kubrick

Connecting to SQL Databases with Python

11 April 2023

Day One Contents

kubrick

Connecting to Databases from Python

Configuring PyODBC

Querying Data & Processing Results

Working with Data Types

Connecting to Databases from Python
Database API Specification

kubrick

Python has a well-established standard pattern for accessing databases -
the DBI specification, currently in its v2.0 release
See https://peps.python.org/pep-0249/

DBI came out of PEP-249 and specifies what you should expect in an API for accessing databases

The standard is very useful as it allows programmers to understand the features of a database driver quickly and become productive with it, so long as it adheres to the DBI Specifications

ODBC Open Database Connectivity

kubrick

ODBC was established as a standard for Database Connectivity in the 1990's
(way before Python was developed)

The Standard was proposed from work done at Microsoft and Samba and widely adopted by other database vendors

The Standard defines how to connect to a database and how data will be passed between applications (clients) and databases

ODBC was initially available on the Windows platform, but it was soon ported to Mac OsX and Linux

Almost every database vendor or developer will have created an ODBC driver for their database
The ODBC driver is supplied as a piece of executable code that is installed on the operating system
ODBC drivers operate at quite a low level and are generally written in C or C++
They provide the API that higher level software interacts with.

ODBC drivers are available for Microsoft SQL Server, Oracle, MariaDB, PostgreSQL, SQLite and many other databases and data sources (such as Excel)

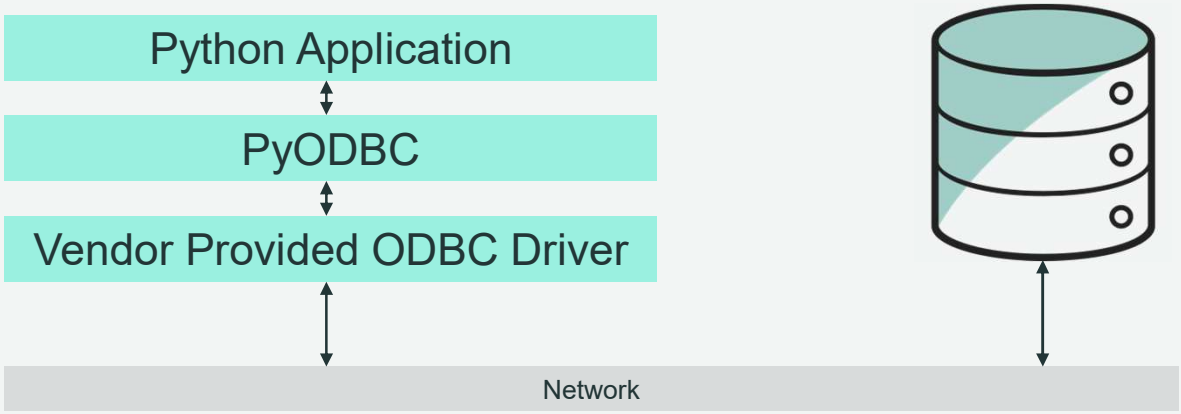
PyODBC – Python's DBI V2 Interface to ODBC

kubrick

PyODBC provides an implementation of the Python DBI V2.0 interface on top of a vendor provided ODBC Driver

It provides Python programmers with the familiar DBI API and provides a useful degree of insulation from the differences in the underlying database technologies

pyodbc has a splendid support wiki at github.com/mkleehammer/pyodbc/wiki and is essential post course reading



PyODBC – Benefits for Python Programmers

kubrick

Operating System Independent
Same Python code runs on Windows, Mac, Linux

Database Independent
But there are still some differences in the way databases operate

Simple and familiar programming interface from DBI V2.0

Setting Up the PyODBC Development Environment

kubrick

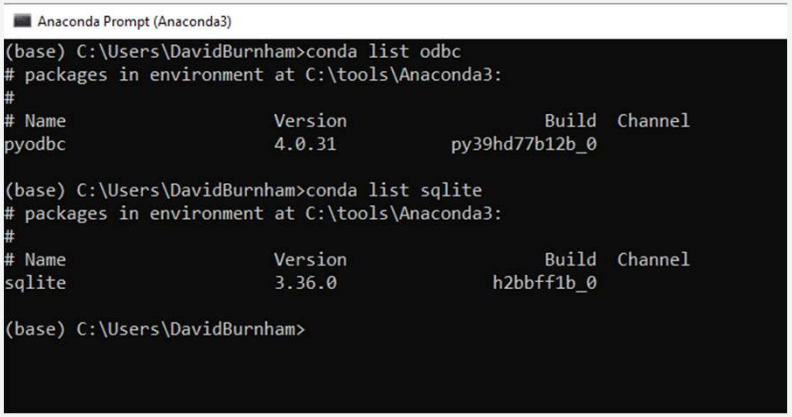
The set up is designed to work on a Windows 10 PC
We will be connecting to 2 different databases in these sessions

- a SQL Server Database you accessed earlier in your SQL training
- a local SQLite file-based database

- It will help to install SQL Server Management Studio for looking at SQL objects
  https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16

- The ODBC driver for Microsoft SQL Server is available at :
  https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-driver-for-sql-server?view=sql-server-ver16

- Anaconda Python – should already have pyodbc and sqlite installed
  use “conda install” to install if not
- If using another Python distribution
  create a virtual env and install the PyODBC module with pip



- Visual Studio Code with the Python extensions is my preferred development IDE, but you can use a the Jupyter notebook server that comes with Anaconda

PyODBC Development Environment

kubrick

Essential:

Python 3.7+ (Anaconda works fine or you can install vanilla Python and install the additional modules with pip3
PyODBC & Sqlite3 python modules
Microsoft SQL Server ODBC Driver (V17+) – download from Microsoft
(https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-driver-for-sql-server?view=sql-server-ver16)
Development IDE – one that you are productive with Visual Studio Code is my preference

