Unit Testing and Test-Driven Development

Charles Boyd

email@charlesboyd.me

charlesboyd.me

April 2016

Presentation Outline

- Unit Testing
  - The problem (with PyTest example)
  - Unit test definition and concepts
  - Example unit tests (with Jasmine)
  - Unit testing in the software pyramid
- Test-Driven Development (TDD)
  - TDD definition and concepts
  - TDD process and steps
  - Three Laws of TDD
- Unit Testing and TDD Compared
- Tools for Unit Testing and TDD
- Live Demo: Unit Testing with TDD
  - JavaScript + Jasmine (TDD walkthrough)
  - Python + PyTest (Brief example)
- Related Development Practices and Concepts
- Using Unit Testing and TDD
  - Writing Good Unit Tests
  - Benefits and Advantages
  - Special Considerations
  - Notes and Tips
- Dive Deeper
  - Resources and Further Reading
  - Setting up PyTest
  - Review Questions
- Questions and Discussion

The Problem

```
def add_one(x):
    return x + 1
```

How can we test this one function before it is used elsewhere in a program?

What if it was more complex?

What if it was in an extremely large system?

What if we wanted to test it automatically so when it's modified, we can easily make sure it still works?

Answer: "Unit Testing"

Source Code

Unit Tests

```
import pytest

def test_example_1():
    assert add_one(3) == 4

def test_example_2():
    assert add_one(-3) == -2

def test_example_3():
    assert add_one(5) == 20
```

Will Pass

A single "assertion"

A test title
(Should be more descriptive…)

Will Pass

Will Fail
(Need to fix this test…)

Unit Test

- A test that ***invokes a small, testable unit of work*** in a software system and then ***checks a single assumption*** about the resulting output or behavior.
- Key concept: ***Isolation*** from other software components or units of code
- Low-level and focused on a ***tiny part*** or "unit" of a software system
- Usually written by the ***programmers*** themselves using common tools
- Typically written to be fast and run with other unit tests in automation
- Form of ***white-box*** testing that focuses on the implementation details
- Typically uses coverage criteria as the exit criteria
- Definition of a "unit" is ***sometimes ambiguous***
  - A unit is commonly considered to be the "smallest testable unit" of a system
  - Object-oriented (OO) languages might treat each object as a unit
  - Functional or procedural languages will likely treat each function as a unit
  - Many testing frameworks allow sets of unit tests to be grouped, allowing tests to be targeted at the function level and grouped by their parent object

References: Some concepts from *martinfowler.com/bliki/UnitTest.html* and *artofunittesting.com/definition-of-a-unit-test/*

Example: Unit Tests (JavaScript + Jasmine)

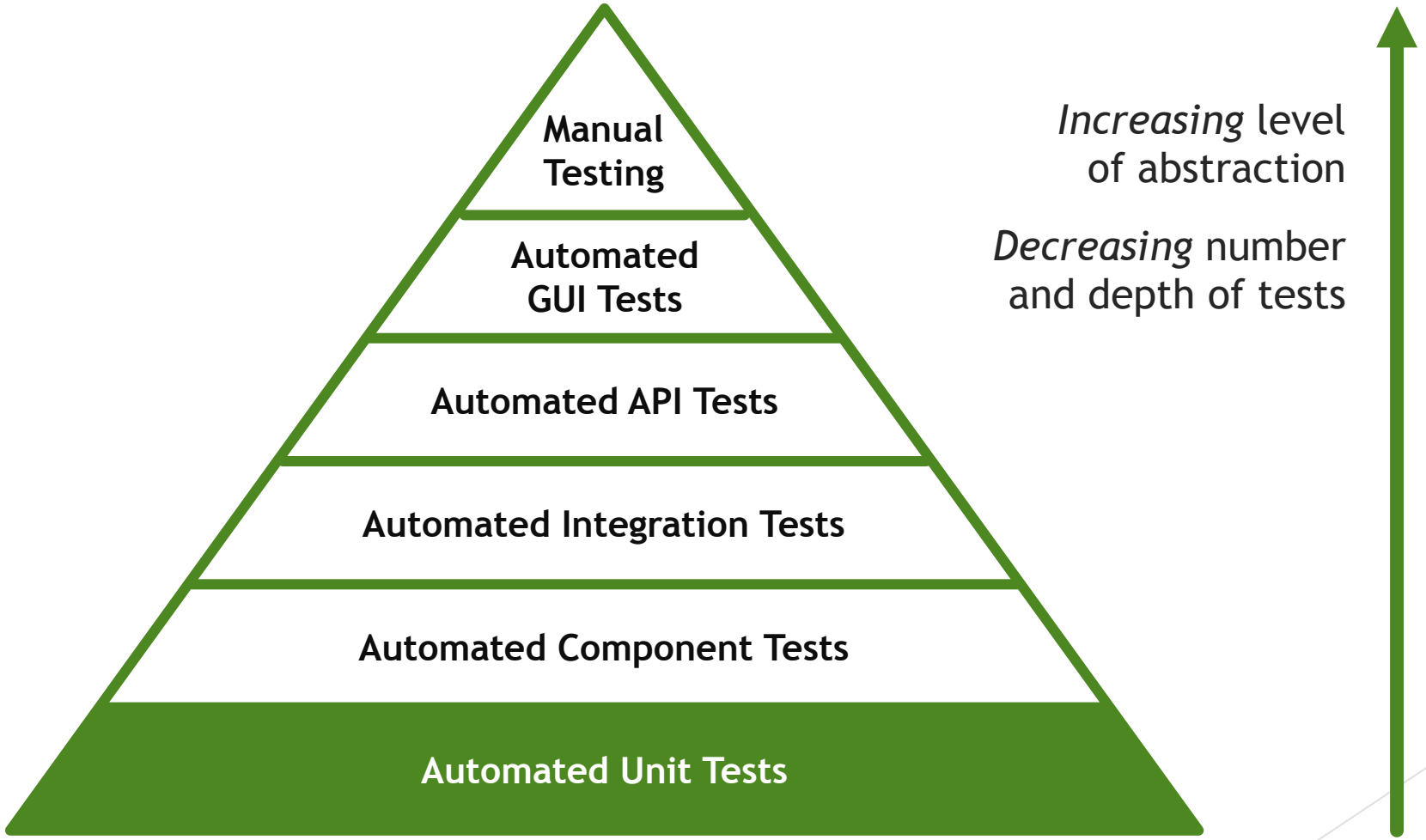
```
function isPositive(x){
    return x >= 1;
}
```

The unit tests revealed one or more defects in this code; important because other functions may rely on this one.

Source Code

Results	Unit Tests
✓	`expect(isPositive(5)).toEqual(true);`
✓	`expect(isPositive(-5)).toEqual(false);`
✓	`expect(isPositive(1)).toEqual(true);`
✓	`expect(isPositive(-1)).toEqual(false);`
✗	`expect(isPositive(0.5)).toEqual(true);`
✓	`expect(isPositive(-0.5)).toEqual(false);`
✗	`expect(isPositive(0)).toEqual(true);`
✗	`expect(isPositive('!')).toThrowError(TypeError);`

Software Testing “Pyramid”



References: Based on *watirmelon.com/tag/testing-pyramid/*

Test-Driven Development (TDD)

- A software development process where a unit's ***tests are written before*** the unit's implementation and also guide the unit's development as the tests are ***executed repeatedly*** until they succeed, signaling completed functionality.
- The TDD process steps are commonly shortened to "Red, Green, Refactor"
- Should be used each time a new function, feature, object, class, or other software unit will be developed

