Tema 6. Introducción a las pruebas unitarias de código.

Vicente Benjumea García

Programación-II

Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 6. Introducción a las pruebas unitarias de código.

- Introducción al desarrollo de pruebas unitarias de código.
- Desarrollo de pruebas unitarias de código.
- Estudio de cobertura.
- Anexo. Captura de la entrada y salida de datos.

Introducción al desarrollo de pruebas unitarias de código

Objetivos en el Desarrollo de Software

- Desarrollar un sistema software que sea **correcto**:
  - Que se comporte según las **especificaciones** del problema a resolver.
  - Que cumpla los **requisitos** y **restricciones** especificados.
  - Que cuando sea ejecutado, resuelva el problema **correctamente**, para **todos los valores** posibles, y **siempre** que sea ejecutado.

- Es muy **difícil** tener la **seguridad** de que el software es **correcto**, incluso para proyectos de miles de millones de euros.
  - Cuanto más complejo es un sistema software, más difícil es tener la seguridad de su corrección.
- Incluso los mejores programadores y analistas cometen errores.
  - Cuanto más complejo es un determinado problema, más complejo es el software que lo resuelve.
- Incluso los errores más simples (conversión de tipos, desbordamientos aritméticos, divisiones por cero, etc) hacen incorrecto al software.
- Las **consecuencias** del software erróneo pueden ser muy **costosas**, tanto en vidas humanas, como monetarias, tiempo y esfuerzo.
- Debemos seguir **metodologías de programación y estrategias adecuadas** para desarrollar **software de calidad**, que minimice las posibilidades de introducir errores en el software, y facilite la depuración (detección y corrección) de los errores.

Introducción al desarrollo de pruebas unitarias de código

¿ Como comprobar si un programa es correcto ?

- **Prueba experimental**: se ejecuta el programa múltiples veces para conjuntos de datos adecuadamente seleccionados, y se comprueba si los resultados son los esperados.
  - Si durante alguna prueba se produce algún **error**, entonces el programa es **incorrecto**.
  - Sin embargo, aunque las pruebas sí hayan producido los resultados esperados, **NO** se puede concluir que el programa sea correcto.
  - **La prueba experimental puede ser empleada para mostrar la presencia de errores, pero nunca su ausencia**.

LA PRUEBA EXPERIMENTAL
PUEDE MOSTRAR QUE UN PROGRAMA SEA INCORRECTO,
PERO NUNCA PUEDE MOSTRAR QUE UN PROGRAMA SEA CORRECTO.

- La única forma de demostrar la corrección de un programa es mediante la **Verificación Formal** de programas.
  - Es un método matemático basado en técnicas de **demostración de teoremas**.
  - Es un método **complejo** y **difícil** de llevar a la práctica.

Introducción al desarrollo de pruebas unitarias de código

Prueba experimental

- Se debe desarrollar una **batería de casos de prueba** para conseguir una **cobertura** adecuada en la comprobación del software bajo estudio, tanto en número de sentencias de **código ejecutado**, como en las diferentes **bifurcaciones de ramas**.
  - La **selección de datos** para los casos de pruebas debe intentar mostrar los **posibles errores** que estén ocultos en el código, considerando los **valores límite/frontera**.
    - Por ejemplo, si el software comprueba si una persona tiene edad suficiente para votar, en los casos de prueba, se deben considerar los valores frontera 17 y 18.
    - Por ejemplo si el software comprueba si el valor del mes es adecuado, en los casos de prueba, se deben considerar los valores frontera 0, 1, 12 y 13.

Cobertura

- Existen herramientas que permiten comprobar la **cobertura** del software alcanzada en una determinada prueba del software. Este estudio de cobertura ayuda en el **diseño de los casos de prueba**.

Introducción al desarrollo de pruebas unitarias de código

Pruebas de caja negra

- Pruebas de **caja negra**: diseña los casos de prueba del software **sin conocer** ni observar el **código interno** que lo compone. Se basa en la **especificación** y **requisitos** de los componentes del software.
  - El desconocimiento del funcionamiento interno puede influir en realizar un diseño de casos de prueba con una selección de datos **incompleta** o **inadecuada**, que no compruebe todas las decisiones y ramas existentes.

Pruebas de caja blanca

- Pruebas de **caja blanca**: diseña los casos de prueba del software **conociendo el código interno** que lo compone. El conocimiento del código puede ayudar a diseñar **casos de pruebas más completos**, que comprueben las diferentes ramas y caminos que puede seguir la ejecución del software.
  - Diseñar las pruebas basándose en el conocimiento del código puede hacer que los **errores en el código induzcan errores en los casos de prueba**. Por ejemplo si el código no realiza algunas comprobaciones, es posible que los casos de prueba no consideren esas posibilidades.
  - Si se **modifica el código**, puede hacer que las pruebas ya diseñadas no sean válidas, o sean **incorrectas**, por lo que requiere que también se modifiquen las pruebas asociadas.

Introducción al desarrollo de pruebas unitarias de código

Pruebas de caja gris

- Pruebas de **caja gris**: diseña los casos de prueba utilizando la **especificación** y **requisitos** de los componentes del software, así como también el conocimiento *superficial* del **código interno** del software, y de las diferentes ramas y caminos que existen en el código.

Pruebas de robustez

- Pruebas de **robustez**: diseña los casos de prueba enfocados en comprobar el comportamiento del software en presencia de **entradas de datos no válidas**.

