Python with Databases

Databases

Simply speaking database is a form of storing and accessing data.
There are 2 main types of db

- Relational - classic, we will talk about them
- Non-relational - popular variant

Relational db

itemid	orderid	item	amount
5	1	Chair	200.00
6	1	Table	200.00
7	1	Lamp	123.12

customerid	name	email
5	Rosalyn Rivera	rosalyn@adatum.com
6	Jayne Sargent	jayne@contoso.com
7	Dean Luong	dean@contoso.com

orderid	customerid	date	amount
1	4	11/1/17	523.12
2	3	11/15/17	32.99
3	1	11/21/17	23.99

Relational db morphology

Relational database has 2 elements

- table - describes entity via fields (columns)
- link - describe relations between entities
  - 1-to-1 - row in 1st table corresponds to 1 row in the 2nd
  - 1-to-many - row in 1st table corresponds to several rows in the 2nd
  - many-to-many - 1 row can correspond to many rows from the other table and vice versa

This is a table with RUN, DNA-Chip ID and Call Rate fields

RUN	DNA-Chip ID	Call Rate
R2	202179440002_R01C02	952
R2	202179440004_R08C02	923
R2	202179440004_R09C01	912

pig_id	DNA-Chip ID	sex	breed
156	202179440002_R01C02	1	1
157	202179440004_R08C02	2	1
158	202179440004_R09C01	1	1

This is another table with pig_id, DNA-Chip ID, sex and breed fields, which linked 1-to-1 with the 1st table through DNA-Chip ID

Primary key

In addition to meaningful columns which describe your object every table should have a primary key column - service column which is just a number of row in the table (identical to pd index)

Structured Query Language

It is a quite old language. SQL is an interesting language - it is declarative - you say what you wanna get and it fetch it for you. Hence SQL can be understood by majority of people in many cases

```
SELECT item, amount FROM sell WHERE order_id = 1
```

which means - give me item and amount columns from the table sell, where order_id column value is 1

Another aspect is a variety of SQL dialects - languages with small (usually) deviations in functionality. Available types and operations can differ slightly because of this

Syntax

1. SQL is case-insensitive, but there is a convention to use uppercase
2. Commands ends with ; but it's ok to use 1 command without semicolon in the end
3. = is used for equality check
4. -- means comment

Basic commands

Data selection, let's go through it by parts

```
SELECT column FROM table WHERE condition
```

- `SELECT column` - select specified column, several columns separated with `,` can be provided or you can use `*` as in re
- `FROM table` - columns should be obtained from the specified table
- `WHERE condition` - restrict set of obtained rows with some condition

Conditions

You can use following things as conditions

- comparison - `column < value` or >, != and =
- occurrence - `column IN (value1, ...)`

We can get necessary fragments with these commands

Order and limit

Handy in printing the output

```
SELECT column FROM table WHERE condition ORDER BY column DESC LIMIT n
```

- `ORDER BY column` - sort selected data by specified column, several columns separated with `,` can be provided. By default ordered in ascending way, add `DESC` to make sorting in the descending manner
- `LIMIT n` - number of rows to select

Table management

Reviewed commands can't create a table, so let's look at another one

```
CREATE TABLE IF NOT EXISTS table_name (
                                    column1 TYPE CONSTRAINTS,
                                    column2 TYPE CONSTRAINTS,
                                    ...
                                    )
```

- `CREATE TABLE` - well … create a table with name table_name; optionally you can use `IF NOT EXISTS` which means that the table will be created if it doesn't exist

Columns format

After table name we specify column names with their types and constraints.

Type examples

- INTEGER - yes, whole numbers
- REAL - you are right, fractions
- TEXT - yep, it's analogous to string

Constraints

Restrict values of the column and specify primary keys

- PRIMARY KEY - column is a PK, which means that this column will keep numbers from 1, corresponding to the row number; PK makes this column autoincremented - you don't need to do something with it
- UNIQUE - all values in this column should be unique (no duplicates)
- NOT NULL - empty values are prohibited in that column

Python Intermedia

What do we need to work with db in python? Library!

