Database-backed applications

ISYS2120 Data and Information Management

Prof Alan Fekete

University of Sydney

Acknowledge: slides from Uwe Roehm and Alan Fekete, and from the materials associated with reference books (c) McGraw-Hill, Pearson

Database-backed software

- Most of the software used for life and business needs to use data that is found in a database
  - Often, user activities will also modify the data in the database
- A few examples
  - E-commerce (inventory, purchase, order status, order history)
  - Entertainment (catalogue, preferences, history)
  - Social media (posts, community connections)
  - Transport (routes, timetables, current status)
- The software that provides these functionalities, needs to access one or more dbms

Software and queries

- Previously in isys2120, we covered how to write SQL to extract and modify data
- A human typed the query into a query window and ran it against the database, and observed the output
- End-users can’t be expected to know SQL, nor usually to have accounts and access on all the dbms needed for modern life
- Instead, the user invokes some software, and the software submits queries to dbms, gets result table, and displays information to the user
- This lecture is about the structure of that software

Data-intensive Systems

- Three types of functionality (often placed in separate layers of code):

Layer	
Presentation Logic - Input – keyboard/mouse/gestures - Output – monitor/printer/screen	**GUI Interface**
Processing Logic - Business rules - I/O processing	**Procedures, functions, programs**
Data Management (Storage Logic) - data storage and retrieval	**DBMS activities**

- The system architecture determines whether these three components reside on a single computing system (1-tier) or whether they are distributed across several tiers

Presentation layer

- Often, web browser is used as GUI for end-user
  - Available on all devices!
- Application needs to have code that lays out the web pages and provides navigation between them
- Mobile devices may provide an app that directly works with the gestures etc of the device
  - And is targeted for the small screen size
- Lab08 and Asst3 will work with web interface
  - Flask library of Python, to construct a web-server that also runs business logic and data management

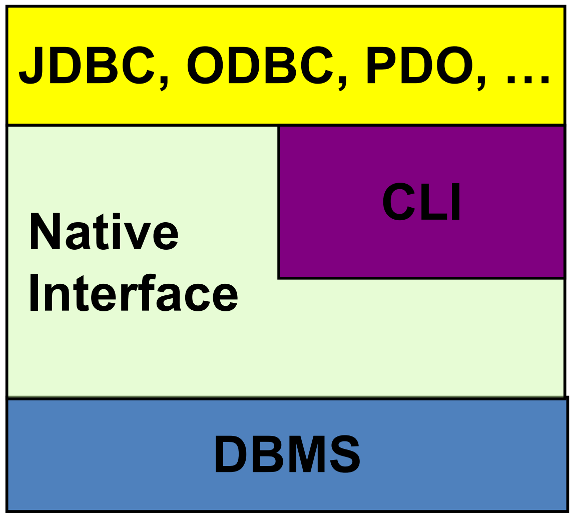
SQL in Application Code

- SQL commands can be called from within a *host language* (such as Python or Java) program.
  - Must include a statement to *connect* to the right database.
  - SQL statements can refer to host variables (including special variables used to return status).

- Two main integration approaches:
  - **Statement-level interface** (SLI)
    - Embed SQL in the host language (Embedded SQL in C, SQLJ)
    - Application program is a mixture of host language statements and SQL statements and directives
    - A special compiler must deal with both aspects
  - **Call-level interface** (CLI) This is what Flask uses
    - Create special API to call SQL commands (JDBC, ODBC, Python, …)
    - SQL statements are passed as arguments to host language (library) procedures / APIs
    - Standard programming language compiler, and program is combined with a library that supports the API

Call-level Interfaces and Database APIs

- Program can invoke methods/procedures in a library with database calls (API)
  - Pass SQL strings from language, present result sets in language-friendly way
  - Supposedly DBMS-neutral
    - a “driver” executes the calls and translates them into DBMS-specific code
    - database can be across a network
- Several Variants
  - **SQL/CLI**: “SQL Call-Level-Interface”
    - Part of the SQL-92 standard;
    - “The assembler under the APIs”
  - **ODBC**: “Open DataBase Connectivity”
    - Side-branch of early version of SQL/CLI
    - Enhanced to: **OLE/db,** and further **ADO.NET**
  - **JDBC**: “Java DataBase Connectivity”
    - Java standard
  - **PDO**
    - Persistency standard for PHP Data Objects



Whose privileges when code is run?

