by()*.

Parameters **desc** (*str*) – The human-readable description for the targeted element. Beginning with a lower-case letter makes the allure test logs look the nicest.

Returns *Target*

4.4 Actions

Actions are the things that an *Actor* can do, using their *Abilities*.

4.4.1 Using Actions

Actions can be used pretty much anywhere. They will typically be used to create *Tasks* or move around in your Features. Here is an example of using the *Click* action:

```
from screenpy.actions import Click

from ..user_interface.homepage import LOGIN_LINK


Perry.attempts_to(Click.on_the(LOGIN_LINK))
```

Actors will always only *attempt* to perform an action. They may not actually have the correct *Abilities*, after all. If an actor is unable to perform an action or task, they will raise an *UnableToPerformError*.

4.4.2 Writing New Actions

Occasionally, you might find that the base actions don't quite cover a unique use case you have for your test suite. Since Screenplay Pattern is built to be extensible, it is easy and encouraged to create your own custom actions to achieve what you need! The only requirement for creating more actions is that they have a `perform_as` method defined which takes in the actor who will perform the action.

A base class for Actions is provided to ensure the required methods are defined: `screenpy.actions.base_action.BaseAction`

Let's take a look at what an extremely contrived custom action, `ChecksTheSpelling`, might look like:

```
# actions/checks_the_spelling.py
from screenpy.actions import BaseAction


class ChecksTheSpelling(BaseAction):
    @staticmethod
    def of_words_in_the(locator):
        return ChecksSpelling(locator)

    def perform_as(self, the_actor):
        the_actor.uses_ability_to(CheckSpelling).to_check()

    def __init__(self, locator):
        self.locator = locator
```

ScreenPy attempts to follow a convention of putting all the static methods first, then the `perform_as` function, and leaving the dunder methods at the bottom. This way the most important methods are first for someone perusing your code.

4.4.3 Tasks

Sometimes, your actors might repeat the same series of actions several times. A grouping of common actions can be abstracted into a Task in your *Tasks*.

A common task for Screenplay Pattern suites is logging in to your application under test. This login task might look something like this:

```
# tasks/login.py
import os

from screenpy import Actor
from screenpy.actions import BaseAction, Click, Enter

from ..user_interface.homepage import (
    SIGN_ON_LINK,
    THE_USERNAME_FIELD,
    THE_PASSWORD_FIELD,
    LOGIN_BUTTON,
)


class LoginSuccessfully(BaseAction):
    """
    Log in to the application successfully.
    """

    @staticmethod
    def using_credentials(username: str, password: str) -> "LoginSuccessfully":
        """
        Supply the credentials for the account.

        Args:
            username: the username to use.
            password: the password to use.
        """
        return LoginSuccessfully(username, password)

    def perform_as(self, the_actor: Actor) -> None:
        """
        Asks the actor to log in to the application.

        Args:
            the_actor: the actor who will perform this task.

        Raises:
            UnableToPerformError: the actor does not have the ability to
                BrowseTheWeb.
        """
        the_actor.attempts_to(
            Click.on(SIGN_ON_LINK),
            Wait.for_the(THE_USERNAME_FIELD).to_appear(),
            Enter.the_text(self.username).into(THE_USERNAME_FIELD),
            Enter.the_text(self.password).into(THE_PASSWORD_FIELD),
            Click.on_the(LOGIN_BUTTON)
        )

    def __init__(self, username: str, password: str):
        self.username = username
        self.password = password
```

And there you have it! Now all you have to do is ask your actor to attempt to `LoginSuccessfully`, and you've got the same set of actions everywhere.

Note that tasks, just like actions, are required to have a `perform_as` method defined. You can use the `BaseAction` class for tasks as well.

4.4.4 Provided Actions

Open

class screenpy.actions.open.**Open**(*location: Union[str, object]*)

A very important action; opens the browser! An Open action is expected to be instantiated via its static *browser_on()* method. A typical invocation might look like:

Open.browser_on(the_homepage_url)

Open.browser_on(HomepageObject)

If you pass in an object, make sure the object has a *url* property that can be referenced by this action.

It can then be passed along to the *Actor* to perform the action.

static browser_on(*location: Union[str, object]*) → screenpy.actions.open.Open

Creates a new Open action which holds its destined location.

