Module 3 : Python Database Integration

Foundation Program 4.1

Copyright Guideline

Confidential Information

Course Information

Course Code: **FP4.1-M3**

Course Name: **Python Database Integration**

Document Number: **003**

Version Number: **1.0**

Pre-requisites

- Recap on Basic concepts in Python
- Object Oriented Programming in Python
- Database fundamentals using Oracle

Recap of Module-1 and Module-2

In module-1 (OOP using Python), we have learnt:

- Introduction to Programming
- Python Fundamentals
- Control Structures
- Data Structures
- Functions
- File Handling
- Exception Handling
- Modules and Packages
- Object Oriented Concept using Python

In module-2 (Relational Database Management System-RDBMS), we have learnt:

- Database Basics : Need for Databases and Relational Model, Keys
- Database Design : Database Development Life Cycle
  - Understanding Data Requirements
  - Logical Database Design
  - Physical Database Design
- Implementation using SQL
  - DDL, DML, DCL and TCL statements
  - Single Row and Aggregate Functions
  - Group By and Having
  - Joins, Sub Queries, Indexes, Views, Transactions
  - PL/SQL
    - Fundamentals of PL/SQL, Cursors and Stored Procedures

Learning Objectives

On completion of this module, the learner should be able to:

1) Understand the need for database connectivity from Python

2) Install cx_Oracle for connecting to Oracle database from Python program

3) Use Select operation to retrieve data from Oracle tables

4) Implement Positional and Named bind variables for writing dynamic queries

5) Perform Create, Insert, Update and Delete operations

6) Handle database related exceptions in Python

Table of Contents

Why Database Programming

A Scenario

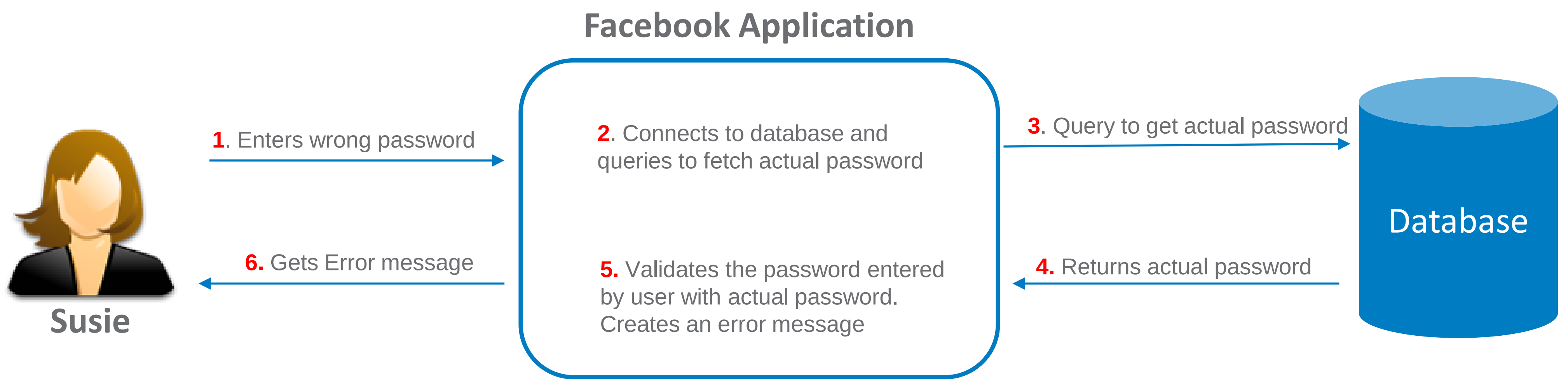
Susie is trying to access Facebook to convey birthday wishes to her friend Julie. Mistakenly she enters wrong password. Immediately she was intimated by Facebook that the password she has entered is not correct.

- Why do you think this happened?
- Where are the username and password for Facebook login stored?
- How does Facebook know that Susie entered a wrong password?

Discuss.

What actually happens?