Nice to haves:

Microsoft SQL Management Studio– download from Microsoft
(https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16)

PyODBC Object Model – Module Level

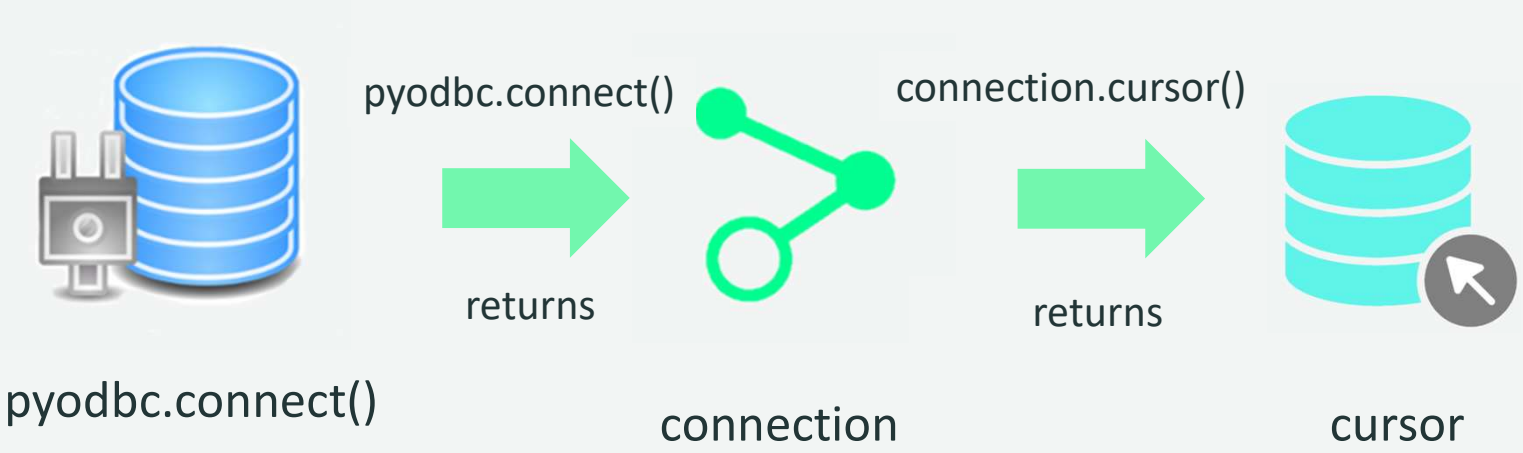
kubrick

Configuration attributes

pyodbc.connect()

Once the pyodbc.connect() function has been called successfully a connection object is created. The connection object has a method (cursor) that returns a cursor allowing the application to submit SQL requests to the database.

PyODBC Object Model – connect returns connection



Once the pyodbc.connect() function has been called successfully a connection object is created.

The connection object has a method (cursor) that returns a cursor allowing the application to submit SQL requests to the database.

PyODBC Object Model – Cursor Object

kubrick

Cursor

cursor.execute()

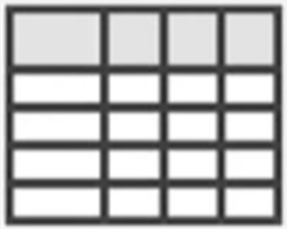
SQL

Cursors are used to pass SQL statements and get results from the database

Cursor

cursor.fetchone()
cursor.fetchval()
cursor.fetchall()
cursor.fetchmany()

