Morelia Documentation

Release 0.9.0

Morelia authors

2019-06-23

Contents

Morelia is a Python Behavior Driven Development (BDD[1]) library.

BDD is an agile software development process that encourages collaboration between developers, QA and business participants.

Test scenarios written in natural language make BDD foundation. They are comprehensible for non-technical participants who wrote them yet unambiguous for developers and QA.

Morelia makes it easy for developers to integrate BDD into their existing unittest frameworks. It is easy to run under pytest, nose, tox or integrate with django, flask or any other python framework because no special code have to be written.

You as developer are in charge of how tests are organized. No need to fit into rigid rules forced by some other BDD frameworks.

Mascot:

If you're scenario writer (product owner/product manager/professional tester) we recommended reading the:

- *Tutorial for scenario writers* and then
- *Gherkin language reference* when you need to create some more advanced scenarios

If you're programmer we recommended reading the:

- *Quick usage guide* and then
- *Gherkin language reference* and
- *API*

[1] Behavior Driven Development https://en.wikipedia.org/wiki/Behavior-driven_development

CHAPTER 1

Contents

1.1 Installation

In order to install Morelia, make sure Python is installed. Morelia was tested with:

- Python 3.5
- Python 3.6
- Python 3.7

1.1.1 Stable release

To install Morelia, run this command in your terminal:

```
$ pip install morelia
```

This is the preferred method to install Morelia, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.1.2 From sources

The sources for Morelia can be downloaded from the GitHub repo.

You can either clone the public repository:

```
$ git clone git://github.com/kidosoft/Morelia
```

Or download the tarball:

```
$ curl  -OL https://github.com/kidosoft/morelia/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

1.2 Quick usage guide

Write a feature description:

```
# calculator.feature

Feature: Addition
    In order to avoid silly mistakes
    As a math idiot
    I want to be told the sum of two numbers

Scenario: Add two numbers
    Given I have powered calculator on
    When I enter "50" into the calculator
    And I enter "70" into the calculator
    And I press add
    Then the result should be "120" on the screen
```

Create standard python's `unittest` and hook Morelia into it:

```
# test_acceptance.py

import unittest

from morelia import verify


class CalculatorTestCase(unittest.TestCase):

    def test_addition(self):
        """ Addition feature """
        verify('calculator.feature', self)
```

Run test with your favourite runner: unittest, pytest, nose, trial. You name it!

```
$ python -m unittest -v test_acceptance  # or
$ pytest test_acceptance.py  # or
$ nosetests -v test_acceptance.py  # or
$ trial test_acceptance.py  # or
$ # django/pyramid/flask/(place for your favourite test runner)
```

And you'll see which steps are missing:

```
F
======================================================================
FAIL: test_addition (test_acceptance.CalculatorTestCase)
Addition feature.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "(..)test_acceptance.py", line 31, in test_addition
    verify(filename, self)
  File "(..)/morelia/__init__.py", line 120, in verify
    execute_script(feature, suite, scenario=scenario, config=conf)
```

(continues on next page)

(continued from previous page)

```
  File "(..)/morelia/parser.py", line 59, in execute_script
    assert not_found == set(), message
AssertionError: Cannot match steps:

    def step_I_have_powered_calculator_on(self):
        r'I have powered calculator on'

        raise NotImplementedError('I have powered calculator on')

    def step_I_enter_number_into_the_calculator(self, number):
        r'I enter "([^"]+)" into the calculator'

        raise NotImplementedError('I enter "50" into the calculator')

    def step_I_enter_number_into_the_calculator(self, number):
        r'I enter "([^"]+)" into the calculator'

        raise NotImplementedError('I enter "70" into the calculator')

    def step_I_press_add(self):
        r'I press add'

        raise NotImplementedError('I press add')

    def step_the_result_should_be_number_on_the_screen(self, number):
        r'the result should be "([^"]+)" on the screen'

        raise NotImplementedError('the result should be "120" on the screen')

----------------------------------------------------------------------
Ran 1 test in 0.013s

FAILED (failures=1)
```

Now implement steps with standard `TestCases` that you are familiar:

```
# test_acceptance.py

import unittest

from morelia import run


class CalculatorTestCase(unittest.TestCase):

    def test_addition(self):
        """ Addition feature """
        verify('calculator.feature', self)

    def step_I_have_powered_calculator_on(self):
        r'I have powered calculator on'
        self.stack = []

    def step_I_enter_a_number_into_the_calculator(self, number):
        r'I enter "(\d+)" into the calculator'  # match by regexp
        self.stack.append(int(number))
```

(continues on next page)

(continued from previous page)

```
    def step_I_press_add(self):  # matched by method name
        self.result = sum(self.stack)

    def step_the_result_should_be_on_the_screen(self, number):
        r'the result should be "{number}" on the screen'  # match by format-like␣
↪string
        self.assertEqual(int(number), self.result)
```

And run it again:

```
$ python -m unittest test_acceptance

Feature: Addition
    In order to avoid silly mistakes
    As a math idiot
    I want to be told the sum of two numbers
Scenario: Add two numbers
    Given I have powered calculator on                          # pass  0.000s
    When I enter "50" into the calculator                       # pass  0.000s
    And I enter "70" into the calculator                        # pass  0.000s
    And I press add                                             # pass  0.001s
    Then the result should be "120" on the screen               # pass  0.001s
.
----------------------------------------------------------------------
Ran 1 test in 0.028s

OK
```

Note that Morelia does not waste anyone's time inventing a new testing back-end just to add a layer of literacy over our testage. Steps are miniature `TestCases`. Your onsite customer need never know, and your unit tests and customer tests can share their support methods. The same one test button can run all TDD and BDD tests.

1.3 Gherkin language reference

Language used to describe features is called "Gherkin". It's a little formalized natural language that's easy to write by non-programmers.

Each feature should be described in separate document.

1.3.1 Comments

To include comments inside feature document start line with hash (#). Everything after this to the end of line will be treated as a comment.

```
# this line is a comment
```

1.3.2 Language directive

If a comment looks like:

```
# language: de
```

Then feature description will be analyzed according to given native language. All supported languages with grammar keywords are here:

https://github.com/kidosoft/Morelia/blob/master/src/morelia/i18n.py

If there's no language directive then English is assumed.

1.3.3 Feature keyword

In each document should be one and only one "Feature" keyword. After "Feature" keyword goes name of a feature and optional description:

```
Feature: Addition
    In order to avoid silly mistakes
    As a math idiot
    I want to be told the sum of two numbers
```

Note that "In order", "As a", and "I want" are not keywords. That's a description. Description is free formed text although below is suggested form:

```
In order to <goal description>
As a <role>
I want to <action>
```

That form allows to look at feature from end user's perspective.

1.3.4 Scenario keyword

Each feature consists of one or more scenarios. Each scenario begins with "Scenario" keyword and it's name. Then go steps describing scenario:

```
Scenario: Add two numbers
    Given I have powered calculator on
    When I enter "50" into the calculator
    And I enter "70" into the calculator
    And I press add
    Then the result should be "120" on the screen
```

1.3.5 Steps

Each scenario consists of many steps. Steps have associated meaning:

- "Given" describes initial state of system
- "When" is used to describe actions
- "Then" is used to describe final state of system

"And" and "But" are used to enumerate more "Given", "When", "Then" steps.

It is suggested that sentences in "Given" part should be written in past tense. "When" part should be written in present tense and "Then" in future tense.

Note: For programmers

For information how methods are matched to steps see matching-steps.

1.3.6 Background keyword

If you have to repeat the same subset of "Given" steps in all of your scenarios you can use "Background" keyword. "Given" steps in "Background" are run as the very first steps in each scenario. E.g. instead of writing:

```
Scenario: Some scenario
    Given some setup
      And some condition
     When a first trigger occurs
     Then something good happens

Scenario: Some other scenario
    Given some setup
      And some condition
     When another trigger occurs
     Then something else happens
```

you can write:

```
Background:
    Given some setup
      And some condition

Scenario: Some scenario
     When a first trigger occurs
     Then something good happens

Scenario: Some other scenario
     When another trigger occurs
     Then something else happens
```

Note: For programmers

Refuse temptation to put into background steps that you need to perform in order to set up tests, which are of no use for scenario writer (e.g. "Set up database") Remember that you use `TestCases` so you can use `setUp`/`tearDown` methods!

1.3.7 Tables

To DRY[1] up a series of redundant scenarios, varying by only "payload" variables, roll the Scenarios up into a table, using <*angles*> around the payload variable names:

```
Scenario: orders above $100.00 to the continental US get free ground shipping
  When we send an order totaling $<total>, with a 12345 SKU, to our warehouse
   And the order will ship to <destination>
  Then the ground shipping cost is $<cost>
   And <rapid> delivery might be available
```

(continues on next page)

[1] Don't repeat yourself http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

(continued from previous page)

```
      |  total | destination            |  cost | rapid |

      |  98.00 | Rhode Island           |  8.25 |  yes  |
      | 101.00 | Rhode Island           |  0.00 |  yes  |
      |  99.00 | Kansas                 |  8.25 |  yes  |
      | 101.00 | Kansas                 |  0.00 |  yes  |
      |  99.00 | Hawaii                 |  8.25 |  yes  |
      | 101.00 | Hawaii                 |  8.25 |  yes  |
      | 101.00 | Alaska                 |  8.25 |  yes  |
      |  99.00 | Ontario, Canada        | 40.00 |   no  |
      |  99.00 | Brisbane, Australia    | 55.00 |   no  |
      |  99.00 | London, United Kingdom | 55.00 |   no  |
      |  99.00 | Kuantan, Malaysia      | 55.00 |   no  |
      | 101.00 | Tierra del Fuego       | 55.00 |   no  |
```

That Scenario will unroll into a series of 12 scenarios, each with one value from the table inserted into their placeholders <*total*>, <*destination*>, and <*rapid*>.

You can use many tables. It would be equivalent of permutation of all given rows.

Example

Below scenario:

```
Scenario: orders above $100.00 to the continental US get free ground shipping
  When we send an order totaling $<total>, with a 12345 SKU, to our warehouse
   And the order will ship to <destination>
   And we choose that delivery should be <speed>
       | speed   |

       | rapid   |
       | regular |

  Then the ground shipping cost is $<cost>

       |  total | destination            |  cost |

       |  98.00 | Rhode Island           |  8.25 |
       | 101.00 | Rhode Island           |  0.00 |
       |  99.00 | Kansas                 |  8.25 |
```

Is equivalent of series of scenarios:

```
Scenario: orders above $100.00 to the continental US get free ground shipping
  When we send an order totaling $<total>, with a 12345 SKU, to our warehouse
   And the order will ship to <destination>
   And we choose that delivery should be <speed>
  Then the ground shipping cost is $<cost>

       | speed   |  total | destination  |  cost |

       | rapid   |  98.00 | Rhode Island |  8.25 |
       | rapid   | 101.00 | Rhode Island |  0.00 |
       | rapid   |  99.00 | Kansas       |  8.25 |
       | regular |  98.00 | Rhode Island |  8.25 |
```

(continues on next page)

(continued from previous page)

```
      | regular | 101.00 | Rhode Island |  0.00 |
      | regular |  99.00 | Kansas       |  8.25 |
```

In above example 2 * 3 = 6 different scenarios would be generated.

Note: Compatibility

For compatibility with other Behavior Driven Development tools you can use "Scenario Outline" keyword instead of "Scenario" and mark table with "Examples" keyword if you prefer. Morelia would not enforce you to do that.

Note: For programmers

For information how methods are matched to steps with tables see matching-tables.

1.3.8 Doc Strings

Sometimes you need to include some larger chunks of text in steps as data. In order to accomplish this you can use doc-strings syntax:

```
Feature: Addition
    In order to avoid silly mistakes
    As a math idiot
    I want to be told the sum of two numbers

Scenario: Add two numbers
    Given I have powered calculator on
    When I enter "50" into the calculator
    And I enter "70" into the calculator
    And I press add
    Then I would see on the screen
        """
        Calculator example
        ==================
         50
        +70
        ---
        120
        """
```

Text enclosed within triple double-quotes will be attached as step's data.

Note: For programmers

Look at matching-docstrings for information how to access data in steps.

1.3.9 Labels

Each feature or scenario can be labeled:

```
@web
@android @ios
Feature: Addition
    In order to avoid silly mistakes
    As a math idiot
    I want to be told the sum of two numbers