Pruebas de regresión

- Las pruebas de **regresión** consisten en la aplicación al código actual de los **casos de prueba** desarrollados para **versiones anteriores** del código, de tal forma que permite comprobar que la nueva versión del código no ha introducido **nuevos errores** en la parte del código que ya estaba funcionando correctamente.

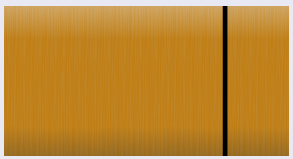
Introducción al desarrollo de pruebas unitarias de código

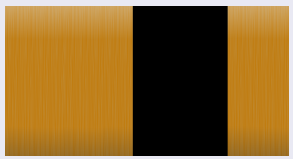
Desarrollo dirigido por pruebas (TDD)

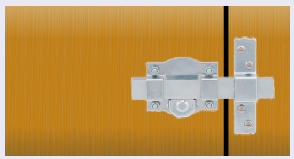
- El **desarrollo dirigido por pruebas** es una metodología en la que primero se diseñan los casos de prueba basados en la especificación y requisitos del sistema (como **caja negra**), y posteriormente se va desarrollando el software, y comprobándolo contínuamente con los casos de prueba previamente diseñados.

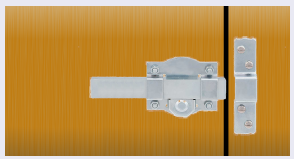
Pruebas de integración

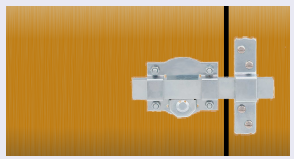
- Las pruebas unitarias comprueban las unidades básicas del software. No obstante, por sí solas no son suficientes, ya que es necesario comprobar que la **unión de los componentes básicos** para crear un sistema complejo también funciona correctamente.

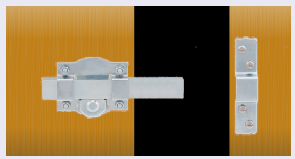

puerta cerrada


puerta abierta
(es una puerta deslizante)


cerradura cerrada


cerradura abierta


puerta cerrada y
cerradura cerrada


puerta abierta y (error)
cerradura cerrada (error)
(es una puerta deslizante)

Pruebas unitarias de código (*unit testing*)

- Las **pruebas unitarias** de código (*unit testing*) comprueban de forma **sistemática** el comportamiento de las unidades de software bajo estudio, según una batería de **casos de prueba** previamente diseñados.
- Cada **prueba unitaria** desarrolla un único **caso de prueba**, que especifica el procedimiento a seguir para comprobar adecuadamente un **determinado comportamiento** de la unidad bajo estudio.
  - Dado un **entorno de prueba** adecuado.
  - A partir de unos **datos de entrada**.
  - Se **ejecuta** la unidad bajo estudio.
  - Se **comprueba** que los **resultados** obtenidos coinciden con los resultados esperados, así como también se comprueban los **efectos** producidos en el entorno de prueba.
  - En caso de ser necesario, se realiza una **limpieza** del entorno de trabajo.

- Para poder comprobar el software, se deben desarrollar múltiples **pruebas unitarias**, según la batería de **casos de prueba** diseñados, con el objetivo de alcanzar una **cobertura** del software adecuada.

Desarrollo de pruebas unitarias de código (*unit testing*)

- Python dispone de varias herramientas que proporcionan marcos de trabajo adecuados para el desarrollo y realización de pruebas unitarias de código.
  - El módulo **unittest**, perteneciente a la librería estándar del lenguaje.
  - También existen otras herramientas externas, por ejemplo **pytest** (`https://docs.pytest.org/`).

- En este curso, desarrollaremos las pruebas unitarias utilizando el módulo **unittest**, perteneciente a la librería estándar del lenguaje.
  - Para cada **módulo de código** que se desee comprobar, se desarrollará otro **módulo de pruebas**, que contenga las **pruebas unitarias** adecuadas para ello.
  - El **nombre** del módulo de pruebas debe comenzar con la palabra **test_** seguida por el nombre del módulo de código que va a comprobar.
    - Por ejemplo, para comprobar el comportamiento del código definido en el módulo **dato.py**, se desarrollará el módulo denominado **test_dato.py**, conteniendo las pruebas unitarias diseñadas para ello.

Desarrollo de pruebas unitarias de código

Módulos de pruebas

- El **nombre** del módulo de pruebas debe comenzar con la palabra **test_** seguida por el nombre del módulo de código que va a comprobar.
- El módulo de pruebas:
  - Debe importar al módulo `unittest` y al **módulo de código a comprobar**.
  - Debe definir una o varias **clases de pruebas**, que contienen los casos de prueba.
  - La función principal **main()** se encarga de **ejecutar** las pruebas especificadas.

```python
# Módulo: test_dato.py  # debe comenzar por test_ seguido por nombre del módulo a comprobar
import unittest         # módulo para pruebas unitarias
import dato             # módulo de código a comprobar

class Test010DatoMethods(unittest.TestCase):
    """Esta clase comprueba los métodos de la clase Dato, del módulo dato."""

class Test020DatoFunctions(unittest.TestCase):
    """Esta clase comprueba las funciones del módulo dato."""

def main() -> None:
    unittest.main(module=__name__, verbosity=2) # permite utilizar desde otros módulos

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

Desarrollo de pruebas unitarias de código

Clases de pruebas

- El objetivo de las **clases de pruebas** es el desarrollo de las pruebas unitarias encargadas de comprobar el funcionamiento del código correspondiente.
  - Cada clase de pruebas define su propio **entorno de pruebas**.
  - Se deben definir tantas **clases de prueba** en el módulo de pruebas, como sean necesarias para tener el código bien estructurado, según la cantidad de *entornos de prueba* diferentes que sean necesarios.
  - El **nombre** de cada clase de pruebas debe comenzar con la palabra **Test**, seguida por un **código de orden**, por el **nombre de la clase** que va a comprobar y **nombre del método** a comprobar.
  - Cada clase de pruebas debe **heredar** de `unittest.TestCase`.
  - Las clases de prueba se **comprueban** en **orden lexicográfico**, por ello se debe especificar el código de orden en el nombre de la clase.

