Database management system (DMS)

# Cuprins

Introducere

Directoare

Cerințe de sistem, Instalare, Upgradare

Dezvoltarea aplicației

Core

Parsarea comenzilor

Interfață grafică

Unit testing

Core

Parsarea comenzilor

Interfață grafică

Assert-uri

Core

Parsarea comenzilor

Interfață grafică

# Introducere

Această documentație conține toate informațiile despre modul în care funcționează și a fost conceput DMS.

Printre acestea se numară

* Concepte importante
* Procesul de instalare și rulare
* Informații de bază despre toate operațiile ce pot fi făcute

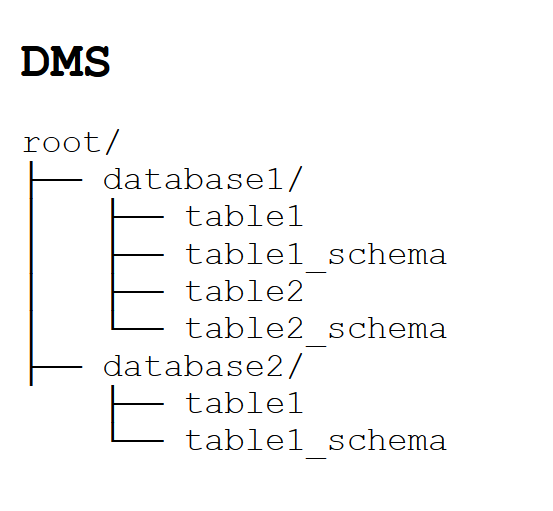
Database Management System

DMS este un program ce permite executarea de instrucțiuni SQL simple la linia de comandă cât și prin intermediul unei interfețe grafice. Momentan sunt implementate următoarele funcționalități SQL:

* Select
* Insert
* Delete
* Create table
* Create database
* Update
* Import/Export as plain text, csv and xml

### Directoare

Baza de date este salvată pe disk într-un director specific. Pentru fiecare bază de date se creează un director, iar pentru fiecare tabelă se creează un fițier schema lângă fișierul ce reține toate informațiile din tabelă.



# Cerințe de sistem, Instalare, Upgradare

Cerințe de sistem:

* Sistem de operare: Windows, Linux
* RAM minim necesar: 1MB
* Spațiu HDD: depinde de cât de mare devine baza de date
* Altele: python > 2.7

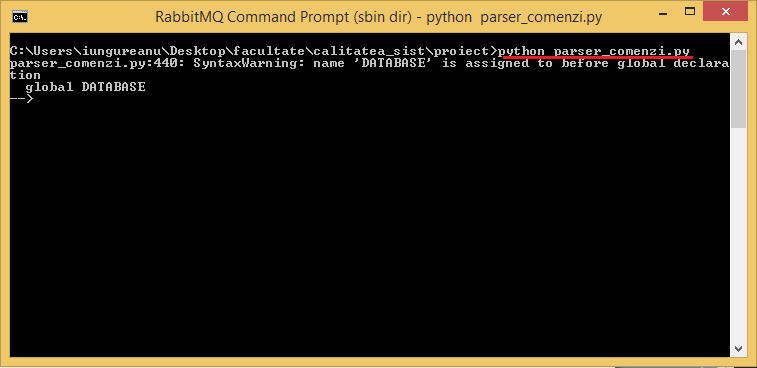
Instalare

Descarcati sursele de pe <http://mega> repo FTW.biz

Upgradare

Din păcate nu oferim suport pentru așa ceva

Pentru a rula aplicația la linia de comandă se rulează scriptul python parser\_comenzi.py.



Pentru a rula interfața grafică, se rulează scriptul *app.py*, sau versiunea cu aserțiuni, *app\_asserts.py*.

# Dezvoltarea aplicației

Aplicația a fost impărțită în 3 componente principale.

1. Core – care se ocupă cu funcționalitatea propriu zisă și expune o serie de funcții care vor fi apelate de celelalte module
2. Parsarea comenzilor de la linia de comandă care va apela funcțiile din core
3. Interfața grafică care va apela funcțiile din core

## 3.1 Core

Acest modul se ocupa cu operatiile I/O si expune functiile necesare celorlalte module pentru feature-urile de DELETE, SELECT, CREATE, etc.

**O scurta descriere a fiecarei functii:**

set\_root\_location(workspace\_path)

* Seteaza calea radacina a spatiului de lucru
* workspace\_path : este calea completa catre spatiul de lucru

create\_database(db\_name)

* Creaza un folder cu numele db\_name in spatiul de lucru presetat

delete\_database(db\_name)

* Sterge folder-ul cu numele db\_name din spatiul de lucru presetat

create\_table(db\_name, table\_name, args\_list)

* Creaza un fisier cu numele table\_name in folder-ul db\_name
* Creaza un fisier cu numele table\_name\_”schema” ce contine campurile din args\_list impreuna cu tipul acestora. Args\_list este de forma [(“camp”, “tip”)]

delete\_table(db\_name, table\_name)

* Sterge fisierul cu numele table\_name din folderul db\_name
* Sterge fisierul "table\_name"\_schema din folderul db\_name

check\_type(upl1, upl2)

* Este o functie ce verifica ca doua uple sa aibe acelasi tip

insert\_in\_table(db\_name, table\_name, args\_list)

