radish Documentation

Release 0.17.1

Timo Furrer

Mar 20, 2024

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

radish is a *Behaviour Driven Development*-Tool completely written in python.

1.1 Why yet another python BDD tool?

In addition to the standard gherkin language features which almost every BDD tool tries to implement radish implements uncommon but useful features like Scenario Loops, Scenario Preconditions and Variables.

CHAPTER

TWO

INSTALLATION

radish is available as a Python 3 package on PyPI and thus installable with *pip*.

2.1 System Wide Installation

To install radish system wide use the following *pip* command:

```
pip install radish-bdd
```

Note: Make sure your user has enough privileges to install a package to the systems folders.

2.2 virtualenv Installation

To install radish in a *virtual python environment* use the following commands:

```
virtualenv radish-env -p python3
source radish-env/bin/activate
pip install radish-bdd
```

2.3 Install from source

To install radish from source you can clone the GitHub repository and use setuptools:

```
git clone https://github.com/radish-bdd/radish
cd radish
python setup.py install
```

CHAPTER

THREE

QUICKSTART

In this chapter we will write our first feature file and python step implementation. More detailed information about Feature files, Scenarios and Steps can be found in the Tutorial chapter.

3.1 Writing the first feature file

Let's assume we've written a really awesome calculator class and want to test it with radish. Our first feature file should test if the calculator is able to correctly sum numbers. Feature files are nothing more than a text file containing a *Feature* with one or more *Scenarios*. Each *Scenario* contains one or more *Steps*:

```
Feature: <My feature title>
    ... Some feature description ...

    Scenario: <My scenario title>
        ... Some steps testing our python code ...
```

To test our *calculator* we could write the following Feature and save it in a file called *features/SumNumbers.feature*:

```
Feature: The calculator should be able to sum numbers
    In order to make sure the calculator
    sums numbers correctly I have the following
    test scenarios:

    Scenario: Test my calculator
        Given I have the numbers 5 and 6
        When I sum them
        Then I expect the result to be 11
```

3.2 Implementing Steps

In order to run our *SumNumbers.feature* feature file we have to tell radish what to do for each Step in our Scenario.

All Steps are implemented in a python module as functions. These python modules are loaded by radish and the Step implementations are automatically matched with the corresponding Steps in the feature file.

Let's write our first feature file called *radish/steps.py*:

```
# -*- coding: utf-8 -*-
```

(continues on next page)

(continued from previous page)

```
from radish import given, when, then

@given("I have the numbers {number1:g} and {number2:g}")
def have_numbers(step, number1, number2):
    step.context.number1 = number1
    step.context.number2 = number2

@when("I sum them")
def sum_numbers(step):
    step.context.result = step.context.calculator.add( \
       step.context.number1, step.context.number2)

@then("I expect the result to be {result:g}")
def expect_result(step, result):
    assert step.context.result == result
```

Each of our Step implementation functions is decorated by radish's *given*, *when* or *then* decorator. The first argument of these decorators is a *regex-similar* expression. These expressions are used to match the Steps from the feature file. A Step can contain parameters which are parsed by radish and passed after to the step implementation function. The first argument of a step implementation function is always the step object itself. The most interesting part about the *step* object is the *step.context* object. This object represents a *Scenario* wide context with dynamic attributes. Our step implementation already uses this *context object* to store the numbers to sum and a *calculator* instance. This calculator instance is created in a hook in the so called *terrain file* module.

3.3 Implementation Terrain

In addition to the Step implementations is possible to implement *hooks* which are called during a run by radish. These hooks are usually implemented in a file called *terrain.py* alongside the step implementation modules. For our *calculator* tests we use the *radish/terrain.py* file to instantiate the calculator object:

```
# -*- coding: utf-8 -*-

from radish import before, after

from calculator import Calculator

@before.each_scenario
def init_calculator(scenario):
    scenario.context.calculator = Calculator(caching=True)

@after.each_scenario
def destory_calculator(scenario):
    del scenario.context.calculator
```

Yes, to be honest in this case it seems like an overkill to have this hooks implementation. Where it becomes really useful and handy are when database, external resources, etc. are involved.

3.4 Run the feature file

So far we've got the following files in our project:

```
features/
    SumNumbers.feature
radish/
    steps.py
    terrain.py
```

With this setup we can just execute the following command and radish will run our feature file:

```
radish features/
```

radish will output the following:

```
Feature: The calculator should be able to sum numbers  # features/SumNumbers.feature
    In order to make sure the calculator
    sums numbers correctly I have the following
    test scenarios:

    Scenario: Test my calculator
        Given I have the numbers 5 and 6
        When I sum them
        Then I expect the result to be 11

1 features (1 passed)
1 scenarios (1 passed)
3 steps (3 passed)
Run 1447487393 finished within 0:0.000436 minutes
```

How does radish find my python modules? radish imports all python modules inside the *basedir*. Per default the *basedir* points to *$PWD/radish* which in our case is perfectly fine. If the python implementation modules are located at another location the *-b* option followed by the path to the files can be given and radish will import the files from this location.

3.5 Run state result

Step:

A Step run state can be one of the following values.

- passed
- failed
- skipped
- pending
- untested

Scenario:

Scenario run state result is set set as follows:

If any Step in the Scenario is did not "pass" then return the run result of the **first** Step that did not pass. As such Scenario run state result is always one of the Step run state values described above.

Feature:

If any Scenario in the Feature is did not "pass" then return the run result of the **first** Step that did not pass. As such Feature run state result is always one of the Step run state values described above.

CHAPTER

FOUR

TUTORIAL

This chapter covers the whole Tutorial about radish and its features.

4.1 Feature files

All tests are written in so-called *feature files*. Feature files are plain text files ending with *.feature*. A feature file can contain only one BDD *Feature* written in a natural language format called Gherkin. However, radish is able to run one or more feature files. The feature files can be passed to radish as arguments:

```
radish features/
radish features/SumNumbers.feature features/DivideNumbers.feature
radish features/unit features/functional
```

4.2 Feature

A Feature is the main part of a *feature file*. Each feature file must contain exactly one *Feature*. This Feature should represent a test for a single feature in your software similar to a test class in your unit code tests. The *Feature* is composed of a *Feature sentence* and a *Feature description*. The feature sentence is a short precise explanation of the feature which is tested with this *Feature*. The feature description as a more verbose explanation of the feature which is tested. There you can answer the *Why* and *What* questions. A *Feature* has the following syntax:

```
Feature: <Feature sentence>
    ... Feature description
    on multiple lines ...
```

A Feature must contain one or more *Scenarios* which are run when this feature file is executed.

```
Feature: <Feature sentence>
    ... Feature description
    on multiple lines ...

    Scenario: <Scenario 1 sentence>
        ... Steps ...

    Scenario: <Scenario 2 sentence>
        ... Steps ...
```

4.3 Scenario

A Scenario is located inside a *Feature*. You can think of a Scenario as of a standalone test case for the feature you want to test. A Scenario contains one or more Steps. Each Scenario must have a unique sentence inside a *Feature*.

```
Feature: My Awesome Feature
    In order to document
    radish I write this feature.

    Scenario: Test feature
        ... Some Steps ...

    Scenario: Test feature with a bad case test
        ... Some Steps ...
```

4.4 Scenario Outline

A Scenario Outline is a more advanced version of a standard Scenario. It allows you to run a Scenario multiple times with different input values. A *Scenario Outline* is defined with *Examples*. The Scenario is run with the input data from each *Example*. The data from the Example can be accessed in a Scenario with the name of the data inside < and >. For example see the following *Scenario Outline* which divides multiple numbers from the *Examples*:

```
Feature: Test dividing numbers
    In order to test the
    Scenario Outline features of
    radish I test dividing numbers.

    Scenario Outline: Divide Numbers
        Given I have the number <number1>
        And I have the number <number2>
        When I divide them
        Then I expect the result to be <result>

    Examples:
        | number1 | number2 | result |
        | 10      | 2       | 5      |
        | 6       | 3       | 2      |
        | 24      | 8       | 3      |
```

Note: a PIPE (|) character within a *Examples* cell can be escaped with a backslash ().

