Aloe Documentation

Release 0.1.20.dev0+g265fd2d.d20190314

Alexey Kotylarov

Mar 14, 2019

Contents

Aloe is a Gherkin-based Behavior Driven Development tool for Python based on Nose.

CHAPTER 1

Running Aloe

The `aloe` helper runs Nose with the Aloe plugin enabled.

`aloe` accepts the same flags as `nosetests` and so these are not extensively documented here.

<feature>
: Run only the specified feature files.

-n `N[,N...]`
: Only run the specified scenarios (by number, 1-based) in each feature. Makes sense when only specifying one feature to run, for example:

```
aloe features/calculator.feature -n 1
```

--test-class
: Override the class used as a base for each feature.

--no-ignore-python
: Run Python tests as well as Gherkin.

-a `attr`
: Run features and scenarios with the given tag. (This is a Nose flag, but works the same for Gherkin tags.)

-a `'!attr'`
: Run features and scenarios that do not have the given tag.

CHAPTER 2

Writing Features

The standard Gherkin syntax is supported, including scenario outlines, doc strings, data tables and internationalization.

2.1 Feature

A feature is a single file that typically defines a single story. It has a name and an optional description, an optional *background* and many scenarios.

```
Feature: Search

    As a user
    I want to do a search for something not in the default categories
    So that I can provide more detailed search parameters
```

A feature may also have *tags*.

2.2 Background

The *background* is an optional section that is run before every scenario and contains steps. It is used to set up fixtures common to each *scenario* of the *feature*.

A background does not have a name or tags.

If a step fails during the background the scenario will fail.

```
Background:
    Given my location is Melbourne, Victoria
```

2.3 Scenario

Scenarios are the individual tests that make up a *feature*. Scenarios have a name and may optionally *tags*. The scenario consists of a number of steps.

If a step fails the *scenario* will fail.

```
Scenario: Check the results
    When I search for "pet food" and press enter

    # A step with a multiline string.
    Then I should see the text:
    """
    1 result found in 0.15 seconds.
    """

    # A step with a table.
    And I should see the results:
        | Name (primaryText) | Description (secondaryText) |
        | Pets Inc           | Your one stop pet shop      |
```

2.4 Scenario Outline

A scenario outline is a template for building scenarios from the rows of a table named `Examples`. Parameters are written in the form `<Parameter>`, where each named parameter must be present in the table.

Scenario outlines have a name and may optionally have *tags*.

```
Scenario Outline: Search is correctly escaped
    When I search for "<Phrase>" and press enter
    Then I should be at <URL>

    Examples:
        | Phrase   | URL               |
        | pets     | /search/pets      |
        | pet food | /search/pet%20food |
```

2.5 Tags

Feature and scenario tags are specified using the form `@tag_name` and are converted to Nose attribute tags, and can be run/excluded using `-a`.

```
Feature: Search

    @integration
    Scenario: Live server works as expected
        When I search for "pet food"
        Then I should see >1 result
```

See docs for the Attribute selector plugin for more information.

2.6 Feature Loading

If features are not specified on the command line, Aloe will look for features in directories that are both:

- Named `features`;
- Located in a directory containing packages, that is, all their parent directories have an `__init__.py` file.

For example, given the following directory structure, only `one`, `three` and `seven` features will be run:

```
one/
    __init__.py
    features/
        one.feature
        two/
            three.feature
   examples/
       four.feature
five/
    __init__.py
    six/
        features/
            seven.feature
eight/
    nine/
        features/
            ten.feature
```

`four` will not be run because it is not in a directory named `features`. `ten` will not be run because its parent directory, `nine`, is not a package. This prevents discovering features of dependent packages if they are in a virtualenv inside the project directory.

CHAPTER 3

Defining Steps

aloe.**step**(*sentence=None*)

Decorates a function, so that it will become a new step definition.

You give step sentence either (by priority):

- with step function argument;
- with function doc; or
- with the function name exploded by underscores.

Parameters can be passed to steps using regular expressions. Parameters are passed in the order they are captured. Be aware that captured values are strings.

The first parameter passed into the decorated function is the `Step` object built for this step.

