CST8333 Programming Language Research Project

Practical Project Part 4—Project Review 2

By

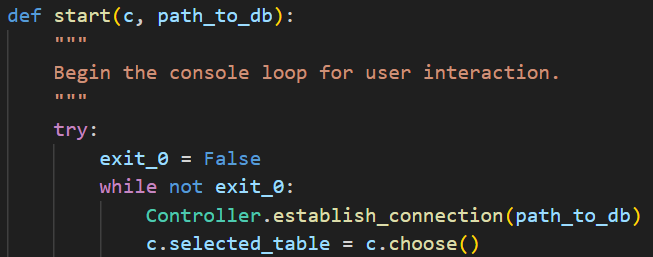
### Ahmad Al-Jabbouri

### 041068196

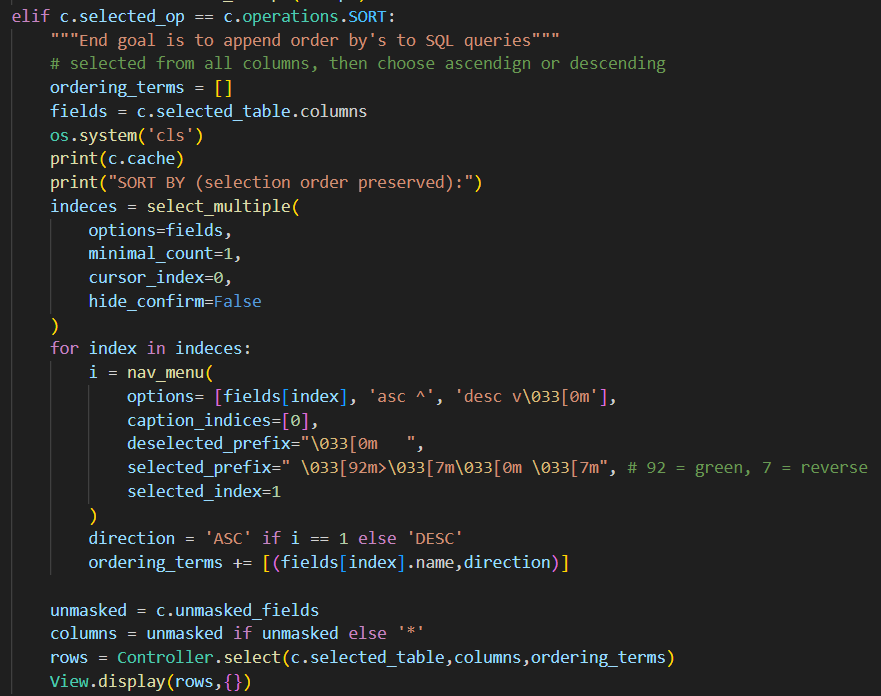
2024-04-07

# Evidence of Learning

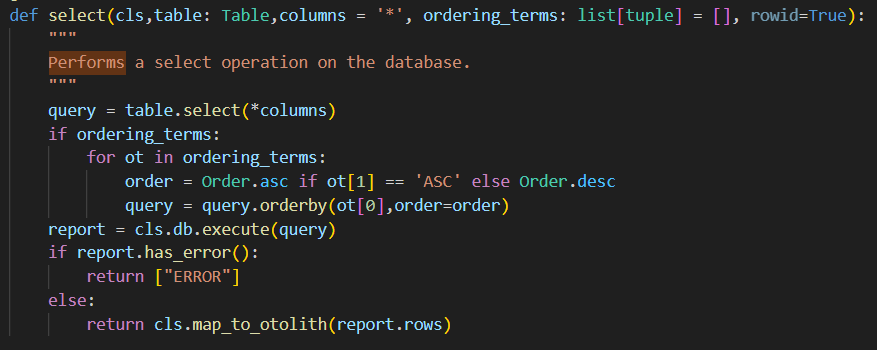
**I choose to implement a sorting feature in my program.**



This code snippet illustrates the starting point of the console loop in the View class. It begins by presenting the user with a menu to select a dataset and establishes a connection to the associated database. Should the user opt to exit, the loop concludes. Otherwise, it advances to displaying the data of the selected table and prompts the user to choose an operation from the second menu.



This code snippet illustrates the implementation of the sorting feature within the View class. It enables the user to select columns for sorting and specify the sort order (ascending or descending). Notably, the order in which the columns are selected corresponds to the priority sequence in which columns are sorted. Subsequently, the selected columns and sort order are utilized to execute a SQL query, with ordering terms appended accordingly. Finally, the sorted results are presented to the user for viewing.