```
# Módulo: test_dato.py  # debe comenzar por test_ seguido por nombre del módulo a comprobar

class Test010DatoMethods(unittest.TestCase):
    """Esta clase comprueba los métodos de la clase Dato, del módulo dato."""

class Test020DatoFunctions(unittest.TestCase):
    """Esta clase comprueba las funciones del módulo dato."""
```

Desarrollo de pruebas unitarias de código

Entorno de pruebas (*test fixture*)

- Cada **clase de pruebas** define su propio **entorno de pruebas** (*test fixture*).
- El **entorno de pruebas** define el entorno de trabajo donde se realizan las pruebas, así como las acciones de limpieza necesarias cuando termine cada prueba. Para ello, se deben definir los siguientes métodos:
  - `setUp(self)`: se ejecuta **antes de cada prueba**, y se encarga de **crear el entorno de trabajo** donde se realizan las pruebas, tales como la creación de los objetos necesarios para realizar las pruebas, la creación de ficheros de datos, etc.
  - `tearDown(self)`: se ejecuta **después de cada prueba**, y se encarga de realizar las acciones necesarias de **limpieza**, después de realizar cada prueba, tales como la destrucción de los objetos creados, la eliminación de los ficheros de datos, etc.

```
class Test010DatoMethods(unittest.TestCase):
    """Esta clase comprueba los métodos de la clase Dato, del módulo dato."""

    def setUp(self) -> None:
        """Crea el entorno de pruebas. Se ejecuta antes de cada prueba."""
        self.dato = dato.Dato(5)

    def tearDown(self) -> None:
        """Destruye el entorno de pruebas. Se ejecuta después de cada prueba."""
        del self.dato
```

Pruebas unitarias (I)

- Las **pruebas unitarias** se definen como **métodos de la clase de pruebas** correspondiente, y podrán utilizar, si es necesario, el **entorno de pruebas** definido por el método `setUp(self)` de la misma clase.
  - Los métodos que desarrollan las pruebas unitarias deben comenzar con la palabra **test_**, seguida por un **código de orden** de varios dígitos, seguido por el **nombre del método** principal que se está comprobando.
  - Los métodos de prueba se **comprueban** en **orden lexicográfico**, por ello se debe especificar el código de orden en el nombre del método.

```
class Test010DatoMethods(unittest.TestCase):
    def test_010_ctor(self) -> None:
        """Deps: __init__, get_value."""
        valor = self.dato.get_value()
        self.assertEqual(valor, 5)
```

Pruebas unitarias (II)

- Las **pruebas unitarias** se definen como **métodos de la clase de pruebas** correspondiente, y podrán utilizar, si es necesario, el **entorno de pruebas** definido por le método `setUp(self)` de la misma clase.
  - Cada prueba unitaria debe comprobar un **único y determinado comportamiento**, característica, o funcionalidad concreta.
    - A veces, varios métodos de la clase a comprobar pueden estar involucrados en una comprobación. Se debe intentar **minimizar la cantidad de métodos**, de la clase a comprobar, involucrados en dicha prueba.
  - Cada método debe tener una documentación (**doc-string**) que proporcione información sobre el caso de prueba que se está realizando, y los nombres de los métodos involucrados en dicha prueba, incluyendo el entorno de la prueba. Si la prueba falla, los métodos involucrados pueden tener algún error.

```
class Test010DatoMethods(unittest.TestCase):
    def test_010_ctor(self) -> None:
        """Deps: __init__, get_value."""
        valor = self.dato.get_value()
        self.assertEqual(valor, 5)
```

Pruebas unitarias (III)

- Cada prueba unitaria suele seguir el siguiente esquema:
  - Sección de **preparación**: prepara el entorno de trabajo y los datos necesarios para ser utilizados en la siguiente sección.
  - Sección de **actuación**: invoca al método bajo prueba, con los datos preparados en la sección anterior.
  - Sección de **comprobación**: comprueba los resultados y efectos que produce la ejecución de la sección anterior, utilizando **asertos** para estas comprobaciones. A continuación se explicará el concepto de **aserto**.

Pruebas unitarias (IV)

- Si durante la ejecución de una prueba unitaria se **lanza** alguna **excepción**, entonces se considera que dicha **prueba unitaria ha fallado**, y por lo tanto **existe algún error** en el código que está siendo comprobado.
  - No obstante, en determinados contextos de comprobación, es correcto que el código a comprobar lance excepciones. En estos casos, las excepciones lanzadas se pueden comprobar con:

```
with self.assertRaises(TypeError):
    self.dato.set_value("hola")  # debería lanzar una excepción
```

Desarrollo de pruebas unitarias de código

Asertos (I)

Método	Comprueba que
`self.assertEqual(a, b)`	`a == b`
`self.assertNotEqual(a, b)`	`a != b`
`self.assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`
`self.assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`
`self.assertTrue(x)`	`bool(x) is True`
`self.assertFalse(x)`	`bool(x) is False`
`self.assertIs(a, b)`	`a is b`
`self.assertIsNot(a, b)`	`a is not b`
`self.assertIsNone(x)`	`x is None`
`self.assertIsNotNone(x)`	`x is not None`
`with self.assertRaises(exc): código`	`código` lanza `exc`

Asertos (II)

Método	Comprueba que
`self.assertGreater(a, b)`	`a > b`
`self.assertGreaterEqual(a, b)`	`a >= b`
`self.assertLess(a, b)`	`a < b`
`self.assertLessEqual(a, b)`	`a <= b`
`self.assertIsInstance(a, b)`	`isinstance(a, b)`
`self.assertNotIsInstance(a, b)`	`not isinstance(a, b)`
`self.assertIn(a, b)`	`a in b`
`self.assertNotIn(a, b)`	`a not in b`
`self.assertCountEqual(a, b)`	a y b mismos elementos, sin importar el orden
`self.fail(msg)`	fallo incondicional del test, muestra msg

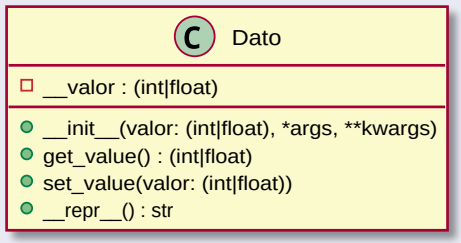
Desarrollo de pruebas unitarias de código. Ejemplo

Abstracción: Dato numérico

Un *Dato numérico* representa un objeto que almacena un determinado valor numérico (*int* o *float*).

- Comportamiento de un *dato numérico*:
  - Especificar el valor numérico almacenado.
  - Consultar el valor numérico almacenado.
  - Modificar el valor numérico almacenado.
  - Obtener la representación textual del *dato numérico*.
- Estado interno del *punto*:
  - El valor numérico almacenado.

Representación gráfica: diagrama de clases en UML



- En los métodos del diagrama no aparece el parámetro `self`.

Desarrollo de pruebas unitarias de código. Ejemplo

Módulo dato.py

```python
# Módulo: dato.py

"""Este módulo define la clase Dato"""

class Dato:
    """clase que almacena un determinado valor numérico"""

    def __init__(self, valor: (int|float), *args, **kwargs) -> None:
        """inicializa un objeto de la clase"""
        if not isinstance(valor, (int, float)):
            raise TypeError("Valor no es número")
        self.__valor = valor
        super().__init__(*args, **kwargs)

    def get_value(self) -> (int|float):
        """devuelve el valor almacenado"""
        return self.__valor

    def set_value(self, valor: (int|float)) -> None:
        """modifica el valor almacenado"""
        if not isinstance(valor, (int, float)):
            raise TypeError("Valor no es número")
        self.__valor = valor

    def __repr__(self) -> str:
        """devuelve la representación textual del objeto"""
        return f"Dato({self.__valor})"
```

```python
def main() -> None:
    """programa principal de prueba"""
    dt = Dato(5)
    assert(repr(dt) == "Dato(5)")
    assert(dt.get_value() == 5)
    dt.set_value(7)
    assert(repr(dt) == "Dato(7)")
    assert(dt.get_value() == 7)
    print("Prueba correcta")

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

Desarrollo de pruebas unitarias de código. Ejemplo

Módulo test_dato.py

```python
# Módulo: test_dato.py      # debe comenzar por test_ seguido por nombre del módulo a comprobar

import unittest             # módulo para pruebas unitarias
import dato                 # módulo de código a comprobar

class Test010DatoCtor(unittest.TestCase):
    """Esta clase comprueba el constructor y método get de la clase
    Dato, del módulo dato.
    El nombre de la clase debe comenzar por Test.
    La clase debe heredar de unittest.TestCase
    Los tests se comprueban en orden lexicográfico.
    El nombre de los tests debe comenzar por test_ seguido por el
    número de orden y el nombre del método principal a comprobar."""

    def setUp(self) -> None:
        """Crea el entorno de pruebas (test fixture).
        Creación de objetos generales utilizables para cada prueba.
        Se ejecuta antes de cada prueba."""
        self.dato = dato.Dato(5)

    def tearDown(self) -> None:
        """Destruye el entorno de pruebas (test fixture).
        Se ejecuta después de cada prueba, para realizar limpieza."""
        del self.dato
```

Desarrollo de pruebas unitarias de código. Ejemplo

Módulo test_dato.py (continuación)

```
# Módulo: test_dato.py (continuación)
# class Test010DatoCtor(unittest.TestCase): # (continuación)

    def test_010_ctor(self) -> None:
        """Deps: __init__, get_value."""
        valor = self.dato.get_value()
        self.assertEqual(valor, 5)

    def test_020_ctor(self) -> None:
        """Deps: __init__, get_value."""
        dt = dato.Dato(5.5)
        valor = dt.get_value()
        self.assertAlmostEqual(valor, 5.5)

    def test_030_ctor(self) -> None:
        """Deps: __init__."""
        with self.assertRaises(TypeError):# La siguiente invocación debería lanzar una excepción
            dt = dato.Dato(3+3j)          # los números complejos son un valor "frontera"

    # Duplicado de test_010_ctor
    # def test_040_get(self) -> None:
    #     """Deps: __init__, get_value."""
    #     valor = self.dato.get_value()
    #     self.assertEqual(valor, 5)
```