- There is some database at the back end of Facebook application where the login credentials for Susie are stored.
- Susie interacts with the Facebook application through a browser which internally connects to a database. The application validates the credentials entered by Susie with the actual credentials stored in the database.
- If they do not match, Facebook returns with an error message informing Susie that she entered a wrong password. If they match, Susie is able to login to her Facebook account successfully.


Facebook Application
1. Enters wrong password
2. Connects to database and queries to fetch actual password
3. Query to get actual password
Database
4. Returns actual password
5. Validates the password entered by user with actual password. Creates an error message
6. Gets Error message
Susie

Need for Database Programming

Interaction between Applications (Retrieving & Storing data)

Any Programming languages which provide database connectivity features.

In this module, we will learn how to connect to Oracle Database from Python Application and execute various queries.....

Python Oracle Integration- Pre-requisites and Installation

Python Modules for Database Connectivity

- Different Python Modules are available for connecting to various databases. They are given in the table below:

Database Name	Python Module
Oracle	**cx_Oracle**
DB2	Pydb2
MySQL	MySQLdb
Microsoft SQL Server	Adodbapi
Microsoft Access	PyPyODBC
Teradata	mxODBC

- In this module, we will be connecting to Oracle Database from Python, hence we will use **cx_Oracle** module.

Pre-requisites

Python Installation and PyDev setup

- Download and Install Python 3.5: https://www.python.org/downloads/
- Download PyDev_3.8.0 or higher version: http://www.pydev.org/download.html
- Download Eclipse Juno or higher version: https://www.eclipse.org/downloads/index.php
- Install Python on your machine
- Copy paste the contents of the plugin folder of PyDev into plugin folder of Eclipse
- In Eclipse open PyDev perspective (Window -> Open Perspective -> Other -> PyDev)
- Create a PyDev Project
- Select Grammar as 3.0
- Configure interpreter by choosing the .exe file of Python installed in your machine

Oracle Installation

- Download Oracle 11g Express Edition http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html
- Note: Please refer to Module 1 and Module 2 for further details on Python and Oracle Installation .

cx_Oracle Installation

- Download and install the relevant version from the following link:

https://pypi.python.org/pypi/cx_Oracle/5.2.1

- You can test successful installation of cx_Oracle by executing following command on Python shell:

```
>>> import cx_Oracle
>>>
```

- *No output* confirms successful installation. Otherwise, error message will be returned.

Connect Database
Create and Insert operations

Easy Shop Retail Application…

- Easy shop want to store their Suppliers, Items and Quotations details in a database application and manipulate information's stored in database.

Suppliers	Supplier ID	Supplier Name	Supplier Contact No	Supplier Email ID

Item	Item Code	Item Type	Description	Price	Reorder Level	Quantity on hand	Category

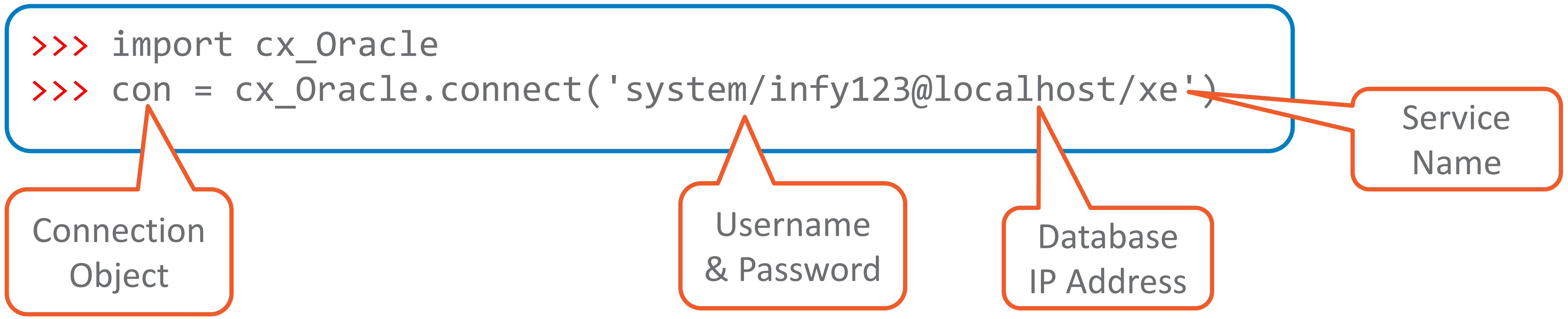
Quotation	Quotation ID	Supplier ID	Item Code	Quoted Price	Quotation Date	Quotation Status