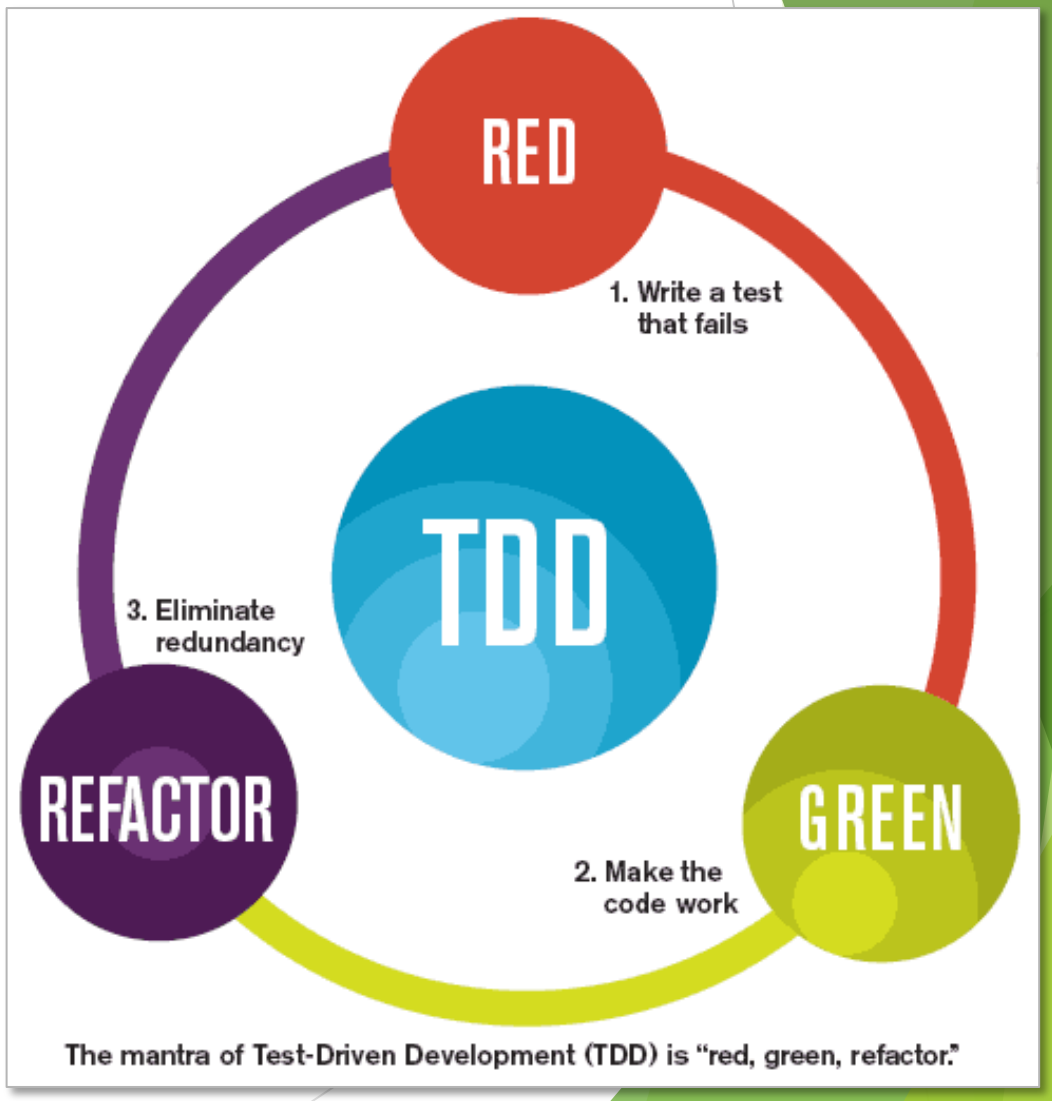


Image from *abhishekmulay.com/blog/2014/12/13/javascript-unit-testing-with-jasmine/*

Test-Driven Development (TDD)

1. **Red**
   A. **Write a new test** for a section of code (the "unit")
   B. **Verify failure** of the *new* test and the success of *existing* tests. If the new test passes immediately, verify that it is not redundant, then start over at step 1a.
2. **Green**
   A. **Write some code** to implement, modify, or develop the unit
   B. **Repeat until the tests pass.** If one or more tests fail, continue coding until all tests pass. If the tests pass, the developer can be confident that the new/modified code works as specified in the test. Stop coding immediately once all tests pass.
3. **Refactor**
   A. **Refactor** the code to improve non-functional code structure, style, and quality
   B. **Confirm tests pass.** If one or more tests fail, the refactor caused problems; edit until all tests pass. If the tests all pass, the developer can be confident that the refactor did not affect any tested functionality.
4. **[Repeat]**

Uncle Bob's Three Laws of TDD

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

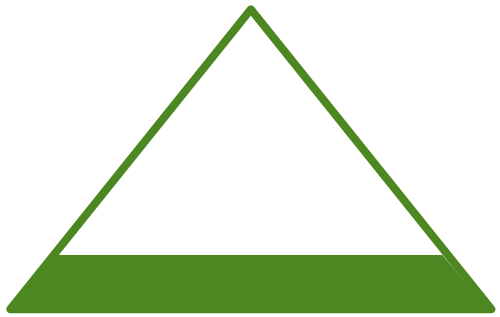
References: From *butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd*

Unit Testing vs TDD

- Unit testing and TDD are distinct concepts
- While closely related and often used together, they could be used separately
- The following slides and demos present the two concepts combined, as they are frequently used together

Unit Testing

Level of testing



Test-Driven Development

Development *process*

1. --------
2. --------
3. --------

Tools

- **Test Framework**
  - Defines the syntax of the tests, provides the function libraries required to write the tests, and includes a basic method to execute the tests
  - Likely language-specific
  - Examples: Jasmine (for JavaScript), PyTest (for Python), JUnit (for Java)
- **Test Runner**
  - Executes all (or a specific subset) of the system's unit tests and presents, displays, or otherwise outputs the results
  - Could be a local test runner on a developer's computer or run on a server (e.g., a CI server)
  - Might also spin up mocks, a virtual environment, or any other resources the tests require
  - A basic test runner is typically built into the test framework itself, possibly run from a command line interface
  - Example: Karma (for web application testing)

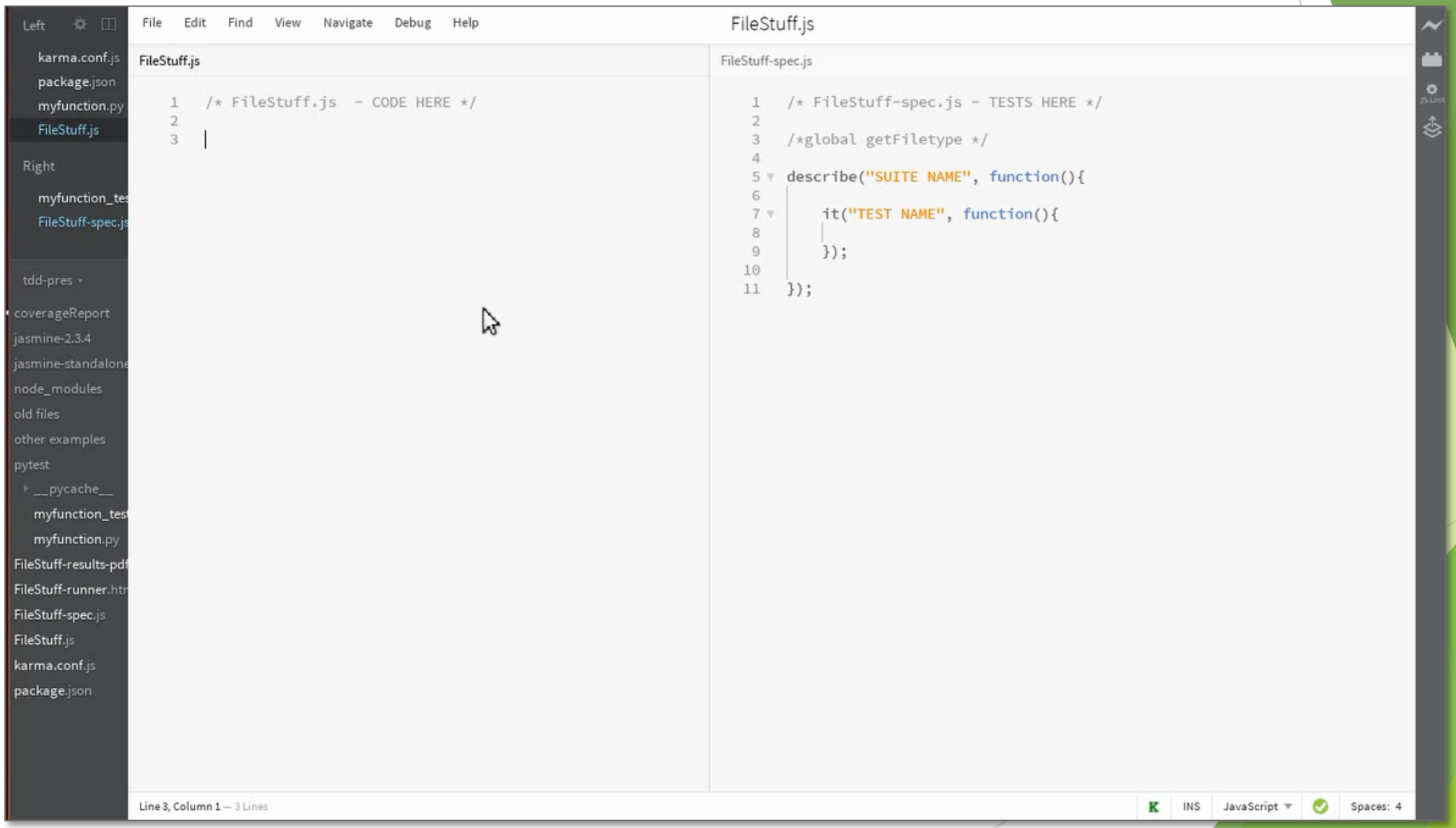
Tools (cont.)