Examples:

```
@step("I go to the shops")
def _i_go_to_the_shops_step(self):
    '''Implements I go to the shops'''

    ...

@step
def _i_go_to_the_shops_step(self):
    '''I go to the shops'''

    ...

@step(r"I buy (\d+) oranges")
def _purchase_oranges_step(self, num_oranges):
    '''Buy a certain number of oranges'''

    num_oranges = int(num_oranges)

    ...
```

Steps can be passed a table of data.

```
Given the following users are registered:
    | Username | Real name |
    | danni    | Danni     |
    | alexey   | Alexey    |
```

This is exposed in the step as `Step.table` and `Step.hashes`.

```
@step(r'Given the following users? (?:is|are) registered:')
def _register_users(self):
    '''Register the given users'''

    for user in guess_types(self.hashes):
        register(username=user['Username'],
                 realname=user['Real name'])
```

Steps can be passed a multi-line "*Python string*".

```
Then I see a warning dialog:
    """
    Changes could not be saved.

    [Try Again]
    """
```

This is exposed in the step as `Step.multiline`.

The registered function will have an `unregister()` method that removes all the step definitions that are associated with it.

3.1 Common regular expressions for capturing data

String

```
Given I logged in as "alexey"
```

```
@step(r'I logged in as "([^"]*)"')
```

Number

```
Then the price should be $12.99
```

```
@step(r'The price should be \$(\d+(?:\.\d+)?)')
```

Path/URI/etc.

```
Given I visit /user/alexey/profile
```

```
@step(r'I visit ([^\s]+)')
```

3.2 Step loading

Steps can and should be defined in separate modules to the main application code. Aloe searches for modules to load steps from inside the `features` directories.

Steps can be placed in separate files, for example, `features/steps/browser.py` and `features/steps/data.py`, but all those files must be importable, so this requires creating a (possibly empty) `features/steps/__init__.py` alongside.

Additional 3rd-party steps (such as aloe_django) can be imported in from your `__init__.py`.

An imported step can be overridden by using `unregister()` on the function registered as a step. It can be then reused by defining a new step with the same or different sentence.

3.3 Tools for step writing

Useful tools for writing Aloe steps.

See also `aloe.world`.

`aloe.tools.`**`guess_types`**(*data*)

Converts a record or list of records from strings contained in outlines, table or hashes into a version with the types guessed.

Parameters **`data`** – a `Scenario.outlines`, `Step.table`, `Step.hashes` or any other `list`, list of lists or list of dicts.

Will guess the following (in priority order):

- `bool` (`true`/`false`)
- `None` (`null`)
- `int`
- `date` in ISO format (`yyyy-mm-dd`)
- `str`

The function operates recursively, so you should be able to pass nearly anything to it. At the very least basic types plus `dict` and iterables.

`aloe.tools.`**`hook_not_reentrant`**(*func*)

Decorate a hook as unable to be reentered while it is already in the stack.

Any further attempts to enter the hook before exiting will be replaced by a no-op.

This is generally useful for step hooks where a step might call `Step.behave_as()` and trigger a second level of step hooks i.e. when displaying information about the running test.

3.4 Writing good BDD steps

It's very easy with BDD testing to accidentally reinvent Python testing using a pseudo-language. Doing so removes much of the point of using BDD testing in the first place, so here is some advice to help write better BDD steps.

1. **Avoid implementation details**

   If you find yourself specifying implementation details of your application that aren't important to your behaviors, abstract them into another step.

Implementation:

```
When I fill in username with "danni"
And I fill in password with "secret"
And I press "Log on"
And I wait for AJAX to finish
```

Behavioral:

```
When I log on as "danni" with password "secret"
```

You can use `Step.behave_as()` to write a step that chains up several smaller steps.

Implementation:

```
Given the following flags are set:
    | flags                      |
    | user_registration_disabled |
    | user_export_disabled       |
```

Behavioral:

```
Given user registration is disabled
And user export is disabled
```

Remember you can generate related steps using a loop.

```
for description, flag in ( ... ):
    @step(description + ' is enabled')
    def _enable_flag(self):

        set_flag(flag, enabled=True)

    @step(description + ' is disabled')
    def _disable_flag(self):

        set_flag(flag, enabled=False)
```

Furthermore, steps that are needed by all features can be moved to a *`each_example()`* callback.

