ISYS2120: Data & Information Management

Week 8: Database Application Development

Alan Fekete

Based on material from Roehm, Khushi

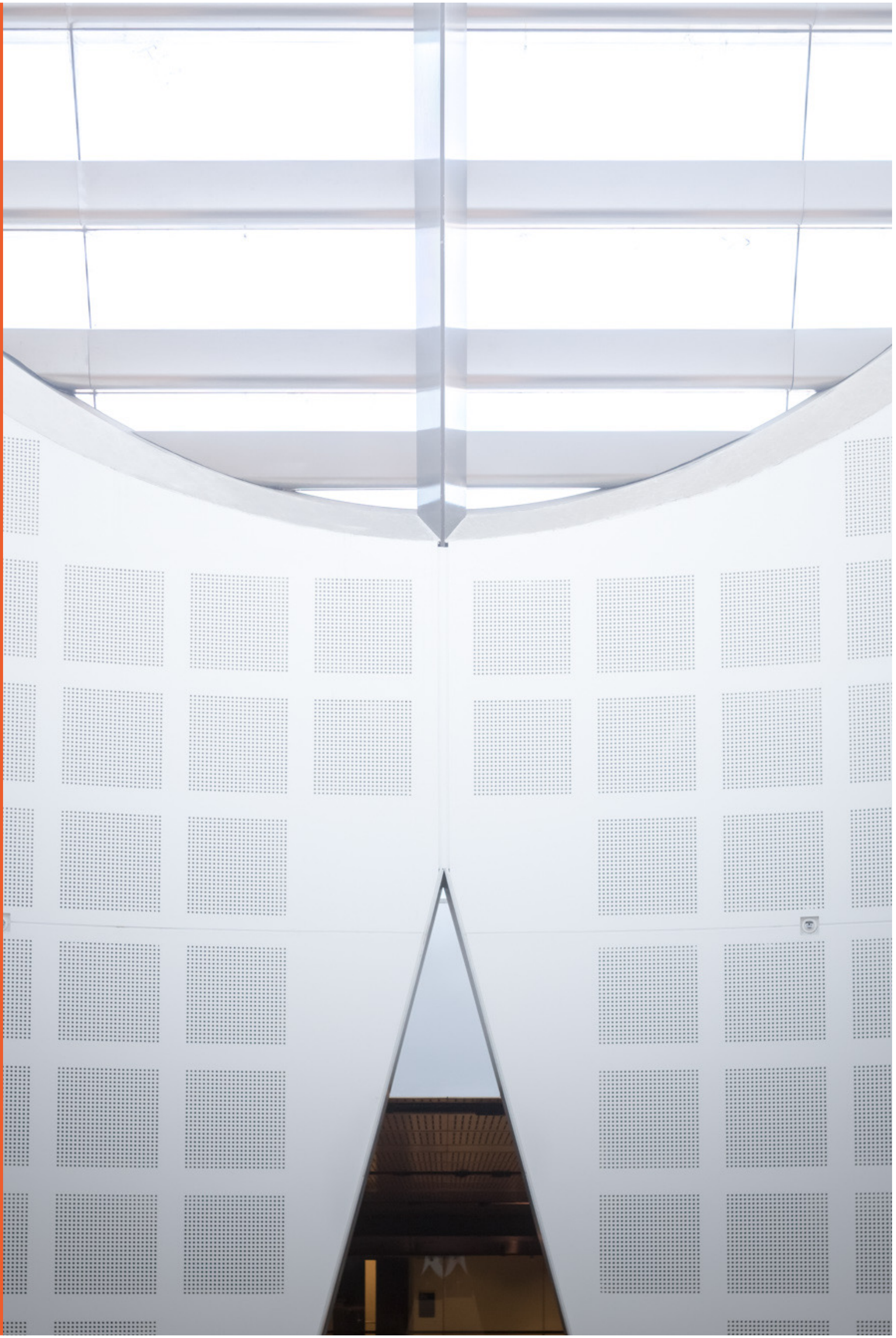
Cf. Kifer/Bernstein/Lewis – Chapter 8

Ramakrishnan/Gehrke – Chapter 6

Silberschatz/Korth/Sudarshan – Chapter 9

Ullman/Widom – Chapter 9

THE UNIVERSITY OF SYDNEY



Database-backed software

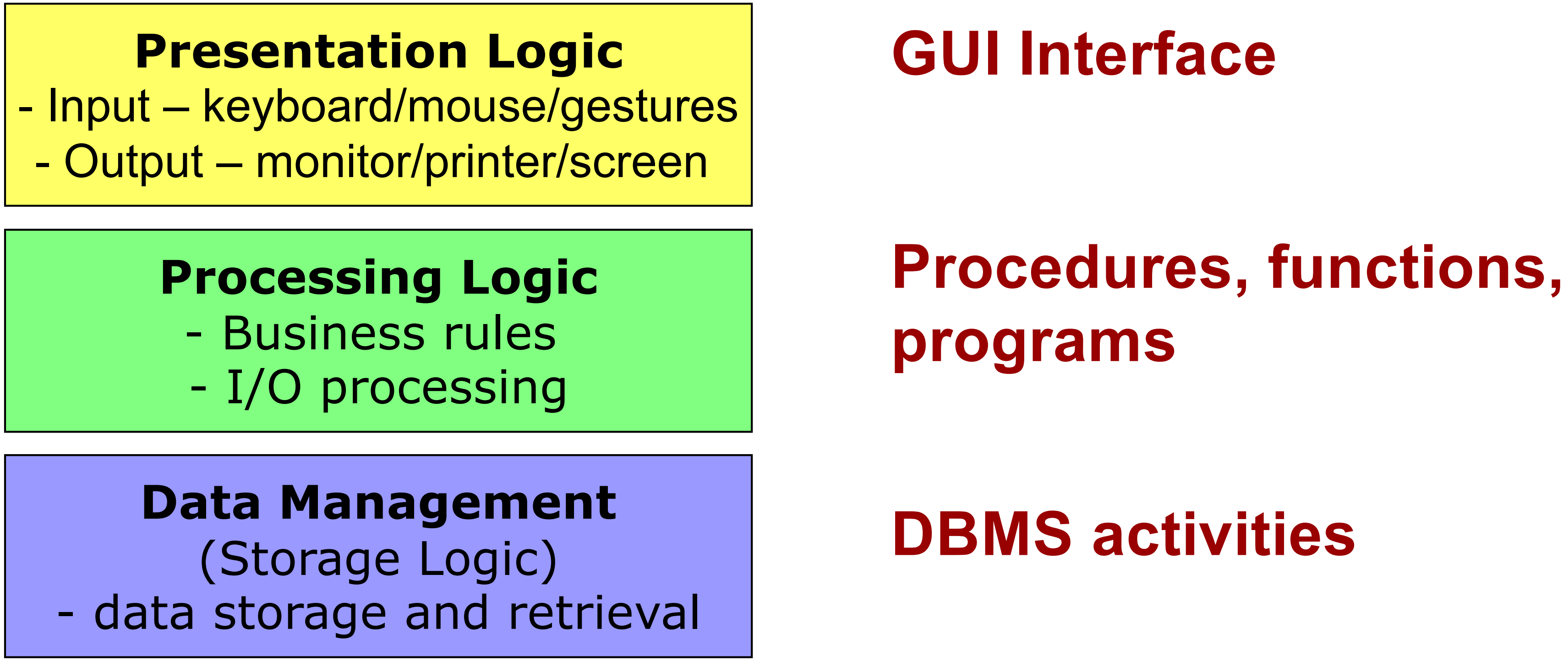
- Most of the software used for life and business needs to use data that is found in a database
  - Often, user activities will also modify the data in the database
- A few examples
  - E-commerce (inventory, purchase, order status, order history)
  - Entertainment (catalogue, preferences, history)
