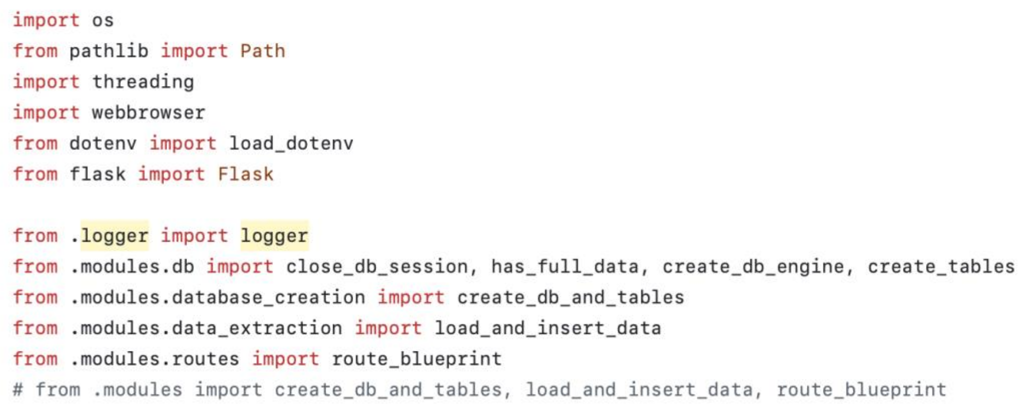
High-level overview of how the script flows - running main.py

There are five main parts to this script. Diagram connecting scripts to the main script shown at the end of doc.

1. The imports
2. Loading configurations
3. Initialises the flask app and registers routes
4. Setting up the database and data ETL
5. Launch web interface and open browser

1. THE IMPORTS



```
import os
from pathlib import Path
import threading
import webbrowser
from dotenv import load_dotenv
from flask import Flask

from .logger import logger
from .modules.db import close_db_session, has_full_data, create_db_engine, create_tables
from .modules.database_creation import create_db_and_tables
from .modules.data_extraction import load_and_insert_data
from .modules.routes import route_blueprint
# from .modules import create_db_and_tables, load_and_insert_data, route_blueprint
```

Standard/third-party imports

os : The standard Python module for interacting with the operating system, such as reading environment variables
pathlib.Path : the more modern Pythonic way to handle file system paths, such as constructing the paths to specified folders or checking if certain files exist.
threading: A standard Python module which runs the code in parallel “threads”/ lets programme do multiple things simultaneously without making the main program wait for them to finish.
webbrowser: Standard module to open URLs in a browser.
dotenv.load_dotenv: Loads environment variables from a .env file.
flask.Flask: The main class to create a Flask web application.

Local import modules

Logger module contains custom logging utility from logger.py
db module contains functions for database connection, table creation, session teardown, and data checks from db.py script
data_extraction module is an ETL (Extract Transform and Load) script that handles functions for parsing input files, enriching with hgvs and clinvar and inserting all data into the DB from the data_extraction.py script
Routes module defines Flask routes blueprint for API/UI.

2. LOADING CONFIGURATIONS

Environment setup: The application reads environment variables (from .env) and sets DB_NAME from the environment or defaults to parkinsons_data.db

```
load_dotenv()  # Load environment variables from .env file

DB_NAME = os.getenv("DB_NAME", "parkinsons_data.db")
```

Setting up the browser helper: Opens the default web browser and navigates to the Flask app's URL, which is set to localhost on port 5000.

```
# Opens the URL when program is run
def open_browser():
    '''The specified URL (whiich is the default created by the command to create the interface)'''
    webbrowser.open_new("http://127.0.0.1:5000/")
```

Initial logging: First thing the programme does is log a message indicating that the Parkinsons Annotator application is starting.

```
def main():
    logger.info("Starting Parkinsons Annotator Application")
```

3. INITIALISES THE FLASK APP AND REGISTERS ROUTES

```
    # Set template_folder to the templates directory within the modules folder, so Flask can find the HTML files
    app = Flask(__name__, template_folder=str(Path(__file__).parent / "templates"))
    app.config['DB_NAME'] = DB_NAME
    app.teardown_appcontext(close_db_session)  # Ensure DB session is closed after each request
    app.register_blueprint(route_blueprint)
```

This section basically creates the Flask app instance and sets up how it's meant to run.

