Universidad Nacional de Ingeniería

Facultad de Ingeniería Eléctrica y Electrónica

Programación Orientada a Objetos (BMA15)

chisTHON: desarrollo de aplicación móvil para el aprendizaje del lenguaje Python

Integrantes:

- Frank Hugo Siesquen Mayerhoffer (20231418E)
- Jose Fernando Soto Salazar (20191232C)
- Sebastian Alonzo Condori Menacho (20221420G)

Docente:

- Yuri Oscar Tello Chancapoma

Lima, 2024

ÍNDICE

Introducción

El proyecto de software **chisTHON**, es una herramienta educativa diseñada para facilitar a los usuarios el aprendizaje de los conceptos básicos de programación del lenguaje Python. La aplicación permite la visualización de textos educativos, navegación por lecciones interactivas e incluye la ejecución de código Python en una consola.

Objetivos del Proyecto

Objetivo General

Desarrollar una aplicación móvil que facilite el acceso y aprendizaje a aquellas personas que empiezan a adentrarse en el mundo de la programación en Python, proporcionando a los usuarios herramientas para explorar los conceptos del lenguaje a través de textos educativos y la ejecución de código.

Objetivos Específicos

- Proveer un entorno de aprendizaje accesible: Permitir a los usuarios que no tengan experiencia previa en programación, acceder a contenidos educativos que les ayuden a aprender los conceptos básicos del lenguaje Python.
- Crear un sistema de autenticación de usuarios: Desarrollar una interfaz de inicio de sesión y registro de usuarios utilizando una base de datos SQL para almacenar y verificar la información de los usuarios.
- Integrar una consola para la ejecución de código Python: Desarrollar una consola integrada en la aplicación que permita a los usuarios ingresar y ejecutar su código en tiempo real.
- Optimizar la interfaz gráfica utilizando las librerías Kivy y KivyMD: Crear una interfaz visual y atractiva para los usuarios para ofrecer una experiencia fluida y funcional.
- Asegurar la confiabilidad del sistema mediante pruebas automatizadas: Implementar pruebas unitarias mediante la librería UNITTEST para validad la funcionalidad de las principales clases, asegurando así que el sistema sea robusto y confiable.

Proyectos similares

- **SoloLearn:** Davit Kocharyan y Yeva Hyusyan crearon SoloLearn, lanzando la primera versión en 2014. Es una plataforma de aprendizaje en línea líder en el ámbito de la programación. Ofrece una experiencia educativa innovadora y accesible, diseñada para personas de todos los niveles, desde principiantes hasta desarrolladores más experimentados. A través de una interfaz intuitiva y una amplia variedad de cursos, permite a los usuarios aprender los fundamentos y conceptos avanzados de múltiples lenguajes de programación.

- **Mimo:** Es una aplicación educativa diseñada para el aprendizaje interactivo de lenguajes de programación y conceptos relacionados con la informática. A través de lecciones cortas y prácticas, permite a los usuarios aprender a programar en diversos lenguajes como: Python, JavaScript, HTML, CSS, y SQL, entre otros. Mimo está orientado tanto a principiantes como a personas con conocimientos previos, ofreciendo un enfoque accesible y gradual que facilita la comprensión de los conceptos más complejos.

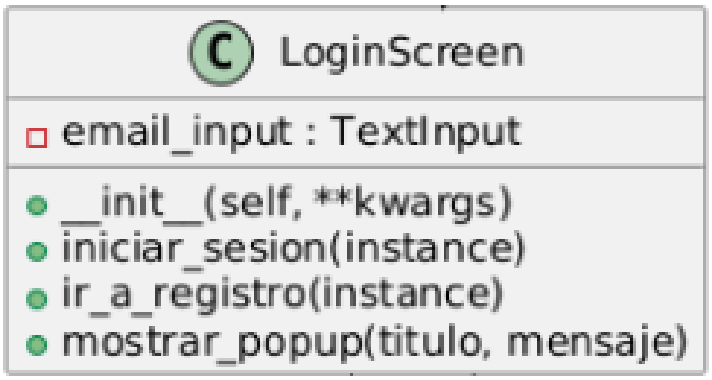
- **PyBites:** Es una plataforma centrada en el aprendizaje de Python a través de retos y ejercicios prácticos. Fue creada por desarrolladores para ayudar a otros a mejorar sus habilidades de Python mediante la resolución de problemas y la construcción de proyectos. La idea detrás de PyBites es que la mejor manera de aprender Python es aplicándolo en situaciones reales y con desafíos que promuevan la resolución de problemas.

