Programming with Databases

Programming with Databases

So far, we have seen ...

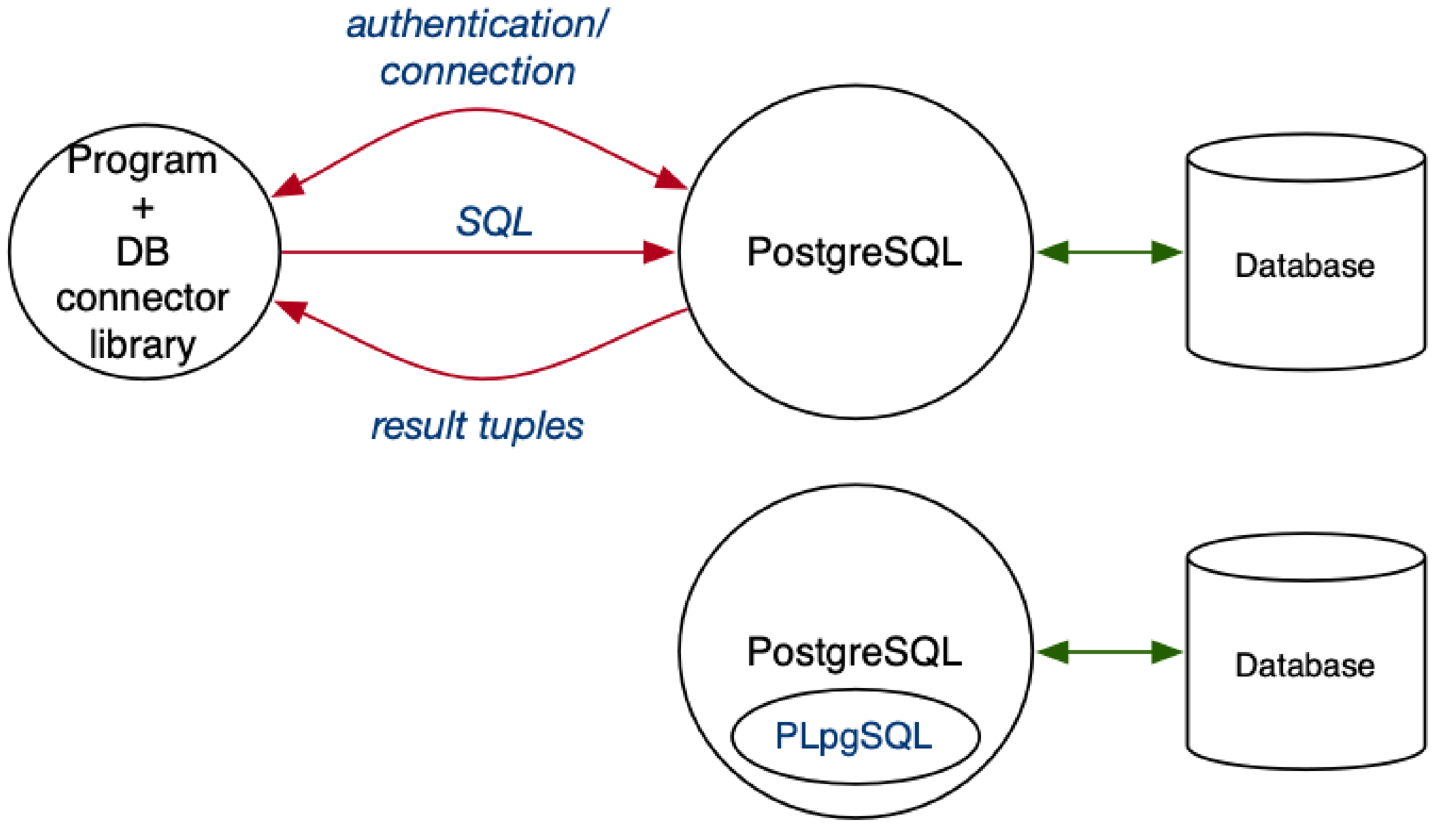
- accessing data via SQL queries
- packaging SQL queries as views/functions
- building functions to return tables
- implementing assertions via triggers

All of the above programming

- is very close to the data
- takes place inside the DBMS

Programming with Databases (cont)

Programming language access to DBMSs



Note the PLpgSQL is "closer" to the DBMS than PLs

Programming with Databases (cont)

Complete applications require code outside the DBMS

- to handle the user interface (GUI or Web)

- to interact with other systems (e.g. other DBs)
- to perform compute-intensive work (vs. data-intensive)

"Conventional" programming languages (PLs) provide these.