* Adauga o linie noua in fisierul table\_name continand valorile sub forma de uple din lista args\_list
* Exemplu : args\_list = [(a,b)] , unde "a" este numele campului si "b" valoarea acestuia

change\_database(old\_db\_name, new\_db\_name)

* Modifica numele folderului old\_db\_name cu new\_db\_name

change\_table\_name(db\_name, old\_table\_name, new\_table\_name)

* Modifica numele fisierului old\_table\_name cu new\_table\_name

change\_fileds\_in\_table(db\_name, table\_name, args\_list)

* Modifica denumirea campurilor din fisierul table\_name\_schema cu cele din args\_list
* Args\_list este o lista de tuple : [(a,b)] unde "a" = numele campului de schimbat, "b" noul nume pentru acel camp

add\_columns\_in\_table(db\_name, table\_name, columns)

* Va adauga toate campurile din lista columns in fisierul table\_name\_schema
* columns este o lista de format [(field1\_name, field1\_type),(field2\_name, field2\_type)]

drop\_column\_in\_table(db\_name, table\_name, column\_name)

* Va sterge campul din fisierul table\_name\_schema

where(value1, op, value2)

* Aceasta functie verifica daca value1 este in relatia “op” cu value2
* Exemplu: where(1, "<", 2) => True

delete\_in\_table(db\_name, table\_name, where\_list)

* Aceasta functie sterge toate liniile din fisierul table\_schema ce se supun conditiilor din lista where\_list
* where\_list este o lista de conditii de forma: [(field\_name, operator, value2) ,(..)]

select\_in\_table(db\_name, table\_name, select\_list, where\_list=[])

* Aceasta functie intoarce o lista de linii ce contin doar campurile din lista select\_list
* Optional se selecteaza doar liniile ce se supun conditiilor din where\_list

get\_schema(db\_name, table\_name)

* Aceasta functie returneaza schema pentru fisierul table\_name

update\_in\_table(db\_name,table\_name, update\_list, where\_list=[])

* Aceasta functie actualizeaza campurile din lista update\_list cu valorile din aceeasi lista
* update\_list este o lista de tuple de forma : [(field\_name, field\_value)]
* Optional actualizeaza doar liniile ce se supun conditiilor din lista where\_list

## 3.2 Parsarea comenzilor

Când se rulează scriptul, se va apela funcția \_\_main\_\_ care preia datele de la linia de comandă și va apela funcția parse\_comanda.

Funcția parse\_comanda va tokeniza datele și în funcție de ce valoare are primul token se va apela o funcție specifică (primul token reprezenta ce comandă SQL trebuie executată). Fiecare funcție are un nume sugestiv pentru ce trebuie să facă (parse\_select, parse\_insert, etc.).

Există 2 funcții care sunt folosite în procesul de parsare: pentru a extrage clauzele where și tokenii aflați între paranteze rotunde. Pentru funcția care extrage tokenii aflați între paranteze, se pargurg tokenii, se elimină caracterul “,” și se salveză rezultatul într-o listă care va fi returnată. Pentru funcția care extrage clauzele where, se elimină caracterul “,” și se salvează clauzele într-o listă de tuple de forma: [(câmp1, operator, câmp2)]

Ex: select nume from table where id == 5 and varsta > 22

Clauze where: [(id, ==, 5), (varsta, >, 22)]

Pentru funcția select se apelează parse\_select, unde se extrag coloanele care vor fi selectate și în caz că avem o clauză where, se va extrage și condițiile din clauză. Momentan se poate folosi doar AND între clauze.

Funcția parse\_create tratează și creearea de baze de date cât și cea de tabele.

Funcția parse\_insert va folosi funcția de extragere a tokenilor aflați între paranteze și va creea o listă de tuple de forma [(coloană\_de\_inserat, valoare\_de\_inserat)]

Ex: insert into tabel (nume, varsta) values (ionut, 22)

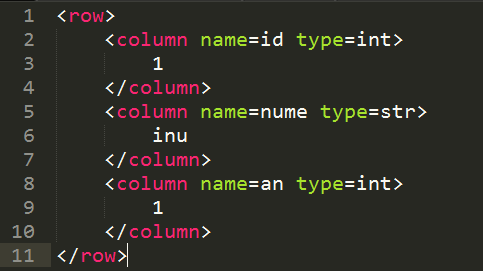
Lista: [(nume, ionut), (varsta, 22)]

Funcția parse\_update va folosi funcția de extragere tokeni din clauze where atât pentru clauzele where cât și pentru ce coloane vor trebui modificate (deoarece cele 2 se aseamană ca sintaxă, diferențele fiind foarte mici)

Funcțiile parse\_drop, parse\_alter, parse\_delete, parse\_use, parse\_change\_database, parse\_export, parse\_import pur și simplu extrag tokenii (nume de tabel, nume de coloane, etc.) și la fel ca la celelalte funcții se apeleaza metodele expuse de core.

Există posibilitatea de a importa și exporta tabele în format plain text, csv și xml. Pentru a face acest lucru se folosesc funcțiile export/import\_[plain,csv,xml]. Pentru export\_plain se scrie un fișier care conține schema tabelului la început, urmat de un separator “--END\_OF\_SCHEMA--”, urmat de datele din tabel, fiecare coloană fiind separată prin caracterul tab. Export\_csv este la fel ca export\_plain, diferența fiind că acum separatorul este caracterul “,”. Export\_xml va creea un fisier xml în care fiecare rând din tabela va fi într-un tag row care va conține taguri pentru fiecare coloană. De asemenea, tagurile pentru coloană vor conține atribute pentru nume și tipul datelor.

