Laboratorul 5: SQLite, interfețe grafice și comunicarea prin cozi de mesaje

Introducere

Observație: Cozile de mesaje de tip System-V sunt disponibile doar pe sisteme de operare UNIX.

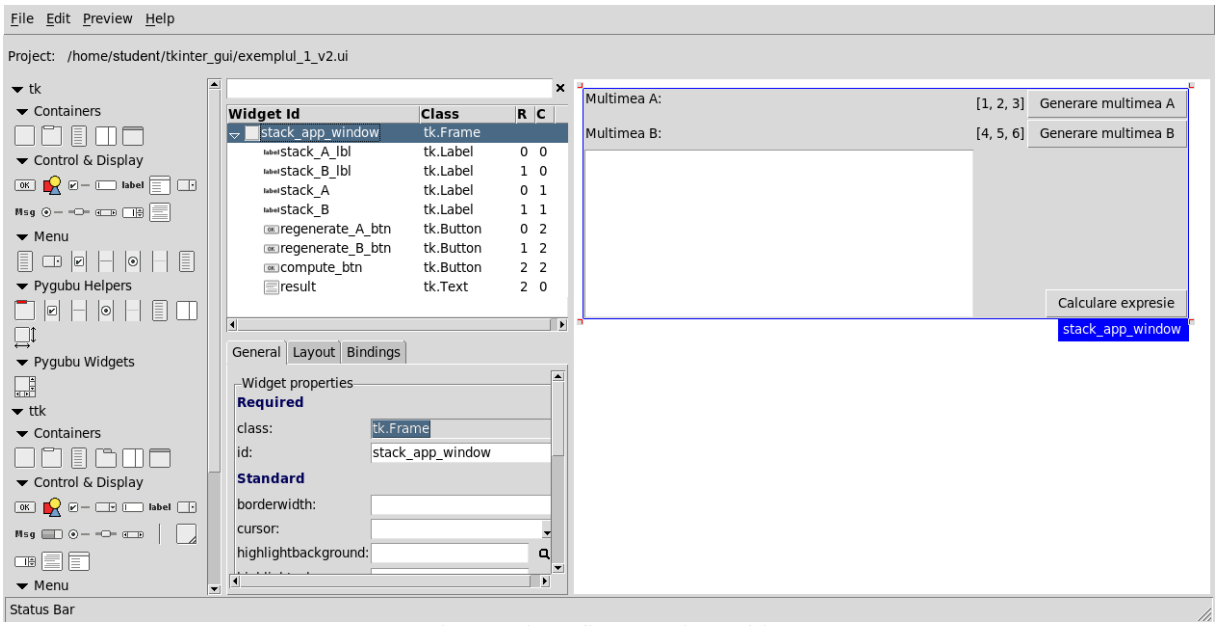
GUI cu Tkinter

Pentru implementarea unei interfețe grafice utilizând Tkinter, vezi cursul 5 de la disciplina Paradigme de Programare. De asemenea, vezi:

- https://tkdocs.com/tutorial/index.html
- https://docs.python.org/3/library/tk.html
- https://pythonbasics.org/tkinter/.
- https://github.com/alejandroautalan/pygubu-designer

Pentru o soluție de tip Drag-and-Drop (Tkinter), se va deschide un terminal și se vor executa următoarele comenzi:

```
sudo apt install python3-pip # pip package manager for python packages
sudo apt install python3-tk  # install tkinter
sudo pip3 install pygubu pygubu-designer # GUI designer for Tkinter
pygubu-designer # run the designer
```



Designer-ul grafic PyGubu (Tkinter)

GUI cu PyQt5

Pentru implementarea unei interfețe grafice utilizând PyQt5, vezi:

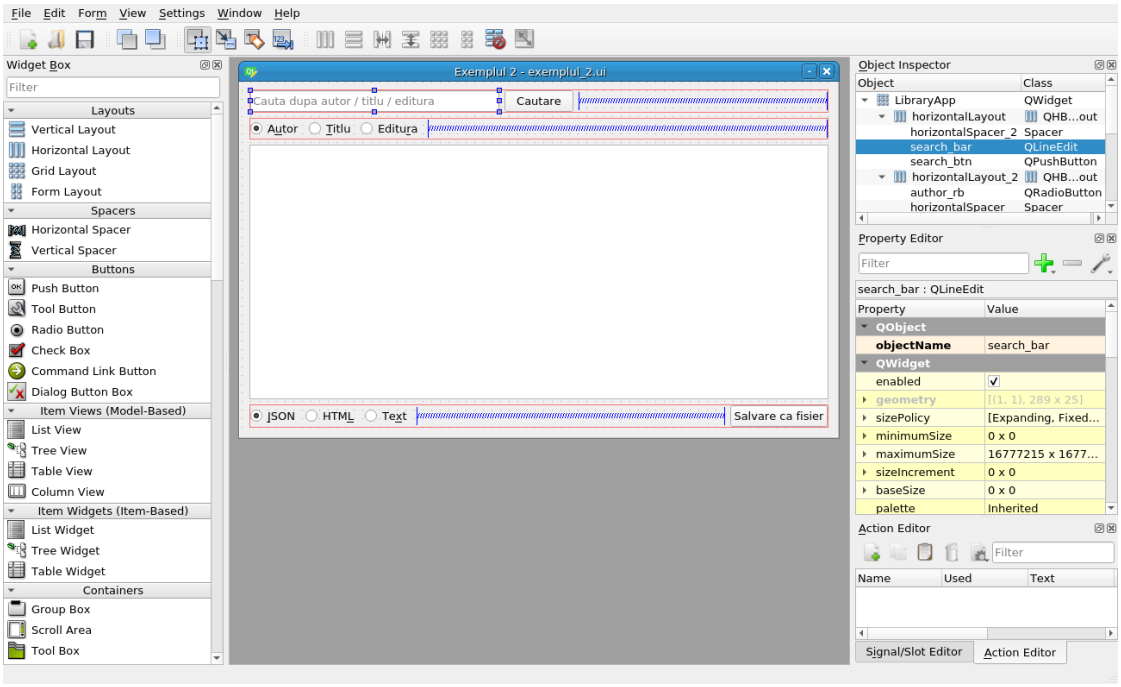
- https://www.riverbankcomputing.com/static/Docs/PyQt5/
- https://pythonspot.com/pyqt5/
- https://likegeeks.com/pyqt5-tutorial/

Pentru un designer grafic (PyQt5), se vor executa în terminal comenzile:

```
sudo apt install libxkbcommon-x11-0
sudo apt install libqt5gui5
wget http://ftp.ro.debian.org/debian/pool/main/x/xcb-util/libxcb-util1_0.4.0-1+b1_amd64.deb
sudo dpkg -i libxcb-util1_0.4.0-1+b1_amd64.deb
```

```
wget http://download.qt.io/official_releases/online_installers/qt-unified-linux-x64-online.run
chmod a+x qt-unified-linux-x64-online.run
./qt-unified-linux-x64-online.run
```

Observație: folosind comenzile de mai sus se pot instala ambele versiuni de Qt care apar în installer (5 și 6).



Designer-ul grafic Qt5 Designer (PyQt5)

Cozi de mesaje

O coadă de mesaje este utilizată pentru comunicarea între procese, sau între firele de execuție (thread-urile) aceluiași proces. Acestea oferă un protocol de comunicare asincron în care emițătorul și receptorul nu au nevoie să interacționeze în același timp (mesajele sunt reținute în coadă până când destinatarul le citește)

Avantajele utilizării cozilor de mesaje:

1. redundanță - procesele trebuie să confirme citirea mesajului și faptul că acesta poate fi eliminat din coadă
2. vârfuri de trafic (traffic spikes) - adăugarea în coadă previne aceste spike-uri, asigurând stocarea datelor în coadă și procesarea lor (chiar dacă va dura mai mult)
3. mesaje asincrone
4. îmbunătățirea scalabilității
5. garantarea faptului că tranzacția se execută o dată
6. monitorizarea elementelor din coadă

SQLite3

Pentru instalare, se execută în terminal:

```
sudo apt install sqlite3 sqlitebrowser
```

sqlitebrowser este un GUI pentru vizualizarea unei baze de date sqlite. Acesta poate fi pornit executând în terminal comanda:

```
sqlitebrowser
```

