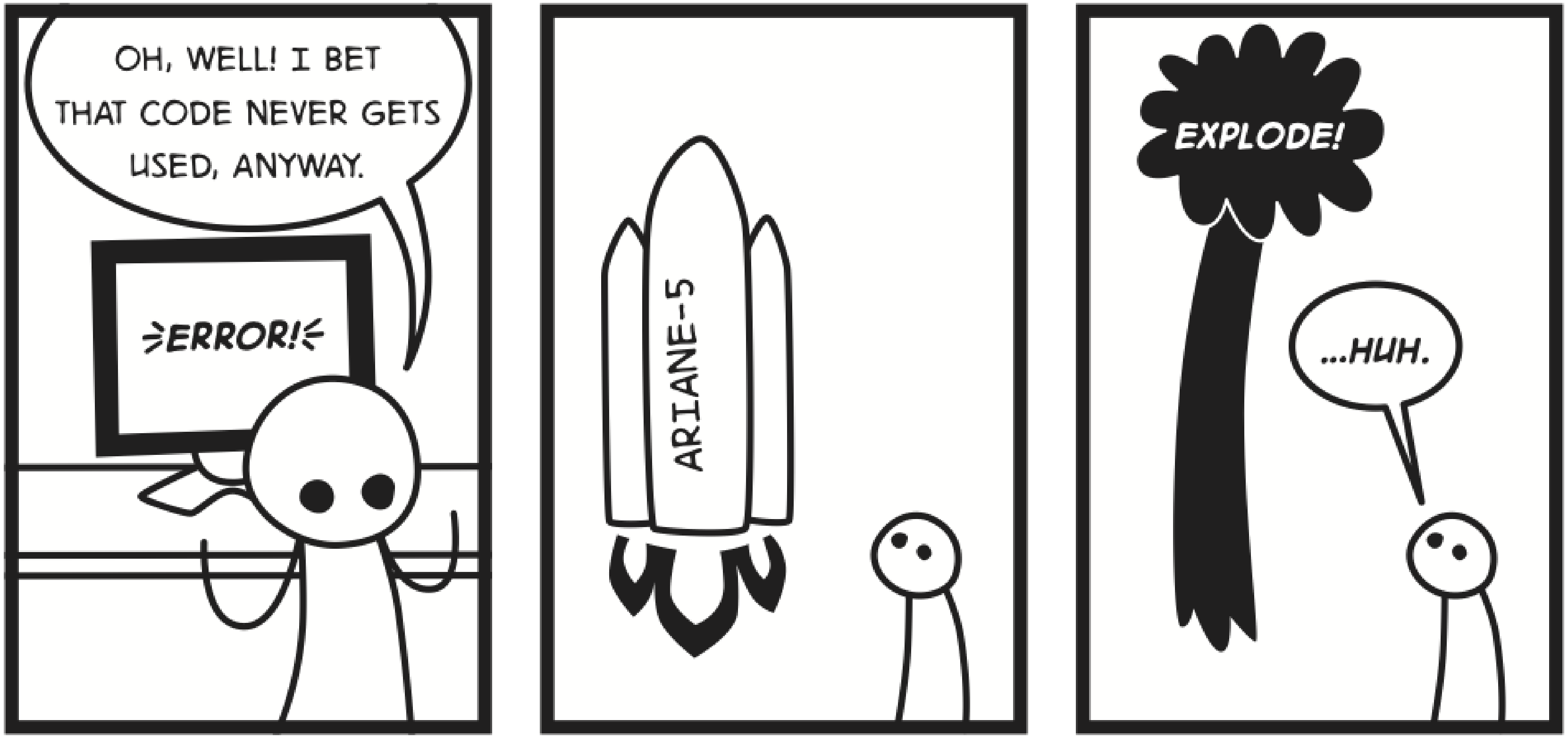
Error Handling, Debugging and Software Testing

COMP 1531, 17s2

Aarthi Natarajan

Week 11

Ariane5 First Launch Failure



Exception Handling

Exceptions

- What is an exception?
  - An **error** that happens during the execution of a program, causing a program to terminate abruptly
  - Could be caused by providing wrong input to the data, run out of memory, file or network resources not available
  - Are different to program bugs
- Exception handling enables handling such situations gracefully and avoid intermittent failures
- Exception handling is critical for creating robust and stable applications

Exception Handling in Python

In Python, when an error occurs:

- An exception is raised through creating a Python object Exception
- The normal flow of the program is disrupted
- This exception must be handled, else program terminates
- Use Python’s Try/Except clause to handle exceptions

Exception Handling in Python

Syntax of a Python <try-except-else> block:

```
try:
      You do your operations here;
      ......................
except ExceptionI:
      If there is ExceptionI, then execute this block.
except ExceptionII:
      If there is ExceptionII, then execute this block.
      ......................
finally:
      Always, execute this block.
else:
      If there is no exception then execute this block.
```

Common exceptions in Python

Exception	Occurence
IOError	If the file cannot be opened
ImportError	If python cannot find the module
ValueError	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value
EOFError	Raised when there is no input from either the input() function or when the end of file is reached.
ZeroDivisionError	Raised when division or modulo by zero takes place for all numeric types.
AssertionError	Raised in case of failure of the Assert statement.

Lecture demos

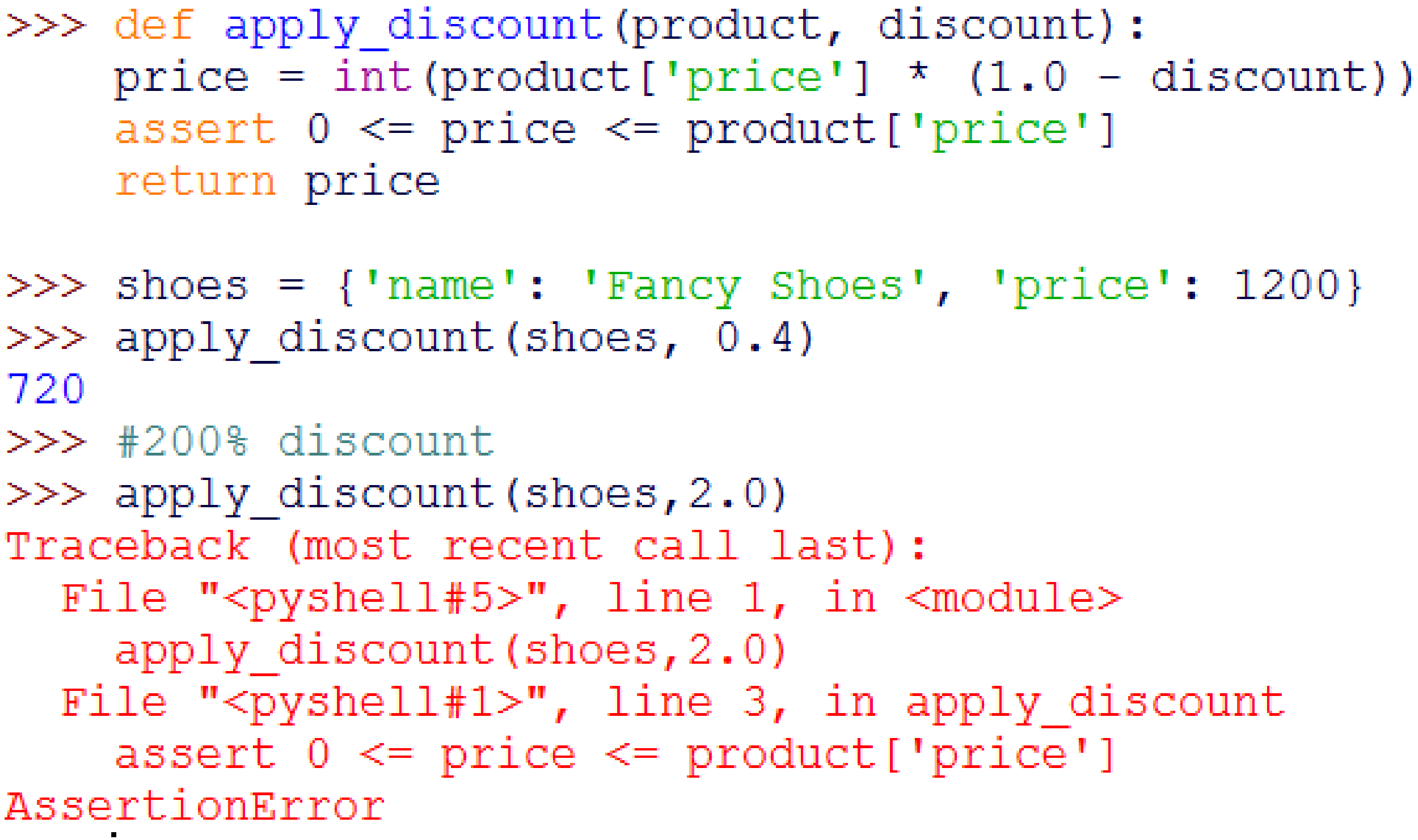
```
exception_1.py
exception_2.py
exception_3.py
```