Ex:

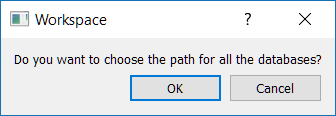


Pentru import se citesc fișierele data ca argumen și în cazul plain și csv se citește schema tabelului și se validează cu datele din fișier, iar pentru xml se folosesc atributele din taguri pentru a recreea schema.

Pentru parsarea comenzilor și import/export nu a fost folosită nici o librarie din python.

## Interfața grafica

Modulul *app.py* se ocupă de interfața grafică a aplicației. Componentele grafice ale aplicației au fost scrise în framework-ul Qt (varianta PyQt – un wrapper pentru python peste codebase-ul Qt-ului). Codul a fost scris modularizat, astfel încât pentru fiecare componentă logică a aplicației (e.g. insert in table, delete from table, select from table, modify table, create table, delete table, create database, delete database, etc.), a fost scrisă o nouă clasă. Aplicația extrage numele directorului unde se află toate bazele de date din fișierul *settings.ini*, câmpul *databasesDirectory*. Astfel, dacă acest fișier nu există, sau acest câmp nu există, sau valoarea câmpului este una invalidă (i.e. directorul nu există sau nu este valid), la executarea aplicației, utilizatorul va putea alege prin intermediul unui *dialog*, directorul ce se va ocupa de bazele de date. (Fig. 1)

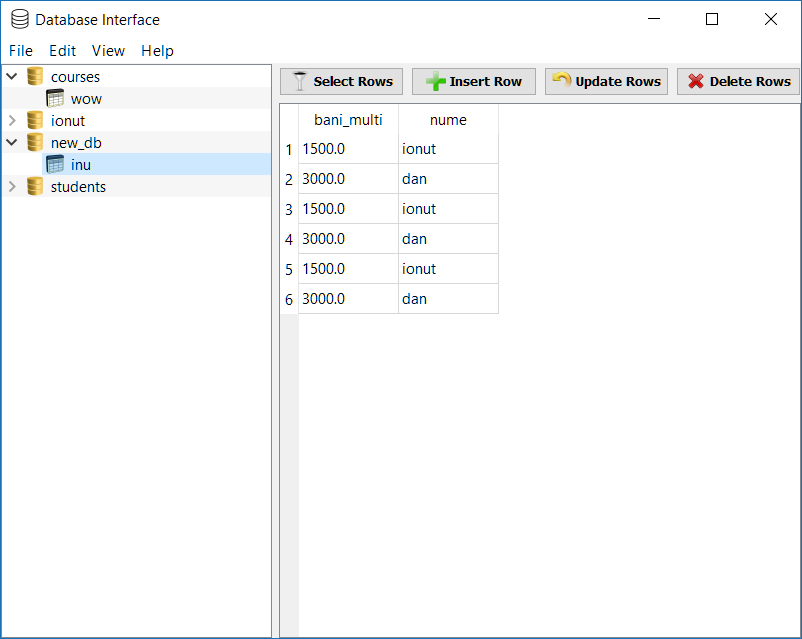


*Fig. 1*

Dialog-ul principal al aplicației conține un meniu cu ajutorul căruia poate efectua diferite operații:

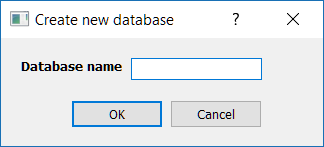
* File
  + New Database
  + Delete Database
  + Workspace
  + Exit
* Edit
  + Create Table
  + Modify Table
  + Delete Table
  + Export Table
  + Import Table
* Help
  + About

Totodată, dialog-ul mai conține și o componentă de tipul TreeView și una de tipul TableView. Cu ajutorul arborelui, utilizatorul va putea selecta bazele de date, iar din interiorul acestora, va putea selecta tabele. O dată ce un tabel a fost selectat, TableView-ul va afișa conținutul acelui tabel, iar butoanele de deasupra TableView-ului vor putea fi accesate pentru a efectua diferite operatii(e.g. Select Rows, Insert Row, Update Rows, Delete Rows). (Fig. 2)



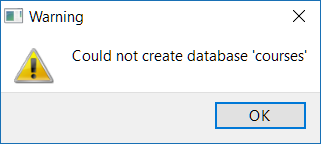
*Fig. 2*

Crearea unei baze de date se poate face accesând meniul *File -> Create Database*, permițându-i utilizatorului sa introducă numele bazei de date. (Fig. 3)



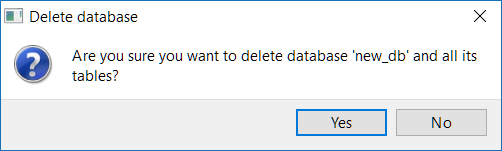
*Fig. 3*

Dacă numele bazei de date există deja, acesta va primi un warning. (Fig. 4)



*Fig. 4*

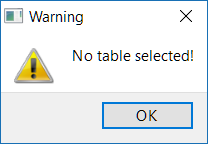
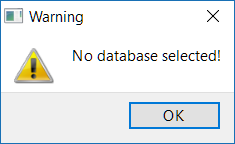
Pentru ștergerea unei baze de date, utilizatorul trebuie să acceseze meniul *File -> Delete Database*, fiind nevoit să selecteze baza de date propriu-zisă, sau, un tabel care aparține bazei de date. (Fig. 5)