Documentația oficială este disponibilă la adresa: https://docs.python.org/3/library/sqlite3.html
Mai jos se regăsește un mic exemplu cu operațiile CRUD:

```
import os
import sqlite3
from collections import namedtuple

Book = namedtuple("Book", ["id", "title", "author", "publisher"])


class DatabaseManager:
    CREATE_CMD = '''CREATE TABLE IF NOT EXISTS books (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    title VARCHAR(100) UNIQUE,
                    author VARCHAR(100) NOT NULL,
                    publisher VARCHAR(100))'''
    INSERT_CMD = '''INSERT INTO books(title, author, publisher)
                    VALUES (?, ?, ?)'''
    SELECT_BY_AUTHOR_CMD = '''SELECT * FROM books WHERE author=?'''
    SELECT_BY_ID_CMD = '''SELECT * FROM books WHERE id = ?'''

    UPDATE_CMD = '''UPDATE books SET title=?, author=?, publisher=?
                    WHERE id=?'''
    DELETE_ALL_CMD = '''DELETE FROM books'''

    CURRENT_PATH = os.path.dirname(os.path.abspath(__file__))
    DATABASE_PATH = os.path.join(CURRENT_PATH, 'books.db')

    def __init__(self):
        with sqlite3.connect(self.DATABASE_PATH) as db:
            cursor = db.cursor()
            cursor.execute(self.CREATE_CMD)
            cursor.close()

    def insert(self, book):
        with sqlite3.connect(self.DATABASE_PATH) as db:
            cursor = db.cursor()
            cursor.execute(self.INSERT_CMD,
                           (book.title, book.author, book.publisher))
            cursor.close()

    def select_by_author(self, author):
        with sqlite3.connect(self.DATABASE_PATH) as db:
            cursor = db.cursor()
            cursor.execute(self.SELECT_BY_AUTHOR_CMD, (author,))
            rows = cursor.fetchall()
            cursor.close()
            return [Book(*row) for row in rows]

    def select_by_id(self, book_id):
```

```
        with sqlite3.connect(self.DATABASE_PATH) as db:
            cursor = db.cursor()
            cursor.execute(self.SELECT_BY_ID_CMD, (book_id,))
            row = cursor.fetchone()
            cursor.close()
            if row:
                return Book(*row)
            return None

    def update(self, book):
        with sqlite3.connect(self.DATABASE_PATH) as db:
            cursor = db.cursor()
            cursor.execute(self.UPDATE_CMD,
                           (book.title, book.author,
                            book.publisher, book.id))
            cursor.close()

    def delete_all(self):
        with sqlite3.connect(self.DATABASE_PATH) as db:
            cursor = db.cursor()
            cursor.execute(self.DELETE_ALL_CMD)
            cursor.close()


database_manager = DatabaseManager()

if __name__ == '__main__':
    book0 = Book(id=None,
                 title="Insula Misterioasa",
                 author="Jules Verne",
                 publisher="Adevarul")
    book1 = Book(id=None,
                 title="Steaua Sudului",
                 author="Jules Verne",
                 publisher="Adevarul")
    book2 = Book(id=None,
                 title="Pupaza din tei",
                 author="Ion Creanga",
                 publisher="Corint")
    # INSERT
    for book in [book0, book1, book2]:
        database_manager.insert(book)

    # SELECT
    print(database_manager.select_by_author("Jules Verne"))
    print(database_manager.select_by_id(2))

    # UPDATE
    book1 = Book(id=2,
                 title="STEAUA SUDULUI",
                 author="JULES VERNE",
                 publisher="Art")
```

```
    database_manager.update(book1)

    # DELETE
    # database_manager.delete_all()
```

Se observă utilizarea repetată a unui bloc **with**. Acesta asigură închiderea automată a conexiunii deschise și este recomandat să fie folosit atunci când este posibil. De asemenea, se remarcă faptul că în locul unei clase, s-a utilizat o tuplă cu nume pentru Book (deci un obiect imutabil după inițializare). Funcțiile au fost grupate într-o clasă *DatabaseManager* care conține comenzile SQL în variabile (pentru simplitate în cazul în care se dorește modificarea acestora și lizibilitate).

