METODOLOGÍA DE PRUEBA DE SISTEMAS
Trabajo Práctico
Ebri Ivana

Consignas particulares para el desarrollo del TP

Modalidad de trabajo: individual.

Fecha de entrega: 15 de noviembre

Aspectos formales de presentación del trabajo

El trabajo deberá estar encabezado por los siguientes datos:

- Título del Trabajo con la herramienta a emplear
- Apellido y nombres del alumno
- Correo Electrónico

El trabajo tendrá una extensión mínima de 6 carillas y una extensión máxima de 12 carillas, ja tamaño de la hoja A4, estar escrito con letra Arial 11, sencillo y márgenes según el formato de este documento.

Formato final de entrega: PDF

Enviar al correo pabloa.perez@bue.edu.ar

Condición de aprobación: presentación en tiempo y forma con la consigna solicitada completa y su exposición.

Desarrollo del TP

Para la elaboración del trabajo practico deberán tener en cuenta los siguientes aspectos:

1. Elección de un marco de trabajo para pruebas unitarias según el lenguaje de programación que utilizara para el desarrollo.
2. Desarrollo una parte de una aplicación según la especificación de un requerimiento funcional.
3. Código de la aplicación por cada test case.
4. Preparación del plan de pruebas (al menos 5 casos de prueba de diferentes funciones).
5. Informe de las pruebas (pueden estar al final de cada test case)
6. Las conclusiones del trabajo (aspectos positivos y negativos de la herramienta que utilizo, con respecto al TP).

Descripción del modelo de negocio

Se requiere una aplicación para una organización de protección civil que permita dar la alerta temprana de emergencias como focos de incendio, inundaciones, ayuda a la comunidad, accidentes etc.

Para ello la aplicación permitir al usuario tomar una imagen del hecho, acontecimiento situación, asociarla con información ya sea de su ubicación como su descripción y enviarla para su solución.

Esto requiere que una parte del soft sea una app móvil para los usuarios receptores y una app web para los que administran y gestionan la información.

Toda la información que se eleva desde los receptores (desde la app móvil) es persistida en una base de datos (información abstracta).

Opciones para el desarrollo: pueden elegir back o front.

Alumno/a: Ebri Ivana

Elección de desarrollo: Back-end

Lenguaje: Python

Módulo de pruebas: Unittest

Entorno de desarrollo: Visual Studio Code

Módulo o lógica del negocio desarrollada: se desarrolló el módulo que conecta, lee y almacena la información enviada por el usuario por medio de la aplicación y los registros recibidos en el centro de gestión para su análisis, procesamiento y accionar frente a las emergencias reportadas, dadas las condiciones de desarrollo se optó por interpretar las imágenes como direcciones URL, por lo tanto, en la base de datos constan como un varchar

---

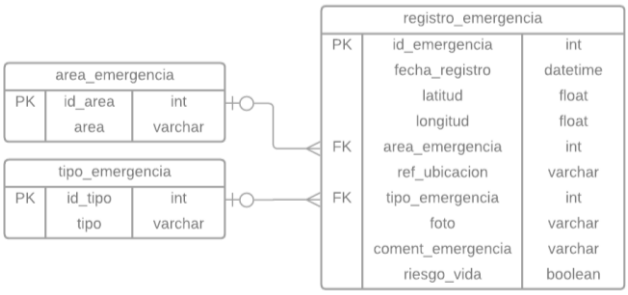
Elección del módulo de pruebas:
Se eligió la herramienta **unittest** dado que es el marco de trabajo estándar para realizar pruebas unitarias en Python. Está incluido en la biblioteca estándar del lenguaje y está diseñada para facilitar la creación y ejecución de pruebas de manera eficiente.

Proporciona las siguientes ventajas y características en la creación de pruebas unitarias:

1. Su estructura es modular: Permite la creación de conjuntos de pruebas organizados y modulares mediante clases y métodos específicos para cada prueba.
2. Cuenta con assertions integradas: Ofrece una amplia gama de métodos de aserción predefinidos para poder evaluar que los resultados de las pruebas sean los esperados. Por ejemplo, “assertEqual”, “assertTrue”, “assertFalse”, etcétera.
3. Fixture para pruebas: Permite el uso de métodos de configuración y limpieza antes y después de las pruebas, lo que garantiza un entorno de prueba consistente y controlado.
4. Es nativo de Python: Por lo que se puede descubrir automáticamente pruebas en un conjunto de archivos o directorios y garantiza la compatibilidad entre el lenguaje y el módulo, lo que simplifica la ejecución de todas las pruebas disponibles.
5. Integrable con otros marcos y herramientas: Aunque es el marco de pruebas unitarias estándar en Python, puede integrarse con otros marcos de pruebas o herramientas de ejecución de pruebas para adaptarse a necesidades más específicas.