- Many databases are accessed indirectly
  - End-user does not write and submit SQL, but rather runs a program that (team of) coders have written to perform useful activity
  - Eg a student changes their address through MyUni
- The program can do lots of checking of whether access is appropriate, before sending SQL to dbms
  - Also the program can filter or summarise data, so user does not see everything the program gets from the dbms
- The program may run with its own appropriate level of privilege (or that or the coders), rather than from the end-user who is the source of request
  - Indeed, the end-user may not have a dbms account at all
- Often, the program has quite a lot of privilege, but this is risky if there are mistakes in the code, or if an attacker can obtain the program's credentials [eg if program uses a password which is stored somewhere, and leaked]

PYTHON DB-API2

a Call-Level API Example

Python

- Python features extensive standard library (modules)
  - Special functionality supported by variety of optional 3rd-party modules
  - For database connectivity, several database-specific python modules
    - e.g. psycopg or pg8000 (PostgreSQL) or cx_oracle (Oracle)
    - **https://pypi.org/project/pg8000/**
  - For dynamic websites:
    - several framework available; in lab8, asst3 we will use Flask
    - Allows to define template pages of html with embedded python code

Python Database API Specification (DB-API)

- DB-API 2.0 was released April 1999
- Defines common functions and API for access modules to different database systems
  - Module API; Connection and Cursor interface definitions
- Works as a generic as a **database abstraction layer**
  - Generic driver model to connect to different database engines via the same API

- URLs to learn more:

https://pypi.org/project/pg8000/
https://www.python.org/dev/peps/pep-0249/

http://initd.org/psycopg/docs
http://www.tutorialspoint.com/postgresql/postgresql_python.htm

https://wiki.python.org/moin/DatabaseProgramming
https://wiki.python.org/moin/UsingDbApiWithPostgres

Python DB-API Example

```
import pg8000

try:
    # connect to the database
    conn = pg8000.connect(database="postgres",user="test",password="secret")

    # prepare to query the database
    curs = conn.cursor()

    # execute a parameterised query
    unit_of_study = "ISYS2120"
    curs.execute("""SELECT name
                      FROM Student NATURAL JOIN Enrolled
                     WHERE uos_code = %(uos)s""", {'uos': unit_of_study} )

    # loop through the resultset
    for result in curs:
         print (" student: " + result[0])

    # clean up
    curs.close()
    conn.close()
except Exception as e: # error handling
   print("SQL error: unable to connect to database or execute query")
   print(e)
```

Core tasks with SQL Interfaces

(1) Establishing a database connection

(2) Static vs. Dynamic SQL

(3) Parameterized SQL and mapping of domain types to data types of host

- Concept of ***host variable***
- How to treat *NULL* values?

(4) Impedance Mismatch:

- SQL operates on sets of tuples
- Host languages like C do not support a set-of-records abstraction, but only a one-value-at-a-time semantic
- Solution: ***Cursor Concept***
  Iteration mechanism (loop) for processing a set of tuples

(5) Error handling

(1) DB Connections from Python

- Session with PostgreSQL started by creating a connection
- Two Variants:
  - Connect with keyword arguments

    ```
    conn =
    pg8000.connect(host='…',database='…',user='X',password='…')
    ```

  - Connect with a Data Source Name (*DSN*) string of the form

    "host=*X* dbname=*Y* user=*U* password=*P*"

    For example for PostgreSQL:

```
conn = pg8000.connect(
"host=postgres.usyd.edu.au dbname=unidb user=U
password=secret")
```

Python Database Connection Modules

- Python support for variety of DBMSs
  - MySQL (module: MySQLdb)
  - PostgreSQL (module: pg8000 or psycopg2)
  - Oracle (module: cx_oracle)
  - IBM DB2 (module: ibm_db)
  - SQL Server (module: pymssql)
  - sqlite (module: sqlite3)
  - …
  - DSN syntax and additional DB parameters vary for each driver
    - Check manuals…