Exemple

Exemplul 1: Cozi de mesaje în python

Această aplicație creează două procese. Primul proces va trimite un mesaj celui de-al doilea și își va încheia execuția. Cel de-al doilea va citi mesajul din coadă și îl va afișa, după care își încheie execuția. Acest lucru este posibil folosind o coadă de mesaje din modulul *multiprocessing*. Se poate testa rezultatul dacă se înlocuiește importul Queue din multiprocessing cu: ***from queue import Queue***

Pentru mai multe detalii, vezi https://docs.python.org/3.8/library/multiprocessing.html

```
from multiprocessing import Process, Queue
# from queue import Queue
from collections import deque


def task0(message_queue):
    result = message_queue.get()
    result.append('task1: Hello World! - sent by task0')
    result.append('Process0 exists')
    message_queue.put(result)


def task1(message_queue):
    all_messages = message_queue.get()
    try:
        message = all_messages.popleft()
        if 'task1:' in message:
            print(message.split(':')[1].strip())
        else:
            print(message)
        all_messages.append('Process1 exits')
        message_queue.put(all_messages)
    except IndexError:
        print("Task1: Empty message queue")



if __name__ == '__main__':
    message_queue = Queue()
    message_queue.put(deque([]))

    process0 = Process(target=task0, args=(message_queue,))
    process1 = Process(target=task1, args=(message_queue,))

    process0.start()
    process1.start()

    process0.join()
    process1.join()

    print(message_queue.get())
```

Exemplul 2: Cozi de mesaje în python - arhitectură client-server

Arhitectura Client-Server permite crearea și înregistrarea unei cozi de mesaje pe server (metoda *register('get_queue', callable=lambda: queue)*), care va fi utilizată în continuare de clienții care se conectează la server-ul creat. Se remarcă definirea și inițializarea unui proces *Worker* care adaugă un mesaj în coadă.

Observație: Apelând metoda *server.serve_forever()* server-ul va rămâne pornit, până la închiderea acestuia prin comanda CTRL+C.

Server

```
import multiprocessing as mp
from multiprocessing.managers import BaseManager


class Worker(mp.Process):
    def __init__(self, message_queue):
        self.message_queue = message_queue
        super(Worker, self).__init__()

    def run(self):
        result = self.message_queue.get()
        result.append('Worker: Hello!')
        self.message_queue.put(result)


queue = mp.Queue()
queue.put([])
worker = Worker(queue)
worker.start()


class QueueManager(BaseManager):
    pass


if __name__ == '__main__':
    QueueManager.register('get_queue', callable=lambda: queue)
    manager = QueueManager(address=('localhost', 50000),
                           authkey=b'your_secret_key')
    server = manager.get_server()
    server.serve_forever()
```

Client 0

Clientul 0 se va conecta la server-ul definit anterior și va adăuga un mesaj în coadă.

```
from multiprocessing.managers import BaseManager


class QueueManager(BaseManager):
    pass
```

```
if __name__ == '__main__':
    QueueManager.register('get_queue')
    manager = QueueManager(address=('localhost', 50000),
                           authkey=b'your_secret_key')
    manager.connect()
    queue = manager.get_queue()
    result = queue.get()
    result.append('Client 0: Hello!')
    queue.put(result)
```

Client 1

Clientul 1 se va conecta la server-ul definit anterior și va afișa întreaga coadă de mesaje.

```
from multiprocessing.managers import BaseManager


class QueueManager(BaseManager):
    pass


if __name__ == '__main__':
    QueueManager.register('get_queue')
    manager = QueueManager(address=('localhost', 50000),
                           authkey=b'your_secret_key')
    manager.connect()
    queue = m.get_queue()
    print("Client 1:", queue.get())
```

Exemplul 3: Intercomunicare între procese C și procese Python prin cozi de mesaje System V

Executabilul generat în urma compilării fișierului *sender.c* va crea coada de mesaje și va scrie în ea un mesaj citit de la tastatură. Este important ca sender-ul să fie executat primul. În caz contrar, receiver-ul din C va aștepta la infinit, iar scriptul în python va afișa un mesaj cu coada de mesaje neinițializată.