*Fig. 5*

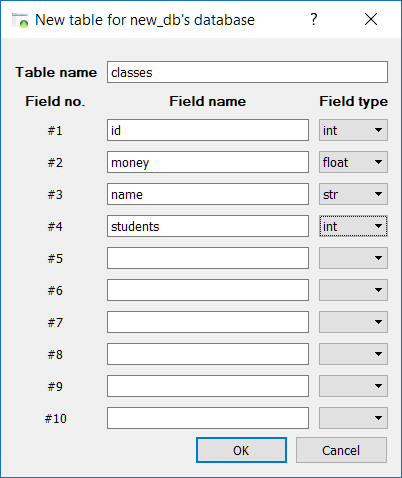
Modificarea directorului ce conține bazele de date se poate face accesând meniul *File -> Workspace*, după care utilizatorul trebuie să selecteze un director cu ajutorul unui *FileDialog*.

Operațiile din meniul *Edit* se pot face doar asupra unui tabel selectat, excepție făcând doar comanda *Create Table*, care poate avea o bază de date selectată. Astfel, dacă nu este indeplinită cerința, se vor afișa unul din două warning-uri. (*Fig 6. a, Fig 6. b).*



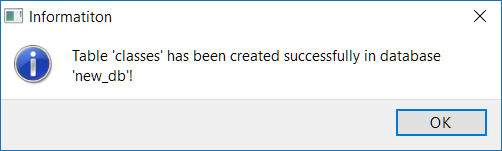
*Fig 6. a Fig 6. b*

Pentru crearea unui tabel, se selectează meniul *Edit -> Create Table*. (Fig. 7)



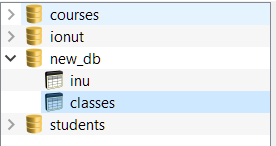
*Fig. 7*

Dacă toate cele 10 câmpuri au fost completate (atât numele cât și tipul), atunci se vor mai adăuga pe rând câte un nou câmp până când ultimul nu se va mai completa. Astfel, la apăsarea butonului *Ok*, dacă nu există probleme (e.g. dacă o tabelă cu același nume există deja), o nouă tabelă cu numele și câmpurile trecute/selectate în TextBox-uri/ComboBox-uri va fi creată. (Fig. 8)



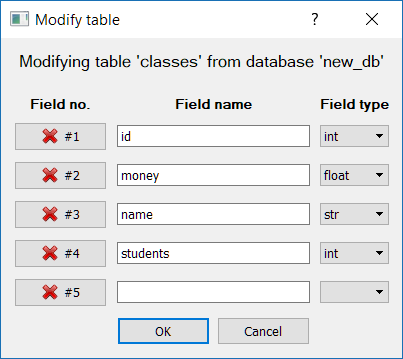
*Fig. 8*

În urma creării tabelei, TreeView-ul a fost actualizat. (Fig. 9)



*Fig. 9*

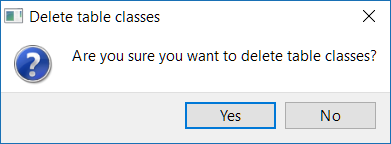
Pentru modificarea unei tabele, se selectează meniul *Edit -> Modify Table*. (Fig. 10)



*Fig. 10*

Cu ajutorul acestui dialog, utilizatorul poate schimba numele câmpurilor, tipul câmpurilor, poate să adauge noi câmpuri (într-o manieră asemănătoare creării unui tabel), sau să elimine anumite câmpuri, apăsând pe butonul X corespunzător câmpului ce trebuie eliminat.

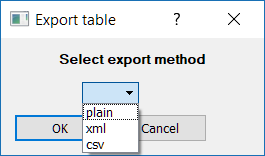
Ștergerea unei tabele se face relativ ușor, selectând meniul *Edit -> Delete Table*. (Fig. 11)



*Fig. 11*

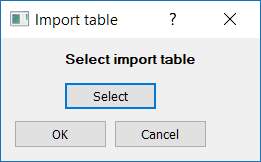
Asupra unui tabel se mai pot face și următoarele două operații, folosind meniul *Edit*:

* *Export Table* (Fig. 12)



*Fig. 12*

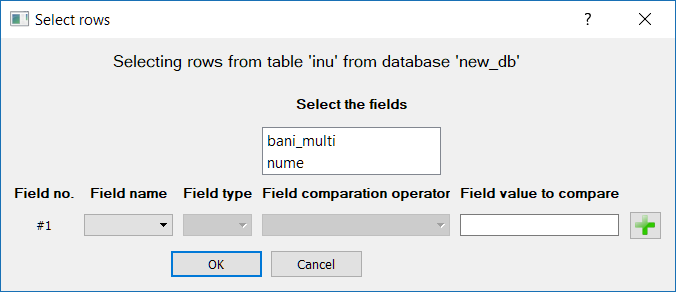
* *Import Table* – la apăsarea butonului *Select* se deschide un FileDialog (Fig. 13)



*Fig. 13*

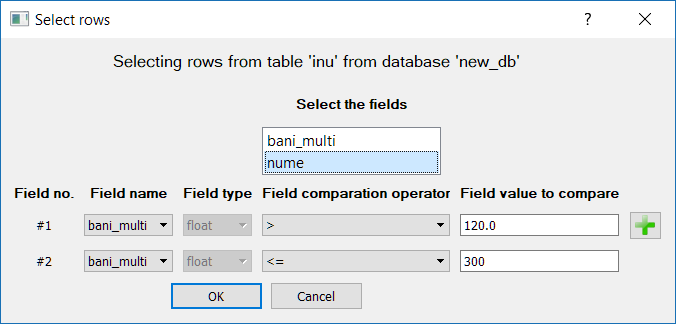
Pentru a putea selecta, insera, actualiza sau șterge date, trebuie să fie selectat un tabel din TreeView.

