

# Puesta en Producción Segura

Unidad 1.

Pruebas de software y Entornos de ejecución controlados



*"En cada búsqueda apasionada  
cuenta más la persecución que el  
objeto perseguido"*

**—El Tao del Jeet Kune Do” (1975), Bruce Lee**

# Objetivos

- **Definir la Calidad:** **Explicar** el objetivo fundamental de las Pruebas de Software, diferenciando claramente entre los conceptos de **Verificación** (construir bien el producto) y **Validación** (construir el producto correcto).
- **Identificar Niveles:** **Identificar y describir** los cuatro niveles de prueba clave dentro del ciclo de vida del desarrollo, tal como se estructuran en el **Modelo en V** (Unitaria, Integración, Sistema y Aceptación).
- **Aplicar Pruebas Unitarias:** **Describir** los casos de prueba ideales para la fase Unitaria, haciendo énfasis en la prueba de **casos límite** y **casos extremos** de manera aislada.
- **Distinguir Integración:** **Diferenciar y analizar** las metodologías de Pruebas de Integración (incremental vs. no incremental), incluyendo los enfoques ascendente y descendente.
- **Clasificar Funcionalidad:** **Clasificar** los tipos de prueba según su propósito, distinguiendo entre **Pruebas Funcionales** (qué hace el sistema) y **Pruebas No Funcionales** (qué tan bien lo hace).
- **Aplicar Caja Negra:** **Utilizar** las técnicas de **Caja Negra** (Particiones de Equivalencia y Valores Límite) para diseñar casos de prueba sin acceder al código fuente interno.
- **Comprender la Cobertura:** **Describir** el objetivo de las técnicas de **Caja Blanca** (White Box) y **explicar** qué mide la métrica de **Cobertura de Sentencias** en el código.



# Contenidos

**01** Introducción

**02** Pruebas en los  
diferentes niveles  
SDLC

**03** Tipos de pruebas

**04** Realización de pruebas  
Unitarias

**05** Pruebas en Python



01

# Introducción



# Pruebas: Introducción

El ciclo de vida del software se compone de procesos muy complejos, por lo que es altamente probable que se produzcan errores:

- De diseño, de implementación, etc.

Además de no satisfacer los requisitos de ejecución, estos errores pueden producir vulnerabilidades que un atacante podría aprovechar.



# Pruebas - Introducción

- **Definición:** Pruebas de software son el proceso de verificar y validar que un software funciona como se espera.
- **Objetivo:** Asegurar la calidad y fiabilidad del software.
- **Personal:** Desarrolladores, o mejor, personal especializado y ajeno al proyecto.



# Importancia de las pruebas

- **Detección Temprana de Errores:** Ahorra tiempo y recursos.
- **Mejora de la Calidad del Software:** Garantiza un software más estable y confiable.
- **Mantenimiento:** Facilita la identificación y corrección de errores en el futuro.



# Verificación vs validación

**Verificación:** comprobación de que el software satisface los requisitos

- ¿el código está bien formado y se comporta como se espera?

**Validación:** comprobación de que el software satisface las necesidades de los usuarios

- ¿satisface todos los casos de uso?



# Verificación formal

- La **verificación formal** es un método por el cual se garantiza que el código cumplirá todas las especificaciones que le hayamos pedido.
  - Prueba matemática
- Desgraciadamente, es demasiado compleja y costosa por lo que sólo se aplica a elementos muy costosos como el hardware o a piezas de software críticas
  - P.ej.: el software de un transbordador espacial.



# Pruebas de software

En lugar de la verificación formal, se suelen hacer una serie de pruebas o tests menos formales para ver si el software cumple lo especificado:

- Si el código se ejecuta sin errores aparentes
- Si el código es capaz de aguantar altos niveles de carga de usuario o de datos a procesar
- Etc.