@wip
Scenario: Add two numbers
    Given I have powered calculator on
    When I enter "50" into the calculator
    And I enter "70" into the calculator
    And I press add
    Then the result should be "120" on the screen
```

Labels are inherited. In above example all steps will be labeled with “web”, “android”, “ios”, “wip”. Labels allows to implement custom logic depending on labels given.

Note: For programmers

Look at labels-matching for information how to access labels in steps.

1.4 Tutorial for scenario writers

Scenarios are meant as tool for easier communication between developers, QA and business participants. This tutorial is written with non-programmers in mind.

Scenarios are written in a little formalized natural language. Programmers expect that each software feature will be written in separate file.

1.4.1 Language

Features can be written in your native language. It is much easier to describe what you expect from software in language that is well known for all participants. E.g. if all participants use Japanese at the beginning of feature file write:

```
# language: ja
```

Language code is two-letter code as listed in ISO 639-1 standard. If you skip that step then English will be assumed.

1.4.2 Feature

Next, write a name of feature that you need.

```
# language: en

Feature: Add an item to basket
```

"Feature" is a keyword and have to be used literally. Every language has it's own set of keywords. E.g. for Spanish (es) it would be look like:

```
# language: es

Característica: Agregar un elemento a la cesta
```

Next goes introductory description. It's an optional free formed text. If you want your programmer to better understand what you expect from him please try to follow a schema:

```
In order to <goal description>
As a <role>
I want to <action>
```

The goal description is essential for programmer to understand in what is a goal of a whole feature and why you want this feature to be added to software. It allows him to structure software in a way that it'll be easier in future add new features and modify existing ones. E.g.:

```
Feature: Add an item to basket

    In order to buy an item
```

would tell him that in the end an item added to basked will be sold.

Describing role should help both you and programmer to start thinking from role's perspective. E.g.:

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
```

From that point both you and programmer should start thinking like a logged user and describe actions in first person as logged user.

Next describe what you as a "role" want to do:

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket
```

In this part you can also write other non-behavioral information for programmer.

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket
```

(continues on next page)

(continued from previous page)

```
    Adding item to basket should not reload page
```

1.4.3 Scenario

Each feature consists of one or more scenarios. Each scenario describes what <role> is doing, step by step, to realize feature. Scenario starts with keyword "Scenario" in your chosen language (e.g. "Scénario" in French) followed by short description of that case:

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket

    Adding item to basket should not reload page

Scenario: Adding from item's description page
```

In above example a logged user will visit item's description page and add item to basket.

Next you write each action that you would perform if you were a <role>. Use a schema:

```
Given <precondition that have to be met>
And  <more preconditions>
But  <more preconditions>
...

When <action that have to be done>
And <next action>
But <next action>
...

Then <observable result of actions>
And <more results>
But <more results>
...
```

"Given", "When", "Then", "And" and "But" are all keywords. In our online store example we could write something like this:

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket

    Adding item to basket should not reload page

Scenario: Adding from item's description page

    Given that the "Alice in Wonderland book" is in online store
    And that item costs "50" USD
    And shipping costs of that item are "5" USD
    And in my basket are "0" items
```

(continues on next page)

(continued from previous page)

```
    And value of my basket is "0" USD
    And shipping costs are "0" USD

    When I visit an item description page
    And I click button "Add to basket"

    Then I see message "'Alice in Wonderland' has been added to basket"
    And I see "1" items in my basket
    And value of my basket become "50" USD
    And shipping costs become "5" USD
```

After "Given" you define what is expected starting state (item is in store; basked is empty). After "When" you write what <role> is performing (logged user visits item page and clicks on button). And after "Then" you write what is ending state (displayed message, one item in basket). "And" and "But" are used to write more actions and conditions in block "Given", "When", "Then".

1.4.4 Alternatives

If there are alternative paths to achieve feature goals you should write scenarios for them too:

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket

    Adding item to basket should not reload page

Scenario: Adding from item's description page

    Given that the "Alice in Wonderland book" is in online store
    And that item costs "50" USD
    And shipping costs of that item are "5" USD
    And in my basket are "0" items
    And value of my basket is "0" USD
    And shipping costs are "0" USD

    When I visit an item description page
    And I click button "Add to basket"

    Then I see message "'Alice in Wonderland' has been added to basket"
    And I see "1" items in my basket
    And value of my basket become "50" USD
    And shipping costs become "5" USD

