prog_bio_6

July 23, 2018

1 Programación para la bioinformática

1.1 Unidad 6: *Testing* y calidad del software

1.1.1 Instrucciones de uso

A continuación, se presentarán explicaciones y ejemplos de uso de pruebas de código o *software tests*. Recordad que podéis ir ejecutando los ejemplos para obtener sus resultados.

1.2 Introducción

En primer lugar, planteemos las siguientes preguntas:

- **£Cómo podemos estar seguros de que el código que hemos escrito es correcto?**
- **£Cómo estamos seguros de que nuestro código es capaz de responder en todos los posibles escenarios?**

Quizá, la respuesta más directa a estas preguntas sea **escribiendo pruebas de código**. Puede parecer extraño que, para probar que efectivamente el código que hemos escrito nos da el resultado correcto y esperado, tengamos que escribir más código que se encargue de verificar el primero. Pero no hacerlo equivale, en el argot, a **caminar sobre la cuerda floja sin red de fondo**.

Las pruebas de código, de aquí en adelante *test*, serán nuestra red, la red que capture los problemas de ejecución de nuestro código y que nos ayude a detectar en una etapa temprana de desarrollo posibles fallos y *bugs*.

Existen diferentes tipos de test: unitarios, de regresión, de punta a punta, de integración... Nosotros nos centraremos en esta unidad en los tests de más bajo nivel, los **unitarios**, que se encargan de comprobar funciones de forma atómica.

Veamos un primer ejemplo:

```
In [1]: from math import sqrt


        # Escribimos una función que realiza una suma de forma un tanto especial:

        def suma_rara(a, b):
            return pow(sqrt(a), 2) + pow(sqrt(b), 2)
```

```
# Ejecutamos nuestra función utilizando 2 y 3 como argumentos
print(suma_rara(2, 3))
```

```
5.0
```

Como se puede entender al analizar el código, la función calcula la suma de los cuadrados de las raíces de los argumentos, *a* y *b*.

Se trata de una forma un tanto extraña de sumar dos números, pero aparentemente correcta. Ahora bien, **£qué pasaría si alguno de los argumentos fuera un número negativo?** Comprobémoslo:

```
In [2]: print(suma_rara(-2, 3))
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-2-32484313b02a> in <module>()
----> 1 print(suma_rara(-2, 3))


<ipython-input-1-013288ca9d46> in suma_rara(a, b)
  5
  6 def suma_rara(a, b):
----> 7     return pow(sqrt(a), 2) + pow(sqrt(b), 2)
  8
  9


ValueError: math domain error
```

Obtenemos un error de ejecución, `ValueError`, que nos indica que hay un problema de dominio matemático, y es que la raíz cuadrada de un número negativo no pertenece al dominio de los números reales. En conclusión: nuestra función tiene un error y no es capaz de sumar uno o dos números negativos.

1.3 *Testing*

Como hemos visto en el apartado anterior, necesitamos una forma de comprobar y *testear* nuestro código. Existen diferentes alternativas en Python, incluídas en la librería estándar o librerías externas. Vamos a ver qué opciones nos proporciona Python por defecto:

1.3.1 1. *Doctest*

Se pueden indicar los propios *test* utilizando *doctest*. *Doctest* pertenece a la librería estándar de Python y es capaz de reconocer porciones de código Python incluídas en la cabecera/documentación de la función y ejecutarlos. Veamos un ejemplo sencillo:

```
In [3]: def square(x):
            """Return the square of x.

            >>> square(2)
            4
            >>> square(-2)
            4
            """
            return x * x


        if __name__ == '__main__':
            import doctest
            doctest.testmod()
```

Como vemos, al ejecutar el anterior código, no aparece ningún mensaje de error. Modifiquemos ahora el código de retorno de la función para comprobar que efectivamente los *test* se están ejecutando:

```
In [4]: def square(x):
            """Return the square of x.

            >>> square(2)
            4
            >>> square(-2)
            4
            """
            return x * 2 * x


        if __name__ == '__main__':
            import doctest
            doctest.testmod()

**********************************************************************
File "__main__", line 4, in __main__.square
Failed example:
    square(2)
Expected:
    4
Got:
    8
**********************************************************************
File "__main__", line 6, in __main__.square
Failed example:
    square(-2)
Expected:
    4
```

```
Got:
    8
**********************************************************************
1 items had failures:
   2 of   2 in __main__.square
***Test Failed*** 2 failures.
```

Como podemos observar, ahora se nos avisa en tiempo de ejecución de que ha habido dos problemas al ejecutar el código y de cuáles son los resultados esperados (*Expected*) y obtenidos (*Got*).

Doctest es muy útil para indicar a otros programadores cómo debe utilizar nuestras funciones a la vez que se prueba de forma muy simple el propio código, aunque no están indicadas para grandes desarrollos.