- **Mocks**
  - Provides a “mock” or simulated implementation of each external dependency or resource required by the methods being tested.
  - May return random, dummy, or cached data
  - The need for mocks and their implementations varies between systems
  - Also known as “subs” or by several other names.
  - Example: A simple program that returns dummy (fake but realistic) data for each outbound request made by a system that uses the Twitter API
- **Coverage Reporter**
  - Determines and provides a report on the test coverage metrics of a set of code
  - Might generate metrics such as statement, branch, function, executions per line, and line coverage grouped by file, class, component, or for the entire system
  - Might be run independently or during each test executed by a test runner
  - Example: Istanbul (for JavaScript)

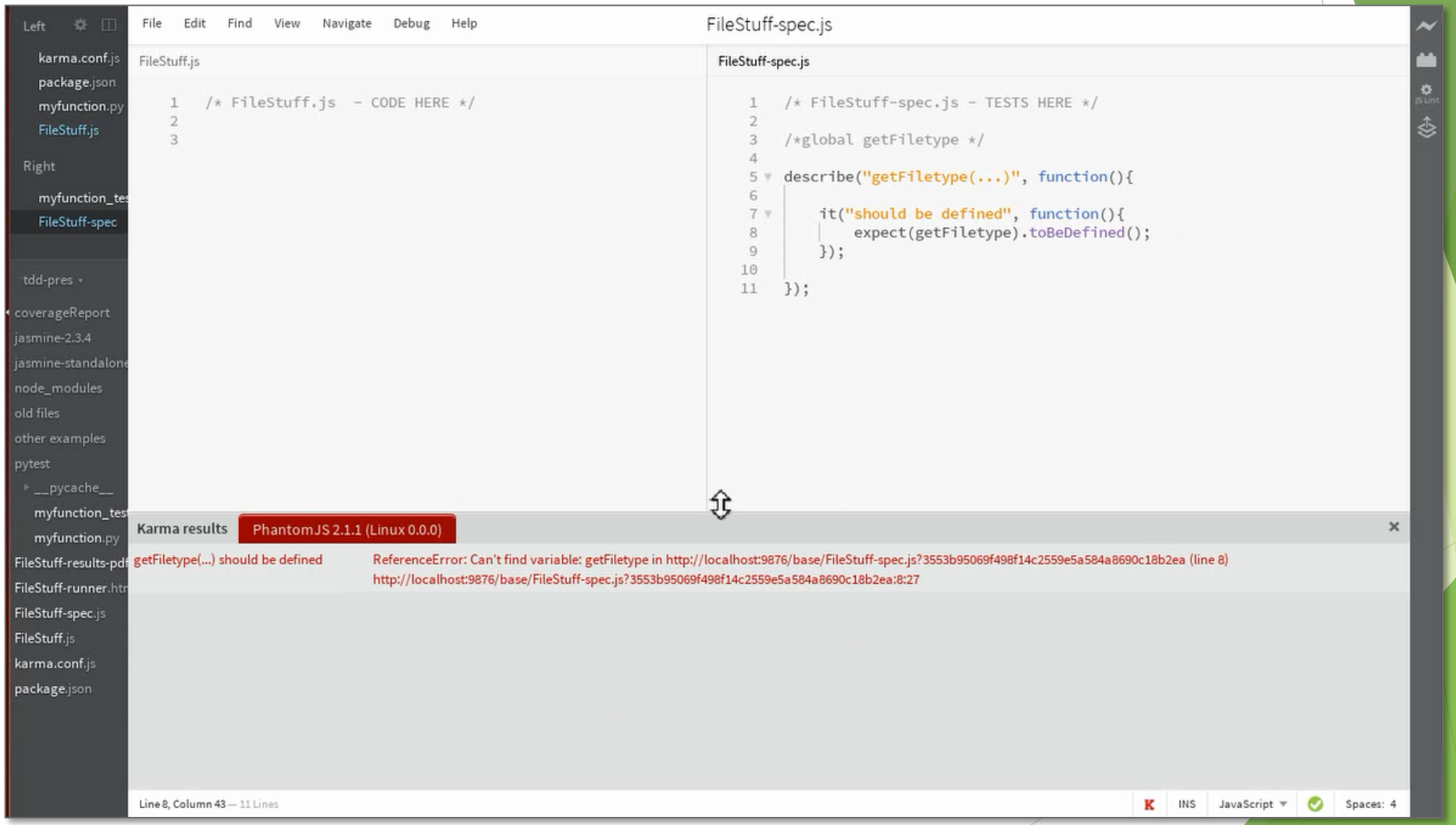
Demo

JavaScript + Jasmine Unit Test Framework

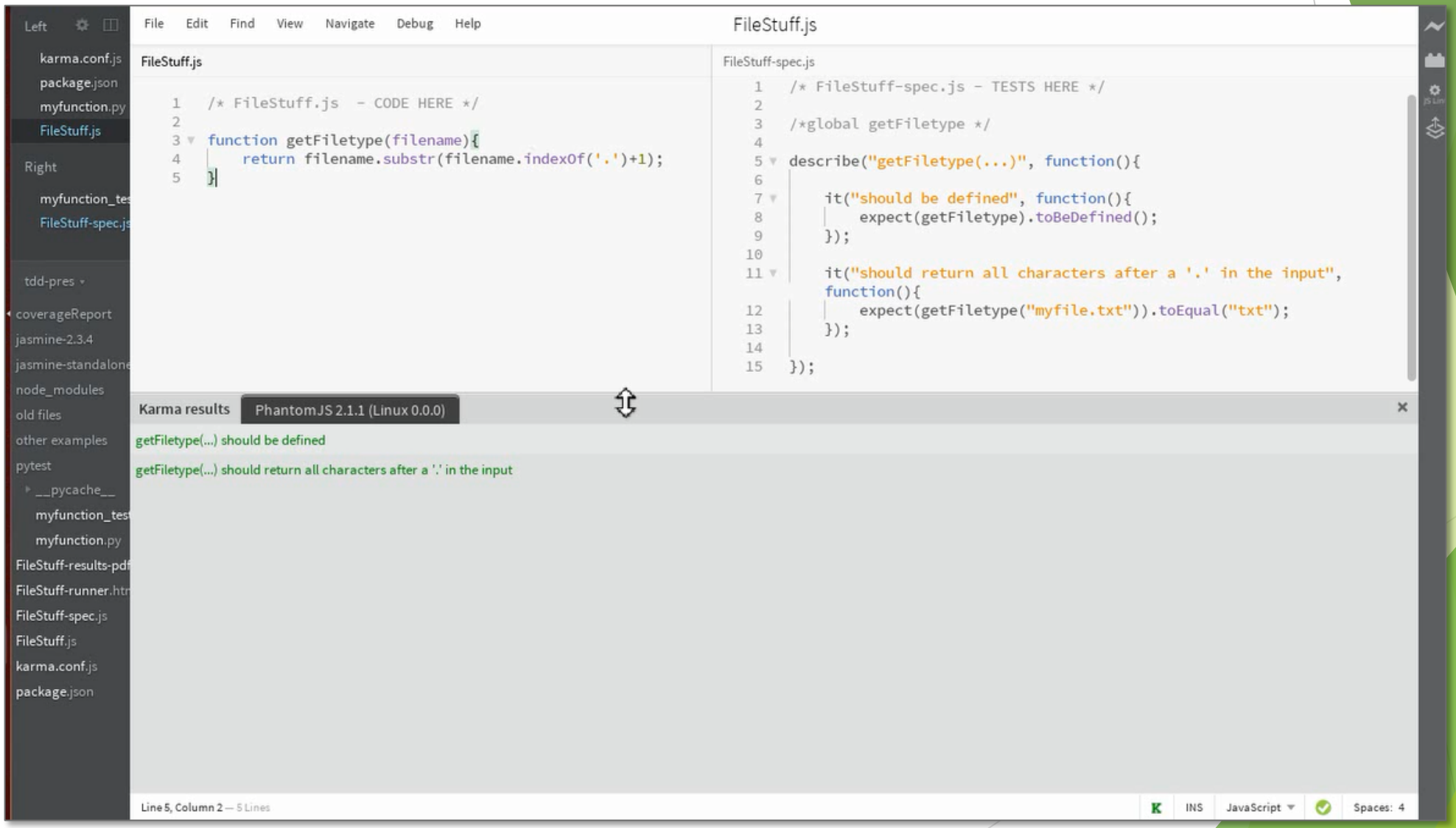
Demo Screenshots: Starting Blank



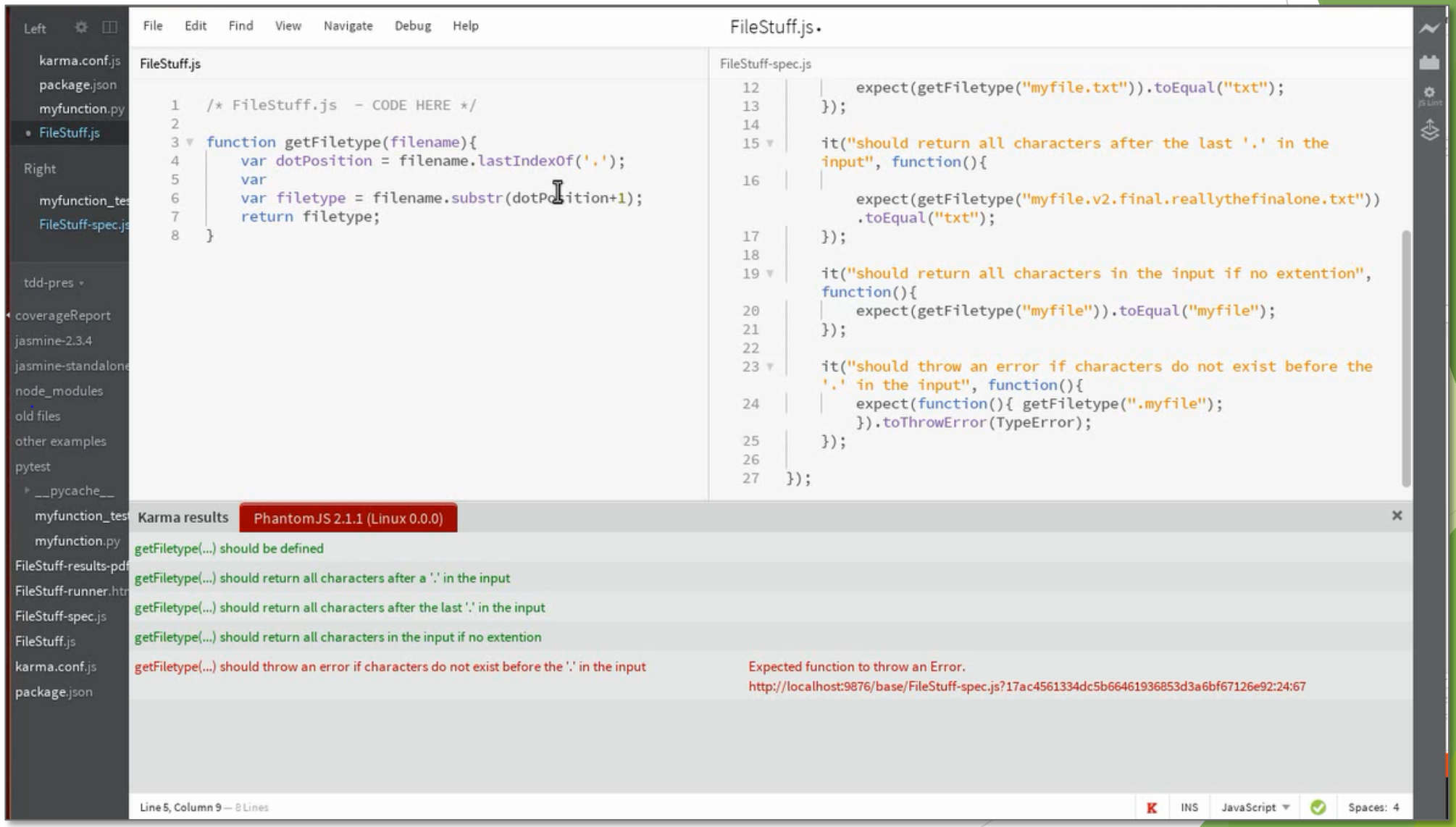
Demo Screenshots: Writing a Test First



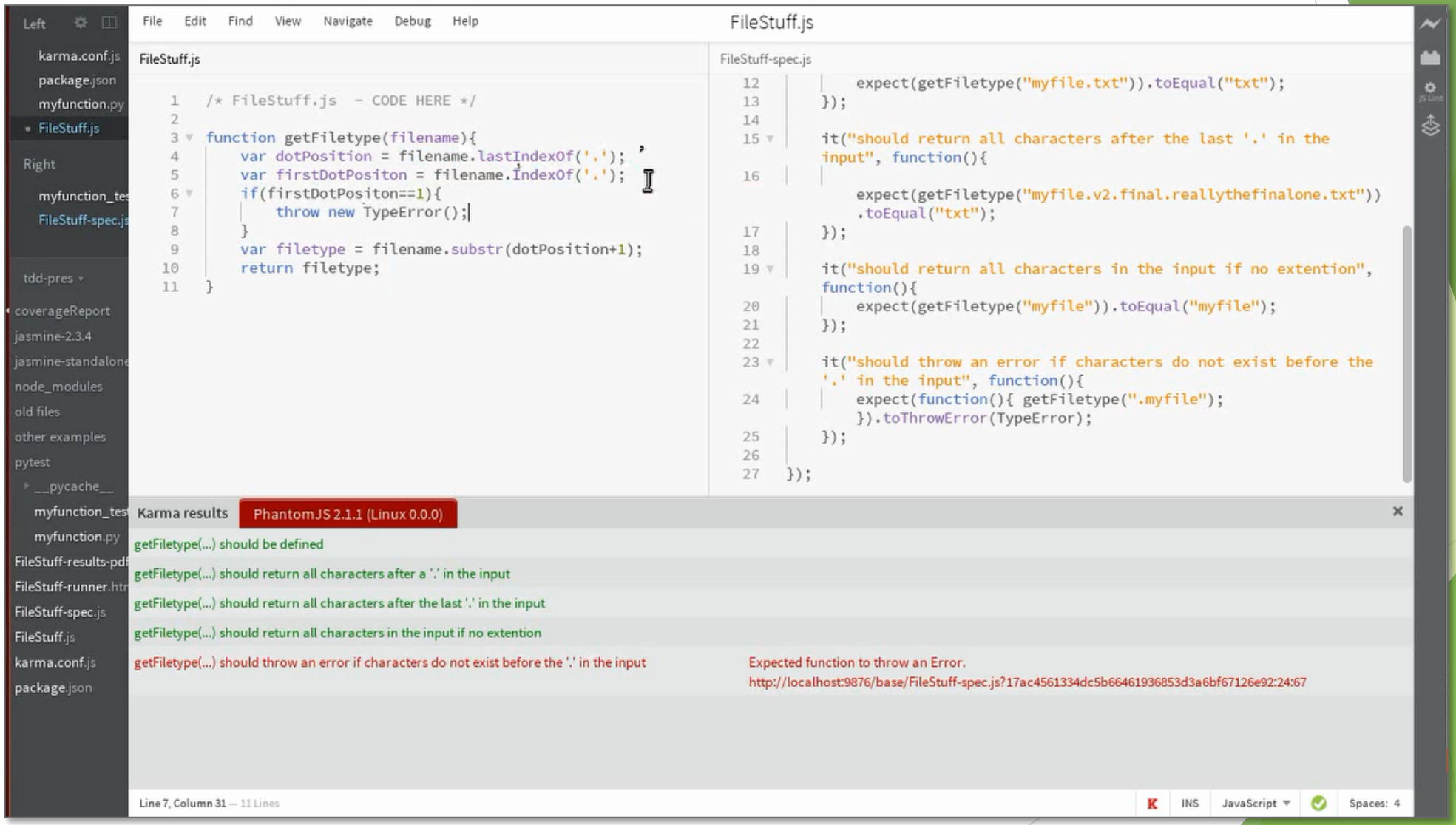
Demo Screenshots: Before Refactor