Scenario Outlines can also be use within the step text. An example is shown in the following *Scenario Outline*:

```
Feature: Test dividing numbers
     with using the step text in to test the
     Scenario Outline features of
     radish I test dividing numbers.

     Scenario Outline: Divide Numbers
       Given I have following numbers
       """
```

(continues on next page)

(continued from previous page)

```
        n1:<number1>,
        n2:<number2>
      """
      When I divide them
      Then I expect the result
      """
        result:<result>
      """

    Examples:
      | number1 | number2 | result |
      | 10      | 2       | 5      |
      | 6       | 3       | 2      |
      | 24      | 8       | 3      |
```

4.5 Scenario Loop

A Scenario Loop is a standard Scenario which is repeated for a given amount of iterations. *Scenario Loops* can often be useful when stabilization tests are performed in a CI environment. Scenario Loops have the following syntax:

```
Feature: My Awesome Feature
    In order to document
    radish I write this feature.

    Scenario Loop 10: Some stabilization test
        ... Some Steps ...
```

Note: Scenario Loops are not standard gherkin

4.6 Scenario Precondition

Sometimes it can be very useful to reuse specific Scenarios. That's why we've decided to implement *Scenario Preconditions* in radish even though it's not common for a BDD tool. Before you start using *Scenario Preconditions* you should really think about the reason why you are using it. Behavior Driven Development Scenarios should be as short and concise as possible without a long list of dependencies. But there will always be these edge cases where it really makes sense to have a precondition for your Scenario. Every Scenario can be used as a Precondition Scenario. *Scenario Preconditions* are implemented as special tags:

```
Feature: My Awesome Feature
    In order to document
    radish I write this feature.

    @precondition(SomeFeature.feature: An awesome Scenario)
    Scenario: Do some crazy stuff
        When I add the following users to the database
            | Sheldon | Cooper |
        Then I expect to have 1 user in the database
```

radish will import the Scenario with the sentence `An awesome Scenario` from the feature file `SomeFeature.feature` and run it before the `Do some crazy stuff` Scenario. The following lines will be written:

```
Feature: My Awesome Feature
    In order to document
    radish I write this feature.

    @precondition(SomeFeature.feature: An awesome Scenario)
    Scenario: Do some crazy stuff
      As precondition from SomeFeature.feature: An awesome Scenario
        Given I setup the database
      From scenario
        When I add the following users to the database
            | Sheldon | Cooper |
        Then I expect to have 1 user in the database
```

As you can see radish will print some information about the Scenario where the Steps came from. radish supports *multiple* and *nested* Scenario Preconditions, too. Recursions are detected and radish will print an appropriate error message.

If you have preconditions in a Scenario it's inconvenient to send it to your colleague or post it somewhere because you have multiple files. radish is able to resolve all preconditions and expand them to a single file. Use the `radish show --expand` command to do so:

```
$ radish show --expand MyFeature.feature
Feature: My Awesome Feature
    In order to document
    radish I write this feature.

    #@precondition(SomeFeature.feature: An awesome Scenario)
    Scenario: Do some crazy stuff
        Given I setup the database
        When I add the following users to the database
            | Sheldon | Cooper |
        Then I expect to have 1 user in the database
```

The information about the precondition is commented out.

Note: Scenario Preconditions are not standard gherkin

4.7 Background

A *Background* is a special case of the Scenario. It's used to add some context to each Scenario of the same Feature. You can think of it as a setup Scenario for the other Scenarios. It consists of one or more Steps in exactly the same way as regular Scenarios. The *Background* is run **after** the *before hooks* of each Scenario but **before** the *Steps* of this Scenario.

A Background consists of an optional *short description* and Steps:

```
Background: [optional short description]
    [zero or more Steps]
```

A simple Background might look like this:

```
Feature: Calculator Addition
    In order to support all four elementary
    binary operations the calculator shall
    implement the binary addition operator.

    Background:
        Given the calculator is started

    Scenario: Adding two positive integers
        Given the integer 5
        And the integer 2
        When the integers are added
        Then the sum is 7
```

Cucumber defined some useful good practices for using backgrounds. It's worth to read them carefully.

4.8 Steps

The steps are the heart piece of every Feature file. A line in a *Scenario* is called *Step*. The steps are the only thing which are really executed in a test. A Step is written in a human readable language. Each step is parsed by radish and matched with a step implementation written in Python. If a Step does not match any step implementation radish will raise an exception and abort the run.

All steps are implemented in Python files located inside the *radish basedirs*. Per default this base directory points to *$PWD/radish*. However, the base directory location can be changed by specifying the *-b* option when triggering radish. You can also specify `-b` multiple times to load from multiple locations. There are several ways how to implement steps. The most common way is by decorating your step implementation functions with one of the following decorators:

- @step(pattern)
- @given(pattern)
- @when(pattern)
- @then(pattern)

The difference between those four decorators is that for the *given*, *when* and *then* decorator the corresponding keyword is prefixed. For example `@given("I have the number")` becomes the pattern `Given I have the number`.

A basic *steps.py* file with some step implementations could look like the following:

```
from radish import given, when, then

@given("I have the number {number:g}")
def have_number(step, number):
    step.context.numbers.append(number)


@when("I sum them")
def sum_numbres(step):
    step.context.result = sum(step.context.numbers)


@then("I expect the result to be {result:g}")
```

(continues on next page)

(continued from previous page)

```
def expect_result(step, result):
    assert step.context.result == result
```

The first example of a *step implementation function* is always an object of type `Step`.

Another way to implement step functions is using an enitre class:

```
from radish import steps

@steps
class Calculator(object):
    def have_number(self, step, number):
        """I have the number {number:g}"""
        step.context.numbers.append(number)

    def sum_numbres(self, step):
        """I sum them"""
        step.context.result = sum(step.context.numbers)

    def expect_result(self, step, result):
        """I expect the result to be {result:g}"""
        assert step.context.result == result
```

With the `@steps` decorator all methods of the given class are registered as steps. The step pattern is always the first line of the docstring of each method. If a method inside the call is not a step implementation you can add the method name to the `ignore` attribute of this class:

```
from radish import steps

@steps
class Calculator(object):

    ignore = ["validate_number"]

    def validate_number(self, number):
        """Validate the given number"""
        ...

    def have_number(self, step, number):
        """I have the number {number:g}"""
        self.validate_number(number)
        step.context.numbers.append(number)
```

4.9 Step Pattern

The pattern for each *Step* can be defined in two ways. The default way is to specify the *Step pattern* in a format similar to the one used by Python's `str.format()` method - but in the opposite way. radish uses parse_type to parse this pattern. The pattern can be a simple string:

```
@given("I sum all my numbers")
...
```

This *Step pattern* doesn't have any arguments. To specify arguments use the `{NAME:TYPE}` format:

```
@given("I have the number {number:g}")
def have_number(step, number):
    ...
```

The argument will be passed as keyword argument to the step implementation function with the specified name. If no name is specified the arguments are positional:

```
@given("I have the numbers {:g} and {:g}")
def have_numbers(step, number1, number2):
    ...
```

Per default the following *types* are supported:

Type	Characters matched	Output type
w	Letters and underscore	str
W	Non-letter and underscore	str
s	Whitespace	str
S	Non-whitespace	str
d	Digits (effectively integer numbers)	int
D	Non-digit	str
n	Numbers with thousands separators (, or .)	int
%	Percentage (converted to value/100.0)	float
f	Fixed-point numbers	float
e	Floating-point numbers with exponent e.g. 1.1e-10, NAN (all case insensitive)	float
g	General number format (either d, f or e)	float
b	Binary numbers	int
o	Octal numbers	int
x	Hexadecimal numbers (lower and upper case)	int
ti	ISO 8601 format date/time e.g. 1972-01-20T10:21:36Z ("T" and "Z" optional)	datetime
te	RFC2822 e-mail format date/time e.g. Mon, 20 Jan 1972 10:21:36 1000	datetime
tg	Global (day/month) format date/time e.g. 20/1/1972 10:21:36 AM 1:00	datetime
ta	US (month/day) format date/time e.g. 1/20/1972 10:21:36 PM 10:30	datetime
tc	ctime() format date/time e.g. Sun Sep 16 01:03:52 1973	datetime
th	HTTP log format date/time e.g. 21/Nov/2011:00:07:11 +0000	datetime
ts	Linux system log format date/time e.g. Nov 9 03:37:44	datetime
tt	Time e.g. 10:21:36 PM -5:30	time
MathExpression	Mathematic expression containing: [0-9 +-*/%.e]+	float
QuotedString	String inside double quotes ("). Double quotes inside the string can be escaped with a backslash	text w/o quotes
Boolean	Boolean value: True: 1, y, Y, yes, Yes, YES, true, True, TRUE, on, On, ON False: 0, n, N, no, No, NO, false, False, FALSE, off, Off, OFF	bool

These standard types (MathExpression, QuotedString and Boolean) can be combined with the following cardinalities:

```
"{numbers:d}"     #< Cardinality: 1    (one; the normal case)
"{number:d?}"     #< Cardinality: 0..1 (zero or one  = optional)
"{numbers:d*}"    #< Cardinality: 0..* (zero or more = many0)
"{numbers:d+}"    #< Cardinality: 1..* (one  or more = many)
```

If you accept one or more Boolean for your step you could therefor do:

```
@given('I have the flags {flags:Boolean+}')
def have_flags(step, flags)
    ...
```

By default the , (comma) is used as a separator, but you are able to specify your own. Let's assume you want to use and instead of ,:

```
from radish import custom_type, register_custom_type, TypeBuilder

@custom_type('Number', r'\d+')
def parse_number(text):
    return int(text)
```

(continues on next page)

(continued from previous page)

```
# register the NumberList type
register_custom_type(NumberList=TypeBuilder.with_many(
    parse_number, listsep='and'))
```

Now you can use `NumberList` as the type in your step pattern. As of now (Mar-2024) parse does not support cardinality, if cardinality is required a custom type needs to be created or the folling issue needs to be adressed: https://github.com/r1chardj0n3s/parse/issues/181

As you've seen you can use the `custom_type` decorator, the `register_custom_type` function and the `TypeBuilder` to extend the default types. This could be useful to directly inject more advanced objects to the step implementations:

```
from radish import custom_type

@custom_type("User", r"[A-Z][a-z]+ [A-Z][a-z]+")
def user_type(text):
    """
     Return a user object by the given name
    """
    if text not in world.database.users:  # no user found
        return None

    return world.database.users[text]
```

This *custom type* can be used like this in the *Step pattern*:

```
from radish import then

@then("I expect the user {user:User} has the email {}")
def expect_user_has_email(step, user, expected_email):
    assert user.email == expected_email, "User has email '{0}'.
       Expected was email '{1}'".format(user.email, expected_email)
```

The `TypeBuilder` provides the following functionality:

: TypeBuilder.with_many(func[,listsep=',']) :
Extend the given parse function to accept multiple values of `func`. See: https://github.com/jenisys/parse_type#cardinality

: TypeBuilder.with_optional(func) :
Make the string parsed by `func` optional. See: https://github.com/jenisys/parse_type#cardinality

: TypeBuilder.make_enum(enum : dict) :
Create a type for an enum represented by a `dict`. See: https://github.com/jenisys/parse_type#enumeration-name-to-value-mapping

: TypeBuilder.make_choice(choices : list) :
Create a type which accepts the values in the given list See: https://github.com/jenisys/parse_type#choice-name-enumeration

: TypeBuilder.make_variant(variants: list) :
Create a type which can be one of the given types See: https://github.com/jenisys/parse_type#variant-type-alternatives

If these *Step patterns* do not fit all your use cases you could use your own **Regular Expression** to match a *Step sentence*:

```
from radish import then

@then(re.compile(r"I expect the user ([A-Z][a-z]+ [A-Z][a-z]+|PENNY&LEONARD)+"))
def complex_stuff(step, user):
    ...
```

The groups matched by the *Regular Expression* are passed to the *step implementation function*.

4.10 Step Behave like

Sometimes it could be useful to call another step within a step. For example it could be useful if you want to change the interface but still support the old steps or if you want to combine multiple steps in one step. This feature is called *behave like* and you can use it as the following:

```
@step("I want to setup the database")
def setup_database(step):
    step.behave_like("I start the database server")
    step.behave_like("I add the system users to the database")
    step.behave_like("I add all roles to the database")
```

4.11 Step Tables

Step Tables are used to provide table-like data to a Step. The *Step Table* syntax looks similar to the *Scenario Outline Examples*:

```
...
Scenario: Check database
    Given I have the following users
       | forename  | lastname | nickname  |
       | Peter     | Parker   | Spiderman |
       | Bruce     | Wayne    | Batman    |
    When I add them to the database
    Then I expect 2 users in the database
```

The *Step Table* can be accessed in the *Step Implementation function* through the `step.table` attribute which is a `list` of `dict`:

```
from radish import given, when, then

@given("I have the following users")
def have_number(step):
  step.context.users = step.table


@when("I add them to the database")
def sum_numbres(step):
  for user in step.context.users:
      step.context.database.users.add(forename=user['firstname'], \
          lastname=user['lastname'], nickname=user['nickname'])
```

(continues on next page)

(continued from previous page)

```
@then("I expect {number:g} users in the database")
def expect_result(step, number):
  assert len(step.context.database.users) == number
```

4.12 Step Text data

Like the *Step Tables* a Step can also get an arbitrary text block as input. The syntax to pass text data to a *Step* looks like this:

```
...
Scenario: Test quote system
   Given I have the following quote
       """
       To be or not to be
       """
   When I add it to the database
   Then I expect 1 quotes in the database
```

To access this text data you can use the `text` attribute on the `step` object:

```
from radish import given, when, then

@given("I have the following quote")
def have_quote(step):
    step.context.quote = step.text


@when("I add it to the database")
def add_quote_to_db(step):
        step.context.database.quotes.append(step.context.quote)


@then("I expect {number:g} quote in the database")
def expect_amount_of_quotes(step, number):
    assert len(step.context.database.quotes) == number
```

Note: Variables from a Scenario Outline are replaced in the step text.

