Test Driven Development (TDD) with Python

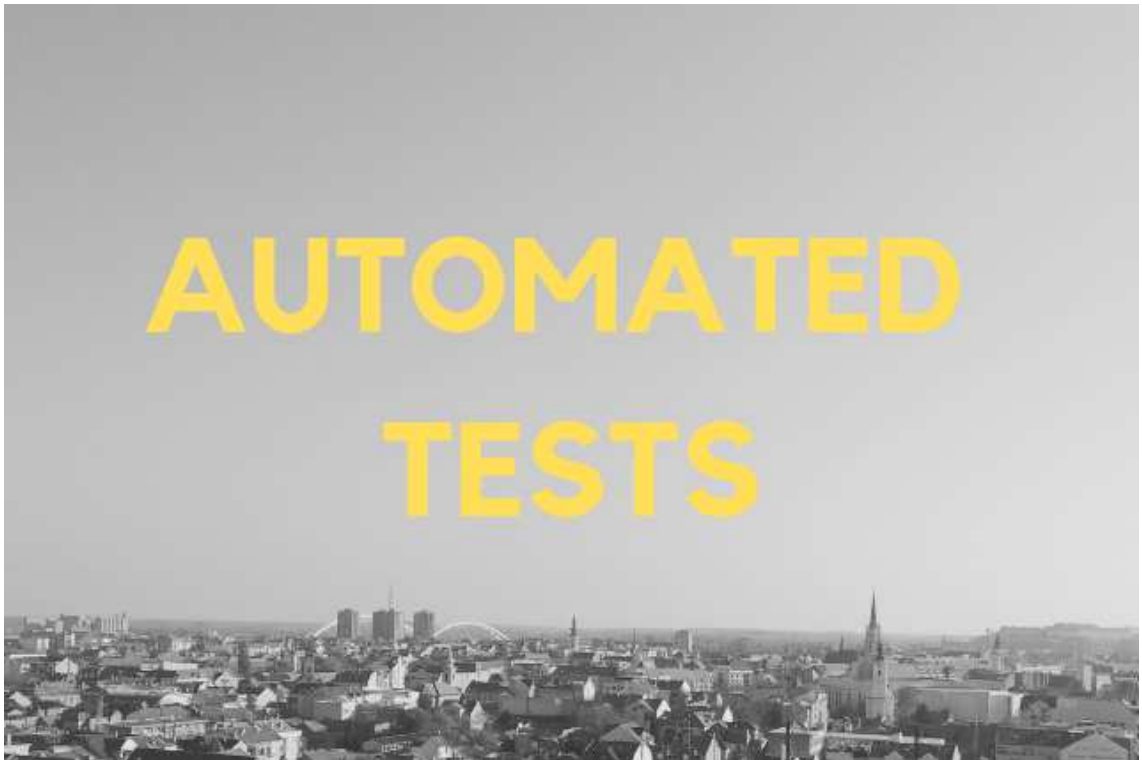
Mar 4, 2019 | Python | 4 comments

Bugs! Every application has it and no matter how careful you are you will create one sooner or later. Especially if you are working on a large enterprise software. Now, I know a bunch of people will have “My code is clean and bugless” attitude, but creating bug free code is extremely hard if not impossible. However, that doesn’t mean that you should give up and just write spaghetti code that just doesn’t work. We – **developers**, should give our best to write **high-quality code**. High-quality means a low number of bugs, among other things.

Apart from that, having a bug in production is extremely expensive. You probably know that comparison, where bug found during development is **100 times cheaper** than finding the same bug during production. So, we should focus on finding our bugs as soon as possible. Our first line of defense is **testing**.