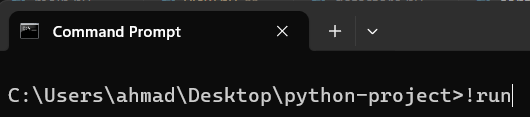
This code snippet showcases the execution of SQL queries with sorting functionality in the Controller class. It constructs a SELECT query based on the specified columns and ordering terms, executes the query using the database connection, and returns the result rows.

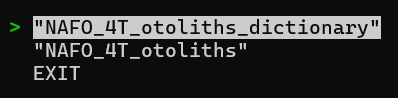
The chosen feature enhances the application by allowing users to sort records based on multiple columns simultaneously, with the option to specify the sort order (ascending or descending). The implementation involves integrating sorting functionality into the existing console interface, enabling users to interactively select columns for sorting and choose the desired order.

The core functionality is implemented within the View and Controller classes. In the View class, the SORT operation prompts the user to select columns and specify the sort order using an interactive menu. Notably, the order in which the user selects the columns determines the priority of sorting. For example, if the user selects "Column B" first and "Column A" second, the records will be sorted primarily by "Column B" and then by "Column A."

Once the user makes their selections, the sorting terms are passed to the Controller class, where a SQL query is constructed with ordering terms appended. The query is then executed, and the sorted results are displayed back to the user.

# Program Demonstration via Screen Shots

Figure 1: Launching program using included bat script (Windows only)

Figure 2: First menu: User selects dataset

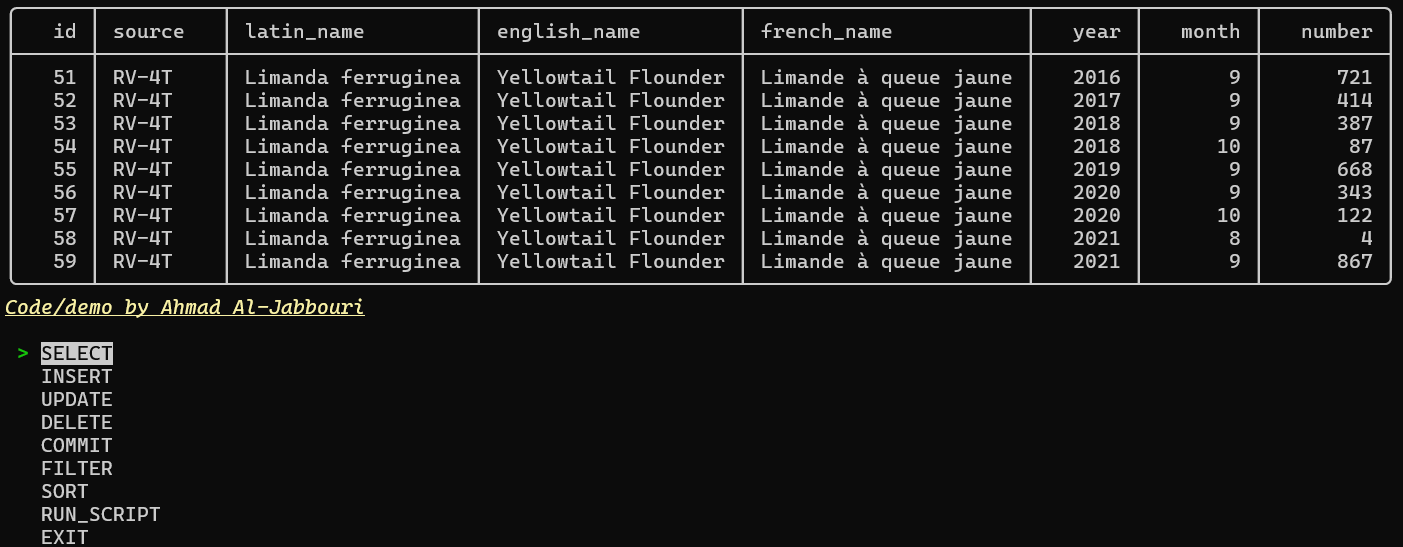
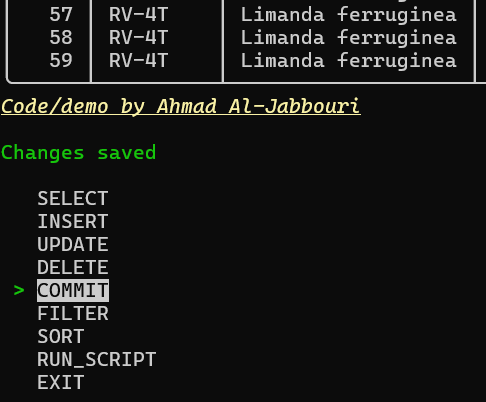
Figure 3: Second menu: Table is loaded from dataset and user is presented with second menu

