Laboratory Worksheet Six

NOTE: *You should complete all previous worksheets before continuing*

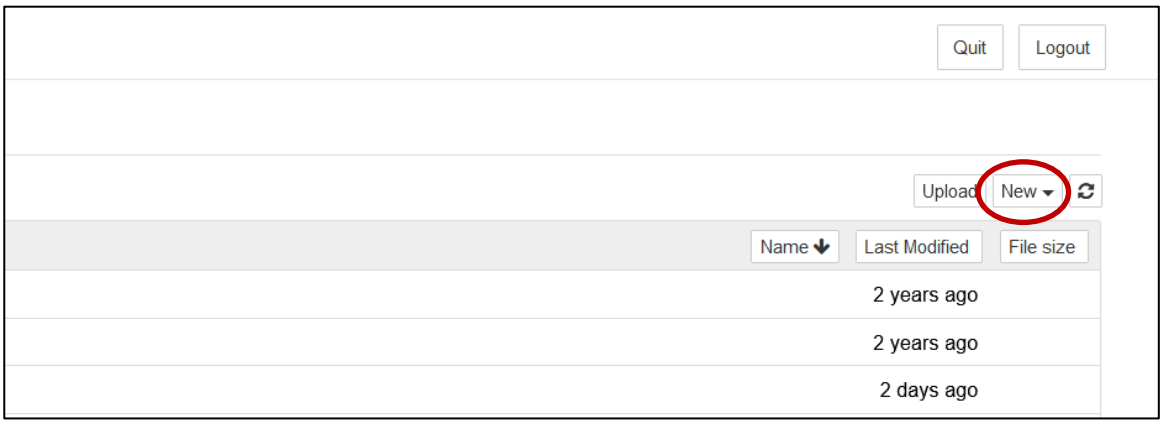
NOTE: *This worksheet includes work necessary for Assignment 1*

One effective tool for writing and refining Python code for data analysis is Jupyter Notebooks. Jupyter Notebooks enable you to write code in individual sections (known as "cells") and execute each block of code separately, saving you the trouble of writing and rewriting a complete program. If you need to make any changes after that, you can go back, amend them and rerun the program in the same window.

The foundation of Jupyter Notebook is IPython, an interactive way to run Python that uses the Read-Eval-Print-Loop (REPL) model. The Jupyter Notebook front-end interface is in communication with the IPython Kernel, which does the computational work. Jupyter Notebooks provide further functionality to IPython, such as the ability to store your code and output.

PART 1: CREATING A JUPYTER NOTEBOOK

- Press the Windows Key and search for “Jupyter Notebook” then click on “Jupyter Notebook (Anaconda3)”. This will open a terminal window and you will be asked how you want to open this file. Click Google Chrome and a Jupyter tab will open in Google Chrome.
- Anaconda Distribution can be downloaded for personal computers using the instructions be found in Additional Resources on Blackboard.
- Click ‘New’ button (circled in image below) in the top-right of the screen:

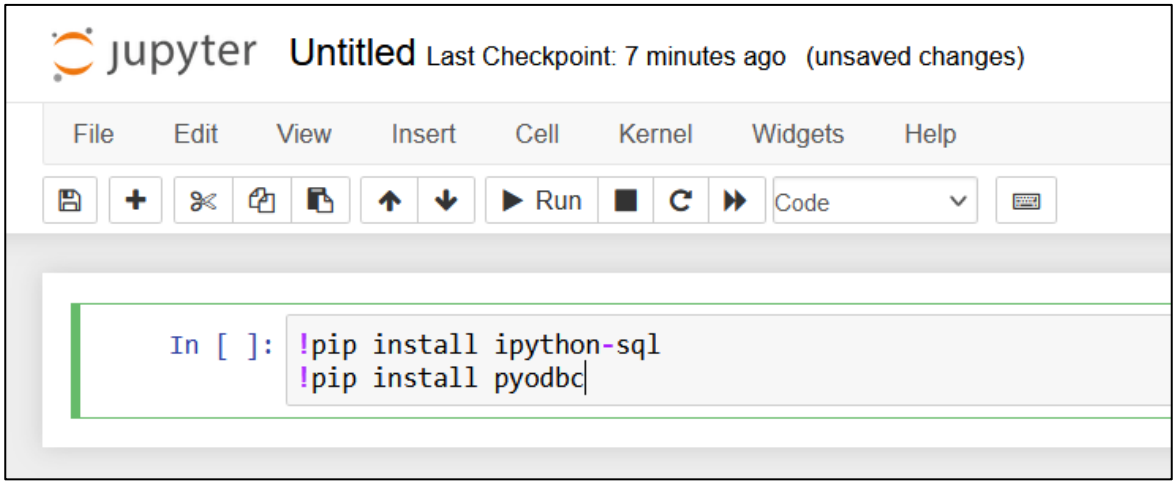


- Click “Python 3 (ipykernel).
- Your Jupyter Notebook has now been created.

IMPORTANT! Make sure to regularly save your Jupyter Notebook throughout this practical. The first time you save you will be prompted to give the Notebook a name. Call it something meaningful such as "COM106 Week 6".

PART 2: CONNECTING JUPYTER NOTEBOOK TO SQL SERVER

- Python code can now be written and executed in your Jupyter Notebook. Next, we will look at how we can connect the Jupyter Notebook to the PROJEMP database in MS SQL SERVER.
- First, there are a few packages we need to install:
    - You will add code to the Jupyter Notebook by typing in the box next to the 'In []:'
    - **Type** *!pip install ipython-sql*
    - Press the 'Enter' key on your keyboard to take a new line within the box.
    - **Type** *!pip install pyodbc*
    - You should now have something like in the screenshot below:



    - Click the 'Run' button above the cell.
    - There will now be a * symbol between the square brackets next to the cell, much like: 'In [*]'. This indicates that the code you've written is in the process of running. Once this finishes, the * will change to the number 1 to show that this is cell number 1 in your Jupyter Notebook.
    - You will also see a lot of text beneath the box which is indicating that the packages are now installed.
- Scroll down to the next available cell - 'In []:'
- With the necessary packages installed, we now need to explicitly import any necessary Python Libraries into the Jupyter Notebook.
    - Click on the next cell.

- **Type** *import pandas as pd*
- Press the Enter key to take a new line.
- **Type** *import pyodbc*
- As before, now click the 'Run' button.
- The necessary Libraries have now been imported.

- Now, we can create a connection between our Jupyter Notebook and the MS SQL SERVER database PROJEMP.
  - First, we create a connection by writing code that specifies the driver, the server name, the username and the password we want to use (on your personal computer you don't need to include **uid** or **pwd** in the connection string).
  - Type *conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")*
  - Pree the Enter key to take a new line.
  - **Type** *conn.close()*
  - Click 'Run' and verify no errors appear. There should be no output from the code at this stage.
    - The two lines of code will open a connection to the database and then close the connection. It is important to remember that the connection to the database must be opened before any queries can be written. It is equally as important to remember to **close the connection after finishing with the connection.**

PART 3: QUERING SQL SERVER DATABASE FROM JUPYTER NOTEBOOK

Creating a SELECT Query

- Now that we know how to establish a connection from our Jupyter Notebook to our MS SQL SERVER database, we can begin querying the database.

NOTE: DON'T RUN ANY CODE UNTIL INSTRUCTED.

  - Open the connection with the database. **type** *conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")*
  - On a new line, **type** *cursor= conn.cursor()*
    - This will allow us to access to the cursor class which is able to be used to access the established connection.
  - On a new line, **type** *selectQuery = "SELECT * FROM projemp.dbo.emp"*

- This saves our SELECT SQL query to a variable called ***selectQuery***.

