Task 3: checkLogins

Features

- **Create-Mode**:
  - Create a Database populated with the data in */var/log/auth.log*
  - Test-suite: Run a set of SQL-queries to test the Database
- **Append-Mode**: Add new entries to the database without recreateing it
- **Interactive-Mode**: Process SQL-queries on the created Database interactively
- **User-Mode**: Display data through a simple CLI and export it to a CSV file

Downloads

- Script: checkLogins
- Reference: Test-suite tables
- Test-suite query as CSV

How to Use

Setup

To run the checkLogins script requires the third-party libraries psycopg2 and rich for Python and PostgreSQL as third-party implementation of relational databases. The Python libraries can be installed using pip (lines 1 and 2) while PostgreSQL is provided by apt (line 3).

```
pip install psycopg2-binary
pip install rich
apt install postgresql
```

To use PostgreSQL, it is convenient to create a new SQL-user with a password who is able to create new databases. For that, switch first to the user postgres (line 1). Then start the PostgreSQL interactive terminal psql (line 2). Within psql, create a SQL-user (line 3) and provide them with a password (line 4). Since the script creates a database, give the SQL-user the permission to do that (line 5). To check if the SQL-user was created, list all SQL-users (line 6). Then exit (lines 7 and 8).

```
jan@linux:~$ sudo -iu postgres
postgres@linux:~$ psql
postgres=# CREATE USER ser;
postgres=# \password ser
postgres=# ALTER USER ser CREATEDB;
postgres=# \du
postgres=# \q
postgres@linux:~$ exit
```

Creation of the Database

Running the checkLogins script in create-mode (explicitely with arguments *-c* or *--create*) requests the username and password of the SQL-user created previously. It assembles a database owned by the SQL-user which contains the data of the Log-file */var/log/auth.log*.

```
jan@linux:~$ python3 check_logins_OOP.py -c

| checkLogins |

Provide login details for the database.
Username: ser
Password: ******

| Create-mode |

Creating the database "auth_logs"...
Database created.

Creating and filling the database relations...
Database relations created and filled.

Creating sample queries and output...
Created sample queries and output.
```

Inspection of the Database

To test whether the database was created correctly and allows for queries to extract data, the checkLogins script runs a Test-suite and writes the results into the file *auth_tests.txt*. For comparison, the results of the Test-suite can be downloaded.

> One example which is answered in the Test-suite is the question for the usernames, IP addresses, number of failed login attempts and whether the user exists on the system. To get access to all four attributes selected in the first two lines, the table for the sessions has to be joined with the tables of the users and IP-addresses by using the respective IDs as common denominator.
>
> ```
> SELECT users.user_name, ip_addresses.ip_address,
> sessions.fail_count, users.user_exists
> FROM sessions
> JOIN users ON sessions.user_id = users.user_id
> JOIN ip_addresses ON sessions.ip_id = ip_addresses.ip_id;
> ```
>
> The entity-relationship diagram discussed later provides the knowledge to decide when to join and when all requested information is already at hand.

To manually access the database via queries, three options are available. Running the checkLogins script in user-mode (explicitely with arguments *-u* or *--user*) allows to infer attributes through a simple CLI.

```
jan@linux:~$ python3 check_logins_OOP.py -u

| checkLogins |

Provide login details for the database.
Username: ser
Password: ******

| User-mode |


Available attributes:
user_name, user_exists, ip_address, pid,
fail_count, login_status, first_date_time, last_date_time

Available filters:
> Restrict arguments: arg <, >, <=, >=, =, != value
> Simple functions: max(arg), min(arg)
> Counting: count(arg), count(arg=value)
> Sorting: asc(arg), desc(arg)
> Date: 'yyyy-mm-dd'
> Time: 'HH:[MM:[SS]]'
> Case-sensitive Regex: arg ~ regex

Export previous output to csv: export filename
Syntax: statement_1, statement_2, ... statment_n;

Press q+Enter to quit.

??? user_name, user_exists;
+-------------------------+
| user_name | user_exists |
|-----------+-------------|
| spiderman | True        |
| ironman   | True        |
| postgres  | False       |
| jan       | True        |
| mira      | True        |
| ast       | True        |
| platon    | False       |
| ide       | True        |
| obe       | True        |
| cicero    | False       |
| socrates  | False       |
| odysseus  | False       |
+-------------------------+

Available attributes:
user_name, user_exists, ip_address, pid,
fail_count, login_status, first_date_time, last_date_time

Available filters:
> Restrict arguments: arg <, >, <=, >=, =, != value
> Simple functions: max(arg), min(arg)
> Counting: count(arg), count(arg=value)
> Sorting: asc(arg), desc(arg)
> Date: 'yyyy-mm-dd'
> Time: 'HH:[MM:[SS]]'
> Case-sensitive Regex: arg ~ regex

Export previous output to csv: export filename
Syntax: statement_1, statement_2, ... statment_n;

Press q+Enter to quit.

??? q
```

The second option is to run the checkLogins script in interative-mode (no arguments or explicitely with arguments *-i* or *--interactive*) which allows to compose and run queries.

```
jan@linux:~$ python3 check_logins_OOP.py -i

| checkLogins |

Provide login details for the database.
Username: ser
Password: ******

| Interactive-mode |


Enter query or q+Enter to quit:
SELECT * FROM users;
+-----------------------------------+
| user_id | user_name | user_exists |
|---------+-----------+-------------|
| 1       | spiderman | True        |
| 3       | ironman   | True        |
| 27      | postgres  | False       |
| 28      | jan       | True        |
| 39      | mira      | True        |
| 47      | ast       | True        |
| 55      | platon    | False       |
| 71      | ide       | True        |
| 73      | obe       | True        |
| 74      | cicero    | False       |
| 93      | socrates  | False       |
| 125     | odysseus  | False       |
+-----------------------------------+

Enter query or q+Enter to quit:
q
```

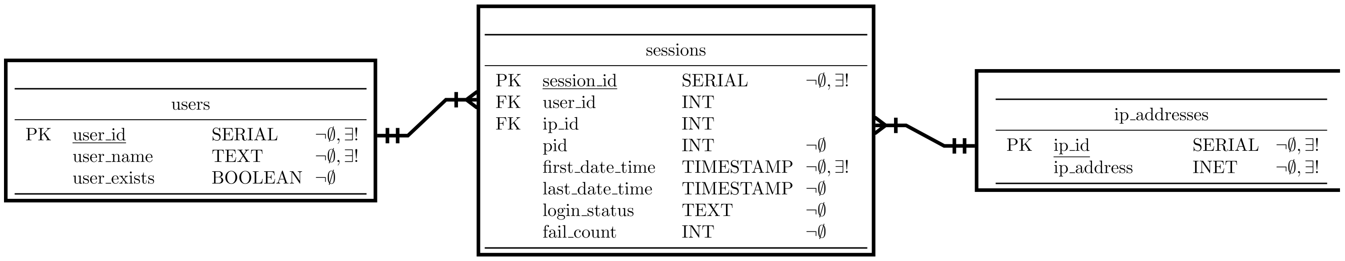
The last option is to use psql to access the database. First switch to the user postgres (line 1). Then start the PostgreSQL interactive terminal psql (line 2). Switch over to your created SQL-user (line 3). Check if the database *auth_logs* was created by listing all tables (line 4). Connect to the database *auth_logs* (line 5). Compose and run a query (line 6).

```
jan@linux:~$ sudo -iu postgres
postgres@linux:~$ psql
postgres=# SET ROLE ser;
postgres=> \l
postgres=> \c auth_logs
auth_logs=# SELECT * FROM users;
 user_id | user_name | user_exists
---------+-----------+-------------
       1 | spiderman | t
       3 | ironman   | t
      27 | postgres  | f
      28 | jan       | t
      39 | mira      | t
      47 | ast       | t
      55 | platon    | f
      71 | ide       | t
      73 | obe       | t
      74 | cicero    | f
      93 | socrates  | f
     125 | odysseus  | f
(12 rows)
auth_logs=# \q
postgres@linux:~$ exit
```

Composition of queries

To compose queries to answer questions, one needs a layout of the database: Which tables are there and how are they related. Such a layout is provided by an entity-relationship diagram (ERD) which is shown below for the *auth_logs* database. Central to everything are the sessions with their session ID as unique identifier. They are connected to the users and IP addresses by means of their unique IDs.

The keys in the first, the datatype in the third and further constraints in the fourth column of the tables are not relevant to compose queries that only access the data without altering it. Also the types of relationship denoted by different connectors are not relevant for this type of queries. More on these topics will be discussed in the section on Tasks & Design choices.



To get access to the usernames, IP addresses, number of failed login attempts and whether the user exists on the system, one first has to join the tables sessions, users and IP addresses.

Dependencies

Python Standard Library:

- re
- subprocess
- sys
- getpass
- csv
- itertools
- textwrap
- functools
- readline
- datetime
- os

Third-party libraries:

- psycopg2
- rich

Third-party software:

- PostgreSQL

Technical Summary

The checkLogins script organizes the data contained in the Log-file */var/log/auth.log* in a database using Python as interface and PostgreSQL to access and manipulate the data. It provides four modes, one to create the database, one to append data to the database and two to interact with it. For all modes the user specifies a SQL-user and their password to own the database.

In Create-mode, first the database *auth_logs* is created. Then the relations are created and filled with the data from */var/log/auth.log*. After adding foreign key constraints, the Testing-suite consisting of several SQL-queries is executed.

In Append-mode, the database is extended with new data using the timestamp of the last entry of the database as reference.

In Interactive-mode, the user can enter their own SQL-queries to access and manipulate the database.

In User-mode, the user can display attributes within the database through a simple CLI and export the results in CSV-files.

Features & Extensibility

- **Create-Mode**: In Create-Mode, the checkLogins script first copies the Log-file */var/log/auth.log* into the users home-directory (for sudo into /home). It then extracts the data from the Log-file and creates the database *auth_logs*. Both, Log-file and database name, are hard-coded for convenience. In the end, the copy of the Log-file is deleted.

  To **create another database using a different logfile**, the path to the Log-file, the variables *db_name* and *db_tests*, as well as the LogFile methods *process_log* and *message_filter*, and the Database methods *initialize_relations*, *setup_relations* and *test_suite* have to be adapted.

  In *process_log* and *message_filter*, the filtering rules have to be modified.

  In *initialize_relations*, the relation name, attribute names, attribute types, relation keys, attribute constraints and levels have to be specified for the new database. Foreign keys are specified by the relation name and attribute name to which the foreign key links. Since primary keys automatically satisfy the *UNIQUE* and *NOT NULL* constraints, the constraint entry of the primary key has to be empty. In case *SERIAL* primary keys are used, relations with foreign keys have to specified with the level-attribute "child", while relations without any foreign keys have the level-attribute "parent". Furthermore, the foreign keys must not have a *NOT NULL* constraint. The level-attribute is used for child-relations to fetch their foreign key values from the parent-relations.

In *setup_relations* the arguments fueling the generator used to process the Log-file line-by-line have to be adapted, as well as starting the generator has to be modified.

In *test_suite*, the *tasks* describing the test cases and the *queries* containing the test cases have to be adapted.

To make the script more flexible, one can read the location of the Log-file, the database name and file name of the Test-suite output from the user and add one method *get_location(self,location)* to the LogFile class and two methods *get_name(self,name)* and *get_tests(self,tests)* to the *Database* class to alter the instance attributes *self._location*, *self.name* and *self.tests*.