Scenario: Adding item from search result page

    Given that the "Alice in Wonderland book" is in online store
    And that item costs "50" USD
    And shipping costs of each item is "5" USD
    And in my basket are "0" items
    And value of my basket is "0" USD
    And shipping costs are "0" USD

    When I visit a search page
```

(continues on next page)

(continued from previous page)

```
    And I enter "Alice in Wonderland" in search box
    And I click button "Search"
    And I see button "Add to basket" next to item "Alice in Wonderland"
    And I click button "Add to basket"

    Then I see message "'Alice in Wonderland' has been added to basket"
    And I see "1" items in my basket
    And value of my basket become "50" USD
    And shipping costs become "5" USD
```

1.4.5 Tables

Sometimes you want give more sample values to show expected behaviour. E.g. if value of basked exceed 100 USD you can give your customers discount on shipping costs. Let say that above 100 USD shipping costs are 0 USD. You can write second scenario for this case:

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket

    Adding item to basket should not reload page

Scenario: Adding from item's description page

    Given that the "Alice in Wonderland book" is in online store
    And that item costs "50" USD
    And shipping costs of that item are "5" USD
    And in my basket are "0" items
    And value of my basket is "0" USD
    And shipping costs are "0" USD

    When I visit an item description page
    And I click button "Add to basket"

    Then I see message "'Alice in Wonderland' has been added to basket"
    And I see "1" items in my basket
    And value of my basket become "50" USD
    And shipping costs become "5" USD

Scenario: Adding from item's description page without shipping costs

    Given that the "Alice in Wonderland book" is in online store
    And that item costs "50" USD
    And shipping costs of that item are "5" USD
    And in my basket are "2" items
    And value of my basket is "90" USD
    And shipping costs are "5" USD

    When I visit an item description page
    And I click button "Add to basket"

    Then I see message "'Alice in Wonderland' has been added to basket"
    And I see "3" items in my basket
```

(continues on next page)

(continued from previous page)

```
    And value of my basket become "140" USD
    And shipping costs become "0" USD

Scenario: Adding item from search result page
    # ...
```

When more such rules appear sometimes it's easier to make a table. Above example can be shortened:

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket

    Adding item to basket should not reload page

Scenario: Adding from item's description page

    Given that the "Alice in Wonderland book" is in online store
    And that item costs <cost> USD
    And shipping costs of each item is "5" USD
    And in my basket are <initial_items> items
    And value of my basket is <initial_value> USD
    And shipping costs are <initial_shipping> USD

    When I visit an item description page
    And I click button "Add to basket"

    Then I see message "'Alice in Wonderland' has been added to basket"
    And I see <items> items in my basket
    And value of my basket become <value> USD
    And shipping costs become <shipping> USD

        | cost | initial_items | initial_value | initial_shipping | items | value |␣
↪shipping |
        | 50   | 0             | 0             | 0                | 1     | 50    | 5␣
↪       |
        | 50   | 2             | 90            | 10               | 3     | 140   | 0␣
↪       |

Scenario: Adding item from search result page
    # ...
```

Names within <*angles*> will be replaced with values from rows in table. You can easily extend table for other special cases adding new rows.

1.4.6 Comments

When you want to add some comments inside features file, just write in a new line beginning with "#":

```
Feature: Add an item to basket

    In order to buy an item
    As a logged user
    I want to add items into basket
```

(continues on next page)

(continued from previous page)

```
    Adding item to basket should not reload page

Scenario: Adding from item's description page

    Given that the "Alice in Wonderland book" is in online store
    And that item costs <cost> USD
    And shipping costs of each item is "5" USD
    And in my basket are <initial_items> items
    And value of my basket is <initial_value> USD
    And shipping costs are <initial_shipping> USD

    When I visit an item description page
    # see description page mockup in file "description_page.jpg"
    And I click button "Add to basket"

    Then I see message "'Alice in Wonderland' has been added to basket"
    And I see <items> items in my basket
    And value of my basket become <value> USD
    And shipping costs become <shipping> USD

        | cost | initial_items | initial_value | initial_shipping | items | value |␣
↪shipping |
        | 50   | 0             | 0             | 0                | 1     | 50    | 5␣
↪        |
        | 50   | 2             | 90            | 10               | 3     | 140   | 0␣
↪        |
```

1.5 API

1.6 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.6.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/kidosoft/morelia/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

Morelia could always use more documentation, whether as part of the official Morelia docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/kidosoft/morelia/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.6.2 Get Started!

Ready to contribute? Here's how to set up *morelia* for local development.

1. Fork the *morelia* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/morelia.git
```