Cronograma del proyecto

Semana	Responsable	Tarea asignada	Descripción	Inicio	Fin
1	Todos	Investigación y selección de librerías útiles	Se seleccionarán las tecnologías y librerías a usar en el desarrollo de la aplicación	29/09/2024	06/10/2024
2	Jose Soto	Planificación y diseño de la interfaz de usuario	Se desarrollará el funcionamiento principal de la aplicación con el usuario, está etapa se centra en el desempeño del rendimiento a la hora de la interacción	07/10/2024	15/10/2024
3	Frank Siesquen	Desarrollo del backend y base de datos	Creación de la base de datos en la cual se almacenará la información cada usuario (nombre de usuario, correo electrónico)	16/10/2024	26/10/2024
4	Sebastian Condorí	Implementación del módulo de aprendizaje	Implementa el módulo de aprendizaje, y la pantalla de este en el código	27/10/2024	31/11/2024
5	Sebastian Condorí	Implementación de los textos, consola y test al módulo de aprendizaje	Crea los textos de aprendizaje para el módulo, así como la consola para la ejecución de código Python y un test para poner aprueba los conocimientos de los usuarios	01/11/2024	18/11/2024
6	Jose Soto	Integración del fronted con el backend	Crea prototipos de la interfaz utilizando Kivy y Kivymd. Define cómo se verá la app: pantallas de inicio, menús, y las secciones donde se enseñarán los conceptos de Python	19/11/2024	26/11/2024
7	Frank Siesquen	Pruebas automatizadas	Utilizando la librería Unittest se añadirán cuatro pruebas automatizadas al código	27/11/2024	02/12/2024
8	Todos	Documentación	Se recopila toda la información del proyecto y se redacta en un documento	03/12/2024	07/12/2024

Clases y métodos del código (POO)

Clase LoginScreen: Representa la pantalla de inicio de sesión.



Métodos:

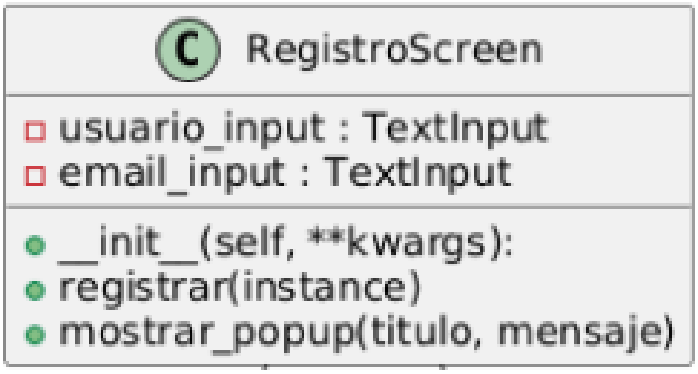
- **__init__(self, **kwargs):** Constructor de la clase que inicializa la pantalla de inicio de sesión.
  - Crea un BouxLayout vertical para los elementos de la interfaz.
  - Agrega widgets como etiquetas, un campo de texto para ingresar el correo electrónico y dos botones (uno para iniciar sesión y otro para ir a la pantalla de registro).
  - Asocia eventos a los botones mediante bind.

- **iniciar_sesion(self, instance):** Método vinculado al botón "Ingresar".
  - Verifica si el correo electrónico ingresado existe en la base de datos (pyodbc).
  - Si existe, cambia la pantalla a 'home'; Si no, muestra una ventana emergente con un mensaje de error.

- **ir_a_registro(self, instance):** Cambia la pantalla actual a 'registro'.

- **mostrar_popup(self, titulo, mensaje):** Muestra un Popup con un título y un mensaje especificado.

Clase RegistroScreen: Representa la pantalla de registro de nuevos usuarios.



Métodos:

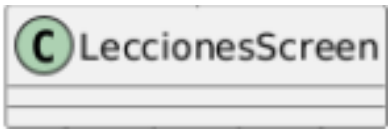
- **__init__(self, **kwargs):** Constructor que configura la interfaz de usuario para el registro.
  - Agrega campos de entrada para el nombre de usuario y el correo electrónico.
  - Incluye un botón para registrar al usuario, vinculando el evento de clic al método registrar.