returns

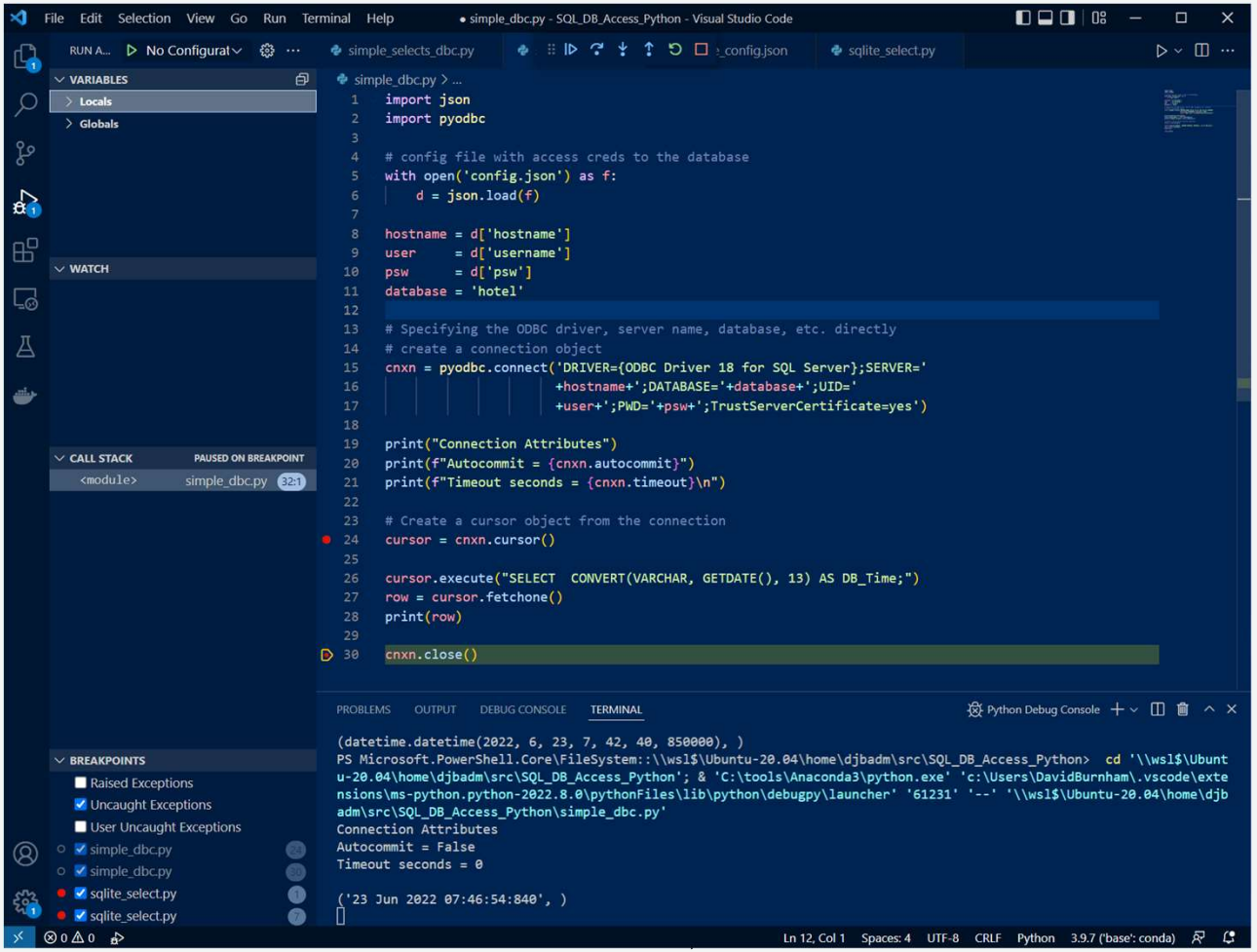


Row, Value or a Rowset

Cursors are python representations of a database cursor – they may or may not be implemented on top of database cursors. They are not identical.

Demonstration – Connecting to a Database

kubrick



Exercise Ex1

kubrick

Make sure your PC Development environment is set up to develop with Python PyODBC
Connect to the database we used for the SQL training
Here is a json fragment for your config.json file :

```
{       "hostname": "ec2-35-176-213-89.eu-west-2.compute.amazonaws.com",
        "database": "hotel",
        "username": “YourUserName",
        "psw": "********"
}
```

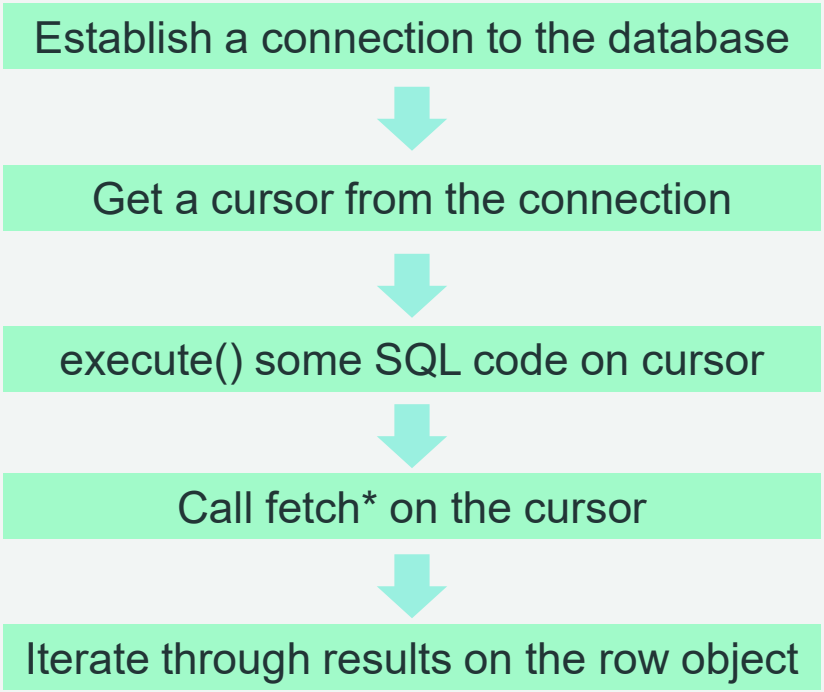
Identify the Operating System the database is running on :
Hint look up the information stored in the global variable @@VERSION

Work as individuals (you need to all set up a working pydbc development environment)
The exercise should take about 20 minutes maximum

As always message me if you are having difficulties … ☺

Basic Query Flow

kubrick



Points of note about pyodbc cursor metadata

kubrick

After a piece of SQL has been executed on a pyodbc cursor – the cursor contains useful metadata
The **cursor.description** attribute is populated with the column name and datatype for each row being returned - the attribute is a tuple of tuples

cursor.messages contains any messages from the database that may have resulted from the cursor execution