Pentru selectarea datelor dintr-un tabel, trebuie apăsat butonul *Select Rows* de deasupra TableView-ului. (Fig. 14)



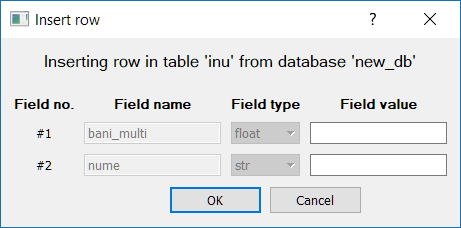
*Fig. 14*

Acest dialog permite selectarea câmpurilor care vor fi afișate (dacă nu se selectează niciunul atunci vor fi selectate toate), iar dacă este nevoie de o clauză *WHERE*, atunci se poate selecta numele unui câmp, operatorul care să fie folosit la comparație și valoarea cu care să se facă comparația. Astfel, după ce s-a completat un astfel de câmp, la apăsarea butonului ***PLUS*** se va insera un nou câmp care va fi folosit în conjuncție cu toate celelalte. (Fig. 15)



*Fig. 15*

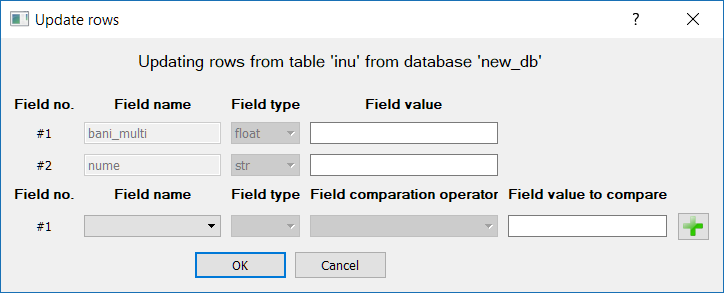
Pentru inserarea datelor într-un tabel, trebuie apăsat butonul *Insert Row*. (Fig. 16)



*Fig. 16*

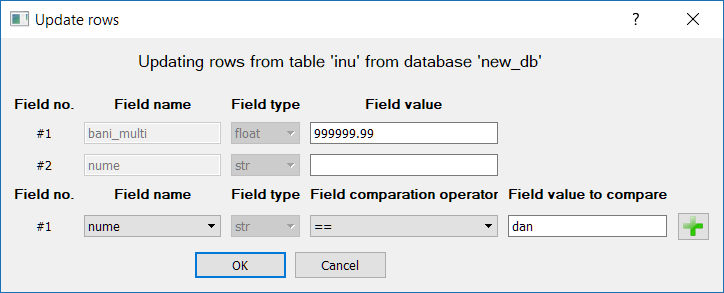
Acest dialog va afișa o lista cu toate câmpurile tabelei în care trebuie inserată noua linie, permițându-I utilizatorului sa introducă noile valori pentru câmpurile tabelei.

Pentru actualizarea datelor dintr-un tabel, se poate folosi butonul *Update Rows*. (Fig. 17)



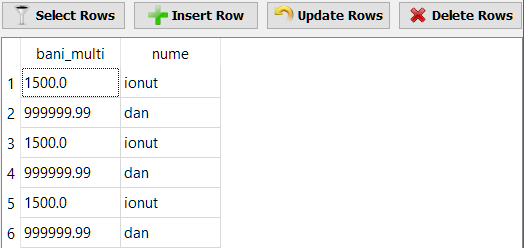
*Fig. 17*

Datele care trebuie actualizate vor fi date în câmpurile tabelei (într-o manieră asemănătoare inserării datelor), iar dacă este nevoie de o clauză *WHERE*, atunci se pot selecta criteriile cu ajutorul câmpurilor, a operatorilor și a valorilor ce trebuie comparate (într-o manieră asemănătoare selectării datelor). (Fig. 18)



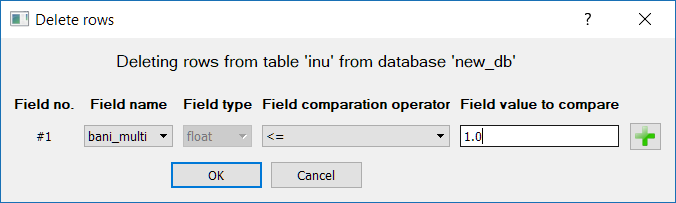
*Fig. 18*

Așadar, după apăsarea butonului *Ok*, se poate observa cu TableView-ul s-a actualizat. (Fig. 19)



*Fig. 19*

Pentru ștergerea datelor dintr-o tabelă, se poate apăsa butonul *Delete Rows*. (Fig. 20)



*Fig. 20*

Cu ajutorul acestui dialog, se pot selecta multiple câmpuri pentru clauza *WHERE*, într-o manieră asemănătoare selectării datelor.

