Programming technologies

Testing

Definition of Testing

Unit testing is a method for testing software that looks at the smallest testable pieces of code, called units, which are tested for correct operation. By doing unit testing, we can verify that each part of the code, including helper functions that may not be exposed to the user, works correctly and as intended.

The idea is that we are independently checking each small piece of our program to ensure that it works. This contrasts with regression and integration testing, which tests that the different parts of the program work well together and as intended.

Automated testing is the execution of your test plan (the parts of your application you want to test, the order in which you want to test them, and the expected responses) by a script instead of a human. Python already comes with a set of tools and libraries to help you create automated tests for your application.

Types of Testing

Unit Tests vs. Integration Tests

Think of how you might test the lights on a car. You would turn on the lights (known as the test step) and go outside the car or ask a friend to check that the lights are on (known as the test assertion). Testing multiple components is known as **integration** testing.

If you have a fancy modern car, it will tell you when your light bulbs have gone. It does this using a form of **unit** test.

A unit test is a smaller test, one that checks that a single component operates in the right way. A unit test helps you to isolate what is broken in your application and fix it faster.

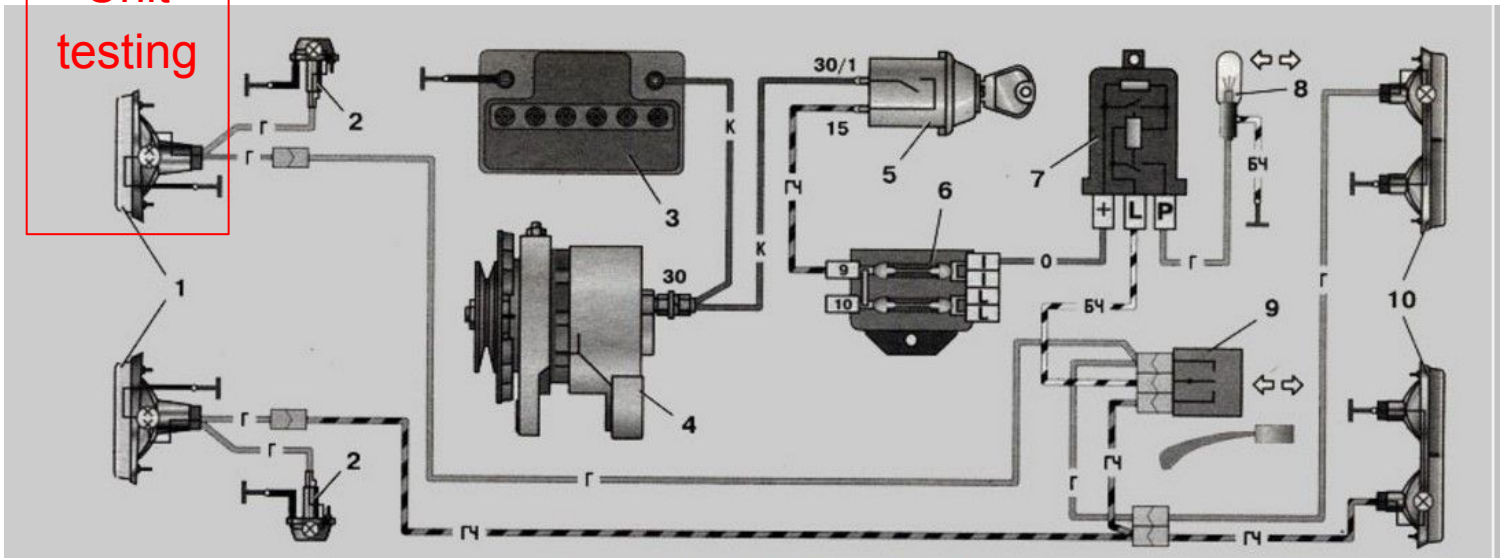
You have just seen two types of tests:

- An integration test checks that components in your application operate with each other.
- A unit test checks a small component in your application.

Types of Testing

Unit testing

Integration testing



Scheme of direction indicators

Example 1

```
def get_area_rectangle(height, width):
    return height * width

if __name__ == "__main__":
    print(get_area_rectangle(3,2))
    print(get_area_rectangle(4,2))
```

```
def get_area_rectangle(height, width):
    return height * width

if __name__ == "__main__":
    assert get_area_rectangle(3, 2) == 6
    assert get_area_rectangle(4, 2) == 8, "Incorrect"
    #assert get_area_rectangle(4, 2) == 7, "Incorrect"
```

Example 2

```
# Password requirements:
# 8+ symbols
# 1 digit
# 1 lower case char
# 1 upper case char
# If length of password less than 8 chars tnam return "To short"
# If the password consists of only: digits, or lowercase chars, or uppercase chars - return
"Weak"
# If password consist of symbols form 2/3 groups - return "Good"
# If password consist of symbols from each group - return "Excellent"

def get_password_level(password):
    pass

if __name__ == "__main__":
    assert get_password_level("123") == "To short", "To short"
```

