PYTHON PROGRAMMING

(18IS5DEPYP)

Unit – 5

(DATABASE PROGRAMMING, NETWORKING)

Mrs. Bhavani K

Assistant Professor

Dept. of ISE, DSCE

DSCE
Dept. of Information Science & Engg

Unit - 5

- **DATABASE PROGRAMMING**:
  - DBM Databases
  - SQL Databases

- **NETWORKING:**
  - Creating a TCP Client
  - Creating a TCP Server

DATABASE PROGRAMMING

DSCE
Dept. of Information Science & Engg

Database Programming

- Python provides an API (Application Programming Interface) for working with SQL databases and it is normally distributed with the SQLite 3 database as standard.
- Another kind of database is a *DBM (Database Manager) that stores any* number of key–value items.
- Python's standard library comes with interfaces to several DBMs, including some that are Unix-specific.
- If the available DBMs and the SQLite database are insufficient, the Python **Package Index, pypi.python.org/pypi**, has a large number of database-related packages, including the bsddb DBM ("Berkeley DB"), and interfaces to popular client/server databases such as DB2, Informix, Ingres, MySQL,ODBC, and PostgreSQL.

DSCE
Dept. of Information Science & Engg

Database Programming

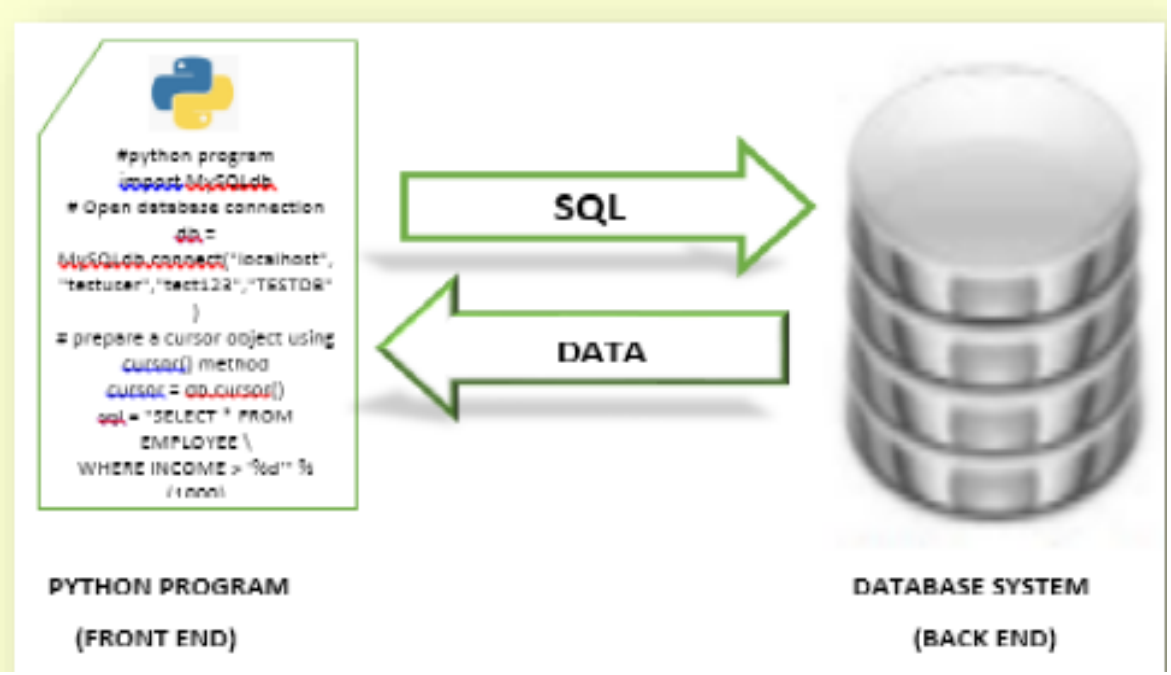
- There is another way to interact with SQL databases—use an ORM (Object Relational Mapper).

- Two of the most popular ORMs for Python are available as third-party libraries :
  - SQLAlchemy (www.sqlalchemy.org) and
  - SQLObject (www.sqlobject.org)

- One particularly nice feature of using an ORM is that it allows us to use Python syntax—creating objects and calling methods—rather than using raw SQL.

Interface python with SQL Database

- Generalized form of Interface of python with SQL Database can be understood with the help of this diagram.



Interface python with SQL Database contd..

- Form/any user interface designed in any programming language is FrontEnd whereas data given by database as response is known as Back-End database.
- SQL is just a query language, it is not a database. To perform SQL queries,we need to install any dbms
  - for example Oracle, MySQL, PostGresSQL, SQLServer, DB2etc.
- Using SQL in any of the dbms :
  - databases and tables can be created
  - data can be accessed, updated and maintained.
- The Python standard for database interfaces is the PythonDB-API.
- Python Database API supports a wide range of database servers, like mysql, postgressql, Informix, oracle, Sybase etc.

DSCE
Dept. of Information Science & Engg

DBM Databases

- Python provides a database API that is very useful when needed to work with different type of databases.
  - The data are stored within a DBM (database manager) persistent dictionaries that work like normal Python dictionaries except that the data is written to and read from disk.
- Amongst the different DBM, the **anydbm** module offers an alternative to choose the best DBM module available.
  - If there is a need of a very specific feature of another DBM module, use the **anydbm** module.
- The DBM modules work when the data needs can be stored as key/value pairs.
- Use such DBM persistent dictionary when :
  - data needs are simple
  - small amount of data

DSCE
Dept. of Information Science & Engg

DBM Databases

- DBM is used by UNIX (and UNIX like) operating system.
- The dbm library is a simple database engine written by Ken Thompson.
- These databases use binary encoded string objects .
- The database stores data by use of a single key in fixed-size buckets and uses hashing techniques to enable fast retrieval of the data by key.
- The dbm package contains following modules:
  - **dbm.gnu** module : an interface to the DBM library version as implemented by the GNU project.
  - **dbm.ndbm** module : provides an interface to UNIX nbdm implementation.
  - **dbm.dumb** : used as a fallback option in the event, other dbm implementations are not found. This requires no external dependencies but is slower than others.