Connect to Oracle Database

- After successful installation of cx_Oracle, we need to establish a connection with Oracle database from Python program.
- **connect()** function helps to connect to Oracle database from Python and returns the connection object.
- **Syntax:**

```
<<connection_object>> =
cx_Oracle.connect("<<username>>/<<password>>@<<database_name>>/<<service_name>>")
```

- **Example:**



```
>>> import cx_Oracle
>>> con = cx_Oracle.connect('system/infy123@localhost/xe')
```

Closing Database Connection

- Connected Database must be closed at the end of the program using **close()** function.
- **Syntax:**

```
<<connection_object>>.close()
```

- **Example:**

```
>>> import cx_Oracle
>>> cx_Oracle.connect('system/infy123@localhost/xe')
>>> # Lines of code goes here
>>>
>>>
>>> con.close()
```

Close Database Connection

Commit() Method

- Commit() used to save the changes made in database otherwise the changes will not be reflected in database.

- **Syntax:**

```
<<connection_object>>.commit()
```

- **Example:**

```
>>> import cx_Oracle
>>> cx_Oracle.connect('system/infy123@localhost/xe')
>>> # Lines of code goes here
>>>
>>> con.commit()
>>> con.close()
```

Saves the changes in Database

Rollback() Method

- We used commit() function to commit the changes to database.
- rollback() function of connection class can be used, if we do not want to commit the changes made to database. It reverses the effect of all the statements after the last commit.
- Please note that changes once committed can not be rolled back.
- **Syntax:**

```
<<connection_object>>.rollback()
```

- **Example:**

```
>>> import cx_Oracle
>>> cx_Oracle.connect('system/infy123@localhost/xe')
>>> # Lines of code goes here
>>>
>>> con.rollback()
>>> con.close()
```

Rollback the changes in Database

Cursor for Database Operations

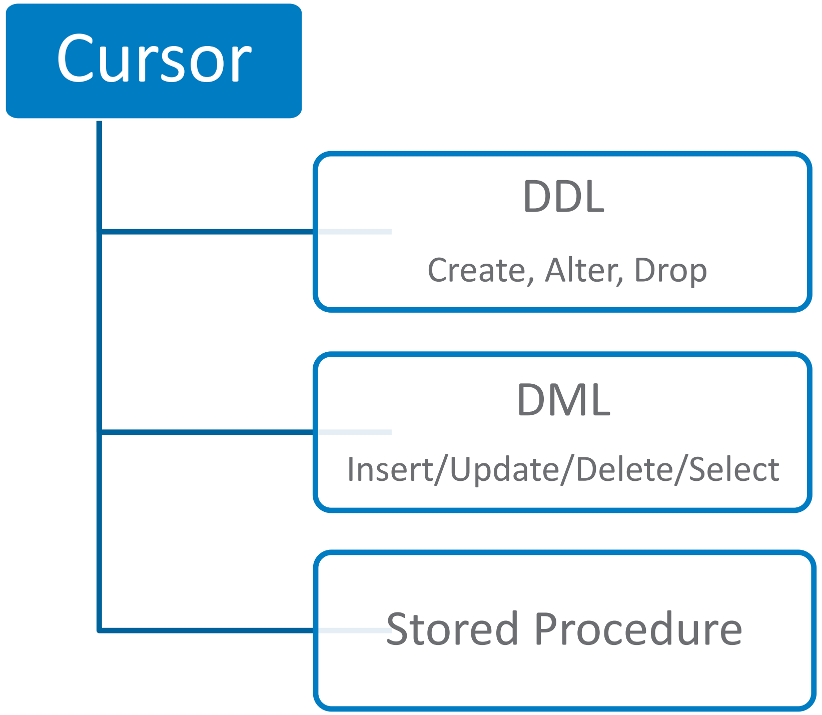
- A cursor holds the rows returned by SQL statement. It helps us to traverse over the records returned.
- We need a Cursor object to do all database operations.
- **cursor()** function is used to create cursor.
- **Syntax:**

```
<<cursor_object>> = <<connection_object>>.cursor()
```