Desarrollo de pruebas unitarias de código. Ejemplo

Módulo test_dato.py (continuación)

```
class Test020DatoSet(unittest.TestCase):
    """Esta clase comprueba los métodos set y get de la clase Dato, del módulo dato."""
    def setUp(self) -> None:
        """Crea el entorno de pruebas (test fixture)."""
        self.dato = dato.Dato(5)
    def tearDown(self) -> None:
        """Destruye el entorno de pruebas (test fixture)."""
        del self.dato

    def test_010_set(self) -> None:
        """Deps: __init__, get_value, set_value."""
        self.dato.set_value(7)
        valor = self.dato.get_value()
        self.assertEqual(valor, 7)

    def test_020_set(self) -> None:
        """Deps: __init__, get_value, set_value."""
        self.dato.set_value(7.7)
        valor = self.dato.get_value()
        self.assertAlmostEqual(valor, 7.7)

    def test_030_set(self) -> None:
        """Deps: __init__, get_value, set_value."""
        with self.assertRaises(TypeError):# La siguiente invocación debería lanzar una excepción
            self.dato.set_value(3+3j)     # los números complejos son un valor "frontera"
        valor = self.dato.get_value()
        self.assertEqual(valor, 5)
```

Desarrollo de pruebas unitarias de código. Ejemplo

Módulo test_dato.py (continuación)

```python
# Módulo: test_dato.py (continuación)
class Test030DatoRepr(unittest.TestCase):
    """Esta clase comprueba el método repr de la clase Dato, del módulo dato."""

    def setUp(self) -> None:
        """Crea el entorno de pruebas (test fixture)."""
        self.dato = dato.Dato(5)

    def tearDown(self) -> None:
        """Destruye el entorno de pruebas (test fixture)."""
        del self.dato

    def test_010_repr(self) -> None:
        """Deps: __init__, __repr__"""
        valor = repr(self.dato)
        self.assertEqual(valor, "Dato(5)")
```

Desarrollo de pruebas unitarias de código. Ejemplo

Módulo test_dato.py (continuación)

```python
# Módulo: test_dato.py (continuación)

class Test040DatoMain(unittest.TestCase):
    """Esta clase comprueba la función main del módulo dato."""

    def test_010_main(self) -> None:
        """Deps: main, resto de métodos del módulo."""
        valor = dato.main()          # si lanza excepciones, falla el test
        self.assertIsNone(valor)

def main() -> None:
    unittest.main(module=__name__, verbosity=2) # permite utilizar desde otros módulos

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

Desarrollo de pruebas unitarias de código. Ejemplo

Salida de las pruebas unitarias (ejecución de test_dato.py)

```
test_010_ctor (__main__.Test010DatoCtor.test_010_ctor)
Deps: __init__, get_value. ... ok
test_020_ctor (__main__.Test010DatoCtor.test_020_ctor)
Deps: __init__, get_value. ... ok
test_030_ctor (__main__.Test010DatoCtor.test_030_ctor)
Deps: __init__. ... ok
test_010_set (__main__.Test020DatoSet.test_010_set)
Deps: __init__, get_value, set_value. ... ok
test_020_set (__main__.Test020DatoSet.test_020_set)
Deps: __init__, get_value, set_value. ... ok
test_030_set (__main__.Test020DatoSet.test_030_set)
Deps: __init__, get_value, set_value. ... ok
test_010_repr (__main__.Test030DatoRepr.test_010_repr)
Deps: __init__, __repr__ ... ok
test_010_main (__main__.Test040DatoMain.test_010_main)
Deps: main, resto de métodos del módulo. ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
```

Estudio de cobertura

Estudio de Cobertura

- Existen herramientas (módulo **coverage**: `https://coverage.readthedocs.io/`) que permiten comprobar la **cobertura** del software alcanzada en una determinada prueba del software.
  - Se debe realizar el estudio de cobertura, tanto del número de sentencias de **código ejecutado**, como en **bifurcaciones de ramas**.
  - Este estudio de cobertura ayuda en el **diseño de los casos de prueba**.

- Para poder utilizar el módulo **coverage** desde un programa Python (*Thonny*) es conveniente que el **módulo de pruebas unitarias** especifique la función **main** como se indica a continuación:

```
import unittest          # módulo para pruebas unitarias
import dato              # módulo de código a comprobar

class Test010DatoMethods(unittest.TestCase):
    """Esta clase comprueba los métodos de la clase Dato, del módulo dato."""

def main() -> None:
    unittest.main(module=__name__, verbosity=2) # permite utilizar desde otros módulos

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

Estudio de cobertura para dato.py y test_dato.py

Salida de texto (cobertura de dato.py y test_dato.py)

```
Name           Stmts   Miss Branch BrPart  Cover   Missing
-----------------------------------------------------------
dato.py           24      1      6      1    93%   39
test_dato.py      51      1      6      1    96%   98
-----------------------------------------------------------
TOTAL             75      2     12      2    95%
```

Salida a fichero index.html (cobertura de dato.py y test_dato.py)

Coverage report: 95%

coverage.py v6.5.0, created at 2024-02-02 20:21 +0100

Module	*statements*	*missing*	*excluded*	*branches*	*partial*	*coverage*
dato.py	24	1	0	6	1	93%
test_dato.py	51	1	0	6	1	96%
Total	**75**	**2**	**0**	**12**	**2**	**95%**

