TESTING BASICS

**11.1.2018**

**Authors: Axel Neergaard**

# **INTRODUCTION**

The following document will show the basics of test writing for the Libby project. The tests should be extensive and wide. This does not mean that one should write long tests, but almost the contrary, short ones are preferred. This ensures that we have multiple atomic tests that report independently for each part of the program to be tested.

*All* cases that exist in the program should - preferably - tested. As an example, this means that each section of an if statement should be tested - even if this would seem to be trivial - to ensure code coverage and test quality. Each part not tested cannot be said to be known to work (even though a *working* program in this sense does not mean bug-free, as a test can only show the presence of bugs [paraphrased by Edsger W. Dijkstra]).

**PACKAGES/MODULES**

The code coverage tool used, for now, is the [*cov* plugin for pytest](https://pypi.python.org/pypi/pytest-cov). To install this plugin run the following (with PIP):

pip install pytest-cov

The plugin enables one to run pytest by the normal command with the option:

(pytest) --cov=<test-to-be-run> (<tests>)

(Do not include the brackets). Where <test-to-be-run> is just the test to be covered. If you want to cover all tests run just insert a dot(.):

pytest --cov=. <tests>

This will include a table to the results with a percentage covered for each file.

If one wants to see more details about what is covered, such as each part of the program executed, one can append the option:

(pytest --cov=.) --cov-report html (<tests>)

This will add a directory named *htmlcov* to the folder where the command was executed. If one opens the file *index.html* one can see the percentage covered for each file with links to the file itself to see in detail which lines were executed, and which not.

**In short** to test everything with coverage and the getting the results to a nice html file, run the following (insert a separate test case inside the brackets at the end if only one test should be run):

pytest --cov=. --cov-report html (<tests>)

**WRITING TESTS (IN SHORT)**

The tests are currently run with [Python’s built-in unit tests *unittest*](https://docs.python.org/3/library/unittest.html). One has to import this module and create a class that extends *unittest.TestCase*:

import unittest

class TestName(unittest.TestCase):

…

To create atomic tests one has to create functions (with Python classes’ mandatory *self* argument):

def test\_something(self):

…

The class created has the following functions that can be used to test certain criteria with boolean values:

self.assertTrue(statement) # check that the statement holds true

self.assertFalse(statement) # check that the statement does not hold true

self.assertEquals(statement1, statement2) # check that two statements are equal

More can be found on the documentation page for Python’s unit tests.

To create a function that executes only in the beginning or the end of the test suite (i.e. basically the whole test) the unit test module has two built in functions, that have to have the decorator *@classmethod*:

@classmethod

def setUpClass(cls): # the mandatory self argument is called ‘cls’ here

…

@classmethod

def tearDownClass(cls): # the same case as above

…

These can be used to e.g. check that certain criteria (such as the existence of folders and possibly create these) before the actual suite is run, or check that some files get deleted.

*unittest* also provide two functions that always be run before a certain test-function is run (these do not need decorators):

def setUp(self):

…

def tearDown(cls):

…

***Important:*** if one wants to create a function or statement clause that should not be checked for code coverage (i.e. gets ignored in the code coverage) on can simply comment the following at the beginning of the clause:

# pragma: no cover

Some examples:

def do\_not\_cover(self): # pragma: no cover

# will not be included in the code coverage statistics

…

if something: # pragma: no cover

# nothing here will be covered

…

However, this is not advisable as there will be no code coverage statistics for these parts and thus one cannot know if they got executed or not. They may be useful in some situations, however, such as ensuring that some directories or files actually got deleted after the suite if the test itself might have failed. **Skipping code coverage** should only be done in the tests themselves, and not in the actual programs.

**WRITING TESTS FOR INTENTS**

When writing tests for intents, you have to remember to add a JSON file that would simulate an input from a user to Amazon Lex. For getting a personalized version you can switch ‘debug’ to ‘True’ in the ‘main\_handler.py’ file in your Lambda function so that the Amazon Lex bot answers with the whole input instead of going through your Lambda function. Otherwise you can copy the works of others. Usually these files are named the same as your Python testing file, with a ‘.json’ at the end.

## **IDEAS AND NOTES FOR THE FUTURE**

During a (Skype) meeting with Verto (08/01/2018 - doc: Sprint 2/ Meeting docs/08.01.2018-skype-meet-notes) there was a mention about [*SonarQube*](https://www.sonarqube.org/). SonarQube provides Continuous Code Quality inspections for programs and can be integrated with CI-tools. SonarQube can detect bad code smells and continuously inspects the code. This could be possible option to integrate into our development.

## **LINKS AND MORE INFORMATION**

*This guide is kept short as to only be a small introduction and reminder when testing. It is not extensive as the documentation for each part mentioned are well written and explains everything in much greater detail.*

pytest-cov:

<https://pypi.python.org/pypi/pytest-cov>

<https://pytest-cov.readthedocs.io/en/latest/>

Python unittest:

<https://docs.python.org/3/library/unittest.html>