We consider four families of programming language:

- logic programming, functional programming
- procedural programming, object-oriented programming

and how they might interface with the relational model.

Programming with Databases (cont)

Requirements of an interface between PL and RDBMS:

- mechanism for connecting to the DBMS
- mapping betwen tuples and PL objects
- mechanism for mapping PL "requests" to queries
- mechanism for iterating over query results

Distance between PL and DBMS is variable, e.g.

- **`libpq`** allows C programs to use PG structs
- JDBC transmits SQL strings, retrieves tuples-as-objects
- FP interfaces operate on lists (sets) of tuples

Programming with Databases (cont)

Levels of abstraction in DB/PL interfaces

- high: DB hidden behind objects, no DB schema/SQL, portable
  - e.g. Active Record, SQLalchemy, ...
- medium: write schema, SQL construction functions, portable?
  - e.g. CodeIgniter DB interface
- low: write schema, write SQL, not portable?
  - e.g. JDBC, PDO, COMP3311 DB library, ...

Since this is a DB course, we consider low-level abstraction

Functional Languages and RDBMS

Functional programming languages (FPLs) (e.g. Haskell)

- understand tuples and sets (arrays) of tuples
- provide methods for extracting tuple components
- list comprehensions provide a mechanism for
  - generating results, filtering, joining, ...
- but are missing e.g. foreign key checking

Examples courtesy of Stefan Stanczyk, University of Linz, Austria. Note: the Haskell code is approximate; no guarantees it compiles.

Functional Languages and RDBMS (cont)

Example: airports, airlines database

```
data Country = GBR | USA | NZL  deriving (Eq,Show)
data Airline = BA | UA | NZ  deriving (Eq,Show)
data Airport = LHR | JFK | LAX | AKL deriving (Eq,Show)
```

```
allCountries :: [Country]
allCountries = [GBR, USA, NZL]

allAirlines :: [Airline]
allAirlines = [BA, UA, NZ]

allAirports :: [Airport]
allAirports = [LHR, JFK, LAX, AKL]

type CountryData  =  (String,Integer)
countryInfo :: Country -> CountryData
countryInfo GBR = ("Britain", 60)
countryInfo UA = ("United States", 250)
countryInfo NZ = ("New Zealand", 3)

type AirlineData  =  (String,Country)
airlineInfo :: Airline -> AirlineData
airlineInfo BA = ("British Airways", GBR)
airlineInfo UA = ("United Airlines", USA)
airlineInfo NZ = ("Air New Zealand", NZL)

type AirportData  =  (String,String,Country)
airportInfo :: Airport -> AirportData
airportInfo LHR = ("Heathrow", "London", GBR)
airportInfo JFK = ("JFK Intl", "New York", USA)
airportInfo LAX = ("LA Intl", "Los Angeles", USA)
airportInfo AKL = ("Auckland", "Auckland", NZL)
```

Functional Languages and RDBMS (cont)

```
-- extracting fields

countryName :: Country -> String
countryName x = first (countryInfo x)
countryPopulation :: Country -> Integer
countryPopulation x = second (countryInfo x)

airlineName :: Airline -> String
airlineName x = first (airlineInfo x)
airlineHome :: Airline -> Country
airlineHome x = second (airlineInfo x)

airportName :: Airport -> String
airportName x = first (airportInfo x)
airportCity :: Airport -> String
airportCity x = second (airportInfo x)
airportCountry :: Airport -> Country
airportCountry x = third (airportInfo x)
```