- On a new line, **type** *result = cursor.execute(selectQuery)*
  - This will execute the SQL query and save the query result into a variable called ***result***.
- The result then needs to be iterated over to access each row of the result table. This is achieved using a 'for loop'.
  - On a new line, **type***:*

```
for row in result:
        print(f'{row}')
```

- Finally, we need to close the connection.
  - On a new line, you need to unindent the cursor so that it is at the furthest point to the left of the box and then **type** *conn.close()*
- Your cell should now look like the screenshot below:

```
In [1]: conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")
        cursor = conn.cursor()
        sql = "SELECT * FROM projemp.dbo.emp"
        result = cursor.execute(sql)
        for row in result:
            print(f'{row}')
        conn.close()
```

- We will look at specifics of the ***print()*** statement later in the practical.
- Now that the query is complete, we can execute it by clicking the **'Run'** button.
- Your output should show all records from the **emp** table, as demonstrated below:

```
('e1 ', 'armstrong      ', Decimal('50000.00'), 56, None, 'd1 ')
('e10', 'jones          ', Decimal('50000.00'), 49, 'e1 ', 'd3 ')
('e11', 'kelly          ', Decimal('50000.00'), 36, 'e7 ', 'd2 ')
('e12', 'mccoy          ', Decimal('50000.00'), 29, 'e3 ', 'd2 ')
('e13', 'neeson         ', Decimal('50000.00'), 36, 'e19', 'd1 ')
('e14', 'pearson        ', Decimal('50000.00'), 35, 'e17', 'd3 ')
('e15', 'pearse         ', Decimal('50000.00'), 28, 'e21', 'd1 ')
('e16', 'quinn          ', Decimal('50000.00'), 54, 'e2 ', 'd1 ')
('e17', 'roberts        ', Decimal('50000.00'), 27, 'e4 ', 'd3 ')
('e18', 'smyth          ', Decimal('50000.00'), 34, 'e21', 'd3 ')
('e19', 'trainor        ', Decimal('50000.00'), 39, 'e7 ', 'd1 ')
('e2 ', 'breen          ', Decimal('50000.00'), 21, 'e4 ', 'd3 ')
('e20', 'urquhart       ', Decimal('50000.00'), 22, 'e11', 'd3 ')
('e21', 'vance          ', Decimal('50000.00'), 19, 'e10', 'd1 ')
('e3 ', 'carroll        ', Decimal('50000.00'), 31, 'e10', 'd3 ')
('e4 ', 'deehan         ', Decimal('50000.00'), 48, 'e1 ', 'd1 ')
('e5 ', 'evans          ', Decimal('50000.00'), 45, 'e11', 'd1 ')
('e6 ', 'flynn          ', Decimal('50000.00'), 30, 'e17', 'd2 ')
('e7 ', 'greer          ', Decimal('50000.00'), 24, 'e1 ', 'd2 ')
('e8 ', 'hamill         ', Decimal('50000.00'), 38, 'e3 ', 'd1 ')
('e9 ', 'irwin          ', Decimal('50000.00'), 48, 'e2 ', 'd2 ')
```

Creating an INSERT INTO Query

- To create a query that inserts data into the database, a lot of what was used for the SELECT query can be reused.
- **Type** the following code into a new cell (but don't run it yet):

```
conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")
cursor = conn.cursor()
insertQuery = "INSERT INTO projemp.dbo.emp (eno, ename, salary, age, supno, dno) VALUES (?, ?, ?, ?, ?, ?);"
values = ['e22', 'watson', 35000.00, 43, 'e1', 'd3']
cursor.execute(insertQuery, values)
conn.commit()
conn.close()
```

- There are some changes in this code from the SELECT query:
  - Instead of defining the values in the ***insertQuery*** variable, ?s (parameter codes) are used in the brackets after VALUES. The actual values are defined in a Python list.
  - When the code is executed, both the ***insertQuery*** variable and ***values*** list must be entered.
  - Once the query has been executed, the data to be inserted into the database is queued and so ***conn.commit()*** must be used to finally insert the data.
- **Run the code.**
- Now, open MS SQLSERVER and verify that the record has been inserted into the **emp** table from the **PROJEMP** database.

