



# Lecture 3 - Introducing Psycopg, and CRUD using Psycopg

Instructor: Suresh Melvin Sigera  
Email: [suresh.sigera@cuny.csi.edu](mailto:suresh.sigera@cuny.csi.edu)



# Agenda

- Transaction
  - ACID
  - Recovery
- Introducing Psycopg
  - Setting-up development environment
  - Installing Psycopg
- CRUD using Psycopg
  - Creating tables
  - Reading data
  - Updating data
  - Deleting data
- QA



# Transaction

A transaction is a unit of work which involves a set of database operations. So In transaction either all of the operations are executed, or none of the operations executed. I.e., We need to complete all the operation under a single transaction successfully to call it a **successful transaction**.



# What is a Transaction?

Any action that reads from and/or writes to a database may consist of

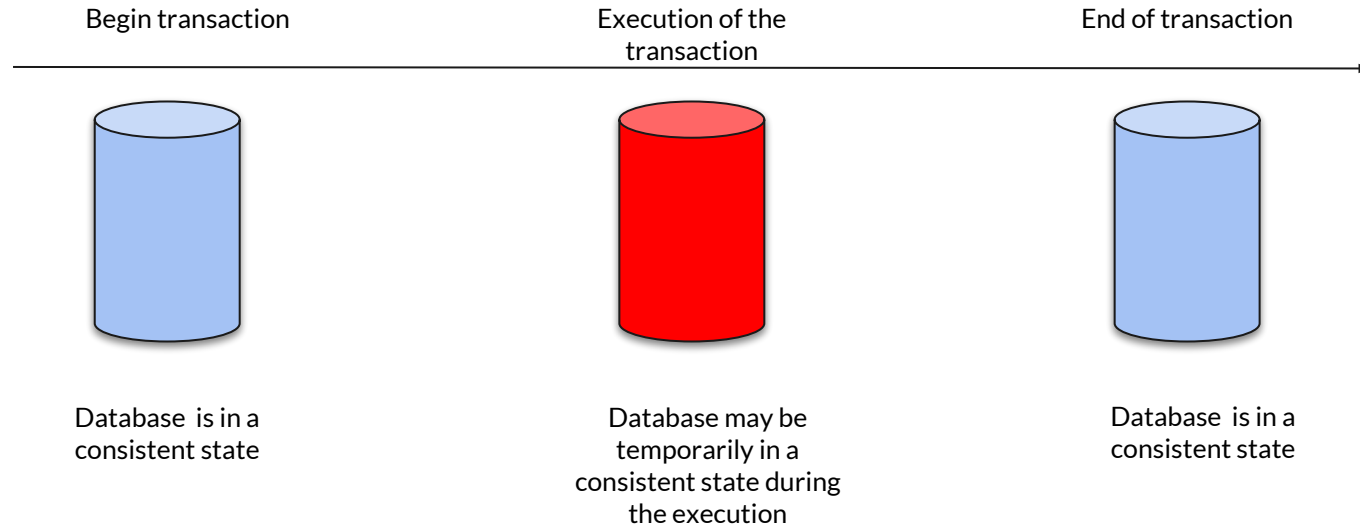
- Simple SELECT statement to generate a list of table contents
- A series of related UPDATE statements to change the values of attributes in various tables
- A series of INSERT statements to add rows to one or more tables
- A combination of SELECT, UPDATE, and INSERT statements



# What is a Transaction?

- A logical unit of work that must be either entirely completed or aborted
- Successful transaction changes the database from one **consistent** state to another
  - One in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests
  - The equivalent of a single SQL statement in an application program or transaction