Estudio de cobertura para dato.py y test_dato.py

Cobertura de código dato.py

Coverage for **dato.py**: 93%

24 statements | 23 run | 1 missing | 0 excluded | 1 partial

« prev ^ index » next coverage.py v6.5.0, created at 2024-02-02 20:21 +0100

```
1  # Módulo: dato.py
2
3  """Este módulo define la clase Dato"""
4
5  class Dato:
6      """clase que almacena un determinado valor numérico"""
7
8      def __init__(self, valor: (int|float), *args, **kwargs) -> None:
9          """inicializa un objeto de la clase"""
10         if not isinstance(valor, (int, float)):
11             raise TypeError("Valor no es número")
12         self.__valor = valor
13         super().__init__(*args, **kwargs)
14
15     def get_value(self) -> (int|float):
16         """devuelve el valor almacenado"""
17         return self.__valor
18
19     def set_value(self, valor: (int|float)) -> None:
20         """modifica el valor almacenado"""
21         if not isinstance(valor, (int, float)):
22             raise TypeError("Valor no es número")
23         self.__valor = valor
24
25     def __repr__(self) -> str:
26         """devuelve la representación textual del objeto"""
27         return f"Dato({self.__valor})"
28
29 def main() -> None:
30     """programa principal de prueba"""
31     dt = Dato(5)
32     assert(repr(dt) == "Dato(5)")
33     assert(dt.get_value() == 5)
34     dt.set_value(7)
35     assert(repr(dt) == "Dato(7)")
36     assert(dt.get_value() == 7)
37
38 if __name__ == "__main__":                                    38 ↛ 39
39     main()
```

line 38 didn't jump to line 39, because the condition on line 38 was never true

Estudio de cobertura para dato.py y test_dato.py

Cobertura de código test_dato.py

Coverage for **test_dato.py**: 96%

51 statements | 50 run | 1 missing | 0 excluded | 1 partial

« prev ^ index » next coverage.py v6.5.0, created at 2024-02-02 20:21 +0100

```python
# Módulo: test_dato.py      # debe comenzar por test_ seguido por nombre del módulo a c

import unittest             # módulo para pruebas unitarias
import dato                 # módulo de código a comprobar

class Test010DatoCtor(unittest.TestCase):
    """Esta clase comprueba el constructor y método get de la clase
    Dato, del módulo dato.
    El nombre de la clase debe comenzar por Test.
    La clase debe heredar de unittest.TestCase
    Los tests se comprueban en orden lexicográfico.
    El nombre de los tests debe comenzar por test_ seguido por el
    número de orden y el nombre del método principal a comprobar."""

    def setUp(self) -> None:
        """Crea el entorno de pruebas (test fixture).
        Creación de objetos generales utilizables para cada prueba.
        Se ejecuta antes de cada prueba."""
        self.dato = dato.Dato(5)

    def tearDown(self) -> None:
        """Destruye el entorno de pruebas (test fixture).
        Se ejecuta después de cada prueba, para realizar limpieza."""
        del self.dato

    def test_010_ctor(self) -> None:
        """Deps: __init__, get_value."""
        valor = self.dato.get_value()
        self.assertEqual(valor, 5)

    def test_020_ctor(self) -> None:
        """Deps: __init__, get_value."""
        dt = dato.Dato(5.5)
        valor = dt.get_value()
        self.assertAlmostEqual(valor, 5.5)

    def test_030_ctor(self) -> None:
        """Deps: __init__."""
        with self.assertRaises(TypeError):# La siguiente invocación debería lanzar una
            dt = dato.Dato(3+3j)          # los números complejos son un valor "fronte

class Test020DatoSet(unittest.TestCase):
    """Esta clase comprueba los métodos set y get de la clase Dato, del módulo dato."""
    def setUp(self) -> None:
        """Crea el entorno de pruebas (test fixture)."""
        self.dato = dato.Dato(5)
    def tearDown(self) -> None:
        """Destruye el entorno de pruebas (test fixture)."""
        del self.dato

```

test_dato.py: 96% | 50 | 1 | 0 | 1

```python

    def test_010_set(self) -> None:
        """Deps: __init__, get_value, set_value."""
        self.dato.set_value(7)
        valor = self.dato.get_value()
        self.assertEqual(valor, 7)

    def test_020_set(self) -> None:
        """Deps: __init__, get_value, set_value."""
        self.dato.set_value(7.7)
        valor = self.dato.get_value()
        self.assertAlmostEqual(valor, 7.7)

    def test_030_set(self) -> None:
        """Deps: __init__, get_value, set_value."""
        with self.assertRaises(TypeError):# La siguiente invocación debería lanzar una
            self.dato.set_value(3+3j)      # los números complejos son un valor "front
        valor = self.dato.get_value()
        self.assertEqual(valor, 5)

class Test030DatoRepr(unittest.TestCase):
    """Esta clase comprueba el método repr de la clase Dato, del módulo dato."""

    def setUp(self) -> None:
        """Crea el entorno de pruebas (test fixture)."""
        self.dato = dato.Dato(5)

    def tearDown(self) -> None:
        """Destruye el entorno de pruebas (test fixture)."""
        del self.dato

    def test_010_repr(self) -> None:
        """Deps: __init__, __repr__"""
        valor = repr(self.dato)
        self.assertEqual(valor, "Dato(5)")

class Test040DatoMain(unittest.TestCase):
    """Esta clase comprueba la función main del módulo dato."""

    def test_010_main(self) -> None:
        """Deps: main, resto de métodos del módulo."""
        valor = dato.main()         # si lanza excepciones, falla el test
        self.assertIsNone(valor)

def main() -> None:
    unittest.main(module=__name__, verbosity=2) # permite utilizar COVERAGE desde Pytho

if __name__ == "__main__":                                    97 98
    main()
```