  - 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 inoformation 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):



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

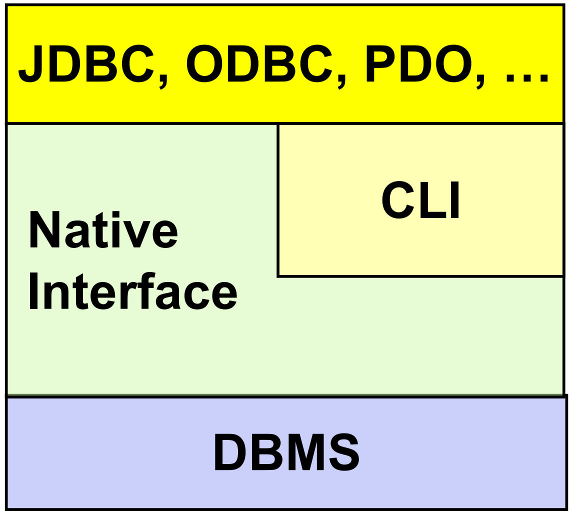
- Often, web browser is used as GUI for end-user
  - Available on all devices!
- Application needs to have code that lays out the web pages and provides navigation between them
- Mobile devices may provide an app that directly works with the gestures etc of the device
  - And is targeted for the small 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
    - Compiler must deal with both aspects
  - **Call-level interface** (CLI)
    - Create special API to call SQL commands (JDBC, ODBC, Python, …)
    - SQL statements are passed as arguments to host language (library) procedures / APIs

