Instituto Tecnológico de Buenos Aires

Certificación profesional de Python

Implementación de un programa que permita leer datos de una API de finanzas, guardarlos en una base de datos y graficarlos.

Autores

Andía, Francisco Tomás. D.N.I: 40130619

Boullon, Franco, D.N.I: 37878575

Ferraro, Bruno Gastón D.N.I: 43725174

Gozzarino, Nahuel, D.N.I: 39614483

Pedrozo, Sebastián Gabriel D.N.I: 26436178

SEPTIEMBRE, 2022

INDICE DE CONTENIDO

INTRODUCCIÓN.

El siguiente trabajo final integrador tendrá como objetivo principal dominar el lenguaje Python poniendo en práctica todos los conocimientos adquiridos a lo largo del curso.

Se profundizará cada uno de los temas tratados y se desarrollará un programa que permita interactuar y manipular los datos obtenidos desde una API de finanzas.

Gracias a la herramienta SQLite, se generarán diferentes bases de datos, que luego serán manipuladas con el objetivo de obtener diferentes visualizaciones. Los autores tendrán el desafío de transformar una gran cantidad de datos en información de calidad para así ofrecer al usuario una herramienta útil y sencilla.

ORGANIZACIÓN.

El programa estará compuesto por los siguientes módulos:

- core.py: script principal que pondrá el programa en funcionamiento.
- menu.py: la interfaz que mostrará el menú principal por la terminal.
- db.py: encargado de manejar la gestión de los datos.
- manager.py: encargado del envío de la información a la base de datos de acuerdo al ticker y fechas ingresadas por el usuario.
- api.py: encargado de validar los datos ingresados por el usuario y el pedido de requests a la API.
- grafico.py: encargado de graficar la información de un ticker.
- helpers.py: contiene una función para limpiar la terminal.

MÓDULO CORE.

Aquí se ejecutará la creación de la base de datos **tickers** y se llamará al módulo **“menu.py”** a través de la función **loop().**

MÓDULO HELPERS.

En esta parte del programa se incluyó la función **clear().** La misma detecta el sistema operativo utilizado y de acuerdo a ello utiliza la sintaxis para la limpieza de la terminal cada vez que se ingresa a las funciones **loop()** y **loop2()** del módulo “menu.py”.

MÓDULO MENÚ.

En este módulo se le pedirá al usuario que elija entre la opción de “Actualización de datos”, “Visualización de datos” o “Salir” del programa. Esto se realizará a través de una estructura **while True**: dentro de la función **loop().**

```
def loop():
    while True:
        helpers.clear() # LIMPIA LA TERMINAL
        print("============================")
        print("   BIENVENIDO AL GESTOR   ")
        print("============================")
        print("[1] Actualización de datos")
        print("[2] Visualización de datos")
        print("[3] Salir                  ")
        print("============================")
        option = input("> ")
        helpers.clear() # LIMPIA LA TERMINAL
```

```
        if option == '1':
            manager.ticker()  #Se detallará en el módulo "manager".
```

En la opción 1, “Actualización de datos”, se le pedirá al usuario ingresar un ticker, una fecha de inicio y una fecha de fin. Esta funcionalidad del programa será detallada correctamente en el módulo manager. (página 13)

En caso de que el usuario seleccione la opción 2, “Visualización de datos”, se llamará a la función **loop2()** dentro del mismo módulo para ingresar a otro menú con la estructura **while True** donde se le pedirá nuevamente al usuario que elija entre la opción “Resumen”, “Gráfico de Ticker” o “Volver” al menú anterior.

```
def loop2():
    while True:
        helpers.clear() # LIMPIA LA TERMINAL
        print("====================")
        print("[1] Resumen          ")
        print("[2] Gráfico de ticker")
        print("[3] Volver           ")
        print("====================")
```

En la opción 1, “Resumen” se imprimirá por terminal un resumen de los datos guardados en la base de datos.

```
        if option == '1':
            db.mostrar_resumen_db()
```

En la opción 2, “Gráfico de Ticker” se le permitirá al usuario elegir entre dos tipos de gráficos para visualizar los datos guardados de un determinado ticker.

```
        elif option == '2':
            while True:
                helpers.clear() # LIMPIA LA TERMINAL
                print("================================================")
                print("          SELECCIONE TIPO DE GRÁFICO            ")
                print("================================================")
                print("[1] Evolución Lineal de Precios                 ")
                print("[2] Gráfico de Velas (Precio Apertura y Cierre)")
                print("[3] Volver                                      ")
                print("================================================")
```

Una vez que el usuario decida que tipo de grafico quiere realizar, se ejecutarán las siguientes líneas de código, según corresponda. Cabe destacar que las funciones que se muestran a continuación, serán detalladas más adelante en sus respectivos módulos, pero resulta necesario presentárselas al lector.

```
                if option == '1':
                    grafico.graficar_ticker_1()
                elif option == '2':
                    grafico.graficar_ticker_2()
                elif option == '3':
                    print("Volviendo al menú anterior...\n")
                    break
                else:
                    print("Opción incorrecta")
                input("\nPresiona ENTER para continuar...")
        elif option == '3':
            print("Volviendo al menú anterior...\n")
            break
        else:
            print("Opción incorrecta")
        input("\nPresiona ENTER para continuar...")
```

MÓDULO BASE DE DATOS.

Dicho módulo se desarrollará en el archivo “db.py”. En el mismo se realizan las conexiones con la base de datos, la creación de las tablas y una serie de pruebas y funciones relacionadas con las mismas. Estas funciones serán luego llamadas por otros módulos, tal como“manager.py”.

Utilizaremos dos tablas que nos servirán como almacenamiento de datos: tickers.db, y resumen.db.

Para ello, se realizará una conexión con la base de datos y se ejecutan las líneas que le siguen al comando CREATE TABLE para definir el nombre y el formato de las columnas que tendrá nuestras tablas “tickers” y “resumen” respectviamente.

```
conexion = sqlite3.connect("tickers2.db")
cursor = conexion.cursor()
try:
   cursor.execute("""
       CREATE TABLE tickers(
           nombre VARCHAR (10),
           fecha DATE (15),
           volumen INTEGER (10),
           precio_apertura FLOAT (10),
           precio_cierre FLOAT (10),
           precio_mas_alto FLOAT (10),
           precio_mas_bajo FLOAT (10))""")
   cursor.execute("""
       CREATE TABLE resumen(
           ticker VARCHAR (10) PRIMARY KEY,
           fecha_inicio DATE (10),
           fecha_fin DATE (10))""")
except sqlite3.OperationalError:
```

```
   print("Las tablas de tickers y resumen ya existen.")
else:
   print("Las tablas de tickers y resumen se han creado correctamente.")
conexion.close()
```

La tabla **resumen** contará con solo un registro por cada ticker **(ticker se establece como primary key)** registrado en la base, mientras que la tabla **tickers** contará con **tantos registros como períodos diferentes** se hayan consultado.

