# Swiss Tournament Project
**by Lydia Kalinkina, IPND student, Stage 5, back-end path**
**Project is located at:**
https://github.com/playwriter/fullstack-nanodegree-vm.git

## Overview

Working on this project involved mastering the relational databases and the PostgreSQL database in particular, and writing and running a Python module that uses the database. The goal set was to keep track of players and matches in a Swiss system tournament.

For educational purposes Udacity provided students with the basis, namely: with detailed description including design notes, the repo and templates for this project including tournament.sql, tournament.py, and tournament_test.py. The template file tournament.py contains stubs of several functions. Each function has a docstring that says what it should do. The template file tournament.sql required to put the database schema in the form of SQL create table commands. The file tournament_test.py contains unit tests to test the functions written in tournament.py.

Technical requirements concerned installing and running a set of virtual machines, such as:
To use this code, you must have PostgreSQL and Python installed.

Then VirtualBox, installed from https://www.virtualbox.org/wiki/Downloads,

Vagrant, installed from https://www.vagrantup.com/downloads, and
Cloning of Vagrant VM for ud197 (git clone http://github.com/udacity/fullstack-nanodegree-vm fullstack).

**README** part should describe how to run the project.

In a nutshell it comes down to the following.
Copying tournament.py, tournament.sql, and tournament_test.py into the folder vagrant/tournament folder
SSH into the virtual machine using Vagrant SSH
From the vagrant/tournament folder in the shell, run psql -f tournament.sql to build the tables and view
From the same folder, run python tournament_test.py to test the schema.
To run the test suite (exercising all of the Python functions for the tournament database):
From a GitHub shell:

cd fullstack/vagrant

1. vagrant up (you can turn off the VM with 'vagrant halt')
2. vagrant ssh (from here you can type 'exit' to log out)
3. cd /vagrant/tournament
4. psql -f tournament.sql
5. python tournament_results.py

It can be added that one must make sure database tournament exists. For that can be used the following.

```bash

vagrant@vagrant-ubuntu-trusty-32:/vagrant/tournament$ psql
```

```
psql (9.3.5)
Type "help" for help.
vagrant=> CREATE DATABASE tournament;
CREATE DATABASE
vagrant=> \q
```

load SQL schema

```bash
vagrant@vagrant-ubuntu-trusty-32:/vagrant/tournament$ psql tournament < tournament.sql
```

run test

```bash
vagrant@vagrant-ubuntu-trusty-32:/vagrant/tournament$ python tournament_test.py
```

If everything works you receive: “Success! All tests passed!”
This is what I received when just began working on the project: “

```
1. Old matches can be deleted.

2. Player records can be deleted.

3. After deleting, 1:52 19.03.2016() returns zero.

4. After registering a player, countPlayers() returns 1.

5. Players can be registered and deleted.

6. Newly registered players appear in the standings with no matches.

7. After a match, players have updated standings.

8.After one match, players with one win are paired.

Success!  All tests pass!
```

After that I ran into a series of problems. To begin with program asked: “could not connect to server: Connection refused (0x0000274D/10061). Is the server running on host "localhost" (127.0.0.1) and accepting TCP/IP connections on port 5432?” I was not alone, and I read posts where people complained that hey received exactly the same messages. There was also an Error connecting to the database.

This is why, after running through quite a few resources, I included helpers programs and comments into my tournament.py program.

Below, please, find my codes for this project.