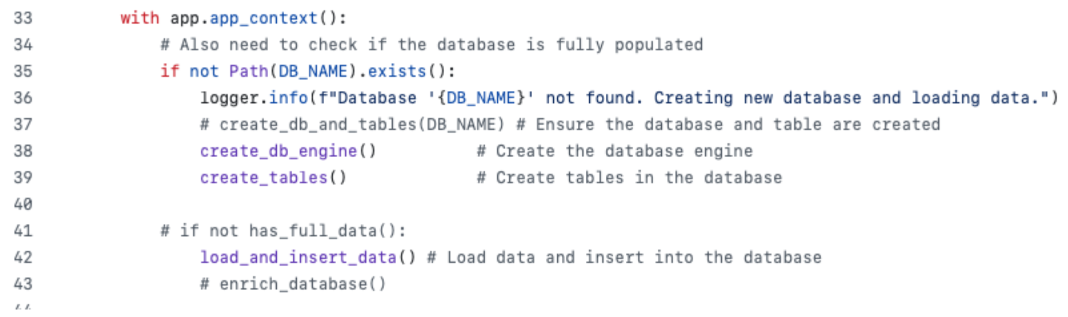
Sets the **template folder** to the templates directory within the modules folder so that Flask can find the HTML files.

It **configures** the database name (DB_NAME) in the Flask app's configuration.

It registers the close_db_session() function within the Flask app **teardown_appcontext** function to be called after each HTTP request, ensuring that any open database sessions are properly closed. This is important to prevent Python from keeping excess objects in memory, stale sessions (holding onto old request data), or connection issues (still holding onto the connections from the old sessions) when handling multiple requests.

Then it registers all routes defined in **route_blueprint** (API endpoints or web pages) with the Flask app. Blueprint routes handle the API/UI endpoints. This is like a mini flask template which allows you to initialise the flask app (without it breaking due to asking for routes) and lets you orchestrate it, which means you can organise your code into separate components instead of putting all of routes.py routes into the main script.

4. SETTING UP THE DATABASE AND DATA ETL



```
with app.app_context():
    # Also need to check if the database is fully populated
    if not Path(DB_NAME).exists():
        logger.info(f"Database '{DB_NAME}' not found. Creating new database and loading data.")
        # create_db_and_tables(DB_NAME) # Ensure the database and table are created
        create_db_engine()          # Create the database engine
        create_tables()             # Create tables in the database

    # if not has_full_data():
        load_and_insert_data() # Load data and insert into the database
        # enrich_database()
```

Application context: Required for operations that depend on the Flask app's context (e.g., database operations).

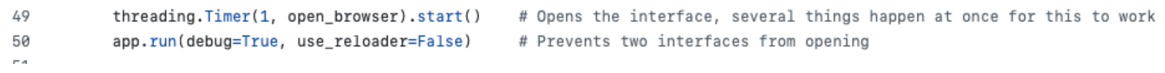
Database check: If the database file does not exist, it creates the database engine and tables, using the db.py script. The db.py script houses all the functions related to database and table generation. They all belong in one script, which holds all functions orchestrating the DB.

create_db_engine: This creates a session using an SQLAlchemy engine connected to the SQLite database. Creating this session factory will allow database interactions to occur. It also enables foreign keys support in SQLite, SQLite does not enforce foreign keys by default and will now let them be controlled by SQLAlchemy. It sets global variables for engine which is SQLAlchemy engine object and session object for database transactions.

create_table: This creates all tables in the database if they do not already exist. It checks if engine exists; if not, it calls the create_db_engine() function.
Once it has set up a session allowing for database manipulation, it uses the SQLAlchemy **Base metadata** to create tables. Base is the declarative base defining all ORM (Object relational mapping) models, this is the template for the database schema - all tables and how they are linked together. Base template is defined in models.py for Variant table, Patient table and Connector table.

Data loading: Calls load_and_insert_data() to parse input files and populate dataframes with initial data, enrich with HGVS and clinvar data, and then insert these into the database.