Parameters **location** – The URL to open when this action is performed, or an object containing a *url* property that holds the URL to open when this action is performed.

Returns *Open*

perform_as(*the_actor: screenpy.actor.Actor*) → None

Asks the supplied actor to perform this Open action, using their ability to browse the web.

Parameters **the_actor** – The *Actor* who will perform the action.

Raises |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

static their_browser_on(*location: Union[str, object]*) → screenpy.actions.open.Open

Syntactic sugar for *browser_on()*.

Click

class screenpy.actions.click.**Click**(*target: screenpy.target.Target*)

Clicks on an element! A Click action is expected to be instantiated via its static *on()* or *on_the()* methods. A typical invocation might look like:

Click.on_the(PROFILE_LINK)

It can then be passed along to the *Actor* to perform the action.

static on(*target: screenpy.target.Target*) → screenpy.actions.click.Click

Syntactic sugar for *on_the()*.

static on_the(*target: screenpy.target.Target*) → screenpy.actions.click.Click

Creates a new Click action with its crosshairs aimed at the provided target.

Parameters **target** – The *Target* describing the element to click.

Returns *Click*

perform_as(*the_actor: screenpy.actor.Actor*) → None

Asks the actor to find the element described by the stored target, and then clicks it. May wait for another target to appear, if *then_wait_for()* had been called.

Parameters **the_actor** – the *Actor* who will perform the action.

Raises

- |DeliveryError| – an exception was raised by Selenium.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

then_wait_for(*target: screenpy.target.Target*) → screenpy.actions.click.Click
Syntactic sugar for *then_wait_for_the()*.

then_wait_for_the(*target: screenpy.target.Target*) → screenpy.actions.click.Click
Supplies a target to wait for after performing the click.

This method has been deprecated as of version 1.0.0. Please use the included *Wait* action instead. This method will be removed in version 2.0.0.

Parameters **target** – The *Target* describing the element to wait for after performing the click.

Returns *Click*

Clear

class screenpy.actions.clear.**Clear**(*target: screenpy.target.Target*)
Clears the text from an input field. A Clear action is expected to be instantiated by its static *the_text_from()* method. A typical invocation might look like:

Clear.the_text_from(COMMENT_FIELD)

It can then be passed along to the *Actor* to perform the action.

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to performs the Clear action, clearing the text from the targeted input field using their ability to browse the web.

Parameters **the_actor** – The *Actor* who will perform this action.

Raises |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

static the_text_from(*target: screenpy.target.Target*) → screenpy.actions.clear.Clear
Syntactic sugar for *the_text_from_the()*.

static the_text_from_the(*target: screenpy.target.Target*) → screenpy.actions.clear.Clear
Creates a new Clear action with the provided text.

Parameters **target** – the *Target* from which to clear the text.

Returns *Clear*

Enter

class screenpy.actions.enter.**Enter**(*text: str*, *mask: bool = False*)
Enters text into an input field. An Enter action is expected to be instantiated by its static *the_text()* method. A typical invocation might look like:

Enter.the_text("Hello world!").into(COMMENT_FIELD)

It can then be passed along to the *Actor* to perform the action.

into(*target: screenpy.target.Target*) → screenpy.actions.enter.Enter
Supplies the target to enter the text into. This is most likely an input field.

Parameters **target** – The *Target* describing the input field.

Returns *Enter*

into_the(*target: screenpy.target.Target*) → screenpy.actions.enter.Enter
Syntactic sugar for *into()*

on(*target: screenpy.target.Target*) → screenpy.actions.enter.Enter
Syntactic sugar for *into()*

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to perform the Enter action, entering the text into the targeted input field using their ability to browse the web.

If this Enter object's *then_hit()* method was called, it will also hit the supplied keys. Finally, if the *then_wait_for()* method was called, it will wait for the supplied target to appear.

Parameters **the_actor** – the *Actor* who will perform this action.

Raises

- |DeliveryError| – an exception was raised by Selenium.
- |UnableToActError| – no target was supplied.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

static the_keys(*text: str*) → screenpy.actions.enter.Enter
Syntactic sugar for *the_text()*.

static the_password(*text: str*) → screenpy.actions.enter.Enter
Syntactic sugar for *the_secret()*.

static the_secret(*text: str*) → screenpy.actions.enter.Enter
Creates a new Enter action with the provided text, but will mask the text for logging. The text will appear as "[CENSORED]" in the report. It is expected that the next call will be to the instantiated Enter object's *into()* method.