Figure 4: Insert demonstration

Figure 5: Update demonstration

Figure 6: Delete demonstration

Figure 7: No change is persisted unless COMMIT option is selected

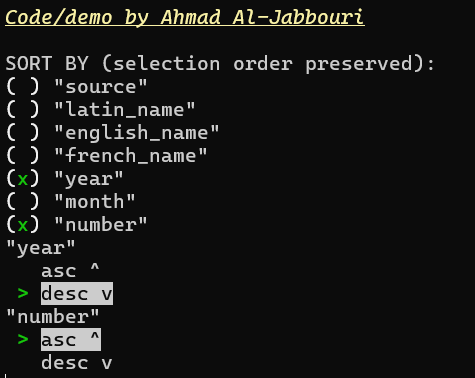
Figure 8: [New feature] Demonstration of sorting. User selects columns to sort by then specifies order of data within column.

Figure 9: Result of sorting. Table is now sorted by year from greater to lesser, and then by number from fewer to greater

# Source Code Commenting Example

|  |
| --- |
| main.py |
| # Code by Ahmad Al-Jabbouri  from pathlib import Path  from logging import info  from utils.src.config import read\_ini  from utils.src.signature import sign  from presentation.view import View  from business.controller import Controller  def main():      """      Main function to start the program.      """      # get current path      src = Path(\_\_file\_\_).parent      info(f"Operating from: {src}")        # load configs      \_ini = read\_ini(src/'config.ini')      # call consoleview      View.start(src/\_ini['Datasources']['database']) # loop      info("Program closed")      # signature      print(sign())  if \_\_name\_\_ == "\_\_main\_\_":      main() |