Las funciones por su parte también realizarán una conexión a la base de datos, y se recorrerá con un cursor la consulta que servirá para leer o modificar la tabla en cuestión.

```
def is_ticker_in_db(valor):
   conexion = sqlite3.connect("tickers.db")
   cursor = conexion.cursor()
```

La función “is_ticker_in_db” verifica si el ticker se encuentra o no en la base de datos. Para esto el código crea una lista de tickers, la cual contendrá un solo valor para cada ticker ingresado en la base, es decir, el ticker de Apple (AAPL) no podrá repetirse dos veces ya que esto será validando en la tabla resumen.

Luego de recorrer la tabla y leer los valores de la variable ticker, el comando cursor.fetchall() traerá todos los registros leídos a la variable “data” en forma de lista. Se recorrerá la lista tickers y se verificará que el valor que la función recibe no esté presente en la lista anteriormente creada.

```
def is_ticker_in_db(valor):
    conexion = sqlite3.connect("tickers.db")
    cursor = conexion.cursor()
    cursor.execute("SELECT ticker FROM resumen")
    data = cursor.fetchall()
    tickers = []
    for ticker in data:
        tickers.append(ticker[0])
    conexion.close()
    if valor in tickers:
        return True
    else: return False
```

A modo explicativo presentamos a continuación la función **tabla_tickers_db:**

Luego de conectar y crear al cursor, el programa ejecutará la línea principal que modificará la tabla tickers.. El comando executemany realiza la acción de insertar en la tabla para cada una de las tuplas valores. Cada tupla contiene

(ticker,fecha_inicio,fecha_fin,precio_apertura,precio_cierre,precio_mas_alto,precio_mas_bajo)

```
def tabla_tickers_db(valores):
    conexion = sqlite3.connect("tickers.db")
    cursor = conexion.cursor()
    cursor.executemany("INSERT INTO tickers VALUES (?,?,?,?,?,?,?)",
valores)
    conexion.commit()
    conexion.close()
```

A modo análogo ocurre el mismo proceso para **tabla_resumen_db** pero en este caso se guardará en la tabla los valores: *ticker, fecha_inicio, fecha_fin,* con el comando *"INSERT INTO…"*

Una vez finalizada la consulta, se realiza un commit() para confirmar los cambios en la base de datos y un close() para cerrar la conexión previamente creada.

```
def tabla_resumen_db(valores):
    conexion = sqlite3.connect("tickers.db")
    cursor = conexion.cursor()
    cursor.execute("INSERT INTO resumen VALUES (?,?,?)", valores)
    conexion.commit()
    conexion.close()
```

La próxima función: **actualizar_resumen_db** modificará los valores de fecha de inicio, fecha de fin para el ticker que se necesite actualizar en la tabla resumen. Para ello se ejecutará en la consulta, luego de conectar a la base de datos, la línea “UPDATE resumen SET…”

```
def actualizar_resumen_db(valores):
    conexion = sqlite3.connect("tickers.db")
    cursor = conexion.cursor()
    cursor.execute("UPDATE resumen SET fecha_inicio= ?, fecha_fin= ?
WHERE ticker= ?", valores)
    conexion.commit()
    conexion.close()
```

La última función de este módulo es **mostrar_resumen_db.**
Dicha función mostrará los tickers que se encuentran guardados en la base de datos, con su fecha de inicio, y fecha final del período.

```
def mostrar_resumen_db():
    conexion = sqlite3.connect("tickers.db")
    cursor = conexion.cursor()
    cursor.execute("SELECT * FROM resumen")
```

```
    tickers = cursor.fetchall()
    print("Los tickers guardados en la base de datos son:")
    for ticker in tickers:
        print(f"{ticker[0]:4} - {ticker[1]} <-> {ticker[2]}")
    conexion.close()
```

MÓDULO MANAGER.

Este módulo se desarrolla en el archivo “manager.py”. El mismo cuenta con una función, llamada **ticker(),** que es llamada por el módulo “menu.py”. Las primeras tareas serán obtener el ticker y las fechas de inicio y fin ingresadas por el usuario a través de dos funciones del módulo “api.py”, **validar_ticker** y **validar_fechas** respectivamente:

```
def ticker():
    ticker = api.validar_ticker()
    fecha_inicio, fecha_fin = api.validar_fechas(ticker)
```

Una vez obtenido el ticker y las fechas de inicio y fin se procede a verificar si el ticker ingresado es nuevo o existente en la base de datos a través de función **is_ticker_in_db** del módulo “db.py”:

```
if not db.is_ticker_in_db(ticker):
```

Si el ticker es nuevo, se procede a pedir los datos a la API por medio de la función **request_api** del módulo “api.py”, pasando como parámetros el ticker, la fecha de inicio y la de fin:

```
datos = api.request_api(ticker,fecha_inicio,fecha_fin)
```

Una vez obtenidos los datos, se utiliza la función **tabla_tickers_db** del módulo “db.py” para guardar los datos en la tabla **tickers**:

```
db.tabla_tickers_db(datos)
```

Con la función **tabla_resumen_db** del módulo “db.py” se guardan en la tabla **resumen** el ticker y las fechas de inicio y fin:

```
db.tabla_resumen_db((ticker, fecha_inicio, fecha_fin))
```

Ahora bien, si el ticker ingresado es existente, lo primero que se realizará es un llamado a la tabla **resumen** de la base de datos para obtener las fechas de inicio y fin cargadas para dicho ticker (fecha_inicio_db y fecha_fin_db respectivamente) y se pasarán junto con las fechas de inicio y fin ingresadas por el usuario, al formato datetime (esto se realiza para poder definir las fechas de inicio y fin para el pedido de datos a la API)