- **Append-Mode**: In Append-Mode, the checkLogins script copies the Log-file */var/log/auth.log* into the users home-directory (for sudo into /home). It then extracts the data from the Log-file *auth.log* and appends the new data to the database *auth_logs*. Both, Log-file and database name, are hard-coded for convenience. In the end the copy of the Log-file is deleted.

  To use Append-Mode for another database, the criterion used to identify the last added entry has to be modified. In case there is no reference, the Append-Mode can be simplified to parse the entire database.
- **Interactive-Mode**: In Interactive-Mode, the database *auth_logs* can be accessed. The name of the database is hard-coded for convenience.

  To **access other databases**, the variable *db_name* has to be adapted. To make the scipt more flexible, one can read the database name from the user and add the method *get_name(self,name)* to the *Database* class to alter the instance attribute *self.name.*
- **User-Mode**: In User-Mode, the database *auth_logs* can be accessed by means of a simple CLI. The name of the database is hard-coded for convenience.

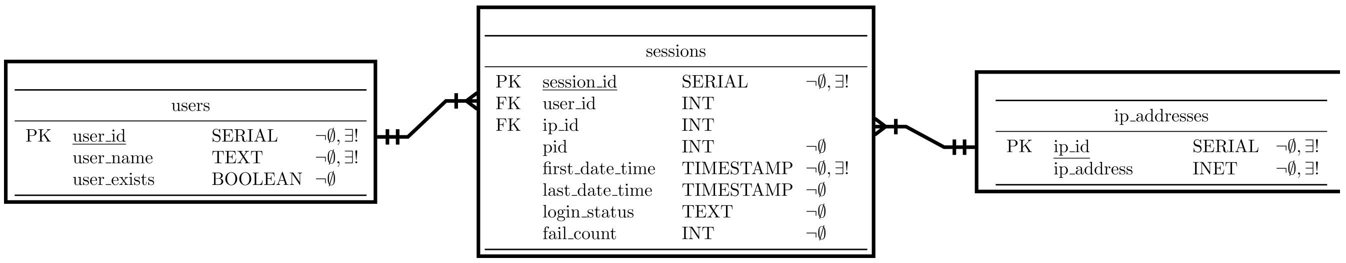
  This mode is similar to Interactive-Mode, but less flexible, since it relies on the relations implemented in the method *initialize_relations*. The structure of the interface should be generic enough to **use it for other databases**, once they are implemented in the method *initialize_relation*
  . However this has not been tested yet.
- **Test-suite**: The Test-suite is executed when running the checkLogins script in Create-Mode and its output is stored in the file with hard-coded name *auth_tests.txt*.

  To **extend the Test-suite**, the Database method *test_suite* has to be adapted. In it, the *tasks* describing the test cases and the *queries* containing the test cases have to be extended.

  To make the script more flexible, one can read the file name of the Test-suite output from the user and add a method *get_tests(self,tests)* to the *Database* class to alter the instance attribute *self.tests*.

Tasks & Design choices

Entity-relationship diagram



users

PK	user_id	SERIAL	$\neg\emptyset, \exists!$
	user_name	TEXT	$\neg\emptyset, \exists!$
	user_exists	BOOLEAN	$\neg\emptyset$

sessions

PK	session_id	SERIAL	$\neg\emptyset, \exists!$
FK	user_id	INT	
FK	ip_id	INT	
	pid	INT	$\neg\emptyset$
	first_date_time	TIMESTAMP	$\neg\emptyset, \exists!$
	last_date_time	TIMESTAMP	$\neg\emptyset$
	login_status	TEXT	$\neg\emptyset$
	fail_count	INT	$\neg\emptyset$

ip_addresses

PK	ip_id	SERIAL	$\neg\emptyset, \exists!$
	ip_address	INET	$\neg\emptyset, \exists!$

Relations

The Log-file */var/log/auth.log* contains the following entries:

- timestamp with date and time
- service (usually) with a PID
- message

Using the **PID** to bundle messages into sessions, the only two **timestamps** relevant are the **first** and **last** received from a session. The message in turn contains additional information specific to the service. For the service sshd, these informations are:

- whether a login-attempt has failed or succeeded
- the **name of the user** attempting to login
- the **IP address** of the user attempting to login

The first point allows to derive the **login-status**, i.e. whether the session led to a successful login or not and the overall **number of failed login-attempts** within a session. Having access to the users existing in a system, the username allows to determine whether the user attempting the login actually exists. Note that users existing on the system have a UID, while users which do not exist on the system do not have one. The UID can thus only be used to uniquely identify users existing on the system. However users not existing on the system should also be added to the database, even if it just serves to block an IP-address. The keywords marked in green correspond to the attributes to be inserted into the database.

Relationships

To determine which relations are reasonable, the attributes are separated if they are related by a many-to-one relationship and grouped if they are related by a one-to-one relationship. Multiple sessions can involve the same user or the same IP address. Multiple users can also share the same (public) IP address. In constrast a session can only have one PID, one first and last timestamp, one login-status and one fail-count.

Constraints

The constraints are determined by logical and technical considerations. Primary keys are *UNIQUE* and *NOT NULL*. Without any integer IDs, the *first_date_time*, *user_name* and *ip_address* uniquely identify the entities. Note that PIDs are only unique in the moment and are reused later. However primary keys should be integers, since these provide unique values at a small memory cost and they can be automatically incremented. Therefore the logical primary keys keep their *UNIQUE* and *NOT NULL* constraints, but are represented by *SERIAL* IDs. All other attributes, except for the foreign keys, are constrained to be *NOT NULL*.

The foreign keys do not have a *NOT NULL* constraint for a technical reason. When inserting data into the relations, all SERIAL IDs are automatically incremented, but their values are not automatically copied to the foreign keys which get the value *NULL*. The option *ON UPDATE CASCADE* only updates the foreign key upon an **update** of the primary key. Before that happens, the primary and foreign key have to be "connected" by having the same value. After a SERIAL increment of the primary keys, such a "connection" is not established. Within PostgreSQL, such an automatic update could be implemented using triggers and trigger functions. Here, the automatic copy-mechanism is implemented within Python in the method *fetch_fk*.

Datatypes

For IDs, the SERIAL datatype is used to accomplish automatic increments. The foreign keys are correspondingly of type INT. Further straightforward choices are the *fail_count* as *INT*, the *first_date_time* and *last_date_time* as *TIMESTAMP*s, the *user_name* as *TEXT*, the *user_exists* as *BOOLEAN* and the *ip_address* as *INET*. The *pid* is considered as an *INT*, since its values lie in the matching range. While the login_status is binary, using a *BOOLEAN* would require an explanation, whether a failed login-attempt is represented by *True* or *False*. Therefore a *TEXT* with the two options *Success* and *Failed* is chosen.

Main function

Using the hard-coded Log-file name and location, a *LogFile* object is created. After creating a *User* object, the Log-file is copied to the directory determined by the users home-attribute (home-directory of regular user, /home for root). With the hard-coded database name, the copied Log-file and file name of the Test-suite output, a *Database* object is created. Next, a *SQLUser* object is created and used to ask the user of the script to specify the SQL-user to own the database and their password. These credentials are checked and requested anew if they do not allow access to the *postgres* database.

Depending on the command line argument, the Create-, Append-, Interactive-, User- or Setup-Mode is selected. The default mode without any argument is User-Mode.

In Create-Mode, first the database is (re-)created. Then the relations are initialized and created within the Database. Thereby also primary keys, *NOT NULL* and *UNIQUE* constraints as well as foreign keys are specified. In case the database already exists and is currently in use (e.g. in psql), the user is notified. Finally, the Test-suite is executed.

In Append-Mode, first the database relations as they are known in Python are initialized. Next, new data is appended to the database. Finally, the Test-suite is executed.

In Interactive-Mode, an interactive SQL-session is started.

In User-Mode, the user can request data from the database with a less flexible, but simpler syntax than in Interactive-mode.

In Setup-Mode, first Append-Mode is executed, except for the Test-suite. Then a cronjob is created to run the script regularly in Append-Mode without the Test-suite. Since for that purpose username and password of the SQL-user cannot be requested interactively but have to be stored, it is moved into its own, internal mode.

After the selected mode is executed, the copied Log-file is removed.

```
print("\n| checkLogins |\n")

if (len(sys.argv)==2):
    mode = sys.argv[1]
else:
    mode = "-u"

log_file = LogFile("auth.log","/var/log")
user = User()
log_file.copy_log(user.home)

db_name = "auth_logs"
db_tests = "auth_tests.txt"
db = Database(db_name,log_file,db_tests)

# get login data
sql_user = SQLUser()
if (mode != "--cron_job"):
    print("Provide login details for the database.")
    connected = False
    while (not connected):
        sql_user.get_login()
        connected = db.check_credentials(sql_user)
else:
    sql_user.get_login_env()
```

```
if (mode=="-c" or mode=="--create"):
    print("\n| Create-mode |\n")

    try:
        print(f"Creating the database \"{db_name}\"...")
        db.create_database(sql_user)
        print("Database created.\n")

        print("Creating and filling the database relations...")
        db.initialize_relations()
        db.setup_relations(sql_user,user.list)
        print("Database relations created and filled.\n")

    except psycopg2.errors.OperationalError:
        msg = "Error: Database cannot be rebuild " \
                +"since it is currently in use.\n" \
                +"Using existing database."
        print(msg)

    print("Creating sample queries and output...")
    db.test_suite(sql_user)
    print("Created sample queries and output.")

elif (mode=="-a" or mode=="--append"):
    print("\n| Append-mode |\n")

    print("Appending to database relations...")
    db.initialize_relations()
    db.append(sql_user,user.list)
    print("Database extended.\n")

    print("Creating sample queries and output...")
    db.test_suite(sql_user)
    print("Created sample queries and output.\n")

elif (mode=="-i" or mode=="--interactive"):
    print("\n| Interactive-mode |\n")

    interactive_queries(sql_user,db.name)

elif (mode=="-u" or mode=="--user"):
    print("\n| User-mode |\n")

    db.initialize_relations()
    db.interface(sql_user)

elif (mode=="-s" or mode=="--setup"):
    print("\n| Setup-mode |\n")

    # append to database from cron-job
    db.initialize_relations()
    db.append(sql_user,user.list)

    # setup cronjob
    cronjob = CronJob("0","0","*","*","*","python3",f"{__file__} --cron_job")
    if (not cronjob.active):
        cronjob.add_cronjob(sql_user)

elif (mode=="--cron_job"):
    # append to database from cron-job
    db.initialize_relations()
    db.append(sql_user,user.list)


# clean up
log_file.rm_log()
```

- **Q**: Why are the database name, Log-file name, Log-file location and file name of the Test-suite output hard-coded?
  A: Since there is only one database, one Log-file and one Test-suite output, the variables are hard-coded for convenience. An extension is discussed in the section on Features & Extensibility.
- **Q**: Why is User-Mode the default mode?
  A: Since Create-Mode *DROP*s the database and re-creates it, any changes in the database since creation would be lost by accidentally forgetting the command-line argument. Append-Mode also would modify the database. Setup-Mode could lead with an inexperienced user to a damaged crontab file. While Interactive-Mode does not directly alter the database, it gives direct access to the database which can then be altered by the user. Furthermore, it requires knowledge of the PostgreSQL syntax to interact with the database. It is thus neither safe, nor pleasant to use. User-Mode on the other hand does not change the database directly and also does not empower the user to alter it. It is also simple to control.
- **Q**: What is meant by (re-)created?
  A: In case the database already exists, it is *DROP*ped and created anew. If it does not exist, it is created.
- **Q**: What is meant by initialization of relations and setup of relations?
  A: During initialization of relations, the *Relation* objects in Python are created. During setup of relations, the relations are created in the database.