  These modules are internally used by Python's shelve module.

open()

- The open() function allows following flags:

Value	Meaning
'r'	Open an existing database for reading only (default)
'w'	Open an existing database for reading and writing
'c'	Open database for reading and writing, creating it if it doesn't exist
'n'	Always create a new, empty database, open for reading and writing

DSCE
Dept. of Information Science & Engg

Quickstart with DBM

- The following example illustrates the use of a DBM module that is used to store data into a DB using dictionary-like syntax:

```
>>> import anydbm
>>> # open a DB. The c option opens in read/write mode and creates the file if needed.
>>> db = anydbm.open('websites', 'c')
>>> # add an item
>>> db["item1"] = "First example"
>>> print(db['item1'] "First example" )
>>> # close and save
>>> db.close()
```

DSCE
Dept. of Information Science & Engg

DBM Example

- The dbm object's whichdb() function tells which implementation of dbm is available on current Python installation.

```
>>> dbm.whichdb('mydbm.db')
 'dbm.dumb'
>>> import dbm
>>> db=dbm.open('mydbm.db','n')
>>> db['name']= ‘Raj Dev'
>>> db['address']=‘KS Layout Bangalore'
>>> db['PIN']=‘560078'
>>> db.close()
```

DSCE
Dept. of Information Science & Engg

shelve module

- The shelve module in Python's standard library is a simple yet effective tool for persistent data storage when using a relational database solution is not required.
- The shelve module provides a wrapper around a DBM that allows us to interact with the DBM as though it were a dictionary.
- The shelve module converts the keys and values to and from bytes objects.
- The shelf object defined in this module is dictionary-like object which is persistently stored in a disk file. This creates a file similar to dbm database on UNIX like systems.
- Only string data type can be used as key in this special dictionary object, whereas any picklable object can serve as value.

shelve module contd..

- The shelve module defines three classes as follows –

Sl.No.	Module & Description
1	**Shelf** This is the base class for shelf implementations. It is initialized with dict-like object.
2	**BsdDbShelf** This is a subclass of Shelf class. The dict object passed to its constructor must support first(), next(), previous(), last() and set_location() methods.
3	**DbfilenameShelf** This is also a subclass of Shelf but accepts a filename as parameter to its constructor rather than dict object.

shelve module contd..

- Easiest way to form a Shelf object is to use open() function defined in shelve module which return a DbfilenameShelf object:

  open(filename, flag = 'c', protocol=None, writeback = False)

- The filename parameter is assigned to the database created.
- Default value for flag parameter is ‘c’ for read/write access. Other flags are ‘w’ (write only) ‘r’ (read only) and ‘n’ (new with read/write)
- Protocol parameter denotes pickle protocol writeback parameter by default is false. If set to true, the accessed entries are cached.
- Every access calls sync() and close() operations hence process may be slow.

shelve methods contd..

- The Shelf object has following methods available –

Sl.No.	Method & Description
1	**close() :** synchronise and close persistent dict object.
2	**sync() :** Write back all entries in the cache if shelf was opened with writeback set to True.
3	**get() :** returns value associated with key
4	**items() :** list of tuples – each tuple is key value pair
5	**keys() :** list of shelf keys
6	**pop() :** remove specified key and return the corresponding value.
7	**update() :** Update shelf from another dict/iterable
8	**values() :** list of shelf values

shelve Example1

- Following code creates a database and stores dictionary entries in it:

```
import shelve
s = shelve.open("test")
s['name'] = "Ajay"
s['age'] = 23
s['marks'] = 75
s.close()
```

- This will create test.dir file in current directory and store key-value data in hashed form.

shelve Example1 contd..

```
#To access value of a particular key in shelf.

s=shelve.open('test')
print("Age = ", s['age'])

s['age']=25
s.get('age')
print(" Modified Age = ", s['age'])

items = list(s.items())
print("Items = ", items)

print("Keys= " , list(s.keys()))
print("Values = " , list(s.values()))
```

Output:

```
Age =  23
Modified Age =  25
Items =  [('name', 'Ajay'), ('age', 25), ('marks', 75)]
Keys=  ['name', 'age', 'marks']
Values =  ['Ajay', 25, 75]
```

shelve Example1 contd..

```
#To remove a key-value pair from shelf

print("Marks = ", s.pop('marks'))
items = print("Items = ", list(s.items()))
```

Output:

```
Marks =  75
Items =  [('name', 'Ajay'), ('age', 25)]
```

shelve Example1 contd..

- To merge items of another dictionary with shelf use update() method:

```
# merge items of another dictionary
d={'salary':10000, 'designation':'manager'}
s.update(d)
items1 = list(s.items())
print("Updated items = ", items1)
```

Output:
Updated items = [('name', 'Ajay'), ('age', 25), ('salary', 10000), ('designation', 'manager')]

DSCE
Dept. of Information Science & Engg

shelve Example2

```
# shelve_create.py

import shelve

s = shelve.open('test_shelf.db')

try:
    s['key1'] = { 'int': 10, 'float':9.5,
      'string':'Sample data' }
finally:
    s.close()
```

```
# shelve_access.py

import shelve

s = shelve.open('test_shelf.db')

try:
    existing = s['key1']
    print('Existing db items = ',existing)
finally:
    s.close()
```