02

# Pruebas en los diferentes niveles del SDLC



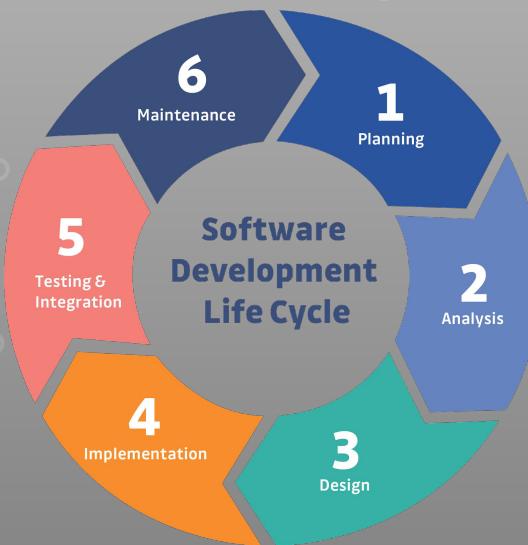
# Niveles en el ciclo de vida de desarrollo software

## ¿Qué es el SDLC?

El SDLC es el ciclo de vida de desarrollo software

## Fases del SDLC

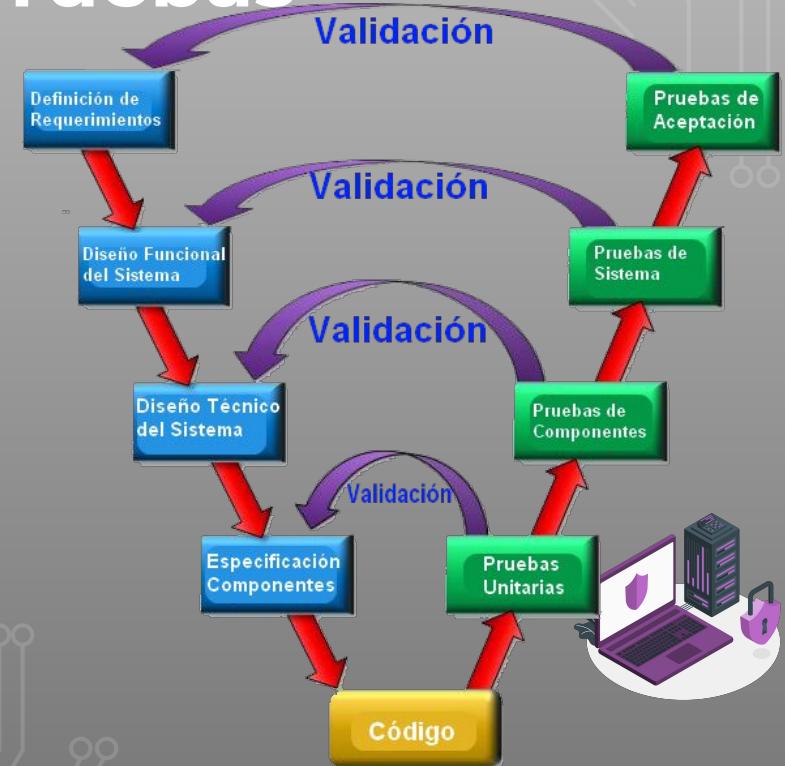
- Planificación
- Análisis
- Diseño
- Implementación
- Pruebas
- Integración
- Mantenimiento



