Instructions:

This is an individual assignment. The first four are in-lab, the last one is take home.

Download and unzip the file assignment-jan28.zip.

Rename the directory to <your *rollnumber*>_*jan28*.

When you are ready to submit to moodle, zip up this directory and upload the rollnum_jan28.zip file to the inlab or take home assignments on moodle, as appropriate.

The exercises below will ask you to fill in the TODO sections of relevant files.

Reflection and Metadata

Data that describes data is called *metadata*. For example, in Java, you can supply an object reference to the java.lang.reflect API and learn about the class of the object, the fields and methods supported by the class, their exact type signatures, and so on. There are two important aspects of this facility: (a) The descriptions are available to a program at run time, unlike say, in C, where it is hidden inside the C compiler. (b) The descriptions are expressed as structures of ordinary Java objects.

A system that provides metadata is called *reflective*. A reflective system allows the user to build tools to dynamically discover properties of objects and to interact with them.

Python supports similar facilities with built-in functions such as *type(), dir()* and *isinstance()*. It makes even the documentation available at run time (__doc__ attribute). Using reflection, the end user can create new tools, such as interactive shells (such ipython or jshell).

Reflection and databases

The SQL92 standard specifies a way to dynamically explore the structure of tables, columns, constraints, views and other objects in a database. Instead of providing an API, this meta information is itself available in a set of standard tables! We will use only the following tables for the following exercises.

a. **Informaton_schema.columns**: It stores information about all columns of all tables in the system. We will only need those tables where schema = ‘public’ (user-defined)
b. **Information_schema.table_constraints:** A *constraint* is simply a list of column names that together have a common property, such as primary key, foreign key and so on.The

table_constraints table maps a table name to the list of constraints defined on it. A table can have many constraints defined on it, and at most one primary key constraints.

c. **Information_schema.key_column_usage**: Maps constraint_name to a list of table+column names. You will need only those constraints where constraint_type = 'PRIMARY KEY' or 'FOREIGN KEY'.
d. **information_schema.referential_constraints.** This table associates a foreign key constraint (from one table) to a primary or unique key constraint on another table. This way we can discover which table refers to another table.

Exercise 1a (XData inlab). Metadata exploration in psql

Get a list of all user-defined tables, their columns and corresponding datatype. Mark a column with an asterisk if it is part of a primary key. The desired output is as follows:

```
 table_name | column_name  | is_pk |     data_type
------------+--------------+-------+-------------------
 advisor    | i_id         |       | character varying
 advisor    | s_id         | *     | character varying
 classroom  | capacity     |       | numeric
 classroom  | building     | *     | character varying
 classroom  | room_number  | *     | character varying
 course     | title        |       | character varying
 ...
```

Hint: Table (a) above gets you everything except `is_pk`. Table (b) and (c) together tell you which columns are primary keys. Consider using "case" expressions to display '*'.

Submit this query using xdata, but just use the relation name without the prefix "information_schema.'.

Exercise 1b (XData inlab). Recursive queries.

The XData assignment also has a question on recursive queries. Details are on XData.

Python - DB access

The next exercise is to obtain the same information from a python script. Python has a "DB2.0" specification that standardizes access to a database.

Exercise 2 (Inlab): postgres metadata in python

We will use python3 and the psycopg2 driver for postgres.

2a. Basic DB access.

1. pip3 install psycopg2
2. Load the university data into your database if it is not already there. https://www.db-book.com/db7/university-lab-dir/sample_tables-dir/index.html
3. Modify config.py and fill in postgres connection parameters. This is done so that we can run your code using our configuration
4. Fill in the TODO parts of config.py and dbexec.py and run it as follows

   ```
   > python3 dbexec.py  'select * from student'.
   ```

   Expected output is of the form;

   ```
   id,name,dept_name,tot_cred
   24746,Schrefl,History,4
   79352,Rumat,Finance,100
   76672,Miliko,Statistics,116
   ```

You will need the following methods/fields from different objects in the psycopg2 driver: *psycopg2.connect()*, *cursor(), cursor.execute()*, *cursor.fetchall(), cursor.description*.