line 97 didn't jump to line 98, because the condition on line 97 was never true

Estudio de cobertura para pruebas unitarias

Código para realizar el estudio de cobertura para pruebas unitarias

```
#---------------------------------
# Nombre del fichero: pycoverage.py
#---------------------------------
import sys
import coverage.cmdline

def exec_coverage(args: list[str]) -> None:
    sys.argv.clear()
    sys.argv.append("coverage.py")
    sys.argv.extend(args)
    coverage.cmdline.main()

def main() -> None:
    exec_coverage("erase".split())

    exec_coverage("run --source=. --branch --omit pycoverage.py,unittesting_utils.py -m unittest discover".split())

    exec_coverage("report --no-skip-covered -i -m --omit pycoverage.py,unittesting_utils.py".split())

    exec_coverage("html --no-skip-covered -i --omit pycoverage.py,unittesting_utils.py".split())

    exec_coverage("erase".split())

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

- Genera el fichero html `htmlcov/index.html` con el análisis de cobertura individualizado sobre cada módulo, e información adicional.

Estudio de Cobertura

- Se debe realizar el estudio de cobertura, tanto del número de sentencias de **código ejecutado**, como en **bifurcaciones de ramas**.
  - No obstante, aunque se alcance un **100% de cobertura**, tanto en número de sentencias de *código ejecutado*, como en *bifurcaciones de ramas*, **NO se puede tener la garantía de que el software sea correcto**, ya que puede haber combinaciones de datos que no hayan sido comprobadas.

- Es importante alcanzar el **100% de cobertura** del código que se desee comprobar, tanto en número de sentencias de *código ejecutado*, como en *bifurcaciones de ramas*, pero eso sólo **no es suficiente**. Además:
  - Es muy importante el diseño de una **batería de casos de prueba** adecuada.
  - Es muy importante la **selección** de los **datos** para los casos de prueba, teniendo en cuenta los **valores límite/frontera**.

Estudio de cobertura media_01.py y test_media_01.py

```python
# Módulo: media_01.py
def calc_media(lista_nums: list[int|float]) -> float:
    """devuelve la media de la lista de números."""
    suma = 0
    if len(lista_nums) > 0:
        for num in lista_nums:
            suma += num
    return suma // len(lista_nums)

# Este código tiene dos ERRORES
# (división por cero, y división entera)
# aunque ha superado las pruebas
# y tiene un 90% de cobertura
```

```python
# Módulo: test_media_01.py
import unittest
import media_01

class Test010MediaFunctions(unittest.TestCase):
    def test_010_calc_media(self) -> None:
        """Deps: calc_media"""
        valor = media_01.calc_media([3.5, 4.5, 7.0])
        self.assertAlmostEqual(valor, 5.0)

def main() -> None:
    unittest.main(module=__name__, verbosity=2)
```

```
test_010_calc_media (test_media_01.Test010MediaFunctions.test_010_calc_media)
Deps: calc_media ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
Name                Stmts   Miss Branch BrPart  Cover   Missing
---------------------------------------------------------------
media_01.py             6      0      4      1    90%   4->7
test_media_01.py        8      0      0      0   100%
---------------------------------------------------------------
TOTAL                  14      0      4      1    94%
```

Estudio de cobertura `media_01.py` y `test_media_01.py`

Coverage for **media_01.py**: 90%

6 statements | 6 run | 0 missing | 0 excluded | 1 partial

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:42 +0200

```
1  def calc_media(lista_nums: list[int|float]) -> float:
2      """devuelve la media de la lista de números."""
3      suma = 0
4      if len(lista_nums) > 0:                                  4 ↓ 7
5          for num in lista_nums:
6              suma += num
7      return suma // len(lista_nums)
```

line 4 didn't jump to line 7, because the condition on line 4 was never false

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:42 +0200

Coverage for **test_media_01.py**: 100%

8 statements | 8 run | 0 missing | 0 excluded | 0 partial

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:45 +0200

```
1  import unittest
2  import media_01
3
4  class TestMediaFunctions(unittest.TestCase):
5      def test_010_calc_media(self):
6          """Deps: calc_media"""
7          valor = media_01.calc_media([3.5, 4.5, 7.0])
8          self.assertAlmostEqual(valor, 5.0)
9
10 def main():
11     unittest.main(module=__name__, verbosity=2)
```

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:45 +0200

Estudio de cobertura media_02.py y test_media_02.py

```python
# Módulo: media_02.py
def calc_media(lista_nums: list[int|float]) -> float:
    """devuelve la media de la lista de números."""
    media = 0
    if len(lista_nums) > 0:
        suma = 0
        for num in lista_nums:
            suma += num
        media = suma // len(lista_nums)
    return media

# Este código tiene un ERROR
# (división entera)
# aunque ha superado las pruebas
# y tiene un 100% de cobertura
```

```python
# Módulo: test_media_02.py
import unittest
import media_02

class Test010MediaFunctions(unittest.TestCase):
    def test_010_calc_media(self) -> None:
        """Deps: calc_media"""
        valor = media_02.calc_media([3.5, 4.5, 7.0])
        self.assertAlmostEqual(valor, 5.0)

    def test_020_calc_media(self) -> None:
        """Deps: calc_media"""
        valor = media_02.calc_media([])
        self.assertAlmostEqual(valor, 0.0)

def main() -> None:
    unittest.main(module=__name__, verbosity=2)
```

```
test_010_calc_media (test_media_02.Test010MediaFunctions.test_010_calc_media)
Deps: calc_media ... ok
test_020_calc_media (test_media_02.Test010MediaFunctions.test_020_calc_media)
Deps: calc_media ... ok
----------------------------------------------------------------------
Ran 2 tests in 0.000s
OK
Name               Stmts   Miss Branch BrPart  Cover   Missing
--------------------------------------------------------------
media_02.py            8      0      4      0   100%
test_media_02.py      11      0      0      0   100%
--------------------------------------------------------------
TOTAL                 19      0      4      0   100%
```

Estudio de cobertura `media_02.py` y `test_media_02.py`

Coverage for **media_02.py**: 100%

8 statements | 8 run | 0 missing | 0 excluded | 0 partial

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:42 +0200