# Transaction state

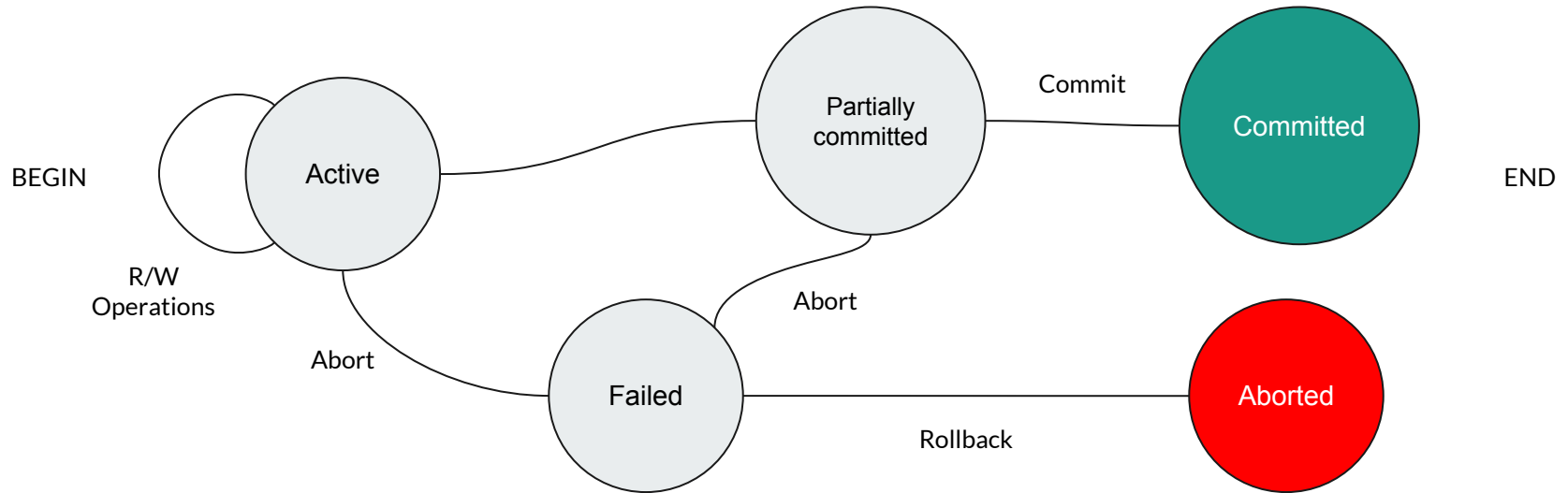




# Transaction state

- Active - the initial state; the transaction stays in this state while it is executing
- Partially committed - after the final statement has been executed
- Failed - after the discovery that normal execution can no longer proceed
- Aborted - after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction can be done only if no internal logical error
  - kill the transaction
- Committed – after successful completion

# Transaction state







# Evaluating transaction results

- Not all transactions update the database
- SQL code represents a transaction because database was accessed
- Improper or incomplete transactions can have a devastating effect on database integrity
- Some DBMSs provide means by which user can define enforceable constraints based on business rules
- Other integrity rules are enforced automatically by the DBMS when table structures are properly defined, thereby letting the DBMS validate some transactions



# Transaction processing

- For example, a transaction may involve
  - The creation of a new invoice
  - Insertion of an row in the LINE table
  - Decreasing the quantity on hand by 1
  - Updating the customer balance
  - Creating a new account transaction row
- If the system fails between the first and last step, the database will no longer be in a consistent state



# ACID aka Transaction properties

- **Atomicity:** either the entire set of operations happens or none of it does
- **Consistency:** the set of operations taken together should move the system for one consistent state to another consistent state.
- **Isolation:** each system perceives the system as if no other transactions were running concurrently (even though odds are there are other active transactions)
- **Durability:** results of a completed transaction must be permanent - even IF the system crashes



# Transaction ACID example

Think about the following using case: **Transfer \$50 from account A to account B**

1. read(A)
2.  $A := A - 50$
3. write(A)
4. read(B)
5.  $B := B + 50$
6. write(B)

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions



# Transaction ACID example

- Atomicity
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state. Failure could be due to software or hardware. The system should ensure that updates of a partially executed transaction are not reflected in the database
- Consistency
  - Money isn't lost or gained
- Isolation
  - If between steps 3 and 6 (of the fund transfer transaction), another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be). In other words, queries shouldn't see A or B change until completion
- Durability
  - Once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.



