Contoso

UNIT TEST CON PYTHON

PRUEBAS DE UNIDAD

CONTENIDO 01

Las pruebas unitarias son un método de prueba de software mediante el cual las unidades individuales de código fuente, como funciones, métodos y clases se prueban para determinar si son aptos para el Uso.

Intuitivamente, se puede ver una unidad como la parte más pequeña comprobable de una aplicación. Las pruebas de Unidad son fragmentos de código cortos creados por los programadores durante el proceso de desarrollo.

Forman la base para las pruebas de componentes.

Temas:

Qué es y cómo funciona Test Unit de Python?

Cómo se utiliza?

Cuáles son las opciones que permite utilizar?

Contoso

DIFERENCIA ENTRE PRUEBAS MANUALES / AUTOMÁTICAS

PRUEBAS MANUALES

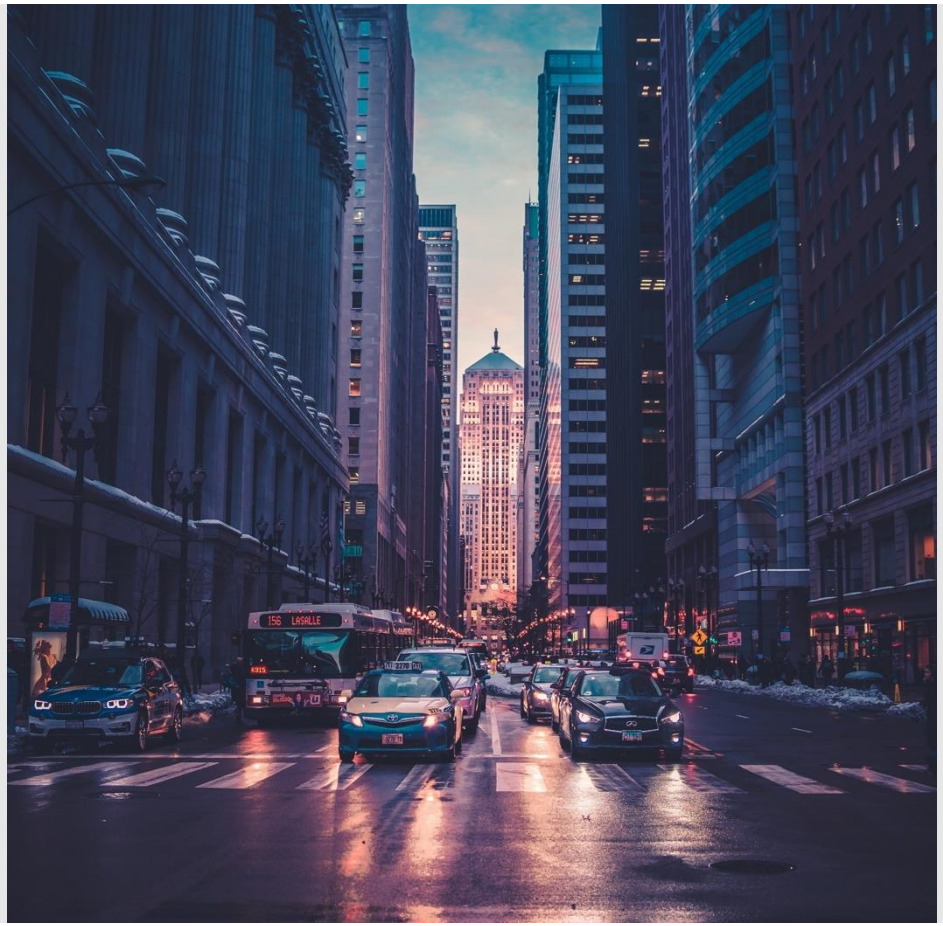
Es la ejecución de casos de prueba manualmente sin soporte de herramientas.

• Dado que los casos de prueba son ejecutados por recursos humanos se consume tiempo y puede ser tedioso.

• Como los casos de prueba deben ser ejecutados manualmente se requieren los probadores.

• Es menos fiable, ya que las pruebas pueden no realizarse con precisión debido a errores humanos.

• No se puede hacer ninguna programación para escribir pruebas sofisticadas para obtener información oculta.



PRUEBAS AUTOMÁTICAS

Tomar el soporte de la herramienta y ejecutar la casos de prueba es la automatización de pruebas.

• La automatización ejecuta casos de prueba significativamente más rápido que el ser humano.

• La inversión en recursos es menor con casos de prueba que se ejecutan mediante el uso de la automatización.

• La automatización de pruebas realizan con precisión las operación, por esto son más confiables.

• Los evaluadores pueden programar pruebas sofisticadas para sacar a la salida información oculta.

QUÉ ES “PYUNIT”?

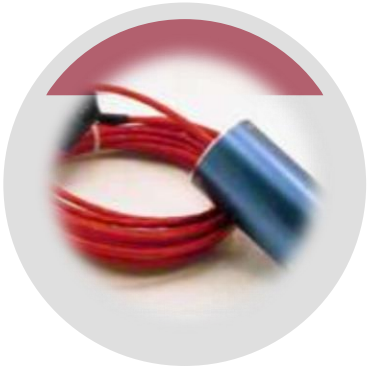
"PyUnit", es un lenguaje similar a Junit, desarrollado por Kent Beck y Erich Gamma. PyUnit forma parte de la **biblioteca estándar de Python a partir de la versión 2.1**.

El Framework “PyUnit” admite la **automatización de pruebas**, el uso **compartido de la configuración** y el **apagado código** para pruebas, agregación de **pruebas** en **colecciones** e **independencia** de las pruebas y el framework de los informes.