- **registrar(self, instance):** Método para registrar un nuevo usuario.
  - Comprueba si el correo electrónico ya existe en la base de datos.
  - Si no existe, lo inserta en la tabla Usuarios.
  - Si el registro es exitoso, muestra una ventana emergente de confirmación y cambia a la pantalla 'home'.

- **mostrar_popup(self, titulo, mensaje):** Similar al método en LoginScreen. Muestra una ventana emergente para mensajes de error o confirmación.

Clase HomeScreen: Pantalla principal después del inicio de sesión. Es una subclase de MDScreen y no tiene métodos adicionales definidos en este código.

Clases: LeccionesScreen, Modulo1Screen, Modulo2Screen, Modulo3Screen, ConsoleScreen, Test1Screen, ResultadosScreen



- Cada una de estas clases representa una pantalla específica.
- Son subclases de MDScreen y funcionan como contenedores para widgets definidos en los archivos .kv.
- No tienen métodos propios definidos en este código.

Clase MainApp: Es la clase principal que define la aplicación.

Métodos:

- **build(self):**
  - Configura el tema y la paleta de colores.
  - Carga los archivos .kv que contienen la configuración de las interfaces.
  - Define y registra todas las pantallas en un ScreenManager.

- **cargar_texto_modulo1, cargar_texto_modulo2, cargar_texto_modulo3:**
  - Leen archivos de texto correspondientes a la teoría de los módulos.
  - Si el archivo no existe, devuelve un mensaje de error.

- **on_start(self):** Se ejecuta al iniciar la aplicación.
  - Registra el tiempo de inicio y carga el tiempo acumulado de uso desde un archivo.

- **on_stop(self):** Se ejecuta al cerrar la aplicación.
  - Calcula el tiempo total de uso de la aplicación y lo guarda en un archivo.

- **load_usage_time(self):** Carga el tiempo acumulado de uso desde el archivo usage_time.txt.

- **save_usage_time(self):** Guarda el tiempo acumulado de uso en el archivo usage_time.txt.

- **go_to_console(self):** Cambia la pantalla actual a 'console' y guarda la pantalla anterior.

- **go_back(self):** Vuelve a la pantalla anterior almacenada.

- **run_code(self):** Método que ejecuta código Python ingresado en un campo de texto de la pantalla 'console'. Redirige la salida estándar y errores de captura.

- **evaluate_answers(self):** Evalúa las respuestas del test y actualiza la barra de progreso en la pantalla 'results'.

- **clear_checkboxes(self):** Desmarca todas las casillas de verificación en la pantalla del test.

- **reset_quiz(self):** Reinicia el quiz y muestra una barra de progreso en 0.

Pruebas automatizadas

Las pruebas automatizadas ayudan a garantizar que el software funcione como se espera, incluso cuando se realizan cambios o se añade nueva funcionalidad. Al facilitar la detección de errores y mejorar la calidad del producto, son una parte esencial de un proceso de desarrollo ágil y eficiente. En este caso se utiliza 4 clases para las 4 pruebas que se harán.

Clase TestLoginScreen: Esta clase prueba la funcionalidad de inicio de sesión.

```
class TestLoginScreen(unittest.TestCase):
    def setUp(self):
        self.app = MDApp.get_running_app()
        self.login_screen = LoginScreen(name='login')
        self.app.root = ScreenManager()
        self.app.root.add_widget(self.login_screen)

    def test_iniciar_sesion_exito(self):
        self.login_screen.email_input.text = "usuario1@ejemplo.com"
        self.login_screen.iniciar_sesion(None)
        self.assertEqual(self.app.root.current, 'home')

    def test_iniciar_sesion_error(self):
        self.login_screen.email_input.text = "no_registrado1@ejemplo.com"
        self.login_screen.iniciar_sesion(None)
        self.assertNotEqual(self.app.root.current, 'home')
```

Métodos:

- **test_iniciar_sesion_exito():** Esta prueba simula un inicio de sesión exitoso. Ingresa el email "usuario1@ejemplo.com" al campo de email y llama al método iniciar_sesion() de la pantalla de login. Luego, verifica si

la pantalla actual de la aplicación (app.root.current) es "home", lo que indica un inicio de sesión correcto.