Note:
db modules need to be installed first as part of the Python installation…

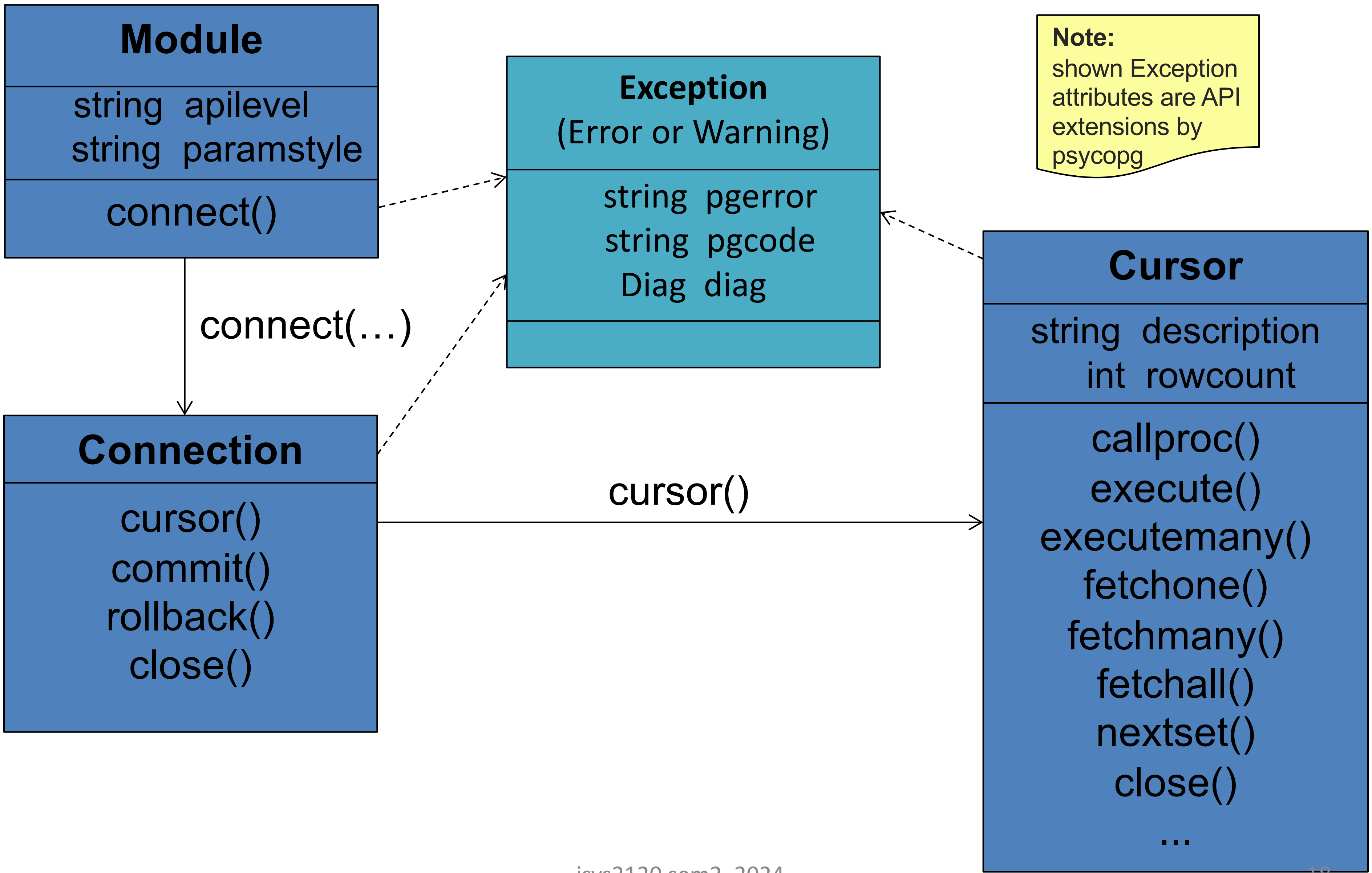
- Example for Oracle:

```
dsnStr = cx_oracle.makedsn("oracle10g.it.usyd.edu.au",1521,"ORCL")
conn = cx_oracle.connect(user="myuser",password="mypass",dsn=dsnStr)
```

pg8000 Connection Simple Example

```
import pg8000

# connect to the database
try:
    conn = pg8000.connect(database='foo', user='dbuser', password='pwd')
except:
    print("unable to connect to database")

# query database
curs = conn.cursor()
try:
   curs.execute("SELECT name FROM Student WHERE studID=4711")
except:
   print("unable to execute query")

… Do Actual Work ….

# cleanup
curs.close()
conn.close()
```