4.13 Skipping a Step

In some situations it might be required to skip a step under certain conditions. For e.g. ;

```
...
Scenario: Test quote system
  Given I have the following quote in target DB
      """
      To be or not to be
```

(continues on next page)

(continued from previous page)

```
    """
  When I found 2 quotes in the DB
  Then I delete one of them
```

To skip the step if *To be or not to be* quote could not be found:

```
from radish import given, when, then

@given("I have the following quote in target DB")
def have_quote_in_target_db(step):

    # code that would check the query in the DB

    if query is None:
         step.skip()
         return

    # Assuming this query includes data that we fetched from DB.
    # which might be a list of dictionaries.
    step.context.result = query


@when("I found {number:g} quotes in the DB")
def found_n_quotes_in_the_db(step, number):
     if not hasattr(step.context, "result"):
         step.skip()

     assert len(step.context.result) == number

     step.context.database.delete_id = step.context.result[0]['id']

@then("I expect {number:g} quote in the database")
def expect_amount_of_quotes(step, number):
    if not hasattr(step.context, "result"):
        step.skip()

    assert an_internal_function_to_delete_db_row(step.context.database.delete_id) is True
```

4.14 Tags

Tags are a way to group or classify Features and Scenarios. Radish is able to only run Features or Scenarios with specific Tags. Tags are declared with a similar syntax as decorators in Python:

```
@regression
Feature: Some important feature
   In order to demonstrate
   the Tag feature in radish
   I write this feature.

   @good_case
```

(continues on next page)

(continued from previous page)

```
  Scenario: Some good case test
      ...

  @bad_case
  Scenario: Some bad case test
      ...
```

Note: a Scenario inherits all tags of the Feature it is defined in!

Tags can also be used for additional meta data.

```
@author mario @date Sat, 12 Aug 2023 18:41:23 +0200
@reviewer luigi
Feature: Some important feature
   In order to demonstrate
   the Tag feature in radish
   I write this feature.
```

When triggering radish you can pass the `--tags` command line option followed by a tag expression. Tag expressions are parsed with tag-expressions. Only these Features/Scenarios are ran.

Run all regression tests:

```
radish features/ --tags regression
```

Run all *good case* or *bad case* tests:

```
radish features/ --tags 'good_case or bad_case'
```

Run all tags with some argument, example: find all tagged as authored by tuxtimo

```
radish features/ --tags 'author(tuxtimo)'
```

Tags with argument will have the argument inside open and closing parenthesis. *@tag(value)* and *@tag value* are the same tag and can be filtered as *tag(value)*. Tags values with spaces are not supported/behave unexpected! Only the part before the first space is used for filtering. *@tag(value)* and *@tag(value 1)* are the same tag for filtering and both will be matched with *tag(value)*.

4.15 Constants

Constants are specific *Tags* which define a constant which can be used in the *Steps*. This could be useful when you have values which are used in several points in a Feature and which should be named instead of appear as magic numbers. A sample use-case I've seen is specifying a base temperature:

```
@constant(base_temperature: 70)
Feature: Test heater
   In order to test my
   heater system I write
   the following scenarios.

   Scenario: Test increasing the temperature
       Given I have the room temperature ${base_temperature}
```

(continues on next page)

(continued from previous page)

```
    When I increase the temperature about 5 degrees
    Then I expect the temperature to be ${base_temperature} + 5
```

Note: Constants are not standard gherkin

4.16 Terrain and Hooks

In addition to the Step implementation radish provides the possibility to implement Hooks. These Hooks are usually placed in a file called *terrain.py* inside the *base directory*. Hooks can be used to setup and tear down the Features, Scenarios or Steps. There are two different Hook types:

- `before`
- `after`

These can be combined with the following Hook subjects:

- `all`
- `each_feature`
- `each_scenario`
- `each_step`

Hooks can be registered by adding these Hook types and subjects as decorators to Python functions:

```
from radish import before

from database import Database

@before.each_scenario
def connect_database(scenario):
    scenario.context.database = Database(name="foobar")
    scenario.context.database.connect()
```

The Python functions must accept the respective model object and in the case of `all` a second argument which is the radish run marker (a unique run id):

```
from radish import after

@after.all
def cleanup(features, marker):
    os.remove('foo')
```

4.16.1 Ordered Hooks

Sometimes it can be useful to explicitly order your Hooks instead of relying on the registration order. Each Hook accepts an optional `order: int` keyword argument. The Hooks are called in ascending order for all `before` Hooks and in descending order for all `after` Hooks. So for example the following hooks:

```
from radish import before, after

@before.each_step(order=2)
def before_second(step):
    """Will be called as second before hook for each step"""
    print("BEFORE: 2nd")

@after.each_step(order=2)
def after_second(step):
    """Will be called as second after hook for each step"""
    print("AFTER: 2nd")

@before.each_step(order=1)
def before_first(step):
    """Will be called as first before hook for each step"""
    print("BEFORE: 1st")

@after.each_step(order=1)
def after_first(step):
    """Will be called as first after hook for each step"""
    print("AFTER: 1st")
```

would yield the following output:

```
BEFORE: 1st
BEFORE: 2nd
AFTER: 2nd
AFTER: 1st
```

The default order is `100` for every Hook and so the order depends on the registration order of the Hook which corresponds to the import and source code order.

4.16.2 Tagged Hooks

If you are using *Tags* you can specify that a certain Hook is only called for Features, Scenarios or Steps with the according tags.

```
from radish import after

@after.each_scenario(on_tags='bad_case or crash')
def cleanup(scenario):
    # do some heavy cleanup!
    pass
```

4.17 Contexts

As you may have noticed: each Feature and Scenario has it's own context. You can dynamically add attributes to this context. All Steps in a Scenario have the same context. This is the preferred way to share data between steps over the `world` object.

```
from radish import before, given

@given("I have the number {number:g}")
def have_number(step, number):
   # accessing Scenario specific context
   step.context.number = number


@before.each_feature
def setup(feature):
    # accessing Feature specific context
    feature.context.setup = True
```

4.18 World

The `world` is a "global" radish context. It is used by radish to store the configuration and other utility functions. It can be accessed by importing it from the `radish`. The `world` object is a *threadlocal* object so it is safe to use in threads.

You should not be using `world` to store data in scenarios and steps, that is what *Contexts* are for.

The `config` attribute of world `world` contains a `Configuration` object with named and positional arguments passed into radish. A basic transformation is applied to each of the arguments to turn it into a python attribute: As such "-" is replaced with "_", "–" is removed, and "<" and ">" characters are removed.

For example `--bdd-xml` argument can be accessed using `world.config.bdd_xml`, and the argument `<features>` are accesses as `world.config.features`.

```
from radish import world

# print basedir
print(world.config.basedir)

# print profile
print(world.config.profile)
```

Sometimes it's useful to have specific variables and functions available during a whole test run. These variables and functions can be added to the `world` object:

```
from radish import world, pick
import random

world.x = 42

@pick
def get_magic_number():
    return random.randint(1, world.x)
```

The `pick` decorator adds the decorated function to the `world` object. You can use this function later in a step implementation or another hook:

```
from radish import before, world

from security import Tokenizer

@before.each_scenario
def gen_token(scenario):
   scenario.context.token = Tokenizer(world.get_magic_number())
```

4.19 BDD XML Report

Radish can report in the BDD XML format using `--bdb-xml`. The format of the XML is defined as follows:

XML declaration

```
<?xml version='1.0' encoding='utf-8'?>
```

<testrun> is a top level tag

agent
: Agent of the test run composed of the user and hostname of the machine. Format: user@hostname

duration
: Duration of test run in seconds rounded to the 10 decimal points.

starttime
: Start time of the testrun run. Format: combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

endtime
: End time of the testrun run. Format: combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

example:

```
<testrun>
  agent="user@computer"
  duration="0.0005660000"
  starttime="2017-02-18T07:06:55">
  endtime="2017-02-18T07:06:56"
>
```

The **<testrun>** contains the following tags

<feature> tag

id
: Test run index id of the Feature. First feature to run is 1, second is 2 and so on.

sentence
: Feature sentence.

result
: Run state result of Feature run as described in *Run state result*

testfile
Path to the file name containing the feature. The path is relative to the `basedir`.