## tournament.py

```
#!/usr/bin/env python
#
# tournament.py -- implementation of a Swiss-system
tournament
#
import psycopg2
import psycopg2.extensions
from psycopg2.extensions import b
# we have to import the Psycopg2 extras library!
import psycopg2.extras
import sys
import collections
import itertools
from random import sample, choice, randrange
from operator import itemgetter, mul
from itertools import starmap, repeat, chain, cycle, tee, \
    groupby, count, combinations, starmap, islice
try:
    from itertools import imap as map, izip as zip, ifilter as filter, \
        izip_longest as zip_longest, ifilterfalse as filterfalse
except ImportError as err:
    from itertools import zip_longest, filterfalse
def connect():
    """Connect to the PostgreSQL database.  Returns a database
connection."""
def main():
          #Define the connection string
          conn_string = 'host="localhost" dbname="tournament"
user="postgres" password="secret"'
          # print the connection string to be used to connect
          print "Connecting to database\n   ->%s" % (conn_string)
          # get a connection, if a connect cannot be made an
exception will be raised here
          conn = psycopg2.connect(conn_string)
          # We refactor our connect() method
          # to deal not only with the database connection
          # but also with the cursor
          # since we can assign and return
          # multiple variables simultaneously.
          try:
                    db =
psycopg2.connect("dbname={}".format(database_name))
                    cursor = db.cursor()
                    return db, cursor
          except:
                    print("Error when connecting the server")
          # By specifying a name for the cursor
          # psycopg2 creates a server-side cursor, which prevents
all of the
          # records from being downloaded at once from the server.
          cursor = conn.cursor('cursor_unique_name',
cursor_factory=psycopg2.extras.DictCursor)
          cursor.execute('SELECT * FROM table LIMIT 1000')
          # Because cursor objects are iterable we can just call 'for -
in' on
          # the cursor object and the cursor will automatically
advance itself
          # each iteration.
          # This loop should run 1000 times, assuming there are at
least 1000
          # records in 'my_table'
          row_count = 0
          for row in cursor:
                    row_count += 1
          print "row: %s    %s\n" % (row_count, row)
          # conn.cursor will return a cursor object; this cursor will
          # be used to perform queries
          print "Connected!\n"

if __name__ == "__main__":
          (main)

def deleteMatches():
    """Remove all the match records from the database."""
    db = psycopg2.connect("dbname=tournament")
    c = db.cursor()
    query = "TRUNCATE matches;"
    cursor.execute(query)
    db.commit()
    db.close()
def deletePlayers():
    """Remove all the player records from the database."""
    conn = connect()
    cursor = conn.cursor()
    query = "DELETE FROM players;"
    cursor.execute(query)
    conn.commit()
    conn.close()
def countPlayers():
    """Returns the number of players currently registered."""
    conn = connect()
    cursor = conn.cursor()
    query = "SELECT count(*) AS num FROM players;"
    cursor.execute(query)
    players_count = cur.fetchone()[0]
    conn.commit()
    conn.close()
    return count
def registerPlayer(name):
    """Adds a player to the tournament database.
    The database assigns a unique serial id number for the
player. (This
      should be handled by your SQL database schema, not in your
Python code.)
      Args:
      name: the player's full name (need not be unique).
      """
      conn = connect()
      cursor = conn.cursor()
      query = "INSERT INTO players (name) VALUES (%s);"
      parameter = (name,)
      cursor.execute = (query, parameter)
      conn.commit()
      conn.close()
def playerStandings():
      """Returns a list of the players and their win records, sorted by
wins.
      The first entry in the list should be the player in first place, or a
player
      tied for first place if there is currently a tie.
      Returns:
      A list of tuples, each of which contains (id, name, wins,
matches):
      id: the player's unique id (assigned by the database)
      name: the player's full name (as registered)
      wins: the number of matches the player has won
      matches: the number of matches the player has played
      """
      conn = connect()
      cursor = conn.cursor()
      query = ("SELECT players.id, players.name,
COUNT(matches.winner = players.id) AS wins, "
             "Count(matches.*) AS games"
             "FROM players LEFT JOIN matches "
             "ON players.id = matches.winner OR players.id =
matches.loser,"
             "GROUP BY players.id, players.name "
             "ORDER BY wins DESC")
      # Because cursor objects are iterable we can just call 'for - in'
on
      # the cursor object and the cursor will automatically advance
itself
      # each iteration.
      # This loop should run as many times as there are
      # records in the table.
      parameter = (wins,)
      cursor.execute(query, parameter)
      row_count = 0
      for row in cursor:
          row_count += 1
          print "row: %s    %s\n" % (row_count, row)
      playerStandings = cur.fetchall() #Fetches all remaining rows of
a query result, returning a list.
```

```
    conn.close()
    return playerStandings
def reportMatch(winner, loser):
    """Records the outcome of a single match between two
players.
    Args:
    winner: the id number of the player who won
    loser: the id number of the player who lost
    """
    conn = connect()
    cursor = conn.cursor()
    query = "INSERT INTO matches (winner_id, loser_id)
VALUES (%s, %s);"
    parameter = (winner, loser)
    cursor.execute(query, parameter)
    conn.commit()
    conn.close()
def swissPairings():
    """Returns a list of pairs of players for the next round of a
match.

    Assuming that there are an even number of players
registered, each player
    appears exactly once in the pairings.  Each player is paired
with another
    player with an equal or nearly-equal win record, that is, a
player adjacent
    to him or her in the standings.

    Returns:
      A list of tuples, each of which contains (id1, name1, id2,
name2),
      first player's unique id
      name1: the first player's name
      id2: the second player's unique id
      name2: the second player's name
    """
    # For swissPairings consulted GitHub, Stack Overflow
    # and the recipes section of Python's
    # itertools docs: https://docs.python.org/2/library/itertools.html
    # and the Python Standard Library.
    # Iterate through the list and build the pairings to return results
    db = psycopg2.connect("dbname=tournament")
    cursor = conn.cursor()
    query = ("SELECT players.id, players.name,
COUNT(matches.winner = players.id) AS wins, FROM players
LEFT JOIN matches, GROUP BY players.id, players.name
ORDER BY Wins;")
    parameter = (id, name, matcheup, wins)
    cursor.execute(query, parameter)
    ids   = [ x[0] for x in c.fetchall () ] # unpack tuples
    names = [ x[0] for x in c.fetchall () ] # unpack tuples
    pairs = zip ([ x for x, in ids], [x for x, in names])
    results = []
    pair = []
    standings = playerStandings()
    # standings = [(id1, name1, wins1, matches1), (id2, name2,
wins2, matches2)]
    # [id1, id2, id3, id4, id5, id6, id7, id8] = [row[0] for row in
standings]
    # pairings = swissPairings()
    pairingsiterator = itertools.izip(*[iter(standings)]*2)
    pairings = list(pairingsiterator)
    for pair in pairings:
        id1 = pair[0][0]
        name1 = pair[0][1]
        id2 = pair[1][0]
        name2 = pair[1][1]
        matchup = (id1, name1, id2, name2)
        results.append(matchup)
    return results
```