If you want to write reusable steps, you can sometimes mix behavior and declaration.

```
Then I should see results:
    | Business Name (primaryText) | Blurb (secondaryText) |
    | Pet Supplies.com            | An online store for...  |
```

2. **Avoid conjunctions in steps**

   If you're writing a step that contains an *and* or other conjunction consider breaking your step into two.

   Bad:

   ```
   When I log out and log back in as danni
   ```

   Good:

   ```
   When I log out
   And I log in as danni
   ```

   You can pass state between steps using `world`.

3. **Support natural language**

   It's easier to write tests if the language they support is natural, including things such as plurals.

   Unnatural:

```
Given there are 1 users in the database
```

   Natural:

```
Given there is 1 user in the database
```

   This can be done with regular expressions.

```
@step('There (?:is|are) (\d+) users? in the database')
```

CHAPTER 4

Hooks

Hooks can be installed to run `before`, `around` and `after` part of the test.

Hooks can be used to set up and flush test fixtures, apply mocks or capture failures.

class aloe.**before**

@all

Run this function before *everything*.

Example:

```
from aloe import before

@before.all
def before_all():
    print("Before all")
```

@each_feature

Run this function before each *feature*.

Parameters **feature** (Feature) – the feature about to be run

Example:

```
from aloe import before

@before.each_feature
def before_feature(feature):
    print("Before feature")
```

@each_example

Run this function before each *scenario example*.

Parameters

- **scenario** (Scenario) – the scenario about to be run

- **outline** (*dict*) – the outline of the example about to be run
- **steps** (*list*) – the steps about to be run

Example:

```
from aloe import before

@before.each_example
def before_example(scenario, outline, steps):
    print("Before example")
```

@each_step

Run this function before each *step*.

Parameters **step** (Step) – the step about to be run

Example:

```
from aloe import before

@before.each_step
def before_step(step):
    print("Before step")
```

class aloe.**after**

Run functions *after* an event. See *aloe.before*.

Example:

```
from aloe import after

@after.each_step
def after_step(step):
    print("After step")
```

class aloe.**around**

Define context managers that run *around* an event. See *aloe.before*.

Example:

```
from contextlib import contextmanager

from aloe import around

@around.each_step
@contextmanager
def around_step(step):
    print("Before step")
    yield
    print("After step")
```

CHAPTER 5

World

As a convenience, Aloe provides a `world` object that can be used to store information related to the test process. Typical usage includes storing the expected results between steps, or objects or functions that are useful for every step, such as an instance of a Selenium browser.

Aloe does not explicitly reset `world` between scenarios or features, so any clean-up must be done by the callbacks.

class aloe.**world**
Store arbitrary data. Shared between hooks and steps.

CHAPTER 6

Features, Scenarios and Steps

6.1 Feature

class aloe.parser.**Feature**

A complete Gherkin feature.

Features can either be constructed *from_file()* or *from_string()*.

description

The description of the feature (the text that comes directly under the feature).

dialect

The Gherkin dialect for the feature.

classmethod from_file(*filename*, *language=None*)

Parse a file or filename into a *Feature*.

classmethod from_string(*string*, *language=None*)

Parse a string into a *Feature*.

location

Location as 'filename:line'

classmethod parse(*string=None*, *filename=None*, *language=None*)

Parse either a string or a file.

tags

Tags for a feature.

Tags are applied to a feature using the appropriate Gherkin syntax:

```
@tag1 @tag2
Feature: Eat leaves
```

6.2 Background

class aloe.parser.**Background**
The background of all *Scenario* in a *Feature*.

feature
The *Feature* this scenario belongs to.

location
Location as 'filename:line'

6.3 Scenario

class aloe.parser.**Scenario**
A scenario within a *Feature*.

name
The name of this scenario.

feature
The *Feature* this scenario belongs to.

outlines
The examples for this scenario outline as a list of dicts mapping column name to value.

location
Location as 'filename:line'

outlines_table
Return the scenario outline examples as a table.

tags
Tags for the *feature* and the scenario.

6.4 Step

class aloe.parser.**Step**
A single statement within a test.