- **test_iniciar_sesion_error():** Esta prueba simula un inicio de sesión fallido. Ingresa el email "no_registrado1@ejemplo.com" al campo de email y llama al método iniciar_sesion() de la pantalla de login. Luego, verifica si la pantalla actual de la aplicación NO es "home", lo que indica un inicio de sesión fallido.

Clase TestRegistroScreen: Esta clase prueba la funcionalidad de registro de usuario.

```
class TestRegistroScreen(unittest.TestCase):
    def setUp(self):
        self.app = MDApp.get_running_app()
        self.registro_screen = RegistroScreen(name='registro')
        self.app.root = ScreenManager()
        self.app.root.add_widget(self.registro_screen)

    def test_registrar_exito(self):
        self.registro_screen.usuario_input.text = "nuevo_usuario"
        self.registro_screen.email_input.text = "nuevo2@ejemplo.com"
        self.registro_screen.registrar(None)
        self.assertEqual(self.app.root.current, 'home')

    def test_registrar_error(self):
        self.registro_screen.usuario_input.text = "usuario_existente"
        self.registro_screen.email_input.text = "existente2@ejemplo.com"
        self.registro_screen.registrar(None)
        self.assertNotEqual(self.app.root.current, 'home')
```

Métodos:

- **test_registrar_exito():** Esta prueba simula un registro exitoso. Ingresa un nombre de usuario y email nuevos a los campos correspondientes y llama al método registrar() de la pantalla de registro. Luego, verifica si la pantalla actual de la aplicación (app.root.current) es "home", lo que indica un registro correcto.
- **test_registrar_error():** Esta prueba simula un registro fallido. Ingresa un nombre de usuario y email ya existentes a los campos correspondientes y llama al método registrar() de la pantalla de registro. Luego, verifica si la pantalla actual de la aplicación NO es "home", lo que indica un registro fallido.

Clase TestMainApp: Esta clase prueba la funcionalidad de cargar un texto de un archivo.

```
class TestMainApp(unittest.TestCase):
    def setUp(self):
        self.app = MainApp()
        self.app.build()

    def test_cargar_texto_exito(self):
        resultado = self.app.cargar_texto()
        self.assertNotEqual(resultado, "No se pudo cargar el texto. Verifique que el archivo 'texto_largo.txt' esté en el directorio correcto.")

    def test_cargar_texto_error(self):
        self.app.cargar_texto = lambda: "No se pudo cargar el texto. Verifique que el archivo 'texto_largo.txt' esté en el directorio correcto."
        resultado = self.app.cargar_texto()
        self.assertEqual(resultado, "No se pudo cargar el texto. Verifique que el archivo 'texto_largo.txt' esté en el directorio correcto.")
```

Métodos:

- **test_cargar_texto_exito():** Esta prueba simula la carga exitosa de un texto. Llama al método cargar_texto() de la aplicación principal y verifica si el resultado NO es el mensaje de error indicando que el archivo no se pudo cargar.
- **test_cargar_texto_error():** Esta prueba simula la carga fallida de un texto. Sobreescribe el método cargar_texto() para que devuelva el mensaje de error y luego lo llama. Verifica si el resultado es igual al mensaje de error esperado.

Clase TestConsoleScreen: Esta clase prueba la funcionalidad de ejecutar código desde la consola de la aplicación.

```
class TestConsoleScreen(unittest.TestCase):
    def setUp(self):
        self.app = MDApp.get_running_app()
        self.console_screen = ConsoleScreen(name='console')
        self.app.root = ScreenManager()
        self.app.root.add_widget(self.console_screen)

    def test_run_code_exito(self):
        self.console_screen.ids.code_input.text = "print('Hola Mundo')"
        self.app.run_code()
        self.assertIn("Hola Mundo", self.console_screen.ids.output_console.text)

    def test_run_code_error(self):
        self.console_screen.ids.code_input.text = "print(1/0)"
        self.app.run_code()
        self.assertIn("Error", self.console_screen.ids.output_console.text)
```