# Unit testing

Pentru partea de unit testing s-a folosit framework-ul unittest. Trebuie creeată o clasă care să moștenească unittest.TestCase și apoi să adaugăm teste care să acopere o gamă cât mai largă de erori. În caz că este nevoie să setăm date înainte de teste se poate folosi metoda setUp(self) în care inițializăm datele de care avem nevoie.

Fiecare modul a fost testat independent de celălalt

## 4.1 Core

Acest modul are doua Test Case-uri:

* Pentru Database
* Pentru Table

Majoritatea testelor folosesc “assertRaises” si doua tipuri de exceptii "FileExistsError", “FileNotFoundError". Aproape toate functiile din modulul database\_utils fac operatii I/O pe disc astfel ca este critic pentru ele sa se verifice existenta bazei de date ca folder dar si a tabelei ca fisier.

**Exemplu:**

def test\_create\_table2(self):

       self.assertRaises(FileNotFoundError, create\_table, "baza", "tabela", [])

O alta mare categorie de teste se refera la tipul de exceptie “IndexError” deoarece multe functii primesc ca parametrii liste. Pentru a testa aruncarea acestei exceptii am dat ca si parametri functiilor de test liste goale dar si liste cu tuple goale. Spre exemplu:

def test\_insert\_in\_table3(self):

       self.assertRaises(IndexError, insert\_in\_table,  self.db, "tabela", [(), (), (), ()])

Putem crea o alta categorie de teste in jurul exceptiei “ValueError”. Un exemplu ar putea fi incercarea de a schimba numele unui camp cu unul ce deja exista in tabela.

def test\_change\_fields\_in\_table4(self):

       self.assertRaises(ValueError, change\_fields\_in\_table, self.db, "tabela", [("field6", "field7")])

Un alt exemplu ar fi adaugarea unei coloane ce deja exista:

def test\_add\_column\_in\_table4(self):

        self.assertRaises(ValueError, add\_column\_to\_table, self.db, "tabela", [("field1", "str")])

Un ultim exemplu ar fi folosirea unui parametru gol:

def test\_drop\_column\_in\_table3(self):

        self.assertRaises(ValueError, drop\_column\_in\_table, self.db, "tabela", "")

A patra categorie de teste este despre testarea functiilor de conversie dar si a functiilor ce sunt folosite pentru a evalua conditiile “where”.

EX 1:

def test\_where1(self):

       self.assertEqual(where(1, "<", 2), True)

EX 2:

def test\_convert1(self):

       types = ['int', 'str', 'float']

       values = ['5', 'ceva', 'str']

       self.assertRaises(TypeError, convert, types, values)

EX 3:

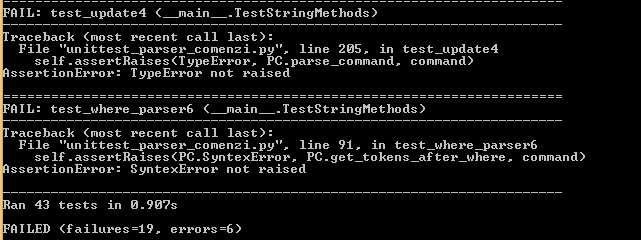
def test\_convert\_string1(self):

       temp\_lista = [5,6,7]

       self.assertEqual(convert\_to\_string(temp\_lista), ['5','6','7'])

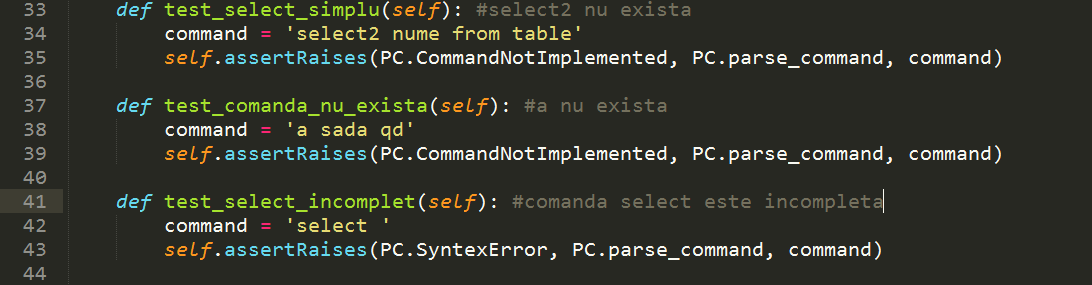
## 2 Parser comenzi

Pentru a rula testele se execută scriptul python unittest\_parser\_comenzi.py.