Pentru compilare și execuție, se vor executa într-un terminal deschis în folder-ul curent, următoarele comenzi:

```
gcc sender.c -o sender
./sender
# respectiv
gcc receiver.c -o receiver
./receiver
# scriptul in python poate fi executat cu comanda:
python3 sender_receiver.py
```

Pentru mai multe detalii cu privire la System V IPC (Inter-Process Communication) din python, se poate consulta documentația oficială, disponibilă la adresa: http://semanchuk.com/philip/sysv_ipc/

- **sender.c**

```c
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    // gcc sender.c -o sender
    // ./sender
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("message_queue_name", 'B');
    printf("key is %d \n" , key);

    // msgget creates a message queue and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data : ");
    fgets(message.mesg_text, 101, stdin);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);

    // display the message
    printf("Data send is : %s \n", message.mesg_text);

    return 0;
}
```

- **receiver.c**

Se observă faptul că diferența majoră față de programul anterior e înlocuirea apelului funcției msgsnd (message send) cu msgrcv (message receive).

```c
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;
```

```c
int main()
{
    // gcc receiver.c -o receiver
    // ./receiver
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("message_queue_name", 'B');

    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n", message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

- **sender_receiver.py**

Pentru a instala dependențele, se execută într-un terminal deschis în folder-ul curent comenzile:

```
python3 -m venv env
venvact  # alias for: source env/bin/activate
pip3 install sysv-ipc==1.0.1
```

Terminalul va rămâne deschis cu mediul de lucru virtual pornit pentru a putea executa scriptul cu comanda:

```
python3 sender_receiver.py
```

Codul sursă:

```python
import sysv_ipc


def send_message(message_queue, message):
    message_queue.send(message)


def receive_messages(message_queue):
    for item in message_queue.receive():
        if type(item) == bytes:
            print("received: {}".format(item.decode()))
```