duration
Duration of Feature run in seconds rounded to the 10 decimal points.

starttime
Start time of the Feature run. Format: combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

endtime
End time of the Feature run. Format: combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

example:

```
<feature
  id="1"
  sentence="Step Parameters (tutorial03)"
  result="failed"
  testfile="./example.feature"
  duration="0.0008730000"
  starttime="2017-02-18T07:06:55"
  endtime="2017-02-18T07:06:55"
>
```

The **<feature>** tag contains the following tags:

<description> tag:

tag content
CDATA enclosed description of the feature.

```
<description>
  <![CDATA[This feature test following functionality
  - awesomeness
  - more awesomeness
  ]]>
</description>
```

<scenarios> tag:

Contains list of **<scenarios>** tags

example:

```
<scenarios>
```

The **<scenarios>** tag contains the following tags:

<scenario> tag:

id
Test run index id of the Scenario. First scenario to run is 1, second is 2 and so on.

sentence
Scenario sentence.

result
Run state result of Scenario run as described in *Run state result*

testfile
Path to the file name containing the Scenario. The path is relative to the `basedir`.

duration
Duration of Scenario run in seconds rounded to the 10 decimal points.

starttime
Start time of the Scenario run. Format: combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

endtime
End time of the Scenario run. Combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

example:

```
<scenario
  id="1"
  sentence="Blenders"
  result="failed"
  testfile="./example.feature"
  duration="0.0007430000"
  endtime="2017-02-18T07:06:55"
  starttime="2017-02-18T07:06:55"
>
```

The **<scenario>** tag contains the following tags:

<step> tag:

id
Test run index id of the Step. First Step to run is 1, second is 2 and so on.

sentence
Step sentence.

result
Run state result of Step run as described in *Run state result*

testfile
Path to the file name containing the Step. The path is relative to the `basedir`.

duration
Duration of Step run in seconds rounded to the 10 decimal points.

starttime
Start time of the Step run. Format: combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

endtime
End time of the Step run. Format: combined date and time representations, where date and time is separated by letter "T". Format: YYYY-MM-DDTHH:MM:SS

example:

```
<step
  id="1"
  sentence="Given I put &amp;quot;apples&amp;quot; in a blender"
  result="passed"
  testfile="./example.feature"
```

(continues on next page)

(continued from previous page)

```
  duration="0.0007430000"
  endtime="2017-02-18T07:06:55"
  starttime="2017-02-18T07:06:55"
>
```

The **<step>** MAY tag contains the following tags if error has occurred:

<failure> tag:

message
: Test run index id of the Step. First Step to run is 1, second is 2 and so on.

type
: Step sentence.

tag content
: CDATA enclosed failure reason specifically exception traceback.

example:

```
<failure message="hello" type="Exception">
   <![CDATA[Traceback (most recent call last):
     File "/tmp/bdd/_env36/lib/python3.6/site-packages/radish/stepmodel.py", line 91, in␣
↪run
         self.definition_func(self, \*self.arguments)  # pylint: disable=not-callable
     File "/tmp/bdd/radish/radish/example.py", line 34, in step_when_switch_blender_on
       raise Exception("show off radish error handling")
   Exception: show off radish error handling
  ]]>
</failure>
```

4.20 Cucumber json Report

Radish can write cucumber json result file after run using –cucumber-json=<ccjson>.

With local tools like Cucumber json report generator

> java -jar cucumber-sandwich.jar -n -f path/to/the/folder/containing/json -o path/to/folder/to/generate/reports/into

Or Jenkins Cucumber Reports Plugin

You can simply generate Pretty HTML Reports for Cucumber

4.20.1 Embedding data in cucumber report

With radish it is simple to enrich your reports with additional text, html or image data

Here are few code examples:

```
@then("I put some text to my report")
def put_text(step):
    step.embed("This text goes into the report")
```

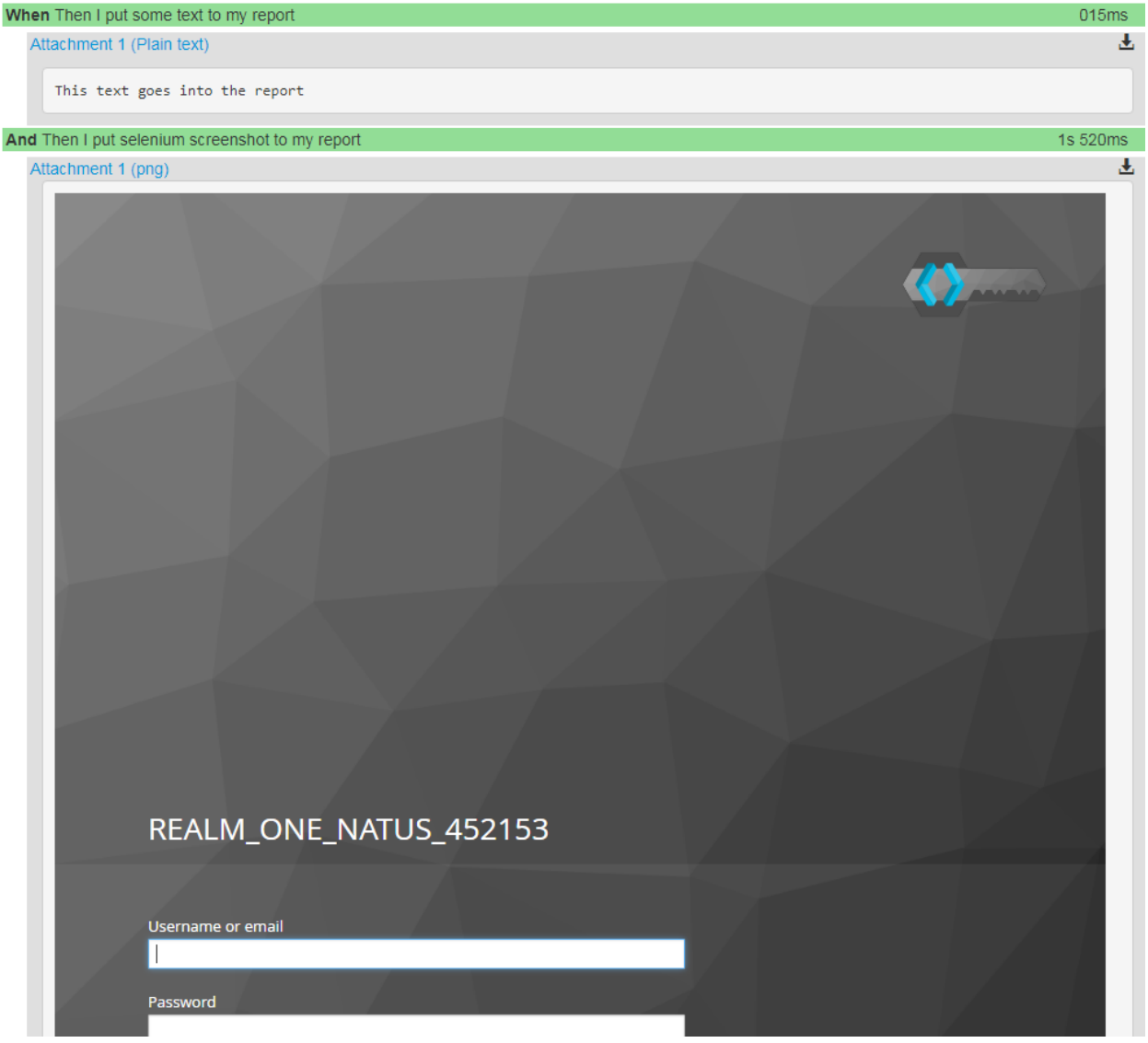
(continues on next page)

(continued from previous page)

```
@then("I put selenium screenshot to my report")
def put_selenium_screenshot(step):
    step.embed(context.web_driver.get_screenshot_as_base64(),
               mime_type='image/png',
               encode_data_to_base64=False)

@then("I put page source to my report")
def put_selenium_page_source(step):
    step.embed(context.web_driver.page_source,
               mime_type='text/html')
```

Html report output screen example:



4.21 Testing Step Patterns

New since radish version v0.3.0

Radish provides a nice way to test if the implemented step pattern (`@step(...)`) match the expected sentences. This is especially useful if you provide a set of step implementations and someone else is going to use them and implement the feature files.