- **Example:**

```
>>> import cx_Oracle
>>> con = cx_Oracle.connect("system/infy123@localhost/xe")
>>> cur = con.cursor()
>>> # Lines of code goes here

>>> con.close()
```

Cursor Object



Execute() Method

- **Execute()** methods is used to execute the SQL Commands.

- **Syntax:**

```
<<cursor_object>>. execute(Query, Bind Variables)
```

- **Example:**

```
>>> import cx_Oracle
>>> con = cx_Oracle.connect("system/infy123@localhost/xe")
>>> cur = con.cursor()
>>> cur.execute(“SQL Query”)
>>> # Lines of code goes here
>>> con.close()
```

Executes the query

Consider a Table

- Consider a table "**Supplier**" in Oracle database. Structure and sample data for this table is given below.
- **Table Structure:**

Column Name	Type	Size	Description
supplierid	Varchar2	10	Primary key
suppliername	Varchar2	30	Cannot be null
suppliercontactno	Varchar2	30	Cannot be null
emailid	Varchar2	30	

Demo Code for CREATE Query:

Execute() – Create Query

```
import cx_Oracle

con=cx_Oracle.connect(" system/infy@XE ")

cur=con.cursor()

cur.execute("""CREATE TABLE supplier(

            supplierid VARCHAR2(10) PRIMARY KEY,

            suppliername VARCHAR2(30) NOT NULL,

            suppliercontactno VARCHAR2(30) NOT NULL,

            supplieremailid VARCHAR2(30)

            )

                    """)

con.commit()

con.close()
```

Note: The triple quotes """ or ''' are used so that the SQL query can be visually organized for clarity. With triple quotes (three consecutive single or double quotes) lines of code can be broken into several lines which is not possible with single or double quotes.

Consider the Supplier Table

- **Sample Data:**

supplierid	suppliername	suppliercontactno	emailid
S1001	Giant Store	203-237-2079	rachell@easy.com
S1002	EBATs	115-340-2345	ebats@easy.com
S1003	Shop Zilla	203-123-3456	shopzilla@easy.com
S1004	VV Electronics	115-340-6756	vvelectronics@easy.com

Note: We would refer to this table for all further examples.

Demo Code for INSERT Query:

Execute() – Insert Query

```
import cx_Oracle

con=cx_Oracle.connect(" system/infy@XE ")

cur=con.cursor()

cur.execute("""INSERT INTO supplier VALUES ('S1001','Giant Store','203-237-2079', 'rachel1@easy.com')""")

cur.execute("""INSERT INTO supplier VALUES ('S1002','EBATs','115-340-2345','ebats@easy.com')""")

cur.execute("""INSERT INTO supplier VALUES ('S1003','Shop Zilla','203-123-3456', 'shopzilla@easy.com')""")

cur.execute("""INSERT INTO supplier VALUES ('S1004','VV Electronics','115-340-6756', 'vvelectronics@easy.com')""")

con.commit()

con.close()
```

Inserting four row in Supplier table

Save changes to Database

Guided Activity : Assignment 1: Database Connectivity

Hands-On : Assignment 2: Database Connectivity

Retrieving Data from Database

Retrieve Data from Database

- Once the connection with database is established, we can use Select queries to retrieve the data from database tables.
- All the records fetched by the query will be stored in cursor_object.