LogFile class

The *LogFile* class has two "private" instance attributes:

- *self._name* (*str*): Name of the Log-file
- *self._location*(*str*): Path to the Log-file

```
def __init__(self,name,location):
    """Constructs necessary attributes of the LogFile object."""

    self._name = name
    self._location = location
```

- **Task**: To avoid accidentally altering the Log-file, it is copied to a destination, where it can be used and modified.
  - **Implementation**: Using the call method of the *subprocess* module, the copy command *cp* in the shell is invoked. After copying, the location of the copy is stored in the *_location* attribute.

  ```
  def copy_log(self,destination):
      """
      Copies Log to destination.

          Parameters:
              destination (str): Absolute path
      """

      cmd = f"cp {self._location}/{self._name} {destination}"
      subprocess.call(cmd,shell=True)
      self._location = destination
  ```

- **Task**: After using the copy of the Log-file, it should be removed to leave a clean directory and prevent unauthorized access to the Log-file.
  - **Implementation**: Using the call method of the *subprocess* module, the remove command *rm* in the shell is invoked.

  ```
  def rm_log(self):
      """Removes Log file."""

      cmd = f"rm {self._location}/{self._name}"
      subprocess.call(cmd,shell=True)
  ```

- **Task**: Read the system log file.
  - **Implementation**: Read the system log file by means of a generator, one line at a time. Account for a missing file and a lack of permissions.

  ```
  def _read_log(self):
      """Reads from Log file, one line at a time."""

      fname = f"{self._location}/{self._name}"
      try:
          with open(fname,"r") as logfile:
              for line in logfile:
                  yield line
      except FileNotFoundError:
  ```

```
        msg = f"The file {fname} does not exist."
        print(msg)
    except PermissionError:
        msg = f"You lack permission to read {fname}."
        print(msg)
```

- **Q**: Does the system log file fit into memory when reading it?
  A: This is unclear. A generator is used to read the system log file, one line at a time.
- **Q**: Would it be expedient to store the content of the system log file in a specific data structure?
  A: Since the size of the system log file is unclear, it is read line by line. With the size of the system log file unclear, it is also unclear how many lines passing the filters there are. Appending the lines passing the filters to a list is therefore not sensible.

- **Task**: Examine the Log-file and extract the attributes described in the ERD.
  - **Implementation**: First, the header corresponding to the extracted attributes is returned, which allows to interpret the lists of values returned later.

    The dictionary *logged_sessions* is used to store all but the *login_status* and the *last_date_time* attribute which (can) change in every line of the Log-file. This serves to enforce certain key-entries in the list for *NOT NULL* constraints in the relations, to allow for cumulative entries, as well as to memorize initial values for the different sessions, The *service_whitelist* restricts the Log-file to the sshd service, while the *message_blacklist* omits lines with messages containing the blacklists content. Before the whitelist and blacklist are applied,each line of the Log-file is roughly filtered into five parts which give the (compound) attributes *date_time*, *service* +PID and the *message*. In case date_time is earlier than a provided *buffer_time*, the next line is read. This is used in Append-mode to skip lines of already terminated ssh login-sessions. After separating *service* and PID, the white- and blacklist are used. Next, the *logged_sessions* dictionary, its *fail_count* and *first_date_time* entries are initialized for the new PID. Within the utility method *message_filter* containing filters for the messages, the other entries of the *logged_sessions* dictionary are populated. In case the *UNIQUE* key-entries *ip_address* of the *ip_addresses* relation and *user_name* of the *users* relation are not filled, the next line in the Log-file is considered, until all key-entries for a given PID are accumulated in the dictionary. Before yielding the line containing all attributes, the *date_time* of the line is compared with a *break_time*. This is used in Append-mode to skip lines of possibly still running ssh-logins which are already contained in the database. The *buffer_time* and *break_time* serve to suppress terminated ssh login-sessions, but populate the dictionaries with entries of still running ssh login-sessions without entering lines already contained in the database. Once the *break_time* is passed, the sorted line of attributes is yielded.

```
def process_log(self,user_list,buffer_time,break_time):
    """Examines Log and extracts relevant data."""

    header = ["pid","fail_count","login_status", \
              "first_date_time","last_date_time", \
              "ip_address", \
              "user_name","user_exists"]
    yield header


    # purpose:  enforce key-entries, allow cumulative
    # entries (e.g. counters) and initial values (e.g. start time)
    # for each session
    logged_sessions = dict()

    # filter lists
    service_whitelist = ["sshd"]
    message_blacklist = ["(sshd:session)","Server listening"]

    for line_log in self._read_log():

        # rough filter
        pattern = r"^(.+?)T(.+?)\s(.+?)\s(.+?):\s(.+?)$"
        line = re.findall(pattern,line_log)[0]
        # remove timezone
        date_time = line[0]+" "+line[1].split("+")[0]
        pid = "-1"
        service = line[3]
        message = line[4]


        # start accumulating entries after buffer time
        # (lifetime of ssh login-session before break_time)
        if (buffer_time > datetime.fromisoformat(date_time)):
            continue


        # resolve service, pid
        if ("[" in service):
            pattern = r"^(.+?)\[(.+?)\]$"
            service,pid = re.findall(pattern,service)[0]
        # filter service, messages
        if (service in service_whitelist and \
            not any([bl_entry in message \
                 for bl_entry in message_blacklist])):

            if (pid not in logged_sessions.keys()):
                # initialization
                logged_sessions[pid] = dict()
                logged_sessions[pid]["fail_count"] = 0
                logged_sessions[pid]["first_date_time"] = date_time

            logged_sessions,login_status = self.message_filter( \
                                                 logged_sessions, \
                                                 pid,message, \
                                                 user_list)

            # pass line only if key-entries are present
            if ("ip_address" not in logged_sessions[pid].keys() or \
                "user_name" not in logged_sessions[pid].keys()):
                continue


            # pass entries after break time
            if (break_time >= datetime.fromisoformat(date_time)):
                continue


            # line according to header
            line_sorted = [pid, \
                           logged_sessions[pid]["fail_count"], \
                           login_status, \
                           logged_sessions[pid]["first_date_time"], \
                           date_time, \
                           logged_sessions[pid]["ip_address"], \
                           logged_sessions[pid]["user_name"], \
                           logged_sessions[pid]["user_exists"]]
            yield line_sorted
```

    - **Q**: How are the entries of the blacklist determined?
      A: Heuristically, i.e. by looking at the messages in the Log-file and determining the common entry which highlights messages without any of the attributes of interest.
    - **Q**: What is caught by the third group of the rough filter?
      A: The hostname of the system.
    - **Q**: Why are *login_status* and *date_time* not entries in *logged_sessions*?
      A: Using a dictionary here to store data is problematic since it fills up the RAM with the content of the file. Since the size of the file is unknown, the RAM may not be sufficient. The overall goal is thus to store as few things in such a dictionary as possible. Since *login_status* and (*last_*)*date_time* change with each line of the Log-file, there is no need to store them.

      Note that in case the Log-file should indeed be too large at some point in time, the file can be split into parts and the Append-Mode can be used to append each part at a time.
    - **Q**: Why is this function also a generator, just like the function to read the Log-file?
      A: Since the lines are not written into the database in this function, they would have to be stored in RAM. To avoid this, a generator is used.

      Note that from the outside, this function looks like the function reading the file, albeit a processed file. Hence wrapping generators allows to create a processing pipeline. Generators of generators are like cogs where the innermost cog only moves once all outer cogs act. Another interesting behavior is the continue statment within a generator. In the cog-metaphor, the gear ratio can be altered.
- **Task**: Infer the attributes *ip_address*, *user_name*, *user_exists*, *fail_count* and *login_status* from the messages in the Log-file.
    - **Implementation**: To get the IP address four numbers separated by dots with between one and three digits are searched. For the username, the strings "password for " or "user " or "user=" **not** followed by the strings "invalid" or "unknown" are followed by the username as a word. To determine if a user exists on the system, it is checked whether the username is contained in the list of usernames contained in */etc/passwd* which is obtained in the constructor of the *User* object. An increment by one of the *fail_count* is determined by the words "Failed password" at the beginning of a message. After an initial failed login, further messages are bundled, such that an increment by <n> can be obtained by searching for a number with one or more digits in a string "message repeated <n> times: [Failed password". A successful login attempt is characterized by the words "Accepted password" at the beginning of a message.

```
@staticmethod
def message_filter(logged_sessions,pid,message,user_list):
    """Filter a message for attributes."""
    # IP address
    pattern = r"("+r"[0-9]{1,3}\."*3+r"[0-9]{1,3})"
    search = re.search(pattern,message)
    if (bool(search)):
        logged_sessions[pid]["ip_address"] = search.group(1)

    # username, user existence
    pattern = r"(password for |user |user=)(?!invalid|unknown)(\w+)"
    search = re.search(pattern,message)
    if (bool(search)):
        logged_sessions[pid]["user_name"] = search.group(2)
        logged_sessions[pid]["user_exists"] = \
                bool(logged_sessions[pid]["user_name"] in user_list)

    # fail count
    pattern = r"^Failed password"
    search = re.search(pattern,message)
    if (bool(search)):
        logged_sessions[pid]["fail_count"] += 1
    pattern = r"message repeated ([0-9]+) times: "+ \
              r"\[ Failed password"
    search = re.search(pattern,message)
    if (bool(search)):
        logged_sessions[pid]["fail_count"] += int(search.group(1))

    # login status
    login_status = "Failed"
    pattern = r"^Accepted password"
    if (bool(re.search(pattern,message))):
        login_status = "Success"

    return logged_sessions,login_status
```

CronJob class

The CronJob class has seven "private" and one public instance attributes:

- *self._m (str)*: Value of minute field in crontab
- *self._h (str)*: Value of hour field in crontab
- *self._dom (str)*: Value of day-of-month field in crontab
- *self._mon (str)*: Value of month field in crontab
- *self._dow (str)*: Value of day-of-week field in crontab
- *self._exe (str)*: Executable/program called in crontab
- *self._file (str)*: File executed in crontab
- *self.active (bool)*: Cronjob entered in crontab (active) or not (inactive)

- **Task**: Check if the cronjob already exists in the crontab file.
    - **Implementation**: To check if the cronjob already exists in the crontab file, crontab -l is executed. In case no cronjob exists, the request returns a failure which raises an CalledProcessError exception. This is handled in a try-except statement.

```
def __init__(self,m,h,dom,mon,dow,exe,file):
    """Constructs all necessary attributes for the CronJob object."""

    self._m = m
    self._h = h
    self._dom = dom
    self._mon = mon
    self._dow = dow
    cmd = f"which {exe}"
    self._exe = subprocess.check_output(cmd,shell=True,text=True) \
                        .strip()
    self._file = file

    self.active = True
    cmd = "crontab -l"
    try:
        crontab = subprocess.check_output(cmd,shell=True,text=True, \
                                            stderr=subprocess.DEVNULL)
    except subprocess.CalledProcessError:
        # if no crontab for user
        crontab = ""

    # check if cronjob for __file__ is set
    if (self._file not in crontab):
        self.active = False
```

- **Task**: Run the script regularly to keep the database up-to-date with the Log-file.
  - **Implementation**: Inform the user about the ensuing creation of a cronjob and wait for an input by the user. Add an exemplary cronjob to the crontab file, then display the crontab file of the user.