|  |
| --- |
| view.py |
| import os, sys  from blessed import Terminal  from enum import Enum  from logging import info, exception  from business.controller import Controller  from cutie import select as nav\_menu, select\_multiple  from pypika import Table  from utils.src.signature import sign  from tabulate import tabulate  class View:      """      Class representing the user interface view for interacting with the application.      """      class operations(Enum):          """          Enum class defining different operations available in the view.          """          SELECT = 0          INSERT = 1          UPDATE = 2          DELETE = 3          COMMIT = 4          FILTER = 7          SORT = 8          RUN\_SCRIPT = 5          EXIT = 6            def \_\_str\_\_(self):              """              Returns only the name without the prefix.              """              return self.name      selected\_op = None      selected\_table = None      selected\_columns: list = None      fields: list = []      unmasked\_fields = []      cache: str = ''      display\_settings: dict = {}        @classmethod      def start(c, path\_to\_db):          """          Begin the console loop for user interaction.          """          try:              exit\_0 = False              while not exit\_0:                  Controller.establish\_connection(path\_to\_db)                  c.selected\_table = c.choose()                  if c.selected\_table == 'EXIT':                      exit\_0 = True                      exit\_1 = True                  else:                      exit\_0 = False                      exit\_1 = False                      c.cache = c.get\_table\_data(c.selected\_table,View.operations.SELECT)                  while not exit\_1:                      View.display(c.cache,{})                      c.selected\_op = c.choose\_operation()                      if c.selected\_op == c.operations.EXIT:                          exit\_1 = True                      elif c.selected\_op == c.operations.COMMIT:                          s = Controller.process(c.selected\_op)                          os.system('cls')                          print(c.cache)                          print(s)                      elif c.selected\_op == c.operations.RUN\_SCRIPT:                          os.system('cls')                          print("PASTE THE SQL SCRIPT HERE:")                          print("To end recording, do Ctrl+Z on Windows, Ctrl+D on Linux")                          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")                          script = sys.stdin.readlines()                          Controller.execute\_script(script)                      elif c.selected\_op == c.operations.SORT:                          """End goal is to append order by's to SQL queries"""                          # selected from all columns, then choose ascendign or descending                          ordering\_terms = []                          fields = c.selected\_table.columns                          os.system('cls')                          print(c.cache)                          print("SORT BY (selection order preserved):")                          indeces = select\_multiple(                              options=fields,                              minimal\_count=1,                              cursor\_index=0,                              hide\_confirm=False                          )                          for index in indeces:                              i = nav\_menu(                                  options= [fields[index], 'asc ^', 'desc v\033[0m'],                                  caption\_indices=[0],                                  deselected\_prefix="\033[0m   ",                                  selected\_prefix=" \033[92m>\033[7m\033[0m \033[7m", # 92 = green, 7 = reverse                                  selected\_index=1                              )                              direction = 'ASC' if i == 1 else 'DESC'                              ordering\_terms += [(fields[index].name,direction)]                          unmasked = c.unmasked\_fields                          columns = unmasked if unmasked else '\*'                          rows = Controller.select(c.selected\_table,columns,ordering\_terms)                          View.display(rows,{})                      else: # SELECT and DML here                          data = c.collect\_data(op=c.selected\_op,table=c.selected\_table)                          otoliths = Controller.process(op=c.selected\_op,table=c.selected\_table,data=data)                          c.display(otoliths,c.display\_settings)          except Exception as e:              exception("What happened?")      @classmethod      def get\_table\_data(c,table,op):          """"""          data = c.collect\_data(op=op,table=table)          return Controller.process(op=op,table=c.selected\_table,data=data)        def print\_table(list\_: list):          """          Print the given list in a tabular format.          """          View.display(list\_)        def choose():          """          Choose a table from the available tables.          """          options =  Controller.get\_tables() + ['EXIT']          os.system('cls')          chosen = nav\_menu(                  options + ['\033[0m'],                  caption\_indices=[len(options)],                  deselected\_prefix="\033[0m   ",                  selected\_prefix=" \033[92m>\033[7m\033[0m \033[7m", # 92 = green, 7 = reverse                  selected\_index=0              )          return options[chosen]      def choose\_operation():          """          Choose an operation from the available operations.          """          ops = list(View.operations)          x,y = View.get\_coordinates()          chosen = nav\_menu(                  ops + ['\033[0m'],                  caption\_indices=[len(ops)],                  deselected\_prefix="\033[0m   ",                  selected\_prefix=" \033[92m>\033[7m\033[0m \033[7m", # 92 = green, 7 = reverse                  selected\_index=0              )          x,y = View.get\_coordinates()          return ops[chosen]      def collect\_data(table: Table,op: operations):          """          Collect user input to send to controller based on the selected operation.          """          match op:              case View.operations.FILTER:                  """Pick columns"""                  columns = table.columns                  print("USE ARROWS ^v TO SELECT COLUMNS THEN ENTER TO CONFIRM:\n")                  ints = select\_multiple(                      options=columns,                      ticked\_indices=list(range(len(columns))),                      minimal\_count=1,                      cursor\_index=len(columns),                      hide\_confirm=False                  )                  data = [columns[i] for i in ints]                  View.selected\_columns = data              case View.operations.INSERT:                  """Input for each field"""                  data = []                  for field in table.columns:                      data += [input(str(field) + ": ")]              case View.operations.UPDATE:                  """Need a target, value, and id"""                  data = {}                  data['id'] = input("Which row? (id): ")                  print("Which column needs update? : ")                  target\_index = nav\_menu(                      table.columns + ['\033[0m'],                      caption\_indices=[len(table.columns)],                      deselected\_prefix="\033[0m   ",                      selected\_prefix=" \033[92m>\033[7m\033[0m \033[7m", # 92 = green, 7 = reverse                      selected\_index=0                  )                  data['target'] = table.columns[target\_index]                  data['value'] = input("New value: ")                  return data              case View.operations.DELETE:                  """Need only id"""                  data = input("Input ROWID to delete: ")              case View.operations.COMMIT:                  data = None              case View.operations.SELECT:                  """Pick columns"""                  data = table.columns          return data        def chunks(lst, n):          """          Split a list into chunks of size n.          """          for i in range(0, len(lst), n):              yield lst[i:i + n]        def display(data: list[object], settings): #TODO          """          Display the data in a tabular format.          """          if isinstance(data,str):              # print(data)              pass          else:              headers = data[0].as\_keys()              os.system('cls')              View.cache = '' # for quick reprint              for group in View.chunks(data, 10):                  View.cache += View.to\_table(group,headers=headers)                  View.cache += sign()              print(View.cache)      def to\_table(data:list[object], headers='firstrow') -> str:          """          Convert the data into a tabular format.          """          matrix = []          for obj in data:              row = []              for val in obj.as\_values():                  row += [val]              matrix += [row]          T = tabulate(                  headers=headers,                  tabular\_data=matrix,                  tablefmt='rounded\_outline'              )          View.selected = T          return T        def get\_coordinates():          return Terminal().get\_location()      @classmethod      def \_\_str\_\_(cls):          """          String representation of the class.          """          return f"{cls}" |