Parameters **text** – the text to enter into the target, but it's a secret.

Returns *Enter*

static the_text(*text: str*) → screenpy.actions.enter.Enter
Creates a new Enter action with the provided text. It is expected that the next call will be to the instantiated Enter object's *into()* method.

Parameters **text** – the text to enter into the target.

Returns *Enter*

then_hit(**keys*) → screenpy.actions.enter.Enter
Supplies additional keys to hit after entering the text, for example if the keyboard ENTER key should be pressed.

Parameters **keys** – the keys to hit afterwards. These are probably the constants from Selenium's Keys, but they can be strings if you know the codes.

Returns *Enter*

then_press(**keys*) → screenpy.actions.enter.Enter
Syntactic sugar for *then_hit()*.

then_wait_for(*target: screenpy.target.Target*) → screenpy.actions.enter.Enter
Supplies the target to wait for after entering text (and hitting any additional keys, if this object's *then_hit()* method was called).

This method has been deprecated as of version 1.0.0. Please use the included *Wait* action instead. This method will be removed in version 2.0.0.

Parameters **target** – the *Target* to wait for after entering text.

Returns *Enter*

Enter2FAToken

class screenpy.actions.enter_2fa_token.**Enter2FAToken**(*target: screenpy.target.Target*)

Enters the current two-factor authentication token into an input field. An Enter2FAToken action is expected to be instantiated by its static *into_the()* method. A typical invocation might look like:

Enter2FAToken.into_the(2FA_INPUT_FIELD)

It can then be passed along to the *Actor* to perform the action.

static into(*target: screenpy.target.Target*) → screenpy.actions.enter_2fa_token.Enter2FAToken

Syntactic sugar for *into_the()*

static into_the(*target: screenpy.target.Target*) → screenpy.actions.enter_2fa_token.Enter2FAToken

Provide the input field into which to enter the 2FA token.

Parameters **target** – the *Target* describing the input field.

Returns *Enter2FAToken*

perform_as(*the_actor: screenpy.actor.Actor*) → None

Asks the actor to perform the Enter2FAToken action, which will get the current token using the actor's AuthenticateWith2FA ability.

Parameters **the_actor** – the *Actor* who will perform this action.

Raises |UnableToPerformError| – if the actor does not have the abilities to *AuthenticateWith2FA* and *BrowseTheWeb*.

Select

class screenpy.actions.select.**Select**

Selects an option from a dropdown menu. This is an entry point that will create the correct specific Select action that will need to be used, depending on how the option needs to be selected. Some examples of invocations:

Select.the_option_named("January").from_the(MONTH_DROPDOWN)

Select.the_option_at_index(0).from_the(MONTH_DROPDOWN)

Select.the_option_with_value("jan").from_the(MONTH_DROPDOWN)

It can then be passed along to the *Actor* to perform the action.

static the_option_at_index(*index: Union[int, str]*) → screenpy.actions.select.SelectByIndex

Instantiate a SelectByIndex class which will select the option at the specified index. This index is 0-based.

Parameters **index** – the index (0-based) of the option to select.

Returns *SelectByIndex*

static the_option_named(*text: str*) → screenpy.actions.select.SelectByText

Instantiate a SelectByText class which will select the option with the given text.

Parameters **text** – the text of the option to select.

Returns *SelectByText*

static the_option_with_value(*value: str*) → screenpy.actions.select.SelectByValue
Instantiate a SelectByText class which will select the option with the given text.

Parameters **value** – the value of the option to select.

Returns *SelectByText*

class screenpy.actions.select.**SelectByText**(*text: str*, *target: Optional[screenpy.target.Target] = None*)
A specialized Select action that chooses the option by text. This class is meant to be accessed via the Select action's static *the_option_named()* method. A typical invocation might look like:

Select.the_option_named("January").from_the(MONTH_DROPDOWN)

It can then be passed along to the *Actor* to perform the action.

from_(*target: screenpy.target.Target*) → screenpy.actions.select.SelectByText
Syntactic sugar for *from_the()*.

from_the(*target: screenpy.target.Target*) → screenpy.actions.select.SelectByText
Provides the target to select the option from.