Python DB-API 2.0 Objects



Python Database API Interfaces

- Connection Management
  - **pg8000.connect()** connects to a database
  - **conn.cursor()** creates a cursor object for query execution
- Start SQL statements
  - **curs.execute()** for static SQL, and also parameterized SQL queries
  - **curs.callproc()** for executing a stored procedure including parameters
- Result retrieval
  - **curs.fetchone()** retrieves next row of a result or **None** when no more data
  - **curs.fetchall()** retrieves the whole (remaining) result set, and returns it as a list of tuples
- Transaction control
  - **conn.commit()** successfully finishes (commits) current transaction
  - **conn.rollback()** aborts current transaction
- Error Handling
  - Via standard exception handling of Python

Side Note on DB Connections

- Establishing a database connection takes some time…
  - Network communication, memory allocation, dbs authorization
- So do this only once in your program
  - … **not over and over** for individual SQL queries
- Modern, multi-threaded applications will typically want to have a pool of connections that are re-used
  - Might be handled by your runtime library (that's what happens in Python)
  - But for, e.g., Java programs better be mindful of connection costs!

(2) Static vs. Dynamic SQL

- SQL constructs in an application can take two forms:
  - Static SQL statements:
    Useful when SQL query is fully known at compile time
    - no parameters are allowed in the query string
    - only useful in context of compiled languages such as C
  - Dynamic SQL statements:
    Application determines SQL statements at *run time* as values of host language variables that are manipulated by directives.
    - Challenge: Python is not a compiled language;
      everything in Python/pg8000 is by definition dynamic SQL…
    - This means we have to be careful on how we construct any query
      and in particular how parameters are passed to the database

DB-API: Executing SQL Statements

- Three different ways of executing SQL statements:
  - ***cursor*.execute**(*sql*) semi-static SQL statements
  - ***cursor.*execute**(*sql,params*) parameterized SQL statements
  - ***cursor.*callproc**(*call,args*) invoke a stored procedure in DBMS
  - ***cursor*.executemany**(*sql,seq_of_params*) repeatedly executes parameterized SQL statements

- In DB-API 2.0,
  - Need to create new cursor and re-issue SQL statement each time when parameters change – or if possible use **executemany()**
  - Some other APIs offer “prepared statements” – parsed and optimized once in the dbms, then re-executed over and over with different parameters

Python DB-API with fixed SQL

- Simplest way to execute an unchanging SQL query:

```
import pg8000

try:
    # connect to the database
    conn = pg8000.connect(database='foo', user='dbuser', password='pwd')

    # query database
    curs = conn.cursor()
    curs.execute("SELECT name FROM Student WHERE studID=4711")
    result = curs.fetchone()
    print(result)

    # cleanup
    curs.close()
    conn.close()

except:
   print("unable to connect to db or to execute query")
```

DB-API: Batch Insert Example

- Example: executing batch INSERT statements

```
import pg8000

try:
    # connect to the database
    conn = pg8000.connect(database='foo', user='dbuser', password='pwd')

    # prepare list of insert values (3 students enrolling in ISYS2120)
    params = [(4711, ISYS2120'), (4712,'ISYS2120'), (4713,'ISYS2120') ]

    # execute INSERT statement batch
    curs = conn.cursor()
    curs.executemany("INSERT INTO Enrolled VALUES (%s,%s)", params)
    conn.commit() # cf. next week on transactions

    # cleanup
    curs.close()
    conn.close()

except:
   print("unable to connect to db or to execute query")
```

DB-API: Parameterized Queries

- Two (safe) approaches for passing query parameters:
  (because execute() will do any necessary escaping / conversions for parameter markers)

1. Anonymous Parameters

```
studid = 12345
cursor.execute(
        "SELECT name FROM Student WHERE sid=%s",
(studid,) )
```

This comma is no mistake, but needed with single parameters

parameter marker

2. Named Parameters

```
studid = 12345
cursor.execute(
      "SELECT name FROM Student WHERE sid=%(sid)s",
{'sid': studid} )
```

named parameter marker

(3) Parameterized SQL & Host Variables

- Data transfer between DBMS and application
- Mapping of SQL domain types to data types of host language
- Python DB-API:
  - Host variables are normal *dynamically typed* Python variables; automatic conversion to/from SQL types done by pg8000 in execute():

```
studid = 12345
stmt   = cursor.execute(
            "SELECT name FROM Student WHERE sid=%s",
            (studid,) )
```