|  |
| --- |
| controller.py |
| from business.models.otolith import Otolith  from persistence.datastore import DataStore  from pypika import Query, Table, Field, Order  class Controller:      """      Class responsible for controlling interactions between the view and the data.      """        db: DataStore = None        @classmethod      def establish\_connection(cls, path\_to\_db) -> None:          """          Establishes a connection to the database.          """          cls.db = DataStore(path\_to\_db)        @classmethod      def get\_tables(cls):          """          Retrieves all tables from the database.          """          return cls.db.get\_tables()        @classmethod      def process(cls,op,table=None,data=None):          """          Processes the operation based on the given parameters.          """          match op.value:              case 0 | 7:                  return cls.select(table,data)              case 1:                  return cls.insert(table,data)              case 2:                  return cls.update(table,data)              case 3:                  return cls.delete(table,data)              case 4:                  return cls.db.commit()        @classmethod      def execute\_script(cls, script):          """          Executes the given script in the database.          """          return cls.db.execute\_script("\n".join(script))        @classmethod      def select(cls,table: Table,columns = '\*', ordering\_terms: list[tuple] = [], rowid=True):          """          Performs a select operation on the database.          """          query = table.select(\*columns)          if ordering\_terms:              for ot in ordering\_terms:                  order = Order.asc if ot[1] == 'ASC' else Order.desc                  query = query.orderby(ot[0],order=order)          report = cls.db.execute(query)          if report.has\_error():              return ["ERROR"]          else:              return cls.map\_to\_otolith(report.rows)        @classmethod      def insert(cls, table: Table, values: list[object]):          """          Performs an insert operation on the database.          """          report = cls.db.execute(              table.insert(\*values)          )          if report.has\_error():              return ["ERROR"]          else:              return cls.map\_to\_otolith(report.rows)      @classmethod      def update(cls, table, data):          """          Performs an update operation on the database.          """          report = cls.db.execute(              table.update()              .set(data['target'], data['value'])              .where(Field('rowid') == data['id'])          )          if report.has\_error():              return ["ERROR"]          else:              return cls.map\_to\_otolith(report.rows)      @classmethod      def delete(cls, table: Table, rowid: int):          """          Performs a delete operation on the database.          """          report = cls.db.execute(              Query().from\_(table).delete().where(Field('rowid') == rowid)          )          if report.has\_error():              return ["ERROR"]          else:              return cls.map\_to\_otolith(report.rows)        @classmethod      def map\_to\_otolith(cls,rows):          """          Maps rows to Otolith objects.          """          return [Otolith(\*\*row) for row in rows] |