    One remaining obstacle is to save the access data of the SQL-user for the cronjob without hard-coding it. One avenue is to save the credentials in environment variables. To set environment variables persistently from within Python appears to be tricky. The present implementation has not been successfully tested yet. The planned feature on the Setup-mode essentially refers to storing the credentials.

```
def add_cronjob(self,sql_user):
    """Adds cronjob to crontab."""

    print("Create a cronjob to execute this "+ \
            "run Append-mode regularly.\n")
    input("Press Enter to modify the exemplary cronjob "+ \
            "on the last line... ")

    # add cronjob to crontab
    job = f"{self._m}\t{self._h}\t"+ \
          f"{self._dom}\t{self._mon}\t{self._dow}\t"+ \
          f"{self._exe} {self._file}"
    cmd = f"(crontab -l; echo \"{job}\") | crontab -"
    print(cmd)
    subprocess.call(cmd,shell=True)
    # allow for modification of cronjob or approval
    cmd = "crontab -e"
    subprocess.call(cmd,shell=True)

    # store credentials in environment variable
    print("Please store your SQL-user credentials \n"+\
            "in the following way:\n"+ \
           f"export CHECK_LOGIN_USR=<username>\n"+ \
           f"export CHECK_LOGIN_PWD=<password>")
```

  - **Q**: Should I create a cronjob or is there an alternative?
    A: Since the script is supposed to run on Linux servers, using a cronjob is straightforward.
  - **Q**: Should I automatically append the cronjob to the crontab file or let the user enter it?
    A: To allow for comfortable controls for the user while maintaining their agency, the cronjob is added to the crontab file and shown to the user to be approved or modified. Note that any modification of the crontab file happens after the user is forwarded to the crontab file, hence no backup of the crontab has to be created to account for a KeyboardInterrupt.

    There exists the crontab module in Python, however it is not part of the Python standard library and would introduce additional dependencies.

Relation class

The *Relation* class has six public and five "private" instance attributes:

- *self.name* (*str*): Name of the relation
- *self.attrs* (*tuple* of *str*): Attribute names of the relation
- *self.types* (*tuple* of *str*): Data types of the attributes
- *self.keys* (*tuple* of *str*): Strings specifying private and foreign keys
- *self.cstrs* (*tuple* of *str*): Strings specifying *NOT NULL* and *UNIQUE* constraints
- *self.level (str)*: Flag with the following values:
  - "child": relation has foreign key values depending on the primary key of other relations
  - "parent": relation has no foreign key values depending on the primary key of other relations
- *self._sql_name*: SQL-conform version of *self.name*
- *self._sql_attrs*: SQL-conform version of *self. attrs*
- *self._sql_types*: SQL-conform version of *self.types*
- *self._sql_cstrs*: SQL-conform version of *self.cstrs*
- *self._attr_dict* (dict of str:int): Dictionary to translate lines of the input into attributes of the relation, see method *create_attr_dict*.

The datatypes of the SQL-conform versions can be inferred from the code block and the psycopg2 documentation.

```
def __init__(self,name,attrs,types,keys,cstrs,level):
    """Constructs necessary attributes of the Relation object."""

    self.name = name
    self.attrs = attrs
    self.types = types
    self.keys = keys
    self.cstrs = cstrs
    self.level = level

    self._sql_name = sql.Identifier(self.name)
    self._sql_attrs = tuple(map(sql.Identifier,self.attrs))
    self._sql_types = tuple(map(sql.SQL,self.types))
    self._sql_cstrs = tuple(map(sql.SQL,self.cstrs))

    self._attr_dict = {}
```

- **Q**: What is th purpose of the level-attribute?
  A: The purpose of the level-attribute is to identify relations with foreign keys which have to fetch their foreign key values from *SERIAL* primary keys.

  Background: When inserting data into the relations, all SERIAL IDs are automatically incremented, but their values are not automatically copied to the foreign keys which get the value *NULL*. The option *ON UPDATE CASCADE* only updates the foreign key upon an **update** of the primary key. Before that happens, the primary and foreign key have to be "connected" by having the same value. After a SERIAL increment of the primary keys, such a "connection" is not established. Within PostgreSQL, such an automatic update could be implemented using triggers and trigger functions. Here, the automatic copy-mechanism is implemented within Python in the method *fetch_fk*.
- **Q**: Why are SQL-conform representations of strings needed?
  A: In case data provided by the user contains SQL-statements and quotes to counter escaping quotes, the user can execute SQL queries although they should not have permission to do so. This issue is called SQL-injection. The psycopg2 documentation elaborates on this page more on the topic and has the sql module to counter it. The webcomic xkcd has a nice illustration of the possible consequences.
- **Q**: Why is there no SQL-conform version of the key introduced?
  A: The keys encompass primary and foreign keys. Since they are applied at different times and require different handling, they are kept as tuples of strings until processed.
- **Q**: Why is *_attr_dict* only an instance attribute and not a class attribute?
  A: If *_attr_dict* were a class attribute, all relations created by the class would belong to the same Log-file. To allow for multiple Log-files with different headers, *_attr_dict* has to be an instance attribute. For the same reason the Log-file name is not a class attribute.

- **Task**: Since the primary keys are set automatically via *SERIAL* and the foreign keys have to be updated "manually" by using the method *fetch_fk*, one only has to fill the non-key attributes with data.

  Another case where key-attributes are avoided is the interface in User-mode. Since these IDs do not carry any meaning and are "only" used to join relations, the user should neither see them, nor have access to them. Therefore only non-key attributes are available in User-mode.
  - **Implementation**: First, the indices of the primary and foreign keys in the keys-attribute are determined. These indices are then excluded when creating a new list of attributes. The argument *is_sql* serves to determine whether the original attributes or the attributes with SQL-conform string should be used.

    ```
    def safe_attrs(self,is_sql=False):
        """Returns non-key attributes."""

        serial_indices = [self.keys.index(key) for key in self.keys
                            if key!=""]
        if (is_sql):
            attrs = [attr for attr in self._sql_attrs
                        if self._sql_attrs.index(attr) not in serial_indices]
        else:
            attrs = [attr for attr in self.attrs
                        if self.attrs.index(attr) not in serial_indices]
    ```

  - **Q**: Why are the non-key attributes not added as additional lists as attributes?
    A: The SQL-conform strings are already unnecessary and use up memory. To avoid any more memory going to waste, the removal of key-attributes is transferred into a function.
- **Task**: To add data to a relational database consisting of relations, the relations first have to be created. This is accomplished with CREATE queries.
  - **Implementation**: The primary key is extracted from the tuple keys and converted into an SQL-conform string. To fit the different entries properly into a query, they are first alternatingly combined into tuples using the *zip_longest* method of the *itertools* module (e. g. zip[(a,b),(c,d)] = (a,c),(b,d)). In contrast to the *zip* method, here the longest iterable determines the length of the resulting object. All "empty" parts of shorter iterables are filled with the *fillvalue* (e.g. zip_longest[(a,b,c),(d,e),fillvalue=h] = (a,d),(b,e),(c,h)). Next, the arguments are flattened into comma separated groups of space separated keywords (e.g. (a,d),(b,e),(c,h) a" "d, b" "e, c" "h).

Finally, the names and flattened arguments are inserted into the query SQL-string, where each argument of the format method is entered at the location of a set of empty braces while preserving the order. This gives the format expected for a query used to create a relation (e.g. CREATE TABLE IF NOT EXISTS <relation name> (<attr1> <type1> <cstrs1>, <attr2> <type2> <cstrs2>, ...)) which is used in the method *setup_relations*. Note that the clause "IF NOT EXISTS" prevents a relation from being created if it already exists.

```
def query_create(self):
    """Constructs query to CREATE the relation."""

    # get primary key
    pkey = tuple(key for key in self.keys if key=="PRIMARY KEY")
    sql_pkey = tuple(map(sql.SQL,pkey))
    # combine parts of arguments
    args_zip = tuple(itertools.zip_longest(self._sql_attrs, \
                                           self._sql_types, \
                                           sql_pkey, \
                                           self._sql_cstrs, \
                                           fillvalue=sql.SQL("")))
    args_flat = sql.Composed(sql.SQL(', ').join( \
                            [sql.SQL(' ').join(tpl) for tpl in args_zip]))

    query = sql.SQL("CREATE TABLE IF NOT EXISTS {} ({});") \
           .format(self._sql_name,args_flat)

    return query
```

- **Task**: After creating a relation, it has to be filled with data. This is accomplished with INSERT queries.
    - **Implementation**: Following the documentation of the psycopg2 sql module, queries can be assembled in a two-step process. First, relation names and attributes are entered at placeholders marked by empty braces. Furthermore, additional placeholders for actual values denoted by *sql.Placeholder()* given by *%s* are inserted at the last set of empty braces. The resulting query with placeholders (e.g. INSERT INTO <relation name> (<attr1>, <attr2>, ...) VALUES (%s, %s, ...)) can then be used together with actual data to fill the relation. It is used in the method *setup_relations*. Note that the clause "ON CONFLICT DO NOTHING" prevents a tuple (row) from being added twice. The clause "ON CONFLICT DO UPDATE" performs an update in case a constraint is violated. Here, the entire row is updated if the *UNIQUE* constraint of the *first_date_time* attribute is violated.

```
def query_insert(self):
    """Constructs query to INSERT tuples into the relation."""

    # get attributes not directly or indirectly set via serial
    attrs = self.safe_attrs(is_sql=True)

    if (self.name != "sessions"):
        # parent relations
        query = sql.SQL("""INSERT INTO {} ({}) VALUES ({})
                        ON CONFLICT DO NOTHING;""").format( \
                sql.Identifier(self.name), \
                sql.SQL(', ').join(attrs), \
                sql.SQL(', ').join(sql.Placeholder() * len(attrs)))
    else:
        # child relation
        query = sql.SQL("""INSERT INTO {} ({}) VALUES ({})
                        ON CONFLICT ({})
                        DO UPDATE SET ({}) = ({});""").format( \
                sql.Identifier(self.name), \
                sql.SQL(', ').join(attrs), \
                sql.SQL(', ').join(sql.Placeholder() * len(attrs)), \
                sql.SQL('first_date_time'), \
                sql.SQL(', ').join(attrs),
                sql.SQL(', ').join([sql.SQL('EXCLUDED.')+entry \
                        for entry in attrs]))

    return query
```

    - **Q**: Why does the sessions relation have the "ON CONFLICT DO UPDATE" clause while the others "DO NOTHING"?
      A: In contrast to the other relations, sessions has attributes which are updated over several lines of the Log-file. The clause is used to keep only the newest line.
- **Task**: To preserve referential integrity, i.e. agreement of values, of the primary key of one relation and a copy of it in another relation used for *JOIN*s, foreign key constraints are added to the relation.