Parameters **target** – the *Target* describing the dropdown or multi-select element to select the option from.

Returns *SelectByText*

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to attempt to find the dropdown element described by the stored target, then performs the select action.

Parameters **the_actor** – The *Actor* who will perform the action.

Raises

- |DeliveryError| – an exception was raised by Selenium.
- |UnableToActError| – no target was supplied.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

class screenpy.actions.select.**SelectByIndex**(*index: Union[int, str]*, *target: Optional[screenpy.target.Target] = None*)
A specialized *Select* action that chooses the option by its index. This class is meant to be accessed via the Select action's static *the_option_at_index()* method. A typical invocation might look like:

Select.the_option_at_index(0).from_the(MONTH_DROPDOWN)

It can then be passed along to the *Actor* to perform the action.

from_(*target: screenpy.target.Target*) → screenpy.actions.select.SelectByIndex
Syntactic sugar for *from_the()*.

from_the(*target: screenpy.target.Target*) → screenpy.actions.select.SelectByIndex
Provides the target to select the option from.

Parameters **target** – The *Target* describing the dropdown or multi-select element to select the option from.

Returns *SelectByIndex*

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to attempt to find the dropdown element described by the stored target, then performs the select action.

Parameters **the_actor** – The *Actor* who will perform the action.

Raises

- |DeliveryError| – an exception was raised by Selenium.
- |UnableToActError| – no target was supplied.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

class screenpy.actions.select.**SelectByValue**(*value: Union[int, str], target: Optional[screenpy.target.Target] = None*)

A specialized Select action that chooses the option by its value. This class is meant to be accessed via the Select action's static *the_option_with_value()* method. A typical invocation might look like:

Select.the_option_with_value("jan").from_the(MONTH_DROPDOWN)

It can then be passed along to the *Actor* to perform the action.

from_(*target: screenpy.target.Target*) → screenpy.actions.select.SelectByValue
Syntactic sugar for *from_the()*.

from_the(*target: screenpy.target.Target*) → screenpy.actions.select.SelectByValue
Provides the target to select the option from.

Parameters **target** – The *Target* describing the dropdown or multi-select element to select the option from.

Returns *SelectByValue*

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to attempt to find the dropdown element described by the stored target, then performs the select action.

Parameters **the_actor** – The *Actor* who will perform the action.

Raises

- |DeliveryError| – an exception was raised by Selenium.
- |UnableToActError| – no target was supplied.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

AcceptAlert

class screenpy.actions.accept_alert.**AcceptAlert**
Accepts an alert. An AcceptAlert action is expected to be instantiated as it is, no static methods for this one. The only invocation looks like:

AcceptAlert()

It can then be passed along to the *Actor* to perform the action.

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to perform the AcceptAlert action.

Parameters **the_actor** – The *Actor* who will perform this action.

Raises

- |BrowsingError| – no alert was present.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

DismissAlert

class screenpy.actions.dismiss_alert.**DismissAlert**
Dismisses an alert. An DismissAlert action is expected to be instantiated as it is, no static methods for this one. The only invocation looks like:

DismissAlert()

It can then be passed along to the *Actor* to perform the action.

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to perform the DismissAlert action.

Parameters **the_actor** – The *Actor* who will perform this action.

Raises

- |BrowsingError| – no alert was present.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

RespondToThePrompt

class screenpy.actions.respond_to_the_prompt.**RespondToThePrompt**(*text: str*)
Responds to a javascript prompt by entering the specified text and accepting the prompt. RespondToThePrompt is expected to be instantiated using its with_() static method. A typical instantiation might look like:

RespondToThePrompt.with_("I am big. It's the pictures that got small.")

It can then be passed along to the *Actor* to perform the action.

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to perform the RespondToPrompt action.

Parameters **the_actor** – The *Actor* who will perform this action.

Raises

- |BrowsingError| – no alert was present.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

static with_(*text: str*) → screenpy.actions.respond_to_the_prompt.RespondToThePrompt
Specifies the text to enter into the prompt.

Parameters **text** – the text to enter.