Assert in Python

- Powerful debugging aid, to test conditions
- Use assertions as internal self-checks to identify unrecoverable errors potentially caused by a program bug
- Not a mechanism for handling run-time errors such as “file not found”
- An AssertError is raised, if the assert condition fails
- Python’s Assert Syntax:

```
assert_stmt ::= "assert" expression1 ["," expression2]
```

Lecture demo: assert_1.py

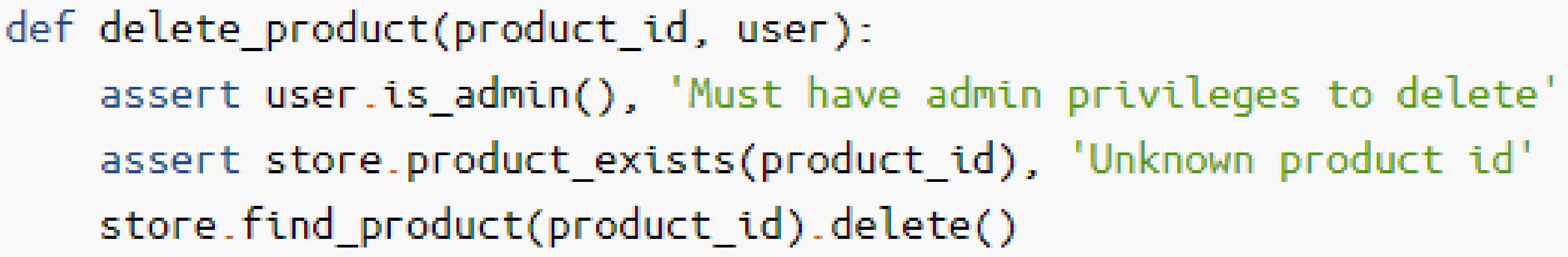


```
>>> def apply_discount(product, discount):
    price = int(product['price'] * (1.0 - discount))
    assert 0 <= price <= product['price']
    return price

>>> shoes = {'name': 'Fancy Shoes', 'price': 1200}
>>> apply_discount(shoes, 0.4)
720
>>> #200% discount
>>> apply_discount(shoes,2.0)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    apply_discount(shoes,2.0)
  File "<pyshell#1>", line 3, in apply_discount
    assert 0 <= price <= product['price']
AssertionError
```

Assert guarantees that discounted prices cannot be lower than $0 or higher than original price

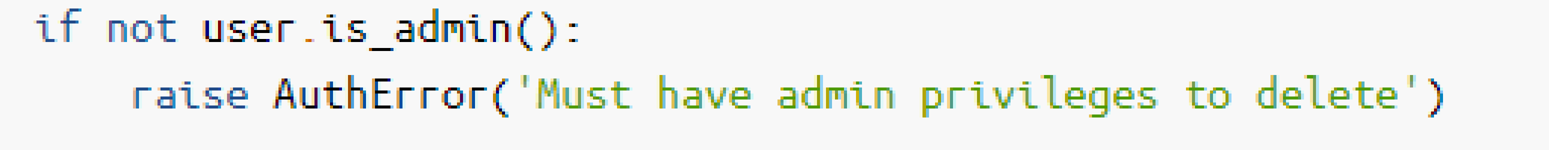
Common Pitfalls in using Assert

- Do not use asserts for data validation or data processing
- Asserts can be turned off globally
- Can cause dangerous side-effects



```
def delete_product(product_id, user):
    assert user.is_admin(), 'Must have admin privileges to delete'
    assert store.product_exists(product_id), 'Unknown product id'
    store.find_product(product_id).delete()
```

- Code above has two serious issues:
  - Checking for admin privileges with an assert statement is dangerous.
  - The product_exists() check is skipped when assertions are disabled
- Use validation exceptions



```
if not user.is_admin():
    raise AuthError('Must have admin privileges to delete')
```

- **Golden rule: Do not use assertions for data validation!**

Overview of Software Testing

A **fault**, also called “**defect**” or “**bug**,” is an erroneous hardware or software element of a system that can cause the system to fail (inadvertent bugs or malicious features)

- How do you define testing?
- And what can be tested?
- How do you test?