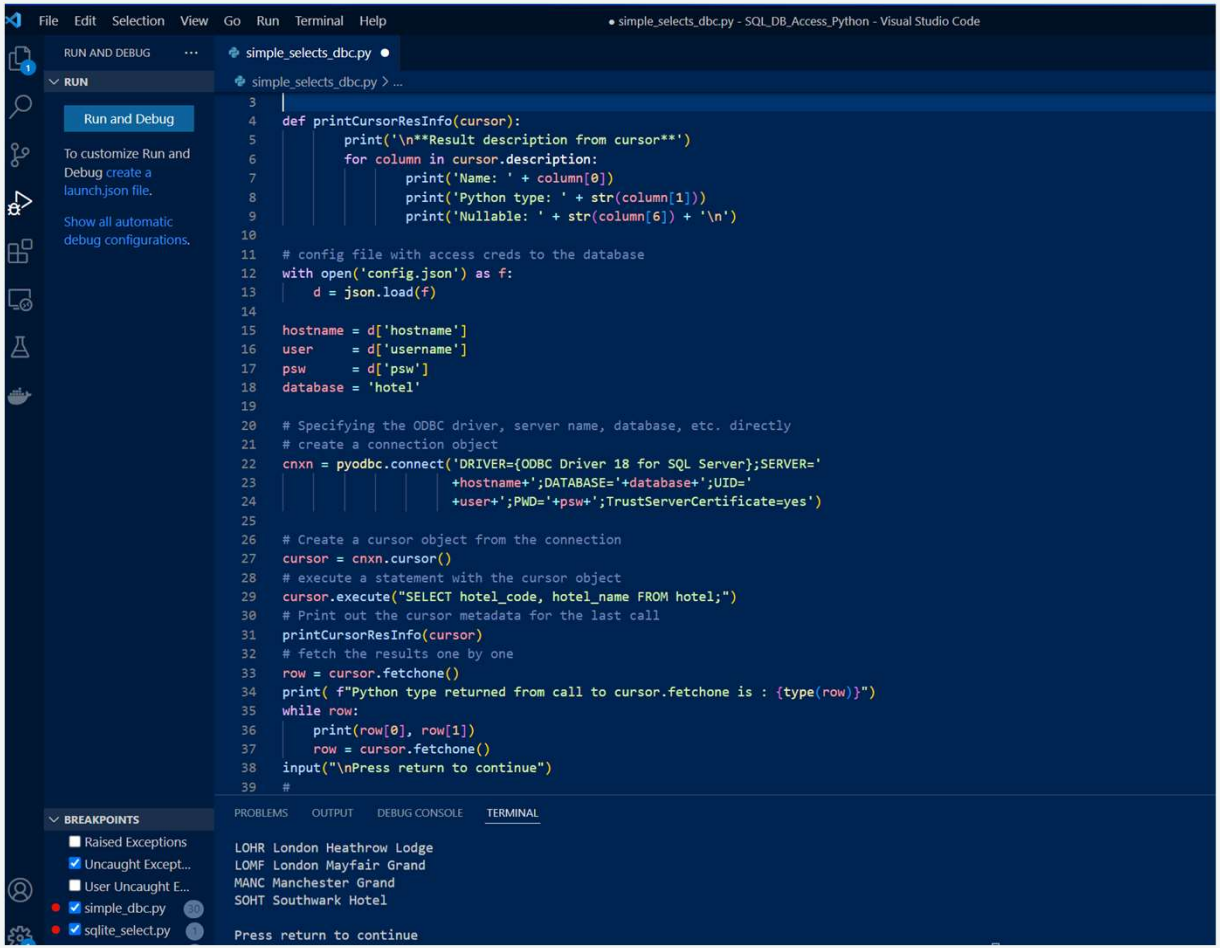
cursor.rowcount contains the number of rows that have been affected by any DML statements.

Its implementation is database driver specific and select statements (which don't change any rows) just return a cursor.rowcount of -1 for the SQL Server ODBC driver

See https://github.com/mkleehammer/pyodbc/wiki/Cursor for further details

Demo – Executing Select Queries

kubrick



Using parameters with execute() in SELECTs

kubrick

Parameters can be passed to a SQL statement by including a ? character in the text of the SQL and passing the values to substitute for the ? character as parameters.

Parameters can also be used with any other SQL statements such as inserts, updates, deleted and stored procedure executions. This is extremely useful as it ensures you do not have to build SQL statements using string concatenation which is a major cause of SQL Injection vulnerabilities in applications.

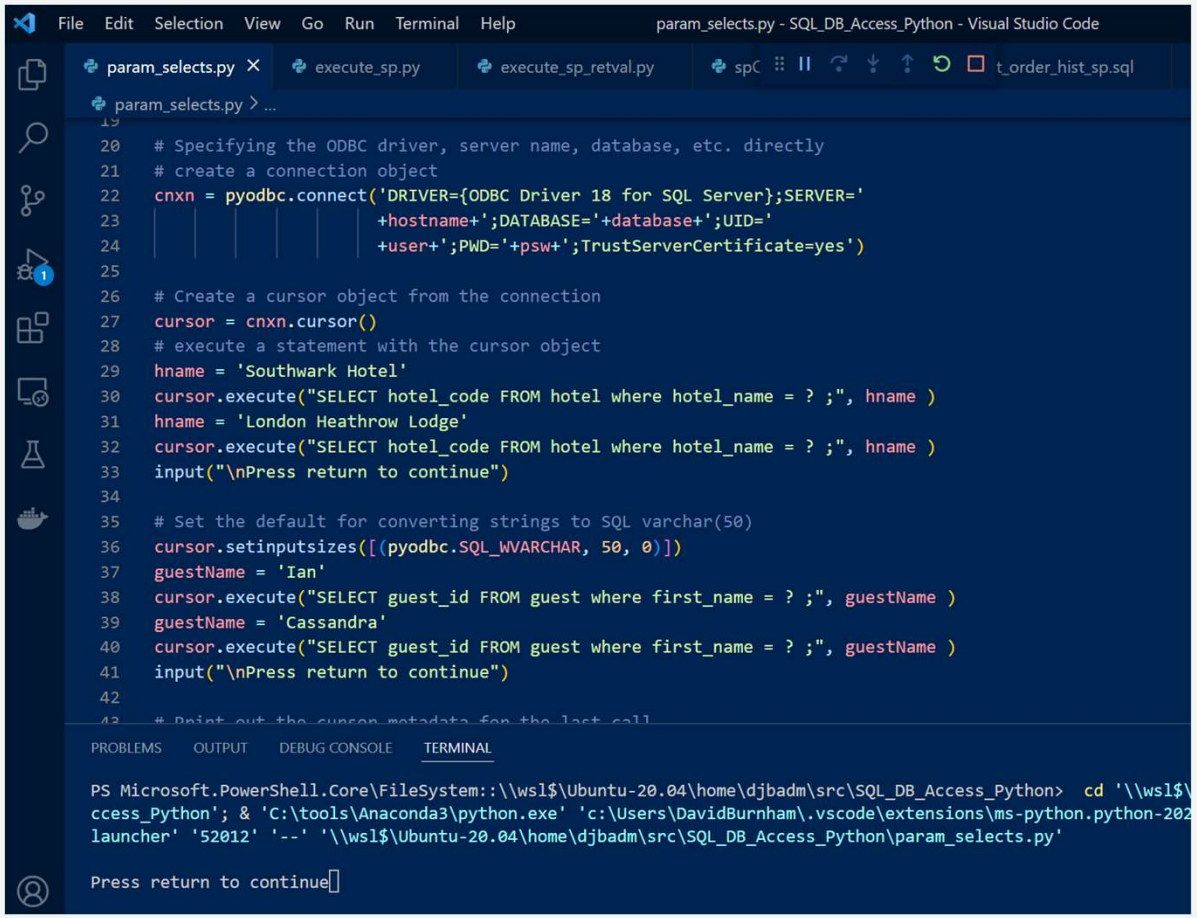
There is an issue with the pyodbc module and the SQL Server driver you need to be aware of though. The pyodbc driver tries to optimize SQL varchar variable sizes and will parse each python string variable and create a new (different) SQL submission to the driver for each different string length.
This is very bad for database optimisation as it results in many SQL execution plans being generated for what is essentially the same SQL statement.