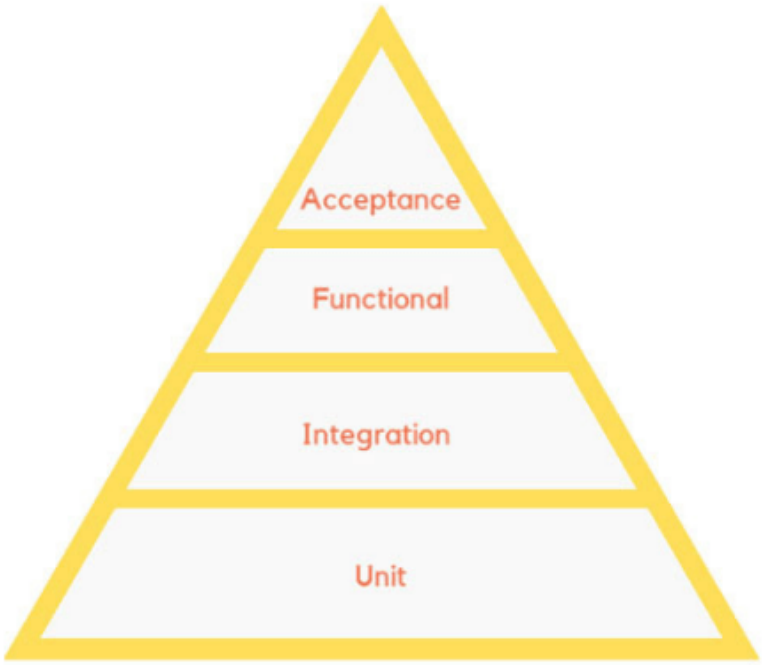
We could test our applications manually, just by running them and clicking around. However, this approach has many **pitfalls**. First one is that it is time-consuming, which basically means expensive. This way of testing makes regression testing extremely hard too. Imagine that you've just added a new feature into your application. You will have to make sure that this new feature didn't break any of the old functionalities, which means testing your whole application from the beginning. Again, time-consuming and costly. What is the solution? **Automated testing** of course.

Automated Tests



As we could see, manual testing is not really working for us especially if we want to detect issues early during development phase. So, we decided to automate our tests. Based on the level of abstraction that tests are done they can be:

- **Unit Tests** – It is a piece of a code that invokes another piece of code (unit) and checks if an output of that action is the same as the desired output.
- **Integration Tests** – It is testing a unit without full control over all parties in the test. They are using one or more of its outside dependencies, such as a database, threads, network, time, etc.
- **Functional Tests** – It is testing a slice of a functionality of the system, observing it as a black box and verifying the output of the system to the functional specification.
- **Acceptance Tests** – It is testing does the system fulfill expected business and contract requirements. It is considering the system as a black box.



Pyramid of Tests

These definitions are a bit lose, but you get the point. In the image above, you can see the so-called Pyramid of Tests. It displays the number of tests that we should have in our application per type of test. We can see that we have the largest number of unit tests. For the purpose of this article, we will consider only this type of tests, since they are crucial for the TDD process, as we will see in a little bit.

Unit Tests

As previously mentioned unit tests are testing the functionality of a unit. This brings us to a philosophical question about what exactly is “unit”? A **unit** is the set of actions between the invocation of a method in the system and a single noticeable output of that system. The equally philosophical answer, right? The important thing to understand here is that the unit test is a piece of code that tests another piece of code. Over the years, this type of tests turned out to be one of the **best tools** for increasing software quality. They were introduced by Kent Back in Smalltalk back in the 1970s and since then they are used in pretty much any programming language.
So, how can we write unit tests in Python? Unit tests are always written using some sort of unit test framework. For Python that is module *unittest*. Here is how we can use this module to write our first tests:

```
import unittest



class FirstTestClass(unittest.TestCase):



    def test_upper(self):

        self.assertEqual('rubiks code'.upper(), 'RUBIKS CODE')
```

```
if __name__ == '__main__':

    unittest.main()
```