En resumen, **unittest** es una elección sólida para realizar pruebas unitarias en Python debido a su integración directa con el lenguaje, su estructura modular y sus capacidades para verificar y validar el comportamiento de las funciones y clases en un entorno controlado.

Desarrollo del módulo de la aplicación según el requerimiento funcional:
Dada la amplitud del requerimiento presentado se optó por trabajar sobre el intercambio y almacenaje de datos entre el usuario y la aplicación y entre esta y el centro de gestión que la recibirá.



De manera inicial mediante el módulo **Peewee** y Python se armaron las clases que modelaran las tablas existentes en la base de datos para asegurar la persistencia de los mismos, estos están en un archivo exclusivo a esta función, llamado '**App_db.py**', contine la ruta de la base de datos, la clase **BaseModel** (Modelo que define sobre qué base de datos operaran todas las tablas, cada clase heredará de esta) y la definición de las clases **AreaEmergencia** y **TipoEmergencia** (representan la elección sobre listas desplegables en la aplicación real del usuario) y **ResgistroEmergencia** (que constituyen los registros de cada reporte de las emergencias, de esta tabla se alimenta el centro de gestión para visualizar las emergencias reportadas).
Luego se cuenta con el documento '**App_tablas_ref.py**' el cual contine la definición de las reglas de negocio y la forma de interactuar y cargar información a la tabla de **registro_emergencia**, esto está definido en la clase **App_emergencia**, cuyos métodos son:

- _conectar_db: como método privado contiene la conexión con la base de datos.
- _mapear_orm: utilizado para generar la base de datos creando las 3 tablas, fue empleado durante el desarrollo, sin embargo, no debería tener uso a posterior ya que se cuenta con la base de datos creada y conectada.
- _cargar_datos: a su vez fue empleada durante el desarrollo para cargar las opciones de los desplegables de Área y Tipo de emergencia.
- nueva_emergencia: método a emplear cuando el usuario pretende notificar una emergencia, en el caso de desarrollo planteado los datos se solicitan por consola dada la falta de front, únicamente acepta el ingreso por consola, no permite el paso de los mismos como parámetros. Contiene las verificaciones de los datos enviados, dejándote en un loop que te solicita el reingreso de la información incorrecta para continuar, finalmente con la carga de datos completa guarda el reporte en la tabla.
- _ultimas_emergencias: método auxiliar que en base a la cantidad de registros solicitados (valor entero positivo pasado como parámetro) retorna un listado de los últimos X registros ingresados de emergencias.
- envio_avisos: método cuyo uso está planteado desde el centro de control donde visualizarían los últimos registros para proceder con las llamadas o asistencias de emergencia pertinentes, solicita por consola (y verifica que cumpla con los requisitos de ser un valor entero y positivo mayor a 0) se le indique la cantidad de reportes a mostrar, recurre a la función _ultimas_emergencias pasándole esta cantidad de registros requeridos, para mostrarle con un formato especifico las ultimas emergencias reportadas; emplea la función ciudad_cercana que consigue en base la latitud y longitud pasadas (en una aplicación móvil real estas no serían tipeadas, sino tomadas de la localización GPS del propio móvil) para obtener la denominación del área donde se reporta la emergencia.
- ciudad_cercana: emplea la librería **geopy.geocoders** para obtener a partir de la latitud y longitud el nombre de la ubicación, región, ciudad o pueblo donde se reportó esta emergencia.

El documento '**Test-App_emergencias.py**' es en el que se encuentran definidas las clases y métodos de pruebas unitarias

Finalmente, el último documento es '**App_emergencias.py**' es donde se ejecuta la función main donde se instancia la aplicación y se pueden ejecutar a partir de ella sus diferentes métodos y funciones, cuenta con la importación de la clase App_emergencia.