We can avoid this by explicitly setting the size of strings that pyodbc will use for strings :

```
cursor.setinputsizes(pyodbc.SQL_WVARCHAR, 50 ,0)  # sets strings to be converted to varchar(50)
```

Demo – Executing Select Queries with Parameters

kubrick



Inserts updates deletes using pyodbc

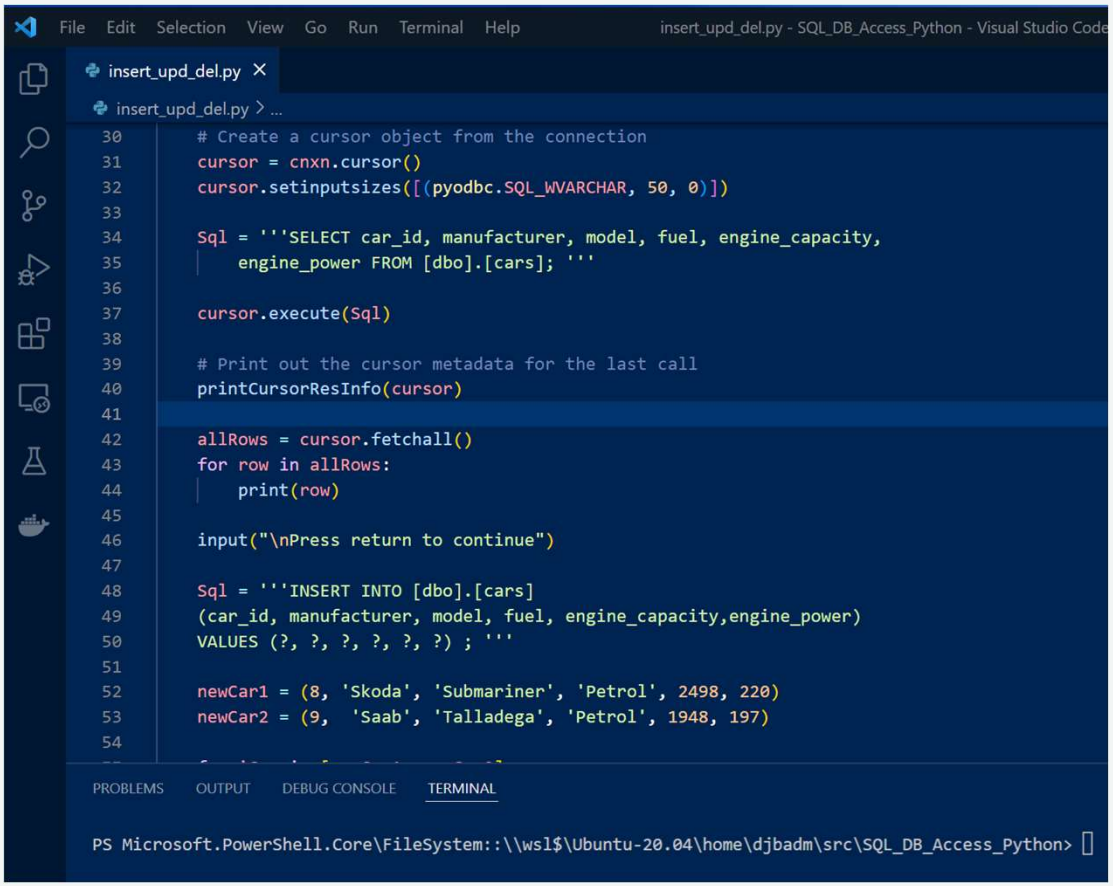
kubrick

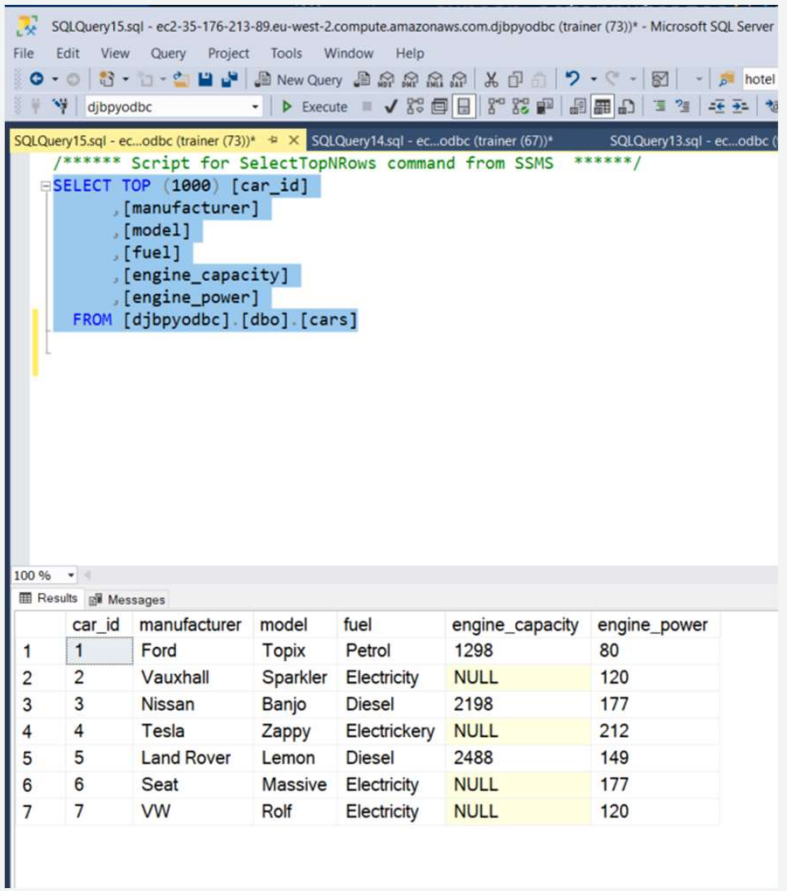
SQL INSERT, UPDATE and DELETE statements can all be called and parameterised in the same way that SELECT statements can be. In the next demonstration we will see how this can be performed.

For the purposes of the demonstration we will set the connection.autocommit parameter to True, which will cause the change to be applied to the database after each statement is executed. This is not the default, but we will discuss how this parameter is managed later in the course.

Demo – Inserts Updates & Deletes with Parameters

kubrick





	car_id	manufacturer	model	fuel	engine_capacity	engine_power
1	1	Ford	Topix	Petrol	1298	80
2	2	Vauxhall	Sparkler	Electricity	NULL	120
3	3	Nissan	Banjo	Diesel	2198	177
4	4	Tesla	Zappy	Electrickery	NULL	212
5	5	Land Rover	Lemon	Diesel	2488	149
6	6	Seat	Massive	Electricity	NULL	177
7	7	VW	Rolf	Electricity	NULL	120

Exercise Ex2
Selecting inserting and updating with parameters

kubrick

Do this exercise in your pods

Set up the bikestore tables in your own database on the "ec2-35-176-213-89.eu-west-2.compute.amazonaws.com" database server using the the 2 SQL scripts provided in the chat. Use the v_customer_order_totals view to write a python function that takes the customer number as a parameter returns the order history for that customer

Write a python script to add 5% to the price of all products made by Trek. What is the sum total of all of the products made by Trek in the products catalogue?

The exercise should take approximately 45 minutes

Calling Stored Procedures using pyodbc

kubrick

Calling stored procedures from pyodbc provides a more of a challenge. A well written stored procedure will return either a simple result set (in a very similar way that select statements do) or a simple value parameter.

A poorly designed stored procedure can return multiple different result sets from multiple select statements and also output parameters. Fortunately, pyodbc can cope with all of these eventualities.