Creating an UPDATE Query

- **Type** the following code into a new cell (but don't run it yet):

```
conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")
cursor = conn.cursor()
updateQuery = "UPDATE projemp.dbo.emp SET salary = ?;"
updateValue = 50000.00
cursor.execute(updateQuery, updateValue)
conn.commit()
conn.close()
```

- Much like for the INSERT INTO query, we do not explicitly define values in for the query variable. Instead, we use a ? to indicate that a value will be included here by using another variable. This other variable is the ***updateValue*** variable. This is the value that will replace the ? in the ***updateQuery*** variable when the code is executed.

- Similarly to the INSERT INTO query, the update to be completed is queued until it is committed.
- **Run the code.**
- Now, open MS SQLSERVER and verify that the record from the **emp** table has been updated.

Creating a DELETE Query

- **Type** the following code into a new cell (but don't run it yet):

```
conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")
cursor = conn.cursor()
deleteQuery = "DELETE FROM projemp.dbo.emp WHERE eno = ?;"
deleteValue = 'e22'
cursor.execute(deleteQuery, deleteValue)
conn.commit()
conn.close()
```

- Again, we follow the same pattern in that we do not explicitly define values in the query variable but instead must use a separate variable which defines the value for the query.
- **Run the code.**
- Now, open MS SQLSERVER and verify that the record from the **emp** table has been deleted.

Creating a JOIN Query

- Writing a JOIN query includes a lot of similar code used for a SELECT query.
- Read and run the following code in a new cell:

```
conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")
cursor = conn.cursor()
selectQuery = "SELECT ename, pname FROM projemp.dbo.EMP INNER JOIN projemp.dbo.WORKS
ON projemp.dbo.EMP.eno = projemp.dbo.WORKS.eno INNER JOIN projemp.dbo.PROJ ON
projemp.dbo.WORKS.pno = projemp.dbo.PROJ.pno WHERE pname = ?;"
pnameValue = 'database'
result = cursor.execute(selectQuery, pnameValue)
for row in result:
    print(f'{row}')
conn.close()
```

- This code lists all employees who work on the database project.
- Notice how the actual query itself is no different to the query we would write in MS SQL SERVER to achieve the same results expect for not explicitly stating the value of pname (and the inclusion of ***projemp.dbo.***).

Handling the Query Result

- As we receive a result returned from SELECT queries, it's important to consider how to handle the query result.
- **Run** the following code in a new cell **(IT SHOULD RETURN AN ERROR – unhashable type: 'pyodbc.ROW')**:

```
conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")
cursor = conn.cursor()
selectQuery = "SELECT eno FROM projemp.dbo.emp"
result = cursor.execute(selectQuery)
for row in result:
   print({row})
conn.close()
```

- This is caused by attempting to access a row from the result of the query without specifying a column name.
- We need to use F-strings to retrieve the row without specifying any particular column name. F-strings make the result more readable and concise. They are also necessary when we are trying to print a row without specifying column names. We use F-strings by encapsulating the contents of the ***print()*** statement with two quotation marks and by preceding the first quotation mark with **f**
    - Update the previous cell so that the **print** statement looks like *print(f'{row}')*
    - **Run** the updated code and note how the output now works.

PART 4: EXERCISES

Before starting this section, make sure you have saved your Jupyter Notebook. Create a new Jupyter Notebook titled "Week 6 Exercises" and write your answers to the following queries in there.

Outer Joins, Self Joins and working with NULL

In the previous worksheets we haven't been overly concerned with the possibility of NULL values in the database. In fact, the **PROJEMP** database contains only one **NULL** value. Have a look at the **EMP** table; the supervisor number, supno, of employee 'armstrong' is **NULL**. Presumably, 'armstrong' is the head of the organisation and so doesn't have a supervisor.

In this section we will change some of the data in the database to introduce other **NULL** values and see how **NULL** can be useful in outer join and self join queries. We'll also see why the possibility of **NULL** values must be considered in formulating queries and how, if

NULL isn't considered, a seemingly correct query can return incorrect or incomplete results.