- Note: in statement-level interface such as ESQL/C:
Host variables must be declared before usage

Variables shared by host and SQL

```
EXEC SQL BEGIN DECLARE SECTION;
      int  studid = 12345;
      char sname[21];
EXEC SQL END DECLARE SECTION;
```

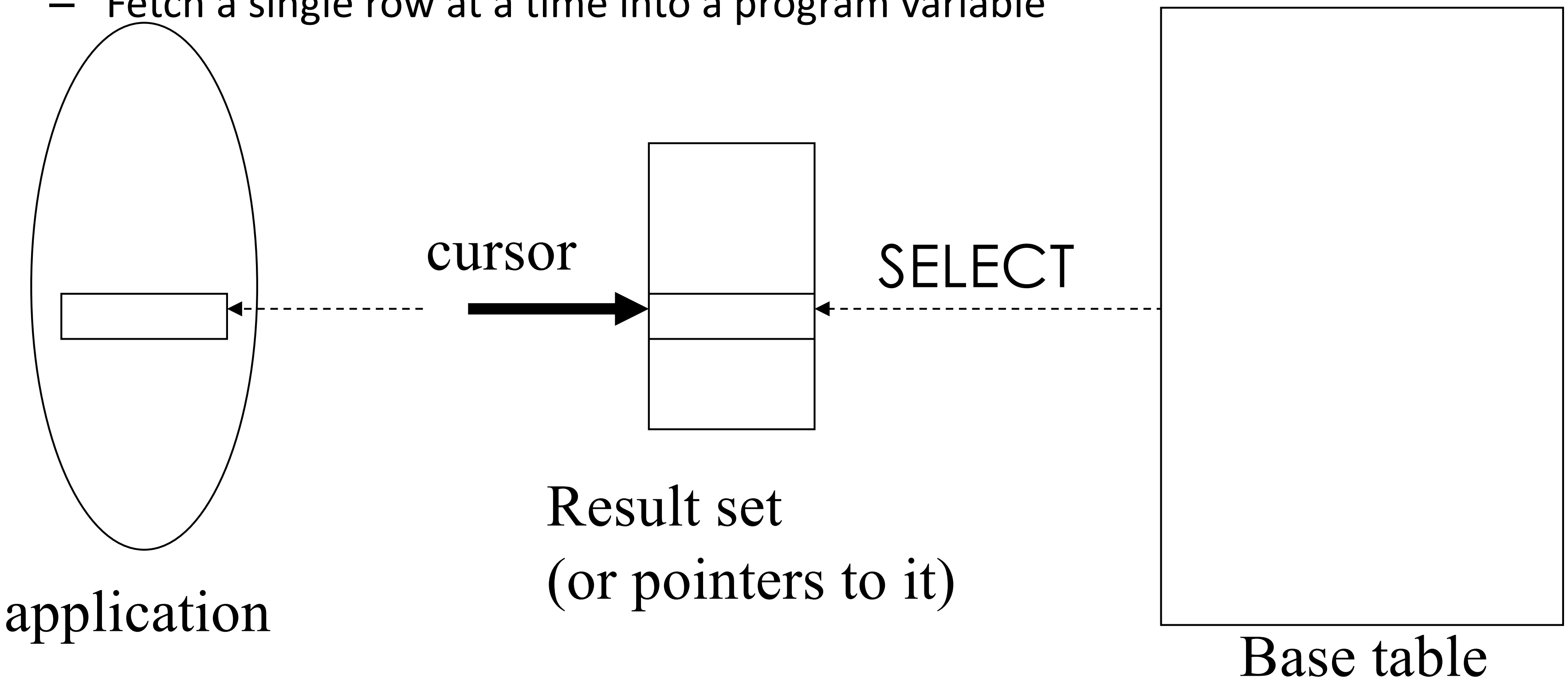
Specifying Date/Time: Type Objects

- Providing date or time values is very database specific with different database configurations requiring particular formats
  - eg. "26.04.2016" (AU) versus "04/26/2014" (US)
- DB-API has helper objects to safely convert database types
  - **Date** (*year* , *month* , *day*)
  - **Time** (*hour* , *minute* , *second*)
  - **Timestamp** (*year* , *month* , *day* , *hour* , *minute* , *second*)
  - **DateFromTicks** (*ticks*)
  - **TimeFromTicks** (*ticks*)
  - **TimestampFromTicks** (*ticks*)
  - **Binary** (*string*)