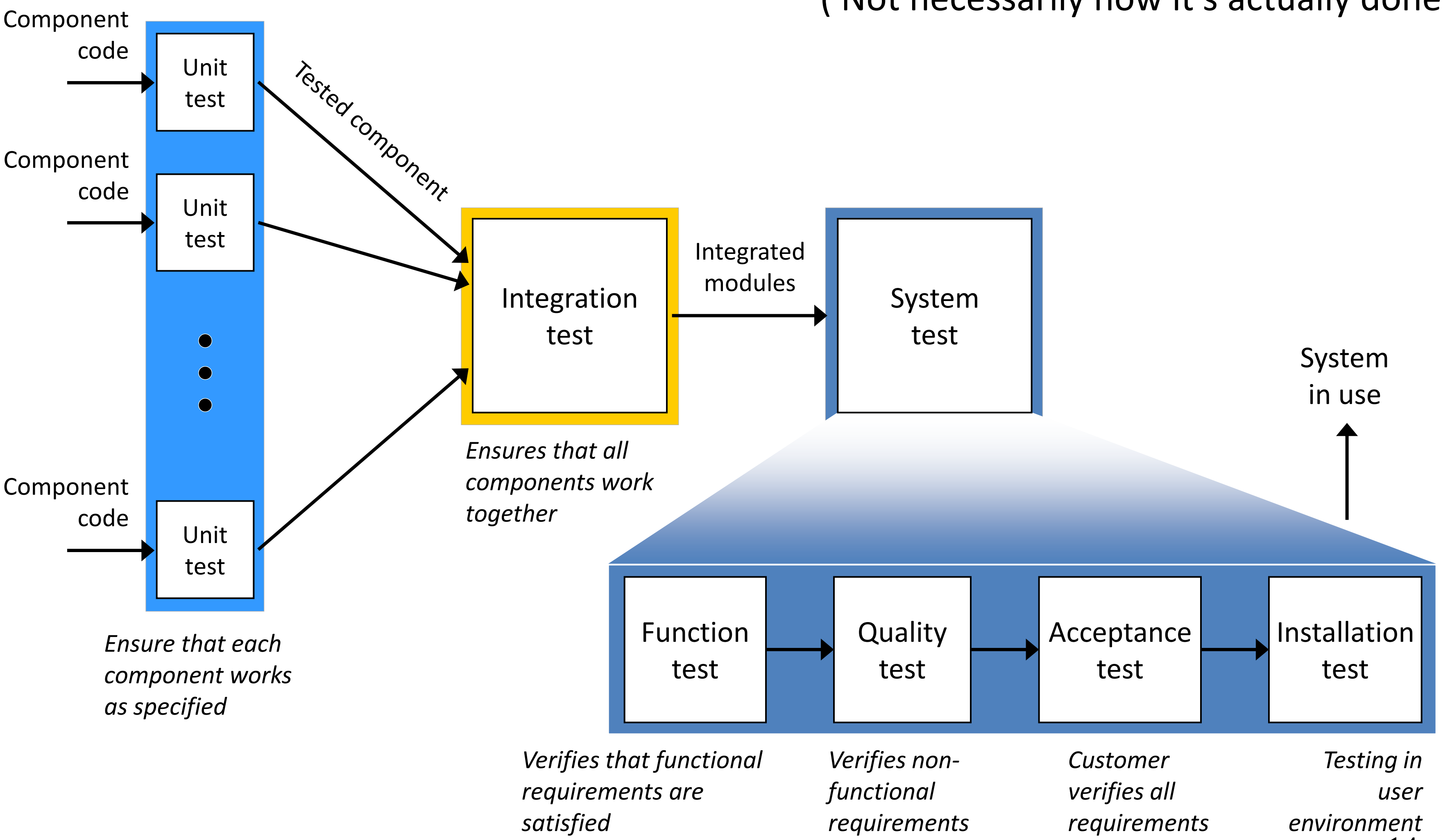
“Testing shows the presence, not the absence of bugs.”
— Edsger W. Dijkstra

Why is testing hard ?

- A key tradeoff of testing:
  - testing as many potential cases as possible while keeping the economic costs limited
- Our goal is to find faults as cheaply and quickly as possible.
  - Ideally, we would design a single “right” test case to expose each fault and run it
- In practice, we have to run many “unsuccessful” test cases that do not expose any faults

Logical Organization of Testing

(Not necessarily how it's actually done!)



Component code

Unit test

Component code

Unit test

Component code

Unit test

Tested component

Integration test

Integrated modules

System test

System in use

Ensure that each component works as specified

Ensures that all components work together

Function test

Quality test

Acceptance test

Installation test

Verifies that functional requirements are satisfied

Verifies non-functional requirements

Customer verifies all requirements

Testing in user environment

Acceptance Tests - Examples

Input data

↓

- Test with the user-id and password of a student to login into the survey system - successful login (pass) ←**Expected result**
- Test with the user-id and incorrect password of a student to login into the survey system – authentication error (pass)
- Test with user-id and password of an admin to login into the survey system – successful login (pass)
- Test adding a question as admin to the question bank with valid text (pass)
- Test as admin adding a question to the question bank with empty text (pass)

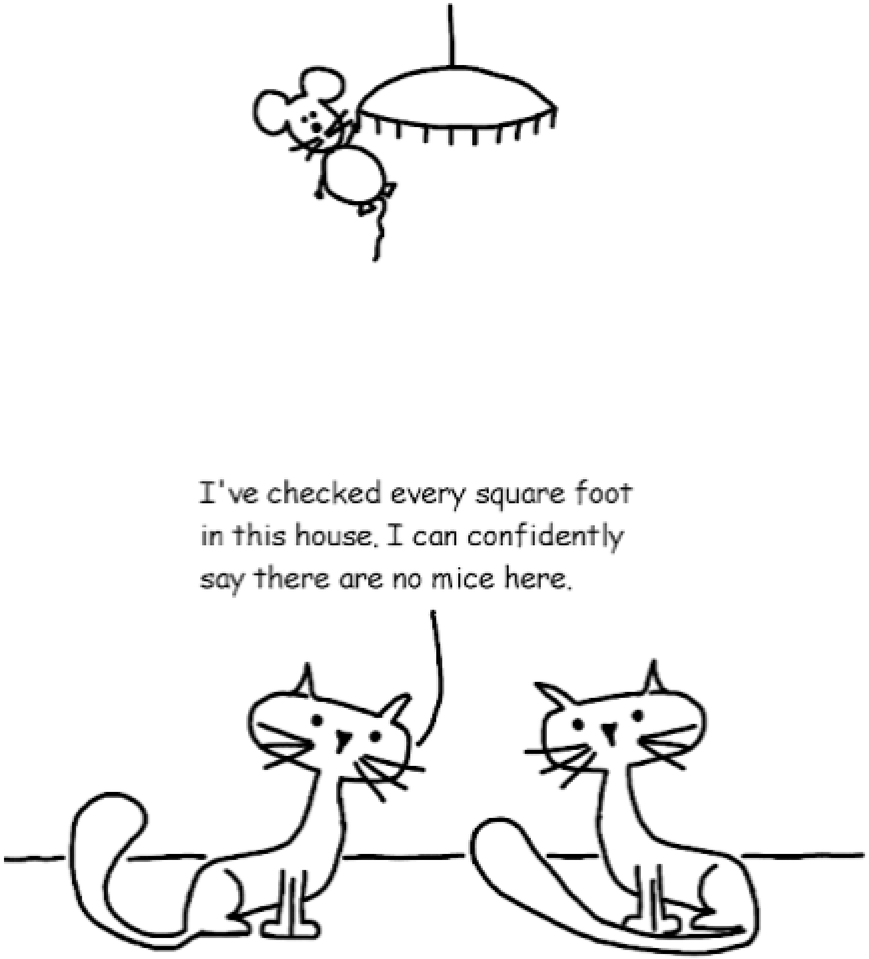
Example: Test Case for Use Case

Test-case Identifier:	TC-1
Use Story Tested:	US-1, main success scenario for student
Pass/fail Criteria:	The test passes if the student id and password are successfully authenticated against the credentials of the student stored in the database and the student is taken to the student dashboard
Input Data:	A valid student id and password
Test Procedure:	Expected Result:
Step 1. Type in a valid student id and password	The input credentials are authenticated against the credentials in the database and the user is taken to the student dashboard

Some other common terminology

- Black box testing
  - A testing approach commonly adopted by customers, such as UAT
  - Test a running program with a set of inputs without looking at the implementation
- White box testing
  - Testing program with test data with knowledge of implementation (system architecture, algorithms used, program code)
- Regression testing
  - Verifying software that was previously developed and tested still performs after the program changed or its interfaces with other software

Test Coverage


I've checked every square foot in this house. I can confidently say there are no mice here.

Absence of proof is not proof of absence.