Returns RespondToTheText

SwitchTo

class screenpy.actions.switch_to.**SwitchTo**(*target: Optional[screenpy.target.Target]*)
Switches to something, most likely an iframe, or back to default. A SwitchTo action is expected to be instantiated by its static *the()* or *default()* methods, or on its own. A typical invocation might look like:

SwitchTo.the(ORDERS_FRAME)

SwitchTo.default()

It can then be passed along to the *Actor* to perform the action.

static default() → screenpy.actions.switch_to.SwitchTo
Switches back to the default frame, the browser window.

Returns *SwitchTo*

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to perform the SwitchTo action.

Parameters **the_actor** – The *Actor* who will perform this action.

Raises |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

static the(*target: screenpy.target.Target*) → screenpy.actions.switch_to.SwitchTo
Provide the element to switch to.

Parameters **target** – the *Target* describing the element to switch to.

Returns *SwitchTo*

Wait

class screenpy.actions.wait.**Wait**(*seconds: int = 20*, *target: Optional[screenpy.target.Target] = None*)
Waits for an element to fulfill a certain condition. A Wait action is expected to be instantiated by its *for_()* method, followed by one of its strategies. By default, the *to_appear()* strategy is used. Wait can also be instantiated with an integer, like Wait(30), which will set the timeout to be used. Some examples of invocations:

Wait.for_the(LOGIN_FORM)

Wait.for_the(WELCOME_BANNER).to_contain_text("Welcome!")

Wait.for(CONFETTI).to_disappear()

Wait(10).seconds_for_the(PARADE_FLOATS).to_appear()

It can then be passed along to the *Actor* to perform the action.

static for_(*target: screenpy.target.Target*) → screenpy.actions.wait.Wait
Creates a new Wait action holding the provided target.

Parameters **target** – The *Target* to wait for.

Returns *Wait*

static for_the(*target: screenpy.target.Target*) → screenpy.actions.wait.Wait
Syntactic sugar for *for_()*

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to perform the Wait action, using the contained strategy and any extra arguments provided.

Parameters **the_actor** – The *Actor* who will perform this action.

Raises

- |UnableToActError| – no target was supplied.
- |UnableToPerformError| – the actor does not have the ability to *BrowseTheWeb*.

seconds_for(*target: screenpy.target.Target*) → screenpy.actions.wait.Wait
Sets the target after invoking *Wait* with the number of seconds you want wait to allow the target to fulfill the expected condition. For example:

Wait(60).seconds_for(CONFETTI).to_disappear()

This will ask the actor to wait up to 60 seconds for CONFETTI to disappear before throwing an exception.

Parameters **target** – The *Target* to wait for.

Returns *Wait*

seconds_for_the(*target: screenpy.target.Target*) → screenpy.actions.wait.Wait
Syntactic sugar for *seconds_for()*

to_appear() → screenpy.actions.wait.Wait
Uses Selenium's "visibility of element located" strategy. This is the default strategy, so calling this is not strictly necessary.

Returns *Wait*

to_be_clickable() → screenpy.actions.wait.Wait
Uses Selenium's "to be clickable" strategy.

Returns *Wait*

to_contain_text(*text: str*) → screenpy.actions.wait.Wait
Uses Selenium's "text to be present in element" strategy.

Parameters **text** – the text to expect to be present.

Returns *Wait*

to_disappear() → screenpy.actions.wait.Wait
Uses Selenium's "invisibility of element located" strategy.

Returns *Wait*

using(*strategy: object*) → screenpy.actions.wait.Wait
Uses the given strategy to wait for the target.

Parameters **strategy** – the condition to use to wait. This can be one of Selenium's Expected Conditions, or it can be a custom Callable that accepts a Tuple[By, str] locator.

Returns *Wait*

Pause

class screenpy.actions.pause.**Pause**(*number: float*)
Pauses the actor's actions for a set amount of time. This class should only be used when absolutely necessary. You must call one of the "..._because" methods to pass in the reason for your pause; an *UnableToActError* will be raised if no reason was given when the actor attempts to perform this action.

A Pause action is expected to be instantiated by its static *for_()* method, followed by one of the methods that supply a reason (seconds_because, second_because, or milliseconds_because). A typical invocation might look like:

Pause.for_(500).milliseconds_because("the welcome banner needs to hide.")

It can then be passed along to the *Actor* to perform the action.

static for_(*number: int*) → screenpy.actions.pause.Pause
How many seconds or milliseconds to wait for.