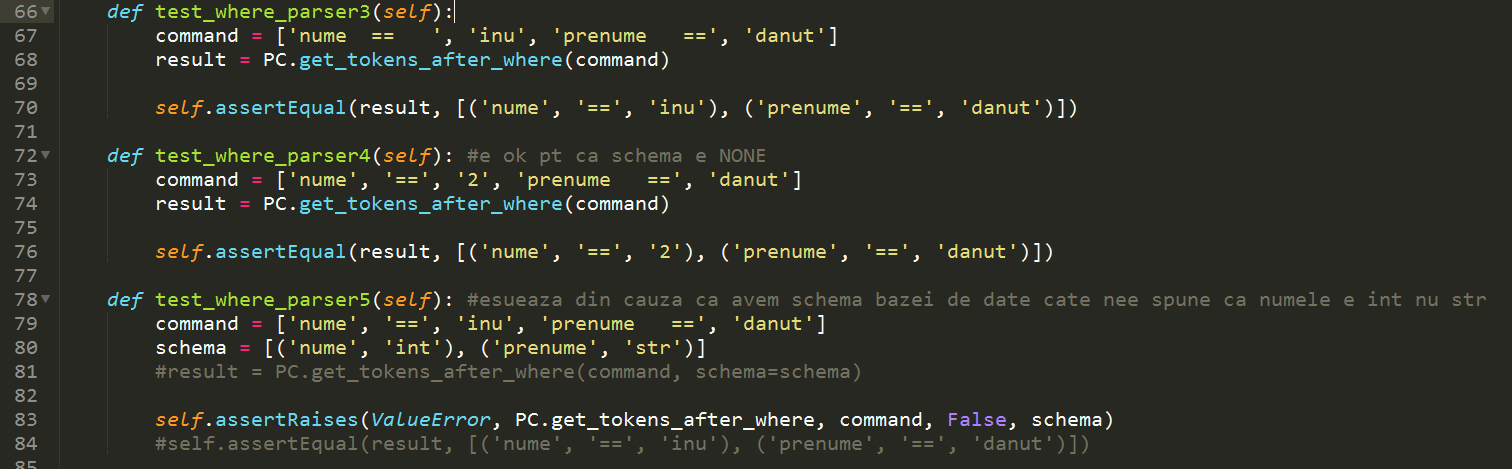


Testele au acoperit o gamă largă de erori ce pot apărea.

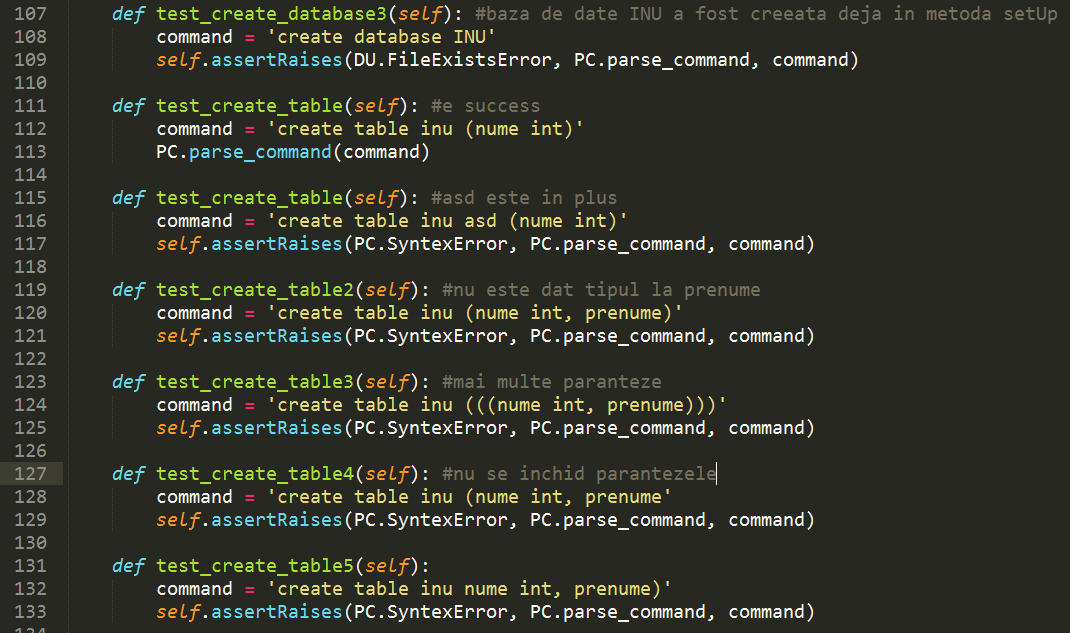
Comenzi invalide, incomplete sau inexistente



Comenzi care să testeze cum se descurcă parserul în caz că adăugăm mai multe spații, dacă avem schema tabelului sau nu



Teste în care o bază de date cu acelați nume există deja, cuvinte în plus în interiorul comenzii, nu este dat tipul coloanei, se adaugă mai multe paranteze sau nu se deschid/inchid paranteze sau lipsesc cu totul.



Alte tipuri de teste: cuvinte cheie greșite, lipsă anumite cuvinte cheie, insert/update date în tabele care nu corespund cu schema tabelului, adăugarea de spații în plus, dacă la comenzile de tip import schemele se potrivesc (cea din fisier cu cea din tabel).