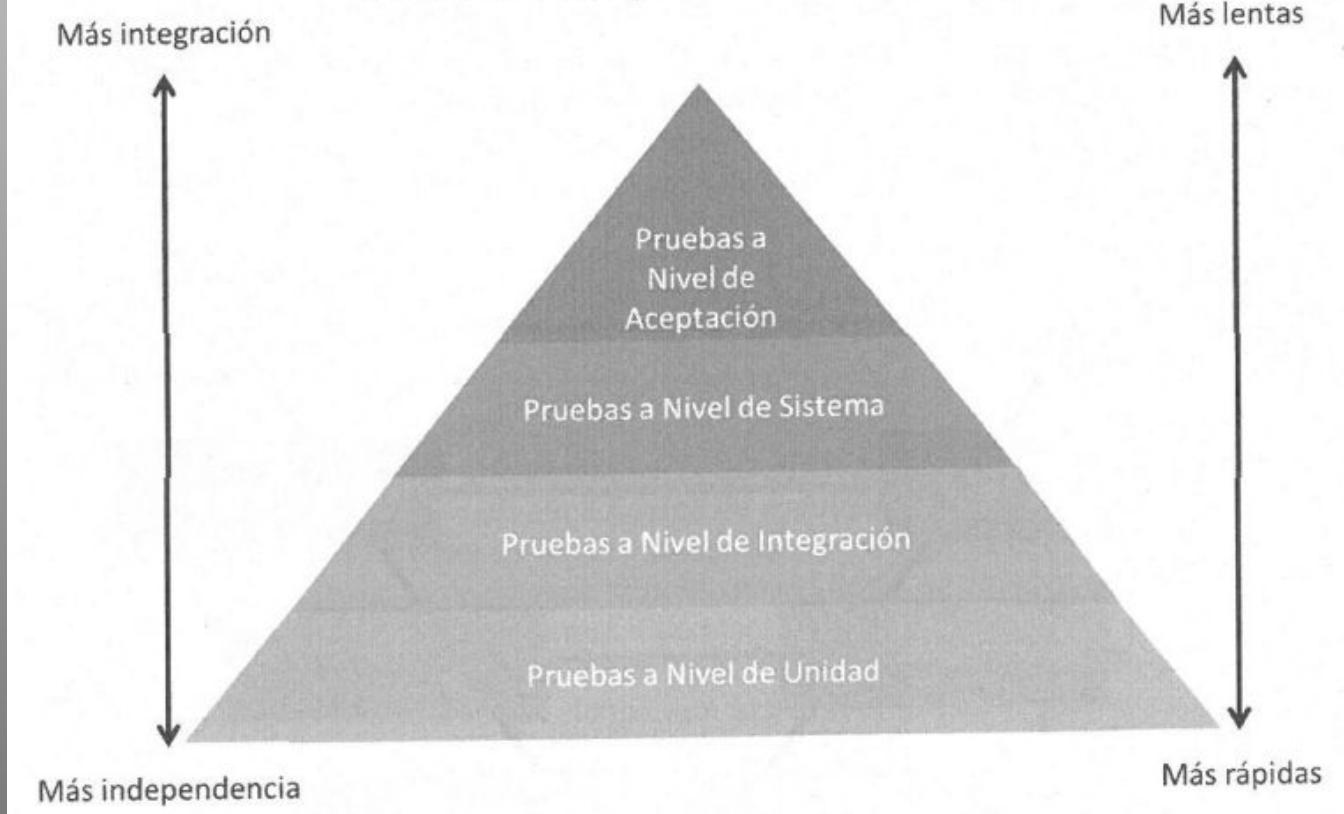
# Etapas o Niveles de Pruebas

## Modelo en V:

Indica cómo se debe de llevar a cabo la verificación y validación.



## Niveles de prueba software



# Pruebas Unitarias

**Definición:** Verifican componentes individuales en el nivel más bajo de la aplicación.

**Objetivo:** Asegurar que cada unidad de código funcione correctamente de manera aislada.

**Ejemplos:** Junit, PHPUnit, Nunit,Mocha -Chai, Unittest, Pytest.



# Pruebas unitarias

Las **pruebas unitarias** son pruebas funcionales, y por lo general automatizadas, de pequeñas porciones de software (p.ej.: módulos, clases) para tratar de ver si funcionan correctamente.

Se prueban especialmente casos extremos. P. ej.: si pasamos valores nulos, valores límite en un rango, tipos de datos o codificaciones inesperadas, etc.

También existe la variante de hacer testing aleatorio, creando objetos dispares.



# Pruebas de Integración

**Definición:** Verifican interacciones entre componentes y módulos del sistema.

**Objetivo:** Asegurar que los componentes integrados funcionen correctamente en conjunto.

**Ejemplos:** Pruebas de integración de APIs, pruebas de interacción entre módulos de software.



# Metodologías de Pruebas de integración

- **Pruebas no incrementales:** Se integran todos los componentes y se prueba el sistema de una vez.
- **Pruebas incrementales:** Se realiza integración gradual.
- **Integración desdendente:** Es incremental y se realiza desde el componente de más alto nivel.
- **Integración asdendente:** Es incremental y se realiza desde el componente de más bajo nivel.
- **Integración Mixta:** Conjuntamente varias estrategias.