## R/W conflicts : database operations matrix

T1	T2	Result
Read	Read	No conflict
Read	Write	Conflict
Write	Read	Conflict
Write	Write	Conflict



# Transaction management with SQL

- ANSI has defined standards that govern SQL database transactions
- Transaction support is provided by two SQL statements: **COMMIT** and **ROLLBACK**
- ANSI standards require that, when a transaction sequence is initiated by a user or an application program, it must continue through all succeeding SQL statements until one of four events occurs



# Transaction management with SQL

- A **COMMIT** statement is reached- all changes are permanently recorded within the database
- A **ROLLBACK** is reached – all changes are aborted and the database is restored to a previous consistent state
- The end of the program is successfully reached – **equivalent to a COMMIT**
- The program abnormally terminates and a rollback occurs





# The Transaction log

Keeps track of all transactions that update the database. It contains:

- A record for the beginning of transaction
  - For each transaction component (SQL statement)
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - "Before" and "after" values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction
- Increases processing overhead but the ability to restore a corrupted database is worth the price



# The Transaction log

- Increases processing overhead but the ability to restore a corrupted database is worth the price
- If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to a previous state
- The log is itself a database and to maintain its integrity many DBMSs will implement it on several different disks to reduce the risk of system failure



# Concurrency control

- The coordination of the simultaneous execution of transactions in a multiprocessing database is known as concurrency control
- The objective of concurrency control is to ensure the serializability of transactions in a multiuser database environment



# Concurrency control

- Important simultaneous execution of transactions over a shared database can create several data integrity and consistency problems
- The three main problems are:
  - lost updates
  - uncommitted data
  - inconsistent retrievals



# Uncommitted data problem

- Uncommitted data occurs when two transactions execute concurrently and the first is rolled back after the second has already accessed the uncommitted data
  - This violates the isolation property of transactions (hint: ACID)



# Inconsistent retrieval problem

- Occur when a transaction calculates some aggregate functions over a set of data while transactions are updating the data
  - Some data may be read after they are changed and some before they are changed yielding inconsistent results



# Concurrency control with optimistic methods

- Phases are read, validation, and write
  - Read phase – transaction reads the database, executes the needed computations and makes the updates to a private copy of the database values.
    - All update operations of the transaction are recorded in a temporary update file which is not accessed by the remaining transactions
  - Validation phase – transaction is validated to ensure that the changes made will not affect the integrity and consistency of the database
    - If the validation test is positive, the transaction goes to the writing phase. If negative, the transaction is restarted and the changes discarded
  - Writing phase – the changes are permanently applied to the database



# Recovery

Transactions should be durable, but we cannot prevent all sorts of failures

- System crashes
- Power failures
- Disk crashes
- User mistakes
- Sabotage
- Natural disasters





# Recovery

Prevention is better than cure

- Reliable OS, and security
- UPS and surge protectors
- RAID arrays
- Cloud services and Cloud backups
- Can't protect against everything though



# Don't be too confident

- Crashes can occur during rollback or restart!
  - Algorithms must be idempotent
- Must be sure that log is stored separately from data (on different disk array; often replicated off-site!)
  - In case disk crash corrupts data, log allows fixing this
  - Also, since log is append-only, don't want have random access to data moving disk heads away



# psycopg2

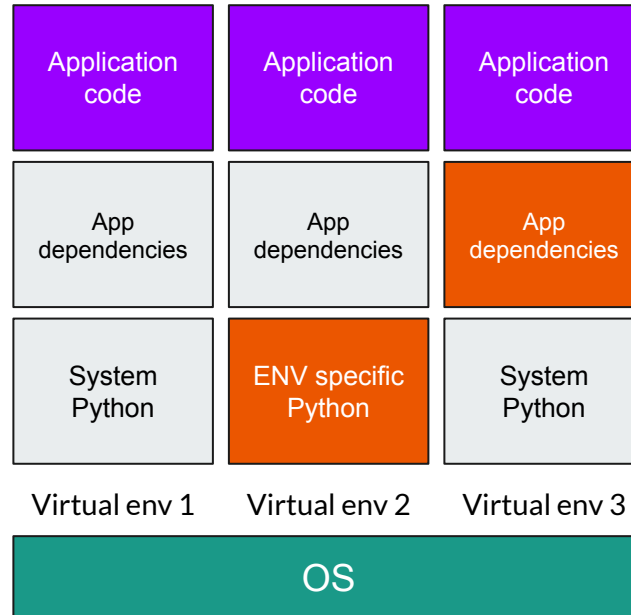
- psycopg is a PostgreSQL database adapter for the Python\_ programming language. We are using the version 2, a complete rewrite of the original code to provide new-style classes for connection and cursor objects and other sweet candies.
- You can visit the home page <http://initd.org/psycopg/>