Functional Languages and RDBMS (cont)

```
# how many airlines are there?

q1 = length allAirlines
```

```
# names of all countries

q2 = [ countryName c | c <- allCountries ]
#or
q2 = map countryName allCountries

# which airports are located in the USA?

q3 = [ ap | ap <- allAirports, airportCountry = USA ]

# which airports are in United's home country?

q4 = [ ap | ap <- allAirports,
            airportCountry ap = (airlineHome UA) ]
```

Functional Languages and RDBMS (cont)

```
# country (or countries) with smallest population

smallestPop :: [Country] -> Integer
smallestPop cs = minimum (map countryPopulation allCountries)

q5 = [ c | c <- allCountries,
           countryPopulation c = smallestPop allcountries]

# which airline(s) have their home in Britain
```

```
q6 = [ a | a <- allAirlines,
           countryName (airlineHome a) = "Britain" ]
# or
q6 = [ a | a <- allAirlines, b <- allCountries,
           countryName b = "Britain", airlineHome a = b ]

# which airlines have home in small population country?

q6 = [ a | a <- allAirlines, b <- q5, airlineHome a = b ]
```

Logic Languages and RDBMS

Logic programming languages (LPLs) (e.g. Prolog/Datalog)

- have a notion of tuples (aka ground facts)
- understand sets of tuples (aka ground predicates)
- have a notion of querying (deduction)
- provide flexible result instantiation (variables)

LPLs are an excellent fit with the relational model.

If developed first, may have been better than SQL.

Logic Languages and RDBMS (cont)

Example: airports, airlines database

```
country('GBR', 'Britain', 60).
country('USA', 'United States', 250).
country('NZL', 'New Zealand', 3).
country('AUS', 'Australia', 22).

city('London', 'GBR').
city('New York','USA').
city('Los Angeles','USA').
city('Auckland','NZL').
city('Sydney','AUS').
city('Melbourne','AUS').

airline('BA', 'British Airways', 'GBR').
airline('UA', 'United Airlines', 'USA').
airline('NZ', 'Air New Zealand', 'NZL').
airline('QF', 'Qantas', 'AUS').

airport('LHR', 'Heathrow', 'London').
airport('JFK', 'JFK Intl', 'New York').
airport('LAX', 'LA Intl', 'Los Angeles').
airport('AKL', 'Auckland', 'Auckland').
airport('SYD', 'Kingsford-Smith', 'Sydney').
airport('MEL', 'Tullamarine', 'Melbourne').
```

Logic Languages and RDBMS (cont)

```
-- extracting fields

countryName(C,N) :- country(C,N,_).
countryPopulation(C,P) :- country(C,_,P).

airlineName(A,N) :- airline(A,N,_).
airlineHome(A,H) :- airline(A,_,H).

airportName(A,N) :- airport(A,N,_).
airportCity(A,C) :- airport(A,_,C).
airportCountry(A,C) :- airport(A,_,X), city(X,C).

# usage ...
?- countryName('GBR',N).
N = Britain
?- airportCountry('MEL',C).
C = Australia
```

Logic Languages and RDBMS (cont)

```
# how many airlines are there?

q1(N) :- findall(A,airline(A,_,_),All), length(All,N).
```

```
# names of all countries

q2(C) :- country(_,C,_).

# which airports are located in the USA?

q3(A) :- airport(A,_,C), city(C,'USA').

# which airports are in United's home country?

q4(A) :- airline('UA',_,Country),
         city(City,Country), airport(A,_,City).
```

Logic Languages and RDBMS (cont)

```
# country (or countries) with smallest population

smallestPop(P) :-
   findall(Pop,country(_,_,Pop),Pops), min_list(Pops,P).

q5(C) :- smallestPop(Min), country(C,_,Min).

# which airline(s) have their home in Britain

q6(A) :- airline(A,_,C), country(C,'Britain',_).

# which airlines have home in small population country?
```

```
q7(A) :- airline(A,_,C), q5(C).
```

Procedural Lanaguages and RDBMS

Most RDBMSs have a low-level C library which provide

- **`structs`** for tuples, queries, connections
- function to establish DB connection (authentication)
- function to initiate a query
  - send SQL query string to DBMS via established connection
  - return "handle" to access tuples in result set
- function to retrieve next tuple in result set
- methods for extracting contents of tuple

Higher-level languages and scripting languages typically have a DB interface.