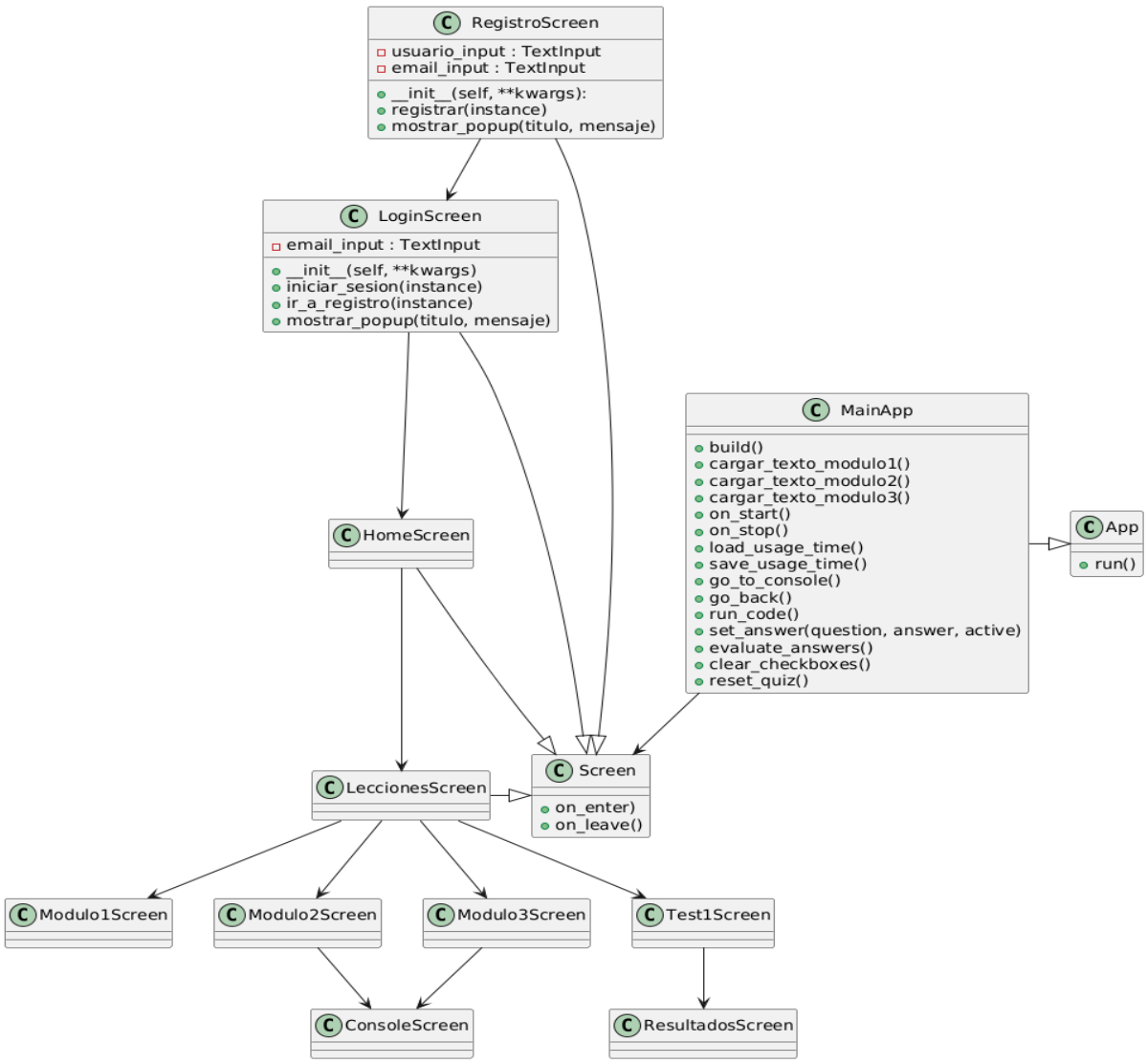
Métodos:

- **test_run_code_exito():** Esta prueba simula la ejecución exitosa de código. Ingresa el código "print('Hola Mundo')" en el campo de entrada de código y llama al método run_code() de la aplicación. Luego, verifica si la salida de la consola (output_console.text) contiene el texto "Hola Mundo".
- **test_run_code_error():** Esta prueba simula la ejecución fallida de código. Ingresa el código "print(1/0)" (división por cero) en el campo de entrada de código y llama al método run_code() de la aplicación. Luego, verifica si la salida de la consola (output_console.text) contiene la palabra "Error", indicando un error en la ejecución.