Test case de la app de emergencias:

TC-UT-01

Nro TC	**TC-UT-01**
Nombre de TC	Conexión a base de datos
Descripción	Verificar que la conexión a la base de datos se realiza correctamente
Supuestos y condiciones previas	La aplicación se encuentra instanciada y se ejecuta la conexión a la base
Datos de prueba	-
Pasos a ejecutar	Instanciar y ejecutar la clase TestAppEmergenciaConexion
Resultado esperado	La base de datos está conectada
Resultado real/condiciones posteriores	Se ha conectado a la base de datos Se cierra la conexión
Estado de prueba	**APROBADO**

Test unitario

```
class TestAppEmergenciaConexion(unittest.TestCase):
    def setUp(self):
        self.app = App_emergencia()
        self.app._conectar_db()

    def tearDown(self):
        self.app._cerrar_db()
        sqlite_db.close()

    def test_conectar_db(self):
        """Verificar que la conexión a la base de datos se realiza correctamente"""
        self.assertTrue(sqlite_db.connect, "La conexion no se realizó")
```

TC-UT-02

Nro TC	**TC-UT-02**
Nombre de TC	Desconexión a base de datos
Descripción	Verificar que se cierra correctamente la conexión a la base de datos
Supuestos y condiciones previas	La aplicación se encuentra instanciada, se intenta cerrar la conexión, por si estuviese abierta, y se ejecuta la conexión a la base
Datos de prueba	-
Pasos a ejecutar	Instanciar y ejecutar la clase TestAppEmergenciaCierre
Resultado esperado	La base de datos está desconectada
Resultado real/condiciones posteriores	Se ha desconectado la base de datos
Estado de prueba	**APROBADO**

Test unitario

```
class TestAppEmergenciaCierre(unittest.TestCase):
```

```
    def setUp(self):
        self.app = App_emergencia()
        self.app._cerrar_db()
        self.app._conectar_db()

    def test_cerrar_conexion(self):
        """Verificar que la conexión a la base de datos se cierra correctamente"""
        self.app._cerrar_db()
        self.assertTrue(sqlite_db.is_closed(), "La conexion no se encuentra cerrada")
```

TC-UT-03

Nro TC	**TC-UT-03**
Nombre de TC	Emergencias solicitadas
Descripción	Se verifica si lee y devuelve de manera correcta la cantidad de emergencias solicitadas
Supuestos y condiciones previas	Se inicializo la aplicación
Datos de prueba	Cantidad de emergencias = 2
Pasos a ejecutar	Instanciar la clase TestAppEmergenciaMethod Ejecutar el método test_ultimas_emergencias
Resultado esperado	Devuelve una lista con 2 elementos (las dos emergencias solicitadas)
Resultado real/condiciones posteriores	La longitud de la lista es de 2
Estado de prueba	**APROBADO**

Test unitario

```
class TestAppEmergenciaMethods(unittest.TestCase):
    def setUp(self):
        self.app = App_emergencia()
    def test_ultimas_emergencias(self):
        """Verificar si se devuelven las últimas 2 emergencias correctamente"""
        emergencias=self.app._ultimas_emergencias(2)
        cant_registros = len(emergencias)
        self.assertEqual(cant_registros, 2)
```

TC-UT-04

Nro TC	**TC-UT-04**
Nombre de TC	Verificar coordenadas con direccion
Descripción	Se verifica que la función de rastreo de dirección este devolviendo la dirección correcta
Supuestos y condiciones previas	Se inicializo la aplicación
Datos de prueba	Ubicación 1 = -34.763704,-58.208684 Ubicación 2 = -34.544762,-58.450766
Pasos a ejecutar	Instanciar la clase TestAppEmergenciaMethod Ejecutar el método test_ciudad_cercana

Resultado esperado	Retorna las direcciones correctas a partir de las coordenadas enviadas
Resultado real/condiciones posteriores	Retorna “Diagonal Lisandro de la Torre, Berazategui, Partido de Berazategui, Buenos Aires, B1880BFG, Argentina” y “21642, Conector Interciclovías, Barrio River, Belgrano, Buenos Aires, Comuna 13, Ciudad Autónoma de Buenos Aires, C1424BCL, Argentina”
Estado de prueba	**APROBADO**