```
if __name__ == '__main__':
    try:
        # put the key (integer) as parameter (in this case: -1)
        message_queue = sysv_ipc.MessageQueue(-1)
        send_message(message_queue, "Python says: Hello!")
        receive_messages(message_queue)
    except sysv_ipc.ExistentialError:
        print("Message queue not initialized. Please run the C program
first")
```

Se remarcă valoarea -1 de la inițializarea cozii de mesaje. Aceasta a fost preluată din output-ul executabilului *sender*.

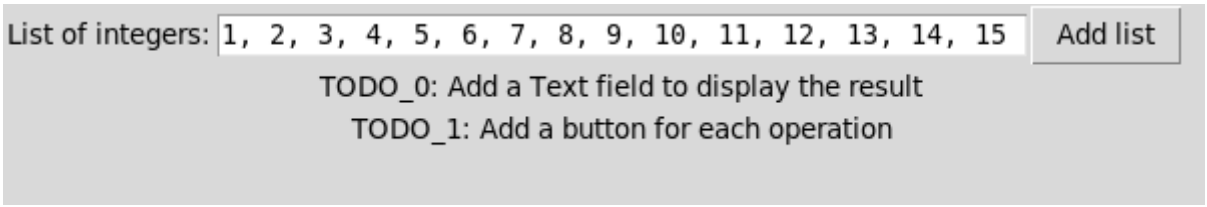
Aplicații și teme

Aplicații de laborator:

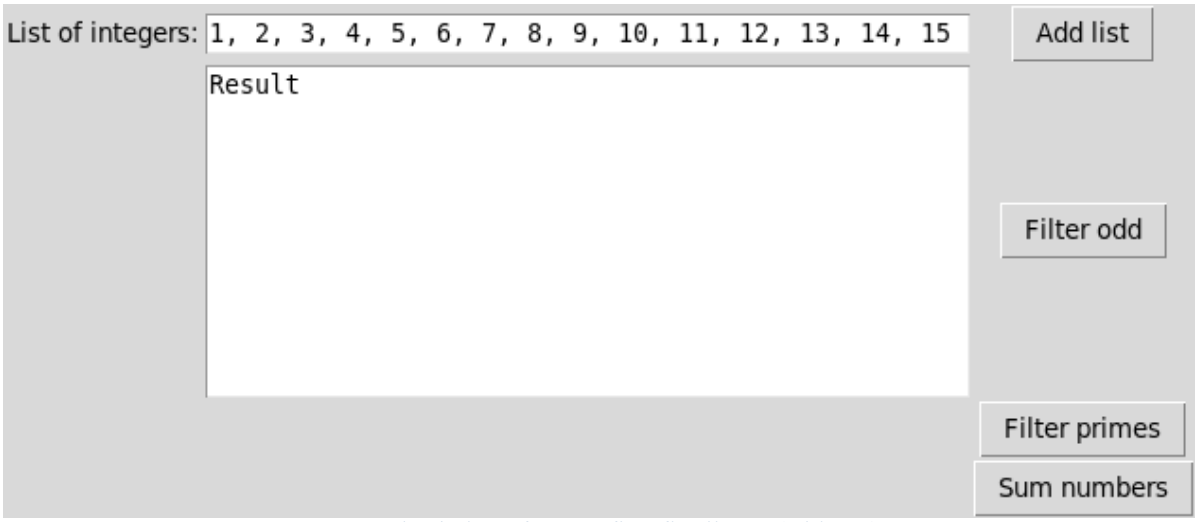
1. Să se proiecteze și să se implementeze o interfață grafică cu Tkinter care să preia o listă de numere întregi și să aplice următoarele prelucrări:
    - filtrarea numerelor impare din coada de mesaje
    - filtrarea numerelor prime din coadă
    - sumarea elementelor din coadă

    Fiecare prelucrare constituie un task realizat de un nou proces. Toate procesele utilizează aceeași coadă de mesaje. La finalul fiecărei prelucrări, rezultatul va fi afișat pe interfață. Pentru fiecare funcționalitate nou adăugată, se execută *git add* și *git commit*.

    Se dă scheletul pentru interfața grafică, precum și funcționalitatea de preluare a numerelor, rezultatul fiind o listă de numere întregi.



Interfață grafică realizată cu Tkinter



Exemplu de interfață grafică finalizată (Tkinter)

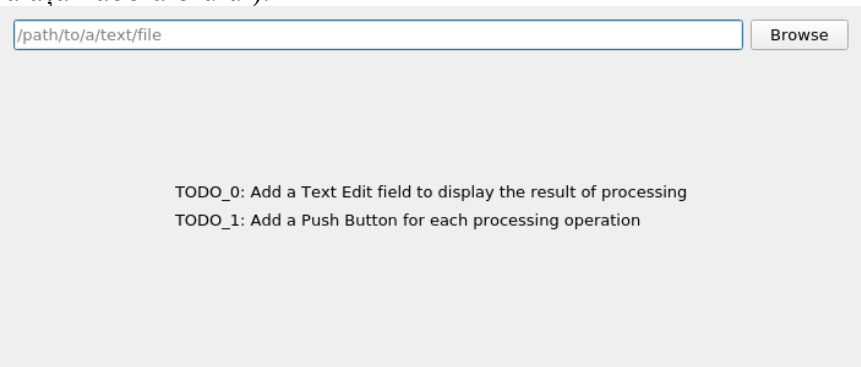
2. Să se proiecteze și să se implementeze o aplicație jurnal, folosind PyQt5, care să:
    - preia dintr-un fișier text câte un citat la întâmplare la fiecare deschidere a aplicației și să îl afișeze pe pagina principală;
    - aibă o zonă de text editabilă în care se pot adăuga intrări în jurnal, sau se pot consulta intrările vechi (deci trebuie adăugate două butoane: Load și Save); intrările în jurnal pot fi salvate individual pe disk ca fișiere text cu un timestamp ca denumire;
    - **[OPȚIONAL]:** să se implementeze funcționalitatea de adăugare a citatelor noi în fișierul text din interfața grafică.

2. Modificați aplicația 2 astfel încât să folosească în locul fișierelor text, două baze de date SQLite, una pentru citate și cealaltă pentru intrările în jurnal. Se cere de asemenea și diagrama E-R.

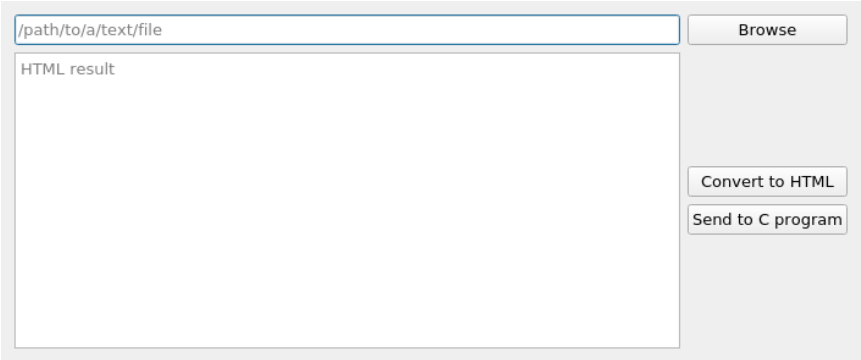
Teme pe acasă:

1. Să se proiecteze și să se implementeze o interfață grafică cu PyQt5 cu un câmp text în care să poate fi introdusă calea către un fișier text și un buton de upload. Aplicația python va citi conținutul fișierului și îl va converti într-un fișier html (cu alte cuvinte, trebuie să fie

delimitat titlul și paragrafele). Rezultatul prelucrării va fi trimis printr-o coadă de mesaje către aplicația din limbajul C, care va valida conținutul cu un regex și îl va scrie într-un fișier de ieșire. Pentru fiecare funcționalitate nou adăugată, se execută *git add* și *git commit*.
