Lecture Two: Object-oriented programming SQL Databases JSON Files (Notebook) Exercise Two-Objected-oriented programming

Object Oriented Prorgamming A program is made up of many cooperating objects, instead of being the "whole program" - each object is a little "island" within the program and cooperatively working with other objects, a program is made up of one or more pbjects working together - objects make use of each other's capabilities.

Object is a self-contained Code and Data A Key aspect of the Oject approach is to break the prblem into smaller understandable parts (divide and conquer) Objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects, even function is Objects....

Some Definitions Class is a template Method or Message - A defined capability of a class; Field or attribute - A bit of data in a class; Object or Instance - A particular instance of a class;

Terminology: Class, the class contains all the thing's characteristics including its attributes, fields, and the thing's behaviour- what the thing can do, which is methods, operations, features.

Class is a blueprint or factory that describe the nature of something. Example, the class Dog would consist of traits shared by all dogs breed and fur color (characteristics) and the ability to bark and sit (behaviours).

Terminology: Instance: Instance of the class is a particular object. instance is the actual object created at runtime. In dog class, Lassie (the name) is an instance. The set of values (the instance) of attributes of a particular object is called it's state.

Terminoligy: Methods: Method is verb. a method usually affects one particular object; For example, all dogs can bark, only need to call one particular dog bark.

双击（或按回车键）即可修改

```
x = 10
def my_function():
    pass
print(dir())
```

```
['In', 'Out', 'User', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', '_dh', '_i', '_i1', '_i10', '_i11', '_i2', '_i3', '_i4', '_i5', '_
```

```
dir(x)
```

```
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 '__dir__',
 '__divmod__',
 '__doc__',
 '__eq__',
 '__float__',
 '__floor__',
 '__floordiv__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__index__',
 '__init__',
 '__init_subclass__',
 '__int__',
 '__invert__',
 '__le__',
 '__lshift__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__neg__',
 '__new__',
 '__or__',
 '__pos__',
 '__pow__',
 '__radd__',
 '__rand__',
 '__rdivmod__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rfloordiv__',
 '__rlshift__',
 '__rmod__',
 '__rmul__',
 '__ror__',
 '__round__',
 '__rpow__',
 '__rrshift__',
 '__rshift__',
 '__rsub__',
```

```
'__rtruediv__',
'__rxor__',
'__setattr__',
'__sizeof__',
'__str__',
```

dir(y)

```
x = 'abc'
type(x)

type(2.5)

type(2)

y = list()
type(y)

z = dict()
type(z)
```

```
dict
```

```
dir(x)
dir(y)
dir(z)
```

```
['__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__ior__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__ne__',
 '__new__',
 '__or__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__ror__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

```
# A Sample Class



# Class is a reserved word that defines a template for making objects

class PartyAnimal:
  def __init__(self):
    self.x = 0
  def party(self):
    self.x = self.x + 1
    print("So far", self.x)
# Each PartyAnimal object has a bit of code

an = PartyAnimal()
# Construct a PartyAnimal Object and store in an
an.party()
```

```
an.party()
an.party()
# Tell the an object to run the party() code with it.
```

```
So far 1
So far 2
So far 3
```

```
y = list()
type(y)
dir(y)
```

```
['__add__',
 '__class__',
 '__class_getitem__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'clear',
 'copy',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

```
u = 'Hello Word'
dir(u)
```

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
```

```
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isascii',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join'
```

Object life cycle

Objects are created, used, and dicarded we have spcial blocks of code (methods) that get called -At the moment of creation (constructor)
-At the moment of destruction (destructor) construct a lot, destrcutor are seldom used

```
class PartyAnimal:
    def __init__(self):
        self.x = 0
        print('I am constructed')

    def party(self):
        self.x = self.x + 1
        print('So far', self.x)

    def __del__(self):
        print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains', an)
```

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

```
class PartyAnimal:
  def __init__(self, z):
    self.x = 0
    self.name = z
# Constructors can have additional paramaters. These can be used to set up instance variables for the particular instance of the class (i.e., for the particular object)
    print(self.name, "constructed")
  def party(self):
    self.x = self.x + 1
    print("So far","party count", self.x)
s = PartyAnimal("Sally")
s.party()
j = PartyAnimal("Jim")
j.party()
s.party()
# Inheritance: When we make a new class - we can reuse an exitsing class and inherit all the capabilities of an existing class and then add our own bit to make our new clas
# Another form of store and resue
# Write once - resue many times
# The new class (child) has all the capabilities of the old class (parent) - and then some more.
```

```
Sally constructed
So far party count 1
Jim constructed
So far party count 1
So far party count 2
```