view rawfirst_test.py hosted with ♡ by GitHub

First, we create class *FirstTestClass*, which is inheriting *TestCase* from the *unittest* module. Using this inheritance we are defining test class which contains our tests methods or test cases. These test cases will be registered within *unittest* module and we will be able to run them later. In this class, we are having only one test case *test_upper*.
This method uses function *assertEqual* to verify that the call of the *upper* method on the string really returns the same string with all caps. The module *unittest* has a lot of these functions that start with the word *assert*.
Essentially, every test method should call one of these methods to verify the results and so the test runner can accumulate all test results and produce a report. Finally, at the end of this file, we are calling *unitest.main*. This will run all registered tests. Here is what we get when we run this:

```
E:\python_tdd>python simple_unit_tests.py
----------------------------------------------------
Ran 1 test in 0.000s
OK
```

If we want to know which tests cases are called you can just add *-v* as an argument:

```
E:\python_tdd>python simple_unit_tests.py -v
test_upper (main.FirstTestClass) … ok
----------------------------------------------------
Ran 1 test in 0.000s
OK
```

As we can see we run our one test case, and got the result that it passes, ie. the condition that we check with *assertEqual* is true. Congratulations, you've just run your first test with Python! Now, let's see how we can test some functionality that we made. Take a look at this code:

```
def get_greetings():

    return 'Hello World!'
```

view rawhelloword.py hosted with ♡ by GitHub

It is very simple function *get_greetings* which is just returning *'Hello World!'* string. This is how we test it:

```
import unittest

from helloworld import get_greetings



class HelloworldTests(unittest.TestCase):
```

```
        def test_get_helloworld(self):

            self.assertEqual(get_greetings(), 'Hello World!')



    if __name__ == '__main__':

        unittest.main()
```

view rawtests_helloworld.py hosted with ♡ by GitHub

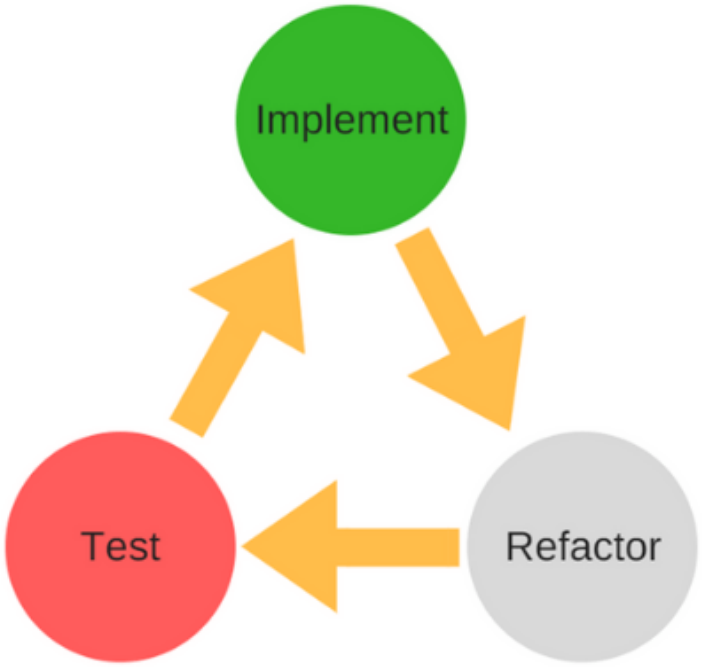
Pretty easy, right? We just import function from the file, write class that inherits *unittest.TestCase* and verify the result using *assertEqual* within test method *test_get_helloworld*. Output looks like this:

```
E:\python_tdd>python tests_helloworld.py -v
test_get_helloworld (main.HelloworldTests) … ok
----------------------------------------------------------
Ran 1 test in 0.001s
OK
```

What is Test Driven Development?