    - **Implementation**: The strings specifying foreign keys are extracted from the tuple keys and split into the referenced relation and attribute. Next, the referenced attribute is used to identify the attribute acting as foreign key. Finally, all components are used to assemble the query (e.g. ALTER TABLE <relation name> ADD CONSTRAINT <constraint name> FOREIGN KEY (<foreign key attribute>) REFERENCES <referenced relation name> (<referenced primary key>)).

```
def fk_constraints(self):
    """Sets foreign key constraints for the relation."""

    # get foreign keys
    fkeys = tuple(key for key in self.keys if key!="PRIMARY KEY")
    for fkey in fkeys:
        # name, primary key of other relation
        pkey_rel,pkey_attr = fkey.split()
        # foreign key of self
        fkey_attr = self.attrs[self.keys.index(fkey)]

        query = sql.SQL("""ALTER TABLE {} ADD CONSTRAINT {}
                            FOREIGN KEY ({}) REFERENCES {} ({});""") \
               .format(self._sql_name, \
                       sql.Identifier("fk_"+fkey_attr), \
                       sql.Identifier(fkey_attr), \
                       sql.Identifier(pkey_rel), \
                       sql.Identifier(pkey_attr))

        yield query
```

    - **Q**: Why is the function a generator?
      A: There are multiple relations, each with possibly multiple foreign key constraints. To set all of these constraints for all relations, two loops are required. However a *Relation* object only knows about its own foreign keys, while the calling function knows about all relations. To restrict the keys to the scope of the *Relation* object, the loop over the foreign keys is kept inside and made accessbible by using a generator.
- **Task**: A single Log-file line fuels multiple SQL-relations. The entries of such a line thus have to be associated with the correct attribute of the correct relation. Also each line received from the Log-file may need to be modified.
    - **Implementation**: In *create_attr_dict*, the translation between columns from the processed Log-file and attributes of a relation is implemented as a dictionary. Given the column/attribute name, it returns the index of the entry in a row. It is created using the header of the processed Log-file.

      In *convert_line*, the line is sorted using the *_attr_dict.*

```
def create_attr_dict(self,src_attr):
    """
    Creates dictionary to sort lines of input with attributes
    src_attr to match attributes of the relation.
    """

    # get attributes not directly or indirectly set via serial
    attrs = self.safe_attrs(is_sql=False)

    self._attr_dict = {attr:src_attr.index(attr) for attr in attrs}


def convert_line(self,line):
    """
    Converts line of input to properly fit into the database.
    Task: Sorts line to match attributes of the relation.
    """

    line_sorted = [line[self._attr_dict[attr]] \
                   for attr in self._attr_dict.keys()]

    return line_sorted
```

- **Task**: To present the results of queries of the Test-suite in a clear manner, the table representations of relations are created and written to file or console. To allow further processing of the results of a query, a method to write them to a CSV file is added.
    - **Implementation**: The rich module is used to display the relations as tables. For compatibility reasons, the ouput is set to ASCII. The *create_table* method is used prior to both, the *print_table* and *write_table* methods. The *write_table* method is used in the *test_suite* method and *print_table* is used in the *interactive_queries* function. To write data to a CSV file, the csv module is used.

```
@staticmethod
def create_table(header,content):
    """
    Creates table representation of a query with header
    and content.
    """

    # setup
    table = Table(box=rich.box.ASCII)
    # columns
    for attr in header:
        table.add_column(attr)
    # add rows
    for entry in content:
        table.add_row(*entry)

    return table


@staticmethod
def print_table(table):
    """Displays table representation of a query."""

    console = Console()
    console.print(table)


@staticmethod
def write_table(table,fname):
    """Writes table representation of a query to file fname."""

    rich.print(table,file=fname)


@staticmethod
def export_csv(content,fname):
    """Export query as csv file."""

    try:
        with open(fname,"w",newline="") as csvfile:
            writer = csv.writer(csvfile,delimiter=",",quotechar="\"", \
                                quoting=csv.QUOTE_MINIMAL)
            for line in content:
                writer.writerow(line)
    except PermissionError:
        msg = f"Error: You lack permission to create {fname}."
        print(msg)
```

- **Q**: Why is this a static method?
  A: A static method is owned by the class which does not use any of its attributes or methods. It is used for utility functions for which it makes sense to associate them to a class. The functions *create_table*, *print_table* and *write_table* are used to represent a query result as a table for output to file or console. Since in PostgreSQL a query result can be saved as a *VIEW* and used as a relation, it makes sense to add these methods to the *Relation* class.

Decorator

- **Task**: Many methods establish a connection with the PostgreSQL-server and thus encounter the same errors. To handle these errors without repeating code, a decorator is used.
  - **Implementation**: A decorator is a wrapper around a function which modifies the wrapped functions behavior. The outer function name will be used for the decorator as *@check_db_connection_exists* and added above the definition of a function. To preserve the *__name__* and *__doc__* attributes of the wrapped function, the *@wraps* decorator of the functools module is used. Within the decorated function, catching the generic OperationalError informs the user of a failed connection to the database. This can have multiple reasons: the wrong SQL-user, the wrong password or a non-existing database. There do not seem to be more specific exceptions for the different reasons as stackoverflow1 and stackoverflow2 suggest.

```
def check_db_exists(function):
    """Decorator checking if the database exists."""

    @wraps(function)
    def decorated(*args):
        try:
            function(*args)
        except psycopg2.errors.OperationalError:
            msg = "Error: Cannot find database."
            print(msg)
    return decorated
```

Database class

The *Database* class has four public instance attributes:

- *self.name* (*str*): Name of the database
- *self.file* (LogFile object): Log-file object providing access to the processed Log-file
- *self.tests*(*str*): File name of the Test-suite output
- *self.relations* (*tuple* of *Relation* objects): Tuple of relations belonging to the database

```
def __init__(self,name,file,tests):
    """Constructs necessary attributes of the Database object."""

    self.name = name
    self.file = file
    self.tests = tests
    self.relations = None
```

- **Task**: Check if the credentials for the SQL-user are correcly provided.
    - **Implementation**: Try to establish a connection to the *postgres* database using the provided credentials. Return False if the credentials are insufficient.

    ```
    @staticmethod
    def check_credentials(sql_user):
        """Checks Username and Password."""

        connected = False
        try:
            conn = psycopg2.connect(dbname="postgres",
                                    host="localhost",
                                    port="5432",
                                    user=sql_user.name,
                                    password=sql_user.passwd)
            conn.close()
        except psycopg2.errors.OperationalError:
            msg = "Error: Wrong Username or Password."
            print(msg)
        else:
            connected = True

        return connected
    ```

    - **Q**: Why is this a static method?
      A: A static method is owned by the class which does not use any of its attributes or methods. It is used for utility functions for which it makes sense to associate them to a class. The *check_credentials* function checks the login credentials of the SQL-user by connecting to the *postgres* database, but does not need any attributes from the class. It thus checks these boxes.
- **Task**: Create a database.
    - **Implementation**: After establishing a connection to the PostgreSQL server, the query to create the database is assembled and executed. In case the database already exists, it is dropped. Note that to create a database, the query has to be treated as a transaction block by setting *conn.autocommit* to *True*.

    ```
    def create_database(self,sql_user):
        """Creates database."""

        conn = psycopg2.connect(dbname="postgres",
                                host="localhost",
                                port="5432",
                                user=sql_user.name,
                                password=sql_user.passwd)
        conn.autocommit = True
        cursor = conn.cursor()
    ```

```
    # remove database if it exists already
    query = sql.SQL("DROP DATABASE IF EXISTS {};") \
            .format(sql.Identifier(self.name))
    cursor.execute(query)
    # create database
    query = sql.SQL("CREATE DATABASE {};") \
            .format(sql.Identifier(self.name))
    cursor.execute(query)

    cursor.close()
    conn.close()
```

- **Q**: Why is the header and footer containing the setup and closure of the connection to the PostgreSQL-server not moved into a decorator?
  A: Moving the connection setup and closure into a decorator would be possible with a decorator with arguments discussed on this an d this website, if the *cursor* variable defined therein would not be needed in the wrapped function. To make the *cursor* variable accessible to the wrapped function, one would either have to work with unnamed arguments *args and **kwargs and extend kwargs in the decorator or add the cursor variable to the global namespace of the decorated function. The prior option makes the code hard to understand and the latter makes the code not threadsafe, which contradicts the ACID properties database transactions should possess. See more on stackoverflow. Since such a decorator would make the code in fact less readable, this is not pursued.

- **Task**: To create relations, all their names, attributes, attribute types, primary keys, foreign keys, other constraints and levels have to be set.
  - **Implementation**: The names, attributes, attribute types, primary keys, foreign keys, other constraints and levels of all relations to be created are separated into *tuples* and used to initialize the corresponding *Relation* objects. Note that this is an instance method since it sets the *relations* attribute of a *Database* object.

    To shorten the notation, constructs like *["NOT NULL"]*3 are used to create three entries of "NOT NULL" in a row. Note, that the tuples do not have to have the same length and only the attributes which should have constraints have to be specified. Attributes without constraints standing before attributes with constraints have to be entered as empty strings (e.g. a PRIMARY KEY already fulfills the *NOT NULL* and *UNIQUE* constraints). Note that tuples containing only one argument need a **trailing comma**. Otherwise they are not considered an iterable. Also note that the attribute names must match the names in the header, introduced in the method *process_log,* to associate the columns with the attributes. It is furthermore important that the "child" relations with foreign keys come **after** the relations with the corresponding primary keys within the *relations* list.

```
def initialize_relations(self):
    """Initializes relations for the database."""

    # initialize Relation objects
    # sessions
    name = "sessions"
    attrs = ("session_id","ip_id","user_id","pid","fail_count", \
             "login_status","first_date_time","last_date_time")
    types = ("SERIAL","INTEGER","INTEGER","INTEGER","INTEGER", \
             "TEXT","TIMESTAMP","TIMESTAMP")
    keys = ("PRIMARY KEY","ip_addresses ip_id","users user_id")
    cstrs = (*[""]*3,*["NOT NULL"]*3,"NOT NULL UNIQUE","NOT NULL")
    level = "child"
    sessions = Relation(name,attrs,types,keys,cstrs,level)
    # users
    name = "users"
    attrs = ("user_id","user_name","user_exists")
    types = ("SERIAL","TEXT","BOOLEAN")
    keys = ("PRIMARY KEY",)
    cstrs = ("","NOT NULL UNIQUE","NOT NULL")
    level = "parent"
    users = Relation(name,attrs,types,keys,cstrs,level)
    # ip_addresses
    name = "ip_addresses"
    attrs = ("ip_id","ip_address")
    types = ("SERIAL","INET")
    keys = ("PRIMARY KEY",)
    cstrs = ("","NOT NULL UNIQUE")
    level = "parent"
    ip_addresses = Relation(name,attrs,types,keys,cstrs,level)
    # relations
    self.relations = [users,ip_addresses,sessions]
```

  - **Q**: Why are the attributes "first_date_time", "user_name" and "ip_address" set to *UNIQUE*?
    A: As discussed in the motivation of the ERD, these three attributes are the logical primary keys. For technical reasons they are represented by integers. As logical primary keys they are set to be *UNIQUE* and *NOT NULL*.
  - **Q**: What is the purpose of the level-attribute?
    A: The purpose of the level-attribute is to identify relations with foreign keys which have to fetch their foreign key values from *SERIAL*

primary keys.