- William Cowper

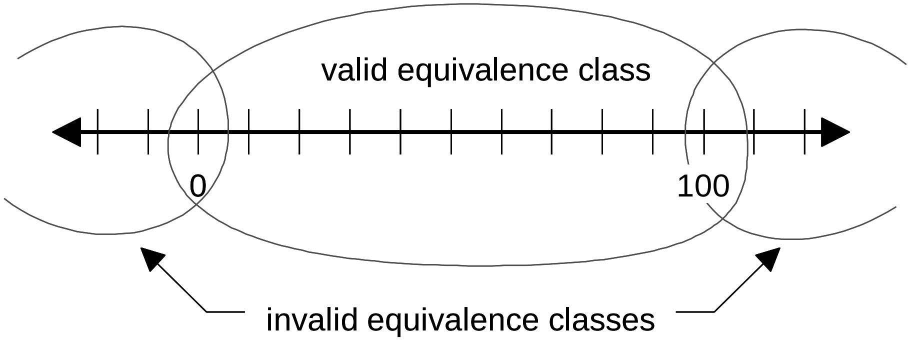
Test Coverage

- **Test coverage** measures the degree to which the specification or code of a software program has been exercised by tests
- **Code coverage** measures the degree to which the source code of a program has been tested
- **Code coverage criteria** include:
  - equivalence testing
  - boundary testing
  - control-flow testing
  - state-based testing

Code Coverage: Equivalence Testing

- **Equivalence testing** is a black-box testing method that divides the space of all possible inputs into equivalence groups such that the program "behaves the same" on each group
- Two steps:
  1. partitioning the values of input parameters into equivalence groups
  2. choosing the test input values

Equivalence classes:



Code Coverage: Boundary Testing

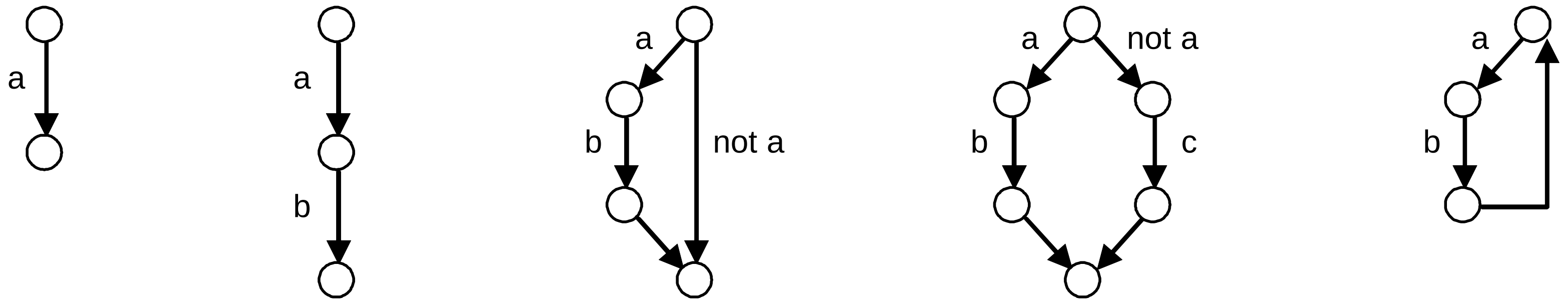
- **Boundary testing** is a special case of equivalence testing that focuses on the boundary values of input parameters
  - Based on the assumption that developers often overlook special cases at the boundary of equivalence classes
- Selects elements from the "edges" of the equivalence class, or "outliers" such as
    - zero, min/max values, empty set, empty string, and null
    - confusion between > and >=
    - etc.

Code Coverage: Control-flow Testing

- Statement coverage : Each statement executed at least once by some test case
- Edge coverage: Every edge (branch) of the control flow is traversed at least once by some test case
- Condition coverage : Every condition takes TRUE and FALSE outcomes at least once in some test case
- Path coverage : Finds the number of distinct paths through the program to be traversed at least once

Constructing the control graph of a program for Edge Coverage:

a;	a; b;	*if* a *then* b;	*if* a *then* b *else* c;	*while* a *do* b;



Unit Testing Frameworks

Python Unit Testing Framework - "PyUnit"

- Python language version of JUnit (used for Java testing)
- Uses module `unittest` to support test automation

Important concepts:

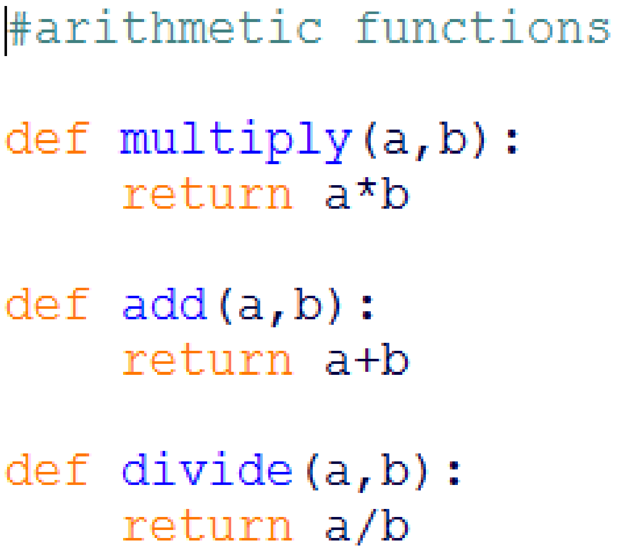
- ***text fixture***: preparation tasks/clean up actions e.g., create temporary databases, directories
- ***test case***: smallest unit of testing, that checks for a specific response to a particular set of inputs (uses a base class `TestCase`, to create new test cases)
- ***test suite*** - a collection of test cases, test suites, or both used to aggregate tests that should be executed together.
- ***test runner*** - orchestrates the execution of tests and provides the outcome to the user

Important Points for writing a single test

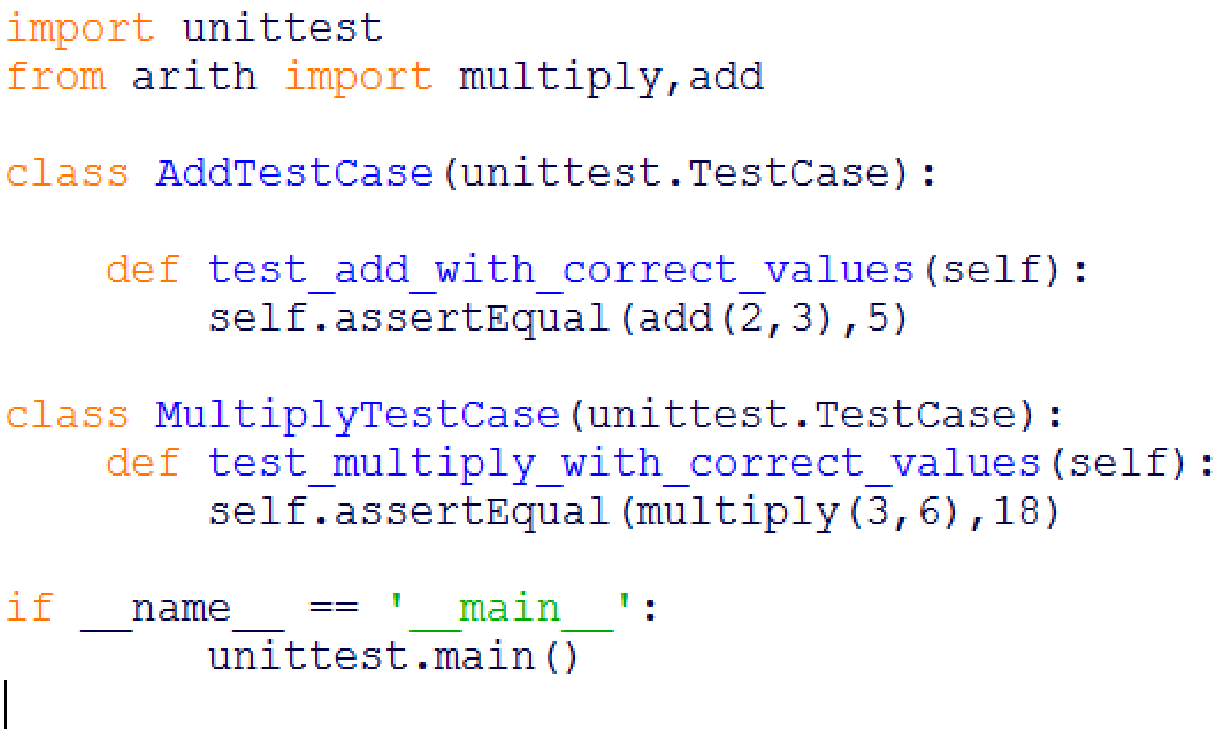
- Every class is a sub-class of `unittest.TestCase`
- Every test function should start with `test` name
- Use `assert` functions to check for an expected result
- Define initialisation tasks by overriding `setup()` method, which is called before a test method is run
- Define clean-up tasks by overriding `teardown()` method, which is called after a test method is run
- Run Test with `python –m unittest –v test_module`