For documentation, see: https://www.python.org/dev/peps/pep-0249/

Exercise 2b. Obtaining metadata

1. Fill in the TODO parts of pgmeta.py. It uses the information_schema tables (from exercise 1) and packages them in terms of Meta, Table, and Column objects. These classes are defined in `dbmeta.py` for your convenience; see the usage information at the beginning.
   Please collect metadata only for non-system classes, that is, those for which schema = 'public'
2. Run `python3 pgmeta.py`
   Expected output is like this:

   ```
   [advisor|s_id|i_id]
   [advisor] -> [student]
   [advisor] -> [instructor]
   [classroom|building|room_number|capacity]
   [course|course_id|title|dept_name|credits]
   ```

```
[course] -> [department]
[department|dept_name|building|budget]
…
```

Step 2c. Visualization

Go to nomnoml.com; the text on the page is editable. Delete that text and replace it with the output from the previous step. You should see a graph. In the next exercise, we will get the graph from a python based web server.

Python http

Exercise 3 (Inlab). Introduction to python http library

We will use create an *extremely* rudimentary web server to serve up the information obtained in the previous exercise. The objective of this exercise is to acquaint you (or remind) you of the http protocol)

Steps:

1. Fill in the TODO section in the pyweb.py file.
2. From a browser go to http://localhost:8000/meta. You should see the exact output from the previous exercise. This is because the output of pgmeta.get_meta_data is wrapped up inside <pre></pre>

Python + JS + Visualization

Exercise 4 (**Take-home**). Understanding the node ecosystem

The objective of this exercise is to understand the modern way in which javascript code is packaged and made available to the browser. See pgweb2.py, a slight modification of the previous exercise. The code remains identical, and only the html sent to the browser is different.

Steps:

1. `> npm`. Verify that it is version 6.13.
   `> node`. Verify that it is version 13.6
2. `> mkdir view` # This will contain all the browser-related view elements.
3. `> cd view`
4. `> npm install webpack; npm install nomnoml`
5. `> mkdir src  # Under view`
6. Create a file `src/index.js` with the line “global.
7. nomnoml = require(‘nomnoml’)
8. Run webpack (in the view directory)
   `> ./node_modules/.bin/webpack`
   This should create a dist/main.js
9. Modify pyweb2.py. As mentioned above, only the html sent to the browser is different; the python code does not need any modification. See https://github.com/skanaar/nomnoml for an example of what the html should look like.

Exercise 5. (**Take-home)** Two-way interaction, html forms + db

This exercise introduces HTML forms. When the server is contacted at http://localhost:8000/add, one should be able to select a table a dropdown list of tables, and then fill in a form containing a field for each column. Once the user hits submit, the backend inserts the corresponding record. All the html is generated dynamically.

Now for some details.

1. Open ex5-sample.html. This is the desired view. You need to produce dynamic variations of this for any table. **It is essential that you follow the same format, for ease of automated evaluation.**
2. Note that it has two forms. The upper part contains a drop-down list of table names and a select table button. The lower form shows a set of fields that are specific to a selected table specific to the student table, and an add button.
3. Note that the first time, there is no table selection, so the bottom form isn’t shown. When a table is selected, the corresponding set of field names and text fields are shown. When the user hits Add on the browser, the do_POST() method gets called. It retrieves the user input data from the message, creates a SQL insert query dynamically and executes it. If successful, it responds with the same form with fields emptied. If unsuccessful, it responds with the form filled as the user had done.
4. We have not supplied any code for this exercise, but you can start with pyweb2 and add a do_POST method to handle submissions.

Corrections in zip file.

1. dbmeta.py: Change db_exec to dbexec
2. dbmeta.py: If your version of python3 is old, it is likely that you will get an error in the first line. If so, comment out “from __future__ import annotations”, and add the following two declarations:

```
class Column: pass
class Table: pass
```