Test unitario

```
class TestAppEmergenciaMethods(unittest.TestCase):
    def setUp(self):
        self.app = App_emergencia()
    def test_ciudad_cercana(self):
        """Verificar si la función ciudad_cercana devuelve la ubicación esperada"""
        berazategui = self.app.ciudad_cercana(-34.763704,-58.208684)
        river= self.app.ciudad_cercana(-34.544762,-58.450766)
        self.assertEqual(berazategui.address,"Diagonal Lisandro de la Torre,
Berazategui, Partido de Berazategui, Buenos Aires, B1880BFG, Argentina")
        self.assertEqual(river.address, "21642, Conector Interciclovías, Barrio River,
Belgrano, Buenos Aires, Comuna 13, Ciudad Autónoma de Buenos Aires, C1424BCL,
Argentina")
```

TC-UT-05

Nro TC	**TC-UT-05**
Nombre de TC	Existencia de tablas de bd
Descripción	Comprueba que las tablas que componen la base de datos existan
Supuestos y condiciones previas	Se inicializo la aplicación
Datos de prueba	-
Pasos a ejecutar	Instanciar la clase TestAppEmergenciaMethod Ejecutar el método test_mapear_orm
Resultado esperado	Las tres tablas existen
Resultado real/condiciones posteriores	Las 3 tablas de la bd existen
Estado de prueba	**APROBADO**

Test unitario

```
class TestAppEmergenciaMethods(unittest.TestCase):
    def setUp(self):
        self.app = App_emergencia()
    def test_mapear_orm(self):
        """Verificar que la función _mapear_orm crea las tablas correctamente"""
        self.app._mapear_orm()
        #verifico si existen todas
        self.assertTrue(AreaEmergencia.table_exists())
```

```
        self.assertTrue(TipoEmergencia.table_exists())
        self.assertTrue(RegistroEmergencias.table_exists())
```

TC-UT-06

Nro TC	**TC-UT-06**
Nombre de TC	Comprobación de desplegables
Descripción	Las tablas que componen los desplegables tienen mas de 0 registros y contienen la cantidad de opciones de acuerdo a las reglas de negocio establecidas
Supuestos y condiciones previas	Se inicializo la aplicación
Datos de prueba	-
Pasos a ejecutar	Instanciar la clase TestAppEmergenciaMethod Ejecutar el método test_datos_desplegables
Resultado esperado	Las tablas no están vacías y contienen: - Áreas 7 registros - Tipo 5 registros
Resultado real/condiciones posteriores	Áreas no contiene 0 registros Áreas contine 7 registros Tipo no contiene 0 registros Tipo contine 5 registros
Estado de prueba	**APROBADO**

Test unitario

```
class TestAppEmergenciaMethods(unittest.TestCase):
    def setUp(self):
        self.app = App_emergencia()
    def test_datos_desplegables(self):
        """Verificar que las tablas de Área y Tipo de emergencia no se encuentran
vacías"""
        areas_count = AreaEmergencia.select().count()
        tipos_count = TipoEmergencia.select().count()
        # Realizar las aserciones
        self.assertNotEqual(areas_count, 0)
        self.assertEqual(areas_count, 7)
        self.assertNotEqual(tipos_count, 0)
        self.assertEqual(tipos_count, 5)
```

Conclusiones:

La herramienta UnitTest es una herramienta poderosa y fundamental en el desarrollo de software, ya que permite probar unidades aisladas de código para asegurarse de que funcionen como se espera. Sin embargo, aunque la herramienta en sí misma puede ser relativamente sencilla de usar, la verdadera complejidad a menudo radica en la planificación y la creación de casos de prueba efectivos.

El proceso de definir y establecer los casos de prueba es crucial, aquí es donde la verdadera dificultad puede surgir: identificar todos los posibles escenarios y resultados que se deben probar

para garantizar un comportamiento correcto del código. Esto implica anticipar y considerar una amplia gama de situaciones, desde las más simples hasta las más complejas, y asegurarse de que el código responda adecuadamente a todas ellas.
Además, el tiempo y esfuerzo dedicado a la elaboración de estos casos de prueba pueden ser considerablemente más largo que el proceso de escribir la misma cantidad de código.
Una vez establecidos los casos de prueba, el siguiente desafío reside en escribir el código que verificará estos casos y comparará los resultados esperados con los resultados reales. Esta fase requiere una comprensión detallada del funcionamiento del código y la implementación cuidadosa de las pruebas para asegurar una cobertura completa y precisa.

Por tanto, si bien la herramienta UnitTest en sí puede ser relativamente directa, la complejidad y el tiempo que implica el diseño y la elaboración de casos de prueba exhaustivos, junto con la implementación de la lógica de verificación, son aspectos críticos y que consumen mucho tiempo en el proceso de desarrollo de software.