Output:
Existing db items = {'int': 10, 'float': 9.5, 'string': 'Sample data'}

DSCE
Dept. of Information Science & Engg

shelve Example3

```
# shelve_create.py

import shelve

s = shelve.open('test_shelf.db')

try:
    s['key1'] = { 'int': 10, 'float':9.5,
      'string':'Sample data' }
finally:
    s.close()
```

```
# shelve_withoutwriteback.py

import shelve

s = shelve.open('test_shelf.db')
try:
    print(s['key1'])
    s['key1']['new_value'] = 'this was not here before'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print(s['key1'])
finally:
    s.close()
```

Output:

```
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

shelve Example3 contd…

```
# shelve_create.py

import shelve

s = shelve.open('test_shelf.db')

try:
    s['key1'] = { 'int': 10, 'float':9.5,
      'string':'Sample data' }
finally:
    s.close()
```

```
# shelve_withwriteback.py

import shelve

s = shelve.open('test_shelf.db', writeback=True)
try:
    print(s['key1'])
    s['key1']['new_value'] = 'this was not here before'
    print(s['key1'])
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print(s['key1'])
finally:
    s.close()
```

Output:

```
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data', 'new_value': 'this was not here before'}
{'int': 10, 'float': 9.5, 'string': 'Sample data', 'new_value': 'this was not here before'}
```

DSCE
Dept. of Information Science & Engg

Python - Object Serialization

- Object serialization is the process of converting state of an object into byte stream.
  - This byte stream can further be stored in any file-like object such as a disk file or memory stream.
  - It can also be transmitted via sockets etc.
- Deserialization is the process of reconstructing the object from the byte stream.
- Python refers to serialization and deserialization by terms pickling and unpickling respectively.
- The 'pickle' module bundled with Python's standard library defines functions for serialization (dump() and dumps()) and deserialization (load() and loads()).

DSCE
Dept. of Information Science & Engg

Pickle Example

- The dump() and load() functions of pickle module respectively perform pickling and unpickling of Python data. The dump() function writes pickled object to a file (or file like object) and load() function unpickles data from file back to Python object.

```
import pickle
f=open("pickled.txt","wb")     # write - binary
dct={"name":"Rajeev", "age":23, "Gender":"Male", "marks":75}
pickle.dump(dct,f)
f.close()
```

When above code is executed, the dictionary object's byte representation will be stored in 'pickled.txt' file. The file must have 'write and binary' mode enabled.

```
import pickle
# read binary
f=open("pickled.txt","rb")
d=pickle.load(f)
print (d)
f.close()
```

Output:
{'name': 'Rajeev', 'age': 23, 'Gender': 'Male', 'marks': 75}

On the other hand load() function unpickles or deserializes data from binary file back to Python dictionary.

DSCE
Dept. of Information Science & Engg

Pickle Example

- The pickle module also consists of dumps() function that pickles Python data to a string representation. Use loads() function to unpickle the string and obtain original dictionary object.

```
from pickle import dumps
dct={"name":"Rajneesh", "age":23, "Gender":"Male","marks":75}
dctstring=dumps(dct)
print(dctstring)
```

```
from pickle import loads
dct=loads(dctstring)
print(dct)
```

Output:

```
b'\x80\x03}q\x00(X\x04\x00\x00\x00nameq\x
01X\x08\x00\x00\x00Rajneeshq\x02X\x03\x00
\x00\x00ageq\x03K\x17X\x06\x00\x00\x00Ge
nderq\x04X\x04\x00\x00\x00Maleq\x05X\x05
\x00\x00\x00marksq\x06KKu.'
```

Output:

```
{'name': 'Rajneesh', 'age': 23,
'Gender': 'Male', 'marks': 75}
```

Difference between pickle and shelve

- Pickle is a native Python object serialization format.
  - The pickle interface provides four methods: dump, dumps, load, and loads.
    - The dump() method serializes to an open file (file-like object).
    - The dumps() method serializes to a string.
    - The load() method deserializes from an open file-like object.
    - The loads() method deserializes from a string.
- pickle is for serializing some object (or objects) as a single bytestream.
- Shelve builds on top of pickle and implements a serialization dictionary where objects are pickled, but associated with a key (some string), so you can load your shelved data file and access your pickled objects via keys.
  - This could be more convenient were you to be serializing many objects.

pickle Example

```
import pickle

integers = [1, 2, 3, 4, 5]

with open('pickle-example.p', 'wb') as pfile:
    pickle.dump(integers, pfile)

with open('pickle-example.p', 'rb') as pfile:
    integers = pickle.load(pfile)
    print(integers)
```

Output:
[1, 2, 3, 4, 5]

shelve Example

```
import shelve

integers = [1, 2, 3, 4, 5]

with shelve.open('shelf-example', 'c') as shelf:
    shelf['ints'] = integers

with shelve.open('shelf-example', 'r') as shelf:
    for key in shelf.keys():
        print(repr(key), repr(shelf[key]))
```

Output:
'ints' [1, 2, 3, 4, 5]

DSCE
Dept. of Information Science & Engg

str() vs repr() in Python

- repr() returns a string that holds a printable representation of an object.
- The __str__() and __repr__() methods both give us strings in return. So what sets them apart?
  - The goal of __repr__ is to be unambiguous and that of __str__ is to be readable.
  - __repr__ is kind of official and __str__ is somewhat informal.
- Example :

```
>>> s='Hello'
>>> print(str(s))
Hello
>>> print(repr(s))
'Hello'
```

- The *print* statement and *str()* function make a call to __str__, but repr() makes on to __repr__.
- Any string at the **interpreter** prompt makes a call to __str__(), but an object at the prompt makes a call to __repr__().

str() vs repr() in Python contd..