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



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 (PostgreSQL) or cx_oracle (Oracle)
    - **http://initd.org/psycopg/docs/**
  - For dynamic websites:
    - several framework available; we will use Flask..
    - Allows to define template pages 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:
  https://wiki.python.org/moin/DatabaseProgramming
  https://wiki.python.org/moin/UsingDbApiWithPostgres
  http://initd.org/psycopg/docs
  http://www.tutorialspoint.com/postgresql/postgresql_python.htm
  https://www.python.org/dev/peps/pep-0249/

Python DB-API Example

```
import psycopg2

try:
    # connect to the database
    conn = psycopg2.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 = psycopg2.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 = psycopg2.connect(
"host=postgres.usyd.edu.au dbname=unidb user=U password=secret")
```

connectionParameters *db login*

Details: http://initd.org/psycopg/docs/module.html#psycopg2.connect

Python Database Connection Modules

- Python support for variety of DBMSs
  - MySQL (module: MySQLdb)
  - PostgreSQL (module: 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…

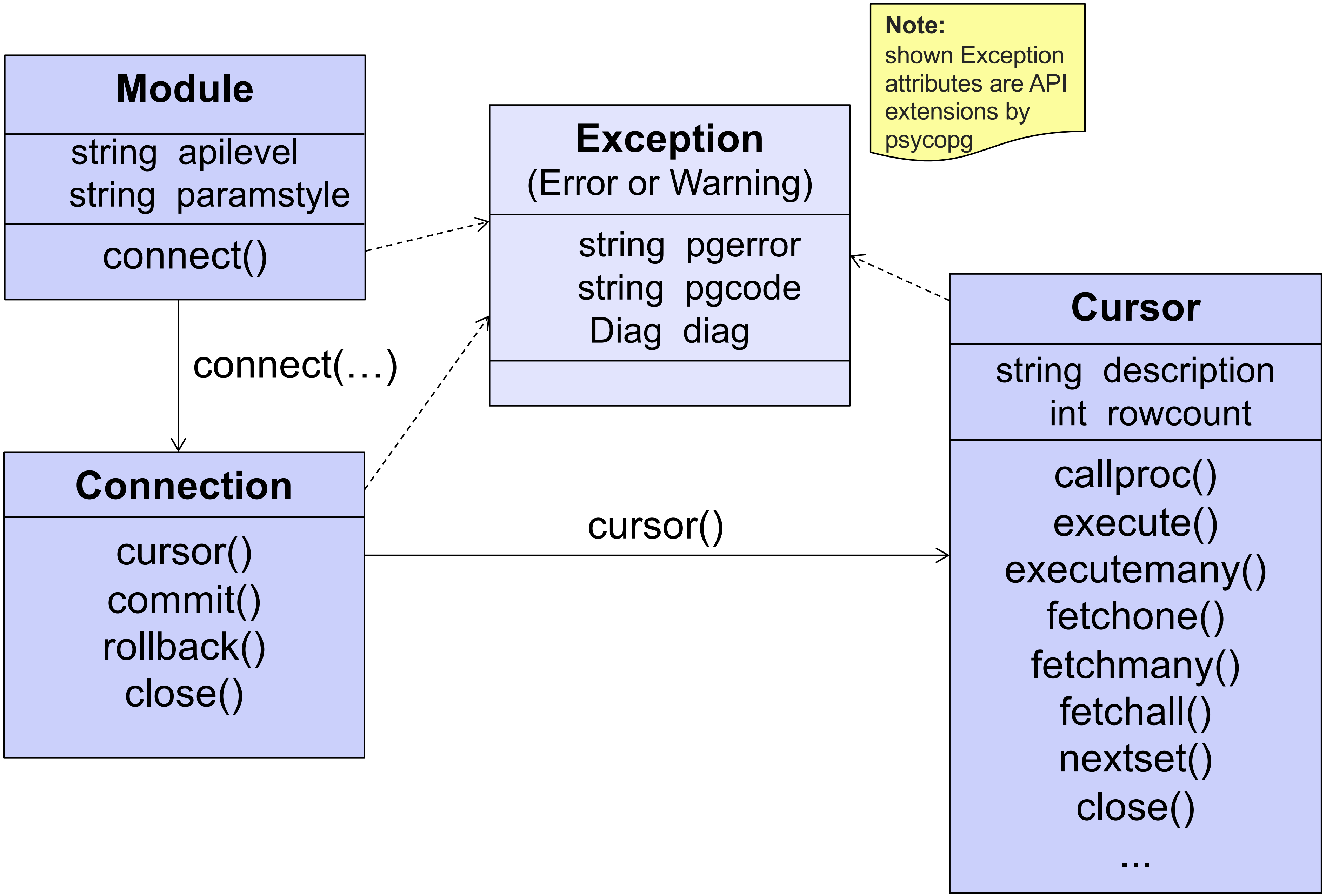
- Example for Oracle:

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

psycopg Connection Simple Example

```
import psycopg2

# connect to the database
try:
    conn = psycopg2.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
  - **psycopg2.connect()** connects to a database
  - **conn.cursor()** creates a cursor object for query execution
- Start SQL statements
  - **execute()** for static SQL, and also parameterized SQL queries
  - **callproc()** for executing a stored procedure including parameters
- Result retrieval
  - **fetchone()** retrieves next row of a result or **None** when no more data
  - **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
  - … but **not** 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 constructs 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/psycopg2 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
  - ***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 some static SQL query:

```
import psycopg2

try:
    # connect to the database
    conn = psycopg2.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 psycopg2

try:
    # connect to the database
    conn = psycopg2.connect(database='foo', user='dbuser', password='pwd')

    # prepare list of insert values (3 students enrolling in INFO2120)
    params = [(4711, INFO2120'), (4712,'INFO2120'), (4713,'INFO2120') ]

    # 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")
```

WARNING:
Never Query by "String-Concatenation"

Never, never, NEVER ever use Python string concatenation (+) or string parameter interpolation (%) to place variables into a SQL query string!

=> otherwise your program is vulnerable to SQL Injection attacks

```
query ="""SELECT  E.studId  FROM  Enrolled E
          WHERE  E.uosCode = """ + uosCode + \
          "  AND E.semester = " + semester

cursor.execute( query )
```

string concatenation

- *simple approach to construct a variable query*
- *concatenates query from different string-parts*
- *executes the constructed SQL string directly in DBMS*
- ***Warning**: prone to errors and even hacking when input strings allowed to be entered by a user*

SQL Code Injection Vulnerability

- **SQL-Injection**
  to infiltrate a SQL database with own SQL commands.
  - Can be used to execute SQL statements with elevated privileges or to impersonate another user.

- Without direct database connection (e.g. web application)
  - Injecting SQL via un-checked user input.
  - Exploiting buffer overflows.
    - Oracle standard packages have many buffer overflows.
  - Output on attacker's screen.
- With a direct database connection
  - SQL Injection in built-in or user-defined procedures.
  - Buffer overflows in built-in or user-defined procedures.
    - Risk when a procedure is not defined with the AUTHID CURRENT_USER keyword (executes with the privileges of the owner

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,) )
```

parameter marker

This comma is no mistake, but needed with single parameters

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 psycopg in execute():