Test Driven Development (TDD) is an evolutionary approach to building and designing software solutions. It is consisting of small cycles in which we are writing a unit test, that will initially fail, and then implementing the minimum amount of code to pass that test. After that code can be refactored to follow some good principles. Refactoring has a safety net, because we wrote the tests already, so we can reshape our solution stress-free.
These three important steps of TDD are easy to remember by using something that I like to call the **TDD mantra**. It goes like this: **Red – Green – Refactor**. Red is corresponding with the phase in which we write a test that will fail. Then we implement the code to make

previously written test pass meaning it is – Green. And finally we refactor our code – and we don't really have a color for that one. So there you go, TDD Mantra – Red, Green, Refactor.



TDD Process | TDD Mantra

You might wonder what is the difference between just writing unit tests for your code and TDD? In general, we are using unit tests in both cases. The crucial difference between TDD and traditional testing is **the moment** in which we are writing the tests. When we are writing code using TDD we first write the tests and then the code itself, and not another way around.

The benefit of this approach is that we are minimizing the possibility of forgetting to write tests for some part of the code. Ideally, we end up with the code that is fully tested upfront and solutions that are implemented using TDD usually have 90%-100% of code covered with tests. Of course, when our code is tested it is less likely that we have a bug in our system. Another important difference is that we are writing **small chunks of code** to satisfy our test. This way the process itself drives our design and forces us to keep things simple. By using TDD we avoid creating over complicated designs and overengineered systems. Arguably this is the **biggest benefit** of this approach. When we use it we end up with clearer design and API. This approach also forces you to design classes properly and to follow good code principles like SOLID and DRY. Personally, I find this way of development as a great procrastination killer and a great **motivator**. It kinda keeps you in the zone.

Solving a problem using TDD

Before we proceed let's examine what kind of problem we are trying to solve. Do you guys like TV show *Rick and Morty*? I love it. The show follows the adventures of cynical mad scientist Rick Sanchez and his grandson Morty Smith. Rick owns a portal gun and takes Morty to different dimensions/universes. Different versions of these characters inhabit those other dimensions. The Citadel is the place where Ricks and Mortys have formed a society built by their counterparts from an infinite amount of realities. We are in luck because we have a request from The Citadel for one Python module. Here are the user stories:

1. **A user is able to assign Ricks and Mortys a universe number**
2. **A user is able to add residents to the Citadel**
3. **A user is able to turn all Ricks with assigned Mortys to pickles (watch s03e03)**

Ok, let's start from the first user story and work our TDD magic to the last user story. The first user story tells us that we should have two classes, one for Rick and one for Morty. However, since we are using TDD, we write the unit tests first. We implement *Rick* test class like this:

```
import unittest

from rick import Rick



class RickTests(unittest.TestCase):

    def test_universe(self):

        rick = Rick(111)

        self.assertEqual(rick.universe, 111)



if __name__ == '__main__':

    unittest.main()
```

view rawrick_tests1.py hosted with ♡ by GitHub

Of course if we run this test we will get an error saying that Rick class is not existing:

```
======================================================================
ERROR: test_universe (__main__.RickTests)
Traceback (most recent call last):
  File "rick_tests.py", line 5, in test_universe
    rick = Rick(111)
NameError: name 'Rick' is not defined

Ran 1 test in 0.001s
FAILED (errors=1)
```

We need to define the class and initialize it through the constructor with the value for the *universe*:

```
class Rick(object):

    def __init__(self, universe):

        self.universe = universe
```

view rawrick1.py hosted with ♡ by GitHub

Now, when we re-run the tests, we get this:

```
test_universe (__main__.RickTests) … ok
----------------------------------------------------
Ran 1 test in 0.000s
OK
```

We are following the same pattern for Morty. First the failing test:

```
import unittest

from morty import Morty



class MortyTests(unittest.TestCase):

    def test_universe(self):

        morty = Morty(111)

        self.assertEqual(morty.universe, 111)



if __name__ == '__main__':
```

```
        unittest.main()
```

view rawmorty_test1.py hosted with ♡ by GitHub

Followed by the implementation:

```
    class Morty(object):

        def __init__(self, universe):

            self.universe = universe
```

view rawmorty1.py hosted with ♡ by GitHub

And passed test:

```
test_universe (main.MortyTests) … ok
----------------------------------------------------
Ran 1 test in 0.000s
OK
```