3. Assuming you have poetry installed (https://poetry.eustace.io/), this is how you set up your fork for local development with poetry:

```
$ cd morelia/
$ make develop      # creates virtualenv and installs dependencies with poetry
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass static analysis and the tests, including testing other Python versions with tox:

```
$ make test         # runs tests for development environment
$ make test-all     # runs tests for every supported environment with tox
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

1.6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5+ and for PyPy3. Check https://travis-ci.org/kidosoft/morelia/pull_requests and make sure that the tests pass for all supported Python versions.

1.7 Credits

1.7.1 First Author

- Phlip

1.7.2 Development Lead

- Jakub STOLARSKI (dryobates) <jakub.stolarski@gmail.com>

1.7.3 Contributors

- Iza STOLARSKA (izabeera)
- Michael SOUZA (midnighteuler)
- Hugo (hugovk)

1.8 Version History

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

1.8.1 Version: Unreleased

1.8.2 Version: 0.9.0 (2019-06-23)

ADDED

- setUpFeature/tearDownFeature, setUpScenario/tearDownScenario, setUpStep/tearDownStep (#105)

- hiding irrelevant morelia's code from tracebacks (#111)
- new recommended "verify" function with API simpler than in run (#118)
- configuration inside pyproject.toml (#119)

CHANGED

- dropped support for setUp/tearDown executed before/after each scenario (#105)

1.8.3 Version: 0.8.3 (2019-04-16)

FIXED

- attribute error when comment on first line after scenario (#116)

1.8.4 Version: 0.8.2 (2019-04-12)

FIXED

- incorrectly matching steps in languages other than english (#113)
- incorrect matching methods with parse format (#106).

1.8.5 Version: 0.8.1 (2019-04-11) [YANKED]

FIXED

- incorrect matching methods with parse format (#106).

1.8.6 Version: 0.8.0 (2019-02-09)

REMOVED

- dropped support for python 2.7 and 3.4

1.8.7 Version: 0.7.1 (2019-01-15)

ADDED

- support for running single scenarios (#16)
- support for passing string to morelia.run instead of external feature file

REMOVED

- dropped support for python 3.3

1.8.8 Version: 0.6.5 (2016-10-12)

CHANGED

- added new line before printing Feature

1.8.9 Version: 0.6.3 (2016-07-10)

CHANGED

- removed ReportVisitor
- removed branching on multiple When's

FIXED

- error while reporting missing steps

1.8.10 Version: 0.6.2 (2016-06-10)

FIXED

- incorrect handling labels inside steps

1.8.11 Version: 0.6.1 (2016-03-29)

FIXED

- regression in reporting unicode exceptions

1.8.12 Version: 0.6.0 (2016-03-28)

ADDED

- reporting on all failing scenarios

1.8.13 Version: 0.5.2 (2016-02-21)

CHANGED

- by default all missing steps are now shown

FIXED

- rendering issues in README.rst and docs

1.8.14 Version: 0.5.1 (2016-02-20)

FIXED

- bug with setUp/tearDown methods called twice
- bug with double run of background steps when show_all_missing=True

1.8.15 Version: 0.5.0 (2015-05-30)

ADDED

- labels in feature files
- tags decorator
- step's text payload

1.8.16 Version: 0.4.2 (2015-05-10)

FIXED

- bug with matching utf-8 docstrings with unicode predicate

1.8.17 Version: 0.4.1 (2015-05-07)

FIXED

- bug with comments support in scenarios with tables

1.8.18 Version: 0.4.0 (2015-04-26)

ADDED

- support for Background keyword
- support for different output formatters
- Examples keyword as no-op

CHANGED

- folding missing steps suggestions for more condense output

1.8.19 Version: 0.3.0 (2015-04-14)

ADDED

- support for matching methods by str.format-like ({name}) docstrings
- example project

CHANGED

- showing all missing steps instead of only first

1.8.20 Version: 0.2.1 (2015-04-06)

ADDED

- support for Python 3
- native language support

CHAPTER 2

Indices and tables

- genindex
- modindex
- search