Important Points for writing a single test

- Every class is a sub-class of **unittest.TestCase**
- Every test function should start with **test** name



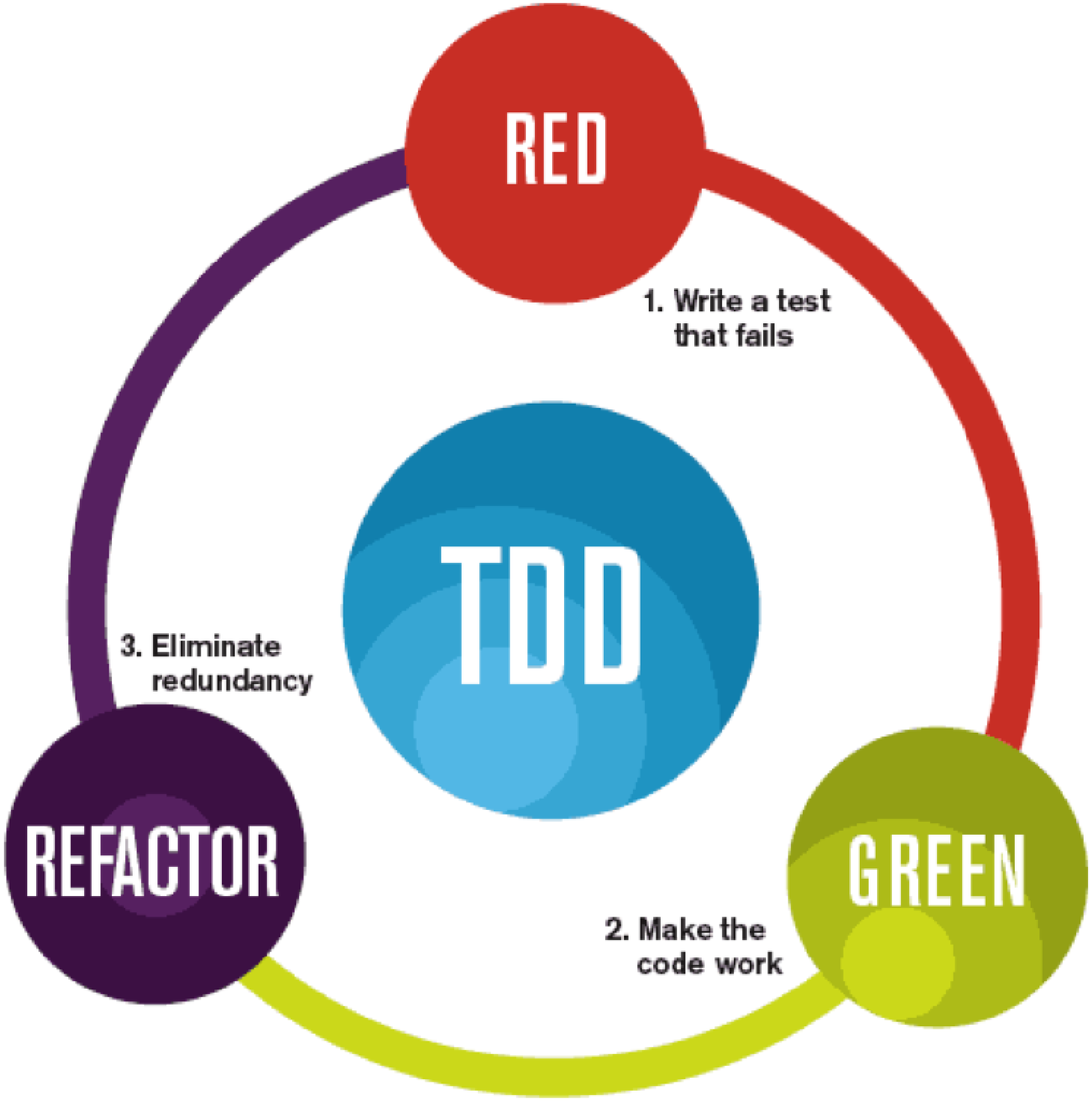
```
#arithmetic functions

def multiply(a,b):
    return a*b

def add(a,b):
    return a+b

def divide(a,b):
    return a/b
```



```
import unittest
from arith import multiply,add

class AddTestCase(unittest.TestCase):

    def test_add_with_correct_values(self):
        self.assertEqual(add(2,3),5)

class MultiplyTestCase(unittest.TestCase):
    def test_multiply_with_correct_values(self):
        self.assertEqual(multiply(3,6),18)

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

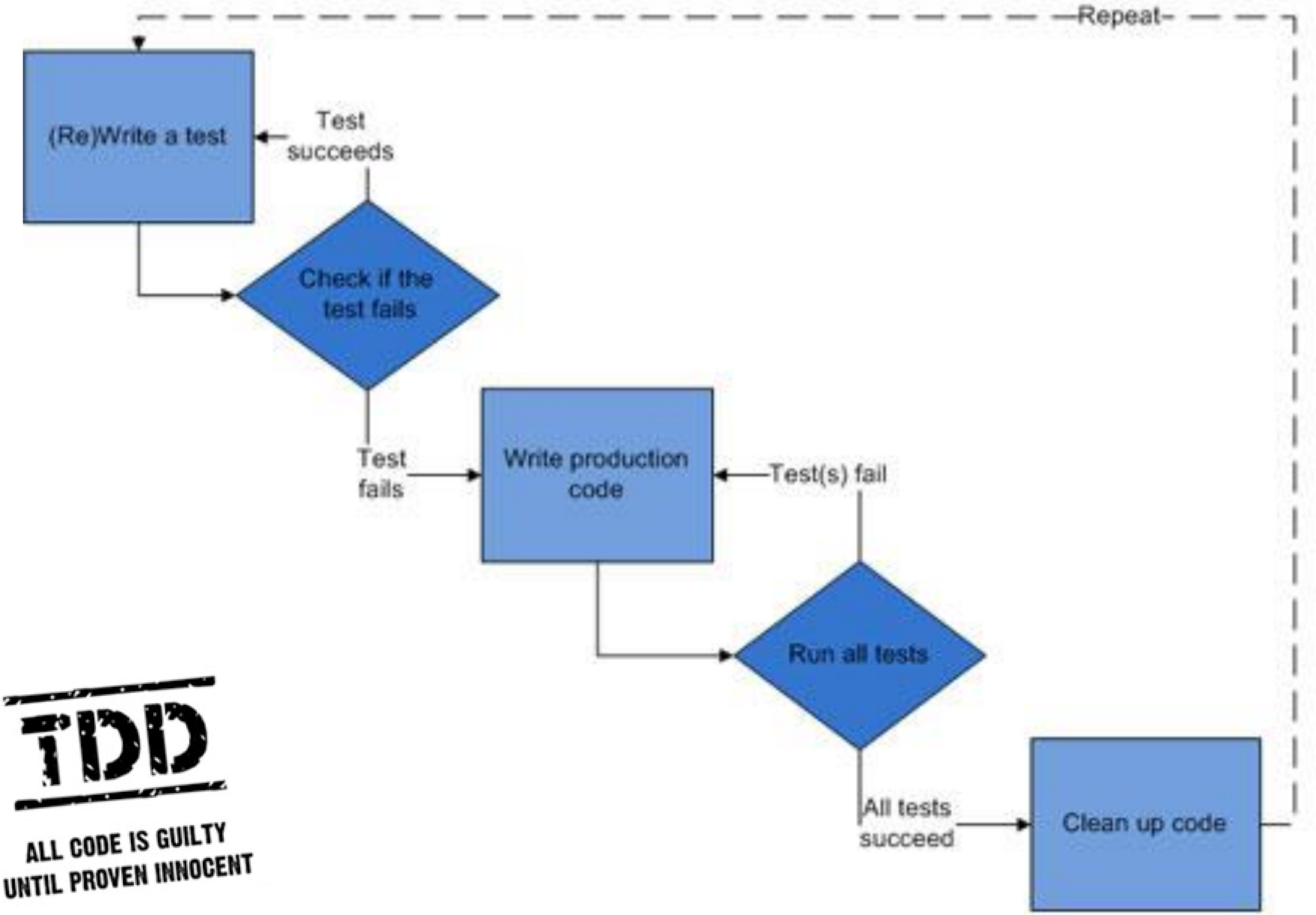
Test Driven Development

- Every step in the development process must start with a plan of how to verify that the result meets a goal