# Pruebas del Sistema

**Definición:** Verifican el sistema completo.

**Objetivo:** Asegurar que el sistema cumpla con los requisitos especificados y funcione como un todo.

**Ejemplos:** Pruebas de extremo a extremo (end-to-end), pruebas funcionales completas.



# Pruebas del Sistema

- Pruebas de integración del sistema completo que comprueba el funcionamiento global del Sistema.
- No es necesario realizarlas por alguien que conozca el sistema.
- Pruebas desde el punto de vista del usuario.
- Permiten validar arquitectura y requisitos.



# Pruebas de Aceptación

**Definición:** Validan que el sistema cumple con los requisitos y expectativas del usuario final.

**Objetivo:** Obtener la aprobación del cliente o usuario final antes de la implementación. Se hacen en sucesivas rondas, cada vez con entornos más realistas, y se elaboran encuestas, estadísticas, etc.

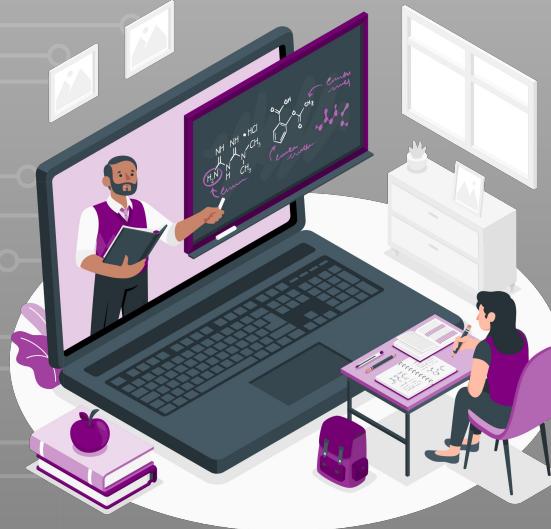
**Ejemplos:** Pruebas de usuario, pruebas de aceptación del cliente.



# Tipos de pruebas de Aceptación

- **Pruebas de humo:** revisión rápida para ver “si no explota”.
- **Pruebas alpha:** En el lugar del desarrollo por un cliente.
- **Pruebas beta:** Por el cliente en entornos de trabajo.
- **Pruebas de aceptación:** tras liberar la versión del software al público en general.





03

# Tipos de pruebas



# Clasificación de las pruebas

Las pruebas se pueden clasificar de diversos modos

- Por el modo de llevarlas a cabo:
  - **Pruebas manuales** vs **pruebas automáticas**.
- Por la ejecución o no del código:
  - **Pruebas estáticas** vs **pruebas dinámicas**.
- Según lo que verifican:
  - **Pruebas funcionales** vs **pruebas no funcionales**.
- Por el enfoque:
  - **Pruebas de caja negra** vs **pruebas de caja blanca**.



# Pruebas manuales vs. pruebas automáticas

- Las **pruebas manuales** requieren esfuerzo y por tanto son muy costosas.
- Como el software cambia constantemente, se intenta **automatizar las pruebas** de forma que puedan lanzarse en cualquier momento del ciclo de desarrollo y obtener un informe de resultados de forma inmediata.
  - ¡Sin prácticamente ningún coste!
  - Problema: no siempre es posible hacerlo.



# Pruebas estáticas vs. pruebas dinámicas

- Las **pruebas estáticas** son aquellas que se realizan sin ejecutar el código de la aplicación.
  - **Revisión de la documentación.**
  - **Inspección del código: Manual o empleando herramientas automatizadas.**
- Las **pruebas dinámicas** requieren la ejecución del código. Estas también se puede realizar de forma **manual o automáticamente**.
- También tenemos una conjugación de las estáticas y dinámicas: **herramientas de análisis interactivo.**



# Pruebas funcionales vs. pruebas no funcionales

Las pruebas **funcionales** son aquellas cuyo objetivo es comprobar que el Sistema o sus partes funciona como se espera.