# Setting up Anaconda distribution for Python

Create a new environment named py35, install Python 3.5	<code>conda create --name CSC424 python=3.5</code>
Activate the new environment to use it	WINDOWS: <code>activate CSC424</code> LINUX, macOS: <code>source activate CSC424</code>
Install psycpg2	<code>conda install -c anaconda psycpg2</code>
Install pipreqs	<code>pip install pipreqs</code>

# Setting up Anaconda distribution for Python





## psycopg2

- The factory function `connect ()` of module `Psycopg2` is used for creating a connection object
- The `connect ()` function accepts parameters corresponding to the PostgreSQL database name and the authentication details of the PostgreSQL User
- A cursor object is created using the obtained connection object
- Using a cursor object any SQL Statements supported by the PostgreSQL can be executed



## psycopg2

- Cursors in Psycopg are not thread safe. The thread safety while using the same cursor object across multiple Python threads need to be checked with the Psycopg documentation
- With a call to the `execute ()` method of the cursor, a SQL statement supported by the PostgreSQL can be executed
- Upon calling `execute ()` method, one among the data retrieval methods like `fetchone ()`, `fetchmany ()` and `fetchall ()` is called to get the actual row data for processing inside the Python program



## Low-level database access with psycopg2

psycopg2 is one of the most popular PostgreSQL drivers for Python. It is compliant to the Python's DB API 2.0 specification. It is mostly implemented in C and uses the library, libpq. It is thread safe, which means that you can share the same connection object between several threads. It can work both with Python 2 and Python 3.





# Connecting to a database

Connections to databases are handled by the objects of the class `connection`. These objects represent database sessions in the application. The objects are created using the function `connect()` from the module `psycopg2`. This way of creating connections is defined by the DB API 2.0.

To specify the location and authenticate in the database, a connection string can be used, like this:

- ```
conn = connect("host=db_host user=some_user dbname=database"
               "password=$secreT")
```
- ```
conn = connect(host="db_host", user="some_user", dbname="database",
               password="$secreT")
```




# Connection parameters

The basic connection parameters are:

- `dbname` – the database name (database is a deprecated alias) `user` – user name used to authenticate
- `password` – password used to authenticate
- `host` – database host address
- `port` – connection port number

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week3/psycopg2-part3.py>



## psycopg2 - connect

To specify the location and authenticate in the database, a connection string can be used, like this:

```
conn = connect("host=db_host user=some_user dbname=database"  
"password=$secreT")
```

Alternatively, named parameters can be used, like this:

```
conn = connect(host="db_host", user="some_user",  
dbname="database", password="$secreT")
```



## psycopg2 - cursors

```
cursor = connection.cursor()
```

- not a real database cursor, only an API abstraction
- think “statement handle”



## psycopg2 - server-side cursors

```
cur = connection.cursor(name='mycursor')
```

- a real database cursor
- use for large result sets



# psycopg2 - closing a database session

Make the changes to the database persistent

- `conn.commit()`

Close the communication with database

- `cur.close()`
- `conn.close()`



## psycopg2 - create tables

To create a new table in a PostgreSQL database, you use the following steps:

- First, construct a CREATE TABLE statement
- Next, connect to the PostgreSQL database by calling the `connect ()` function. The `connect ()` function returns a connection object
- Then, create a cursor object by calling the `cursor ()` method of the connection object
- After that, execute the CREATE TABLE by calling the `execute ()` method of the cursor object
- Finally, close the communication with the PostgreSQL database server by calling the `close ()` methods of the cursor and connection objects

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week3/psycopg2-suppliers-create.py>



## psycopg2 - transaction

In psycopg, the connection class is responsible for handling transactions. When you issue the first SQL statement to the PostgreSQL database using a cursor object, **psycopg creates a new transaction**. From that moment, **psycopg executes all the subsequent statements in the same transaction**. **If any statement fails, psycopg will abort the transaction**.

The connection class has two methods for terminating a transaction: `commit()` and `rollback()`. If you want to **commit all changes to the PostgreSQL database permanently**, you call the `commit()` method. And **in case you want to cancel the changes**, you call the `rollback()` method. Closing the connection object or destroying it using the `del` will also result in an implicit rollback.





## psycopg2 - transaction

It is important to notice that a simple `SELECT` statement will start a transaction that may result in undesirable effects such as table bloat and locks. Therefore, if you are developing a long-living application, you should call the `commit()` or `rollback()` method before leaving the connection unused for a long time.