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

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

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

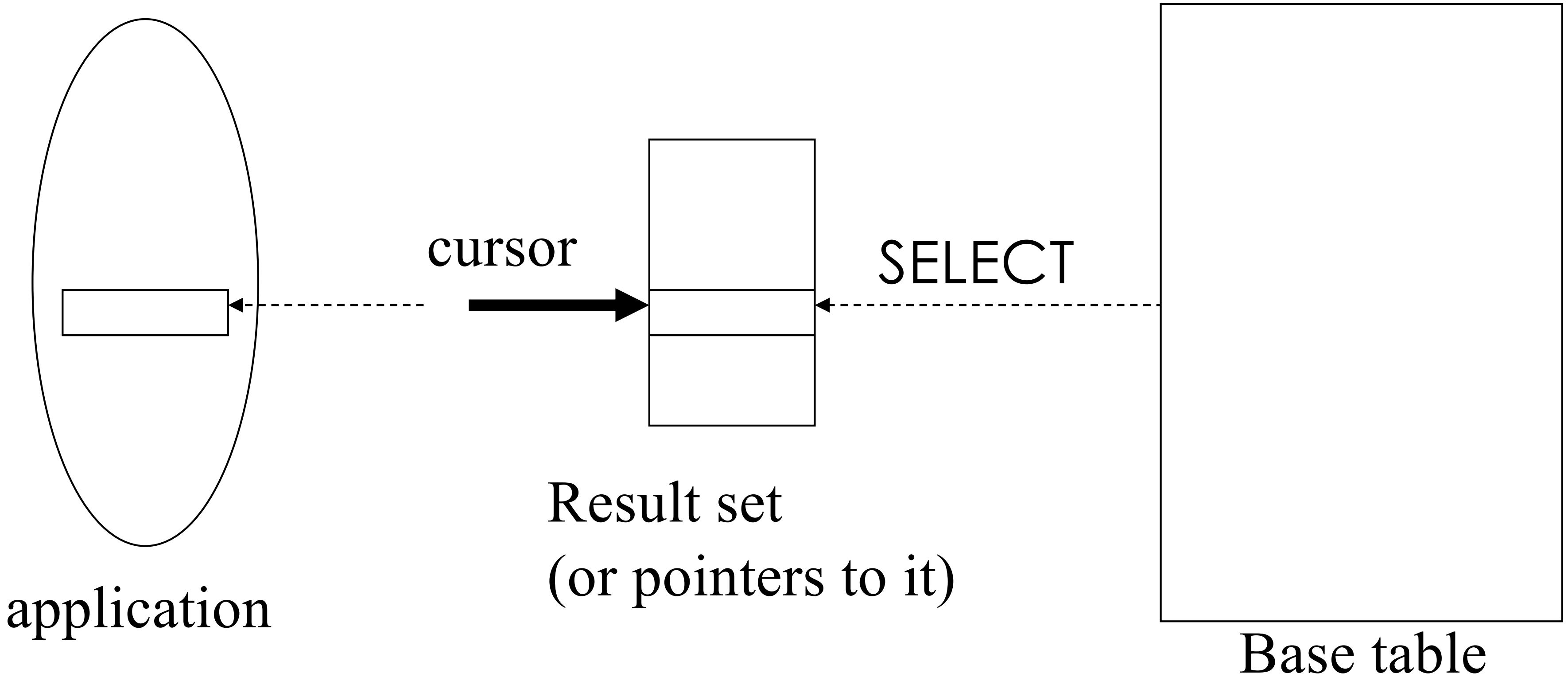
Variables shared by host and SQL

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: Impedance Mismatch)

- **Problem**: SQL deals with tables (of arbitrary size); host language program deals with fixed size buffers
  - How is the application to allocate storage for the result of a SELECT statement?
- **Solution**: Cursor concept
  - Fetch a single row at a time



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, psycopg returns tuples with fetch() / fetchall()
  - fields can only be addressed positionally
- As an extension, psycopg 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 psycopg2
from   psycopg2.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)
cusr.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:

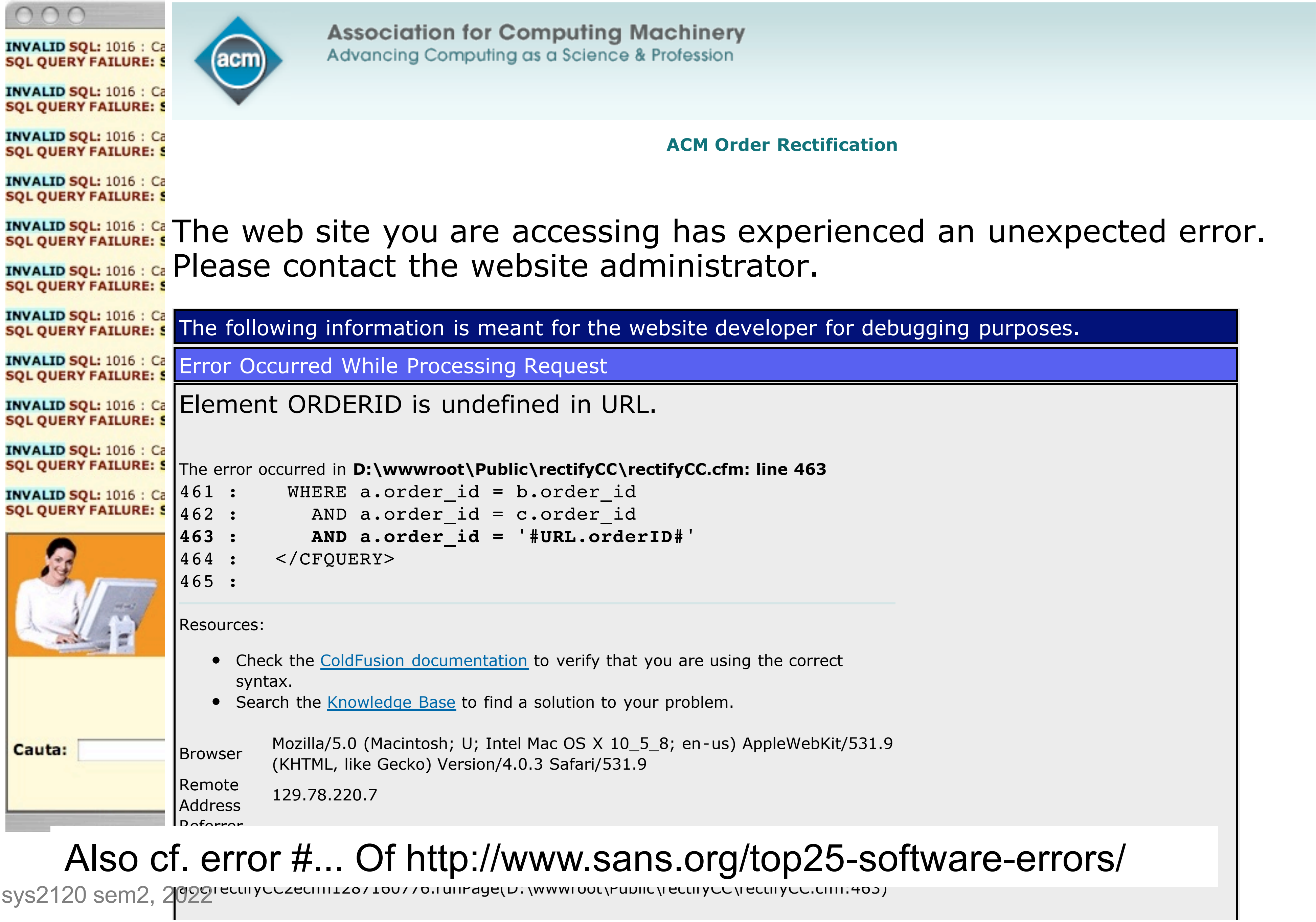
```
# Ensure variable is defined
try:
    x
except NameError:
    x = None

# 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…

You should avoid this!



INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

INVALID SQL: 1016 : Ca
SQL QUERY FAILURE: S

Association for Computing Machinery
Advancing Computing as a Science & Profession

ACM Order Rectification

The web site you are accessing has experienced an unexpected error.
Please contact the website administrator.

The following information is meant for the website developer for debugging purposes.

Error Occurred While Processing Request

Element ORDERID is undefined in URL.

The error occurred in **D:\wwwroot\Public\rectifyCC\rectifyCC.cfm: line 463**

```
461 :     WHERE a.order_id = b.order_id
462 :       AND a.order_id = c.order_id
463 :       AND a.order_id = '#URL.orderID#'
464 :    </CFQUERY>
465 :
```

Resources:

- Check the ColdFusion documentation to verify that you are using the correct syntax.
- Search the Knowledge Base to find a solution to your problem.

Browser	Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_8; en-us) AppleWebKit/531.9 (KHTML, like Gecko) Version/4.0.3 Safari/531.9
Remote Address	129.78.220.7
Referrer	

```
rectifyCC2ecfm1287160776.runPage(D:\wwwroot\Public\rectifyCC\rectifyCC.cfm:463)
```

Also cf. error #... Of http://www.sans.org/top25-software-errors/

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 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:
   psycopg2.connect(…)
except psycopg2.Error as e:
   print("Problem connecting to database:")
   print(e.pgerror)
   print(e.pgcode)
```

Demo purpose only. As said before:
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
        |   |__ psycopg2.extensions.QueryCancelError
        |   |__ psycopg2.extensions.TransactionRollbackError
        |__ IntegrityError
        |__ InternalError
        |__ ProgrammingError
        |__ NotSupportedError
```

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

Reprise: Example of Python DB-API

```
import psycopg2

try:
    # connect to the database
    conn = psycopg2.connect(database="postgres",user="test",password="secret")

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

    # execute a parameterised query
    unit_of_study = "INFO2120"
    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 psycopg2.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

Protecting a Web Database

- Be careful to check all parameters which can end up in such SQL statements!
  - Never trust user provided data!
- Use dynamic SQL statements with explicit, type-checked parameters (execute() function with parameter markers)
- Restrict the privileges of the user/role of the web application
  - E.g. with Oracle: Revoke EXECUTE privilege on Oracle standard packages when not needed. Specially for the PUBLIC role.
- Patch, patch, patch ;-)
- Also: NEVER directly return database error messages
  - Not very user-friendly AND it gives attackers hints