- Developer should not create a software artifact (a UML diagram, or source code) unless they know how it will be tested
- An important principle in XP, Scrum



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

© https://businessanalystlearnings.com/technology-matters/2014/8/13/test-driven-vs-behaviour-driven-development



Repeat

(Re)Write a test

Test succeeds

Check if the test fails

Test fails

Write production code

Test(s) fail

Run all tests

All tests succeed

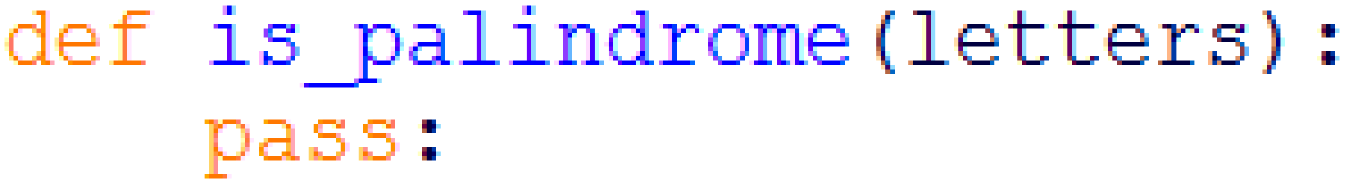
Clean up code

TDD

ALL CODE IS GUILTY UNTIL PROVEN INNOCENT

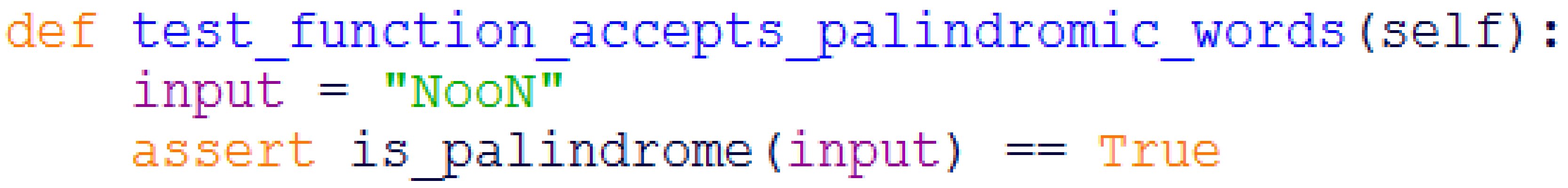
Lecture demo: palin.py - A TDD Example

Write a function to check whether a given input string is a palindrome

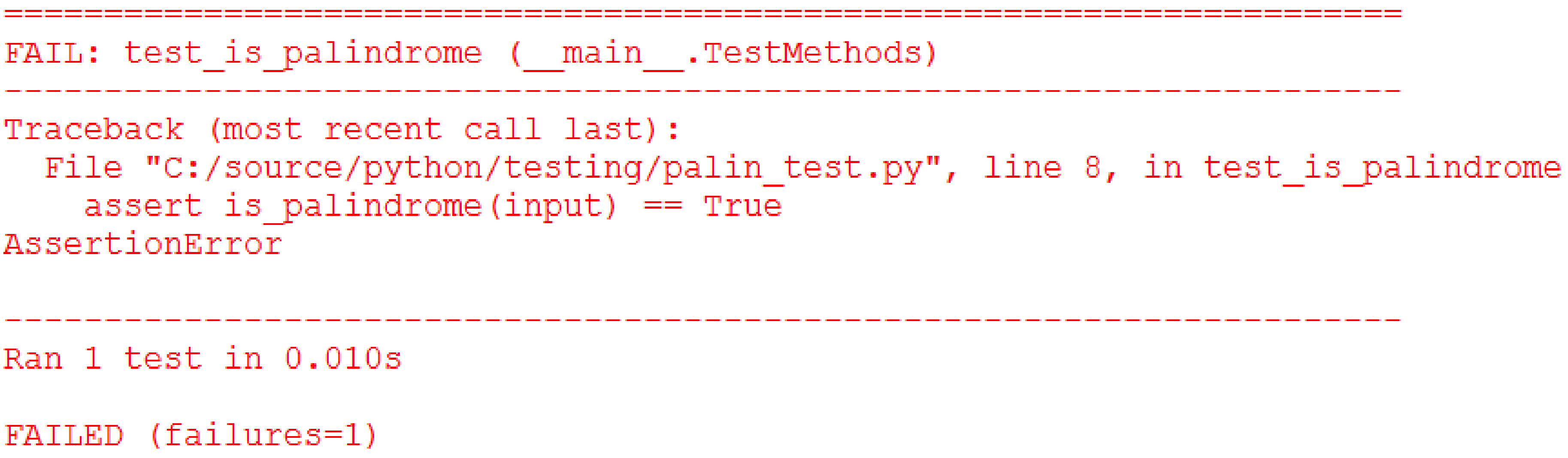


```
def is_palindrome(letters):
    pass:
```

Step 1: Write a test case that fails

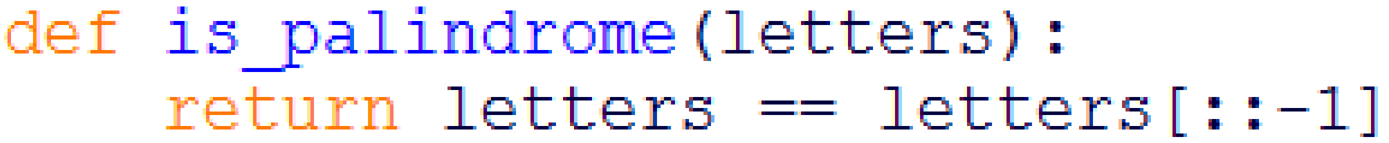