Las pruebas **no funcionales** son aquellas que tienen por meta verificar otros requisitos relacionados con la calidad de los sistemas implementados: Accesibilidad, usabilidad, buen rendimiento, escalabilidad, seguridad, compatibilidad, portabilidad, etc.



# Pruebas Funcionales

**Definición:** Verifican que el sistema funcione de acuerdo con las especificaciones y requisitos funcionales.

**Objetivo:** Asegurar que cada función del software opere correctamente.

**Ejemplos:** Pruebas de casos de uso, exploratorias, pruebas de regresión, compatibilidad de entorno, de humo, de sanidad, de mono...



# Pruebas No Funcionales

**Definición:** Verifican aspectos no funcionales del sistema como rendimiento, seguridad, usabilidad.

**Objetivo:** Asegurar que el sistema cumpla con los requisitos de rendimiento y otros criterios no funcionales.

**Ejemplos:** Pruebas de carga, de estrés, de estabilidad, de rendimiento, de compatibilidad, de recuperación, de instalación, estructurales, de configuración, de escalabilidad, de tolerancia a fallos y de usabilidad.



# Pruebas funcionales: pruebas de regresión

Son las pruebas funcionales que se realizan **cada vez que se produce una nueva versión del software**, buscando:

- Si se introducen errores nuevos o se revelan errores que ya estaban previamente.
- Si una nueva característica introduce errores en otras características, sean nuevas o viejas.

Pueden incluir repetir todos los niveles de pruebas anteriores al completo o sólo partes.



# Pruebas no funcionales: pruebas de rendimiento

Las **Pruebas de Rendimientos** son pruebas no funcionales cuyo objetivo es comprobar si un programa:

- Se ejecuta rápido
- Consume pocos recursos

Hay varios tipos de pruebas de rendimiento dependiendo del objetivo buscado.

- A veces se hacen sin detallar los requisitos: p.ej.: Cuál debería ser la carga soportada.



# Diferentes pruebas de rendimiento

- **Pruebas de carga:** ver como se comporta el sistema con un número esperado de usuarios o de datos a procesar.
- **Pruebas de estrés:** aumentan la carga poco a poco para conocer el punto de ruptura del sistema.
- **Pruebas de estabilidad:** ver como se comporta el sistema a largo plazo con una carga normal.
- **Pruebas de pico:** ver como se comporta el sistema cuando hay picos y bajones repentinos de carga

# Pruebas no funcionales: pruebas de compatibilidad

Se consideran pruebas no funcionales para **comprobar el funcionamiento en distintos entornos.**

- Distintos sistemas operativos.
- Distintos navegadores.
- Entre versiones del mismo producto (p. ej.: pruebas de compatibilidad hacia atrás).

Los errores detectados pueden ser tanto funcionales como estéticos.



# Pruebas no funcionales: pruebas de usabilidad

Son pruebas no funcionales para comprobar si el usuario tiene problemas para interactuar con el sistema:

- Si produce **errores inesperados**.
- Si es capaz de llevar a cabo sus actividades de **forma rápida y eficiente**.
- Si tiene dificultades **tras no usar el software** durante un período de tiempo.
- Si se producen emociones positivas o negativas.



# Pruebas no funcionales: de escalabilidad y tolerancia a fallos.

Las **pruebas de escalabilidad** son pruebas no funcionales para comprobar **si el sistema puede soportar un incremento en la demanda de uso**: por ejemplo, si se pueden agregar nodos extra

Las **pruebas de tolerancia a fallos** comprueban si el sistema **es capaz de seguir trabajando cuando uno o varios los nodos que lo integran, fallan**.



# Pruebas de caja negra vs. pruebas de caja blanca

Las **pruebas de caja negra** sólo tienen en cuenta las entradas y salidas del componente a analizar, no cómo está programado.

- Para revisar la compatibilidad de interfaces, etc.

Las pruebas de **caja blanca** sí tienen en cuenta el código de los subsistemas del componente.