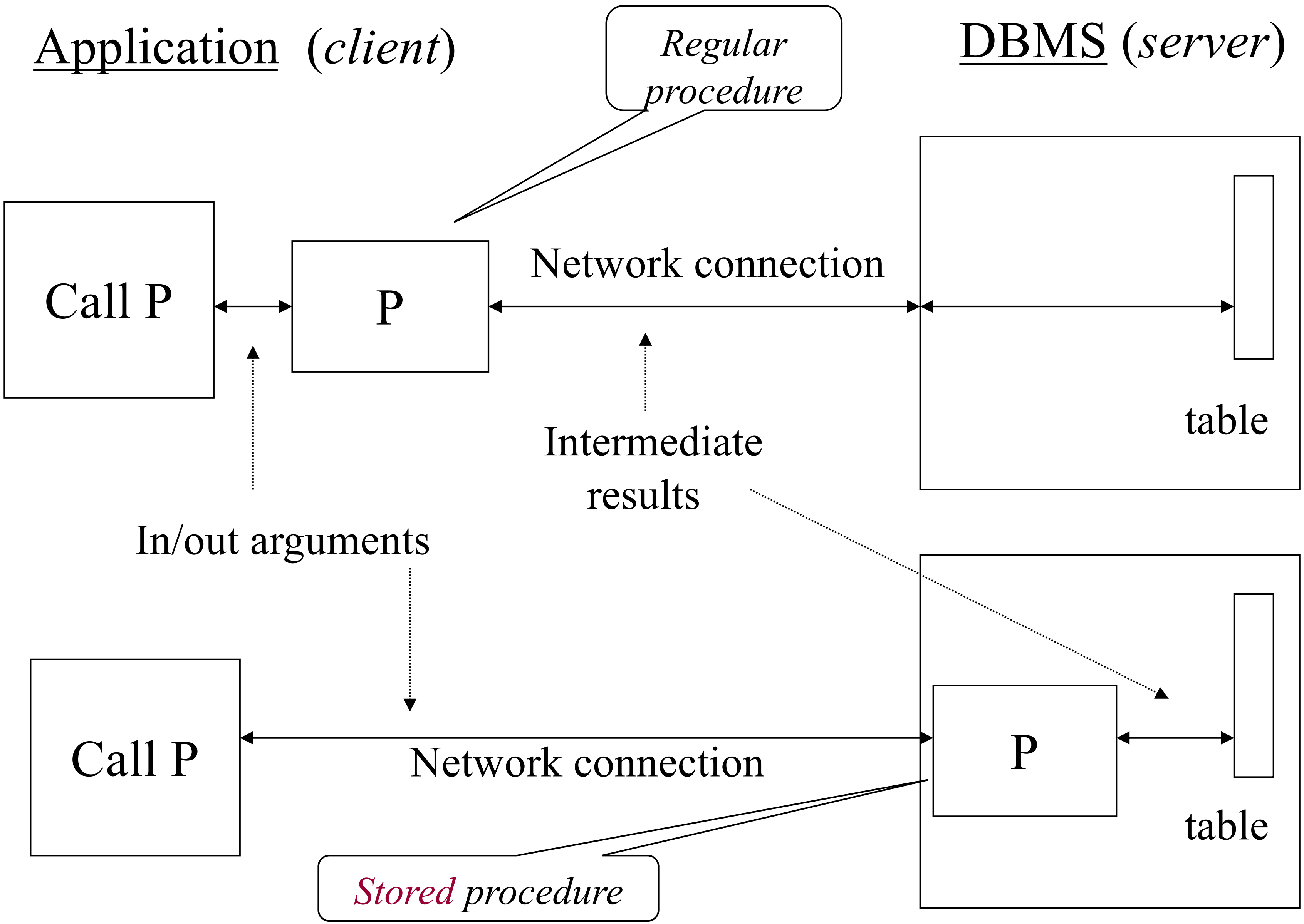
Stored Procedures (Server-side application logic)

- 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 named procedures, can also be functions

Stored Procedures



SQL/PSM

- Stored Procedures not only have full access to SQL
- All major database systems provide extensions of SQL to a simple, general purpose language
  - SQL:1999 Standard: SQL/PSM
  - PostgreSQL: PL/pgSQL
  - Oracle: PL/SQL (syntax differsfrom PostgreSQL!!)
  - Microsoft SQL Server: T-SQL
- Extensions
  - Local variables, loops, if-then-else conditions
- Calling Stored Procedures: CALL statement
  - Example: `CALL ShowNumberOfEnrolments();`