Parameters **number** – the number of seconds or milliseconds to sleep for.

Returns *Pause*

milliseconds_because(*reason: str*) → screenpy.actions.pause.Pause
Tells the Pause instance to use milliseconds and provides a reason for the pause. Hard waits are the worst of all wait strategies, so providing a reason will help explain why it was necessary to use this strategy.

Parameters **reason** – the reason for needing to pause.

Returns *Pause*

perform_as(*the_actor: screenpy.actor.Actor*) → None
Asks the actor to take their union-mandated break.

Parameters **the_actor** – the *Actor* who will perform this action.

Raises |UnableToActError| – no reason was supplied.

second_because(*reason: str*) → screenpy.actions.pause.Pause
Syntactic sugar for Pause.seconds_because

seconds_because(*reason: str*) → screenpy.actions.pause.Pause
Tells the Pause instance to use seconds and provides a reason for the pause. Hard waits are the worst of all wait strategies, so providing a reason will help explain why it was necessary to use this strategy.

Parameters **reason** – the reason for needing to pause.

Returns *Pause*

Debug

class screenpy.actions.debug.**Debug**
In long chains of actions, it can be difficult to drop a debugger in the right place. This action can be placed anywhere in the chain to give you a debugger in the middle of the action chain. This action uses Python 3.7+'s breakpoint() call if it can, otherwise it will default to pdb.set_trace().

A Debug action is expected to be instantiated in the standard way. A typical instantiation will always look like:

Debug()

It can then be passed along to the *Actor* to perform the action.

perform_as(*the_actor: screenpy.actor.Actor*) → None
Activates a debugger.

Parameters **the_actor** – the *Actor* who will perform this action.

4.5 Questions

Questions are asked by an actor about the current state of the page or application. They are the first half (the "actual value") of ScreenPy's test assertions (the other half, *Resolutions*, is next).

4.5.1 Asking Questions

Typically, you will not be asking a question without an expected answer. This is how you do test assertions in ScreenPy:

```
from screenpy.questions import Text
from screenpy.resolutions import ReadsExactly

from ..user_interface.homepage import WELCOME_MESSAGE

Perry.should_see_the((Text.of_the(WELCOME_MESSAGE), ReadsExactly("Welcome!")), )
```

We'll talk about *Resolutions* next, but that call to *should_see_the()* is taking in our question. Behind the curtain, our actor is investigating the current state of the application (using their ability to *BrowseTheWeb*) to find out what the text actually says at the locator described by WELCOME_MESSAGE. They take that answer and compare it to the

expected answer passed in by the resolution. If they match, a comedy! Our test passes. If they do not match, a tragedy! Our test fails.

4.5.2 Writing New Questions

It is very likely that you may want to write additional questions, and you are encouraged to do so! The only prescribed method for a question class is an `asked_by` method that takes in an actor. This method will do the work of getting the answer to the question. For example, you may want to take a look at the `asked_by()` method of the *Text* class.

A base class for Questions is provided to ensure the required methods are defined: `screenpy.questions.base_question.BaseQuestion`

4.5.3 Provided Questions

List

class screenpy.questions.list.**List**(*target: screenpy.target.Target*)

Asks for a list of elements, viewed by an *Actor*. This question is meant to be instantiated using its static `of()` or `of_all()` methods. Typical invocations might look like:

List.of(SEARCH_RESULTS)

List.of_all(IMAGES)

It can then be passed along to the *Actor* to ask the question.

Number

class screenpy.questions.number.**Number**(*target: screenpy.target.Target*)

Asks how many of a certain element are on the page, viewed by an *Actor*. This question is meant to be instantiated via its static `of()` method. A typical invocation might look like:

Number.of(SEARCH_RESULTS)

It can then be passed along to the *Actor* to ask the question.

Text

class screenpy.questions.text.**Text**(*target: screenpy.target.Target*, *multi: bool = False*)

Asks what text appears in an element or elements, viewed by an *Actor*. This question is meant to be instantiated using its static `of()` or `of_all()` methods. Typical invocations might look like:

Text.of(THE_WELCOME_HEADER)

Text.of_all(SEARCH_RESULTS)

It can then be passed along to the *Actor* to ask the question.