In a way your step pattern are the interface of your step implementation and **interfaces ought to be tested properly**.

If you've installed radish a command called `radish-test` is available. Install it's dependencies with:

```
pip install radish-bdd[testing]
```

The `matches` sub command is used to test your step pattern inside your *base dirs* (`-b` / `--basedir`) against some sentences defined in a YAML file. We call those files **match configs**. A *match config* file has the following format:

```
- sentence: <SOME STEP SENTENCE>
  should_match: <THE STEP FUNCTION NAME IT SHOULD MATCH>
  should_not_match: <THE STEP FUNCTION NAME IT SHOULD NOT MATCH>
  with_arguments:
    # argument check if implicit type
    - <ARGUMENT 1 NAME>: <ARGUMENT 1 VALUE>
    # argument check with explicit type
    - <ARGUMENT 2 NAME>:
        type: <ARGUMENT 2 TYPE NAME>
        value: <ARGUMENT 2 VALUE>
    # argument check with explicit type and type cast
    - <ARGUMENT 3 NAME>:
        type: <ARGUMENT 3 TYPE NAME>
        value: <ARGUMENT 3 VALUE>
        cast: yes
    # argument check with explicit type and using repr() for the value
    - <ARGUMENT 4 NAME>:
        type: <ARGUMENT 4 TYPE NAME>
        value: <ARGUMENT 4 VALUE>
        use_repr: yes
```

sentence
: **Required**. This is the sentence you want to test. It's an example of a sentence which should match a certain Step pattern.

should_match
: **Required if should_not_match omitted**. This is the name of the Python Step implementation function which you expect the sentence will match with.

should_not_match
: **Required if should_match omitted**. This is the name of a Python Step implementation function which you expect the sentence will **not** match with.

with_arguments
: **Optional for should_match**. This is a list of arguments which you expect will be passed in the Python Step implementation function. The arguments can be specified as key-value pairs or as an object with a *type* and *value* and a boolean value *cast* and a *use_repr* flag. This could be useful if a custom argument expression is used to parse the arguments. The *use_repr* flag should be used when comparing with a user defined type.

4.21.1 Example

Let's assume we have the following `step.py` implementation:

```
from radish.stepregistry import step
from radish import given, when, then

@step("I have the number {number:g}")
def have_number(step, number):
    step.context.numbers.append(number)


@when("I sum them")
def sum_numbers(step):
    step.context.result = sum(step.context.numbers)


@then("I expect the result to be {result:g}")
def expect_result(step, result):
    assert step.context.result == result
```

And a `step-matches.yml` file like this:

```
- sentence: Given I have the number 5
  should_match: have_number
  with_arguments:
      - number:
            type: float
            value: 5.0

- sentence: When I sum them
  should_match: sum_numbers

- sentence: When I divide them
  should_not_match: sum_numbers

- sentence: When I do some weird stuff
  # if no step is given it shouldn't match any at all
  should_not_match:

- sentence: Then I expect the result to be 8
  should_match: expect_result
  with_arguments:
      - result: 8.0
```

We can check the `step.py` implementation against the `step-matches.yml` match config file using the `radish-test` CLI application:

```
radish-test matches tests/step-matches.yml
```

Due to the fact that the `step.py` module is located in `$PWD/radish` we don't have to specify it's location with `-b` or `--basedir`.

For the `radish-test` call above we would get the following output:

```
Testing sentences from tests/step-matches.yml:
>> STEP "Given I have the number 5" SHOULD MATCH have_number ✓
>> STEP "When I sum them" SHOULD MATCH sum_numbers ✓
>> STEP "When I divide them" SHOULD NOT MATCH sum_numbers ✓
>> STEP "Then I expect the result to be 8" SHOULD MATCH expect_result ✓

4 sentences (4 passed)
Covered 3 of 3 step implementations
```

In case of success we get the exit code **0** and in case of failure we'd get an exit code which is greater than **0**.

`radish-test matches` also supports step coverage measurements. Use `--cover-min-percentage` to let `radish-test matches` fail if a certain coverage threshold is not met and use the `--cover-show-missing` command line option to list all not covered steps and their location.

CHAPTER

FIVE

COMMAND LINE USAGE

This chapter describes how to use Radish from the command line. All it's commands, options and arguments.

5.1 Run - Specify Feature files

All arguments which do not belong to any command line option are interpreted as Feature files or Feature file locations. If the argument is a directory all files ending with `.feature` will be run. It's possible to mix files and directories:

```
radish SomeFeature.feature myfeatures/
```

5.2 Run - Specify base directory

Radish searches for and imports *Step* and *Terrain* python files in the `base directories` which by default is set to the `radish` folder inside the current working directory (a.k.a `$PWD/radish`). To specify an alternate path you may use the `-b` or `--basedir` command line option:

```
radish -b tests/radish SomeFeature.feature
radish --basedir tests/radish SomeFeature.feature
```

Since version v0.4.2 you can specify `-b` multiple times to import Python modules containing steps and terrain functions from multiple locations:

```
radish -b tests/radish -b custom/radish SomeFeature.feature
```

Since version v0.7.0 you can use multiple basedirs within one `-b` flag split by a colon (:). Similar to the possibilities you've got with `$PATH`. On Windows it is not possbile to use a colon (:) because it is used in almost any absolute path, e.g. `C:\foo\bar`. Since version v0.11.2 you can use a semicolon (;) on Windows for multiple basedirs.

5.3 Run - Early exit

By default Radish will try to run all specified Scenarios even if there are failed Scenarios during the run. If you want to abort the test run after the first error occurred you can use the `-e` or `--early-exit` option:

```
radish SomeFeature.feature -e
radish SomeFeature.feature --early-exit
```

5.4 Run - Debug Steps

Radish provides the ability to debug each step using a debugger. You can enable that using `--debug-steps` command line option.

```
radish --debug-steps SomeFeature.feature
```

The IPython debugger is used if present. If it isn't the standard Python debugger is used instead. Please consult the official debugger documentation for the common debugger workflow and commands.

For example you can list the variables available by printing `locals()`.

```
ipdb> locals()
{'step': <radish.stepmodel.Step object at 0x7f4d5b6ca400>}
```

As you can see, when a failure happens inside the Step you can see the step arguments such as `step`.

5.5 Run - Show traceback on failure

Radish can display a complete traceback in case a Step fails. You can use the `-t` or `--with-traceback` command line option for that:

```
radish SomeFeature.feature -t
radish SomeFeature.feature --with-traceback
```

5.6 Run - Use custom marker to uniquely identify test run

Radish supports marker functionality which is used to uniquely identify a specific test run. By default the marker is set to the number of seconds from the epoch (01/01/1970). You can specify your own marker using the `-m` or `--marker` command line option.

The marker is also displayed in the summary of a test run:

```
radish SomeFeature.feature -m "My Marker"
radish SomeFeature.feature --marker "My Marker"

... radish output

Run My Marker finished within 0:0.001272 minutes
```

The marker is also passed into all the hooks defined in the terrain files. To see example code please consult *terrain*.