Examples: Java/JDBC, PHP/PDO, Perl/DBI

Above APIs are DB agnostic; DBMS-specific interfaces are also avaiable.

Example: PHP's **`pg_connect()`**, **`pg_query()`**, **`pg_fetch()`**

COMP3311 previous used a simple PHP–PostgreSQL library.

Example PHP/PGSQL Interaction

Assuming same airline database as for previous examples ...

```
$db = dbConnect("dbname=airlines");

# how many airlines are there?

$query = "select count(*) from Airlines";
$nAirlines = dbOneValue($db, $query);

# names of all countries

$query = "select country from Countries";
$result = dbQuer($db, $query);
while ($tuple = dbNext($result))
   echo $tuple[1],"\n";

# which airports are located in the USA?

$query = <<_SQL_
_SQL_;
```

```
$result = dbQuery($db, $query);

# which airports are in United's home country?

q4(A) :- airline('UA',_,Country),
         city(City,Country), airport(A,_,City).
```

Example PHP/PGSQL Interaction (cont)

```
# country (or countries) with smallest population

smallestPop(P) :-
   findall(Pop,country(_,_,Pop),Pops), min_list(Pops,P).

q5(C) :- smallestPop(Min), country(C,_,Min).

# which airline(s) have their home in Britain

q6(A) :- airline(A,_,C), country(C,'Britain',_).

# which airlines have home in small population country?

q7(A) :- airline(A,_,C), q5(C).
```

Object-oriented Languages and RDBMS

Associating OO programs and RDBMS requires

- mapping between DB structures and objects
- representation for connections, queries, result sets

Two major mapping approaches have been developed:

- record-based mapping (e.g. JDBC, PDO)
- object-relational mapping (e.g. Hibernate, Active Record)

Object-oriented Languages and RDBMS (cont)

Record-based Mapping:

- implement database using conventional methods
- implement application code considering DB design
- good: good DB design, can tune DB performance
- bad: manual mapping between application/DB objects

Object-relational Mapping:

- provide data structures in OO (possibly via wrapper)
- system builds DB access methods automatically
- good: easy, maintainable, no SQL
- bad: often leads to poor usage of the database

Object-oriented Languages and RDBMS (cont)

Often DB access is packaged within a large OO framework

- e.g. Java, J2EE and its Data Access Objects (DAOs)

The idea behind this:

- business objects are represented by a collection of values
- values may be spread across multiple tables
- implement a business object class with operations
    - **`create()`** ... inserts new tuple(s), given object values
    - **`getData()`** ... fetch value of (typically) one attribute
    - **`setData()`** ... update value of (typically) one attribute

Object-oriented Languages and RDBMS (cont)

Persistence frameworks (e.g. Hibernate) simplify DAOs

- developer defines own database tables
- developer gives mapping from tables to application objects
- framework produces mapping methods
- developer then programs solely with application objects

However, reference back to DB structures is needed

- if ORM produces poor execution performance
- for some complex data manipulations (which are SQL-friendly)

Unfortunately, solution is often *x*QL rather than SQL.

Object-oriented Languages and RDBMS (cont)

Alternative approach: the Active Record design pattern.

Used in some strongly OO contexts (e.g. Ruby-on-Rails)

Treats tuples as core objects:

- requires user to follow conventions in defining tables
- uses DBMS metadata to derive access methods
- provides access to DB via objects, no SQL needed
- automatically generates methods to access tables

Similar to persistence framework, but uses convention rather than configuration.

PL/DB Interface

Common pattern used by record-based libraries:

```
db = connect_to_dbms(DBname,User/Password);
query = build_SQL("sql_statement_template",values);
results = execute_query(db,query);
while (more_tuples_in(results))
{
   tuple = fetch_row_from(results);
```

```
    // do something with values in tuple ...
}
```

This pattern is used in many different libraries:

- Java/JDBC, PHP/PDO, Perl/DBI, Python/dbapi2, Tcl, ...