The **cursor.nextset()** method moves the window onto the next result set coming back from the stored procedure and returns True, if there is indeed another result set. The new result set can then be accessed using one of the fetch methods we looked at earlier with simple selects.
If there is no further result set calling cursor.nextset() returns False.

Be careful when calling this method as all rows remaining in the current result set are discarded and can only be accessed again by re executing the stored procedure

Handling input and output parameters with Stored Procedures using pyodbc

kubrick

Input parameters with stored procedures are managed in the same way as we saw earlier with selects.
We use one or more ? characters in the call and put substitution values in the parameters.

With output parameters we need to alter the SQL text of the call to declare a variable in TSQL, then reference it in the call to the procedure and then, after the call to the procedure completes,
we select the value in a further statement and it is returned to the pyodbc cursor as another result set

```
5   myInputVariable = 'whatever I want - could be string/int/number whatever'
6
7   tSql = '''DECLARE @out1 int;
8           DECLARE @out2 varchar(50);
9           EXEC [dbo].[someDodgySP] ?, @out1 OUTPUT, @out2 OUTPUT;
10          SELECT @out1 AS my1stVarReturned;
11          SELECT @out2 AS my2ndVarReturned;'''
12
13  cursor.execute(tSql, myInputVariable)
14
15  # Use cursor.nextval to get any result sets from someDodgySP
16  # and then the my1stVarReturned and my2ndVarReturned variables
```

Exercise Ex 3
Selecting from a Stored Procedure with multiple result sets and Output Parameters

kubrick

Use the bikestore tables we created earlier

Write a customer summary stored procedure procedure to summarise a customer's orders and their total spend with the company

The report should contain 3 sections.
1) A header with the customer's first name and last name, their email address and their city and state and phone number

2) The orders section should contain order_id the order date and the total order cost value

3) Finally return their total spend as an output parameter.

Format the results as a report in Python
Who are the top 3 biggest spenders with the bikestore ?