(4) Buffer Mismatch Problem

(also called: Impedance Mismatch)

- **Problem**: SQL deals with tables (of arbitrary size); host language program often deals with fixed size data types (this is less true with Python which has collections)
  - How is the application to allocate storage for the result of a SELECT statement?
- **Solution**: Cursor concept
  - Fetch a single row at a time into a program variable



Mapping of Sets: Cursor Concept

- ***Result set*** – set of rows produced by a SELECT statement
- ***Cursor*** – pointer to a row in the result set.
- Cursor operations:
  - *Declaration*
  - *Open* – execute SELECT to determine result set and initialize pointer
  - *Fetch* – advance pointer and retrieve next row (Python: fetchone() call)
  - *Close* – deallocate cursor

Cursor in Python– via Cursor Interface

- Cursor concept with Python/psycopg:

```
curs = conn.cursor()
curs.execute("SELECT title, name, address FROM Emp")
row = curs.fetchone()
while row is not None:
    print(row)
    row = curs.fetchone()
curs.close()
```

- Cursor objects are iterable, so shorter form is:

```
curs = conn.cursor()
curs.execute("SELECT title, name, address FROM Emp")
for row in curs:
    data = row[0] + "\t" + row[1] + "\t" + row[2] + "\n"
    print(data)
curs.close()
```

You address result columns by position

Cursor in Python – fetchAll()

- Fetchall() returns a Python list with all the result rows
  - Good for **small** results

```
curs.execute("SELECT title, name, address FROM Emp")
resultset = curs.fetchall()
curs.close()
for row in resultset:
    print(row)
```

  - just be mindful that this will be **memory intensive** for large results

Dictionary Cursors

- By default, pg8000 returns <u>tuples</u> with fetch() / fetchall()
  - fields can only be addressed positionally
- As an extension, pg8000 also supports dictionary cursors
  - Result is now a dictionary (associative array) which each field being named by the attribute names from the database schema

```
import pg8000
from   pg8000.extras import RealDictCursor
…
curs = conn.cursor(cursor_factory=RealDictCursor)
curs.execute("SELECT title, name, address FROM Emp")
for row in curs:
   data = row['title'] + "\t" + row['name'] + "\n"
   print(data)
curs.close()
```

NULL Handling in Python

- Remember: In SQL there is a special indication NULL used for unknown or inapplicable value of a column
  - Null value is not the same as 0 nor empty string
- In Python this shows as **None**:

```
cursor.execute("SELECT gender FROM Student …")
result = cursor.fetchone()
if result[0] is None:
     # null value
else:
     # no null value
```

- Other languages require a special *indicator variable*. Eg. C:

```
EXEC SQL select gender into :gender:indicator
                       from Student where
sid=4711;
if ( indicator == -1 )
{ /* null value */ }
else
{ /* no null value */ }
```

Testing for Variable Exists / is None

- In Python, to check for existence of a variable versus whether it has None as value:

```
# Ensure variable is defined
try:
    x
except NameError:
    x = None

# Test whether variable is defined to be None
if x is None :
    some_fallback_operation()
else:
    some_operation(x)
```

[Source: http://code.activestate.com/recipes/59892/]

(5) Error Handling

- Multitude of potential problems
  - No database connection or connection timeout
  - Wrong login or missing privileges
  - SQL syntax errors
  - Empty results
  - NULL values
  - …
- Hence always check database return values,
- Provide error handling code, resp. exception handlers
- Gracefully react to errors or empty results or NULL values
- **NEVER show database errors to end users**
  - Not only bad user experience, but huge security risk, because database errors can report table names, stack trace showing lines of code, etc

Error Handling with Python DB API

- Error handling via normal exception mechanism of Python
  - Errors and warnings are made available as Python exceptions
    - Warning raised for warnings such as data truncation on insert, etc
    - Error exception raised for various db-related errors
- psycopg has an API extension:
  - Exception attributes for detailed SQL error codes and messages
  - **pgerror** string of the error message returned by backend
  - **pgcode** string with the **SQLSTATE** error code returned by backend

Example:

```
try:
   conn = pg8000.connect(database="postgres",user="test",password="secret")

# error handling
except OperationalError as e:
   print("unable to connect to database")
except Exception as e:
   print("Error when querying database")
   print(e)
```

Note: please do not directly print SQL exceptions ;)