## Interfața grafică

Modulul *app\_unittest.py* se ocupă de testarea aplicației, folosind modulul de unittesting

*PyQt4.QtTest*. La inițializarea obiectului *unittest.TestCase*, se creează o nouă bază de date și un tabel. Au fost scrise următoarele funcții de testare:

* test\_grid\_layout – această funcție testează dacă numărul de elemente a unui GridLayout ce aparține unei tabel, a fost modificat după ce s-a șters un element;
* test\_create\_database – această funcție creează o nouă bază de date, simulând introducerea textului “NewDatabase” în câmpul numelui bazei de date, după care simulează apăsarea butonului *Ok*, testând dacă într-adevăr numele TextBox-ului s-a modificat;
* test\_cancel\_create\_database – această funcție testează dacă la apăsarea butonului *Cancel* atunci când se creează o nouă bază de date, TreeView-ul aplicației nu se modifică;
* test\_modify\_tree\_model – această funcție testează dacă numărul total de componente rămâne nenegativ atunci când se șterg mai multe elemente decât numărul total al acestora;
* test\_modify\_table\_widget – această funcție creează un tabel de 2x2, inserând valori de la 0 la 3, după care “curăță” tabelul, verificând dacă dimensiunea acestuia a devenit 0x0;
* test\_disabled\_items – această funcție dezactiveaza o componentă TextField, după care simulează introducerea unui cuvânt de la tastatură, urmând să testeze egalitatea TextField-ului;
* test\_new\_table – această funcție face o testare exaustivă la crearea unei noi tabele, încercând să combine toate tipurile câmpurilor cu toate valorile primitivilor (3x3x10 combinații).



# Assert-uri

Au fost adăugate assert-uri în cod pentru a verifica preconditii, postconditii și invarianți. În python se face acest lucru prin comanda assert {conditie}.

## 5.1 Core

Modulul database\_utils are multe operatii I/O pe disk astfel ca mare parte din apelurile “assert” se refera la existenta fisierelor.

EX 1:

if os.path.exists(db\_location):

   assert os.path.exists(db\_location)

   shutil.rmtree(db\_location)

   assert not os.path.exists(db\_location)

Un alt exemplu de utilizare “assert” in codul nostru este atunci cand vrem sa inseram in tabela o noua intrare si verificam lungimea obiectului nostru temporar cu lungimea continutului fisierului dupa inserare.

EX 2:

assert len(linii) < len(select\_in\_table(db\_name,table\_name,[]))

Am folosit “assert” pentru a verifica daca schmba este schimbata dupa o operatie de adaugarea a unui camp nou.

EX 3:

len\_oschema = len(get\_schema(db\_name, table\_name))

/\* add field operations \*/

alta = len(get\_schema(db\_name, table\_name))

assert len\_oschema < alta

La fel si pentru lunghimea “schemei” dupa o redenumirea unui camp.

EX 4:

oschema = get\_schema(db\_name,table\_name)

/\* rename field operations\*/

oschema2 = get\_schema(db\_name, table\_name)

assert oschema != oschema2

Un ultim exemplu de utilizare “assert” este pentru a verifica ca o operatie de actualizare a tabelei chiar a modificat continutul din fisier.

EX 5:

continut = select\_in\_table(db\_name, table\_name, [])

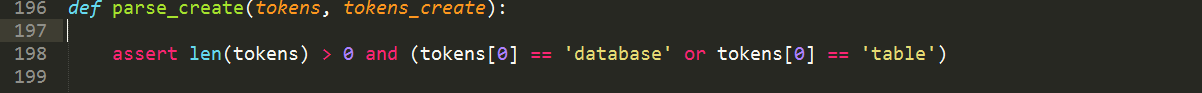
/\*update in table operation\*/

continut2 = select\_in\_table(db\_name, table\_name, [])

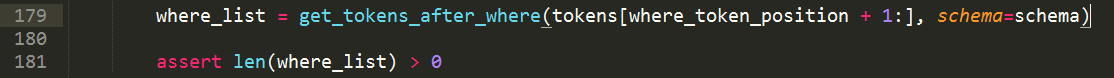
assert continut != continut2

## 5.2 Parser comenzi

În fiecare funcȚie au fost adăugate assert-uri care să valideze faptul că avem tokeni de parsat și că acei tokeni au valorile corecte.

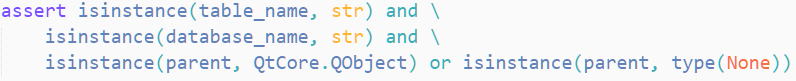


De asemenea au fost adăugate și postcondiții pentru funcțiile ajutătoare (pentru a obține tokeni aflați între paranteze și din clauze where)



## Interfața grafică

Modulul *app\_asserts.py* este derivat din modulul principal *app.py*, introducând aserțiuni. O primă aserțiune a fost adăugată la inițializarea claselor *NewTable*, *SelectRows, InsertRow*, *UpdateRows, DeleteRows, ModifyTable, ImportTable*, respectiv *ExportTable*, verificând dacă tipul variabilei pentru numele bazei de date sau a tabelului primit în constructor este de tipul *str*. (Fig. 21)

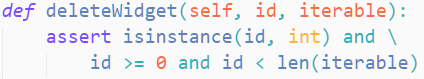


*Fig. 21*

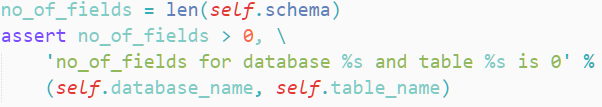
O altă aserțiune destul de folosită a fost adăugată in funcțiile care trebuiau să comparea dimensiunile unor anumite câmpuri. (Fig. 22)



*Fig. 22 a.*



*Fig. 22 b.*



*Fig. 22. c*

Pentru operațiile asupra bazelor de date (e.g. creare, ștergere), s-a verificat dacă directorul unde se află bazele de date încă este valid (i.e. nu a fost modificat/șters extern). (Fig. 23)



*Fig. 23*