```
class PartyAnimal:

  def __init__(self, nam):
    self.x = 0
    self.name = nam
    print(self.name, "constructed")
  def party(self):
    self.x = self.x + 1
    print(self.name, "party count", self.x)

class FootballFan(PartyAnimal):
# FootballFam is a class which extends PartyAnimal. It has all the capabilities of PartyAnimal and more.
  def __init__(self, nam):
```

```
        super().__init__(nam)
        self.points = 0
    def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name, "points", self.points)

s = PartyAnimal("Sally")
s.party()
j = FootballFan("Jim")
j.party()
j.touchdown()
```

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim party count 2
Jim points 7
```

Exercise 2: Object-Oriented Design + File & DB Integration (during tutorial 1) Goal: Apply OOP principles to the project and introduce persistent storage.

Concepts Covered:

 Classes and methods

 JSON file handling

 SQLite with sqlite3

Task Part A: Classes

1. Define a Property class with attributes:

o property_id, location, type, price_per_night, features, tags 2. Define a User class with:

o user_id, name, group_size, preferred_environment, budget

o Method: matches(self, property_obj) – returns True if property is a good fit

```
# Exercise Two Question One
class Property:
  def _init_(self, property_id, location, type, price_per_nity, feature, tags):
    self, property_id = property_id
    self, location = location
    self, type = type
    self, price_per_nity = price_prent_nity
    self. feature = feature
    self.tags = tags

class User:
  def _init_(self, user_id, name, group_size, preferred_enviornment, budget):
    self, user_id = use_id
    self, name = name
    self, group_size = group_size
    self. preferred_enviornment = preferred_enviornment
```

Task Part B: JSON File I/O

1. Create several Property and User objects and convert them to dictionaries.
2. Save these to JSON files using json.dump().
3. Load and recreate the objects from the files using json.load().

双击（或按回车键）即可修改

We can group data and functionality together and create many independent instances of a class

File Processing

A text file can be thought of as a sequence of lines

Use Open() to handle

File handle as a sequence counting lines in a File Reading the Whole File

Searching through a File

OOPS!

Relaitional Databases

Relational databases model data by storing rows and columns in tables. The power of the relational database lies in its ability to efficiently retrieve data from those tables and in particular where there are multiple tables and the relationships between those tables involved in the query.

SQL deals with the different tables to efficiently convert, abstract, seperate and mergers different tables

Additional Notes: A relational database is a set of tuples and each tuple is a row, like an object or a dictionary contains different attributes on self, in the table.

Ternimology

database contains many tables

Relation is table - contains tuples and attributes

Tuple is a row

Attributes - is one of the many elements of data corresponding to the object represented by the row.

A relation (table) is a set of tuples (objects) that have the same attributes(columns). All the date referenced by an attribute are in the same domain and conform to the same constraints.

SQL

SQl is a language that we issue commands to the database:

- Creat data (aka insert)
- Retrieve data (merger and sepreate datasheets)
- Update data
- Delect data

Take the hospital data spreadsheet as an example, it contains eight different relations, more restrictive, and each relation is related with other relations, if we delect appointment in subsheet, we have to delect examination by doctors from the mothersheet.

```
```

Input Files --> Python Programs <-- SQL --> Database File <--> SQLite Broswer| |

```
            |
            Outout  Files  (R,  Excel,  D3.js)
```

End User <---> Application Software(to developer) <- SQL -> Database Data Server <-SQL-> Database Tools --> DBA

SQLite Broswer

- SQLite is a very popular database - it is free and fast and small
- SQLite Browser allows us to directly manipulate SQLite files
- SQLite is embedded in Python and a numebr of other languages,

SQL: Create

```
import sqlite3

conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()

cursor.execute('''
CREATE TABLE IF NOT EXISTS Users (
  name VARCHAR(128),
  email VARCHAR(128)
)
''')

conn.commit()
conn.close()
```

```
###In Pyrhon: Setup a connection and create table

def setup_database():
# Create a connection to the SQLite database
    conn = sqlite3.connect('user.db')
    #Create a cursor object
    cursor = conn.cursor()
    #Create a table for storing user information
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS Users (
        id INTEGER PRIMARY KEY,
        name TEXT,
        age INTEGER
        gender TEXT
        location TEXT
        interests TEXT
        liked_users TEXT
        disliked_users TEXT
```

```
    matches TEXT
)
''')
#Commit the changes and close the connection
conn.commit()
conn.close()
```

SQL: Insert

The insert stareent inserts a rwo into a table

```
import sqlite3

conn = sqlite3.connect('mydatabase.db')
cursor = conn.cursor()

cursor.execute('''
INSERT INTO Users (
  name, email) VALUES('Kristin', 'kf@umich.edu')

''')

conn.commit()
conn.close()
```