- **Fetching Methods to retrieve the records from database:**

Fetchall()	Fetches all the remaining rows from the result-set and returns as a *list* of *tuples*. If no rows remaining in the result-set, it returns an empty *list*.

**Fetchmany(x)
X=arraysize**
	Fetches x rows out of total number of remaining row.
	If x value > count of remaining number of rows, returns all the remaining rows
	If invoked without any args, it returns arraysize number of rows. Default is 50

Fetchone()	Fetches single row from result-set and returns it as a tuple. If no more rows are available, it returns None

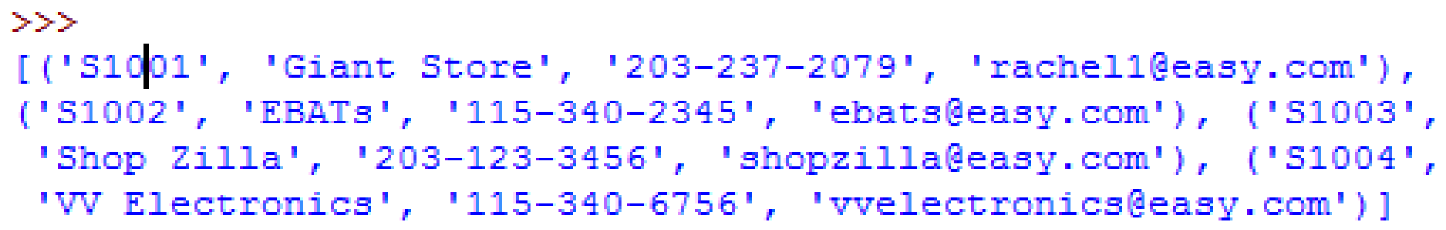
Demo Code for SELECT Query with FETCHALL():

Fetchall() - Fetch Records

```
import cx_Oracle

con=cx_Oracle.connect("system/infy@XE")

cur=con.cursor()

cur.execute("""Select * from supplier""")

print (cur.fetchall())

con.commit()

con.close()
```

Fetches all the records from table

Output



```
>>>
[('S1001', 'Giant Store', '203-237-2079', 'rachel1@easy.com'),
('S1002', 'EBATs', '115-340-2345', 'ebats@easy.com'), ('S1003',
 'Shop Zilla', '203-123-3456', 'shopzilla@easy.com'), ('S1004',
 'VV Electronics', '115-340-6756', 'vvelectronics@easy.com')]
```

Select Operation

- In the previous example we have fetched the details of all the suppliers from Supplier table.
- Suppose, we want to fetch the details of a particular supplier (say supplierid = 'S1001').
- How do we implement this?
- **Example:**

```
>>> import cx_Oracle
>>> con = cx_Oracle.connect("system/infy@XE")
>>> cur = con.cursor()
>>> cur.execute("SELECT * FROM supplier WHERE supplierid = 'S1001'")
>>> print(cur.fetchall())
>>> con.close()
```

Fetching the records based on a criteria

Output

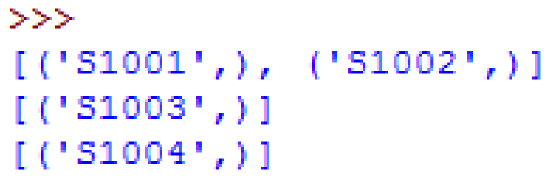
```
>>>
[('S1001', 'Giant Store', '203-237-2079', 'rachel1@easy.com')]
```

Demo Code for SELECT Query with FETCHMANY():

Fetchmany() – Fetch n Records from resultset:

```
import cx_Oracle
con=cx_Oracle.connect("system/infy@XE")
cur=con.cursor()
cur.execute("""Select supplierid from supplier""")
print (cur.fetchmany(2))        # Fetches first 2 records from table
print (cur.fetchmany(1))        # Fetches next 1 records from table. i.e. third record
print (cur.fetchmany(2))        # Fetches next 1 records from table, as only one record left
con.commit()
con.close()
```