Managing Type Conversions between Python and SQL

kubrick

- Differences between the type system between Python and SQL databases
- Implicit type conversions performed by pyodbc
- Approaches to types that cannot be easily converted
  - Convert in-the-database to a type that can be handled by pyodbc
  - Perform output conversion in the pyodbc driver

Type Systems between Python and SQL Databases are Very Different

kubrick

pyodbc does a great job of hiding the differences of approach between the two type systems

Python is Dynamically Typed

- You do not have to declare types on variables
- You can change the type of a variable after you use it
- Any type checking gets done at run-time

Database SQL implementations are Strongly Typed

- You must declare a type for a variable before you use it
- You cannot change a variables type without re-declaring it
- Type checking is done in a compile phase before run-time

Implicit Type conversions performed by pyodbc

kubrick

- Since Python 3 string objects have been rendered in Unicode (utf-16 or utf-8)
- This has made the transferring of string data between Python and SQL databases fairly seamless (as we saw before)
- The recent SQL Server ODBC driver works out of the box with Python 3
- Some other databases and operating system platforms may need more configuration
- The connection methods setEncoding() and setDecoding() can be used to alter how the conversions are performed
- As always Michael Kleehammer's excellent documentation has many examples of common issues experienced on database platforms. (https://github.com/mkleehammer/pyodbc/wiki/Unicode)

Implicit Type conversions performed by pyodbc passing data to the database

kubrick

Python Datatype	Description	ODBC Datatype
None	null	varies (1)
bool	boolean	BIT
int	integer	SQL_BIGINT
float	floating point	SQL_DOUBLE
decimal.Decimal	decimal	SQL_NUMERIC
str	UTF-16LE (2)	SQL_VARCHAR or SQL_LONGVARCHAR (2)(3)
bytes, bytearray	binary	SQL_VARBINARY or SQL_LONGVARBINARY (3)
datetime.date	date	SQL_TYPE_DATE
datetime.time	time	SQL_TYPE_TIME
datetime.datetime	timestamp	SQL_TYPE_TIMESTAMP
uuid.UUID	UUID / GUID	SQL_GUID

1.If the driver supports it, SQLDescribeParam is used to determine the appropriate type. If it is not supported, SQL_VARCHAR is used.
2.The encoding and ODBC data type can be changed using Connection.setencoding(). See the Unicode page for more information.
3.SQLGetTypeInfo is used to determine when the LONG types are used. If it is not supported, 1MB is used.

Implicit Type conversions performed by pyodbc for data coming from the database

kubrick

Description	ODBC Datatype	Python Datatype
NULL	any	None
bit	SQL_BIT	bool
integers	SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT	int
floating point	SQL_REAL, SQL_FLOAT, SQL_DOUBLE	float
decimal, numeric	SQL_DECIMAL, SQL_NUMERIC	decimal.Decimal
1-byte text	SQL_CHAR	str via UTF-8 (1)
2-byte text	SQL_WCHAR	str via UTF-16LE (1)
binary	SQL_BINARY, SQL_VARBINARY	bytes
date	SQL_TYPE_DATE	datetime.date
time	SQL_TYPE_TIME	datetime.time
SQL Server time	SQL_SS_TIME2	datetime.time
timestamp	SQL_TIMESTAMP	datetime.datetime
UUID / GUID	SQL_GUID	str or uuid.UUID (2)
XML	SQL_XML	str via UTF-16LE (1)

1.The encoding can be changed using Connection.setdecoding(). See the Unicode page for more information.
2.The default is str. Setting pyodbc.native_uuid to True will cause them to be returned as uuid.UUID objects.

Approaches to use when there is no compatible type in pyodbc: Convert to a compatible type in the Database:

kubrick

- In this demonstration we will show how we can use functions or methods in the database to render incompatible data types into string types that can be handled by the pyodbc driver.

- Here the EarthquakeData table has a spatial geography type that is specific to the SQL server database.
- We use the StAsText() method to convert it to a textual value

spatial_fail_eg.py

```
25
26  # We can use a python context manager with pyodbc.connect()
27  with  pyodbc.connect(connectString) as cnxn:
28      # Create a cursor object from the connection
29      cursor = cnxn.cursor()
30      cursor.setinputsizes([(pyodbc.SQL_WVARCHAR, 50, 0)])
31
32      # Select on a Geography type in Earthquakeinformation
33      Sql = '''SELECT EarthquakeID, Earthquakeinformation
34              FROM [djbpyodbc].[dbo].[EarthquakeData];'''
35
36      cursor.execute(Sql)
37
38      input("\nPress return to continue")
39      # Print out the cursor metadata for the last call
40      try:
41          allRows = cursor.fetchall()
42      except pyodbc.Error as pyodbcError:
43          print(f"Error: {str(pyodbcError)}")
44
45      input("\nPress return to continue")
46
47      # Define SQL BUT render point as a string
48      Sql = '''SELECT EarthquakeID, Earthquakeinformation.STAsText()
49              FROM [djbpyodbc].[dbo].[EarthquakeData];'''
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

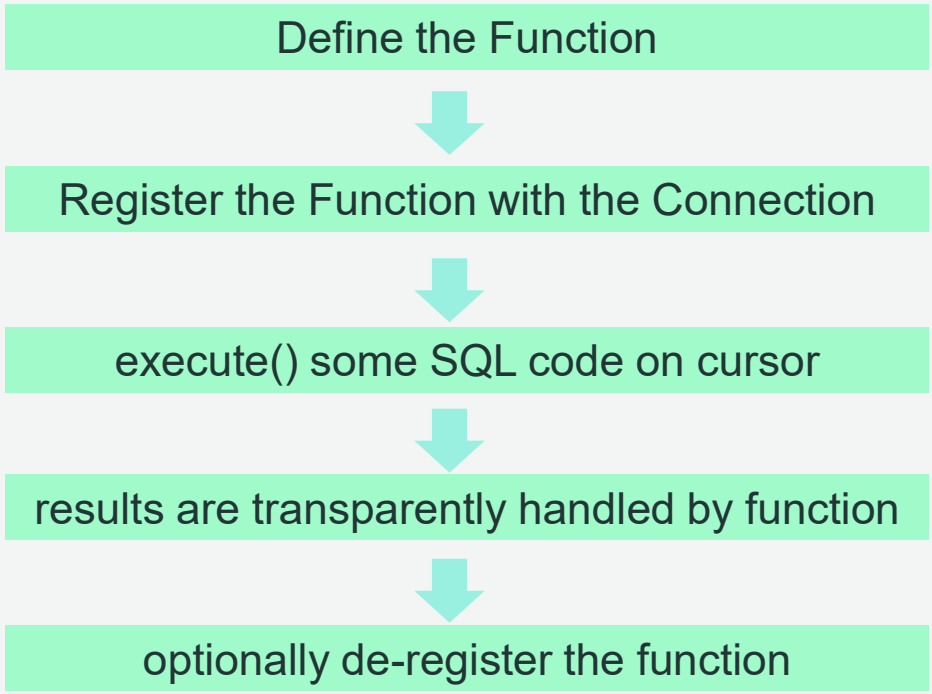
```
Error: ('ODBC SQL type -151 is not yet supported.  column-index=1  type=-151', 'HY106')