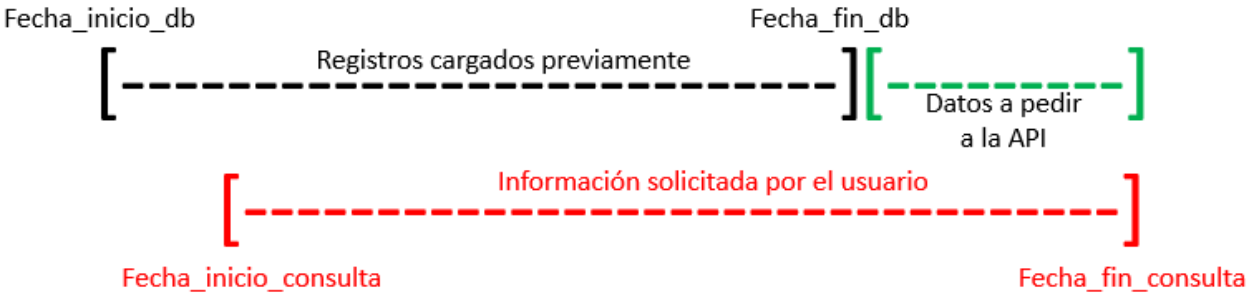
Cabe destacar que las fechas de inicio y fin ingresadas por el usuario se las nombrará de ahora en adelante como fecha_inicio_consulta y fecha_fin_consulta respectivamente.

```
      fecha_inicio_consulta =date.fromisoformat(fecha_inicio)
       fecha_fin_consulta =date.fromisoformat(fecha_fin)

       conexion = sqlite3.connect("tickers.db")
       cursor = conexion.cursor()

      cursor.execute("SELECT fecha_inicio, fecha_fin FROM resumen
WHERE ticker= ?",      (ticker,))

       data = cursor.fetchall()

       fecha_inicio_db = date.fromisoformat(data[0][0])
       fecha_fin_db = date.fromisoformat(data[0][1])

       conexion.close()
```

Dependiendo de las fechas de inicio y fin ya cargadas del ticker y las nuevas fechas de inicio y fin ingresadas por el usuario, con el fin de hacer más eficiente el código, se procederá a pedir a la API sólo los valores de aquellas fechas que no estén cargadas previamente en la base de datos.

A continuación se presentan las 6 situaciones que pueden presentarse en el momento que el usuario solicita actualizar datos de un ticker.

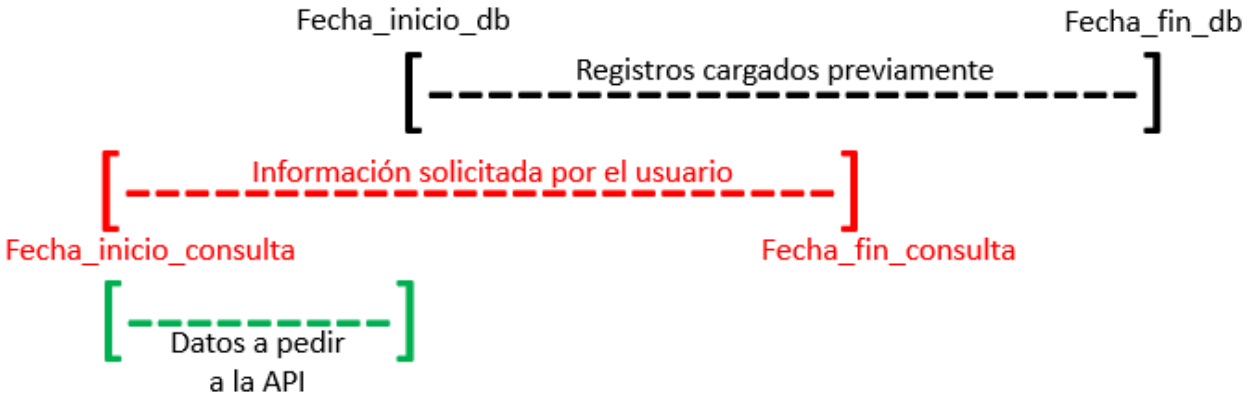
Situación 1:



Si la **fecha_inicio_consulta** es mayor o igual que la **fecha_inicio_db** y menor o igual que la **fecha_fin_db** y a su vez la **fecha_fin_consulta** es mayor que la **fecha_fin_db**, en este caso, sólo se piden los datos entre un día posterior a la **fecha_fin_db** y la **fecha_fin_consulta**, que luego de pasarse a str (para poder hacer el request a la API) pasarán a llamarse **fecha_inicio_request** y **fecha_fin_request** respectivamente; luego se definen las nuevas fechas de inicio y fin para ese ticker, **fecha_nueva_inicio_db** y **fecha_nueva_fin_db** respectivamente, se hace el pedido de datos a la API y se cargan en la tabla **ticker** y se actualiza en la tabla **resumen** por medio de la función **actualizar_resumen_db** del módulo “db.py” las fechas de inicio y fin.

```
if fecha_inicio_db <= fecha_inicio_consulta <= fecha_fin_db and
fecha_fin_consulta > fecha_fin_db:


            fecha_inicio_request = str(fecha_fin_db + timedelta(days=1))
            fecha_fin_request = str(fecha_fin_consulta)


            fecha_nueva_inicio_db = str(fecha_inicio_db)
            fecha_nueva_fin_db = fecha_fin_request


            datos =
api.request_api(ticker,fecha_inicio_request,fecha_fin_request)


            db.tabla_tickers_db(datos)


            db.actualizar_resumen_db((fecha_nueva_inicio_db,fecha_nueva_fin_db
,ticker))
```

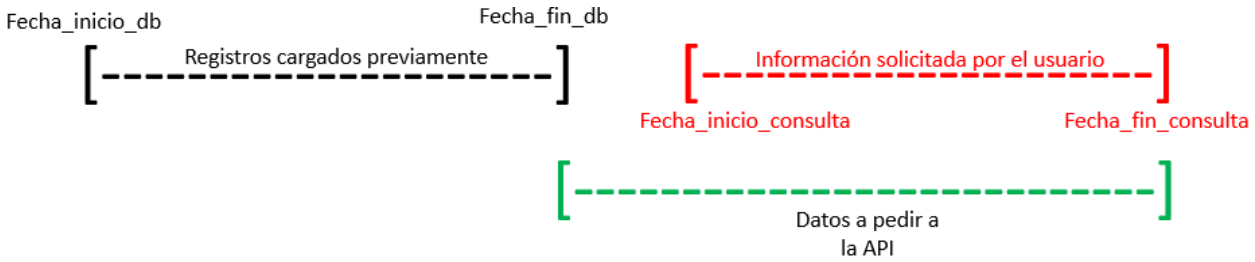
Situación 2:



Si la **fecha_inicio_consulta** es menor que la **fecha_inicio_db** y la **fecha_fin_consulta** es mayor o igual que la **fecha_inicio_db** y menor o igual que la **fecha_fin_db**, en este caso, sólo se piden los datos entre la **fecha_inicio_consulta** y un día anterior a la **fecha_inicio_db** y se procede a definir las fechas para realizar el request a la API, las nuevas fechas de inicio y fin y el pedido de datos y cargas en las tablas como en el caso anterior:

```
elif fecha_inicio_consulta < fecha_inicio_db and fecha_inicio_db <=
fecha_fin_consulta <= fecha_fin_db:


            fecha_inicio_request = str(fecha_inicio_consulta)
            fecha_fin_request = str(fecha_inicio_db - timedelta(days=1))


            fecha_nueva_inicio_db = fecha_inicio_request
            fecha_nueva_fin_db = str(fecha_fin_db)


            datos =
api.request_api(ticker,fecha_inicio_request,fecha_fin_request)


            db.tabla_tickers_db(datos)


            db.actualizar_resumen_db((fecha_nueva_inicio_db,fecha_nueva_fin_db
,ticker))
```

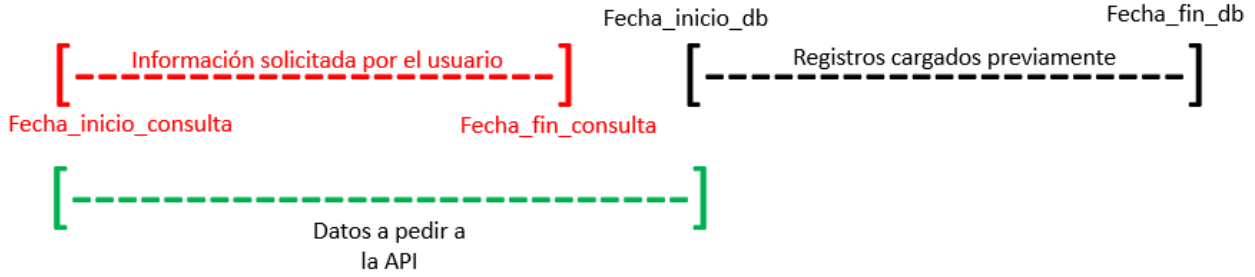
Situación 3:



Si la **fecha_inicio_consulta** es mayor que la **fecha_fin_db**, en este caso, y para no dejar huecos sin información, se piden los datos entre un día posterior a la **fecha_fin_db** y la **fecha_fin_consulta** y se procede a definir las fechas para realizar el request a la API, las nuevas fechas de inicio y fin y el pedido de datos y cargas en las tablas como en los casos anteriores:

```
elif fecha_inicio_consulta > fecha_fin_db:
            fecha_inicio_request = str(fecha_fin_db +
timedelta(days=1))
            fecha_fin_request = str(fecha_fin_consulta)
            fecha_nueva_inicio_db = str(fecha_inicio_db)
            fecha_nueva_fin_db = fecha_fin_request
           datos =
api.request_api(ticker,fecha_inicio_request,fecha_fin_request)
            db.tabla_tickers_db(datos)
            db.actualizar_resumen_db((fecha_nueva_inicio_db,fecha_nueva
_fin_db,ticker))
```

Situación 4:

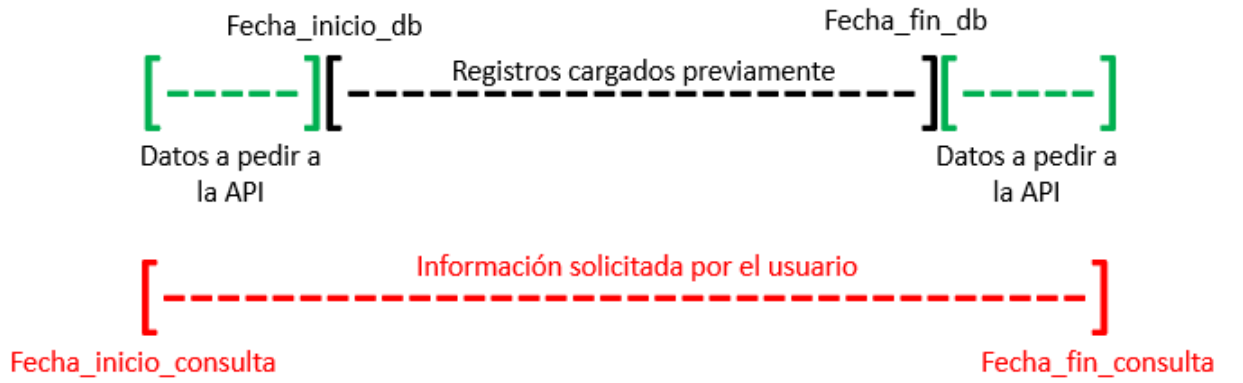


Si la **fecha_fin_consulta** es menor que la **fecha_inicio_db**, en este caso, y para no dejar huecos sin información como en el caso anterior, se piden los datos entre entre la

fecha_inicio_consulta y el dia anterior a la **fecha_inicio_db** y se procede a definir las fechas para realizar el request a la API, las nuevas fechas de inicio y fin y el pedido de datos y cargas en las tablas como en los casos anteriores:

```
elif fecha_fin_consulta < fecha_inicio_db:
            fecha_inicio_request = str(fecha_inicio_consulta)
            fecha_fin_request = str(fecha_inicio_db -
timedelta(days=1))
            fecha_nueva_inicio_db = fecha_inicio_request
            fecha_nueva_fin_db = str(fecha_fin_db)
            datos =
api.request_api(ticker,fecha_inicio_request,fecha_fin_request)
            db.tabla_tickers_db(datos)
            db.actualizar_resumen_db((fecha_nueva_inicio_db,fecha_nuev
a_fin_db,ticker))
```