Se dă scheletul pentru interfața grafică, precum și funcționalitatea de a căuta un fișier pe disk (vezi codul atașat laboratorului).



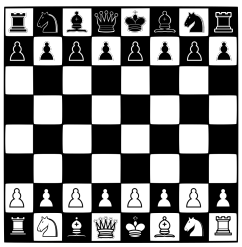
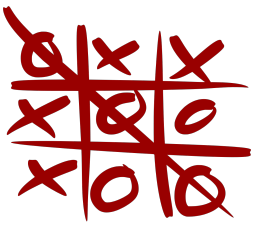
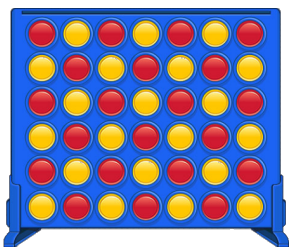
Interfață grafică realizată cu Qt Designer și PyQt5



Exemplu de interfață grafică finalizată (PyQt5)

2. Să se proiecteze și să se implementeze un joc de tip P2P (Peer-to-Peer) pentru doi jucători utilizând Tkinter / PyQt5 / PyGame (vezi documentația: https://www.pygame.org/docs/), care să primească ca input la deschidere numele jucătorului și să aștepte până se conectează alt jucător (o altă instanță a jocului). Comunicarea acțiunilor celor doi jucători (două instanțe ale jocului pornite) se va realiza printr-o coadă de mesaje (System-V). La finalul fiecărei runde se va salva numele jucătorilor și scorul într-o bază de date SQLite. Dacă aceiași doi jucători reiau într-o altă sesiune jocul, se va afișa în partea de sus a ferestrei scorul din baza de date. Se poate implementa orice joc unu la unu, cu condiția ca acțiunile fiecărui jucător să fie comunicate celuilalt (și afișate pe interfață) prin intermediul unei cozi de mesaje, iar scorul să fie memorat într-o bază de date și preluat la inițierea unei sesiuni de joc (dacă există).

   Se poate opta pentru unul dintre jocurile de mai jos, sau se poate propune altul.


   Se cer de asemenea diagrama UML de clase și diagrama E-R.

[BONUS]: Să se proiecteze și să se implementeze un chat multiproces utilizând cozi de mesaje și o bază de date SQLite. În baza de date vor fi stocate următoarele informații:

- date despre utilizatori (nume de utilizator, nume, adresa de e-mail, parola);
- mesajele trimise de utilizatori (id-ul mesajului, id-ul destinatarului, id-ul expeditorului, mesajul propriu-zis)
- **Opțional:** se poate aplica criptare/decriptare pe parolă, sau o funcție de hashing

Se cere:

- diagrama entitate-relație (E-R) pentru baza de date
- diagrama UML de clase
- diagrama de use-case
- interfață grafică (ori cu Tkinter, ori cu PyQt5) care să conțină:
  - un formular de înregistrare a unui utilizator nou
  - un formular de login
  - o fereastră de căutare după numele unui utilizator
  - fereastra de chat: la deschidere, se încarcă (din baza de date) istoricul conversației cu utilizatorul respectiv într-o zonă text editabilă.
- utilizarea cozilor de mesaje pentru conversațiile dintre doi utilizatori și salvarea mesajelor în baza de date (salvarea istoricului). Există două posibilități:
  - pentru fiecare conversație va fi creată o nouă coadă de mesaje prin care doi utilizatori vor comunica
  - o arhitectură Client-Server (deci o singură coadă de mesaje), mesajul conținând trei elemente: *expeditor, destinatar, mesajul propriu-zis*
- se va inițializa un repository git (git init) și se va adăuga fiecare funcționalitate nouă în repository-ul de pe bitbucket (git add <file>, git commit -m "message", git push)