5.7 Run - Profile

Radish allows you to pass custom data to a Terrain hook code or to the Step implementations using the `-p` or `--profile` command line option. This can be used to customize your test runs as needed.

The value specified to the `-p` / `--profile` command line option is made available in `world.config.profile`. Please see *World* for for an example.

A common usage of `profile` s setting it to some environment value such as `stage` or `production`.

```
radish SomeFeature.feature -p stage
radish SomeFeature.feature --profile stage
```

Note: `-p` / `--profile` is being deprecated and will be removed in a future version of Radish. Please use `-u` / `--user-data` instead. See *Arbitrary User Data* for details.

5.8 Run - Dry run

Radish allows you to pass custom flags to a Terrain hook code or to Step implementations using the `-d` or `--dry-run` command line option. This can be used to customize your test runs as needed.

The `-d` / `--dry-run` command line switch is made available in `world.config.dry_run` which is set to `True`. Please see *World* for an example.

```
radish SomeFeature.feature -d
radish SomeFeature.feature --dry-run
```

5.9 Run - Specifying Scenarios by id

Radish can also runs specific scenarios by id using the `-s` or `--scenarios` command line option. The ids are scenarios indexed by the parsing order. The first Scenario in the first Feature will have the id 1, the second scenario the id 2. The Scenario ids are unique within all Features from this run. The value can be a single Scenario id or a comma separated list of Scenario ids:

You can use `--write-ids` command line switch to print Scenario ids. Please consult *Run - Writing out Scenario and Step ids*

```
radish SomeFeature.feature -s 1
radish SomeFeature.feature --scenarios 1,2,5,6
```

5.10 Run - Shuffle Scenarios

Radish can also shuffle the Scenarios by using the `--shuffle` command line option. This is useful when you are trying to detect if any Scenario has unintended side effects on other Scenarios.

```
radish SomeFeature.feature --shuffle
```

5.11 Run - Specify certain Features and/or Scenarios by tags

Radish is able to run only a selection of certain Features and/or Scenarios using the `--tags` command line option. You can specify the tags of Features/Scenarios which should be run. The command line option value has to be a valid tag expression. Radish uses tag-expressions. The following are some valid tag expressions:

```
radish SomeFeature.feature --tags 'regression'
radish SomeFeature.feature --tags 'good_case and in_progress'
radish SomeFeature.feature --tags 'good_case'
radish SomeFeature.feature --tags 'regression and good_case and not real_hardware'
radish SomeFeature.feature --tags 'database or filesystem and bad_case'
radish SomeFeature.feature --tags 'author(tuxtimo)'
```

Be aware that Scenarios inherit the tags from the Feature they are defined it.

To learn how to tag Features and Scenarios please refer to *Tags* section.

5.12 Run - Work in progress

Radish is able change the state of the outcome. Scenarios which are still work in progress and are expected to fail, can be run with:

```
radish SomeFeature.feature --wip
```

To count as a success all Scenarios in this Feature need to fail. If a Scenario passes the run is failed. A suggested workflow is to tag WIP Scenarios with a *@wip* tag an run your tests twice.

```
radish SomeFeature.feature --wip --tags wip
radish SomeFeature.feature --wip --tags 'not wip'
```

5.13 Run - Write BDD XML result file

Radish can report it's test run results to a XML file after a test run using the `--bdd-xml` command line switch. The command line option value must be a file path where the XML file should be written to.

To write the XML file `lxml` is required. Install it with:

```
pip install radish-bdd[bddxml]
```

```
radish SomeFeature.feature --bdd-xml /tmp/result.xml
```

To understand the format BDD XML consult: *BDD XML Report*.

5.14 Run - Code Coverage

Radish can use the `coverage` package to measure code coverage of the code run during the tests using the `--with-coverage` command line option. You can also limit which packages it generates metrics for by providing file paths or package names using `--cover-packages`. The `--cover-packages` command line option is the `--source` command line switch used by `coverage`. See coverage documention

To use the code coverage feature you have to install the necessary extra dependencies with:

```
pip install radish-bdd[coverage]
```

The following options are also available to configure the coverage measurement and report:

–with-coverage
enables the coverage measurement

–cover-packages
specify one or more packages to measure. Multiple package names have to be separated with a comma.

–cover-append
append the coverage data to previously measured data.

–cover-config-file
specify a custom coverage config file. By default the `$PWD.coveragerc` file is read if it exists.

–cover-branches
include branch coverage into the measurement

–cover-erase
erase all previously collected coverage data

–cover-min-percentage
let the radish run file if the given coverage percentage is not reached

–cover-html
generate an HTML coverage report

–cover-xml
generate a XML coverage report

5.15 Run - Write Cucumber JSON file

Radish can report it's test run results to a Cucumber style JSON file after a test run using the `--cucumber-json` command line option. The command line option value must be a file path where the JSON file should be written to.

```
radish SomeFeature.feature --cucumber-json /tmp/result.json
```

Documentation describing the format of the Cucumber JSON file can be found here: https://www.relishapp.com/cucumber/cucumber/docs/formatters/json-output-formatter

5.16 Run - Write JUnit XML file

Radish can report it's test run results to a JUnit style XML file after a test run using the `--junit-xml` command line option. The command line option value must be a file path where the XML file should be written to.

```
radish SomeFeature.feature --junit-xml /tmp/result.xml
```

JUnit allows to add properties only to `testsuite` but tags on scenario level can be useful inside the matching `testcase`. This can be achieved using `--junit-relaxed`.

```
radish SomeFeature.feature --junit-relaxed /tmp/result.xml
```

5.17 Run - Log all features, scenarios, and steps to syslog

Radish provides the *–syslog* command line option which can be used to log all of your features, scenarios, and steps to the syslog. The caveat here is this option is only supported on systems where the Python standard library supports the system logger (syslog). This command line option works well in UNIX and UNIX-like systems (Linux) but will not work on Windows machines.

This can be especially useful for consolidating all of your logging data in one central repository.

```
radish SomeFeature.feature --syslog
```

If you are unfamiliar with the syslog feature, please consult the official syslog documentation.

5.18 Run - Debug code after failure

Radish debugging mechanisms include the ability to drop into either IPython debugger or the Python debugger on code failures using the `--debug-after-failure` command line option. Using IPython is preferred over the standard Python debugger.

If you are unfamiliar with the Python debugger please consult the official debugger documentation.

```
radish SomeFeature.feature --debug-after-failure
```

Please consult *Run - Debug Steps* for debugging tips.

5.19 Run - Inspect code after failure

Radish debugging mechanisms include the ability to drop into a IPython shell upon code failures using the `--inspect-after-failure` command line option.

To inspect code with IPython install the necessary extra dependencies with:

```
pip install radish-bdd[ipython-debugger]
```

```
radish SomeFeature.feature --inspect-after-failure
```

Please consult *Run - Debug Steps* for debugging tips.

5.20 Run - Printing results to console

Note: **Pending** state means "yet to be executed".

The Radish console output is aimed to be powerful and explicit. It uses ANSI color codes and line 'overwriting' to format and color the output to make it more user friendly.

The anatomy of the console output is a follows:

Executing Scenario Step sentences as well as entries in the Scenario Outline Example and Scenario Loop tables are printed to the console first, colored in bold yellow.

As the Scenario Steps, Scenario Outline Example entries and Scenario Loop iterations have finished the execution the "ANSI line jump" is used to replace the printed yellow lines with the outcome of the Step run which is colored in bold green on success or bold red in case of failure.

Exception messages and tracebacks are printed upon failure below the failed Step, Scenario Outline Example or Scenario Loop Iteration entry.