You might notice that this “dance” seems unnatural at first. It takes some time to get used to it, but once you do it is enchanting. You will wonder how you were able to do it another way for years. Ok, so we implemented our first user story. Let’s move onto the second one. This one is pretty easy as well – “A user is able to add residents to the Citadel”. However, if we want to add residents, this means that *Citadel* class should have some sort of the list or array of residents. Let’s first make a function that will return all residents. We write a test for *Citadel* class:

```
    import unittest

    from citadel import Citadel
```

```
class CitadelTests(unittest.TestCase):

    def test_get_all_residents(self):

        citadel = Citadel()

        residents = citadel.get_all_residents()

        self.assertCountEqual(residents, [])



if __name__ == '__main__':

    unittest.main()
```

view rawcitadel_tests1.py hosted with ♡ by GitHub

This test fails, because *Citadel* implementation doesn't exist yet. Since this seems a little bit more complicated than previous implementations, we write something like this and try to make our test pass:

```
class Citadel(object):

    def __init__(self):

        self.__residents__ = []



    def get_all_residents(self):

        pass
```

view rawcitadel1.py hosted with ♡ by GitHub

As you can see, we defined private field *__residents__* and added method *get_all_residents* which is not doing anything at the moment. This way we can run our test, but it fails again:

```
======================================================================
ERROR: test_get_all_residents (main.CitadelTests)
Traceback (most recent call last):
  File "citadel_tests.py", line 10, in test_get_all_residents
    self.assertCountEqual(residents, 0)
  File "C:\Users\NikolaVanja\Anaconda3\lib\unittest\case.py", line
```

```
1165, in assertCountEqual
     first_seq, second_seq = list(first), list(second)
 TypeError: 'NoneType' object is not iterable
 ----------------------------------------------------
 Ran 1 test in 0.002s

 FAILED (errors=1)
```

So, in order to fix this we have to return that private field through the method:

```
class Citadel(object):

    def __init__(self):

        self.__residents__ = []



    def get_all_residents(self):

        return self.__residents__
```

view rawcitadel2.py hosted with ♡ by GitHub

Re-run the test:

```
 test_get_all_residents (main.CitadelTests) … ok
 ----------------------------------------------------
 Ran 1 test in 0.001s
 OK
```

Hooray! Test passes and we are making progress. Still, functionality that satisfies second user story is not implemented. That is why we write another test so that the complete *Citadel* test class now looks like this:

```
import unittest

from citadel import Citadel

from rick import Rick

from morty import Morty



class CitadelTests(unittest.TestCase):

    def test_get_all_residents(self):
```

```
            citadel = Citadel()

            residents = citadel.get_all_residents()

            self.assertCountEqual(residents, [])



        def test_add_resident(self):

            citadel = Citadel()

            rick = Rick(111)

            morty = Morty(111)



            citadel.add_resident(rick)

            citadel.add_resident(morty)

            residents = citadel.get_all_residents()



            self.assertEqual(residents[0], rick)

            self.assertEqual(residents[1], morty)



    if __name__ == '__main__':

        unittest.main()
```

view rawcitadel_tests2.py hosted with ♡ by GitHub

Running this will fail, because we don't have *add_residents* method. So, let's implement it:

```
class Citadel(object):

    def __init__(self):

        self.__residents__ = []



    def get_all_residents(self):

        return self.__residents__



    def add_resident(self, resident):

        self.__residents__.append(resident)
```

view rawcitadel3.py hosted with ♡ by GitHub

Now when we run the test we get this:

```
test_add_resident (__main__.CitadelTests) … ok
test_get_all_residents (__main__.CitadelTests) … ok
----------------------------------------------------
Ran 2 tests in 0.002s
OK
```