We will use sqlite3 library for our purposes. It can connect to sqlite databases, which use lesser subset of SQL and lack some other features of postgre or MySQL DBMS. But it is powerful enough and widely used for different purposes

Good news - all libraries for database interaction follows the same contract, so your code can be reused

Main methods

Well, not very many of them

`connect(db)` - connect to specified db via its path, returns connection to db

Connection object has a number of methods

- `cursor()` - not necessary and changes nothing, skip it
- `execute(query, parameters)` - execute `query` with specified `parameters`
- `executemany(query, parameters)` - same as previous, for inserting several values
- `commit()` - write changes from the previous commit to db; takes much time and necessary to change something in db
- `close()` - release db connection

Now let's create a simple db with 1 table!

```
import sqlite3


# Create connection
connection = sqlite3.connect('learners.db')

# Create table
connection.execute('''CREATE TABLE IF NOT EXISTS learners (
   learner_id INTEGER PRIMARY KEY,
   first_name TEXT,
   last_name TEXT,
   sex INTEGER
   )''')
```

```
# Commit changes
connection.commit()

# Close connection
connection.close()
```

And let's go to the DB Browser for SQLite and look at it
Press Open Database and select learners.db

Adding data

To add data we need another command

```
INSERT INTO table VALUES (column1_value, ...)
```

- `INSERT INTO table` - insert something to the specified `table`
- `VALUES (column1_value, ...)` - specify values which will be inserted into the table (number of values should be equal to number of columns; you can skip primary key column)

```
# Create connection
connection = sqlite3.connect('learners.db')

# Insert values
query = "INSERT INTO learners (first_name, last_name, sex)
                        VALUES ('Dima', 'Biba', 0)"
connection.execute(query)

# Commit changes
connection.commit()

# Close connection
connection.close()
```

What if you want to use variables?

Don't do this

```
fname, lname, sex = 'Shamil', 'Urazbakhtin', 0
query = f"INSERT INTO learners (first_name, last_name, sex)
                        VALUES ({fname}, {lname}, {sex})"
connection.execute(query)
```

Placeholders

Previous way is vulnerable to some abuses. Use placeholders instead

```
# Insert with placeholders, pass values as collection to
parameters in execute
query = "INSERT INTO learners (first_name, last_name, sex)
                        VALUES (?, ?, ?)"
connection.execute(query, ['Shamil', 'Urazbakhtin', 0])

# Commit changes
connection.commit()

# Close connection
connection.close()
```

Executemany

Used for multiple inserts

```
learners = [('Loli', 'Alekseeva', 1),
            ('Katya', 'Yakovleva', 1),
            ('Olya', 'Mazur', 1)]

query = "INSERT INTO learners (first_name, last_name, sex)
                       VALUES (?, ?, ?)"
connection.executemany(query, learners)
```

Links

Now let's add 2 table with courses and attendance, which will have the following structure

- course_id
- course

and

- attendance_id
- course_id
- learner_id

```
connection.execute('''CREATE TABLE IF NOT EXISTS courses (
    course_id INTEGER PRIMARY KEY,
    course TEXT
    )''')

connection.execute('''CREATE TABLE IF NOT EXISTS attendance (
    attendance_id INTEGER PRIMARY KEY,
    course_id INTEGER,
    learner_id INTEGER
    )''')

# Fill courses table with data
query = "INSERT INTO courses (course) VALUES (?)"
connection.executemany(query, (['python'], ['bioinformatics']))
```

attendance table is going to keep records to which class who signed up. We use INTEGER as datatype because we will use ids of learners and courses instead of their names. Why?

Because storing a bunch of integers is much easier than storing such number of strings. This is related to the concept of normalization

Next, as you can see our tables are somehow related - courses and learners tables store data about courses and learners respectively, while attendance use course and student from these tables

1 student can visit many courses and 1 course can be visited by many students

Foreign keys

Links are implemented with foreign keys, which are specified in the table creation. Thus let's recreate the db (you can delete it, or execute next queries)

Drop

Execute this statement for 3 tables in our db - it will delete them

```
DROP TABLE IF EXISTS table_name
```

- `IF EXISTS` has the same meaning as before
- `table_name` is a name of table we wanna delete

