Code description:

- Main: Main driver that manages command-line arguments, allowing selection of input file, output file, normalization level (1NF, 2NF, 3NF, BCNF, 4NF, 5NF), and target (SQL for SQL output, otherwise textual schema output)
    - Verifies that normal form is in the above list
    - Calls process_input
        - Process_input: parses the input in the specified text file, processing into a series of relation objects
        - Repeatedly calls extract_relation on each block of text
            - Extract_relation: Converts a block of text (string list) with fields like Primary key, Tuple, etc. to a corresponding relation object
            - Verifies that no invalid fields are present
                - Valid fields: Attributes, Primary key, Candidate keys, Foreign key, Functional dependency, Multivalued dependency, Multivalued attributes, Data types, Tuple
                - Fields such as Tuple and Functional dependency are repeatable – there can be multiple lines for the same field
                - Verifies that no non-repeatable fields are repeated
                - Verifies that each field is in a readable format
    - Once input file is processed into relation objects, call first_nf, second_nf, …, converting the relations to relations of the desired normalization level
        - First_nf: Decomposes each multivalued attribute into a separate relation with a copy of the primary key attributes, then splits each tuple in the decomposed relation into one for each value in the multivalued attribute, removing the multivalued attribute from the original relation
            - Also uses tuple values to check if any multivalued dependencies arose from the decomposition (extra credit)
        - Second_nf: Identifies and decomposes partial functional dependencies into separate relations, removing the dependent attributes from the original relation
        - Third_nf: Identifies and decomposes transitive functional dependencies into separate relations by checking that each functional dependency X -> Y has a superkey X or prime attribute Y.
        - Bcnf: Removes any remaining non-trivial functional dependencies and decomposes them into separate relations by checking that each functional dependency X -> Y has a superkey X.
        - Fourth_nf: Removes any multivalued dependencies originally specified or identified after first_nf, decomposing the dependent attributes into separate relations.
        - Fifth_nf: Uses tuple values to identify any join dependencies where a relation could be decomposed and still re-created by a natural join, decomposing into the most optimal configuration.
        - Where applicable, each normal form calls fix_foreign_key_references, which verifies that foreign keys correctly refer to relations that still exist and don't

themselves have that attribute as one of their foreign keys, making any necessary modifications if any foreign keys do not follow this.

- Writes output SQL or textual schema to output file
  - Uses return values of each relation's to_sql and __str__ functions, respectively.

Challenges:

- It was found to be very difficult to correctly track each foreign key during each decomposition step – a decomposed relation needs to have foreign keys referring to repeated attributes in the original relation, but these foreign keys become invalid in cases where the original relation is removed, among other issues. This resulted in the creation of the fix_foreign_key_references function, which analyzes each relation and checks if their foreign keys are valid, making any modifications necessary. This ended up working, producing correct output without needing to fully track foreign keys between each composition step.
- It is almost impossible to correctly replicate the names of each decomposed relation (decompose CoffeeShopData to CoffeeShopDrinksOrderData and CoffeeShopOrderSummaryData), but combining the names of certain decomposed attributes with the original relation name, adjusting Data and ID suffixes where necessary, resulted in most normalized relation names matching those in the output specification.
- Detecting multivalued dependencies ended up being simple, requiring 4 tuples in the following format:

```
[other PK values match], a, b
[other PK values match], c, d
[other PK values match], a, d
[other PK values match], c, b
```

- Detecting join dependencies was more difficult, but was implemented by trying to decompose into each combination of 2 relations with 2+ attributes each, with 1 attribute shared between them, then choosing the join dependency that minimizes the number * size of tuples. A join dependency is detected if a relation can be decomposed then natural-joined to result in the same data, so almost any relation with only 1 provided tuple will be considered as having a join dependency – this may result in false-positives, as the same relation with more tuples may not still have a join dependency.