Awesome! We finished two out of three user stories. However, the last one is the trickiest. Let's examine it – A user is able to turn all Ricks with assigned Mortys to pickles. We can detect several tasks within this one sentence. We should be able to assign Morty to a Rick, meaning we need to extend both of those classes. We should be able to turn Rick into a

pickle, as well. Finally, we should be able to do that for all Ricks in the Citadel with assigned Morties. So, let's proceed in that order. First, we extend Morty with *is_assigned* field. Extended *Morty* test class looks like this:

```
import unittest

from morty import Morty



class MortyTests(unittest.TestCase):

    def test_universe(self):

        morty = Morty(111)

        self.assertEqual(morty.universe, 111)



    def test_is_assigned(self):

        morty = Morty(111)

        self.assertFalse(morty.is_assigned)



if __name__ == '__main__':

    unittest.main()
```

view rawmorty_tests2.py hosted with ♡ by GitHub

The new test of the class will fail. We have to extend the *Morty* class implementation as well:

```
class Morty(object):

    def __init__(self, universe):

        self.universe = universe
```

```
            self.is_assigned = False
```

view rawmorty2.py hosted with ♡ by GitHub

Run the test again:

```
test_is_assigned (main.MortyTests) … ok
test_universe (main.MortyTests) … ok
----------------------------------------------------
Ran 2 tests in 0.002s
OK
```

Ok, we are getting closer. Now *Rick* class should be extended so *Morty* can be assigned to *Rick*. We extend the *Rick* test class:

```
import unittest

from rick import Rick

from morty import Morty



class RickTests(unittest.TestCase):

    def test_universe(self):

        rick = Rick(111)

        self.assertEqual(rick.universe, 111)



    def test_has_morty(self):

        rick = Rick(111)

        self.assertEqual(rick.morty, None)



if __name__ == '__main__':

    unittest.main()
```

view rawrick_tests2.py hosted with ♡ by GitHub

Test is failing because we are missing *morty* field in *Rick* class. Let's extend *Rick*:

```
class Rick(object):

    def __init__(self, universe):

        self.universe = universe

        self.morty = None
```

view rawrick2.py hosted with ♡ by GitHub

Ignite tests again:

```
test_has_morty (__main__.RickTests) … ok
test_universe (__main__.RickTests) … ok
----------------------------------------------------
Ran 2 tests in 0.001s
OK
```

Nice, now let's extend tests for *assign* method, through which we will assign Morty to Rick:

```
import unittest

from rick import Rick

from morty import Morty



class RickTests(unittest.TestCase):

    def test_universe(self):

        rick = Rick(111)

        self.assertEqual(rick.universe, 111)



    def test_has_morty(self):

        rick = Rick(111)
```

```
            self.assertEqual(rick.morty, None)


        def test_assing_morty(self):
            rick = Rick(111)
            morty = Morty(111)

            rick.assign(morty)

            self.assertEqual(rick.morty, morty)
            self.assertTrue(morty.is_assigned)


    if __name__ == '__main__':
        unittest.main()
```

view rawrick_tests3.py hosted with ♡ by GitHub

As you can see we are checking two things after Morty is assigned. When we extend *Rick* class to support these changes it looks like this:

```
    class Rick(object):
        def __init__(self, universe):
            self.universe = universe
            self.morty = None
```

```
        def assign(self, morty):

            self.morty = morty

            morty.is_assigned = True
```

view rawrick3.py hosted with ♡ by GitHub

And when we re-run the tests for *Rick* class:

```
test_assing_morty (main.RickTests) … ok
test_has_morty (main.RickTests) … ok
test_universe (main.RickTests) … ok
----------------------------------------------------
Ran 3 tests in 0.002s
OK
```