Output



```
>>>
[('S1001',), ('S1002',)]
[('S1003',)]
[('S1004',)]
```

Demo Code for SELECT Query with FETCHONE():

Fetchone() – Fetch one record from resultset:

```
import cx_Oracle
con=cx_Oracle.connect("system/infy@XE")
cur=con.cursor()
cur.execute("""Select supplierid,suppliername from supplier""")
print (cur.fetchone())
print (cur.fetchone())
print (cur.fetchone())
con.commit()
con.close()
```

Fetches only one record at a time

Output

```
>>>
('S1001', 'Giant Store')
('S1002', 'EBATs')
('S1003', 'Shop Zilla')
```

Guided Activity : Assignment 3: Database: Retrieve Data

Hands-On : Assignment 4: Database: Retrieve Data

Parameters Passing using Bind Variables

Bind Variables

- Till now we have been passing explicit values to Oracle queries.
- We have already seen an example to fetch the record from Supplier table for supplier id = 'S1001'
- Now, consider a scenario where we want to fetch the details of a particular supplier based on supplier id but, the value of supplier id should be taken from the user, through Python application at execution time.
- In this case, we will not be able to pass static values to the query.
- Here, we can pass the values to query dynamically through bind variables which can be mapped to Python variables.

Methods of Passing Bind Variables

- Two ways of using bind variables:
  - **Positional** : Passing the parameter in the same order
  - **Named** : Passing the parameter by calling its name.

```
Execute(query(:bindVariable1,:bindVariable2), (value1, value2))
```

or

```
execute(query(:bindVariable1, :bindVariable2), {bindVariable1 ', value1, ' bindVariable2, value2})
```

Note: value1 and value2 can be Python Variable names as well.

Bind Variables - Positional

- **Example:**

```
import cx_Oracle
con = cx_Oracle.connect("system/infy@XE")
cur = con.cursor()

supplier_ID = 'S1001'
supplier_name = 'Giant Store'

cur.execute("""SELECT * FROM supplier WHERE supplierid = :param1 AND
supplername = :param2""", (supplier_ID, supplier_name))

con.close()
```

Values can also be taken from user

param1 and param2 are bind variables

supplier_ID and supplier_name are Python variables

supplierid and suppliername are database column names

Output

```
>>>
[('S1001', 'Giant Store', '203-237-2079', 'rachel1@easy.com')]
```

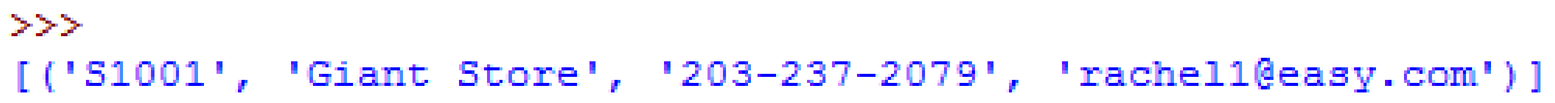
Bind Variables - Named

- **Example:**

```
import cx_Oracle
con = cx_Oracle.connect("system/infy@XE")
cur = con.cursor()
supplier_ID = 'S1001'
supplier_name = 'Giant Store'

cur.execute(""" SELECT * FROM supplier WHERE supplierid = :param1 AND suppliername =
:param2""", {'param1': supplier_ID, 'param2': supplier_name })

con.close()
```

Order in which parameters are passed does not matter

Output



```
>>>
[('S1001', 'Giant Store', '203-237-2079', 'rachel1@easy.com')]
```

Demo code to Insert a record using Positional Bind Variables

```
import cx_Oracle

con=cx_Oracle.connect("system/infy@XE")

cur=con.cursor()

supplier_ID = 'S1005'

supplier_name = 'Victor Electronics'

supplier_contactno = '115-265-4675'

supplier_emailed = 'victorelectronics@easy.com'


cur.execute('INSERT INTO supplier VALUES (:1,:2,:3,:4)',(supplier_ID, supplier_name,
supplier_contactno, supplier_emailid))


con.commit()

con.close()
```