```
>>> import datetime

>>> t=datetime.datetime.now()

>>> str(t) #Readable
'2018-09-07 17:33:24.261778'

>>> repr(t)
'datetime.datetime(2018, 9, 7, 17, 33, 24, 261778)'

>>> t
datetime.datetime(2018, 9, 7, 17, 33, 24, 261778)
```

SQL Databases

- Python DB-API
  - The DB-API is a specification for a common interface to relational databases.
  - Fairly uniform access to (mainly) SQL databases.
  - Balance the effort needed to develop applications and the effort needed to maintain the database interfaces.
  - Implemented for most available SQL database systems.

Some DB-API implementations

- MySQL
  - MySQLdb
- PostgreSQL
  - psycopg
  - PyGresQL
  - pyPgSQL
- Oracle
  - dc_oracle2
  - cx_oracle
- Interbase/Firebird
  - Kinterbasdb
- SAP DB / MaxSQL
  - sapdbapi
- DB2
  - pydb2
- ODBC
  - mxODBC
  - adodbapi

Introduction to Python's DB-API

- Connection objects
- Cursor objects
- Result sets
- Standard exceptions
- Other module contents

Connection object

- Represents a database session
- Responsible for transactions
- Each cursor belongs to a connection
- Constructor:

  cnx = dbi.connect('mydb', user='mly', password='secret')

  The connect parameters vary between databases!

- Methods:

  cur = cnx.cursor()

  cnx.commit() cnx.rollback() cnx.close()

DSCE
Dept. of Information Science & Engg

DB-API 2.0 Connection Object Methods

Syntax	Description
`db.close()`	Closes the connection to the database (represented by the `db` object which is obtained by calling a `connect()` function)
`db.commit()`	Commits any pending transaction to the database; does nothing for databases that don't support transactions
`db.cursor()`	Returns a database cursor object through which queries can be executed
`db.rollback()`	Rolls back any pending transaction to the state that existed before the transaction began; does nothing for databases that don't support transactions

DB-API 2.0 Cursor Object Attributes and Methods

Syntax	Description
`c.arraysize`	The (readable/writable) number of rows that `fetchmany()` will return if no size is specified
`c.close()`	Closes the cursor, `c`; this is done automatically when the cursor goes out of scope
`c.description`	A read-only sequence of 7-tuples (`name`, `type_code`, `display_size`, `internal_size`, `precision`, `scale`, `null_ok`), describing each successive column of cursor `c`
`c.execute(sql, params)`	Executes the SQL query in string `sql`, replacing each placeholder with the corresponding parameter from the *params* sequence or mapping if given
`c.executemany(sql, seq_of_params)`	Executes the SQL query once for each item in the *seq_of_params* sequence of sequences or mappings; this method should not be used for operations that create result sets (such as `SELECT` statements)
`c.fetchall()`	Returns a sequence of all the rows that have not yet been fetched (which could be all of them)
`c.fetchmany(size)`	Returns a sequence of rows (each row itself being a sequence); *size* defaults to `c.arraysize`
`c.fetchone()`	Returns the next row of the query result set as a sequence, or `None` when the results are exhausted. Raises an exception if there is no result set.
`c.rowcount`	The read-only row count for the last operation (e.g., `SELECT`, `INSERT`, `UPDATE`, or `DELETE`) or -1 if not available or not applicable

Connection Object Methods

- The sqlite3.connect() function returns a database object, having opened the database file it is given.
- All queries are executed through a database cursor, available from the database object's cursor() method.
- SQLite supports a limited range of data types :
  - Booleans, numbers, and strings
    - can be extended using data "adaptors", either the predefined ones such as those for dates and datetimes, or custom ones that we can use to represent any data types we like.

Cursor object

- cursor.execute('SELECT * FROM aTable')
- cursor.execute('''SELECT * FROM aTable
  WHERE name = ? AND age = ?''', ('Bill', 23))
  - The syntax varies…
  - cursor.execute('''SELECT * FROM aTable
    WHERE name = %s AND age = %s''', ('Bill', 23))
- cursor.executemany('''SELECT * FROM aTable
  WHERE name = ? AND age = ?''',
  [('Bill', 23), ('Mary', 12), ('Anne', 87)])
- cursor.close() - Do this before commit!

Cursor object

- a_row = cursor.fetchone()
- a_sequence_of_rows = cursor.fetchall()
- a_sequence_of_rows = cursor.fetchmany(100)
- cursor.description
  - Read-only attribute with a sequence of tuples, one per column in the result set. Each tuple contains the following info about the column: name, type code, display size, internal size, precision, scale, nullable.
  - Could also be None... (I.e. after an insert.)

Result set

- One or more rows from a relation.
- fetchone() returns a sequence (often a tuple).
- fetchall() and fetchmany() returns a sequence of sequences, for instance a list of tuples.
- Don't assume more about types than you need!
  – It might not be a list of tuples, but it will be a sequence of sequences.
- Some dbi's have more elaborate row objects
  – There's always addons if yours hasn't...

DB-API Exceptions

- Warning
- Error
  - InterfaceError
  - DatabaseError
    - IntegrityError
    - InternalError
    - NotSupportedError
    - OperationalError
    - ProgrammingError

DB-API module attributes

- dbi.paramstyle
  - format – %s
  - named – :col_name
  - numeric – :1
  - pyformat – %(col_name)s
  - qmark – ?
- dbi.threadsafety – 0, 1...
- dbi.apilevel – e.g. '2.0'

DB-API variations

- Different module names (obviously)
  - import xxx as dbi
- Different parameters for connect()
  - conn = adodbapi.connect('dsn=db1;uid=tom;pwd=x')
  - conn = sqlite.connect('c:/dbs/accounting.db')
- Different parameter styles (?, %s, :1 etc)
- Differences in SQL dialects
- Optional features of the DB-API

db_row

- A wrapper for DB-API result sets (or other sequences of tuples).
- Exposes the “normal” sequence-like access
  - row[0], row[1]
- Provides dictionary-like access to columns
  - row['name'], row['email'], row.keys(), row.values()
- Provides attribute access to columns
  - row.?.name, row.?.email

Placeholders

- In the query, question marks can be used as placeholders.
  - Each ? is replaced by the corresponding value in the sequence that follows the string containing the SQL statement.
  - Named placeholders can also be used .
  - It is recommended for always using placeholders and leaving the burden of correctly encoding and escaping the data items to the database module.
  - Another benefit of using placeholders is that they improve security
    - since they prevent arbitrary SQL from being maliciously injected into a query.

SQLObject

- Object-oriented wrapper over DB-API.
- Insulates the programmer from SQL.
- Not so easy to use with legacy databases.

SQLite in Python

- SQLite is a self-contained, file-based SQL database.
- SQLite comes bundled with Python and can be used in any Python applications without having to install any additional software.
- SQLite databases are fully featured SQL engines that can be used for many purposes.
- The Python Standard Library includes a module called "sqlite3" intended for working with this database.
  - Python sqlite3 is an excellent module with which all possible DB operations can be performed with in-memory and persistent database in applications.

SQLite Features

- auto-commit mode (and other kinds of transaction control)
- the ability to create functions that can be executed inside SQL queries.
- It is also possible to provide a factory function to control what is returned for each fetched record (e.g., a dictionary or custom type instead of a sequence of fields).
- it is possible to create in-memory SQLite databases by passing ":memory:" as the filename.
  - For example, sqlite3.connect(":memory:").
  - A ":memory:" SQLite database will disappear as soon as your Python program exits.
  - This might be convenient to have a temporary sandbox to try something out in SQLite, and don't need to persist any data after program exits.

SQLite database connection

- To query data in an SQLite database from Python, you use these steps:
  - First, establish a connection to the SQLite database by creating a Connection object.
  - Next, create a Cursor object using the cursor method of the Connection object.
  - Then, execute a CREATE/INSERT/SELECT statement.
  - After that, call the fetchall() method of the cursor object to fetch the data.
  - Finally, loop the cursor and process each row individually.

Example

- Consider an inventory of fish that we need to modify as fish are added to or removed from a fictional aquarium.

Example contd..

- **Step 1 — Creating a Connection to a SQLite Database**
  - Connect to a SQLite database by accessing data that ultimately resides in a file on computer.
  - connect to a SQLite database using the Python sqlite3 module:

```
import sqlite3
connection = sqlite3.connect("aquarium.db")
```

    - » import sqlite3 gives Python program access to the sqlite3 module.
    - » The sqlite3.connect() function returns a Connection object that is used to interact with the SQLite database held in the file aquarium.db.
    - » The aquarium.db file is created automatically by sqlite3.connect() if aquarium.db does not already exist on computer.

Example contd..

- Verify created connection object by running:

```
print(connection.total_changes)          // Output:0
```

  - connection.total_changes is the total number of database rows that have been changed by connection.

    » Since we have not executed any SQL commands yet, 0 total_changes is correct.

Example contd..

- **Step 2 — Adding Data to the SQLite Database**

  – After getting connected to the aquarium.db SQLite database, start inserting and reading data from it.

  – In a SQL database, data is stored in tables. Tables define a set of columns, and contain 0 or more rows with data for each of the defined columns.

  – Create a table named fish that tracks the following data:

name	species	tank_number
Sammy	shark	1
Jamie	cuttlefish	7

The fish table will track a value for name, species, and tank_number for each fish at the aquarium. Two example fish rows are listed: one row for a shark named Sammy, and one row for a cuttlefish named Jamie.

Example contd..

- Create this fish table in SQLite using the connection we made in Step 1:

```
# Adding Data to the SQLite Database - first creating table
cursor = connection.cursor()
#cursor.execute("CREATE TABLE fish (name TEXT, species TEXT,
                                    tank_number INTEGER)")
```

connection.cursor() returns a Cursor object.

Cursor objects allow us to send SQL statements to a SQLite database using cursor.execute().

The "CREATE TABLE fish ..." string is a SQL statement that creates a table named fish with the three columns described earlier: name of type TEXT, species of type TEXT, and tank_number of type INTEGER.

Example contd..

- Insert rows of data into the table:

```
# inserting data
cursor.execute("INSERT INTO fish VALUES ('Sammy', 'shark', 1)")
cursor.execute("INSERT INTO fish VALUES ('Jamie', 'cuttlefish', 7)")
print("Records inserted successfully");
```

cursor.execute() is called two times: once to insert a row for the shark Sammy in tank 1, and once to insert a row for the cuttlefish Jamie in tank 7.

"INSERT INTO fish VALUES ..." is a SQL statement that allows us to add rows to a table.

Example contd..

- Step 3 — Reading Data from the SQLite Database

```
# Reading Data from the SQLite Database
rows = cursor.execute("SELECT name, species, tank_number FROM fish").fetchall()
print(rows)

Output:
[('Sammy', 'shark', 1), ('Jamie', 'cuttlefish', 7)]
```

- The cursor.execute() function runs a SELECT statement to retrieve values for the name, species, and tank_number columns in the fish table; fetchall() retrieves all the results of the SELECT statement.

- print(rows) displays a list of two tuples.

Example contd..

- To retrieve rows in the fish table that match a specific set of criteria, use a WHERE clause:

```
target_fish_name = "Jamie"
rows = cursor.execute(
   "SELECT name, species, tank_number FROM fish WHERE name = ?",
   (target_fish_name,),
).fetchall()
print(rows)

Output:
[('Jamie', 'cuttlefish', 7)]
```

cursor.execute(<SQL statement>).fetchall() fetches all the results of a SELECT statement.

The WHERE clause in the SELECT statement filters for rows where the value of name is target_fish_name.

? Is used to substitute target_fish_name variable into the SELECT statement.

Example contd..

- Step 4 — Modifying Data in the SQLite Database

  – Rows in a SQLite database can be modified using UPDATE and DELETE SQL statements.