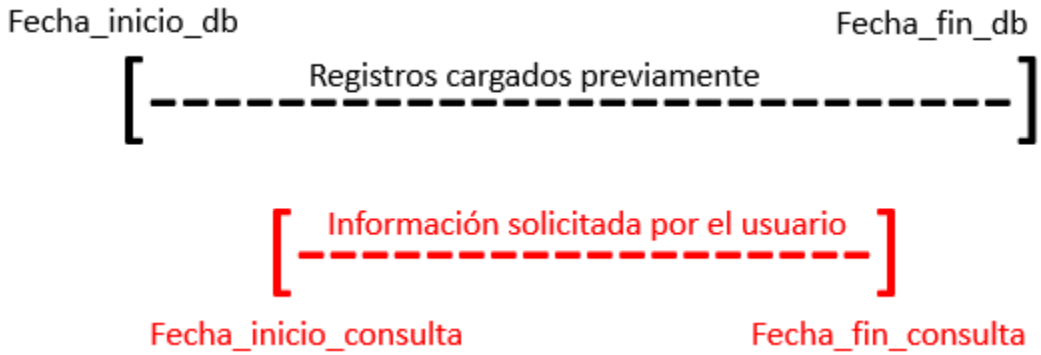
Situación 5:



Si la **fecha_inicio_consulta** es menor que la **fecha_inicio_db** y la **fecha_fin_consulta** es mayor que la **fecha_fin_db**, en este caso se realizan dos pedidos de datos a la API, el primero entre la fecha_inicio_consulta y el día anterior a la **fecha_inicio_db** y el otro entre el día posterior a la **fecha_fin_db** y la **fecha_fin_consulta** y se procede a definir las fechas para realizar los dos requests a la API, las nuevas fechas de inicio y fin y el pedido de datos y cargas en las tablas como en los casos anteriores:

```
elif fecha_inicio_consulta < fecha_inicio_db and fecha_fin_consulta > fecha_fin_db:
            fecha_inicio_request_1 = str(fecha_inicio_consulta)
            fecha_fin_request_1 = str(fecha_inicio_db - timedelta(days=1))
            fecha_inicio_request_2 = str(fecha_fin_db + timedelta(days=1))
            fecha_fin_request_2 = str(fecha_fin_consulta)
            fecha_nueva_inicio_db = fecha_inicio_request_1
            fecha_nueva_fin_db = fecha_fin_request_2
            datos = api.request_api(ticker,fecha_inicio_request_1,fecha_fin_request_1)
            db.tabla_tickers_db(datos)
            datos = api.request_api(ticker,fecha_inicio_request_2,fecha_fin_request_2)
            db.tabla_tickers_db(datos)
            db.actualizar_resumen_db((fecha_nueva_inicio_db,fecha_nueva_fin_db,ticker))
```

Situación 6:



Por último, si la **fecha_inicio_consulta** es mayor o igual que la **fecha_inicio_db** y la **fecha_fin_consulta** es menor o igual que la **fecha_fin_db**, en este caso, no se solicitan datos a la API ni se actualizan las fechas de inicio y fin del ticker ya que los datos de las fechas solicitadas por el usuario se encuentran cargados en las tablas:

```
elif fecha_inicio_consulta >= fecha_inicio_db and fecha_fin_consulta <=
fecha_fin_db:
            print("Datos guardados correctamente")
```

MÓDULO API.

Este módulo, como bien indica el nombre, será el mediador entre el API y el resto de los módulos, entiéndase: Graficar datos, almacenar y modificar la base de datos.

En **Api.py** se definen cuatro funciones, tres de las cuales sirven para validar datos y una última que tiene como propósito realizar el *request* a la API con valores que previamente verificamos que sean válidos.

```
def input_ticker():
```

input_ticker es una función tan simple como esencial para el funcionamiento del programa, ya que, es aquí donde se recibirá el ticker que el usuario quiera almacenar.

Es una función que no requiere ningún argumento. En esta realizamos un *input* acompañado por el *string* “>>> Ingrese el ticker: ” que será guardado como la variable **ticker** a la que luego haremos un *return.*

```
def input_ticker():

    ticker= input(">>> Ingrese el ticker:\n ").upper()

    return ticker
```

Dada la simpleza de esta función sólo se interpuso un problema. La API solo acepta como variable tickers que estén en mayúscula. Entonces, por ejemplo, si el usuario ingresa el ticker AAPL cómo “aapl” sería incorrecto. Ya que el valor ingresado sería en minúscula. Esto fue solucionado utilizando el método **upper** cuya función es tomar cualquier string y devolverla en mayúscula.

```
def validar_ticker():
```

El primer paso de esta función será ejecutar otra función, **input_ticker** para poder obtener el ticker deseado. Una vez hecho esto el programa podrá validarlo antes de enviarlo a la API.

```
def validar_ticker():
    while True:
        ticker= input_ticker()
```

La primera validación que se realiza es verificar que el ticker que ingresó el usuario existió en algún momento. Esto se logrará mediante una de las tantas funciones que tiene la API, particularmente **Previous close.** Esta devuelve ciertos datos de la última vez que el ticker seleccionado cotizo. En nuestro caso solo será necesario utilizar *queryCount* que puede ser igual a 1 o 0. En el caso de que el ticker si haya cotizado en algún momento *queryCount* será igual a 1.Por lo contrario será igual a 0.

Esto estará incorporado en un ***loop while True:*** que llevará a un *break* en el caso de que *queryCount > 0.* En el caso de que esta condición no se cumpla, el programa detectará que el ticker seleccionado nunca insistió, por ende el loop continuará, solicitandole al usuario que ingrese nuevamente el ticker.

La función finaliza devolviendo **ticker.**

```
validacion =
requests.get(f"https://api.polygon.io/v2/aggs/ticker/{ticker}/prev?adjusted=true&
apiKey=_i0PIv435ie2k6p6Oc6rT162DokO4cO6")
        val_data = validacion.json()
        if val_data.get('queryCount') > 0:
            print("Ticker válido")
            break
        print("El ticker ingresado no existe. Intente otra vez")
```

```
def validar_fechas(ticker):
```

Dentro del módulo API es la porción más abundante, tomara solo un argumento; “**ticker**” que, claramente, es el ticker que ingresó previamente el usuario

Esta validación de fechas conendrá dos grandes partes, una para poder validar la fecha de inicio y otra para validar la fecha de fin. Cada una de estas partes estará dentro de un *While True Loop* que a su vez estará dentro de otro gran *While Loop* que abarca tanto la primera como la segunda porción de esta función.

```
def validar_fechas(ticker):
    while True:
            try:
            except ValueError:
               print("Formato de fecha incorrecto")
    while True:
            try:
            except ValueError:
    if fecha_fin_datetime >= fecha_inicio_datetime:
    print("La fecha de inicio no puede ser más reciente que la fecha de
finalización. Intente de nuevo")
```

Como se explicó anteriormente la primera porción estará dentro de un *while loop*. Pero dentro de éste habrá un *try-except* cuya función explicaremos luego.