Selected

class screenpy.questions.selected.**Selected**(*target: screenpy.target.Target*, *multi: bool = False*)

Answers questions about what options are selected in dropdowns, multi-select fields, etc, viewed by an *Actor*. This question is meant to be instantiated using its static `option_from` or `option_from` methods. Typical invocations might look like:

Selected.option_from(THE_STATE_DROPDOWN)

Selected.options_from(INDUSTRIES)

It can then be passed along to the *Actor* to ask the question.

4.6 Resolutions

Resolutions provide an expected answer to questions. They are the second half of test assertions in ScreenPy: the "expected value". (The first half are *Questions*, if you missed that page.)

4.6.1 Using Resolutions

Like *Questions*, you probably will not use a resolution directly. You will typically pass a resolution along with a question into your actor's *should_see_the()* method:

```
from screenpy.questions import Text
from screenpy.resolutions import ReadsExactly

from ..user_interface.homepage import WELCOME_MESSAGE

Perry.should_see_the((Text.of_the(WELCOME_MESSAGE), ReadsExactly("Welcome!")))
```

In that line of code, `ReadsExactly` is returning a PyHamcrest matcher. It will be evaluated later as *should_see_the()* does its job. If the expected value ("Welcome!") matches the actual value retrieved by our question, bravo! Our test passes. If they do not match, boo! Our test fails.

4.6.2 Writing New Resolutions

Resolutions are really just an abstraction barrier for the truly excellent PyHamcrest library. To add your own resolutions, create your resolution class by inheriting from the `BaseResolution` class. All you need to provide in your resolution is a `line` class property, which is just a human readable string for the log, and then to define the `__init__` method.

The custom Resolution's `__init__` method will need to set the expected value, and instantiate the PyHamcrest matcher that your resolution is masking. For several examples, see the documentation of the *Provided Resolutions* below.

4.6.3 Provided Resolutions

ContainsTheText

class screenpy.resolutions.**ContainsTheText**(*substring: str*)

Matches a substring (e.g. *"play" in "screenplay"*).

IsEmpty

class screenpy.resolutions.**IsEmpty**

Matches on an empty collection (e.g. *[]*).

IsEqualTo

class screenpy.resolutions.**IsEqualTo**(*obj: object*)
Matches on equality (i.e. *a* == *b*).

IsNot

class screenpy.resolutions.**IsNot**(*resolution: Any*)
Matches a negated Resolution (e.g. *not ReadsExactly("yes")*).

ReadsExactly

class screenpy.resolutions.**ReadsExactly**(*string: str*)
Matches a string exactly (e.g. *"screenplay" == "screenplay"*).

4.7 Wait Strategies

Automated test scripts are *fast*. When a test runs quickly, sometimes it can try to act on an element that isn't quite ready or hasn't even been drawn yet. ScreenPy allows you to use each of the prominent waiting strategies.

You can also reference Selenium's "Waits" documentation for more information.

4.7.1 Explicit Waits

ScreenPy provides a *Wait* function to wait for certain elements to appear, to be clickable, to contain text, or to disappear. These are included as a convenience because they are the most common strategies required. If those strategies aren't enough, you can also pass in your own strategy. Here are some examples of how this action can be used:

```
from screenpy.actions import Wait

# waits 20 seconds for the sign in modal to appear
Perry.attempts_to(Wait.for_the(LOGIN_MODAL))

# waits 42 seconds for the welcome banner to disappear
Perry.attempts_to(Wait(42).seconds_for(THE_WELCOME_BANNER).to_disappear())

# waits 20 seconds for a custom expected condition
Perry.attempts_to(Wait.for_the(PROFILE_ICON).using(appears_in_greyscale))
```

4.7.2 Implicit Waits

Implicit waiting is handled at the driver level. This is the less preferred method of waiting, but it can be useful in some situations, and does prevent a lot of Wait actions being littered around your action chains. Before you pass the driver in, you can set the implicit wait timeout like so:

```
from selenium.webdriver import Firefox

driver = Firefox()
driver.implicitly_wait(30)
```

(continues on next page)

(continued from previous page)

```
Perry = AnActor.who_can(BrowseTheWeb.using(driver))
```

4.7.3 Hard Waits

This method of waiting is discouraged. However, sometimes you just **need** to pause the script for a few moments, whether it's for *Debugging* or if it's because of an animation that defies any reasonable method of explicitly waiting for it to complete.