Don't give up on me now, we are halfway through the third user story! We need to make Rick "pickable" and turn all Ricks with assigned Mortys in the Citadel into pickles (I never thought I would write down a sentence like this :)). To the *Rick* test class!

```
    import unittest

    from rick import Rick

    from morty import Morty



    class RickTests(unittest.TestCase):

        def test_universe(self):
```

```
        rick = Rick(111)
        self.assertEqual(rick.universe, 111)

    def test_has_morty(self):
        rick = Rick(111)
        self.assertEqual(rick.morty, None)

    def test_assing_morty(self):
        rick = Rick(111)
        morty = Morty(111)

        rick.assign(morty)

        self.assertEqual(rick.morty, morty)
        self.assertTrue(morty.is_assigned)

    def test_has_is_pickle(self):
        rick = Rick(111)
        self.assertFalse(rick.is_pickle)
```

```
if __name__ == '__main__':

    unittest.main()
```

view rawrick_tests4.py hosted with ♡ by GitHub

This seems familiar. Test *test_has_is_pickle* fails because, well, *Rick* class still has no field *is_pickle*. *Rick* class needs to be extended for that:

```
class Rick(object):

    def __init__(self, universe):

        self.universe = universe

        self.morty = None

        self.is_pickle = False



    def assign(self, morty):

        self.morty = morty

        morty.is_assigned = True
```

view rawrick4.py hosted with ♡ by GitHub

Run the tests again:

```
test_assing_morty (main.RickTests) … ok
test_has_is_pickle (main.RickTests) … ok
test_has_morty (main.RickTests) … ok
test_universe (main.RickTests) … ok
----------------------------------------------------
Ran 4 tests in 0.003s
OK
```

Awesome! Now, to the *Citadel* test class. We add one large test:

```
import unittest

from citadel import Citadel
```

```
from rick import Rick
from morty import Morty


class CitadelTests(unittest.TestCase):
    def test_get_all_residents(self):
        citadel = Citadel()
        residents = citadel.get_all_residents()
        self.assertCountEqual(residents, [])

    def test_add_resident(self):
        citadel = Citadel()
        rick = Rick(111)
        morty = Morty(111)

        citadel.add_resident(rick)
        citadel.add_resident(morty)
        residents = citadel.get_all_residents()

        self.assertEqual(residents[0], rick)
```

```
            self.assertEqual(residents[1], morty)


        def test_picle_ricks_with_morties(self):
            citadel = Citadel()
            rick = Rick(111)
            morty = Morty(111)
            rick.assign(morty)

            citadel.add_resident(rick)
            citadel.add_resident(morty)

            citadel.picle_ricks_with_morties()
            residents = citadel.get_all_residents()

            self.assertTrue(residents[0].is_pickle)


    if __name__ == '__main__':
        unittest.main()
```

Here is the explanation. We create all necessary objects, assign Morty to Rick, add both objects into Citadel and call method that should turn all Ricks with Mortys into pickles. Cool, let's reflect that in *Citadel* class implementation:

```
from rick import Rick



class Citadel(object):

    def __init__(self):

        self.__residents__ = []



    def get_all_residents(self):

        return self.__residents__



    def add_resident(self, resident):

        self.__residents__.append(resident)



    def picle_ricks_with_morties(self):

        for resident in self.__residents__:

            if isinstance(resident, Rick):

                if resident.morty: resident.is_pickle = True
```

view rawcitadel3.py hosted with ♡ by GitHub

Alright, let's re-run the tests:

```
test_add_resident (main.CitadelTests) … ok
test_get_all_residents (main.CitadelTests) … ok
test_picle_ricks_with_morties (main.CitadelTests) … ok
----------------------------------------------------
Ran 3 tests in 0.002s
OK
```

Woooohoooo! The tests are passing and we completed our third and final user story!

Conclusion

In this article, we went through several concepts. We explored what kind of automated tests exists. We focused on the unit tests since they are the backbone of Test Driven Development, which we also explained. Finally, we implemented one solution using this technique. You might notice that we haven't done a lot of refactoring because the examples were pretty straight forward. However, we could notice how this way of development is driving our implementation, and how it forces us to write clean and testable code. What we haven't explored in this article is the concept of mocking, which you should check out if you want to be proficient in TDD.

Thanks for reading!