Background: When inserting data into the relations, all SERIAL IDs are automatically incremented, but their values are not automatically copied to the foreign keys which get the value *NULL*. The option *ON UPDATE CASCADE* only updates the foreign key upon an **update** of the primary key. Before that happens, the primary and foreign key have to be "connected" by having the same value. After a SERIAL increment of the primary keys, such a "connection" is not established. Within PostgreSQL, such an automatic update could be implemented using triggers and trigger functions. Here, the automatic copy-mechanism is implemented within Python in the method *fetch_fk*.

- **Task**: When using *SERIAL* primary keys, their values are automatically incremented upon an *INSERT*. However foreign keys intended to be linked to them are not updated automatically and first entered as *NULL*. To be able to join relations, foreign and primary key values have to be synchronized.
    - **Implementation**: Assume all relations have received an *INSERT*, the primary keys have been incremented and the foreign keys are *NULL*. Relations with foreign keys determined by a SERIAL primary key will in the following be called "child relations", while relations without such foreign keys will be called "parent relations". Assume furthermore, that the foreign keys of the child relation have the same name as the *SERIAL* primary keys of the parent relations.

      To UPDATE the NULL values of the foreign keys of a child relation at the last INSERT, one first needs the following:
        - (1) child relations name
        - (2) name of the foreign key of the child relation
        - (3) value of the parents primary key of the last *INSERT*
        - (4) name of the primary key of the child relation
        - (5) value of the primary key of the child relation for the last *INSERT*

      The *UPDATE* query reads: UPDATE (1) SET (2)=(3) WHERE (4)=(5);

      (1) is known, but the other information has to be collected. To get (4), the attribute name of the primary key is inferred. *(5)* is obtained by *SELECT*ing the primary key of the child relation, where the *UNIQUE* attribute has the value in the line just inserted. To get the just inserted value, the header is used.

      (2) is obtained by looping over the foreign keys and extracting the name. Since we assumed that (2) is the same as the name of the primary key of the parent relation, it is called *pkey_attr_parent*. (3) is obtained by *SELECT*ing the primary key of the parent relation, where the *UNIQUE* attribute has the value in the line just inserted. To get the just inserted value, the header is used again.

```
def fetch_fk(self,child_relation,header,line,cursor):
    """
    Fetch primary key values from parent relations
    to insert them into foreign keys of child relations.
    """

    if (child_relation.level=="child"):

        # get child_id from child relation
        pkey_attr_child = tuple(child_relation.attrs[ii]
                                for ii,key in enumerate(child_relation.keys)
                                if key=="PRIMARY KEY")[0]
        unique_attr = tuple(child_relation.attrs[ii]
                            for ii,cstr in enumerate(child_relation.cstrs)
                            if "UNIQUE" in cstr)[0]
        query = sql.SQL("SELECT {} FROM {} WHERE {} = {};").format( \
                sql.Identifier(pkey_attr_child), \
                sql.Identifier(child_relation.name), \
                sql.Identifier(unique_attr), \
                sql.Placeholder())
        unique_val = line[header.index(unique_attr)]
        cursor.execute(query,(unique_val,))
        child_id = cursor.fetchone()


        fkeys = tuple(key for key in child_relation.keys
                      if key!="PRIMARY KEY")
        for fkey in fkeys:
            rel_parent,pkey_attr_parent = fkey.split()

            # get foreign key IDs
            unique_attr = tuple(rel.attrs[ii] \
                                for rel in self.relations \
                                for ii,cstr in enumerate(rel.cstrs) \
                                if rel.name==rel_parent \
                                   and "UNIQUE" in cstr)[0]
            query = sql.SQL("SELECT {} FROM {} WHERE {} = {};").format( \
                    sql.Identifier(pkey_attr_parent), \
                    sql.Identifier(rel_parent), \
                    sql.Identifier(unique_attr), \
                    sql.Placeholder())
            unique_val = line[header.index(unique_attr)]
            cursor.execute(query,(unique_val,))
            parent_id = cursor.fetchone()

            # set foreign key IDs
            query = sql.SQL("UPDATE {} SET {} = {} WHERE {} = {};").format( \
                    sql.Identifier(child_relation.name), \
                    sql.Identifier(pkey_attr_parent), \
                    sql.Placeholder(), \
                    sql.Identifier(pkey_attr_child), \
                    sql.Placeholder())
            cursor.execute(query,(parent_id,child_id))
```

  - **Q**: Is it necessary to make this fetching procedure so complicated?
    A: While this procedure is rather dense, it is also generic and can be applied to all foreign keys of a relation at once. Compound keys may require additional modifications, but this procedure can be used to handle all "simple" distributions of *SERIAL* primary key values to foreign keys.
- **Task**: Create the relations of the database.
  - **Implementation**: First note that this method belongs solely to the Create-mode.

    After establishing a connection to the PostgreSQL server, the *buffer_time* (used in Append-mode, see method *process_log*) and *break_time* (used in Append-mode, see method *process_log*) are set to values far enough in the past to read the whole Log-file. Then the generator yielding the processed Log-file is started.

    First, for each relation separately, the header is used to create the dictionary to translate between the processed Log-file and the attributes of each relation. Next, the relation is created using the query obtained from the method *query_create*. Afterwards, the rest of the processed Log-file is read. For each relation, the currently read line is converted to fit the relation attributes. Then the converted line is inserted in the relation using the method *query_insert*. To update foreign keys linked to *SERIAL* primary keys, the method *fetch_fk* is used.

    Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_exists
def setup_relations(self,sql_user,user_list):
    """Creates and fills relations."""

    conn = psycopg2.connect(dbname=self.name,
                            host="localhost",
                            port="5432",
                            user=sql_user.name,
                            password=sql_user.passwd)
    cursor = conn.cursor()

    # dummy value for time of last database entry
    buffer_time = datetime.now() - timedelta(days=14)
    # dummy value for lifetime of a ssh login session
    break_time = datetime.now() - timedelta(days=14)

    # start generator
    log_processed = self.file.process_log(user_list, \
                                          buffer_time,break_time)

    # create relations
    header = next(log_processed)
    for relation in self.relations:
        relation.create_attr_dict(header)
        query = relation.query_create()
        cursor.execute(query)

    # add constraints
    for relation in self.relations:
        for query in relation.fk_constraints():
            cursor.execute(query)

    # fill relations
    for line in log_processed:
        for relation in self.relations:
            line_converted = relation.convert_line(line)
            query = relation.query_insert()
            cursor.execute(query,line_converted)
            # fetch primary key values to foreign keys
            self.fetch_fk(relation,header,line,cursor)

    conn.commit()
    cursor.close()
    conn.close()
```

- **Q**: Why is the option *conn.autocommit=True* not used?
  A: The autocommit option would "save" the creation of a table and each line after it is inserted into the table. Besides not loosing everything upon a crash, it would help with debugging to see, after which line a crash occurs. Here, it is left out since it is not strictly necessary in contrast to the method create_database. Furthermore, the checkLogins script does not provide a recovery from a crash and the database would have to be re-created anyway.
- **Q**: Why is the dictionary to translate between header and attributes not only created once for the entire *Relation* class?
  A: Defining the dictionary as class attribute would link all *Relation* objects to the same Log-file. In case there are multiple Log-files, another class would be required. To be able to use one class for multiple Log-files, the dictionary either has to be an instance attribute or a list of dictionaries for each Log-file. For the current purpose, the instance attribute is simplest and does not pose a bottleneck.

- **Task**: Append data to the database without having to re-read the entire Log-file or previous Log-files. Allow for a scalable solution, i.e. be aware of RAM restrictions.
  - **Implementation**: After establishing a connection to the PostgreSQL server, the *buffer_time* and *break_time* are set. The *break_time* marks the entry in the database with the last *last_date_time TIMESTAMP*. Using the default parameters of sshd, the maximum lifetime of an ssh login-session is estimated to determine the *buffer_time*, which is this lifetime before the *break_time*. The *buffer_time* is used in the method *process_log* to skip lines of already terminated ssh login-sessions, while the *break_time* serves to skip lines of possibly still running ssh-logins which are already contained in the database. Using these times, the generator yielding the processed Log-file is started.

    First, for each relation separately, the header is used to create the dictionary to translate between the processed Log-file and the attributes of each relation. Afterwards, the part of the processed Log-file determined by the *buffer_time* and *break_time* is read. For each relation, the currently read line is converted to fit the relation attributes. Then the converted line is inserted in the relation using the method *query_insert*. To update foreign keys linked to *SERIAL* primary keys, the method *fetch_fk* is used.

    Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_exists
def append(self,sql_user,user_list):
    """Appends data to the database."""

    conn = psycopg2.connect(dbname=self.name,
                            host="localhost",
                            port="5432",
                            user=sql_user.name,
                            password=sql_user.passwd)
    cursor = conn.cursor()

    # time of last database entry
    query = """
            SELECT last_date_time from sessions
            WHERE last_date_time =
            (SELECT MAX(last_date_time) FROM sessions);
            """
    cursor.execute(query)
    break_time = cursor.fetchone()[0]

    # default parameters for ssh login-session
    # (see man sshd_config, CamelCase at underscore)
    max_auth_tries = 6
    login_grace_time = timedelta(minutes=2)
    delays = timedelta(minutes=1)
    dt = login_grace_time * max_auth_tries + delays
    # lifetime of a ssh login session before
    # time of last database entry
    buffer_time = break_time - dt

    # start generator
    log_processed = self.file.process_log(user_list, \
                                          buffer_time,break_time)

    # create association between Log file and relations
    header = next(log_processed)
    for relation in self.relations:
        relation.create_attr_dict(header)
    # append to relations
    for line in log_processed:
        for relation in self.relations:
            line_converted = relation.convert_line(line)
            query = relation.query_insert()
            cursor.execute(query,line_converted)

            # fetch primary key values to foreign keys
            self.fetch_fk(relation,header,line,cursor)

    conn.commit()
    cursor.close()
    conn.close()
```

- **Q**: Why are two times required to fast-forward the processed Log-file?
  A: While the role of the *break_time* is clear, the *buffer_time* less straightforward. Inside the method *process_log*, initial and cumulative values of the sessions are stored in a dictionary. To complete unterminated ssh login-sessions during the previous run of the Create-/Append-mode, the dicitonaries have to be populated sufficiently.

- **Task**: Provide a simple interface for the user to access the database.
  - **Implementation**: After establishing a connection to the PostgreSQL server, the user is greeted with a set of instructions from the *_if_instructions* method on how to use User-mode. These instructions list the available attributes and operators to restrict the data shown. Next, the input from the user is read in the *_if_read* method. To quit the script, the user can press q+Enter. The *_if_export* method encountered next exports the data of the previously run query as a CSV-file if the user has requested it.

    Then the query of the user is processed. The next four methods extract statements from the input:

    - *_if_count*: Extract *COUNT*-statements.
    - *_if_minmax*: Extract *MIN*- and *MAX*-statements.
    - *_if_sort*: Extract *ASC*- and *DESC*-statements.
    - *_if_where*: Extract filters.

    After extracting the attributes provided by the user, the actual query is constructed in the method *_if_assemble_query*. Finally, the query is executed.

    Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_exists
def interface(self,sql_user):
    """User command line interface."""

    conn = psycopg2.connect(dbname=self.name,
                            host="localhost",
                            port="5432",
                            user=sql_user.name,
                            password=sql_user.passwd)
    cursor = conn.cursor()

    input_quit = ""
    header = ()
    response = []

    while (input_quit!="q"):

        self._if_instructions()

        input_flat,input_quit = self._if_read(input_quit)
        if (input_quit=="q"):
            continue

        exported = self._if_export(input_flat,header,response)
        if (exported):
            continue

        input_flat,count_exist = self._if_count(input_flat)
        input_flat,minmax_exist,minmax_clause = self._if_minmax(input_flat)
        input_flat,sort_clause = self._if_sort(input_flat)
        input_flat,where_clause = self._if_where(input_flat, \
                                                 minmax_exist, \
                                                 minmax_clause)

        # unique attributes
        user_attrs = list(set([flt[0] for flt in input_flat]))
        query = self._if_assemble_query(user_attrs,where_clause, \
                                        sort_clause,count_exist)

        # execute query
        cursor.execute('SAVEPOINT sp;')
        try:
            cursor.execute(query)
        except psycopg2.errors.UndefinedColumn:
            msg = "Error: Cannot find attribute."
            print(msg)
            cursor.execute('ROLLBACK TO SAVEPOINT sp;')
        except psycopg2.errors.UndefinedTable:
            msg = "Error: Cannot find table."
            print(msg)
            cursor.execute('ROLLBACK TO SAVEPOINT sp;')
        else:
            # display table
            header = tuple(name[0] for name in cursor.description)
            response = [tuple(map(str,entry)) for entry in cursor.fetchall()]
            table = Relation.create_table(header,response)
            Relation.print_table(table)

    conn.commit()
    cursor.close()
    conn.close()
```

- **Q**: Can the different extraction methods be interchanged arbitrarily or are they in a fixed sequence?
  A: Some can be moved, but not all of them. The method *_if_sort* can be moved arbitrarily, while *_if_count* and *_if_minmax* have to be called before *_if_where*. However this is due to nested operations.

- **Task**: Provide instructions to the user on how to request data in User-mode.
  - **Implementation**: First, all attributes excluding artificial primary and foreign keys, i.e. integer keys without any meaning but to join relations, are gathered. Then the available relations are listed to the user, followed by the operators available.

```
def _if_instructions(self):
    """Instructions for interface."""

    # get attributes not directly or indirectly set via serial
    attrs_avail = []
    for relation in self.relations:
        attrs = relation.safe_attrs(is_sql=False)
        attrs_avail = [*attrs_avail,*attrs]
    # separate for formatting
    attrs_format = [attrs_avail[ii:min(ii+4,len(attrs_avail))]
                    for ii in range(0,len(attrs_avail),4)]

    # instructions
    print("\nAvailable attributes:\n"+ \
          ",\n".join([", ".join(attr)
                      for attr in attrs_format]))
    filters = """
            Available filters:
            > Restrict arguments: arg <, >, <=, >=, =, != value
            > Range: arg<value_upper, arg>value_lower
            > Simple functions: max(arg), min(arg)
            > Counting: count(arg), count(arg=value)
            > Sorting: asc(arg), desc(arg)
            > Date: 'yyyy-mm-dd'
            > Time: 'HH:[MM:[SS]]'
            > String value: 'value'
            > Case-sensitive Regex: arg ~ regex
            """
    print(textwrap.dedent(filters))
    print("Export previous output to csv: export filename")
    print("Syntax: statement_1, statement_2, ... statment_n;\n")
    print("Press q+Enter to quit.\n")
```

- **Task**: Read the input from the user in User-mode and store it in a structure to enable further processing.

- **Implementation**: The comma-separated statements by the user are read in a while loop until a semi-colon is enountered. Each new line is stored as a string in a list.

  During processing, first anything past the semi-colon is stripped-off. Then the lines are joined together inserting whitespace between them. The whole string is then split at commas, giving a list of the statements. Next each element of the list is stripped of leading and trailing whitespace. To have a homogeneous data structure, each entry is then turned into a list with a string as single element. Since this string may correspond to a statement like ["arg=val"], it will be further processed into a list ["arg","val","="]. A statement consisting only of an attribute will be of the same structure ["attr"]. If the elements were kept as strings, a statement with an operator would turn into a list, while a single attribute would stay a string.

```
@staticmethod
def _if_read(input_quit):
    """Read input from user."""

    input_list = []
    input_flat = []
    while True:
        line = input("??? ")
        input_list.append(line)

        # quit
        if (input_list[0]=="q"):
            input_quit = "q"
            break

        # collect input
        if (";" in line):
            line = line[:line.index(";")]
            input_list[-1] = line
            # list of list with statements
            input_flat = list(
                                map(str.strip, \
                                    " ".join(input_list).split(",") \
                                ) \
                            )
            input_flat = [[flt] for flt in input_flat]
            break

    return input_flat, input_quit
```

- **Task**: Export the last query by the user as a CSV-file.
  - **Implementation**: If the user types the word "export" followed by a name in the first statement block, the query of the previous run is written to file formatted as CSV. It is ensured that the filetype is csv. Leading periods denoting hidden files are stripped away. Exports before the first query are prevented and the user is informed that a query is necessary to be exported. Invalid, i.e. empty filenames are prevented as well.

```
@staticmethod
def _if_export(input_flat,header,response):
    """Export last query as csv file."""

    exported = False
    if ("export" in input_flat[0][0]):
        fname = input_flat[0][0].split()[1].lstrip(".")
        file_type = ""
        pattern = r"\.csv$"
        search = re.search(pattern,fname)
        if (not bool(search)):
            file_type = ".csv"

        if (not bool(header) or not bool(response)):
            print(f"Cannot export to {fname+file_type} "+ \
                    "without query.")
        elif (fname==""):
            print("Invalid filename.")
        else:
            Relation.export_csv([header,*response],fname+file_type)
            print(f"Query exported as {fname+file_type}.")

        exported = True

    return exported
```

- **Task**: Extract *COUNT*-statements from the user input.
  - **Implementation**: First, using the keyword "count" in *cts*, any filter *flt* containing "count(" is converted from ["count(arg)"] to ["arg","count"] and entered as value in a dictionary with its index in *input_flat* as key. Next a flag denoting the existence of a COUNT statement is set. Finally, the contents of the dictionary are entered in *input_flat* at the correct index.

```
@staticmethod
def _if_count(input_flat):
    """Extract count statements from input."""

    cts = "count"
    count_dict = {input_flat.index(flt): \
                  [re.search(cts+r"\((.+?)\)",flt[0]).group(1), \
                   cts] \
                  for flt in input_flat \
                  if cts+"(" in flt[0]}
    count_exist = bool(count_dict)

    for key in count_dict.keys():
        input_flat[key] = [count_dict[key][0]]

    return input_flat, count_exist
```

- **Task**: Extract *MIN*- and *MAX*-statements from the user input.
  - **Implementation**: First, using the keywords "min" and "max" in *minmax*, any filter *flt* containing "min(" or "max(" is converted from ["min(arg)"] to ["arg","min"] or ["max(arg)"] to ["arg","max"] and entered as value in a dictionary with its index in *input_flat* as key. Next a flag denoting the existence of a *MIN*- or *MAX*-statement is set. Then, the contents of the dictionary are entered in *input_flat* at the correct index. In case a minmax statement was encountered, part of an SQL-query called *minmax_clause* is created. While the *MIN*- and *MAX*-statements in SQL are aggregate statements, here they are modified to limit all output to those where the specified attribute has its maximum or minimum value. In case of multiple minmax statements, all entries conforming to any of the minmax statements are displayed.

```
def _if_minmax(self,input_flat):
    """Extract minmax statements from input."""

    minmax = ("min","max")
    minmax_dict = {input_flat.index(flt): \
                   [re.search(mm+r"\((.+?)\)",flt[0]).group(1), \
                    mm] \
                   for flt in input_flat \
                   for mm in minmax \
                   if mm+"(" in flt[0]}
    minmax_exist = bool(minmax_dict)

    for key in minmax_dict.keys():
        input_flat[key] = [minmax_dict[key][0]]

    if (minmax_exist):

        minmax_lst = [sql.SQL("{} = (SELECT {}({}) FROM {})").format( \
                      sql.Identifier(mm[0]), \
                      sql.SQL(mm[1].upper()), \
                      sql.Identifier(mm[0]), \
                      sql.Identifier(relation.name)) \
                      for relation in self.relations \
                      for mm in minmax_dict.values() \
                      if mm[0] in relation.safe_attrs(is_sql=False)]

        minmax_clause = sql.SQL(' OR ').join(minmax_lst)
    else:
        minmax_clause = sql.SQL('')

    return input_flat,minmax_exist,minmax_clause
```

- **Task**: Extract *ASC*- and *DESC*-statements from the user input.
  - **Implementation**: First, using the keywords "asc" and "desc" in *sorts*, any filter *flt* containing "asc(" or "desc(" is converted from ["asc(arg)"] to ["asc","min"] or ["desc(arg)"] to ["desc","max"] and entered as value in a dictionary with its index in *input_flat* as key. Next a flag denoting the existence of a *ASC*- or *DESC*-statement is set. Then, the contents of the dictionary are entered in *input_flat* at the correct index. In case a sorts statement was encountered, part of an SQL-query called *sort_clause* is created.

```
@staticmethod
def _if_sort(input_flat):
    """Extract sort statements from input."""

    sorts = ("asc","desc")
    sort_dict = {input_flat.index(flt): \
                 [re.search(sort+r"\((.+?)\)",flt[0]).group(1), \
                  sort] \
                 for flt in input_flat \
                 for sort in sorts \
                 if sort+"(" in flt[0]}
    sort_exist = bool(sort_dict)

    for key in sort_dict.keys():
        input_flat[key] = [sort_dict[key][0]]

    if (sort_exist):

        sorting = [sql.SQL("ORDER BY {} {}").format( \
                   sql.Identifier(sort[0]), \
                   sql.SQL(sort[1].upper())) \
                   for sort in sort_dict.values()]

        sort_clause = sql.SQL(', ').join(sorting)
    else:
        sort_clause = sql.SQL("")

    return input_flat,sort_clause
```

- **Task**: Extract *WHERE*-statements from the user input.
    - **Implementation**: First, using the binary operators in *ops*, any filter *flt* containing one is converted from ["arg1 op arg2"] to ["arg1"," arg2","op"] and entered as value in a dictionary with its index in *input_flat* as key. Next a flag denoting the existence of a *WHERE*-statement is set. Then, the contents of the dictionary are entered in *input_flat* at the correct index. In case a where statement was encountered, part of an SQL-query is created. If a minmax statement was also in the input, it is incorporated in the where-statement. If there are only minmax statements, part of their query is modified. In all cases the part of an SQL-query is called *where_clause*.