Alternatively, you can set the `autocommit` attribute of the connection object to `True`. This ensures that psycopg will execute every statement and commit it immediately.

The `autocommit` mode is also useful when you execute statements required to execute outside a transaction such as `CREATE DATABASE` and `VACUUM`.



## psycopg2 - transaction

Suppose we need to add a new part and assign the vendors who supply the part at the same time. To do this, first, we insert a new row into the parts table and get the part id. Then, we insert rows into the vendor\_parts table. The following `add_part()` function demonstrates the idea.

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week3/psycopg2-suppliers-transaction.py>



## psycopg2 - insert

To insert a row into a PostgreSQL table in Python, you use the following steps:

- First, connect to the PostgreSQL database server by calling the `connect()` function of the `psycopg2` module. The `connect()` function returns a new instance of the connection class
- Next, create a new cursor object by calling the `cursor()` method of the connection object
- Then, execute the INSERT statement with the input values by calling the `execute()` method of the cursor object
- You pass the INSERT statement to the first parameter and a list of values to the second parameter of the `execute()` method



## psycopg2 - insert one row

In case the primary key of the table is an auto-generated column, you can get the generated ID back after inserting the row. To do this, in the INSERT statement, you use the RETURNING id clause. After calling the `execute()` method, you call the `fetchone()` method of the cursor object to get the id value as follows:

```
id = cur.fetchone()[0]
```



## psycopg2 - insert one row

After that, call the `commit()` method of the connection object to save the changes to the database permanently. If you forget to call the `commit()` method, psycopg will not change anything to the database. Finally, close the communication with the PostgreSQL database server by calling the `close()` method of the cursor and connection objects.

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week3/psycopg2-suppliers-insert-one-row.py>



## psycopg2 - insert multiple rows

The steps for inserting multiple rows into a table are similar to the steps of inserting one row, except that in the third step, instead of calling the `execute()` method of the cursor object, you call the `executemany()` method.

For example, the following `insert_vendor_list()` function inserts multiple rows into the vendors table.

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week3/psycopg2-suppliers-insert-manny-rows.py>



## psycopg2 - update

First, we need create a new cursor object by calling the `cursor()` method of the connection object. Then, execute the UPDATE statement with the input values by calling the `execute()` method of the cursor object.

```
cur.execute(update_sql, (value1,value2))
```

The `execute()` method accepts two parameters. The first parameter is an SQL statement to be executed, in this case, it is the UPDATE statement. The second parameter is a list of input values that you want to pass to the UPDATE statement.



## psycopg2 - update

If you want to get the number of rows affected by the UPDATE statement, you can get it from the `rowcount` attribute of the cursor object after calling the `execute()` method. After that, save the changes to the database permanently by calling the `commit()` method of the connection object. Finally, close the communication with the PostgreSQL database server by calling the `close()` method of the cursor and connection objects. We will use the `vendors` table in the `suppliers` database that we created in the creating table tutorial for the sake of demonstration. Suppose a vendor changed its name and we want to update the changes in the `vendors` table. To do this, we develop the `update_vendor()` function that updates the vendor name based on the vendor id.

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week3/psycopg2-suppliers-update.py>





## psycopg2 - delete

First, create a new database connection by calling the `connect()` function of the `psycopg` module. Next, to execute any statement, you need a cursor object, you call the `cursor()` method of the connection object. Then, execute the DELETE statement. If you want to pass values to the DELETE statement, you use the placeholders (`%s`) in the DELETE statement and pass input values to the second parameter of the `execute()` method.

The DELETE statement with a placeholder for the value of the `id` field is as follows:

```
DELETE FROM table_1 WHERE id = %s;
```



## psycopg2 - delete

To bind value `value_1` to the placeholder, you call the `execute()` method and pass the input value as a tuple to the second parameter like the following:

```
cur.execute(delete_sql, (value_1,))
```

After that, save the changes to the database permanently by calling the `commit()` method of the connection object. Finally, close the communication with the PostgreSQL database server by calling the `close()` method of the cursor and connection objects.

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems/blob/master/week3/psycopg2-suppliers-update.py>



**QA**



# Be up to date

Make sure to star this repo, and click on watch for the code updates.

<https://github.com/sureshmelvinsigera/CSC-424-Advanced-Database-Management-Systems>