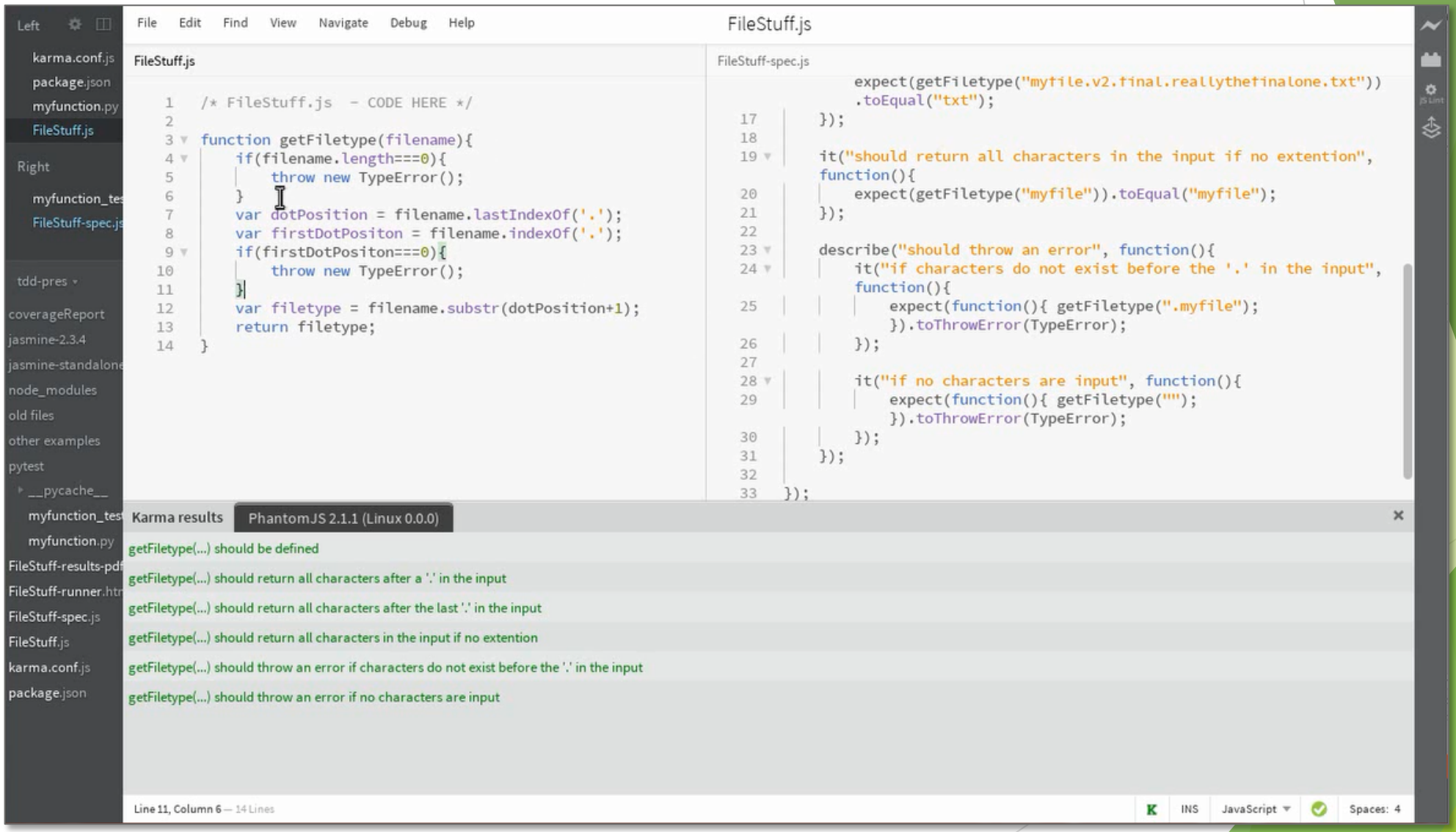
Demo Screenshots: Writing Code



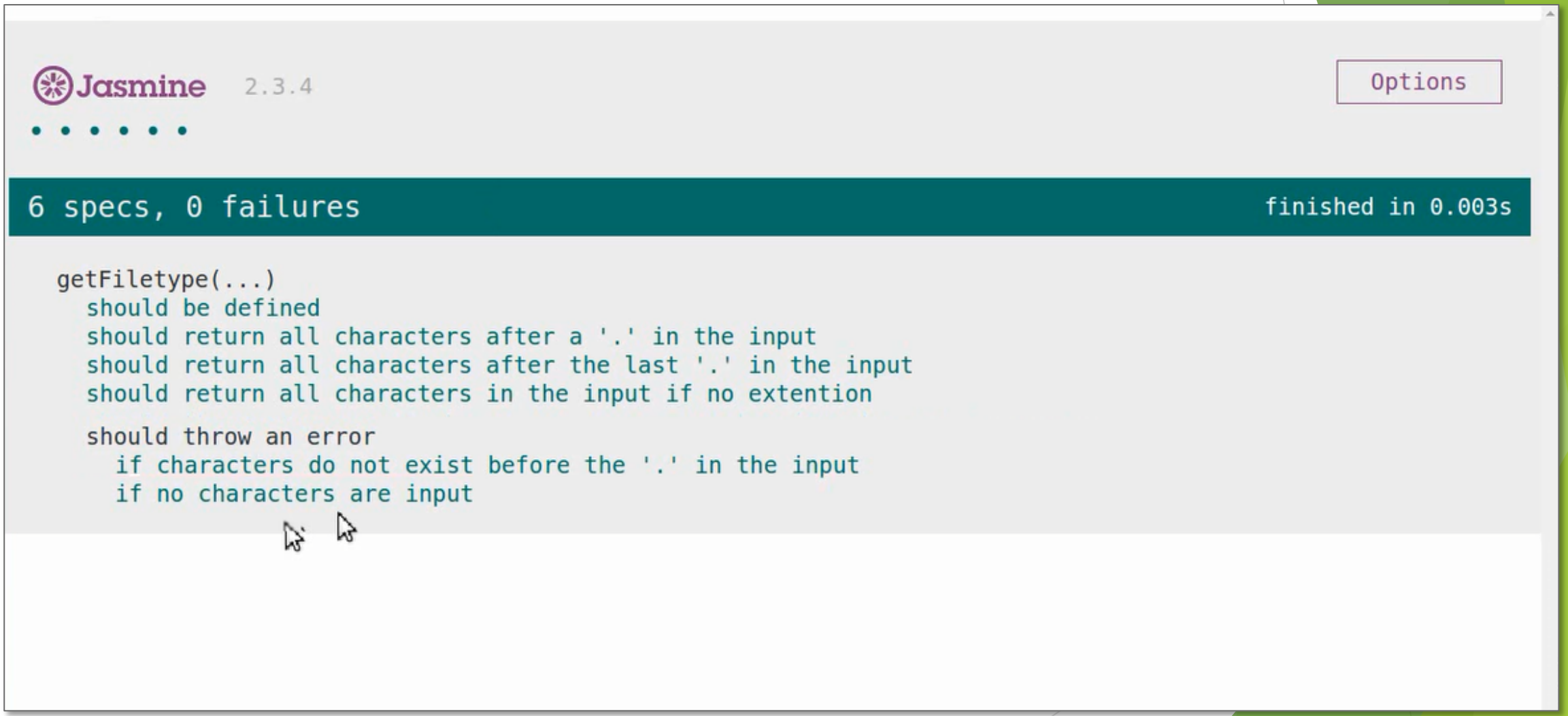
Demo Screenshots: Tests Catch a Defect



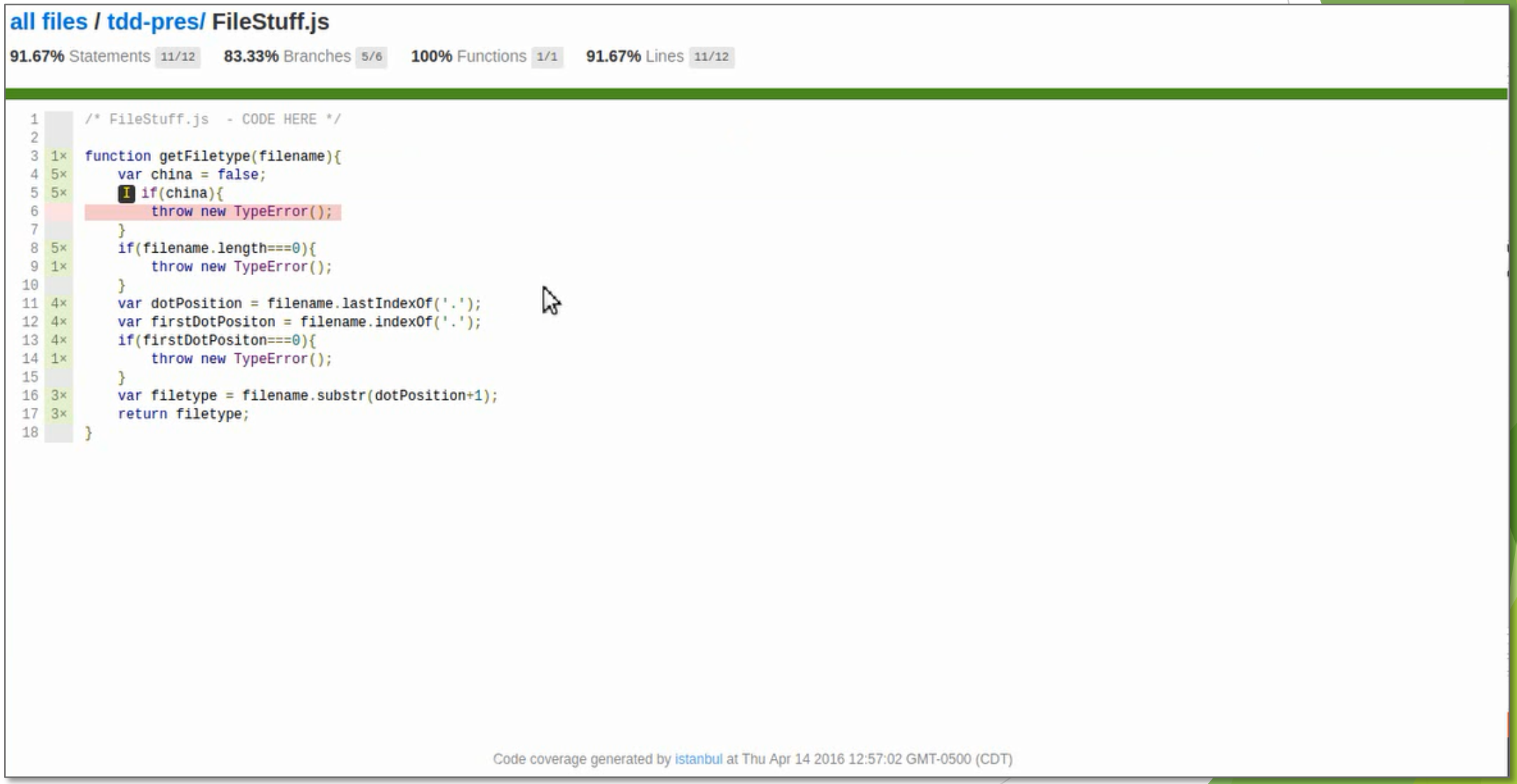
Demo Screenshots: Green so Far



Demo Screenshots: Jasmine Output



Demo Screenshots: Unreachable Branch Coverage



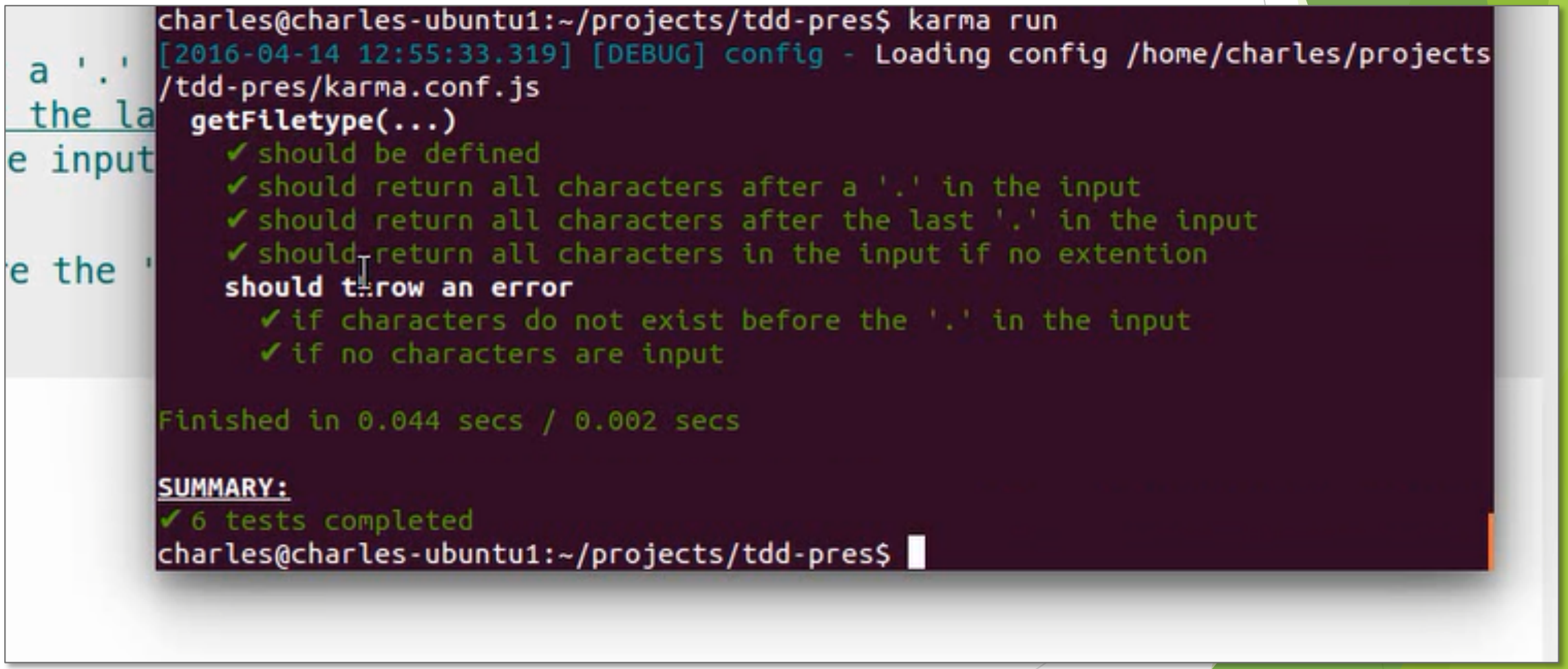
all files / tdd-pres/ FileStuff.js

91.67% Statements 11/12 83.33% Branches 5/6 100% Functions 1/1 91.67% Lines 11/12

```
1    /* FileStuff.js  - CODE HERE */
2
3  1× function getFiletype(filename){
4  5×     var china = false;