```
@staticmethod
def _if_where(input_flat,minmax_exist,minmax_clause):
    """
    Extract where statements from input.
    Includes minmax statements.
    """

    ops = ("<",">","<=",">=","=","!=","~")
    # decompose filters
    where_dict = {input_flat.index(flt): [*flt[0].split(op),op] \
                  for flt in input_flat \
                  for op in ops \
                  if op in flt[0]}
    where_exist = bool(where_dict)

    for key in where_dict.keys():
        input_flat[key] = where_dict[key]

    if (where_exist):
        comparisons = [sql.SQL("{} {} {}").format( \
                       sql.Identifier(flt[0]), \
                       sql.SQL(flt[2]), \
                       sql.SQL(flt[1])) \
                       for flt in input_flat if (len(flt)==3)]

        if (minmax_exist):
            where_clause = sql.SQL("WHERE {} AND ({})").format( \
                           sql.SQL(' AND ').join(comparisons), \
                           minmax_clause)
        else:
            where_clause = sql.SQL("WHERE {}").format( \
                           sql.SQL(' AND ').join(comparisons))
    elif (minmax_exist):
        where_clause = sql.SQL("WHERE ({})").format( \
                       minmax_clause)
    else:
        where_clause = sql.SQL("")

    return input_flat,where_clause
```

- **Task**: Assemble the query posed by the user to pass to PostgreSQL.
  - **Implementation**: Using the list *user_attrs* of attributes provided by the user, each attribute is prepended by its relation and a period to get the format "relation.attribute" which are collected in the list *select_args*. At the same time, the relations containing the attributes are collected in the list *user_relations*.

    In case a *COUNT*-statement is contained in the input, the *sort_clause* is "removed" and the *select_args* are replaced by a *COUNT*(*).

    Next, the actual query is assembled using the already prepared *select_args*, *where_clause* and *sort_clause*. Using the number of *user_relations*, it is determined whether *JOIN*s are necessary or not. It is straightforward to assemble the query for a single involved relation. In case *JOIN*s are necessary it is exploited that any *JOIN* has to involve the *sessions* relation. Since the foreign keys of the *sessions* relation contain the relation names and attributes used for *JOIN*s, the foreign keys only have to be filtered using the relations in *user_relations*. Compared to the case of a single relation, the query "only" involves the *join_clause* in addition.

```
def _if_assemble_query(self,user_attrs,where_clause, \
                        sort_clause,count_exist):
    """Assemble query based on clauses."""

    # list of attributes with associated relations
    # in format "relation.attribute"
    user_relations = []
    select_args = []
    for relation in self.relations:
        user_relation_attrs = [sql.Identifier(relation.name,attr) \
                                for attr in user_attrs \
                                if attr in relation.safe_attrs(is_sql=False)]
        if (bool(user_relation_attrs)):
            user_relations.append(relation.name)
            select_args = [*select_args,*user_relation_attrs]

    # modify if count present
    if (count_exist):
        sort_clause = sql.SQL("")
        select_args = [sql.SQL("COUNT(*)")]

    # assemble query
    if (len(user_relations)==1):
        query = sql.SQL("SELECT {} FROM {} {} {};").format( \
                sql.SQL(', ').join(select_args), \
                sql.Identifier(user_relations[0]), \
                where_clause, \
                sort_clause)
    elif (len(user_relations)>1):
        sessions = [relation for relation in self.relations \
                    if relation.name=="sessions"][0]
        join_clause = [sql.SQL(" INNER JOIN {} ON {} = {}").format( \
                       sql.Identifier(key.split()[0]), \
                       sql.Identifier(key.split()[0],key.split()[1]), \
                       sql.Identifier('sessions',key.split()[1])) \
                       for key in sessions.keys \
                       if key.split()[0] in user_relations]

        query = sql.SQL("SELECT {} FROM {} {} {} {};").format( \
                sql.SQL(', ').join(select_args), \
                sql.Identifier('sessions'), \
                sql.SQL(' ').join(join_clause), \
                where_clause, \
                sort_clause)
    else:
        query = sql.SQL("SELECT unknown_attr FROM sessions")

    return query
```

    - **Q**: How could the *join_clause* be made more generic?
      A: Using the level-attributes, one should be able to design the *join_clause* more generic.

- **Task**: To provide the user with a possibility to check if the checkLogins script fulfilled its task, a Test-suite of several queries is assembled and executed.
    - **Implementation**: The Test-suite consists of a list of short task descriptions and a list of queries to address them. Each query is executed one after another and the output consists of the task description, the executed query as text and the output as ASCII table. The indentation of the task description and queries is removed using the *dedent* method of the *textwrap* module. To illustrate the export into a CSV file, the results of the first query are written to file.

      In case the user does not have permission to write to file, they are informed. Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_exists
def test_suite(self,sql_user):
    """
    Provides sample queries and their output to verify
    the created database.
    """

    tasks = ["All relevant information", \
             "All existing Users", \
             "Fail-counts for users and IP-addresses"]

    queries = ["""
                SELECT sessions.pid, users.user_name, users.user_exists,
                ip_addresses.ip_address,
                sessions.first_date_time, sessions.last_date_time,
                sessions.fail_count, sessions.login_status
                FROM sessions
                INNER JOIN users ON sessions.user_id = users.user_id
                INNER JOIN ip_addresses ON sessions.ip_id = ip_addresses.ip_id;
                """,
                """
                SELECT sessions.pid, users.user_name, ip_addresses.ip_address,
                sessions.first_date_time, sessions.last_date_time
                FROM sessions
                INNER JOIN users ON sessions.user_id = users.user_id
                INNER JOIN ip_addresses ON sessions.ip_id = ip_addresses.ip_id
                WHERE users.user_exists IS TRUE;
                """,
                """
                SELECT users.user_name, ip_addresses.ip_address,
                sessions.fail_count, users.user_exists
                FROM sessions
                JOIN users ON sessions.user_id = users.user_id
                JOIN ip_addresses ON sessions.ip_id = ip_addresses.ip_id;
                """]

    conn = psycopg2.connect(dbname=self.name,
                            host="localhost",
                            port="5432",
                            user=sql_user.name,
                            password=sql_user.passwd)
    cursor = conn.cursor()

    # table representation
    try:
        with open(self.tests,"w",newline="") as test_file:

            for ii in range(len(tasks)):
                test_file.write("\n" \
                               +textwrap.dedent(tasks[ii]).strip() \
                               +":\n")
                cursor.execute(queries[ii])
                header = tuple(name[0] for name in cursor.description)
                response = [tuple(map(str,entry)) \
                            for entry in cursor.fetchall()]
                table = Relation.create_table(header,response)
                test_file.write(textwrap.dedent(queries[ii])+"\n")
                Relation.write_table(table,test_file)
    except PermissionError:
        msg = f"Error: You lack permission to create {fname}."
        print(msg)

    # csv file
    cursor.execute(queries[0])
    header = list(name[0] for name in cursor.description)
    response = [list(map(str,entry)) \
                for entry in cursor.fetchall()]
    Relation.export_csv([header,*response], \
                        self.tests.split(".")[0]+".csv")

    conn.commit()
    cursor.close()
    conn.close()
```

Interactive-Mode

- **Task**: Allow the user direct access to the created database.
  - **Implementation**: After establishing a connection to the PostgreSQL server, the user can enter multi-line SQL-queries using lists and the *join* method of strings. Note that with the readline module also corrections using the arrow keys are possible for the user. Typos in attributes and relations are handled and after the user is done, they can quit.

    Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_exists
def interactive_queries(sql_user,db_name):
    """Run queries interactively."""

    conn = psycopg2.connect(dbname=db_name,
                            host="localhost",
                            port="5432",
                            user=sql_user.name,
                            password=sql_user.passwd)
    cursor = conn.cursor()

    input_quit = ""
    while (input_quit!="q"):

        # read query
        print("\nEnter query or q+Enter to quit:")
        input_list = []
        while True:
            line = input()
            input_list.append(line)
            # quit
            if (input_list[0]=="q"):
                input_quit = "q"
                break
            # collect input
            if (";" in line):
                line = line[:line.index(";")+1]
                input_list[-1] = line
                break
            input_list = []
        if (input_quit=="q"):
            continue

        query = " ".join(input_list)

        # execute query
        cursor.execute('SAVEPOINT sp;')
        try:
            cursor.execute(query)
        except psycopg2.errors.UndefinedColumn:
            msg = "Error: Cannot find attribute."
            print(msg)
            cursor.execute('ROLLBACK TO SAVEPOINT sp;')
        except psycopg2.errors.UndefinedTable:
            msg = "Error: Cannot find table."
            print(msg)
            cursor.execute('ROLLBACK TO SAVEPOINT sp;')
        else:
            # display table
            header = tuple(name[0] for name in cursor.description)
            response = [tuple(map(str,entry)) for entry in cursor.fetchall()]
            table = Relation.create_table(header,response)
            Relation.print_table(table)

    conn.commit()
    cursor.close()
    conn.close()
```

- **Q**: Why is the option *conn.autocommit=True* not used?
  A: The autocommit option would "save" the creation of a table and each line after it is inserted into the table. Besides not loosing everything upon a crash, it would help with debugging to see, after which line a crash occurs. Here, it is left out since it is not strictly necessary in contrast to the method *create_database*. Furthermore, the checkLogins script does not provide a recovery from a crash and the database would have to be re-created anyway.

SQLUser class

The SQLUser class has two public instance attributes:

- *self.name (str)*: Name of the user
- *self.passwd (str)*: Password of the user

- **Task**: To connect to a PostgreSQL-server, a SQL-user and their password are required.
  - **Implementation**: The SQLUser class serves to obtain and store the SQL-user and their password, either interactively or from environment variables. The latter option becomes necessary to run the script via a cronjob.

```
class SQLUser:
...
    def __init__(self):
        """Constructs all necessary attributes for the User object."""

        self.name = ""
        self.passwd = ""


    def get_login(self):
        """Gets login data for SQL server."""

        self.name = input("Username: ")
        self.passwd = getpass("Password: ")


    def get_login_env(self):
        """Gets login data for SQL server from environment variables."""

        self.name = os.getenv('CHECK_LOGIN_USR')
        self.passwd = os.getenv('CHECK_LOGIN_PWD')
```

User class

The User class has one "private" and two public instance attribute:

- *self._name (str)*: Name of the user
- *self.home (str)*: Home-directory of the user
- *self.list (list of str)*: Names of users on the system

```
def __init__(self):
    """Constructs all necessary attributes for the User object."""

    cmd = "whoami"
    self._name = subprocess.check_output(cmd,shell=True,text=True) \
                        .strip()
```

```
if (self._name == "root"):
    # root user: logs in accessible directory
    self.home = "/home"
else:
    # regular user: logs in own home-directory
    self.home = f"/home/{self._name}"

# list of users
shell_blacklist = ["nologin","false"]
user_blacklist = ["sync","postgres"]
self.list = []
fname = "/etc/passwd"
with open(fname,newline="") as userfile:
    reader = csv.reader(userfile, delimiter = ":")
    for line in reader:
        if ((not any([shell in line[6] \
                for shell in shell_blacklist]))
                and (not any([user in line[0] \
                for user in user_blacklist]))):
            self.list.append(line[0])
```

- **Q**: Why is not the range of UIDs (larger and equal 1000) used to infer which user is legitimate and human?
  A: The range of UIDs is only a convention. The login-shell is a better criterion to identify human users.

Planned features

- **Setup-Mode**: Enable running the script as cronjob in Append-mode.
- **GUI**: Port the User-mode into a GUI.
- **Generalization**: Extend the script to organize Logs from other services (e.g. Cron, Telnet).
- **Complexity**: Allow for compound keys.