Exception Hierarchy of Python DB API

- The complete **Exception** inheritance hierarchy for the Python DB API is as follows:

```
StandardError
|__ Warning
|__ Error
    |__ InterfaceError
    |__ DatabaseError
        |__ DataError
        |__ OperationalError
        |__ IntegrityError
        |__ InternalError
        |__ ProgrammingError
        |__ NotSupportedError
```

[cf. http://initd.org/psycopg/docs/module.html?highlight=connect#exceptions]

Reprise: Example of Python DB-API

```python
import pg8000

try:
    # connect to the database
    conn = pg8000.connect(database="postgres",user="test",password="secret")

    # prepare to query the database
    curs = conn.cursor()

    # execute a parameterised query
    unit_of_study = "ISYS2120"
    curs.execute("""SELECT name
                      FROM Student NATURAL JOIN Enrolled
                     WHERE uos_code = %(uos)s""", {'uos': unit_of_study} )

    # loop through the resultset
    for result in curs:
         print (" student: " + result[0])

    # clean up
    curs.close()
    conn.close()

# error handling
except OperationalError as e:
   print("unable to connect to database")

except Exception as e:
   print("Error when querying database")
   print(e)
```

cursor concept

dynamic SQL with safe parameter passing

error handling

Stored Procedures (Server-side application logic)

- Recall: these run application logic within the database server
  - Included as schema element (stored in DBMS)
  - Invoked by the application
- Advantages:
  - Central code-base for all applications
  - Improved maintainability
  - Additional abstraction layer
    (programmers do not need to know the schema)
  - Reduced data transfer
  - Less long-held locks
  - DBMS-centric security and consistent logging/auditing (important!)
- Note: although refered to as procedures, can also be functions

Python DB-API: Calling Stored Procedures

- **Cursor** objects have an explicit callproc() method
  - cursor.callproc() makes the OUT parameters available as resultset
- Example:

```
CREATE FUNCTION test(input VARCHAR,OUT output
VARCHAR) AS $$
BEGIN
    output := UPPER(input);
END $$ LANGUAGE plpgsql;
```

```
import pg8000
conn = pg8000.connect(…)
curs = conn.cursor()
input = "foo bar"
curs.callproc("test", [input] )
output = curs.fetchone()
print(output[0])
```

Pass all IN parameters as a list in order of the function declaration

OUT parameters are returned as resultset

Language support of Stored Procedures

- Programming language virtual machine is often ‘integrated’ with DBMS
  - E.g. Java with Oracle
  - .Net CLR with IBM, Oracle, and SQL Server
  - PostgreSQL: Supports several scripting languages such as perl etc.
- But degree of integration differs heavily
  - If language VM is in a different process from dbms, then performance often suffers

Summary

- **Understand core issues for db-backed development**
  - Data and type conversion: *Host Variables*
  - NULL value semantic: *Indicator variables* and testing methods
  - Impedance Mismatch: *Cursor Concept*
  - *Dynamic* versus *static SQL: Passing parameters in and out*
- **Database APIs**
  - After lab08, you should be able to work with small Python db-accessing programs
- **Server-side database programming**
  - How to use stored procedures to run code inside a DBMS
    - e.g. with PostgreSQL's pl/pgsql
  - Modern database engines provide virtual machine environments to run external code near the data

References

- Silberschatz/Korth/Sudarshan(7ed)
  - Chapter 5.1, 9.1, 9.2, 9.4.2, 9.6, 9.7, 9.8
  - Ch 9 also covers other technologies, including Servlets, JSP, PHP, Django, Hibernate.

Also

- Kifer/Bernstein/Lewis(complete version, 2ed)
  - Chapter 8.1, 8.2, 8.4, 26.7
  - Ch 8 also covers other technologies, including JDBC (quite similar to DB-API), SQLJ
- Ramakrishnam/Gehrke(3ed)
  - Chapter 6.1, 7.5, 7.7, case study in 7.8
  - Chs 6,7 also cover other technologies, including JDBC, SQLJ
- Garcia-Molina/Ullman/Widom(complete book, 2ed)
  - Chapter 9.1-9.5
  - Ch 9 also covers JDBC and PHP

Database Documentation:

- Python DB-API: http://initd.org/psycopg/docs/
  https://wiki.python.org/moin/UsingDbApiWithPostgres
  http://www.tutorialspoint.com/postgresql/postgresql_python.htm