PL/DB Interface (cont)

All record-based libraries have the same overall structure.

They differ in the details:

- whether specific to one database or generic
- whether object-oriented or procedural flavour
- function/method names and parameters
- how to get data from program into SQL statements
- how to get data from tuples to program variables

We use PHP and a simple, locally-developed library for this.

DB/PL Mismatch

There is a tension between PLs and DBMSs

- DBMSs deal efficiently with sets of tuples
- PLs encourage dealing with single tuples/objects

If not handled carefully, can lead to inefficient use of DB.

Note: relative costs of DB access operations:

- establishing a DBMS connection ... very high
- initiating an SQL query ... high
- accessing individual tuple ... low

DB/PL Mismatch (cont)

Consider the PL/DBMS access method, phrased in a made-up DB/PL api

```
--  establish connection to DBMS
db = dbAccess("DB");
query = "select a,b from R,S where ... ";
--  invoke query and get handle to result set
results = runQuery(db, query);
--  for each tuple in result set
while (tuple = getNext(results)) {
    --  process next tuple
    process(tuple['a'], tuple['b']);
}
```

DB/PL Mismatch (cont)

Example database:

```
Students(id, name, ...)
Marks(student, course, mark, ...)
```

where there are

- 1000 Students
- 10000 Marks tuples

DB/PL Mismatch (cont)

Example: find mature-age students

```
query = "select * from Student";
results = runQuery(db,query);
while (tuple = getNext($results)) {
    if (tuple['age'] >= 40) {
        --  process mature-age student
    }
}
```

If 1000 students, and only 50 of them are over 40,
we transfer 950 unnecessary tuples from DB.

DB/PL Mismatch (cont)

E.g. should be implemented as:

```
query = "select * from Student where age >= 40";
results = runQuery(db,query);
while (tuple = getNext(results)) {
        --  process mature-age student
    }
```

Transfers only the 50 tuples that are needed.

DB/PL Mismatch (cont)

Example: find info about all marks for all students

```
query1 = "select id,name from Student";
res1 = runQuery(db,query1);
while (tuple1 = getNext(res1)) {
    query2 = "select course,mark from Marks"+
              " where student = "+tuple1['id'];
    res2 = runQuery(db,query2);
    while (tuple2 = getNext(res2)) {
        --  process student/course/mark info
    }
}
```

We invoke 1001 queries on database (outer query + 1 per student)

If average 10 Marks/Student, 10000 tuples transferred

DB/PL Mismatch (cont)

E.g. should be implemented as:

```
query = "select id,name,course,mark"+
         " from Student s, Marks m"+
         " where s.id = m.student";
results = runQuery(db,query);
while (tuple = getNext(results)) {
    --  process student/course/mark info
}
```

We invoke 1 query, and transfer 10000 tuples

Exercise: DB/PL Mismatch

Give two different approaches for producing a table of

- student ID and name
- maximum mark they ever obtained for any course

Analyse the cost for each in terms of

- number of queries made on the database
- number of tuples read

Assume 1000 student tuples with 10 marks per student (on average).

Exercise: DB/PL Mismatch (cont)

Solution1: most work done by database

```
query = "
select s.id, s.name, max(m.mark)
from   Students s join Marks m on m.student = s.id
group  by s.id, s.name
";
result = runQuery(db,query)
while (tuple = getNext(result)
        print(tuple['id'],tuple['max'])
```

Runs 1 query; tranfers 10000 tuples from DB to program

Exercise: DB/PL Mismatch (cont)

Solution2: multiple queries

```
q1 = "select s.id, s.name from Students";
res1 = runQuery(db,q1)
while (tuple =getNext(result) {
        tuple = getNext(res1)
        sid = tuple['id']
        max = -1
        q2 = "select mark from Marks "+
             "where students = "+sid
        res2 = dbQuery(db, q2)
        while (tup2 = getNext(res2)) {
                mark = tup2['mark']
                if (mark > max) max = mark
}       }
print(sid,max)
```