|  |
| --- |
| datastore.py |
| # Code by Ahmad Al-Jabbouri  import sqlite3  from pypika import Query, Table, Field  from logging import info, exception  from pathlib import PureWindowsPath as Path  class DataStore():      """      A class representing a data store. Use pyPika to build and pass Query to execute!      """      persistence = Path(\_\_file\_\_).parent      def \_\_init\_\_(self, db\_path: Path = None):          """          Initializes the DataStore class.          """          self.connection: sqlite3.Connection = None          if db\_path:              self.connect(db\_path)      def connect(self,db\_path: Path) -> None:          """          Connects to the specified database.          """          info(f"Connecting to {db\_path}...")          try:              self.connection = sqlite3.connect(                  database=db\_path,                  detect\_types=sqlite3.PARSE\_DECLTYPES  # To use in conjunction with converters and adapters              )              self.connection.row\_factory = sqlite3.Row # .execute() will now return Rows instead of tuples. Rows work similar to dict              for value in self.connection.execute("SELECT sqlite\_version()").fetchone():                  info(f"Successfully connected to SQLite version {value}")          except Exception as e:              exception(f"Failed connection: {e}")        def close(self) -> str:          """          Closes the database connection.          """          self.connection.close()          return "Closed"        def commit(self) -> str:          """          Commits changes to the database.          """          try:              self.connection.commit()              return "\033[92mChanges saved\n\033[0m"          except sqlite3.Error as e:              return "ERROR: CHANGES NOT SAVED"        def get\_tables(self) -> list[str]:          """          Retrieves all tables from the database.          """          rows = self.connection.execute(              """SELECT name FROM sqlite\_master WHERE              type = 'table' AND name NOT LIKE 'sqlite%';"""          ).fetchall()          tables = [Table(v) for row in rows for v in row]          for t in tables:              t.columns = self.get\_fields(t)          return tables        def get\_fields(self,T:Table) -> list[str]:          """          Retrieves all fields from the specified table.          """          rows = self.connection.execute(              f"SELECT name FROM pragma\_table\_info({str(T)})"          ).fetchall()          return [Field(v,table=T) for row in rows for v in row]      def execute(self,sql: Query, rowid = True) -> "Report":          """          Executes the specified SQL query.          """          try:              con = self.connection              rep = self.Report(sql=sql)              sql = str(sql).replace('SELECT','SELECT rowid AS id,') if rowid else str(sql)              rows = con.execute(sql).fetchall()              if rows == []:                  table = self.get\_table\_name(rep.sql)                  rows = con.execute(f"SELECT rowid AS id, \* FROM {table}").fetchall()              rep.rows = self.reveal(rows)          except sqlite3.Error as sql\_e:              rep.error = sql\_e          except Exception as ex:              rep.error = ex          finally:              return rep        def execute\_script(self,script) -> str:          """          Executes the specified SQL script.          """          try:              self.connection.executescript(script)              return "Script executed without errors"          except sqlite3.Error as err:              return "Script executed with SQL error"          except Exception as ex:              return "Script executed but error"      def reveal(self,rows) -> list[dict[str]]:          """          Reveals the rows by converting them into dictionaries.          """          revealed\_rows = []          column\_names = [column for column in rows[0].keys()]          for row in rows:              revealed\_row = {}              for c in column\_names:                  revealed\_row[c] = row[c]              revealed\_rows.append(revealed\_row)          return revealed\_rows      @staticmethod      def get\_table\_name(sql:Query) -> str:          """          Extracts the table name from the SQL query.          """          if bool(sql.\_from):              """SELECT or DELETE""" # for future use              t = sql.\_from[0].get\_table\_name()          elif bool(sql.\_insert\_table):              """INSERT"""              t = sql.\_insert\_table.get\_table\_name()          elif bool(sql.\_update\_table):              """UPDATE"""              t = sql.\_update\_table.get\_table\_name()          else:              """unknown"""              t = "\033[31mNOT FOUND\033[0m"          return t      @staticmethod      def toCSV(data,fname="output.csv"):          """          Converts the data into a CSV file.          """          with open(fname,'a') as file:              file.write(",".join([str(j) for i in data for j in i]))      @staticmethod      def summary(rows):          """          Generates a summary of the rows.          """          # split the rows into columns          cols = [ [r[c] for r in rows] for c in range(len(rows[0])) ]            # the time in terms of fractions of hours of how long ago          # the sample was assumes the sampling period is 10 minutes          t = lambda col: "{:.1f}".format((len(rows) - col) / 6.0)          # return a tuple, consisting of tuples of the maximum,          # the minimum and the average for each column and their          # respective time (how long ago, in fractions of hours)          # average has no time, of course          ret = []          for c in cols:              hi = max(c)              hi\_t = t(c.index(hi))              lo = min(c)              lo\_t = t(c.index(lo))              avg = sum(c)/len(rows)              ret.append(((hi,hi\_t),(lo,lo\_t),avg))          return ret        class Report: # wrapper for sqlite list of rows          """Wrapper for sqlite3 Rows"""          def \_\_init\_\_(self, error: sqlite3.Error = None, sql: Query = None, rows: list[sqlite3.Row] = None):              self.error = error              self.sql = sql              self.rows = rows          def has\_error(self):              """              Checks if there is an error.              """              return True if self.error else False          def get\_values(self) -> list[list]:              """              Retrieves the values.              """              l = []              for row in self.rows:                  l += [[val for val in row]]              return l          def get\_columns(self):              """              Retrieves the columns.              """              return self.rows[0].keys() # get keys from first row in list  if \_\_name\_\_ == "\_\_main\_\_":      # TODO TEST CODE HERE      parent = f"{\_\_file\_\_}\\..\\test.py" |