Se le pedirá al usuario que mediante un input ingrese la fecha de inicio. Acto seguido se abrirá un *if* en el cual se corroborará que el formato de la fecha ingresada sea YYYY-MM-DD, que toda la *string* esté compuesta de dígitos y que tenga un largo de exactamente 10 caracteres.

```
            try:
                fecha_inicio = input(">>> Ingrese fecha de inicio
(formato YYYY-MM-DD):\n")

                if re.match("[0-9]{4}-[0-9]{2}-[0-9]{2}", fecha_inicio)
and len(fecha_inicio) == 10:
```

En el caso de que esto no se cumpla, se continuará el loop y el usuario deberá de ingresar de nuevo la fecha hasta que la misma esté en el formato adecuado.

Una vez ingresada la fecha correctamente, la fecha de inicio se transformará a formato **datetime,** luego se hará un request a la API, utilizando su propia función **Previous close** donde usaremos el ticker dado por el usuario para obtener la primera fecha de cotización en la historia de la acción.

```
fecha_inicio_datetime = datetime.strptime(fecha_inicio, "%Y-%m-%d")
primer_cotizacion                                                  =
requests.get(f'https://api.polygon.io/v3/reference/tickers/{ticker}?api
Key=_i0PIv435ie2k6p6Oc6rT162DokO4cO6')
primer_cotizacion_json = primer_cotizacion.json()
fecha_primera_cotizacion                                           =
primer_cotizacion_json.get('results').get('list_date')
```

La API devuelve la fecha como una string entonces utilizando **strptime** la convertiremos a formato **datetime**. Por último obtendremos la fecha del día actual con la función **datetime.now().**

```
date_prim_cot = datetime.strptime(fecha_primera_cotizacion, "%Y-%m-%d")
fecha_actual = datetime.now()
```

En el caso de que la fecha de primera cotización sea más actual que la fecha de inicio se continuará el *while loop* propio de esta porción de la función y el usuario deberá insertar nuevamente la fecha.

```
if fecha_inicio_datetime < date_prim_cot:

   print(f'La primera fecha de cotización para el ticker {ticker} fue
el {datetime.date(date_prim_cot)}. No podrá visualizar nada antes de
esta fecha!')
```

```
    continue
```

Por otro lado en el caso de que en el caso de que la diferencia entre la fecha inicio ingresada por el usuario y la fecha actual sea mayor a 730 días también se continuará con el *while loop* pidiéndole al usuario que ingrese otra fecha. **Esta condición es necesaria debido a que la suscripción gratuita de la API permite solo hasta 2 años de data histórica desde la fecha actual.**

```
if fecha_inicio_datetime <= (fecha_actual - timedelta(days=730)):

    print(f'La fecha de inicio no pueder ser mayor a dos años de la fecha actual: {datetime.date(fecha_actual)}.')
    continue
```

Ahora es donde nos encontraremos con el *try-except.* ¿para qué sirve? Aunque anteriormente se verificó la fecha, existe aún una posibilidad que no se había evaluado. En el hipotético caso de que el usuario ingrese un mes o día mayor a 12 o 31 respectivamente, esto significa que se respetó el formato **YYYY-MM-DD** pero claramente ni el mes 13 ni el día 32 existen y por ende esta situación culminará en un error. Este *try-except* evitará lo antes mencionado.

En el caso de que se verifique correctamente la fecha de inicio se pasará a la porción en la que se corrobora la fecha de fin.

Como se mencionó, este tramo también estará dentro de un *While loop* seguido de un *try-except* con la misma función que la parte anterior.

Se implementará la misma lógica que en la primera parte, chequeando que la fecha ingresada esté en el formato correcto y posteriormente se lo convertirá al formato **datetime**. Luego, usando **Previous close** obtendremos la fecha en la última cotización

de la acción. Este dato está en formato **Unix epoch timestamps** entonces antes de poder utilizarlo necesitaremos transformarlo al formato **datetime.**

```
fecha_fin = input(">>> Ingrese fecha de fin (formato YYYY-MM-DD):\n")

if re.match("[0-9]{4}-[0-9]{2}-[0-9]{2}", fecha_fin) and len(fecha_fin) == 10:

    fecha_fin_datetime = datetime.strptime(fecha_fin, "%Y-%m-%d")

    ultima_cotizacion = requests.get(f'https://api.polygon.io/v2/aggs/ticker/{ticker}/prev?adjusted=true&apiKey=_i0PIv435ie2k6p6Oc6rT162DokO4cO6').json()
    date_ult_cot = datetime.fromtimestamp(ultima_cotizacion["results"][0]["t"]/1000)
```

Una vez hecho esto el programa corrobora mediante un *if* que la fecha de última cotización de la acción sea más reciente que la fecha de fin ingresada por el usuario, de lo contrario deberá ingresar la fecha.

Por último se corrobora que la fecha de inicio no sea posterior a la fecha de fin y se continuará con la siguiente y última función.

```
def request_api(ticker,fecha_inicio,fecha_fin):
```

La última función llamada request_api requiere de 3 argumentos: **ticker**, **fecha_inicio** y **fecha_fin**. Mediante esta función se realizará el *request* a la API de los datos que se modificaran, guardaran o graficaran posteriormente. Una vez realizado el *request* se guardaran los datos necesarios en la lista “**datos_a_cargar”** que, finalmente se ejecuta un *return* del mismo. Los datos guardados son:

- Fecha (En formato Unix epoch timestamps)
- Precio de cierre
- Precio de apertura
- Volumen
- Precio maximo
- Precio minimo

MÓDULO GRÁFICO.

A la hora de analizar la cotización de una acción hay diversos tipos de gráficos de análisis técnico. Dos de ellos son los gráficos de líneas y los gráficos de velas japonesas.

En el siguiente módulo se desarrollará el archivo **“grafico.py”**. Donde allí se encontrarán las líneas de código relacionadas a las dos visualizaciones previamente mencionadas. Las mismas estarán disponible para el usuario y será éste quien dará la orden para que se ejecute una u otra.