```
# Updating Data in the SQLite Database
new_tank_number = 2
moved_fish_name = "Sammy"
cursor.execute(
   "UPDATE fish SET tank_number = ? WHERE name = ?",
   (new_tank_number, moved_fish_name) )

rows = cursor.execute("SELECT name, species, tank_number FROM fish").fetchall()
print(rows)

Output:
[('Sammy', 'shark', 2), ('Jamie', 'cuttlefish', 7)]
```

Issue an UPDATE SQL statement to change the tank_number of Sammy to its new value of 2.

The WHERE clause in the UPDATE statement ensures we only change the value of tank_number if a row has name = "Sammy".

Example contd..

- Issue a DELETE SQL statement to remove a row:

```
#deleting data
released_fish_name = "Sammy"
cursor.execute(
   "DELETE FROM fish WHERE name = ?",
   (released_fish_name,)
)

rows = cursor.execute("SELECT name, species, tank_number FROM fish").fetchall()
print(rows)
```

Output:

```
[('Jamie', 'cuttlefish', 7)]
```

DELETE SQL statement removes the row for Sammy the shark.

The WHERE clause in the DELETE statement ensures to delete a row if that row has name = "Sammy".

Example contd..

- Step 5 — Using with Statements For Automatic Cleanup

```
# closing connection
from contextlib import closing

with closing(sqlite3.connect("aquarium.db")) as connection:
   with closing(connection.cursor()) as cursor:
      rows = cursor.execute("SELECT 1").fetchall()
      print(rows)
```

Output:
[(1,)]

closing is a function provided by the contextlib module.

When a with statement exits, closing ensures that close() is called on whatever object is passed to it.

The closing function is used twice in this example.
to ensure that the Connection object returned by sqlite3.connect() is automatically closed, and
to ensure that the Cursor object returned by connection.cursor() is automatically closed.

Since "SELECT 1" is a SQL statement that always returns a single row with a single column with a value of 1, it makes sense to see a single tuple with 1 as its only value returned by our code.

NETWORKING

NETWORKING

- Networking allows computer programs to communicate with each other, even if they are running on different machines.
  - For programs such as web browsers, this is the essence of what they do
  - for others networking adds additional dimensions to their functionality
    - for example, remote operation or logging, or the ability to retrieve or supply data to other machines.

- Client/server applications are normally implemented as two separate programs:
  - a server that waits for and responds to requests, and
  - one or more clients that send requests to the server and read back the server's response.
  - For this to work, the clients must know where to connect to the server, that is, the server's IP (Internet Protocol) address and port number.
  - Also, both clients and server must send and receive data using an agreed-upon protocol using data formats that they both understand.

NETWORKING contd..

- Python's low-level socket module (on which all of Python's higher-level networking modules are based) supports both IPv4 and IPv6 addresses.
  - It also supports the most commonly used networking protocols, including :
    - UDP (User Datagram Protocol),a lightweight but unreliable connectionless protocol where data is sent as discrete packets (datagrams) but with no guarantee that they will arrive and
    - TCP (Transmission Control Protocol), a reliable connection and stream-oriented protocol.
      - With TCP, any amount of data can be sent and received—the socket is responsible for breaking the data into chunks that are small enough to send, and for reconstructing the data at the other end.

NETWORKING contd..

- UDP is often used
  - to monitor instruments that give continuous readings, and where the odd missed reading is not significant and
  - for audio or video streaming in cases where the occasional missed frame is acceptable.

- Both the FTP and the HTTP protocols are built on top of TCP.

- Client/server applications normally use TCP because they need connection-oriented communication and the reliability that TCP provides.