Tenéis mucha más información sobre *doctest* en la documentación oficial: https://docs.python.org/2/library/doctest.html.

1.3.2 2. *Unittest*

La librería *unittest* es probablemente la más utilizada por los desarrolladores al no necesitar instalarse de forma externa. Tiene algunas limitaciones en cuanto a cómo descubrir el código que se quiere probar de forma automática, etc., pero es la aproximación por defecto a los *test* unitarios en Python.

Veamos un ejemplo con varias funcionalidades de la librería:

```
In [7]: # Primero de todo importamos la librería:
        import unittest


        # Definimos una clase para probar algunas funciones de la clase string
        # a modo de ejemplo:
        class TestStringMethods(unittest.TestCase):

            # Test para probar como pasar un string a mayúsculas:
            def test_upper(self):
                # assertEqual es una función especial que comprueba si ambos
                # argumentos son iguales o no:
                self.assertEqual('foo'.upper(), 'FOO')

            # Test para probar si un string contiene solo mayúsculas:
            def test_isupper(self):
                # assertTrue comprueba si el valor que se devuelve es TRUE
                self.assertTrue('FOO'.isupper())

                # assertFalse hace lo propio para FALSE
                self.assertFalse('Foo'.isupper())

            # Test para comprobar si dividimos un string por espacios de forma
```

```
            # correcta:
            def test_split(self):
                s = 'hello world'
                # Notad que compara entre listas posición por posición:
                self.assertEqual(s.split(), ['hello', 'world'])


        if __name__ == '__main__':
            """
            Por defecto deberíamos utilizar la siguiente línea:

            unittest.main()

            pero tenemos que hacerlo de la siguiente forma para que sea
            posible utilizar unittest en los notebook:
            """
            unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```
...
----------------------------------------------------------------------
Ran 5 tests in 0.006s

OK
```

Como podéis ver, aparece un mensaje que indica el número de tests que se han ejecutado y su resultado. Si alguno de los tests fuera erróneo, aparecería una *E* y no el carácter "ů", y en el caso de que devolvieran un valor erróneo, este sería una *F*.

Si volvemos al ejemplo en el que utilizamos *doctest*, podemos escribir varios *test* de forma diferente:

```
In [6]: import unittest


        def square(x):
            return x * x


        class TestSquareFunction(unittest.TestCase):

            def test_positive(self):
                a = 2.0
                self.assertEqual(square(a), 4.0)

            def test_negative(self):
                a = -3.0
                self.assertEqual(square(a), 9.0)
```

```
        if __name__ == '__main__':
            unittest.main(argv=['first-arg-is-ignored'], exit=False)

...
----------------------------------------------------------------------
Ran 5 tests in 0.024s

OK
```

1.3.3 2.1. Métodos de confirmación

El método *assertEqual* que hemos introducido antes no es el único método de confirmación de la librería *unittest*. Tenéis la lista completa en la documentación oficial, aunque aquí tenéis algunos de los más importantes:

- *assertIsNone(x)*: comprueba si la variable *x* es *None*.
- *assertIn(a, b)*: comprueba si *a* está incluído en *b*, muy útil para listas o tuplas.
- *assertNotIn(a, b)*: comprueba la condición contraria a *assertIn*.

1.3.4 2.2. Estructura de los *test*

Habitualmente, la estructura que seguiremos para escribir nuestros tests será la siguiente:

```
def test_if_square_in_list():
    # Preparamos la estructura del test:
    first_squares = [1.0, 4.0, 9.0, 16.0]

    # Ejecutamos la función que queremos probar:
    a_square = square(2.0)

    # Devolvemos el valor del test:
    assert (a_square in first_square)
```

Es muy importante notar que para que *unittest* descubra los test que queremos ejecutar, la palabra *test* ha de aparecer, ya sea en mayúscula o en cualquier combinación de mayúsculas y minúsculas, al principio del nombre de nuestros *test*.

Otra forma de organizar los *test* es utilizando las funciones ***setUp()*** y ***teardown()*** que se encargan, respectivamente, de preparar la estructura del test y de limpiar, si hace falta, al finalizar el test. Veamos un sencillo ejemplo de ello:

```
import unittest

class TestPhoneBook(unittest.TestCase):
    """ Clase para probar si ciertos países están en la agenda de prefijos."""
```

```
    def setUp(self):
        """ Creamos nuestra agenda de prefijos """
        self.phonebook = {34: 'Spain', 33: 'France', 32: 'Belgium', 44: 'UK'}

    def tearDown(self):
        """ Limpiamos la agenda al finalizar el test """
        self.phonebook = None

    def test_if_spain_included(self):
        """ Probamos si España está incluída. """
        self.assertIn('Spain', self.phonebook.values())
```

1.4 Resumen

Hemos visto diferentes formas de probar nuestro código y algunos ejemplos muy sencillos. En el *notebook* de la entrega trabajaremos más a fondo algunos de estos conceptos.