Procedure Declarations

- Procedure Declarations (with SQL/PSM)

```
CREATE FUNCTION name ( parameter1,…, parameterN ) AS
  local variable declarations
   procedure code;
```

- Stored Procedures can have parameters
  - of a valid SQL type (parameter types must match)
  - three different modes
    - IN arguments to procedure
    - OUT return values
    - INOUT combination of IN and OUT

```
CREATE FUNCTION CountEnrolments( IN uos VARCHAR ) AS
     SELECT COUNT(*)
       FROM Enrolled
      WHERE uosCode = uos;

 CALL CountEnrolments ('INFO2120');
```

PostgreSQL: PL/pgSQL

(cf. http://www.postgresql.org/docs/9.4/static/plpgsql.html)

- Extents SQL by programming language contructs
  - Only knows functions! **CREATE FUNCTION** *name* ***RETURNS*** *...* ***AS***...
  - Compound statements: **BEGIN** … **END**;
  - SQL variables: **DECLARE section**
    variable-name sql-type;
  - Assignments: *variable* **:=** *expression*;
  - IF statement: **IF** *condition* **THEN** … **ELSE** … **END IF**;
  - Loop statements: **FOR** *var* **IN** *range* (**WHILE** *cond*)
    LOOP … **END LOOP**;
  - Return values: **RETURN** *expression*;
  - Call statement: **CALL** procedure(parameters);
  - Transactions: **COMMIT**; **ROLLBACK**;

PL/pgSQL Example

(cf. http://www.postgresql.org/docs/9.4/static/plpgsql-structure.html)

Tip: CREATE OR REPLACE to avoid 'name-already-used'

- PL/pgSQL procedure declaration

```
CREATE OR REPLACE FUNCTION
    name ( parameter1, …, parameterN ) RETURNS sqlType
AS $$
DECLARE
  variable    sqlType;
  …
BEGIN
     …
END;
$$ LANGUAGE plpgsql;
```

optional

Tip: final delimiter must match the one used after AS

- where parameterX is declared as (IN is default):

  [IN|OUT|IN OUT] ***name sqlType***

PostgreSQL PL/pgSQL Example

```
CREATE OR REPLACE FUNCTION RateStudent
   (studId INTEGER, uos VARCHAR) RETURNS CHAR AS $$
  DECLARE
       grade  CHAR;
       marks  INTEGER;
  BEGIN
       SELECT SUM(marks) INTO marks
         FROM Assessment
        WHERE sid=$1 AND uosCode=$2;
       IF    ( marks>84 ) THEN grade := ‘HD’;
       ELSIF ( marks>74 ) THEN grade := ‘D’;
       ELSIF ( marks>64 ) THEN grade := ‘CR’;
       ELSIF ( marks>50 ) THEN grade := ‘P’;
       ELSE                    grade := ‘F’;
       END IF;
       RAISE NOTICE 'Final grade is: %s', grade;
       RETURN grade;
  END;
$$ LANGUAGE plpgsql;
```

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 psycopg2
conn = psycopg2.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 VM is in a different process from dbms, then performance often suffers

Lessons Learned

- **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: security for parametrised queries*
- **Database APIs**
  - After lab08, you should in particular be able to work with small Python db programs
- **Server-side database programming**
  - How to use stored procedures to run code inside a DBMS
    - e.g. with PostgreSQL's pl/pgsql or with Oracle's PL/SQL
  - Modern database engines provide virtual machine environments to run external code near the data

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 8
- Ramakrishnan/Gehrke (3rd edition - the ‘Cow’ book)
  - Chapter 6
- Ullman/Widom (3rd edition of ‘First Course in Database Systems’)
  - Chapter 9 (*covers Stored Procedures, ESQL, CLI, JDBC and PHP)*
- Silberschatz/Korth/Sudarshan – Chapter 9

Database Documentation:

- Python DB-API: http://initd.org/psycopg/docs/
https://wiki.python.org/moin/UsingDbApiWithPostgres
http://www.tutorialspoint.com/postgresql/postgresql_python.htm
- The PostgreSQL Global Development Group: “PostgreSQL 8.2.4 Documentation”, 2009.
- Oracle Corporation: “Oracle 10.1 Database Concepts”,2003.

- MySQL website: http://www.mysql.com