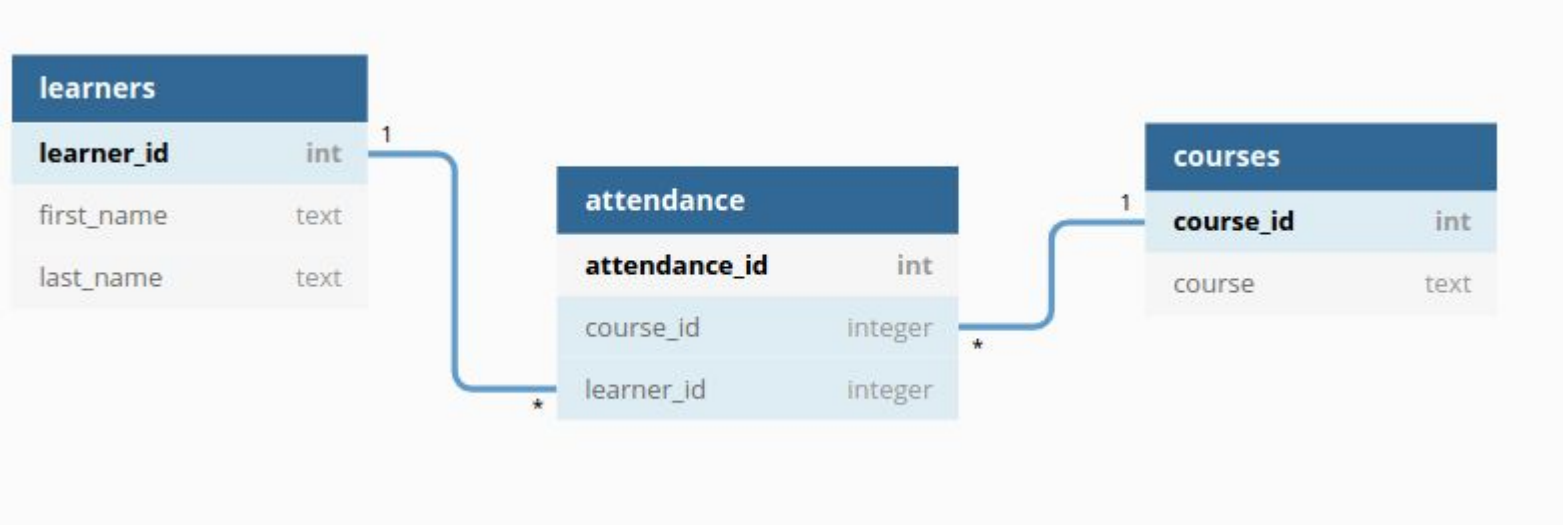
```
# Unique constraint on combination of several columns
connection.execute('''CREATE TABLE IF NOT EXISTS learners (
  learner_id INTEGER PRIMARY KEY,
  first_name TEXT,
  last_name TEXT,
  sex INTEGER,
  UNIQUE (first_name, last_name, sex)
  )''')

connection.execute('''CREATE TABLE IF NOT EXISTS courses (
  course_id INTEGER PRIMARY KEY,
  course TEXT UNIQUE
  )''')
```

```
connection.execute('''CREATE TABLE IF NOT EXISTS attendance (
  attendance_id INTEGER PRIMARY KEY,
  course_id INTEGER,
  learner_id INTEGER,
  FOREIGN KEY (course_id) REFERENCES courses (course_id),
  FOREIGN KEY (learner_id) REFERENCES learners (learner_id)
  )''')
```

Schema

Here is the schema of our db now



```
# This means that student with id 1 visits course
  with id 1
connection.execute('INSERT INTO attendance
                    (course_id, learner_id) VALUES (1, 1)')
```

How to understand what is the course, who is it and so on?

	attendance_id	course_id	learner_id
	Filter	Filter	Filter
1	1	1	1

Join

Everything the same as in pandas, we just have tables instead of dfs here. And notation is more verbose, cause join is a part of SELECT

```
SELECT … JOIN another_table using(shared_column)
```

```
SELECT … JOIN another_table on table.col1 = another_table.col2
```

Both variants are valid

- in the 1st we have column with the same name on which we want to join
- in the 2nd we specify condition of joining

```
# Query examples
SELECT * FROM attendance
JOIN learners USING(learner_id)


SELECT course, first_name, last_name, sex FROM attendance
JOIN learners ON attendance.learner_id = learners.learner_id
JOIN courses ON attendance.course_id = courses.course_id
```

Fetching the results

In previous examples when we execute SQL code from python we usually change our database (create tables or insert rows) but haven't receive any results back. Select queries return subset of rows and we should be able to process them in python, so how does it handle?

We use fetch methods family for this purpose. All of them are applied after standard `execute`. Each row is represented as a tuple (or more sophisticated ordered object)

- `fetchall()` - returns iterable with all rows obtained with the query
- `fetchone()` - returns one row
- `fetchmany(size)` - returns list with number of rows specified in `size`

```
# Get attendance data
query = '''SELECT attendance_id, course, first_name, last_name
           FROM attendance
           JOIN learners ON attendance.learner_id =
learners.learners_id
           JOIN courses ON courses.course_id =
attendance.course_id'''
res = connection.execute(query)

# Look at rows
for row in res.fetchall():
   print(row)

(1, 'python', 'Dima', 'Biba')
```

There is much more to investigate in this field