A *Scenario* or *Background* is composed of multiple *Step*.

scenario
The *Scenario* this step belongs to (if inside a scenario).

background
The *Background* this step belongs to (if inside a background).

test
The instance of `unittest.TestCase` running the current test, or None if not currently in a test (e.g. in a *each_feature()* callback).

testclass
The `unittest.TestCase` used to run this test. Use *test* for the *instance* of the test case.

passed
The step passed (used in `after` and `around`).

failed
The step failed (used in `after` and `around`).

behave_as(*sentence*)
Execute another step.

Example:

```
self.behave_as("Given I am at the market")
```

given(*sentence*)
Execute another step.

Example:

```
self.given("I am at the market")
```

when(*sentence*)
Execute another step.

Example:

```
self.when("I buy two oranges")
```

then(*sentence*)
Execute another step.

Example:

```
self.then("I will be charged 60c")
```

container
The background or scenario that contains this step.

feature
The *Feature* this step is a part of.

hashes
Return the table attached to the step as an iterable of hashes, where the first row - the column headings - supplies keys for all the others.

e.g.:

```
Then I have fruit:
    | apples | oranges |
    | 0      | 2       |
```

Becomes:

```
({
    'apples': '0',
    'oranges': '2',
},)
```

keys
Return the first row of a table if this statement contains one.

location
Location as 'filename:line'

multiline = None
A Gherkin multiline string with the appropriate indenting removed.

```
Then I have poem:
    """
    Glittering-Minded deathless Aphrodite,
    I beg you, Zeus's daughter, weaver of snares,
    Don't shatter my heart with fierce
    Pain, goddess,
    """
```

outline = None
If this step is a part of an outline, the reference to the outline.

parse_steps_from_string(*string*, ***kwargs*)
Parse a number of steps, returns an iterable of *Step*.

This is used by `step.behave_as()`.

sentence = None
The sentence parsed for this step.

table = None
A Gherkin table as an iterable of rows, themselves iterables of cells.

e.g.:

```
Then I have fruit:
    | apples | oranges |
    | 0      | 2       |
```

Becomes:

```
(('apples', 'oranges'), ('0', '2'))
```

CHAPTER 7

Optional Extras

7.1 Factory Boy Integration

Aloe integration with factory_boy to create objects from factories.

Remember when writing BDD tests to describe the behavior you want and not just use Aloe as a syntax for writing complex tests (that defeats the point of BDD). Hide the complexity of setting up the objects in your factory or write a custom step.

To activate these steps import *aloe.steps.factoryboy* into your `steps/__init__.py`.

`aloe.steps.factoryboy.`**`step_from_factory`**(*factory*)

Decorator to register a `factory.Factory` as an Aloe step:

> Given/And I have (a/an/*n*) *object(s)*

An optional table can be passed containing attributes that would be passed as *kwargs* to `factory.Factory.create()`. Multiple rows or a number of objects can be passed to create more than one object. If a number of objects is requested, at most one row can be given, passed as *kwargs* to `factory.Factory.create_batch()`.

The name of the object and its plural can be specified as:

- `_verbose_name` and `_verbose_name_plural` attributes on the factory;
- If the factory creates a Django model, and its name corresponds to the model class name (e.g. `UserFactory` and `User`), `verbose_name` and `verbose_name_plural` of the model;

If neither is specified, the object name is inferred from the factory class name.

Example:

```
@step_from_factory
class RandomUserFactory(factory.Factory):
    '''See Factory Boy docs'''

    class Meta:
```

(continues on next page)

(continued from previous page)

```
        model = models.User

    first_name = factory.Faker('first_name')
    last_name = factory.Faker('last_name')

    _verbose_name = "random user"
```

```
Given I have a random user
# Then I have created 1 user: Lucy Murray (a random name)

Given I have 10 random users
# Then I have created 10 users with different random names

Given I have random users:
    | first_name | last_name |
    | Danielle   | Madeley   |
    | Alexey     | Kotlyarov |
# Then I have created 2 users: Danni and Alexey

Given I have 10 random users:
    | first_name |
    | Joe        |
# Then I have created 10 users all with the first name Joe
```

7.2 Sphinx Extensions

Extensions to Sphinx for documenting Aloe packages.

Add these extensions to your Sphinx `conf.py`:

```
extensions = [
    'sphinx.ext.autodoc',
    'aloe_sphinx.gerkindomain',
    'aloe_sphinx.autosteps',
]
```

7.2.1 Gherkin Domain

aloe_sphinx.gherkindomain

The Gherkin Domain for Sphinx provides additional directives for documenting steps using Sphinx.