The queries to be written in this section will largely follow the template of the SELECT queries written for Jupyter Notebooks that have been provided earlier in the worksheet. For example:

```
conn = pyodbc.connect("Driver={SQL Server};Server=localhost\SQLEXPRESS;uid=sa;pwd=Labuser1")
cursor = conn.cursor()
selectQuery = ""
result = cursor.execute(selectQuery)
for row in result:
   print({row})
conn.close()
```

It is what needs to be written between the double quotes ("") in the line *selectQuery = ""* that requires your attention. The queries you write between the "" will be written how you would normally write them in MS SQL SERVER.

If you are stuck on a question, try answering it by writing the query in MS SQL SERVER first before then writing it in your Jupyter Notebook.

a) Write a query to find how many staff are in each department, reported by **department number** (dno).

Expected result:

```
('d1 ', 9)
('d2 ', 5)
('d3 ', 7)
```

b) Modify your query to find how many staff are in each department, reported by **department name** (dname).

Expected result:

```
('engineering    ', 7)
('information    ', 9)
('service        ', 5)
```

c) The queries above should return results for **three** departments. However, there are **four** departments in the PROJ table (check the table). One of the departments hasn't any employees.

Write a query, using an OUTER JOIN, to find the department number and name of departments with no staff.

Expected result:

```
('d4 ', 'personnel      ')
```

d) Now, write a query to find how many staff are in each department, reported by **department number** (dno), but this time including the department with no employees.

Your query will have included the COUNT() aggregate function. Does it make any difference if you use COUNT(*) or counted on the primary key? Explain.

Expected result:

```
('d1 ', 9)
('d2 ', 5)
('d3 ', 7)
('d4 ', 0)
```

e) Using the method developed in 2d) above, write a query to find how many staff work on each project, reported by project number (pno), including any projects with no staff assigned.

Expected result:

```
('p13', 8)
('p15', 5)
('p19', 10)
('p23', 9)
('p26', 7)
('p31', 0)
```

f) Write a query, using a **self join approach**, to find the name and employee number of employees who are NOT supervisors (i.e., don't supervise any employees).

HINT: When using '**AS**' to create an alias for a table or to create a TEMP (copy) table for self joins, ***projemp.dbo.*** is not needed for the alias e.g. if we wrote:

SELECT projemp.dbo.EMP.eno FROM projemp.dbo.EMP AS TEMP

We could then refer to TEMP without including ***projemp.dbo.***:

*SELECT * TEMP*

We only need to use ***projemp.dbo.*** when referring to tables actually stored in our database.

Expected result:

```
('e12', 'mccoy          ')
('e13', 'neeson         ')
('e14', 'pearson        ')
('e15', 'pearse         ')
('e16', 'quinn          ')
('e18', 'smyth          ')
('e20', 'urquhart       ')
('e5 ', 'evans          ')
('e6 ', 'flynn          ')
('e8 ', 'hamill         ')
('e9 ', 'irwin          ')
```

g) A new employee joins the company. Use SQL to add her to the EMP table as follows:

```
INSERT INTO EMP (eno, ename, age)
VALUES ('e0', 'pat', 35);
```

Use the '**Creating an INSERT INTO Query'** section earlier in the worksheet to help with this question.

Verify that the record has been added to the table in MS SQL SERVER. **Notice, the new employee hasn't been assigned a department or a supervisor and doesn't work on any projects.**

h) Write a query to find the name and employee number of employees who don't work on any projects. Since 'pat' hasn't been assigned to any projects, she should appear in the results.

Expected result:

```
('e0 ', 'pat            ')
```

i) Modify the query developed in 2h) above to find the name, employee number, **department number and department name** of employees who don't work on any projects.

Note: Your query should return 'pat' even though she hasn't been assigned to a department and 'None' is returned instead of 'NULL'.

Expected result:

```
('e0 ', 'pat                ', None, None)
```