In Python: Insert

```
#In Python: Insert
cursor.execute('''
INSERT INTO users (
  user_id, name, age, gender, location, interests, liked_users, disliked_users, matches)
  VALUES (?,?,?,?,?,?,?,?,?)
''', (user.user_id, user.name, user.age, user.gender, user.location, ','.join(user.interests), liked_users, disliked_users, matches))
```

```
import sqlite3

# Assuming you have a User class defined like this:
class User:
    def __init__(self, user_id, name, age, gender, location, interests, liked_users=None, disliked_users=None, matches=None):
        self.user_id = user_id # This user_id is part of the object, not necessarily the database row id
        self.name = name
        self.age = age
        self.gender = gender
        self.location = location
        self.interests = interests if interests is not None else []
        self.liked_users = liked_users if liked_users is not None else []
        self.disliked_users = disliked_users if disliked_users is not None else []
        self.matches = matches if matches is not None else []


# Create a User object
user = User(1, 'John Doe', 30, 'Male', 'New York', ['Hiking', 'Reading'])

# Convert the lists to strings for storage
liked_users_str = ','.join(user.liked_users)
disliked_users_str = ','.join(user.disliked_users)
matches_str = ','.join(user.matches)


conn = sqlite3.connect('user.db') # Connect to the database you created in cell YZRBFEWK_3G8
cursor = conn.cursor()

# Corrected INSERT statement to match the schema in cell YZRBFEWK_3G8
cursor.execute('''
INSERT INTO Users (
  name, age, gender, location, interests, liked_users, disliked_users, matches)
  VALUES (?,?,?,?,?,?,?,?)
''', (user.name, user.age, user.gender, user.location, ','.join(user.interests), liked_users_str, disliked_users_str, matches_str))

conn.commit()
conn.close()

print("User data inserted successfully!")
```

SQL Delete

Deletes a row in a table based on selection criteria

```
DELETE FROM Users WHERE email = 'ted@umich.edu'
```

```
  File "/tmp/ipython-input-3159452667.py", line 1
    DELETE FROM Users WHERE email = 'ted@umich.edu'
           ^
SyntaxError: invalid syntax
```

In Python: Delete (Complex - Cascading '级联' effect)

```
# Now delete the user from the database
cursor.execute('DELECT FROM users WHERE user_id = ?', (user_id))
```

SQL: Update

Allows the updating of a field with a where clause

```
UPDATE Users SET name = 'Charles' WHERE email = 'csev@umich.edu'
```

In Python: Update

```
cursor.execute('''
UPDATE users SET liked_users = ?, disliked_users = ?, matches = ?, WHERE user_id = ?
''', (liked_users, disliked_users, matched, user,user_id))
```

Retrieving Records: Select

The select statement retrieves a group of records - you can either retrieve all the records or a subset of the records with a WHERE clause

```
SELECT * FROM Users

SELECT * FROM Users WHERE email='csev@umich.edu'
```

```
  File "/tmp/ipython-input-2848372393.py", line 1
    SELECT * FROM Users
             ^
SyntaxError: invalid syntax
```

```
### In Python is Fetch
```

```
def fetch_user(user_id):
  conn = sqlite3.connect('user.db')
  cursor = conn.cursor()
  cursor.execute('SELECT * FROM users WHERE user_id = ?', (user_id,))
  user_data - cursor.fetchone()
  conn.close()

  if user_data:
    user_id, name, age, gender, location, interests, liked_users, disliked_users, matches = user_data

    # Convert comma-separated strings back to lists
    interests = interests.split(',') if interests else []
    liked_users = list(map(int, liked_users.split(','))) if liked_users else []
    disliked_users = list(map(int, disliked_users.split(','))) if disliked_users else []
    matches = list(map(int, matches.split(','))) if matches else []

    return User(user_id, name, age, gender, location, interests, liked_users, disliked_users, matches)
  else:
    return None #User not found
```

```python
def fetch_user(user_id):
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()

    cursor.execute('SELECT * FROM users WHERE user_id = ?', (user_id,))
    user_data = cursor.fetchone()

    conn.close()

    if user_data:
        user_id, name, age, gender, location, interests, liked_users, disliked_users, matches = user_data

        # Convert comma-separated strings back to lists
        interests = interests.split(',')
        liked_users = list(map(int, liked_users.split(','))) if liked_users else []
        disliked_users = list(map(int, disliked_users.split(','))) if disliked_users else []
        matches = list(map(int, matches.split(','))) if matches else []

        return User(user_id, name, age, gender, location, interests, liked_users, disliked_users, matches)
    else:
        return None  # User not found
```

SELECT * FROM Users ORDERED BY email

SELECT * FROM Users ORDER BY name DESC