`.. gherkin:restep:: Sentence regex`
Provide the documentation for a Gherkin regular expression step.

For example:

```
.. gherkin:restep:: (?:Given|When|And) I visit the supermarket

    I am at the supermarket.
```

Is rendered as:

Step (?:Given|When|And) I visit the supermarket
I am at the supermarket.

7.2.2 Steps Autodocumenter

aloe_sphinx.autosteps

An autodocumenter for Aloe steps built on top of `sphinx.ext.autodoc`.

This extension will identify functions decorated with `step()` (including private functions) and expose them in your documentation with their step sentence.

CHAPTER 8

Extending Aloe

class aloe.testclass.**TestCase**

The base test class for tests compiled from Gherkin features.

Aloe runs all tests within a `unittest.TestCase`. You can extend this class to run your tests with certain other features, i.e. using Django's `TestCase`.

CHAPTER 9

Extensions

- aloe_django – Django integration for *Aloe*.
- aloe_webdriver – Selenium integration for *Aloe*.

CHAPTER 10

Porting from Lettuce

Aloe, started as a fork of Lettuce, tries to be compatible where it makes sense. However, there are following incompatible changes:

- Aloe aims to use compatible Gherkin syntax, as such the following no longer work:
  - Using `"` to indicate the indent of a multiline string; and
  - Comments after steps.
- The `each_scenario()`, `each_background()` and `outline()` callbacks are removed. Use `each_example()`.
- The `-s` option for running particular scenarios is renamed to `-n`.
- Django-related functionality, including the `harvest` command, is moved to a separate project, aloe_django.
- `terrain.py` has no particular significance. It will be imported but only if it exists at the same directory with the other step definition files, and not above it.
- Step files are loaded using the normal Python import mechanism. This means the directory they are in must have a (possibly empty) *__init__.py*.
- Scenario outlines must be declared with "Scenario Outline", and scenarios without examples must use "Scenario" - Lettuce allowed using either.

CHAPTER 11

Getting Started

Install Aloe:

```
pip install aloe
```

Write your first feature `features/calculator.feature`:

```
Feature: Add up numbers

As a mathematically challenged user
I want to add numbers
So that I know the total

Scenario: Add two numbers
    Given I have entered 50 into the calculator
    And I have entered 30 into the calculator
    When I press add
    Then the result should be 80 on the screen
```

Features are written using the Gherkin syntax.

Now run `aloe features/calculator.feature` and see it fail because there are no step definitions:

```
$ aloe features/calculator.feature
(...)
aloe.exceptions.NoDefinitionFound: The step r"Given I have entered 50 into the
calculator" is not defined

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Now add the definitions in `features/__init__.py`:

```
from calculator import add

from aloe import before, step, world


@before.each_example
def clear(*args):
    """Reset the calculator state before each scenario."""
    world.numbers = []
    world.result = 0


@step(r'I have entered (\d+) into the calculator')
def enter_number(self, number):
    world.numbers.append(float(number))


@step(r'I press add')
def press_add(self):
    world.result = add(*world.numbers)


@step(r'The result should be (\d+) on the screen')
def assert_result(self, result):
    assert world.result == float(result)
```

And the implementation stub in `calculator.py`:

```
def add(*numbers):
    return 0
```

Aloe will tell you that there is an error, including the location of the failing step, as if it was a normal Python test:

```
$ aloe features/calculator.feature

F
======================================================================
FAIL: Add two numbers (features.calculator: Add up numbers)
----------------------------------------------------------------------
Traceback (most recent call last):
  (...)
  File ".../features/calculator.feature", line 11, in Add two numbers
    Then the result should be 80 on the screen
  File ".../aloe/registry.py", line 161, in wrapped
    return function(*args, **kwargs)
  File ".../features/__init__.py", line 25, in assert_result
    assert world.result == float(result)
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Let's implement the function properly:

```
def add(*numbers):
    return sum(numbers)
```

Now it works:

```
$ aloe features/calculator.feature
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

CHAPTER 12

History

Aloe originally started life as a branch of the Python BDD tool Lettuce. Like so many succulents, it grew into so much more than that.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

D

F

G

H

K

L

M

N

O

P

S

T

W