In these situations, as a last resort, ScreenPy offers the *Pause* action. Here are some ways to use it:

```
from screenpy.actions import Pause

Perry.attempts_to(Pause.for_(30).seconds_because("I need to get a new locator."))

Perry.attempts_to(Pause.for_(500).milliseconds_because("the banner animation must␣
↪finish."))
```

CHAPTER 5

Debugging

Debugging in ScreenPy can sometimes be difficult. If you're used to stepping through code using a debugger, getting to the part where your *Actor* is performing their *Actions* can be difficult.

To aid in debugging, the *Debug* action class can be used to drop into a debugger in the middle of any action chain! It hooks into Python 3.7+'s *breakpoint* function if it can, so you can modify your preferred debugger and turn debugging off by manipulating the *PYTHONBREAKPOINT* environment variable. You can read more about this excellent new function by perusing PEP553.

As for the action class, here's an example of an action chain:

```
given(Perry).was_able_to(
    Click.on_the(LOGIN_LINK),
    Enter.the_text(USERNAME).into_the(USERNAME_FIELD),
    Enter.the_password(PASSWORD).into_the(PASSWORD_FIELD),
    Click.on_the(SIGN_IN_BUTTON),
    Wait(60).seconds_for_the(WELCOME_BANNER),
)
```

If we know we have some issue after entering the username and password, but before clicking the sign in button, we can add a *Debug()* call there:

```
given(Perry).was_able_to(
    Click.on_the(LOGIN_LINK),
    Enter.the_text(USERNAME).into_the(USERNAME_FIELD),
    Enter.the_password(PASSWORD).into_the(PASSWORD_FIELD),
    Debug(),  # gives you a debugger here!
    Click.on_the(SIGN_IN_BUTTON),
    Wait(60).seconds_for_the(WELCOME_BANNER),
)
```

Now the test will drop us into either your chosen debugger or *pdb*. You'll need to return a couple times to get back up to the *Actor* class's *attempts_to()* method. From there, you can step through the rest of the actions one at a time, or dive into one if you need to!

5.1 Alternative Method

If you just need the actor to hold on a second while you verify the state of the webpage, you can use the *Pause* action instead, like so:

```
given(Perry).was_able_to(
    Click.on_the(LOGIN_LINK),
    Enter.the_text(USERNAME).into_the(USERNAME_FIELD),
    Enter.the_password(PASSWORD).into_the(PASSWORD_FIELD),
    Pause.for_(20).seconds_because("I need to see something"),  # stops the execution␣
↪here for 20 seconds.
    Click.on_the(SIGN_IN_BUTTON),
    Wait(60).seconds_for_the(WELCOME_BANNER),
)
```

CHAPTER 6

Exceptions

There are several exceptions thrown about in ScreenPy. Mostly they are used to provide extra context when other exceptions are raised.

6.1 Base

class screenpy.exceptions.**ScreenPyError**
The base exception for all of ScreenPy.

6.2 Ability Exceptions

class screenpy.exceptions.**AbilityError**
These errors are raised when an ability fails in some way.

class screenpy.abilities.browse_the_web.**BrowsingError**
Raised when BrowseTheWeb encounters an error.

6.3 Action Exceptions

class screenpy.exceptions.**ActionError**
These errors are raised when an action fails.

class screenpy.exceptions.**DeliveryError**
Raised when an action encounters an error while being performed.

class screenpy.exceptions.**UnableToActError**
Raised when an action is missing direction.

6.4 Actor Exceptions

class screenpy.actor.**UnableToPerformError**
Raised when an actor does not have the ability to perform the action they attempted.

6.5 Target Exceptions

class screenpy.target.**TargetingError**
Raised when there is an issue preventing target acquisition.

CHAPTER 7

Additional Context

Screenplay Pattern uses composition instead of inheritance to form the test suite. The concept was first formed by Antony Marcano— frustrated with Page Object Model files growing unreasonably large— under the name the Journey Pattern.

You can watch Antony's talk about Screenplay Pattern at SeleniumConf2016, which is the same talk that got me interested in this pattern!

You can also see some documentation about screenplay pattern from the folks who made SerenityBDD, the library that ScreenPy is modeled after: The Screenplay Pattern - Serenity/JS Handbook

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

U

W