5  5×     if(china){
6              throw new TypeError();
7          }
8  5×     if(filename.length===0){
9  1×         throw new TypeError();
10         }
11 4×     var dotPosition = filename.lastIndexOf('.');
12 4×     var firstDotPositon = filename.indexOf('.');
13 4×     if(firstDotPositon===0){
14 1×         throw new TypeError();
15         }
16 3×     var filetype = filename.substr(dotPosition+1);
17 3×     return filetype;
18 }
```

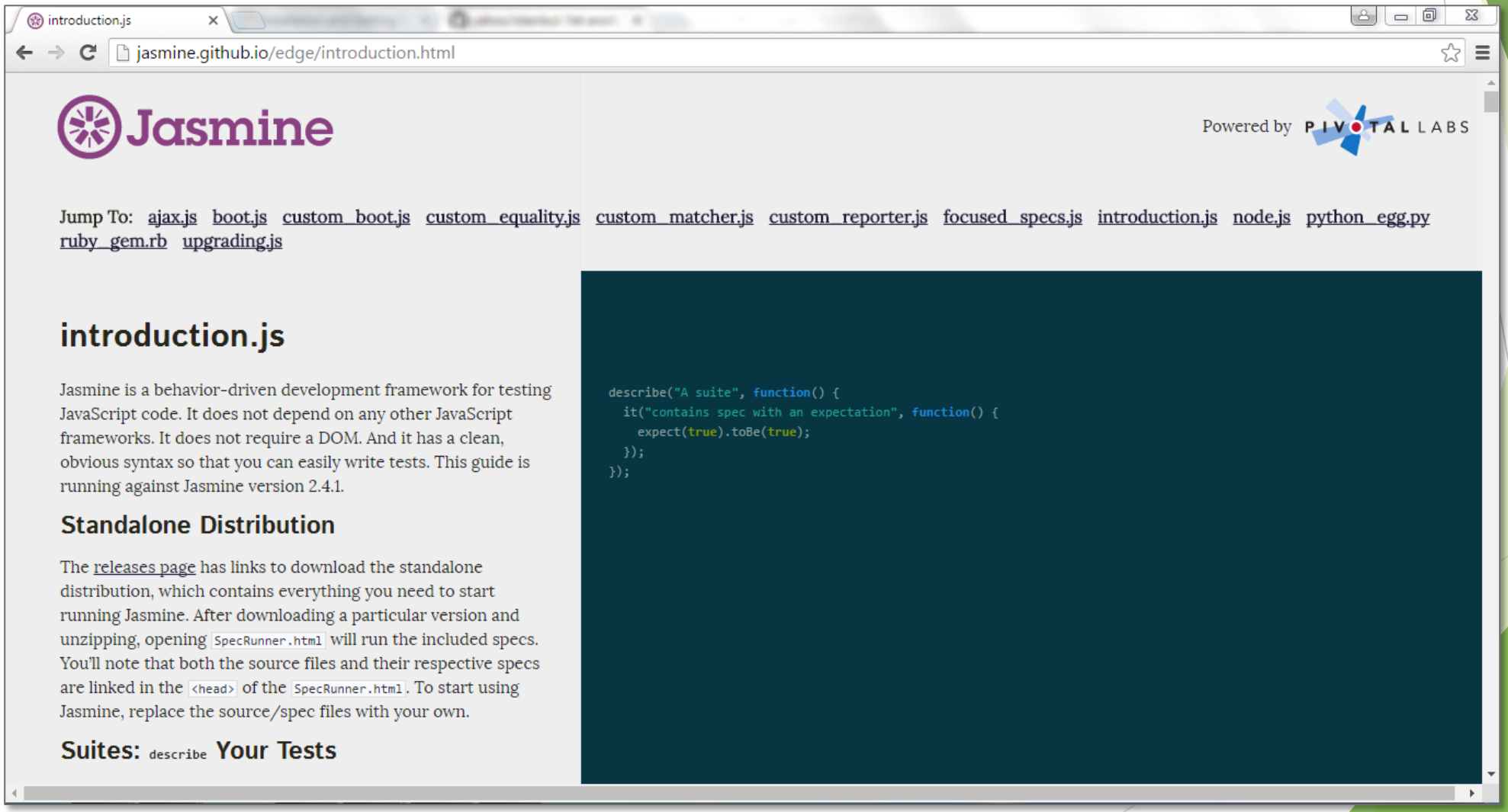
Code coverage generated by istanbul at Thu Apr 14 2016 12:57:02 GMT-0500 (CDT)