Bind variables

Python
Variable Names

Demo Code to Insert a record using Named Bind Variables

```
import cx_Oracle
con=cx_Oracle.connect("system/infy@XE")
cur=con.cursor()


s_id = 'S1005'
s_name = 'Victor Electronics'
s_contactno = '115-265-4675'
s_emailed = 'victorelectronics@easy.com'

cur.execute(""" INSERT INTO supplier VALUES (:paramsupID, :paramsupName, :paramsupcontact, :paramsupemail) """, {'paramsupID' :
s_id, 'paramsupName' : s_name, 'paramsupcontact' : s_contactno, 'paramsupemail' : s_emailid} )

con.commit()
con.close()
```

Python Variable Names

Bind variables

Guided Activity : Assignment 5: Database: Passing Parameters

Hands-On : Assignment 6: Database: Passing Parameters

ExecuteMany Method

Inserting Multiple Records

- **executemany**() function of cursor object is used to execute the same SQL query for different set of data:
- **Syntax:**

```
<<cursor_object>>.executemany("<<operation>> , <<sequence_of_parameters>>")
```

- Consider a table 'Employee' in Oracle database with following specifications:

  Employee (employeeid NUMBER, employeename VARCHAR2(25))

- Inserting multiple records using **positional bind variables**:

```
import cx_Oracle
con = cx_Oracle.connect('oracle/infy123@localhost/xe')
cur = con.cursor()
counter=1001
cur.executemany("INSERT INTO Employee VALUES(:1,:2)",
[(counter,'Fabian'),(counter+1,'Gippy'),(counter+2,'Harry')])
```

Three records are inserted using positional bind variables

Demo Code for Inserting Multiple Records

- Inserting multiple records using named bind variables:

```
import cx_Oracle
con = cx_Oracle.connect('oracle/infy123@localhost/xe')
cur = con.cursor()

counter=1001

cur.executemany("INSERT INTO Employee VALUES (:paramEmpId,:paramEmpName)",
[{'paramEmpId':counter,'paramEmpName':'Fabian'},
{'paramEmpId':counter+1,'paramEmpName':'Gippy'},
{'paramEmpId':counter+2,'paramEmpName':'Harry'}])
```

Three records are inserted using named bind variables

Guided Activity : Assignment 7: Database: Executemany Method

Hands-On : Assignment 8: Database: Executemany Method

Cursor Attributes

Rowcount Attribute

- Contains the number of rows fetched by the cursor
- It's a read only attribute

- **Syntax :**

```
<<cursor_object>>.rowcount
```

- **Example:**

```
cur.execute("SELECT * FROM supplier ORDER BY supplierId")
print("Before fetchall(): " , cur.rowcount)
print(cur.fetchall())                    ''' returns all the records'''
print("After fetchall(): " , cur.rowcount)
```

- **Output:**

```
Before fetchall(): 0
After fetchall(): 4
```

Demo Code to identify number of rows updated:

```
import cx_Oracle
con=cx_Oracle.connect(" system/infy@XE ")
cur=con.cursor()

s_ID = 'S1005'
s_Email = 'batse@easy.com'
s_name = 'Victor Electronics'

cur.execute("UPDATE supplier SET supplieremailid=:1 , supplierName =:2 where supplierid=:3",
(s_Email,s_name , s_ID))

print (cur.rowcount)
con.close()
```

Number of rows updated

Description Attribute

- Returns information about the retrieved columns in the cursor.
- Returns NONE after execution of Insert/Update/Delete statements

- **Syntax :**

```
<<cursor_object>>.description
```

- **Example:**

```
cur.execute("SELECT * FROM supplier ORDER BY supplierId ")
print("Description for SELECT: ", cur.description) '''returns information about the columns'''
cur.execute("DELETE from supplier where supplierId = 'S1001' ")
print ("Description for DELETE: ", cur.description)     '''returns None'''
```

