```
import os
import sys
import argparse
import logging
import re
from collections import deque, OrderedDict
from pprint import pprint
import glob

def getBuilderLogger():
    """ setup the logger to stream to the console """

    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

    consoleLogger = logging.StreamHandler()
    consoleLogger.setLevel(logging.INFO)
    consoleLogger.setFormatter(formatter)
    logging.getLogger('').addHandler(consoleLogger)

    logger = logging.getLogger('SQL Build logger')
    logger.setLevel(logging.INFO)
    return logger

class SQLDeployScriptBuilder(object):
    """
        given an sql folder of the following format
                    sql_folder
                              /tables
                              /indexes
                              /triggers
                              /views
                              /stored_procedures
                              /release_sqls
                              All the files in each of the subfolders should be .sql files

        combines all the db objects in the correct order into a single file
        so that it can be run against the database without any dependency issues.

        For example, this ensures the following order
                     tables -> indexes -> views -> triggers -> stored procedures ->
                     release sqls

        Also, the stored procedures(SP) could have inter dependencies.
                SP-A executes SP-B and SP-C
                SP-B executes SP-D
                SP-C executes SP-D and SP-E

        Note that the nature of the problem ensures that there will no cyclic
        dependencies,
        because we have to compile the child stored procedures before compiling the
        parent ones.

        Although it can be possible by using dynamic sql, this will not happen in our
        projects
        because we are prevented from using such constructs.
```

```
    So, we parse the stored procedures and get dependant stored-procedures and
    represent them
    in a graph, represented as a dict of lists
    For further information please read : http://www.python.org/doc/essays/graphs/

    Then we can traverse the graph and get the ordered stored procedures list and
    combine them into
    the final deploy script

    """

    def __init__(self, sql_folder, output_file):
        self.logger = getBuilderLogger()
        self.sql_folder = sql_folder
        try:
            self.output_file_handle = open(output_file , 'wb')
        except IOError as e:
            self.logger("I/O error ({0}): {1}".format(e.errno, e.strerror))
            raise
        except:
            self.logger("Unexpected error:", sys.exc_info()[0])
            raise

    def build_deploy_script(self):

        folders = ['tables', 'indexes', 'triggers', 'views', 'sprocs', 'release_sqls']
        for folder in folders:
            self.logger.info("Processing - {0} ...".format(folder))
            self._concat_dbitems(self._get_dbitems(folder), folder)

        self.logger.info("Created the deploy script: {0}".format(self.output_file_handle.
        name))
        try:
            self.output_file_handle.close()
        except Exception,e:
            self.logger.info("Output File could not be closed properly !", str(e))
            raise

    def _get_dbitems(self, folder):

        # !!! ONLY THE FILES ENDING WITH .sql EXTENSION WILL BE CONSIDERED !!!
        glob_string = os.path.join(self.sql_folder, folder, r'*.sql')
        dbitems = glob.glob(glob_string)

        if not dbitems:
            self.logger.info(" No *.sql files found in directory : %s " % folder)

        # stored procedures specific logic to re-order the stored procedures by the
        sequence of creation
        if folder == 'sprocs':
            graph = {}
            sproc_filename = {}
            for dbitem in dbitems:
                sproc_file, sproc_name , dep_sproc_names = self._parse_sproc_file(dbitem)
```

```
                graph[sproc_name] = dep_sproc_names
                sproc_filename[sproc_name] = sproc_file
            ordered = self._traverse(graph)
            dbitems = [ sproc_filename[x] for x in ordered ]

        return dbitems

    def _concat_dbitems(self, dbitems, folder):
        out_f = self.output_file_handle
        out_f.write("PRINT '<<Deploying %s ... >>'" % folder)
        out_f.write(os.linesep)
        out_f.write('GO')
        out_f.write(os.linesep)
        out_f.write('GO')
        out_f.write(os.linesep)

        for file in dbitems:
            with open(file) as f:
                for line in f:
                    out_f.write(line)
                out_f.write(os.linesep)

    def _parse_sproc_file(self, sproc_file):
        f = open(sproc_file)
        data = f.read()
        f.close()

        # Remove mutli line comments
        data = re.sub(r'/\*.*?\*/', os.linesep, data, flags=re.DOTALL )
        # Remove single line comments
        data = re.sub(r'--.*', '', data, flags=re.I)

        # Get the stored procedure name
        proc_name = re.findall('(procedure|proc)\s+\w+.(\w+)',data, flags=re.I)[0][-1]

        # Get the dependant stored procedure name(s)
        dep_proc_names = [ x[-1] for x in re.findall('(exec|execute)\s+(\w+)',data, flags
        =re.I)]

        return [ sproc_file, proc_name , dep_proc_names ]

    def _traverse(self, g):
        not_run = deque(g.keys())
        ordered = OrderedDict()

        while not_run:
            item = not_run.pop()
            if item in ordered:
                pass
            elif not g[item]:
                ordered[item]=True
            else:
                flag = True
                for x in g[item]:
                    if x not in ordered:
```

```
                        if not g[x]:
                            ordered[x]=True
                        else:
                            not_run.appendleft(x)
                            flag = False
                if flag:
                    ordered[item]=True
        return ordered.keys()


if __name__ == '__main__':

    parser = argparse.ArgumentParser()
    parser.add_argument("-in_dir", "--input_directory", type=str,
                        required=True, help="Input directory where the db items are
                        present")
    parser.add_argument("-o", "--output_file", type=str,
                        required=True, help="Output file into which the db items SQL
                        scripts are concatenated")
    args = parser.parse_args()

    if not os.path.isdir(args.input_directory):
        print "Input Directory : %s is INVALID" % args.input_directory
        sys.exit(-1)

    if not args.output_file:
       print "Output file is not specified. Please use the --output_file/-o parameter"
       sys.exit(-1)

    builder = SQLDeployScriptBuilder(args.input_directory, args.output_file)
    builder.build_deploy_script()
```