El módulo **unittest** proporciona clases que facilitan el soporte de estas cualidades para un conjunto de pruebas.

Contoso

CÓMO FUNCIONA “PYUNIT”?



TEST FIXTURE

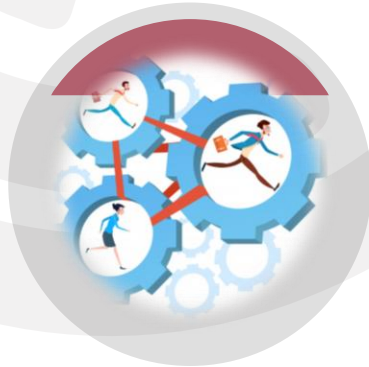
Esto representa la preparación necesaria para realizar una o más pruebas y cualquier acción de limpieza asociada. Esto puede implicar, por ejemplo, crear bases de datos temporales o proxy, directorios o iniciar un proceso de servidor.

TEST CASE

Esta es la unidad de prueba más pequeña. Esto comprueba si hay respuesta a un conjunto particular de entradas. Unittest proporciona una clase base, TestCase, que se puede utilizar para crear nuevos casos de prueba.

TEST SUITE

Esta es una colección de casos de prueba, conjuntos de pruebas o ambos. Esto se utiliza para agregar pruebas que se deben ejecutar juntas. Los conjuntos de pruebas son implementados por la clase TestSuite.



TEST RUNNER

Este es un componente que organiza la ejecución de pruebas y proporciona el resultado al usuario. El corredor puede utilizar una interfaz gráfica, una interfaz textual o devolver un valor especial para indicar los resultados de la ejecución de las pruebas.

Contoso

CÓMO SE UTILIZA “PYUNIT”?

Los siguientes pasos para la redacción de una prueba unitaria simple:

Paso 1: Importe el módulo unittest en su programa.

Paso 2: Defina una función que se va a probar. En el siguiente ejemplo, la función add() debe ser sometida a prueba.

Paso 3: Cree un caso de prueba subclases unittest. TestCase.

Paso 4: Defina una prueba como un método dentro de la clase. El nombre del método debe comenzar con 'test'.

Paso 5: Cada prueba llama a la función assert de la clase TestCase. Hay muchos tipos de afirmaciones. El ejemplo siguiente llama a la función assertEquals().

Paso 6: la función assertEquals() compara el resultado de la función add() con el argumento arg2 y produce assertionError si se produce un error en la comparación.

Contoso

PASO 7:

Se prueba la función add(x, y) validando un caso con la función assertEquals, esta función ejecuta la operación add con los valores específicos 4 y 5. Se espera un resultado de 9. Ahora escriba el llamado al método main de unittest.

```
import unittest
def add(x,y):
    return x+y

class SimpleTest(unittest.TestCase):
    def testadd1(self):
        self.assertEquals(add(4,5),9)

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

Contoso

CÓMO FUNCIONA “PYUNIT”?

PASO 8:

Ejecutar el código y en este caso se obtendrá el siguiente resultado:

```
C:\Python27>python SimpleTest.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

OK	The test passes. ‘A’ is displayed on console
FAIL	The test does not pass, and raises an AssertionError exception. ‘F’ is displayed on console.
ERROR	The test raises an exception other than AssertionError. ‘E’ is displayed on console.

Los posibles resultados serán:

PASO 9:

CUÁLES SON LAS OPCIONES QUE PERMITE UTILIZAR?

El framework de “PyUnit” utiliza la **función assert()** integrada de Python para probar una condición determinada. Si se produce un error en la aserción, se producirá un **AssertionError**. En ese caso, el framework de pruebas identificará la prueba como **Error**. Otras **excepciones** se tratan como Error.

Los tres conjuntos de **funciones de aserción** siguientes se definen en el módulo unittest:

- Aserciones booleanas básicas
- Aserciones comparativas
- Aserciones para Colecciones

Las funciones de aserción básicas evalúan si el resultado de una **operación es True o False**. Todos los métodos assert aceptan un argumento msg que, si se especifica, se utiliza como mensaje de error en caso de error.

assertEqual(*arg1*, *arg2*, *msg=None*)	Test that *arg1* and *arg2* are equal. If the values do not compare equal, the test will fail.
assertNotEqual(*arg1*, *arg2*, *msg=None*)	Test that *arg1* and *arg2* are not equal. If the values do compare equal, the test will fail.
assertTrue(*expr*, *msg=None*)	Test that *expr* is true. If false, test fails
assertFalse(*expr*, *msg=None*)	Test that *expr* is false. If true, test fails
assertIs(*arg1*, *arg2*, *msg=None*)	Test that *arg1* and *arg2* evaluate to the same object.
assertIsNot(*arg1*, *arg2*, *msg=None*)	Test that *arg1* and *arg2* don’t evaluate to the same object.
assertIsNone(*expr*, *msg=None*)	Test that *expr* is None. If not None, test fails
assertIsNotNone(*expr*, *msg=None*)	Test that *expr* is not None. If None, test fails
assertIn(*arg1*, *arg2*, *msg=None*)	Test that *arg1* is in *arg2*.
assertNotIn(*arg1*, *arg2*, *msg=None*)	Test that *arg1* is not in *arg2*.
assertIsInstance(*obj*, *cls*, *msg=None*)	Test that *obj* is an instance of *cls*
assertNotIsInstance(*obj*, *cls*, *msg=None*)	Test that *obj* is not an instance of *cls*

Contoso

FIN DE LA PRESENTACIÓN

PREGUNTAS?