```python
def test_function_accepts_palindromic_words(self):
    input = "NooN"
    assert is_palindrome(input) == True
```



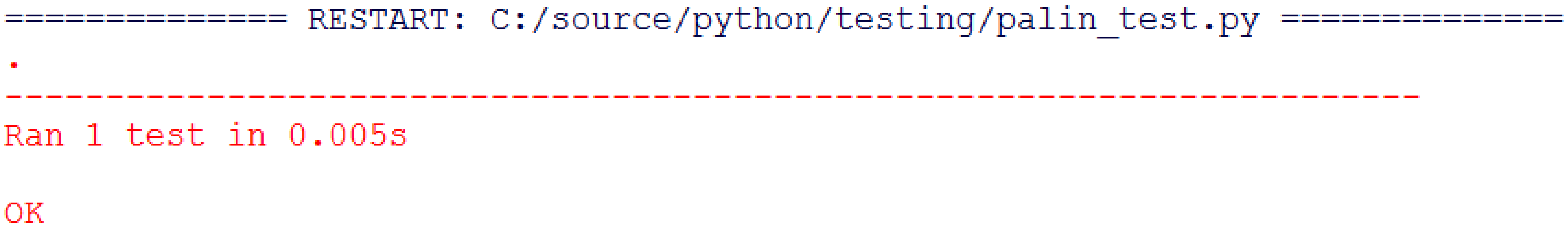
```
======================================================================
FAIL: test_is_palindrome (__main__.TestMethods)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/source/python/testing/palin_test.py", line 8, in test_is_palindrome
    assert is_palindrome(input) == True
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.010s

FAILED (failures=1)
```

Step 2: Write the code to implement the function



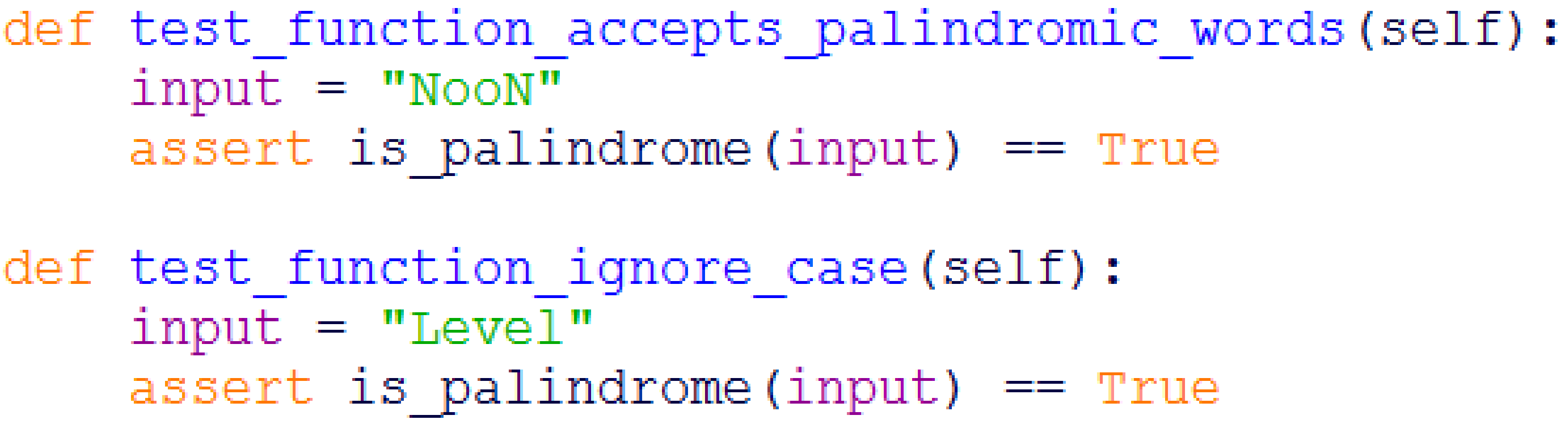
```python
def is_palindrome(letters):
    return letters == letters[::-1]
```

Step 3: Test the code



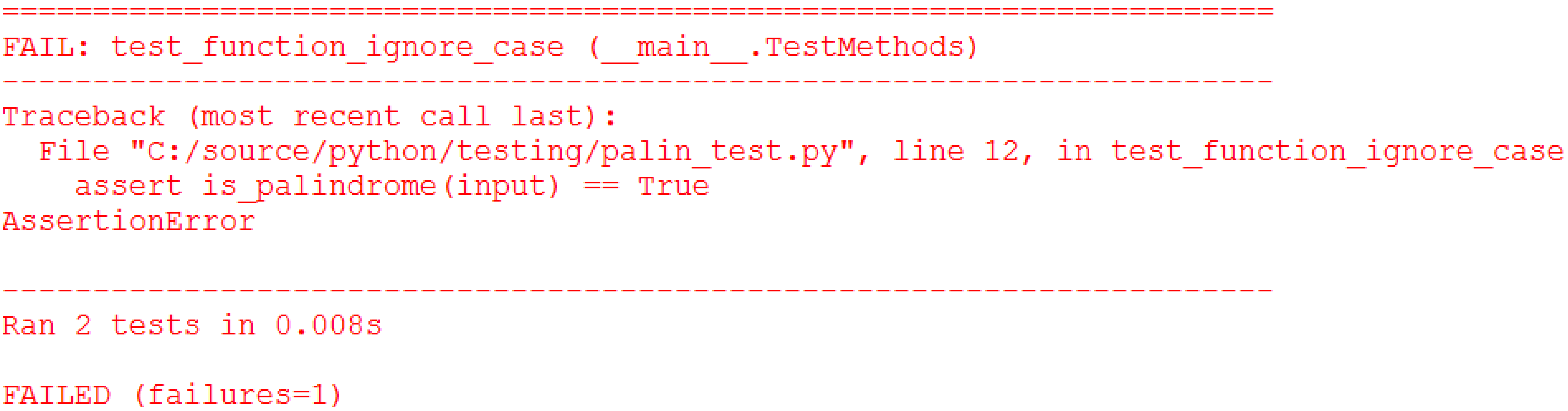
```
============== RESTART: C:/source/python/testing/palin_test.py ==============
.
----------------------------------------------------------------------
Ran 1 test in 0.005s

OK
```

Repeat: Add a 2nd test case



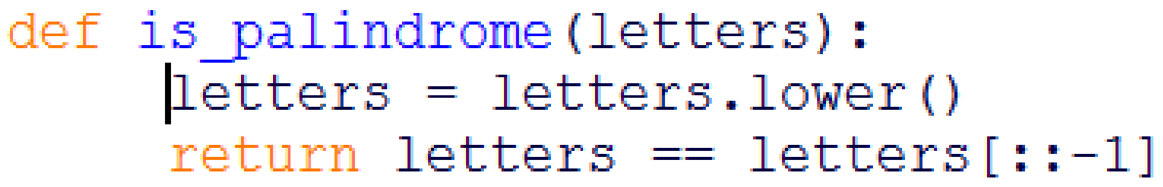
```python
def test_function_accepts_palindromic_words(self):
    input = "NooN"
    assert is_palindrome(input) == True

def test_function_ignore_case(self):
    input = "Level"
    assert is_palindrome(input) == True
```

(Second test fails as expected, as the code to test this scenario isn’t implemented yet)



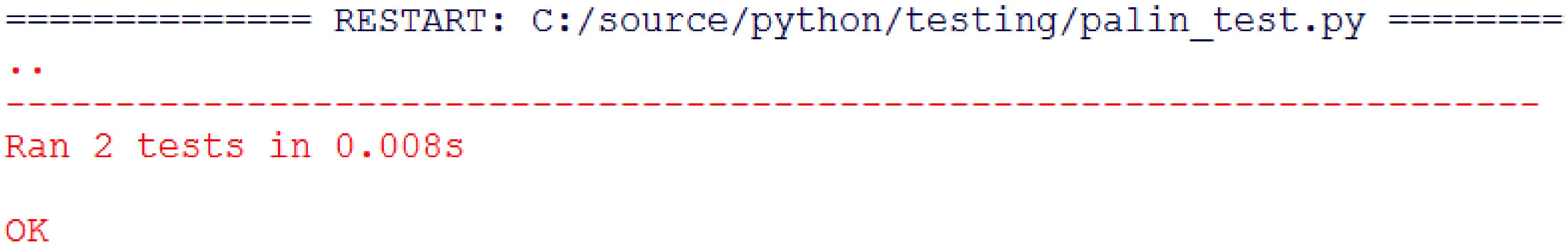
```
======================================================================
FAIL: test_function_ignore_case (__main__.TestMethods)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/source/python/testing/palin_test.py", line 12, in test_function_ignore_case
    assert is_palindrome(input) == True
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.008s