Example

```
def get_password_level(password):
    if len(password) < 8:
        return "To short"
    elif all(ord(e) in range(48, 58) for e in password) \
    or all(ord(e) in range(65, 91) for e in password) \
    or all(ord(e) in range(97, 123) for e in password):
        return "Weak"
    elif any(ord(e) in range(48, 58) for e in password) \
    and any(ord(e) in range(65, 91) for e in password) \
    and any(ord(e) in range(97, 123) for e in password):
        return "Excellent"
    elif (any(ord(e) in range(48, 58) for e in password) and any(ord(e) in range(65, 91) for e in password)) \
    or (any(ord(e) in range(65, 91) for e in password) and any(ord(e) in range(97, 123) for e in password)) \
    or (any(ord(e) in range(97, 123) for e in password) and any(ord(e) in range(48, 58) for e in password)):
        return "Good"

if __name__ == "__main__":
    assert get_password_level("12qqWW") == "To short", "To short"
    assert get_password_level("123456") == "To short", "To short"
    assert get_password_level("qwerty") == "To short", "To short"
    assert get_password_level("ASDFGH") == "To short", "To short"
    assert get_password_level("12345678") == "Weak", "Weak"
    assert get_password_level("asdfghjk") == "Weak", "Weak"
    assert get_password_level("QWERTYUI") == "Weak", "Weak"
    assert get_password_level("QWERTqwer") == "Good", "Good"
    assert get_password_level("QWERT1234") == "Good", "Good"
    assert get_password_level("qwer1234") == "Good", "Good"
    assert get_password_level("qw1234RT") == "Excellent", "Excellent"
```

Example

```
def get_password_level(password):
    if len(password) < 8:
        return "To short"
    if password.isdigit() or (password.isalpha() and password.islower()) or (password.isalpha() and password.isupper()):
        return "Weak"
    if any(e.isdigit() for e in password) and any(e.isalpha() and e.isupper() for e in password) and any(e.isalpha() and e.islower()
for e in password):
        return "Excellent"
    if (any(e.isdigit() for e in password) and any(e.isalpha() and e.isupper() for e in password)) \
            or (any(e.isdigit() for e in password) and any(e.isalpha() and e.islower() for e in password)) \
            or (any(e.isalpha() and e.islower() for e in password) and any(e.isalpha() and e.isupper() for e in password)):
        return "Good"


if __name__ == "__main__":
    assert get_password_level("123") == "To short", "To short passwort"
    assert get_password_level("wer") == "To short", "To short passwort"
    assert get_password_level("WERTY") == "To short", "To short passwort"
    assert get_password_level("werQWE") == "To short", "To short passwort"
    assert get_password_level("2wE") == "To short", "To short passwort"
    assert get_password_level("12345678") == "Weak", "To weak passwort"
    assert get_password_level("abcdefgh") == "Weak", "To weak passwort"
    assert get_password_level("ABCDEFGH") == "Weak", "To weak passwort"
    assert get_password_level("A1234567") == "Good", "Good passwort"
    assert get_password_level("a1234567") == "Good", "Good passwort"
    assert get_password_level("ABCDefgh") == "Good", "Good passwort"
    assert get_password_level("A12345678") == "Good", "Good passwort"
    assert get_password_level("ab1234567") == "Good", "Good passwort"
    assert get_password_level("ABCDefghI") == "Good", "Good passwort"
    assert get_password_level("123ABCdef") == "Excellent", "Excellent passwort"
```

Module unittest

unittest has been built into the Python standard library.

unittest requires that:

- You put your tests into classes as methods
- You use a series of special assertion methods in the unittest.TestCase class instead of the built-in assert statement

Example

```
import unittest
from get_password_level import get_password_level

class TestGetPasswordLevel(unittest.TestCase):

    def test_to_short(self):
        self.assertEqual(get_password_level("12qqWW"), "To short")
        self.assertEqual(get_password_level("123456"), "To short")
        self.assertEqual(get_password_level("qwerty"), "To short")
        self.assertEqual(get_password_level("ASDFGH"), "To short")

    def test_weak(self):
        self.assertEqual(get_password_level("12345678"), "Weak")
        self.assertEqual(get_password_level("asdfghjk"), "Weak")
        self.assertEqual(get_password_level("QWERTYUI"), "Weak")

    def test_good(self):
        self.assertEqual(get_password_level("QWERTqwer"), "Good")
        self.assertEqual(get_password_level("QWERT1234"), "Good")
        self.assertEqual(get_password_level("qwer1234"), "Good")

    def test_excellent(self):
        self.assertEqual(get_password_level("qw1234RT"), "Excellent")

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

Asserts types

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Asserts types

Method	Checks that	New in
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc*	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn*	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*	3.2
`assertLogs(logger, level)`	The `with` block logs on *logger* with minimum *level*	3.4
`assertNoLogs(logger, level)`	The `with` block does not log on *logger* with minimum *level*	3.10

Example

```
from unittest import TestCase
from unittest import main

def division(a, b):
    if b == 0:
        raise ZeroDivisionError()
    if b == 2:
        raise ValueError()
    return a / b

class TestDivision(TestCase):

    def test_division_by_zero(self):
        with self.assertRaises(ZeroDivisionError) as e:
            division(1, 0)

    def test_division_by_2(self):
        with self.assertRaises(ValueError) as e:
            division(1, 2)

    def test_valid_value(self):
        self.assertEqual(division(3, 1), 3)
        self.assertEqual(division(9, 3), 3)
        self.assertNotEqual(division(9, 1), 8)

if __name__ == "__main__":
    main()
```

Useful links

https://realpython.com/python-testing/

https://machinelearningmastery.com/a-gentle-introduction-to-unit-testing-in-python/

https://docs.python.org/3/library/unittest.html#test-cases