Runs 1001 queries; tranfers 10000 tuples from DB to program

Exercise: DB/PL Mismatch (cont)

Solution3: inefficient strategy (no **where** clauses)

```
q1 = "select s.id, s.name from Students";
res1 = runQuery(db,q1)
while (tuple =getNext(result) {
        tuple = getNext(res1)
        sid = tuple['id']
        max = -1
        q2 = "select student, mark from Marks"
        res2 = dbQuery(db, q2)
        while (tup2 = getNext(res2)) {
                if ($tup2['student'] != sid) continue
                mark = tup2['mark']
                if (mark > max) max = mark
}       }
print(sid,max)
```

Runs 1001 queries; tranfers 100000 tuples from DB to program

---

§ Python and DB Access

Python

Python is a very popular programming language

- easy to learn/use
- with a wide range of useful libraries

We assume that you know enough Python to manipulate DBs

- the primary goal is Database, not Python programming

If you're not overly familiar with Python ...

- there will be many examples of Python code in this course
- there are many excellent tutorials online

Psycopg2

Psycopg2 is a Python module that provides

- a method to connect to PostgreSQL databases

- a collection of DB-related exceptions
- a collection of type and object constructors

In order to use Psycopg2 in a Python program

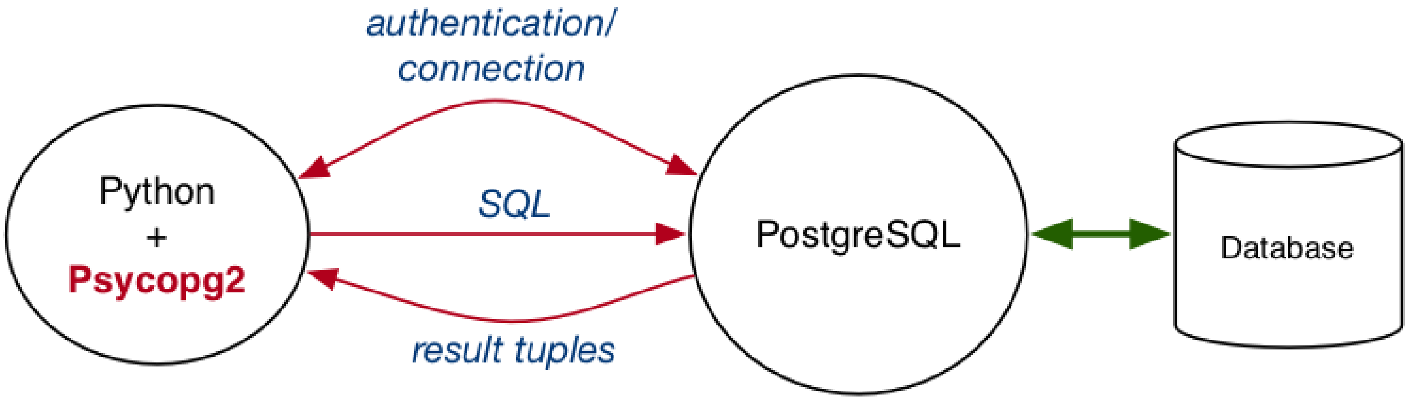
```
import psycopg2
```

Note:

- assumes that the Psycopg2 module is installed on your system
- Psycopg2 is installed on Grieg for both python2 and python3

Psycopg2 (cont)

Where **`psycopg2`** fits in the PL/DB architecture



Database **`connection`**s

`conn = psycopg2.connect(`*DB_connection_string***`)`**

- creates a **`connection`** object on a named database
- effectively starts a session with the database (cf **`psql`**)
- returns a **`connection`** object used to access DB
- if can't connect, raises an exception

DB connection string components

- **`dbname`** ... name of database
- **`user`** ... user name (for authentication)

- **`password`** ... user password (for authentication)
- **`host`** ... where is the server running (default=localhost)
- **`port`** ... which port is server listening on (default=5432)

On Grieg, only **`dbname`** is required.