### tournament.sql

```
-- Table definitions for the tournament project.
--
-- Put your SQL 'create table'
statements in this file; also 'create view'
-- statements if you choose to use it.
--
-- You can write comments in this file by
starting them with two dashes, like
-- these lines here.
-- Clear out any previous tournament
databases.
DROP DATABASE IF EXISTS
tournament;
-- Create database.
CREATE DATABASE tournament;
-- Connect to the DB before creating
tables.
\c
 tournament;
-- Create table for players.
CREATE TABLE players (
id serial PRIMARY KEY,
name text[]
);
-- Create table for games.
CREATE TABLE matches(
game_id serial primary key,
winner integer REFERENCES
players(id),
loser integer REFERENCES
players(id),
);
-- Create view to show standings.
CREATE VIEW standings AS
SELECT players.id,
players.name,
COUNT(matches.winner = players.id)
AS wins,
COUNT(matches.*) AS games
FROM players LEFT JOIN matches
ON players.id = matches.winner OR
players.id = matches.loser
GROUP BY players.id, players.name
ORDER BY wins DESC;
```

## References

To name a few:

1) https://en.wikipedia.org/wiki/Swiss-system_tournament
2) itertools docs: https://docs.python.org/2/library/itertools.html and the Python Standard Library.
3) http://www.postgresql.org/docs/9.3/static/sql-syntax-lexical.html(section 4.1.1)
4) http://stackoverflow.com/questions/2878248/postgresql-naming-conventions
5) http://docs.writethedocs.org/writing/beginners-guide-to-docs/
6) http://www.sphinx-doc.org/en/stable/rest.html#rst-primer
7) Online documentation: http://docs.writethedocs.org/
8) Conference: http://conf.writethedocs.org/