Demo Screenshots: Test Output in Terminal



Demo Screenshots: Jasmine Docs

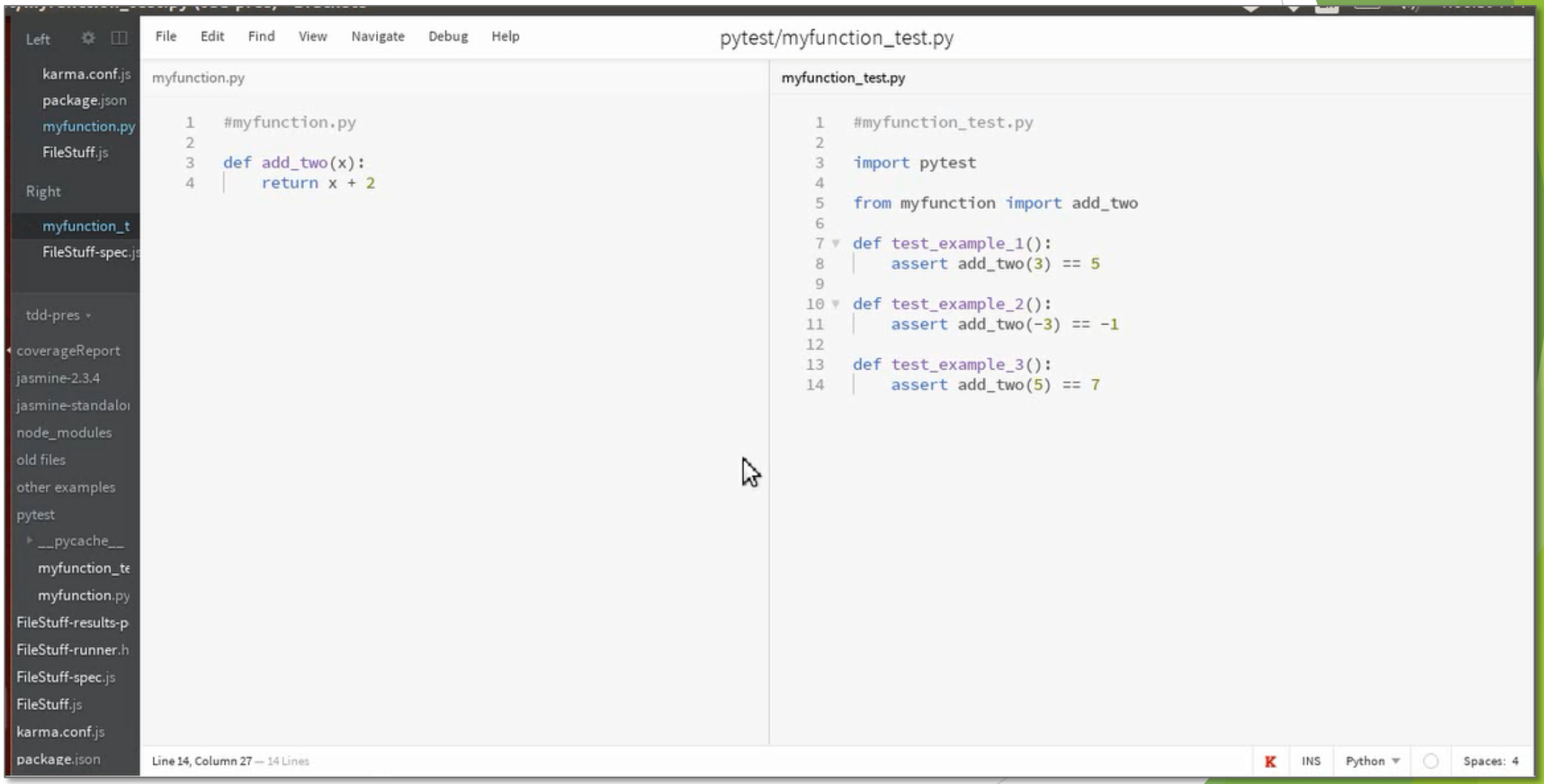
View in Browser



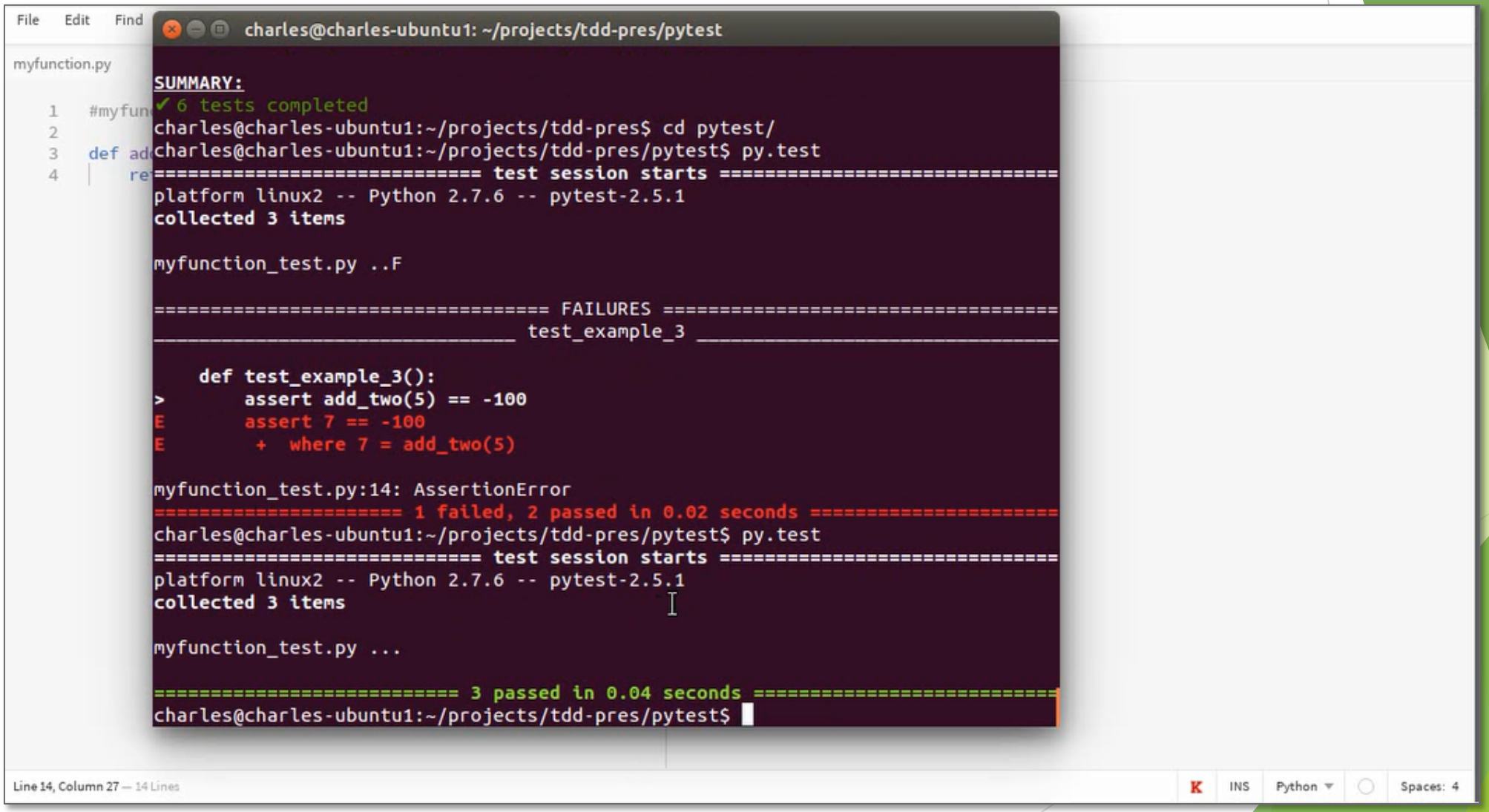
Demo

Python + PyTest Unit Test Framework

Demo Screenshots: Basic PyTest



Demo Screenshots: Running PyTest



Related Development Practices and Concepts

Practicing unit testing and TDD lends itself to other popular software engineering concepts and can encourage their adoption.

- **Regression Testing**
  - The type of software testing that verifies changes to a set of code
  - Checks that a previously-written and previously-tested set code still works correctly after changes are made directly to it or to any of its dependencies
- **Automated Testing**
  - The use of pre-scripted tests on a software system which can be run repeatedly to return consistent and useful results
  - Greatly streamlines regression testing
- **Continuous Integration (CI)**
  - A software engineering process where isolated code changes are immediately tested and reported on as they are added to a larger codebase
  - Developers incorporate their progress and code changes to a common, centralized codebase daily (or more frequently)
  - The goal is to provide rapid feedback to identify and correct defects as soon as they are introduced
  - Requires dedicated tools and for automated tests to exist for the system

References: CI information based on *http://searchsoftwarequality.techtarget.com/definition/continuous-integration*

Related Development Practices and Concepts (cont.)