FAILED (failures=1)
```

Repeat: Add logic to ensure that the second test succeeds



```
def is_palindrome(letters):
    letters = letters.lower()
    return letters == letters[::-1]
```

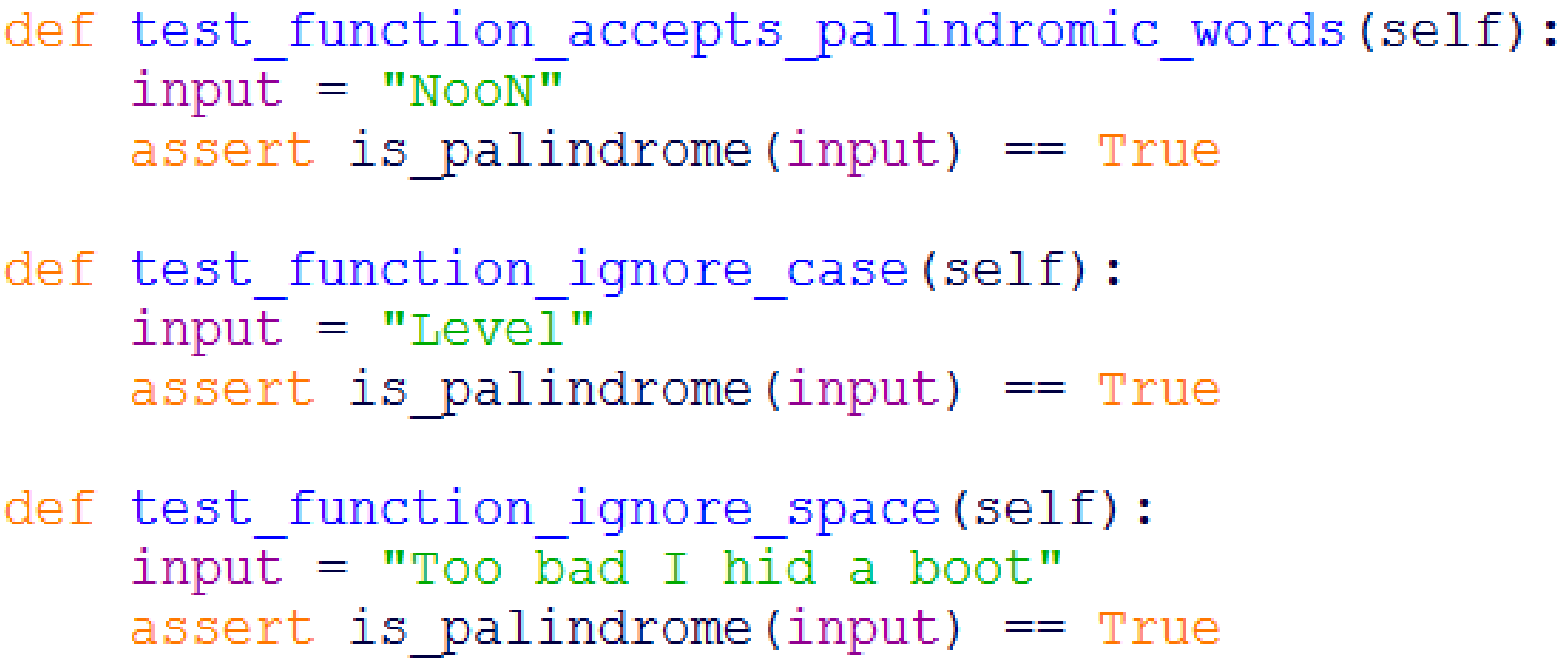
Test the code again to ensure all tests now succeed



```
============== RESTART: C:/source/python/testing/palin_test.py =========
..
----------------------------------------------------------------------
Ran 2 tests in 0.008s

OK
```

(All tests now succeed as expected)

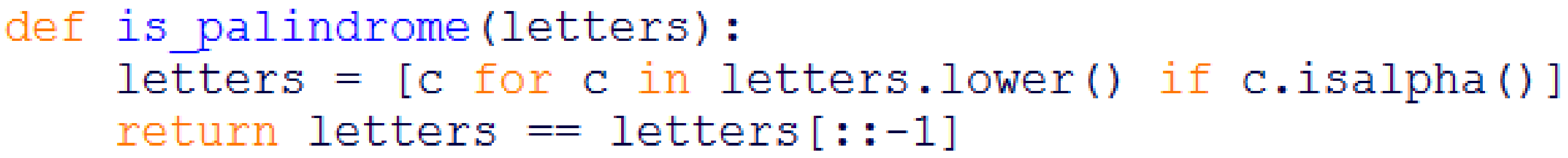
Repeat: Add a 3rd test case and run test harness again



```
def test_function_accepts_palindromic_words(self):
    input = "NooN"
    assert is_palindrome(input) == True

def test_function_ignore_case(self):
    input = "Level"
    assert is_palindrome(input) == True

def test_function_ignore_space(self):
    input = "Too bad I hid a boot"
    assert is_palindrome(input) == True
```

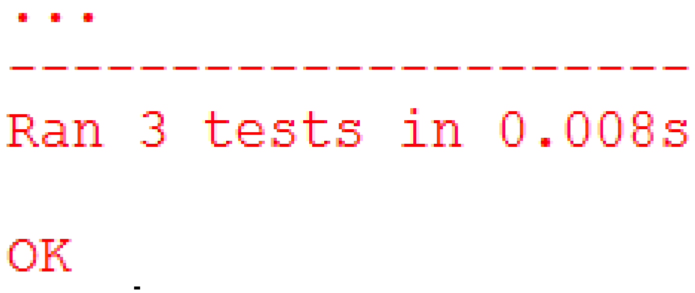
(Third test will fail as expected, as the code to test this scenario isn't implemented yet)

Repeat: Refactor code to ensure that the 3rd test succeeds



```python
def is_palindrome(letters):
    letters = [c for c in letters.lower() if c.isalpha()]
    return letters == letters[::-1]
```

Test the code again to ensure all tests now succeed



```
...
----------------------------------------------------------------------
Ran 3 tests in 0.008s

OK
```

(All tests now succeed as expected)

So, it continues…

- Many cases to consider
- Unit tests help to validate the complex logic and check for regression errors
- Brings the emphasis that every unit of code is tested
- So, the general rule is:
  - Start by writing a test (that fails)
  - Write just enough code to pass the test - You aren’t gonna need it! (YAGNI)
  - Test again, and make corrections until test passes
  - Once passed, refactor code to remove redundancies
  - Move on to next piece of code