- Revisión de la lógica interna, el camino de datos, comprobación de bucles, etc.
- Suelen hacerse antes que las de caja negra.



# Pruebas de Caja Negra

**Definición:** Pruebas en las que el tester no conoce la estructura interna del sistema y se enfoca en la funcionalidad externa.

**Objetivo:** Verificar que las entradas generen las salidas esperadas sin conocer el código interno.

**Ejemplos:** Pruebas basadas en los requisitos y especificaciones, pruebas funcionales.



# Técnicas de pruebas de Caja Negra

- **Técnica de las Clases o Particiones de equivalencia:** Identificamos los tipos de valores diferentes y probamos uno de cada.
- **Técnica de análisis de valores límite:** Pruebas para los extremos de las diferentes clases de equivalencia.
- **Conjetura de errores:** listar otros errores que podrían ocurrir, p.ej.: login con usuario no registrado.



# Pruebas de Caja Blanca

**Definición:** Pruebas en las que el tester conoce la estructura interna del sistema y se enfoca en el código y lógica interna.

**Objetivo:** Verificar el flujo de control y los caminos lógicos dentro del código.

**Ejemplos:** Pruebas de cobertura de código, pruebas de rutas lógicas, revisiones de código.



# Técnicas de pruebas de Caja Blanca

Tenemos varias técnicas para realizar pruebas de caja Blanca:

- Cobertura de Sentencias.
- Cobertura de Decisión.
- Cobertura de Condiciones.
- Cobertura de Decisión/condición.
- Cobertura de Condición Múltiple.
- Cobertura de Caminos.



# Técnicas de pruebas de Caja Blanca

**Cobertura de Sentencias:** Se asegura de que cada línea de código en el programa se haya ejecutado al menos una vez durante las pruebas.

**Cobertura de Decisión:** se asegura que cada condición dentro de una decisión (como un "if" o un "while") se pruebe tanto con un resultado verdadero como falso.

**Cobertura de Condiciones:** verifica que todas las ramas de un flujo de código condicional se hayan ejecutado al menos una vez durante las pruebas.



# Técnicas de pruebas de Caja Blanca

**Cobertura de Decisión/condición:** garantizar que se hayan probado todas las ramas lógicas posibles y que cada condición individual dentro de esas ramas haya sido evaluada tanto verdadera como falsa.

**Cobertura de Condición Múltiple:** garantizar que todas las rutas lógicas posibles dentro de una sección de código que involucra múltiples condiciones sean probadas

**Cobertura de Caminos:** garantiza que todas las rutas de ejecución posibles dentro de un programa han sido probadas.





04

# Realización de pruebas unitarias



# Realización de prueba unitarias

Para realizar las pruebas unitarias de un módulo, vamos a realizar:

- Particiones de equivalencia.
- Análisis de los valores límite.
- Cobertura de caminos.



# Particiones de equivalencia

Divide el dominio de entrada de un programa en clases de datos llamadas **particiones de equivalencia**.

1. Se identifican las particiones de entrada, **dividiendo** el dominio de entrada en **clases** donde se espera que el sistema **maneje los datos de manera similar**.
2. Se definen las **clases válidas** (aquellas que el sistema debe aceptar) y las **clases inválidas** (aquellas que el sistema debe rechazar).
3. Se elige un **valor representativo** de cada partición para ser utilizado en los casos de prueba.

# Análisis de los valores límite.

El **Análisis de los valores límites (AVL)** se centra en probar los límites de los rangos de entrada y salida de un programa.

1. Se identifican valores en los extremos de los rangos de entrada y salida para probar la robustez del sistema y detectar errores en las condiciones de borde.
2. Se diseñan casos de prueba que incluyen valores justo dentro y fuera de estos límites



# Cobertura de Caminos

## Técnica del camino básico.

Halla la complejidad lógica de un diseño que sirve para definición de un conjunto básico.

- Se dibuja el grafo de flujo asociado.
- Se calcula la complejidad ciclomática del grafo
- Se determina un conjunto básico de caminos independientes.
- Se preparan casos de prueba para cada uno de los caminos.





