Cricri Documentation

Release 1.4

Vincent Maillol

Jun 30, 2019

Contents

cricri is a test scenario generator. You define steps using TestState class. Each step has input method, set of tests methods, and one or multiple previous steps. cricri finds all paths in the steps and generates one Test Case for each path. You can disable test steps depending on the path traveled.

CHAPTER 1

Installation

You can install the latest stable release with pip:

```
pip install cricri
```

or install the latest development release (unstable) with git:

```
git clone https://github.com/Maillol/cricri.git cricri
cd cricri
python3 setup.py install
```

CHAPTER 2

Basic example

Here is a simple example with three steps to generate two scenario wich test list method

```
from cricri import TestState, previous


class BaseTestState(TestState):

    @classmethod
    def start_scenario(cls):
        cls.l = [1, 3, 2, 4]

class Create(BaseTestState, start=True):

    def test_1(self):
        self.assertEqual(self.l, [1, 3, 2, 4])


class Reverse(BaseTestState, previous=['Create']):
    def input(self):
        self.l.reverse()

    def test_1(self):
        self.assertEqual(self.l, [4, 2, 3, 1])


class Sort(BaseTestState, previous=['Create', 'Reverse']):
    def input(self):
        self.l.sort()

    def test_1(self):
        self.assertEqual(self.l, [1, 2, 3, 4])


load_tests = BaseTestState.get_load_tests()
```

The *BaseTestState* is created by subclassing *cricri.TestState*. This subclass defines *start_scenario* method in order to store the object to be tested in a class attribute. The *start_scenario* method will be called once at the beginning of each generated scenario.

Each *BaseTestState* subclass defines a scenario step. *start attribute* allow you to define the first step. Here *Create* class is the first step. *previous attribute* allow you to define when step is executed. Here *Reverse* step is executed when *Create* step is done and *Sort* step can be executed when *Create* or *Reverse* step is done.

This three steps define two scenario:

- Create and Sort list.
- Create, Reverse and Sort list.

Each step has input method. This method is called before test methods of step.

The last statement allows unittest to manage this module. It generate all unittest.TestCase classes.

To run this script, use unittest command-line interface:

```
$ python3 -m unittest -v test_scenario_list
```

You will see the following output:

```
create (1/2) ... ok
sort   (2/2) ... ok
create  (1/3) ... ok
reverse (2/3) ... ok
sort    (3/3) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

If you want use **pytest** to launch this script, you should install pytest-cricri

```
pip install pytest-cricri
```

You can launch test with pytest using *–cricri* option

```
pytest --cricri test_scenario_list.BaseTestState
```

CHAPTER 3

condition decorator

The **condition** decorator allows you to have a conditional execution of test method. this function takes a Condition objects such as Previous or Path.

Example:

```
class B1(BaseTestState):
    ...

class B2(BaseTestState):
    ...

class C(BaseTestState, previous=['B1', 'B2']):

    @condition(Previous(['B1']))  # Called when previous step is B1
    def input(self):
        ...

    @condition(Previous(['B2'])  # Called when previous step is B2
    def input(self):
        ...

    @condition(Previous(['B1'])  # Called when previous step is B1
    def test_1(self):
        ...

    @condition(Previous(['B2'])  # Called when previous step is B2
    def test_2(self):
        ...
```

Note that TestState subsubclass can have several input methods if **condition** decorator is used.

CHAPTER 4

Documentation

4.1 Condition object

The Conditions objets are used in **condition** decorator.

You can combine Condition objects using operator.

Operator	Meaning	Example
-	not	- Path('A', 'B')
&	and	Path('A', 'B') & Path('F', 'G')
|	or	Path('A', 'B') | Path('F', 'G')

4.1.1 Built-in Condition

Previous

Previous(step [,step2 [...]]) is enable if last executed step is in given steps.

Example:

@condition(Previous("I", "J"))	
Executed steps	Decorated method is executed
K, I, J	True
K, J, I	True
J, I, K	False
I, J, K	False
J	True
I	True

Path

Path(step [,step2 [...]]) is enable if the given contigious steps have executed.

Example:

@condition(Path("I", "J"))	
Executed steps	Decorated method is executed
I, J	True
J, I, J, I	True
J, I	False
I, K, J	False
K, J	False

Newer

Newer(step1, step2) is enable if step2 execution is newer than step1 execution or step1 has not executed.

Example:

@condition(Newer("I", "J"))	
Executed steps	Decorated method is executed
I, J	True
J, I, J, I	False
J, I	False
I, K, J	True
K, J	True
K, I	False

4.1.2 How to create a custom Condition

You can create a custom Condition by inheriting from Condition class and overriding the __call__ method. The __call__ method takes *previous_steps* parameter - *previous_steps* parameters is a list of executed step names - and return True if decorated method must be executed else False.

Here is a Condition wich is enable when step appears a given number of times:

```
class Count(Condition):

    def __init__(self, step, count):
        self.step = step
        self.count = count

    def __call__(self, previous_steps):
        previous_steps = tuple(previous_steps)
        return previous_steps.count(self.step) ==  self.count
```

4.1.3 Shortcut

Cricri provides shortcut decorators:

shortcut	means
@previous(step [,step2 [...]])	@conditon(Previous(step [,step2 [...]]))
@path(step [,step2 [...]])	@conditon(Path(step [,step2 [...]]))
@newer(step1, step2)	@conditon(Newer(step1, step2))

4.2 Test TCP server

cricri provides *TestServer* class to test server. You must subclasse *TestServer* and define your servers and clients:

```
class TestMyServer(TestServer):

    commands = [
        {
            "name": "application",
            "cmd": ["python3", "-u", "application.py", "{port-1}",
                    "--db-port", "{port-2}"],
            "env": {"PYTHONPATH": "/home/project/chat"}
        },
        {
            "name": "database",
            "cmd": ["docker", "run", "-p", "{port-2}:5420", "db-service"],
        }
    ]
```

Servers are defined in commands list. You will can reference server in test methods by the name using *self.servers['name-defined-in-commands-list']*

4.2.1 commands available parameters

The *commands* must be a list containing dict with:

- name (required) the name of command
- **cmd (required) list of sequence of program arguments the first** element is a program.
- **kill-signal (optional) should be an enumeration members of** :py:class: *signal.Signals*
- env (optional) A dict that defines the environment variables
- extra-env (optional) A dict that add environment variables

4.2.2 Assert server methods

assert_stdout_is(*expected*, *timeout=2*)
Test that server logs *expected* on the stdout before *timeout*.

assert_stderr_is(*expected*, *timeout=2*)
Test that server logs *expected* on the stderr before *timeout*.

assert_stdout_regex(*regex*, *timeout=2*)
Test that server logs on stdout before *timeout* and message matches *regex*.

assert_stderr_regex(*regex*, *timeout=2*)
Test that server logs on stderr before *timeout* and message matches *regex*.

4.3 clients configuration

4.3.1 How to test a HTTP server or REST API

You can create HTTP client using http_clients attribute in a *TestServer* subclasse:

```
class TestRestServer(TestServer):

    http_clients = [
        {
            "name": "Alice",
            "host": "127.0.0.1",
            "port": "{port-1}",
            "extra_headers": [
                ('Content-Type', 'application/json')
            ]
        }
    ]
```

http_client available parameters

HTTPClient.**__init__**(*host*, *port*, *timeout*, *tries*, *wait*, *headers*, *extra_headers=None*)

Parameters

- **host** (*str*) – Server host
- **port** (*int*) – The number of tries
- **tries** (*int*) – The number of tries
- **timeout** (*int*) – Deadline before abort request
- **headers** (*List[Tuple]*) – The default headers used when you call request method without define headers parameter.
- **extra_headers** (*List[Tuple]*) – Always used when you call request method by default extra_headers contains the User-Agent.

Your HTTP clients are instantiate in *clients* class attribute and you may use them in the *input method*:

```
class GetHotels(TestRestServer, start=True):

    def input(self):
        self.clients["Alice"].get("/hotels")
```

HTTPClient methods

HTTPClient.**request**(*method*, *path*, *headers=None*, *timeout=None*, *data=<object object>*)
Performe HTTP request to *path*

Parameters

- **method** (*str*) – HTTP verb such as 'GET', 'POST', . . .
- **path** (*str*) – HTTP server path

- **headers** (*List[Tuple]*) – HTTP headers must be 2-tuple (header, value) if headers is not define *__init__* headers parameter is used.
- **timeout** (*int*) – Deadline before abort request
- **data** (*object*) – HTTP request content. data is serialised regarding the content-type define in the HTTP headers. You can change this behavior updating *serializers* class attribute.

HTTPClient.**get**(**args*, ***kwargs*)
Shortcut to request(GET, ...)

HTTPClient.**post**(**args*, ***kwargs*)
Shortcut to request(POST, ...)

HTTPClient.**put**(**args*, ***kwargs*)
Shortcut to request(PUT, ...)

HTTPClient.**delete**(**args*, ***kwargs*)
Shortcut to request(DELETE, ...)

HTTPClient.**patch**(**args*, ***kwargs*)
Shortcut to request(PATCH, ...)

Response testing

The client stores HTTP response in response attribute using HTTPResponse object. This HTTPResponse object provide methods useful for test server.

HTTPResponse.**assert_header_has**(*header*, *expected_value*, *separator=', '*)
Test if header of HTTP response contains the expected_value.

HTTPResponse.**assert_header_is**(*header*, *expected_values*, *separator=', '*)
Test if value header of HTTP response is the *expected_value*.

HTTPResponse.**assert_status_code**(*status_code*)
Test if status code of HTTP response is *status_code*

HTTPResponse.**assert_reason**(*reason*)
Test if reason of HTTP response is *reason*

Example:

```
class GetHotels(TestRestServer, start=True):

    def test_status_code_should_be_200(self):
        self.clients["Alice"].response.assert_status_code(200)

    def test_content_has_hotel_california(self):
        content = self.clients["Alice"].response.content
        expected = ({
            "name": "California",
            "addr": "1976 eagles street"
        },)

        self.assertCountEqual(content, expected)
```

4.3.2 How to test a TCP server

cricri provides *TestServer* class to test TCP server. You must subclasse *TestServer* and define your servers and clients:

```
class TestChatServer(TestServer):

    commands = [
        {
            "name": "chat-server",
            "cmd": ["python3", "-u", "chat_server.py", "{port-1}",
                    "--db-port", "{port-2}"],
            "env": {"PYTHONPATH": "/home/project/chat"}
        },
        {
            "name": "database",
            "cmd": ["docker", "run", "-p", "{port-2}:5420", "db-service"],
        }
    ]

    tcp_clients = [
        {
            "name": "Alice",
            "port": "{port-1}",
        }
    ]
```

This example define *TestChatServer* class, which define command to launch server and tcp client. Before each scenario running, 'python3 -u chat_server.py {port-1}' is executed and a tcp client is connected to '{port-1}'. The string '{port-1}' will be bound by the fist free TCP port.

You may reference defined clients and servers in your *TestChatServer* subclasses using *clients* and *servers* attributes:

```
class Start(TestChatServer, start=True):

    def test_server_listen(self):
        self.servers['chat-server'].assert_stdout_is(
            'server listen', timeout=2
        )

class AliceAskedNickname(TestChatServer, previous=["Start"]):

    def input(self):
        self.clients["Alice"].send("MY_NAME_IS;Alice;")

    def test_alice_should_receive_ok(self):
        self.clients["Alice"].assert_receive('OK')
```

In this example, the *Start* step class test that server write 'server listen' to stdout. The *AliceAskedNickname* class send 'MY_NAME_IS;Alice;' string to the server and test that Alice receive 'OK'.

Assert TCP client methods

assert_receive(*self*, *expected*, *timeout=2*)
Test that client received *expected* before *timeout*.

assert_receive_regex(*self*, *regex*, *timeout=2*)
Test that client received data before *timeout* and data matches *regex*.

4.4 TIPS

4.4.1 Generate scenario with an other class than unittest.TestCase

You can define the base class of generated TestCase using *base_class* attribute. By default the used class is unittest.TestCase but you can use any unittest.TestCase subclass.

```
class BaseTest(TestState):
    base_class = MyCustomTestCase
```

4.4.2 Create a custom client for TestServer class

classmethod MetaServerTestState.**bind_class_client**(*class_client*)

You can use the *bind_class_client* method to allow each *TestServer* subclass to manage a new *Client*.

To crete a new *Client*, you should subclass *cricri.inet.Client* and define the *validator* static method and the *close* method.

```
from urllib.request import urlopen

from cricri import MetaServerTestState
from cricri.inet import Client


class MyCustomClient(Client):

    attr_name = 'my_custom_clients'

    def __init__(self, host):
        self.host = host
        self.response = None

    @staticmethod
    def validator():
        return {
            'host': str
        }

    def close(self):
        if self.response is not None:
            self.response.close()

    def url_open(self, page):
        self.response = self.urlopen(self.host + '/' + page)
        self.content = self.response.read()

    def assert_page_content(self, text):
        if text not in self.content:
            raise AssertionError(
                '{} in not {}'.format(text, self.content))


MetaServerTestState.bind_class_client(MyCustomClient)
```

CHAPTER 5

Contributing

Contributions are welcome! Clone the repository on GitHub.

Python Module Index

C

Index

Symbols

A

B

C

D

G

P

R