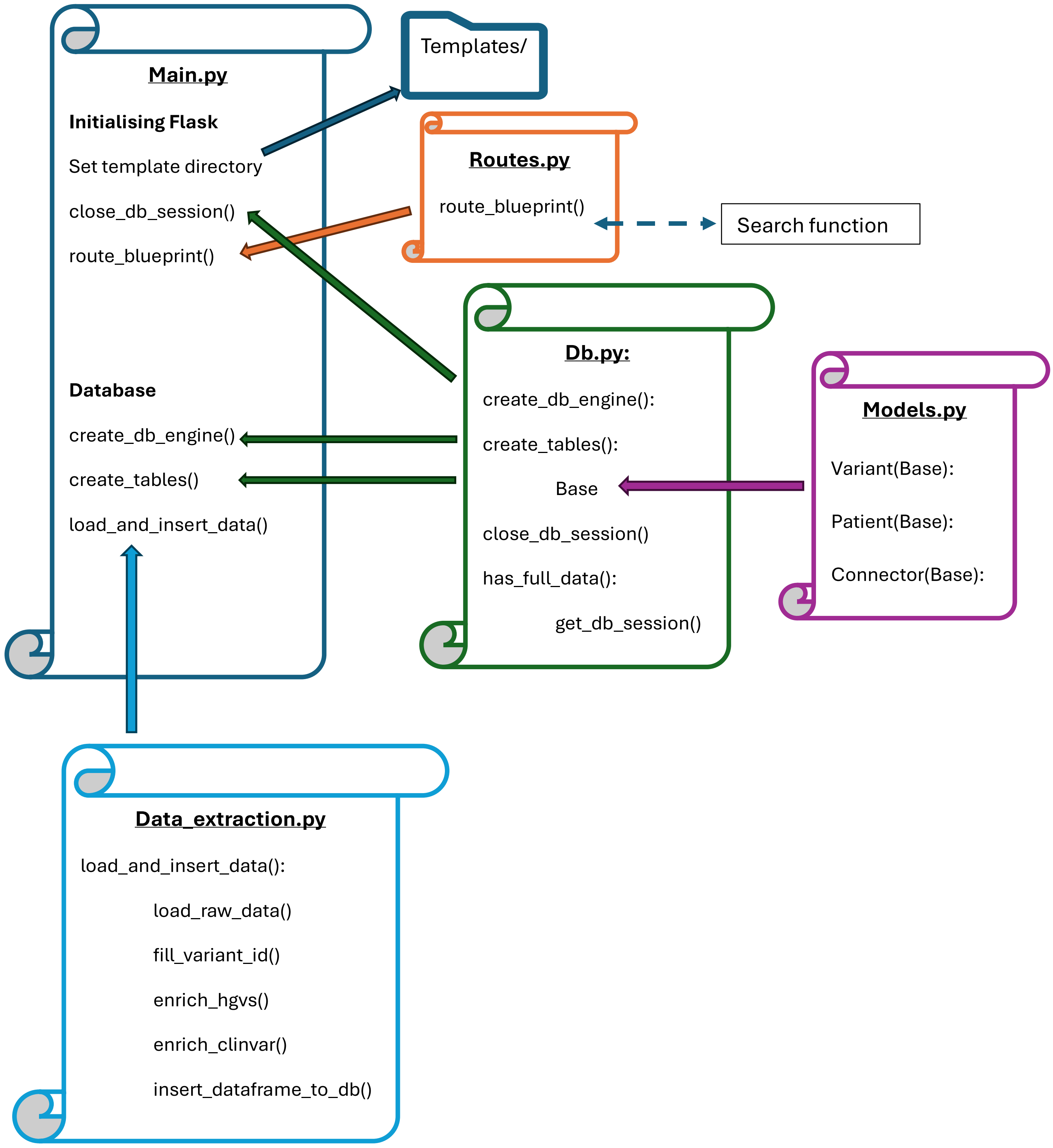
5. LAUNCH WEB INTERFACE AND OPEN BROWSER



```
threading.Timer(1, open_browser).start()    # Opens the interface, several things happen at once for this to work
app.run(debug=True, use_reloader=False)     # Prevents two interfaces from opening
```

This creates a **timer thread** that waits 1 second, then calls open_browser(). This lets the server start first, then opens the browser, as opening the browser before the server is ready, can lead to a connection error.

app.run() then starts the **Flask web server** so the application can handle HTTP requests.


Main.py
Initialising Flask
Set template directory
close_db_session()
route_blueprint()
Database
create_db_engine()
create_tables()
load_and_insert_data()
Templates/
Routes.py
route_blueprint()
Search function
Db.py:
create_db_engine():
create_tables():
Base
close_db_session()
has_full_data():
get_db_session()
Models.py
Variant(Base):
Patient(Base):
Connector(Base):
Data_extraction.py
load_and_insert_data():
load_raw_data()
fill_variant_id()
enrich_hgvs()
enrich_clinvar()
insert_dataframe_to_db()