  - decision that must be made here is whether to send and receive data as lines of text or as blocks of binary data, and if the latter, in what form

NETWORKING contd..

- The advantage of using blocks of binary data as a binary pickle is that the same sending and receiving code can be used for *any application* since almost any arbitrary data can be used in a pickle.

- The disadvantage is that both client and server must understand pickles, so they must be written in Python or must be able to access Python:

  – Example: using Jython in Java or Boost.Python in C++.

- The usual security considerations apply to the use of pickles.

DSCE
Dept. of Information Science & Engg

Socket programming

- Sockets are interior endpoints built for sending and receiving data.
- A single network will have two sockets, one for each communicating device or program.
- The sockets are a combination of an IP address and a Port.
  - A single device can have ‘n’ number of sockets based on the port number that is being used.
  - Different ports are available for different types of protocols.
- Sockets (aka socket programming) enable programs to send and receive data, bi-directionally, at any given moment.
- It works by connecting two sockets (or nodes) together and allowing them to communicate in real time, and is a great option for building numerous apps.
- Some examples of apps that use socket programming are:
  - Web pages that show live notifications (Facebook, eBay)
  - Multiplayer online games (League of Legends, Counter Strike)
  - Chat apps (WhatsApp, WeChat, Slack)
  - Realtime data dashboards (Robinhood, Coinbase)
  - IoT devices (Nest, August Locks)

Socket programming contd..

- Sockets are the endpoints of a bidirectional communications channel.
- Sockets may communicate within a process, between processes on the same machine, or between processes on different machines.
- To achieve Socket Programming in Python, import the **socket** module or framework.
  - This module consists of built-in methods that are required for creating sockets and help them associate with each other.
- To create a socket, use the *socket.socket()* function available in *socket* module, which has the general syntax :

  s = socket.socket (socket_family, socket_type, protocol=0)

  Here is the description of the parameters :

  socket_family − These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.

  socket_type − The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.

  protocol − Typically zero, this may be used to identify a variant of a protocol within a domain and type.

DSCE
Dept. of Information Science & Engg

Server Socket Methods

Sr.No.	Method & Description
1	**s.bind() :** This method binds address (hostname, port number pair) to socket.
2	**s.listen()** This method sets up and start TCP listener.
3	**s.accept()** This passively accept TCP client connection, waiting until connection arrives (blocking).

DSCE
Dept. of Information Science & Engg

Client Socket Methods

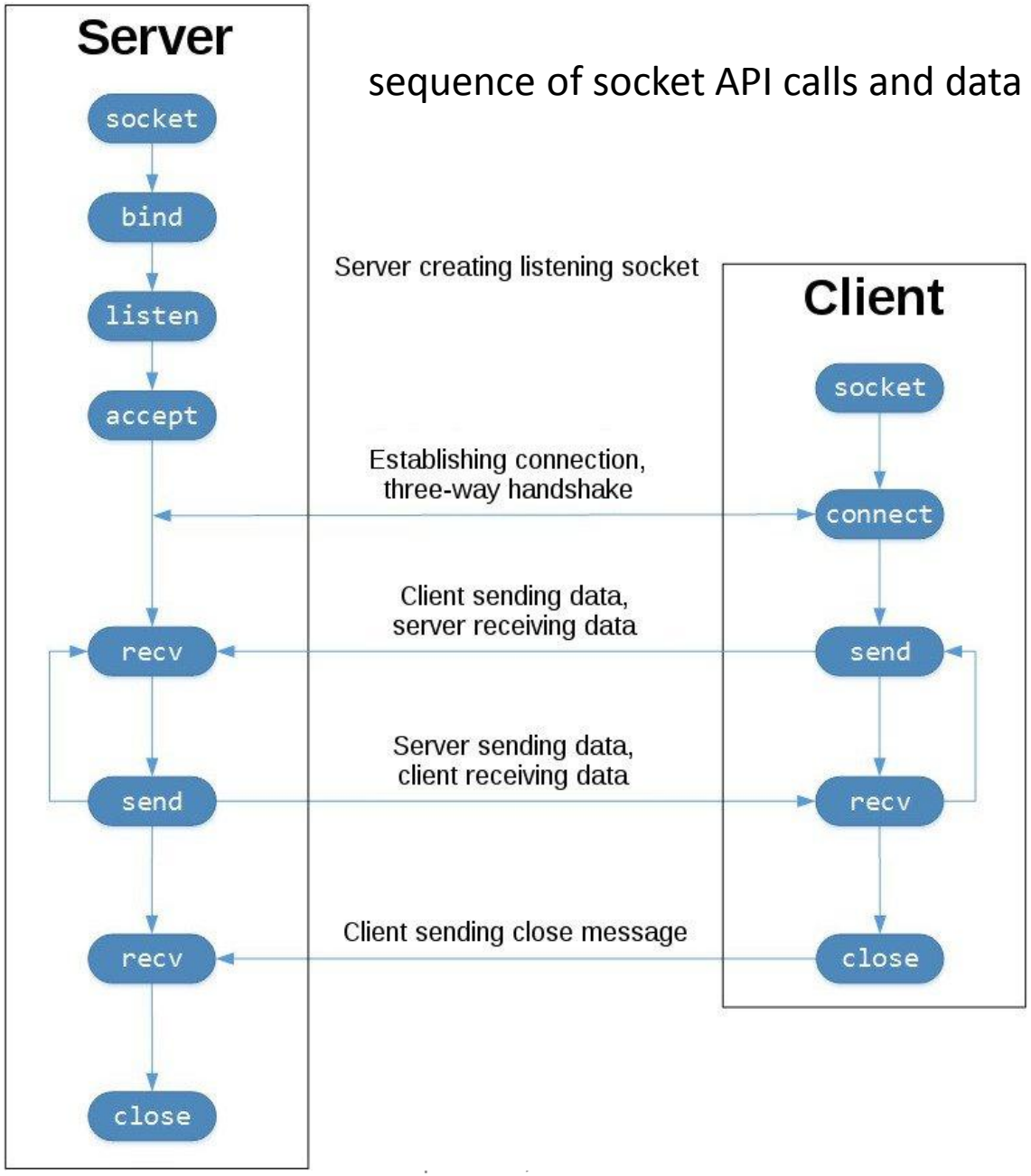
Sr.No.	Method & Description
1	**s.connect()** This method actively initiates TCP server connection.

DSCE
Dept. of Information Science & Engg

General Socket Methods

Sr.No.	Method & Description
1	**s.recv()** This method receives TCP message
2	**s.send()** This method transmits TCP message
3	**s.recvfrom()** This method receives UDP message
4	**s.sendto()** This method transmits UDP message
5	**s.close()** This method closes socket
6	**socket.gethostname()** Returns the hostname.

sequence of socket API calls and data flow for TCP

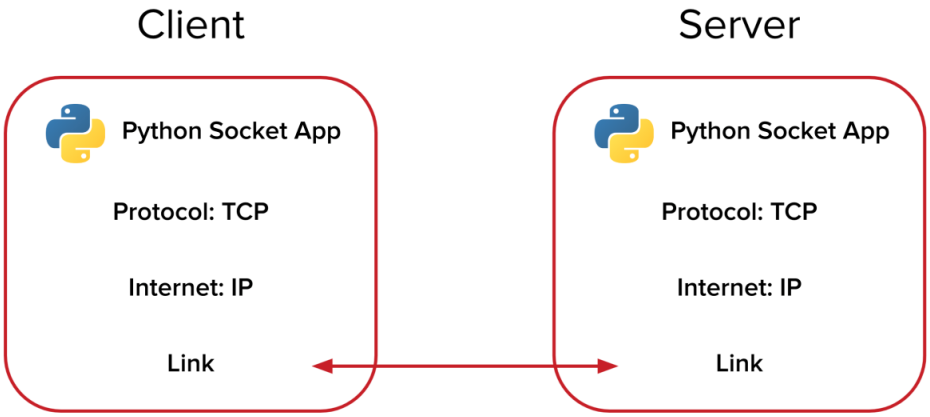


Example

- Sockets can be configured to act as a *server* and listen for incoming messages, or connect to other applications as a *client*.
  - After both ends of a TCP/IP socket are connected, communication is bi-directional.

- For the sake of the example, run the server and clients on the same machine;
  - this means use “localhost” as the IP address

- The port number should be greater than 1023 and is normally between 5001 and 32767, although port numbers up to 65535 are normally valid.



Creating a TCP Server