Radish provides several command line options to help you with console output format.

A common use of Radish is to run it using a script or in a continuous integration setup. Such setups usually do not support "ANSI" color codes or line jumps. This is where the combined use of `--no-ansi` and `--write-steps-once` command line options become handy.

The `--no-ansi` turns off every "ANSI" code which might make the output less readable in a non ANSI ready environment -> like Windows or when redirecting the output to a file. However, since doing that also disables line jumping the step runs will be printed twice to the screen (first print is the executing step, the second is the finished one). Without colors that double print is confusing and can be turned off using `--write-steps-once`.

```
radish SomeFeature.feature --no-ansi
radish SomeFeature.feature --no-ansi --write-steps-once
```

The `--no-line-jump` command line option disables the "overwriting" of the yellow executing lines by the success or failure lines. This is helpful when reviewing and debugging as it shows Steps first executing then finished. It also allows for "print to console" style debugging to be used without ANSI codes destroying them.

```
radish SomeFeature.feature --no-line-jump
```

5.21 Run - dots output formatter

By default the *gherkin* output formatter is used. This formatter prints the Features in a gherkin style. In most of the cases that's the same as the input Feature File content. This gherkin output formatter is rather verbose: all Features, Scenarios and Steps are printed.

You can use the *dots* output formatter with the *-f dots* command line option. Every passed Scenario will be printed as a dot (.). Other possible symbols are:

- *P* for *pending*
- *U* for *untested*
- *S* for *skipped*
- *F* for *failed*

If a Scenario has failed, the failed Step will be printed in the summary in the end:

```
$ radish SomeFeature.feature -f dots

features/SomeFeature.feature: ..FFF..

Failures:
features/SomeFeature.feature: Subtract numbers wrongly
    Then I expect the difference to be 3
      AttributeError: 'int' object has no attribute 'step'

features/SomeFeature.feature: A Scenario Outline - row 0
    Then I expect the sum to be 3
      AssertionError: The expected sum 3 does not match actual sum 11

features/SomeFeature.feature: A Scenario Outline - row 1
    Then I expect the sum to be 9
      AssertionError: The expected sum 9 does not match actual sum 17


1 features (0 passed, 1 failed)
7 scenarios (4 passed, 3 failed)
20 steps (17 passed, 3 failed)
Run 1545585467 finished within a moment
```

5.22 Run - Writing out Scenario and Step ids

Radish provides the *–write-ids* command line option which can be used to enumerate Scenarios and Steps.

This can be useful for bug reporting.

```
1. Scenario: Apple Blender
    1. Given I put couple of "apples" in a blender
    2. When I switch the blender on
    3. Then it should transform into "apple juice"

2. Scenario: Pear Blender
    1. Given I put couple of "pears" in a blender
    2. When I switch the blender on
    3. Then it should transform into "pear juice"
```

It can also be useful when using the `-s` / `--scenarios` command line option since the Scenarios are numbered in the run order.

5.23 Run - Specifying Arbitrary User Data on the command-line

Radish allows you to specify arbitrary user data on the command-line as `key=value` pairs. You can access the user data from your tests by accessing the `world.config.user_data` dictionary.

Note: All keys/values are treated as strings. If you specify the same key more than once, the last occurrence of the key will replace previous occurrences.

```
radish SomeFeature.feature --user-data="my_key=1" --user-data="my_key2=my_value2" -u "my-
↪key3=value3"
```

5.24 Show - Expand feature

Radish Precondition decorated Scenarios are powerful but can be confusing to read on the screen. For that Radish provides `--expand` command line option to expand all the preconditions.

```
radish show SomeFeature.feature --expand
```

5.25 Help Screen

Use the `--help` or `-h` option to show the following help screen:

```
Usage:
    radish show <features>
           [--expand]
           [--no-ansi]
    radish <features>...
           [-b=<basedir> | --basedir=<basedir>...]
           [-e | --early-exit]
           [--debug-steps]
           [-t | --with-traceback]
           [-m=<marker> | --marker=<marker>]
           [-p=<profile> | --profile=<profile>]
           [-d | --dry-run]
           [-s=<scenarios> | --scenarios=<scenarios>]
           [--shuffle]
           [--tags=<tags>]
           [--bdd-xml=<bddxml>]
           [--with-coverage]
           [--cover-packages=<cover_packages>]
           [--cover-append]
           [--cover-config-file=<cover_config_file>]
           [--cover-branches]
           [--cover-erase]
           [--cover-min-percentage=<cover_min_percentage>]
           [--cover-html=<cover_html_dir>]
           [--cover-xml=<cover_xml_file>]
           [--no-ansi]
           [--no-line-jump]
```

(continues on next page)

(continued from previous page)

```
           [--write-steps-once]
           [--write-ids]
           [--cucumber-json=<ccjson>]
           [--junit-xml=<junitxml>]
           [--debug-after-failure]
           [--inspect-after-failure]
           [--syslog]
           [-u=<userdata> | --user-data=<userdata>...]
    radish (-h | --help)
    radish (-v | --version)

Arguments:
    features                                    feature files to run

Options:
    -h --help                                   show this screen
    -v --version                                show version
    -e --early-exit                             stop the run after the first failed step
    --debug-steps                               debugs each step
    -t --with-traceback                         show the Exception traceback when a step␣
↪fails
    -m=<marker> --marker=<marker>               specify the marker for this run␣
↪[default: time.time()]
    -p=<profile> --profile=<profile>            specify the profile which can be used in␣
↪the step/hook implementation
    -b=<basedir> --basedir=<basedir>...         set base dir from where the step.py and␣
↪terrain.py will be loaded. [default: $PWD/radish]
                                                You can specify -b|--basedir multiple␣
↪times. All files will be imported.
    -d --dry-run                                make dry run for the given feature files
    -s=<scenarios> --scenarios=<scenarios>      only run the specified scenarios (comma␣
↪separated list)
    --shuffle                                   shuffle run order of features and␣
↪scenarios
    --tags=<feature_tags>                       only run Scenarios with the given tags
    --expand                                    expand the feature file (all␣
↪preconditions)
    --bdd-xml=<bddxml>                          write BDD XML result file after run
    --with-coverage                             enable code coverage
    --cover-packages=<cover_packages>           specify source code package
    --cover-append                              append coverage data to previous␣
↪collected data
    --cover-config-file=<cover_config_file>     specify coverage config file [default: .
↪coveragerc]
    --cover-branches                            include branch coverage in report
    --cover-erase                               erase previously collected coverage data
    --cover-min-percentage=<cover_min_percentage> fail if the given minimum coverage␣
↪percentage is not reached
    --cover-html=<cover_html_dir>               specify a directory where to store HTML␣
↪coverage report
    --cover-xml=<cover_xml_file>                specify a file where to store XML␣
↪coverage report
```

(continues on next page)

(continued from previous page)

```
    --no-ansi                                    print features without any ANSI␣
↪sequences (like colors, line jump)
    --no-line-jump                               print features without line jumps␣
↪(overwriting steps)
    --write-steps-once                           does not rewrite the steps (this option␣
↪only makes sense in combination with the --no-ansi flag)
    --write-ids                                  write the feature, scenario and step id␣
↪before the sentences
    --cucumber-json=<ccjson>                     write cucumber json result file after run
    --junit-xml=<junitxml>                       write JUnit XML result file after run
    --debug-after-failure                        start python debugger after failure
    --inspect-after-failure                      start python shell after failure
    --syslog                                     log all of your features, scenarios, and␣
↪steps to the syslog
    -u=<userdata> | --user-data=<userdata>...    User data as 'key=value' pair. You can␣
↪specify --user-data multiple times.
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search