Método para todas las clases:

- **setUp():** Este método se ejecuta antes de cada prueba unitaria. En todas las clases, este método inicializa la aplicación (app) y crea una instancia de la pantalla que se va a probar (por ejemplo, login_screen o registro_screen). Además, configura un ScreenManager como raíz de la aplicación y agrega la pantalla de prueba a este administrador.

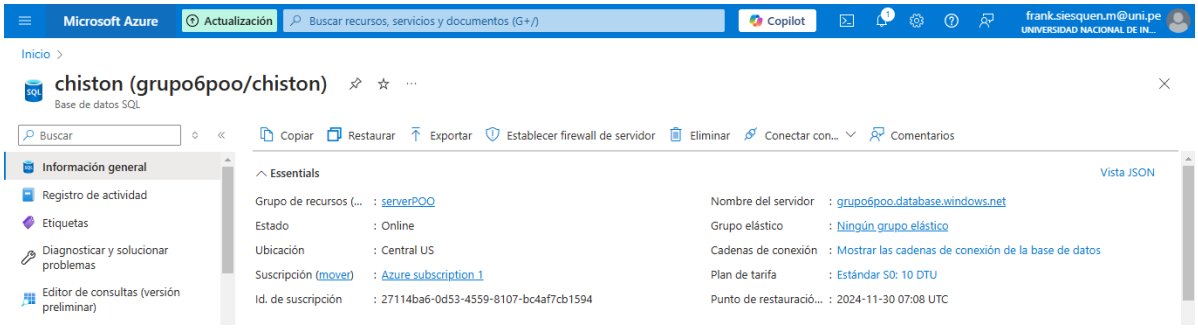
Diagrama UML



Notas adicionales

Conexión a la base de datos

Para el desarrollo de este código, se necesitó la creación de una base de datos que almacene los datos de los usuarios.



- Es una base de datos SQL creada en Microsoft Azure.
- Para conectarla al código se importó la librería **pyodbc**, que nos permite establecer la conexión con la base de datos a través del controlador ODBC.

```
conn_str = (
    "Driver={ODBC Driver 18 for SQL Server};"
    "Server=grupo6poo.database.windows.net;"
    "Database=chiston;"
    "UID=adminchiston;"
    "PWD=Chiston2024;"
)
```

Conclusiones

- ✓ A pesar de ser un proyecto inicial, la app tiene un gran potencial y un gran margen de mejora, como la inclusión de nuevos módulos, integración con plataformas educativas, etc. Este tipo de aplicaciones es esencial en la era digital actual, donde el aprendizaje autónomo y la educación accesible son cada vez más importantes.

- ✓ El desarrollo de screens y la gestión de widgets en Kivy requiere precisión y tiempo, destacando la importancia de conocer a fondo las características de esta herramienta para optimizar el diseño y la funcionalidad. La implementación de pruebas automatizadas resultó crucial para asegurar la calidad del sistema de registro y la experiencia del usuario. La conexión con Azure añade robustez al manejo de datos, consolidando este proyecto como una solución eficiente para el aprendizaje de Python en entornos dinámicos y accesibles desde cualquier lugar.

- ✓ Aunque existen márgenes de mejoras para la aplicación, este proyecto resultó ser satisfactorio para cumplir con el objetivo planteado de facilitar el aprendizaje desde cero sobre los conceptos básicos del lenguaje de programación Python a aquellos usuarios que empiezan a adentrarse a este mundo de la programación.

Referencia Bibliográfica

Python Software Foundation. *unittest – Marco de pruebas unitarias. Documentación de Python 3.13.0*. Recuperado de: https://docs.python.org/3/library/unittest.html#module-unittest

Zepeda, E. (2019, junio 18). *Testeo con tox en Python, tutorial desde cero. Coffee bytes.* Recuperado de: https://coffeebytes.dev/es/testeo-con-tox-en-python-tutorialdesde-cero/

Documentación de KivyMD 1.1.1. Recuperado de: https://kivymd.readthedocs.io/en/1.1.1/index.html

Documentación de Kivy. Recuperado de: https://kivy.org/doc/stable/

Documentación de pyodbc. Recuperado de: https://github.com/mkleehammer/pyodbc/wiki