```
import socket
def server_program():
    # get the hostname
    host = socket.gethostname()
    port = 5000  # initiate port no above 1024

    server_socket = socket.socket()  # get instance
    # look closely. The bind() function takes tuple as argument
    server_socket.bind((host, port))  # bind host address and port together

    # configure how many client the server can listen simultaneously
    server_socket.listen(2)
    conn, address = server_socket.accept()  # accept new connection
    print("Connection from: " + str(address))
    while True:
        # receive data stream. it won't accept data packet greater than 1024 bytes
        data = conn.recv(1024).decode()
        if not data:
            # if data is not received break
            break
        print("from connected user: " + str(data))
        data = input(' -> ')
        conn.send(data.encode())  # send data to the client

    conn.close()  # close the connection
```

```
if __name__ == '__main__':
    server_program()
```

Creating a TCP Client

```
import socket

def client_program():
    host = socket.gethostname()  # as both code is running on same pc
    port = 5000  # socket server port number

    client_socket = socket.socket()  # instantiate
    client_socket.connect((host, port))  # connect to the server

    message = input(" -> ")  # take input

    while message.lower().strip() != 'bye':
        client_socket.send(message.encode())  # send message
        data = client_socket.recv(1024).decode()  # receive response

        print('Received from server: ' + data)  # show in terminal

        message = input(" -> ")  # again take input

    client_socket.close()  # close the connection
```

```
if __name__ == '__main__':
    client_program()
```

Client-server communication – python string

- The main difference between server and client program is, in server program, it needs to bind host address and port address together.

```
ISEDSCE$ python3.6 socket_server.py
Connection from: ('127.0.0.1', 57822)
from connected user: Hi
   -> Hello
from connected user: How are you?
   -> Good
from connected user: Awesome!
   -> Ok then, bye!
ISEDSCE$
```

Sever

```
ISEDSCE $ python3.6 socket_client.py
   -> Hi
 Received from server: Hello
   -> How are you?
Received from server: Good
   -> Awesome!
Received from server: Ok then, bye!
   -> Bye
ISEDSCE$
```

Client

Transferring Python Objects

- Till here you have just got the knack of transferring strings.
- But, Socket Programming in Python also allows you to transfer Python objects as well.
  - These objects can be anything like sets, tuples, dictionaries, etc.
  - To achieve this, import the pickle module of Python.
    - Python pickle module can be used for serializing or de-serializing objects in python.

DSCE
Dept. of Information Science & Engg

Python pickle module : EXAMPLE

```
import pickle

mylist=[1,2,'abc']
mymsg = pickle.dumps(mylist)
print(mymsg)
```

Output:

b'\x80\x03]q\x00(K\x01K\x02X\x03\x00\x00\x00abcq\x01e.'

- 'mylist' is serialized using the dumps() function of the pickle module.
- Also make a note that the output starts with a 'b', meaning it's converted to bytes.
- In socket programming, implement this module to transfer python objects between clients and servers.
  - When pickle is used along with sockets, absolutely anything can be transferred through the network.

Server-Side

```
import socket
import pickle


HEADERSIZE = 10

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), 1243))
s.listen(5)

while True:
    # now our endpoint knows about the OTHER endpoint.
    clientsocket, address = s.accept()
    print(f"Connection from {address} has been established.")

    d = {1:"Client", 2: "Server"}
    msg = pickle.dumps(d)
    msg = bytes(f"{len(msg):<{HEADERSIZE}}", 'utf-8')+msg
    print(msg)
    clientsocket.send(msg)

s.close()  # close the connection
```

Here, d is a dictionary that is basically a python object that needs to be sent from the server to the client. This is done by first serializing the object using dumps() and then converting it to bytes.

Client-Side

```
import socket
import pickle

HEADERSIZE = 10

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 1243))

while True:
    full_msg = b''
    new_msg = True
    while True:
        msg = s.recv(16)
        if new_msg:
            print("new msg len:",msg[:HEADERSIZE])
            msglen = int(msg[:HEADERSIZE])
            new_msg = False

        print(f"full message length: {msglen}")

        full_msg += msg

        print(len(full_msg))

        if len(full_msg)-HEADERSIZE == msglen:
            print("full msg recvd")
            print(full_msg[HEADERSIZE:])
            print(pickle.loads(full_msg[HEADERSIZE:]))
            new_msg = True
            full_msg = b""

s.close()  # close the connection
```

The first while loop will help us keep track of the complete message as well as the message that is being received using the buffer.

Then, if the message received is equal to the complete message, print the message as received complete info following which de-serialize the message using loads().

Client-server communication – python pickle

- The main difference between server and client program is, in server program, it needs to bind host address and port address together.

```
Connection from ('192.168.86.25', 50373) has been established.
b‘38    \x80\x03}q\x00(K\x01X\x02\x00\x00\x00hiq\x01K\x02X\x05\x00\x00\x00thereq\x02u.'
```

Sever

```
new msg len: b'38       '
full message length: 38
16
full message length: 38
32
full message length: 38
48
full msg recvd
b'\x80\x03}q\x00(K\x01X\x06\x00\x00\
x00Clientq\x01K\x02X\x06\x00\x00\x00
Serverq\x02u.'
{1: 'Client', 2: 'Server'}
```

Client

Thank you