```
1  def calc_media(lista_nums: list[int|float]) -> float:
2      """devuelve la media de la lista de números."""
3      media = 0
4      if len(lista_nums) > 0:
5          suma = 0
6          for num in lista_nums:
7              suma += num
8          media = suma // len(lista_nums)
9      return media
```

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:42 +0200

Coverage for **test_media_02.py**: 100%

11 statements | 11 run | 0 missing | 0 excluded | 0 partial

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:43 +0200

```
1  import unittest
2  import media_02
3
4  class TestMediaFunctions(unittest.TestCase):
5      def test_010_calc_media(self):
6          """Deps: calc_media"""
7          valor = media_02.calc_media([3.5, 4.5, 7.0])
8          self.assertAlmostEqual(valor, 5.0)
9
10     def test_020_calc_media(self):
11         """Deps: calc_media"""
12         valor = media_02.calc_media([])
13         self.assertAlmostEqual(valor, 0.0)
14
15 def main():
16     unittest.main(module=__name__, verbosity=2)
```

« prev ^ index » next coverage.py v6.5.0, created at 2023-08-31 18:43 +0200

Anexo. Captura de la entrada y salida de datos (I)

Utilidad para capturar la entrada (input) y la salida de datos (print)

- La clase `IOCapture` permite proporcionar los valores de entrada que serán leídos por la función `input()`. También captura la información mostrada en pantalla por la función `print()`.
- Está definida como un **gestor de contexto**, por lo que se debe utilizar con la sentencia **with** para que la entrada/salida se vuelva a restaurar al finalizar el procesamiento.

```
def test_entrada_salida_01(self) -> None:
    with IOCapture("pepe luis\n6.5\nana luisa\n7.0\neva\n9.0\n\n\n") as io_capture:
        modulo.main()
        valor1 = modulo.funcion_01(7, 8)
        obj1 = modulo.Clase(1, 2, 3)
        valor2 = obj1.metodo_01(4, 5)
    self.assertEqual(valor1, 56)
    self.assertEqual(valor2, 20)
    self.assertEqual(io_capture.get_stderr_value(), "")
    self.assertEqual(io_capture.get_stdout_value(), "Introduce Nombre: "
        "Introduce nota: Introduce Nombre: Introduce nota: "
        "Introduce Nombre: Introduce nota: Introduce Nombre: "
        "Alumno: pepe luis 6.5\nAlumno: ana luisa 7.0\nAlumno: eva 9.0\n")
```

Anexo. Captura de la entrada y salida de datos (II)

Utilidad para capturar la entrada (input) y la salida de datos (print)

```
import io
import sys
from typing import Optional, Any
class IOCapture:
    """Captura sys.stdout, sys.stderr y sys.stdin"""

    def __init__(self, invalue: Optional[str] = None, *args, **kwargs) -> None:
        """recibe el valor inicial para sys.stdin"""
        self.__invalue = invalue
        if not isinstance(self.__invalue, str):
            self.__invalue = ""
        self.__stdin_org = sys.stdin
        self.__stdout_org = sys.stdout
        self.__stderr_org = sys.stderr
        self.__stdin: Optional[io.StringIO] = None
        self.__stdout: Optional[io.StringIO] = None
        self.__stderr: Optional[io.StringIO] = None
        self.__stdout_value: Optional[str] = None
        self.__stderr_value: Optional[str] = None
        super().__init__(*args, **kwargs)
```

Anexo. Captura de la entrada y salida de datos (III)

Utilidad para capturar la entrada (input) y la salida de datos (print)

```
    def __enter__(self) -> 'IOCapture':
        self.__stdout_value = None
        self.__stderr_value = None
        self.__stdin = io.StringIO(self.__invalue)
        self.__stdout = io.StringIO()
        self.__stderr = io.StringIO()

        sys.stdin = self.__stdin
        sys.stdout = self.__stdout
        sys.stderr = self.__stderr

        return self
```

Anexo. Captura de la entrada y salida de datos (IV)

Utilidad para capturar la entrada (input) y la salida de datos (print)

```
    def __exit__(self, exc_type: Any, exc_value: Any, exc_tb: Any) -> None:
        sys.stdin = self.__stdin_org
        sys.stdout = self.__stdout_org
        sys.stderr = self.__stderr_org
        self.__stdout_value = self.__stdout.getvalue()
        self.__stderr_value = self.__stderr.getvalue()
        self.__stdin.close()
        self.__stdout.close()
        self.__stderr.close()
        del self.__stdin
        del self.__stdout
        del self.__stderr
        return None
```

```
    def get_stdout_value(self) -> Optional[str]:
        """devuelve el string capturado de sys.stdout"""
        return self.__stdout_value

    def get_stderr_value(self) -> Optional[str]:
        """devuelve el string capturado de sys.stderr"""
        return self.__stderr_value
```