Example: connecting to a database

Simple script that connects and then closes connection

```
import psycopg2

try:
   conn = psycopg2.connect("dbname=mydb")
   print(conn)  # state of connection after opening
   conn.close()
   print(conn)  # state of connection after closing
except Exception as e:
   print("Unable to connect to the database")
```

which, if **`mydb`** is accessible, produces output:

```
$ python3 ex1.py
<connection object at 0xf67186ec; dsn: 'dbname=mydb', closed: 0>
<connection object at 0xf67186ec; dsn: 'dbname=mydb', closed: 1>
```

Operations on `connections`

`cur = conn.cursor()`

- set up a handle for performing queries/updates on database
- must create a **`cursor`** before performing any DB operations

`conn.close()`

- close the database connection **`conn`**

`conn.commit()`

- commit changes made to database since last **`commit()`**

Plus many others ... see Psycopg2 documentation

Database **Cursors**

Cursors are "pipelines" to the database

Cursor objects allow you to ...

- execute queries, perform updates, change meta–data

Cursors are created from a database **connection**

- can create multiple cursors from the same connection
- each cursor handles one DB operation at a time
- but cursors are not isolated (can see each others' changes)

To set up a **cursor** object called **cur** ...

```
cur = conn.cursor()
```

Operations on **cursors**

cur.execute(*SQL_statement*, *Values***)**

- if supplied, insert values into the SQL statement
- then execute the SQL statement
- results are available via the cursor's fetch methods

Examples:

```
# run a fixed query
cur.execute("select * from R where x = 1");

# run a query with values inserted
cur.execute("select * from R where x = %s", (1,))
cur.execute("select * from R where x = %s", [1])

# run a query stored in a variable
query = "select * from Students where name ilike %s"
pattern = "%mith%"
cur.execute(query, [pattern])
```

Operations on `cursors` (cont)

`cur.mogrify(`*SQL_statement*`,` *Values*`)`

- return the SQL statement as a string, with values inserted
- useful for checking whether **`execute()`** is doing what you want

Examples:

```
query = "select * from R where x = %s"
print(cur.mogrify(query, [1]))
Produces: b'select * from R where x = 1'

query = "select * from R where x = %s and y = %s"
print(cur.mogrify(query, [1,5]))
Produces: b'select * from R where x = 1 and y = 5'

query = "select * from Students where name ilike %s"
pattern = "%mith%"
print(cur.mogrify(query, [pattern]))
Produces: b"select * from Students where name ilike '%mith%'"

query = "select * from Students where family = %s"
fname = "O'Reilly"
print(cur.mogrify(query, [fname]))
Produces: b"select * from Students where family = 'O''Reilly'"
```

Operations on **`cursors`** (cont)

`list = cur.fetchall()`

- gets all answers for a query and stores in a list of tuples
- can iterate through list of results using Python's **`for`**

Example:

```
# table R contains (1,2), (2,1), ...

cur.execute("select * from R")
for tup in cur.fetchall():
    x,y = tup
    print(x,y)    # or print(tup[0],tup[1])

# prints
1 2
2 1
...
```

Operations on **`cursors`** (cont)

`tup = cur.fetchone()`

- gets next result for a query and stores in a tuple
- can iterate through list of results using Python's **`while`**

Example:

```
# table R contains (1,2), (2,1), ...

cur.execute("select * from R")
while True:
   tup = cur.fetchone()
   if tup == None:
      break
   x,y = tup
   print(x,y)

# prints
1 2
2 1
...
```

Operations on **`cursors`** (cont)

`tup = cur.fetchmany(`*nTuples*`)`

- gets next *nTuples* results for a query
- stores tuples in a list
- when no results left, returns empty list

Example:

```
# table R contains (1,2), (2,1), ...

cur.execute("select * from R")
while True:
   tups = cur.fetchmany(3)
   if tups == []:
      break
   for tup in tups:
      x,y = tup
      print(x,y)

# prints
1 2
2 1
...
```

Produced: 13 Sep 2020