- **Iterative and Incremental Development**
  - Iterative development is a software development process where code is designed, developed, and tested in very short, repeated *cycles* by breaking a larger component into smaller units
  - Incremental development is a process where a *single piece* is built at one time
  - Utilized during TDD in a very small scale, where units are added *incrementally* and each unit is developed *iteratively* with the 3-step TDD process
  - Key components of Agile Methodology; often used in a larger scale for the entire system
- **Single-Purpose Methods**
  - A design strategy where all "methods" or "functions" in the system have a "single purpose," such that they perform only one task, are given descriptive names, and are kept very short by frequently calling other methods to do work
  - Any long set of instructions may be broken up into small, focused, and descriptive methods. Each new method is then called by one or more overarching methods.
  - Allows unit tests to be written for a single unit of code more easily
  - For object-oriented design, also utilize the associated "single responsibility principle"

Good Unit Tests…

- Test a **single logical concept** in the system
- Have full control over all the pieces running and uses mocks or stubs to achieve this **isolation** as needed
- Are able to be fully **automated**
- Can be run in **any order**
- Return a **consistent** result for the same test (For example, no random numbers; save those for broader tests)
- Cause **no side effects** and limits external access (network, database, file system, etc.)
- Provide **descriptive** and trustworthy results
- Run **fast** for rapid feedback
- Are written with **readable** and **maintainable** test code, have descriptive test names, and are organized or grouped logically

References: Adapted from *http://artofunittesting.com/definition-of-a-unit-test/*

Benefits of Unit Testing with TDD

- ***Greatly improved regression testing!***
- Allows refactoring without the fear of breaking the code
- Builds the automated unit test suite automatically
- Provides a easy way to test a newly written unit without the burden of the entire program or system
- Produces a short feedback loop and rapid iterations
- Forces the developer to plan ahead
- Potentially reduces development time (in the long-term)
- Helps to ensure new code "works" (to the extent of the tests) *as* it is built
- The tests can serve as a detailed specification for the main code
- Provides some indication of development progress
- Goes hand-in-hand with several other key software development practices
- Allows the developer to always know that a minute ago, "everything worked"

References: Adapted from *http://sd.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first/ and http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd*

Example: Benefit of Unit Testing

```
function absoluteValue(x){
    if(isPositive(x)==false){
        x = x*-1;
    }
    return x;
}
```

The function `isPositive(…)` is now covered by its own unit tests. So, here we trust that it works as its tests specify.

Note: To encourage isolation, you may wish to create mocks for calls to external functions

Source Code

Unit Tests

✓ `expect(absoluteValue(5)).toEqual(5);`

✓ `expect(absoluteValue(-5)).toEqual(5);`

✓ `expect(absoluteValue(1)).toEqual(1);`

✓ `expect(absoluteValue(-1)).toEqual(1);`

✓ `expect(absoluteValue(0)).toEqual(0);`

✓ `expect(absoluteValue('!')).toThroughError(TypeError);`

Considerations: Unit Testing + TDD

- **Can seem to double development time (in the short run)**
  - Practicing TDD can feel like it nearly doubles development time because the developer is writing about twice the amount of code
  - Long term, however, the previously mentioned benefits of TDD can ultimately save development time
  - This issue is sometimes the subject of debate when following TDD
- **Difficult to unit test functions with "side effects" or time-delays**
  - Functions that cause irreversible or otherwise unmockable side effects (such as a physical action with imprecise feedback) may be difficult to unit test
  - To test functions that perform an action but do not return a value, the unit test must check the proper existence/nonexistence of the side effect, which can become difficult
- **No tests for the tests**
  - Unit tests themselves are code and can be written incorrectly.
  - They might fail at first (as required by TDD), but never become "green" or successful (or successful too early) during the "write code" step because the test itself has one or more faults.

Considerations: Unit Testing + TDD (cont.)

- **Special considerations for use with graphical user interface (GUI) development**
  - No clear input and output like a function, so each GUI element must have a constant, descriptive identification to inject actions or verify results
  - Difficult to achieve isolation because some GUI actions cause "behind the scenes" actions. So, to verify success, the unit test must tap into extra units/components.
- **TDD doesn't *always* assist with changes in requirements, system-wide code structure, or some application programming interfaces (APIs)**
  - Especially in the early stages of a project, refactoring or other changes can be so extensive that it needs to "break" some tests.
  - Those tests need to be modified/rewritten, but the corresponding main code for them might already exist, so the new tests are written without following TDD. Remember, however, that TDD is focused on *code base* development, not *test* development
  - Existing unit tests are beneficial during most changes, particularly if a dependency is modified. The tests for each component that relies on that modified dependency can be run to ensure the change didn't break the higher-level component. This doesn't help as much if the dependency's prototype or public API or is changed though.
- **Desire to make all tests "green"**
  - The desire to see a passing or "green" result on all tests can cause developers to skip necessary tests or remove broken tests that should instead be fixed

Considerations: Unit Testing + TDD (cont.)

- **Mocks must be developed and constantly updated**
  - Each external resource or dependency must be mocked to ensure isolation, and this process can become time-consuming
  - Unless the mock is built in/with the actual implementation of a dependency, the mock must be constantly updated as the API of the dependency
- **Sometimes difficult to test "private" functions/methods**
  - Sometimes, even when treating an object/class as a "unit," private functions (particularly complex ones) need to be unit tested individually. This means, depending on the language and/or testing framework, that the private function(s) must be exposed in some manner, violating object-oriented design principles.
- **TDD does not *necessarily* result in "quality" tests and never *guarantees* proper code**
  - Developers might skip edge cases, might write an untested branch by accident, etc.
  - TDD is not designed to build the best tests, it's designed as a development process
- **Tedious**
  - Sometimes TDD can feel like it gets in the way of "just coding."
  - Advice: Try it anyway

Notes/Tips: Unit Testing and TDD

- Remember that TDD is “Test-Driven Development” and focuses on driving the development of the main **codebase, not the development of tests.**
- Get a **good IDE** (integrated developer environment) that allows you to see your code and unit tests side-by-side. For JavaScript, check out Brackets.
- Use a test runner that can **automatically run** your tests when you save.
- Since TDD fits well with several other related development practices (CI, automated testing, iterative/incremental development, Agile, and more), consider **implementing these practices as well.**
- Remember the “Three Laws of TDD” and follow them, but remember that best practices are not always “laws.” Software development can sometimes be more of **an art**.
- Always utilize TDD and unit test code that may be used as production code in the future, **even if just exploring** an idea. Concept code or rough GUI mockups may not require TDD because they are short-lived, but it may help.
- Unit testing and TDD are **not perfect**, but are still very effective practices.

Resources and Further Reading

- Popular Unit Testing Frameworks:
  - Python: pytest (view Getting Started)
  - JavaScript: Jasmine (+ Karma test-runner with a GUI-less browser such as PhantomJS)
  - Java: JUnit
  - List of unit testing frameworks, Wikipedia
- Tips, Tutorials, and Interesting Articles
  - UnitTest by Martian Fowler, software development author/speaker
  - How I Started TDD, by Gary Bernhardt
  - The Three Rules of TDD, Uncle Bob
  - TDD Example Walkthrough (Java), Technology Conversations
  - Testing Culture by Mike Bland, former Googler
  - In TDD, Writing Passing Test Cases Without Production Code
  - Why I Test Private Functions In JavaScript by Philip Walton, Google Engineer
  - Small, Medium, Large by Mike Bland, former Googler
  - Test-Driven Development, By Example, book by Kent Beck
  - What are the key principles of TDD?, by Marc Abraham

PyTest Setup Instructions for Ubuntu (with Terminal)

1. **Python**
   - Check if installed: `python --version`
   - Install if needed: `sudo apt-get install python`
2. **PyTest**
   - Check if installed: `py.test --version`
   - Install if needed: `sudo apt-get install python-pytest`

PyTest Setup Instructions for Windows (with cmd.exe) and other OSs

1. **Python**
   - Check if installed: `python --version`
   - Install if needed: Download at python.org/downloads (Check "add to PATH" when installing)
2. **pip (a Python package manager)**
   - Check if installed: `pip --version`
   - Install if needed: Download bootstrap.pypa.io/get-pip.py then run it (`python get-pip.py`)
2. **PyTest**
   - Check if installed: `py.test --version`
   - Install if needed: `pip install -U pytest`

Your steps may vary by operating system, configuration, previous installs/versions, and/or other factors. *Please research before raising concerns*, but speak up if there is truly an incorrect step above.

Review Questions

- **1. Which is a central concept to unit testing?**
  - a. Declarative
  - b. Isolation
  - c. Standardized
  - d. Multilayered
- **2. What does TDD stand for?**
  - a. Test Declarative Development
  - b. Test Defensive Deduction
  - c. Test-Driven Development
  - d. Test-Defect Deduction
- **3. What is the first step in TDD?**
  - a. Write tests
  - b. Run new unit tests
  - c. Test defect identification
  - d. Code implementation
- **4. Which of the following is *not* one of the listed attributes of a good unit test?**
  - a. Fast Execution
  - b. Consistent Results
  - c. Trustworthy Results
  - d. Dynamic Implementation
- **5. A mock is a(n) _____.**
  - a. Simulated implementation
  - b. Type of unit testing framework
  - c. Offensive test
  - d. Poorly written or structured test

Review Questions – Answers

- **1. Which is a central concept to unit testing?**

  a. Declarative
  c. Standardized
  b. Isolation
  d. Multilayered

- **2. What does TDD stand for?**

  a. Test Declarative Development
  c. Test-Driven Development
  b. Test Defensive Deduction
  d. Test-Defect Deduction

- **3. What is the first step in TDD?**

  a. Write tests
  c. Test defect identification
  b. Run new unit tests
  d. Code implementation

- **4. Which of the following is *not* one of the listed attributes of a good unit test?**

  a. Fast Execution
  c. Trustworthy Results
  b. Consistent Results
  d. Dynamic Implementation

- **5. A mock is a(n) _____.**

  a. Simulated implementation
  c. Offensive test
  b. Type of unit testing framework
  d. Poorly written or structured test

Questions and Discussion

Unit Testing and Test-Driven Development

Charles Boyd

email@charlesboyd.me

charlesboyd.me

April 2016