- **Output:**

```
Description for SELECT:  [('SUPPLIERID', <class 'cx_Oracle.STRING'>, 10, 10, 0, 0, 0), ('SUPPLIERNAME', <class 'cx_Oracle.STRING'>, 30, 30, 0, 0, 1), ('SUPPLIERCONTACTNO', <class 'cx_Oracle.STRING'>, 30, 30, 0, 0, 1), ('SUPPLIEREMAILID', <class 'cx_Oracle.STRING'>, 30, 30, 0, 0, 1)]
Description for DELETE:  None
```

Arraysize Attribute

- To returns number of rows fetched when fetchmany() method is invoked without argument.
- Default arraysize is 50, which can be changed as per requirements.

- **Syntax :**

```
<<cursor_object>>.arraysize
```

- **Example:**

```
cur.execute("""select * from supplier order by supplierId""")
print (cur.fetchmany())          ‘’’returns all the rows’’’
print (cur.arraysize)            ‘’’returns 50 – Default arraysize’’’
cur.arraysize=5                  ‘’’array size changed to 5’’’
print (cur.fetchmany())          ‘’’returns 5 rows as a tuple or list’’’
```

Guided Activity : Assignment 9: Database: Cursor Attributes

Hands-On : Assignment 10: Database: Cursor Attributes

Exception Handling

Exception Handling

Consider the previous example of deleting a record for a specific supplier. We use the following code:

```
import cx_Oracle
con=cx_Oracle.connect(" system/infy@XE ")
cur=con.cursor()
cur.execute(" DELETE FROM supplier WHERE supplier_id ='S1003' ")
con.close()
```

Assume that while writing the above query we mistakenly mentioned incorrect column name `'supplier_id'` instead of actual column name `'supplierid'`. What will happen in this scenario?

Try to execute this. You will get the following exception while executing the query:

```
cx_Oracle.DatabaseError: ORA-00904: "SUPPLIER_ID": invalid identifier
```

This will abruptly terminate the program and the further statements will not get executed. In such scenarios, Exception Handling can be used for smooth exit from the program.

Exception Handling

- cx_Oracle module allows us to fetch the details of error (error code and error message) generated by Oracle. The details are stored in DatabaseError class.
- **Example:**

```
import cx_Oracle
con=cx_Oracle.connect(" system/infy@XE ")
cur=con.cursor()
try:
    print("In try block")
    cur.execute(" DELETE FROM supplier WHERE supplier_id='S1003' ")
    con.commit()
    print("try block ends")
except cx_Oracle.DatabaseError as e:
    print("In except block")
    print("Error details: ",e)
finally:
    print("In finally block")
    con.close()
```

Incorrect Column Name, exception thrown here

This code will not be executed

'DatabaseError' class stores the error details which can be retrieved further

Connection is closed in 'finally' block, will always be executed

- **Output:**

```
In try block
In except block
Error details:  ORA-00904: "ITEMCODE": invalid identifier
In finally block
```

Learning Outcomes – Module 3

In this module we have learnt how to:

- Integrate Python application with Oracle database
- Perform the following database operations from Python application:
  - Create tables
  - Insert/Update/Delete records
  - Retrieve data from tables
- Handle database related exceptions in Python

References

- **Python and Oracle Database Integration:**
  - **Link :** *https://www.youtube.com/watch?v=rPfKXVmYuyg&list=PL1T-f7vLvANmcEOHjtRTYkHxY_UHkerOn&index=1*
  - **Author:** Blaine Carter
  - **Duration:** 3:04
- Learning Python, Fourth Editiony, Mark Lutz
- Core Python Programming, Second Edition, Wesley J. Chun
- Beginning Python, Norton Peter
- Python Essential Reference, Fourth Edition, David M. Beazley
- Python in a Nutshell, Second Edition, Alex Martelli
- Programming Python, Fourth Edition, Mark Lutz
- Online Python Documentation (http://www.python.org)

Thank You