El módulo cuenta con dos funciones totalmente independientes entre sí.

```
def graficar_ticker_1()

def graficar_ticker_2()
```

La primera de ellas, es la encargada de generar un gráfico de lineas. Donde en el eje “x” se representarán fechas. Mientras que en el eje “y” serán visualizados los valores asociados a los diferentes precios de cierre de cada ticker respectivamente.

El precio de cierre fue elegido como una de las variables a graficar debido a que este último se utiliza como referencia cuando los inversionistas evalúan el desempeño a largo plazo, por ejemplo, de una acción. Además puede compararse con los precios de cierres anteriores o con el precio de apertura del mismo día para medir el rendimiento del título valor y es uno de los indicadores que reporta frecuentemente el mercado.

La segunda función está asociada a un gráfico de velas japonesas. Los autores consideran importante la elección de este tipo de gráfico ya que las velas japonesas nos ofrecen información valiosa sobre la cotización de una acción. En otras palabras, le ofrecen al usuario cuatro precios en una misma figura. Además representan el

comportamiento de los inversionistas en ciertos momentos. Por ello, y bajo un estudio de patrones es posible que el usuario pueda determinar próximos comportamientos y escoger mejorar sus inversiones en el mercado.

Los cuatro valores que un gráfico de vela indica para un ticker son:

- Precio al momento de apertura del mercado. “**precio_apertura**” de ahora en adelante.
- Precio máximo alcanzado en el día. “**precio_maximo**” de ahora en adelante.
- Precio mínimo alcanzado en el día. “**precio_minimo**” de ahora en adelante.
- Precio al momento de cierre del mercado. “**precio_cierre**” de ahora en adelante.

Resulta importante destacar que para confeccionar las visualizaciones, el módulo utilizará datos provenientes de otros módulos. Para esto será necesario importar los módulo “db” y “api”. Para cada una de las dos funciones principales y previo a la generación de los gráficos, el módulo realiza los siguientes pasos:

Se define la variable ticker y se le asocia la función **input_ticker()** proveniente del modulo “api”. De esta manera, el programa le solicitará al usuario ingresar el ticker sobre el cual quiere graficar.

```
ticker = api.input_ticker()
```

Luego, por medio de un “if”, se corrobora que exista en la base de datos, información asociada al ticker que ingresó el usuario. Para esto, se utiliza la función **is_ticker_in_db** (ver página 9 - módulo Base de datos). Si el ticker ingresado no se encuentra dentro de la base de datos, se ejecuta la condición “else” y la consola le

devuelve al usuario el siguiente mensaje: “No hay información del ticker {ticker} en la base de datos para graficar”. Caso contrario, si el ticker ingresado existe en nuestra base de datos, se crea una conexión con la base de datos y se genera el data Frame “data” que contendrá la información de la cual se alimentarán las respectivas visualizaciones.

En la conexión a la base datos, un comando execute selecciona todas las columnas de la tabla “tickers” y todas las filas asociadas al ticker que el usuario intenta graficar. Además, en el mismo comando, se ordena que la información se ordene en orden ascendente tomando como referencia la columna “fecha”. Cabe destacar que este ordenamiento de los datos es un paso fundamental para que las visualizaciones se ejecuten de manera clara.

Los nombres de cada columna de nuestra base de datos son almacenados en la variable “column_names”. Esto se logra simplemente recorriendo y guardando en dicha variable los datos asociados a la fila 0 de la tabla ticker.

Por último, todos los datos son almacenados en el data Frame “data”.

```
con = sqlite3.connect('tickers.db')
        cursor = con.cursor()
        cursor.execute("SELECT * FROM tickers WHERE nombre= ? ORDER BY
fecha ASC", (ticker,))
        column_names = [row[0] for row in cursor.description]
        df = cursor.fetchall()
        con.close()
        data = pd.DataFrame(df,columns = column_names)
```

Dependiendo del gráfico que el usuario decida realizar, el programa generará un nuevo **DataFrame** (df) que solo contendrá las filas y columnas que serán utilizadas para generar nuestra visualización.

Para el caso del gráfico de líneas, “df” contendrá sólo 2 columnas. “fecha” y “precio_cierre”.

```
df = pd.DataFrame(data,columns=['fecha','precio_cierre'])
```

Para el grafico de velas japonesas, “df” ahora contendrá 5 columnas. “**fecha**”, “**precio_apertura**”, “**precio_mas_alto**”, “**precio_mas_bajo**” y “**precio_cierre**”.

```
df = pd.DataFrame(data,columns=['fecha', 'precio_apertura',
'precio_mas_alto', 'precio_mas_bajo', 'precio_cierre'])
```

Por último, es momento de utilizar Matplotlib. Una librería de Python especializada en la creación de gráficos en dos dimensiones. Permite crear y personalizar los tipos de gráficos más comunes.

En ambas visualizaciones, una función creará una figura y un conjunto de ejes. Tanto las características de las figuras, como de los ejes serán modificadas aplicando las diferentes funciones que la libreria Matplotlib tiene disponible. Entre ellas podemos destacar la posiblidad de activar las líneas de cuadrícula, autoadjustar la figura al contenido representado como así tambien la posiblidad de configurar el título de cada uno de sus ejes o de la visualización en general

```
fig, ax = plt.subplots()  #Creación de la figura
ax.grid(True) #activación de lineas de cuadrícula
fig.tight_layout() #autoajustar la figura al contenido
ax.set_title("Evolución de Precio de Cierre") #Título del gráfico
```

Para el caso particular del gráfico de velas, el procedimiento de creación y edición de la figura creada es muy similar. Con la diferencia que es necesario importar un módulo matplotlib llamado mplfinance. Dicho módulo habilita a los autores a utilizar la función candlestick_ohlc(), la cual se utiliza para trazar gráficos de velas japonesas.

Esta función en particular ofrece diferentes posibilidades para personalizar el gráfico según los gustos de los autores.