# 05

# Pruebas en Python



# Herramientas de Pruebas en Python

- **Unittest**: Biblioteca estándar de Python para pruebas unitarias.
- **PyTest**: Framework de pruebas robusto y flexible.
- **Nose**: Herramienta de pruebas que extiende unittest.



# Ejemplo

Para la siguiente función:

```
def isnumber(a):
```

```
    try:
```

```
        float(a)
```

```
        return True
```

```
    except ValueError:
```

```
        return False
```



# Clases de equivalencia y valores límite

Clase de Equivalencia	Ejemplos	Resultado Esperado
Enteros válidos	"123", "-5", 0	True
Decimales válidos	"3.14", "-0.1", "1e3"	True
Valores especiales float	"nan", "inf", "-inf"	True
Cadenas no numéricas	"hola", "123abc", "12,3"	False
Cadenas vacías / espacios	"", " "	False
Tipos convertibles	123, 3.14, True	True
Tipos no convertibles	None, [ ], { }	False
Valores límite (bordes)	"1e308", "1e-308", "+3", "--5"	/  según caso

# Pruebas con Unittest

```
import unittest  
class TestIsNumber(unittest.TestCase):  
    def test_enteros_validos(self):  
        self.assertTrue(isnumber("123"))  
        self.assertTrue(isnumber("-5"))  
        self.assertTrue(isnumber(0))  
    def test_decimales_validos(self):  
        self.assertTrue(isnumber("3.14"))  
        self.assertTrue(isnumber("-0.1"))  
        self.assertTrue(isnumber("1e3"))  
....  
if __name__ == "__main__":  
    unittest.main()
```



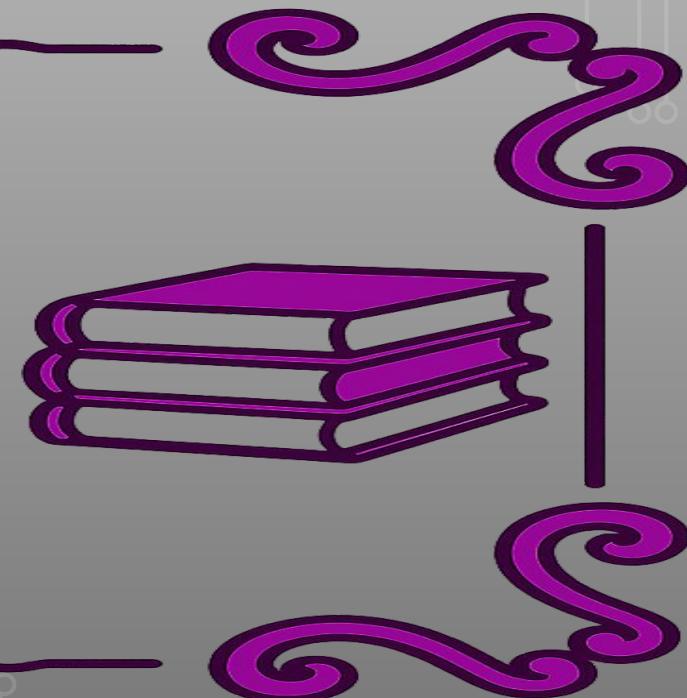
# Pruebas con pytest

```
import unittest  
def test_tipos_convertibles():  
    assert isnumber(123)  
    assert isnumber(3.14)  
    assert isnumber(True)  
def test_tipos_no_convertibles():  
    assert not isnumber(None)  
    assert not isnumber([])  
    assert not isnumber({})  
.....  
if __name__ == "__main__":  
    unittest.main()
```



# Bibliografía y Webgrafía

- Pruebas de software. *Rafael López García.*



# Gracias!

**¿Alguna pregunta?**



[informatica.iesvalledeljerteplasencia.es](mailto:informatica.iesvalledeljerteplasencia.es)



[coordinacion.cenfp@iesvp.es](mailto:coordinacion.cenfp@iesvp.es)



C/ Pedro y Francisco González, s/n  
10600, Plasencia (Cáceres)



927 01 77 74