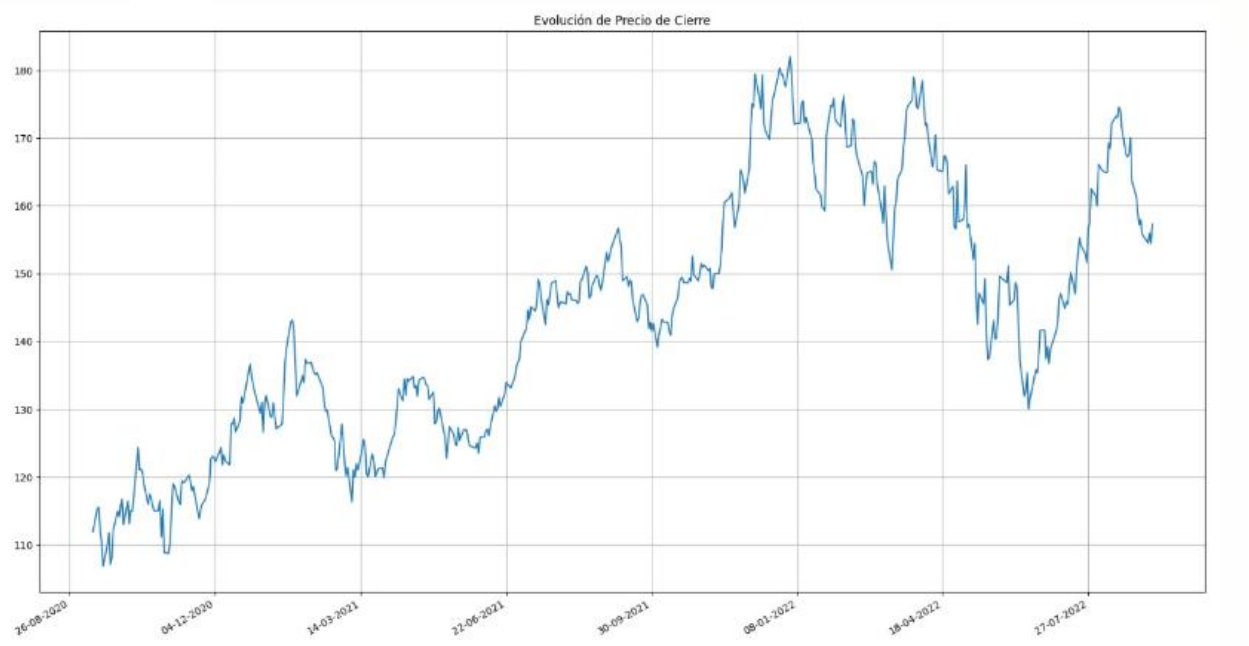
```
candlestick_ohlc(
ax[1],
Df.values,  # valores con los cuales se confecciona el gráfico
width = 1,  # ancho de las velas
colorup = 'green', #color cuando precio_apertura <= precio_cierre
colordown = 'red', #color cuando precio_apertura >= precio_cierre
alpha=0.8) #transparencia de las velas
```

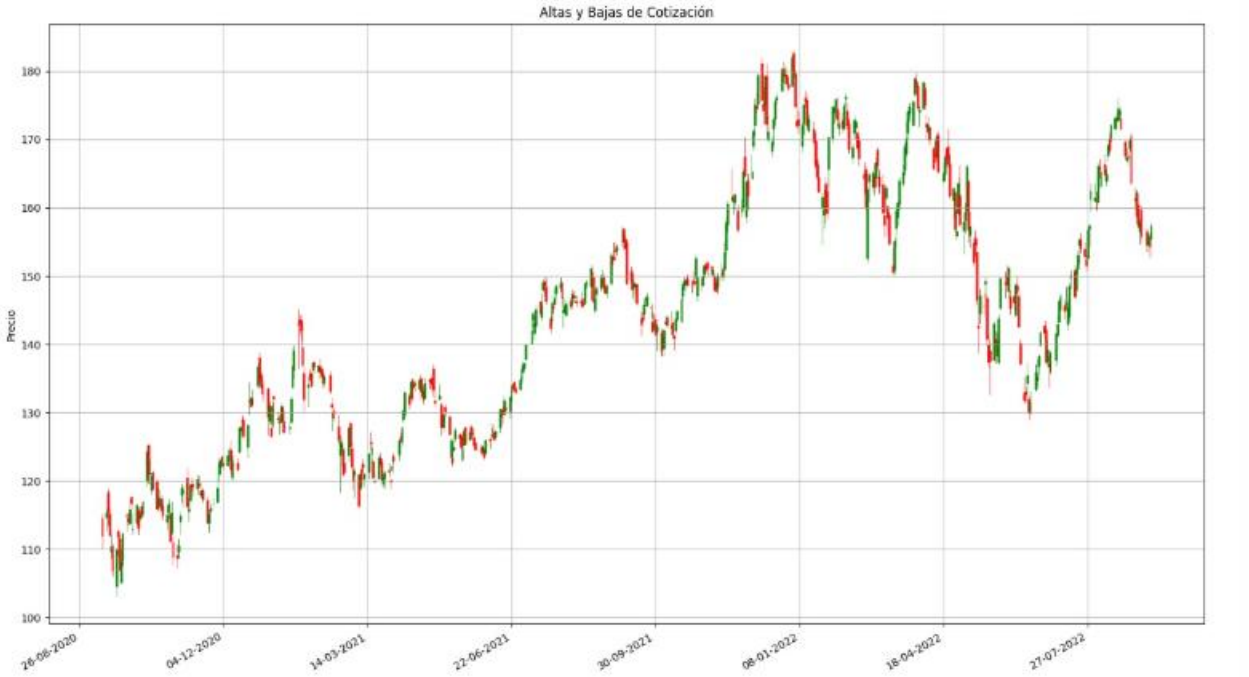
Por último, la función plt.show se utiliza para mostrar la figura previamente definida.

```
plt.show()
```

Cabe destacar que tanto para el gráfico de línea, cómo para el gráfico de velas japonesas, fue necesario utilizar funciones de la librería Pandas para modificar las fechas. De modo tal que las gráficas sean representadas de manera correcta.

Los resultados a los que arriba el usuario por consola son los siguientes:





CONCLUSIONES.

Se cumplieron todos los objetivos propuestos al principio de este trabajo final integrador. Los contenidos teóricos fueron llevados a la practica y se profundizó en cada uno de los diferentes temas desarrollados a lo largo del curso dictado.

El trabajo en grupo fue fundamental para tener una perspectiva clara de lo que implica trabajar en la actualidad en el ámbito de la programación.

Se logró poner en funcionamiento el programa tal cual se solicitaba en la consigna, e incluso se cumplió con una de las funcionalidades extras. La cual mejora sustancialmente la calidad de este trabajo.

El grupo considera que se consolidaron de manera exitosa las bases del lenguaje Python.