Press return to continue

**Result description from cursor**
Name: EarthquakeID
Python type: <class 'int'>
```

Using Output Convertor Functions

kubrick

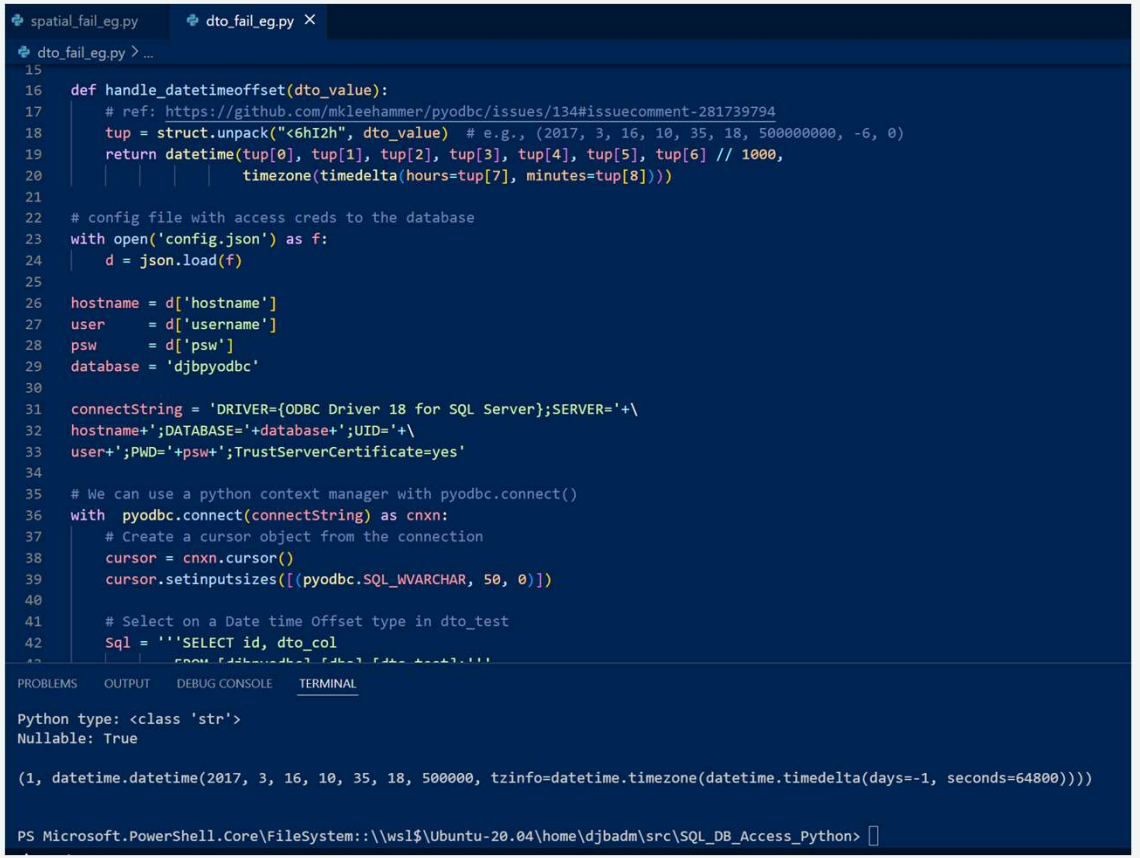


Approaches to use when there is no compatible type in pyodbc: Output converter functions in pyodbc

kubrick

- In this demonstration we will show how we can an output converter function that is run by the pyodbc driver to render incompatible data types into string types that we can use in our python code.

- Here the dto_test table has a datetime offset column, a type that is not handled by ODBC and the pyodbc module natively. In the demo we register a python function handle_datetimeoffset() to convert the datetime offset into a datetime data type we can work with in python.



In-Database Conversion or Output Convertor Functions How to choose which ?

kubrick

In Database Type Conversion

- In-Database conversion can be made to work on both values passing to and from the database
- In-Database conversion is generally easier to code
- In database conversion requires that you are able to make changes to the database and that is not always possible

Output Convertor Functions

- Only convert data coming from the database to pyodbc, passing to the database
- Performed in pyodbc – in Python
- Might be your only option if you cannot make changes to the database

SQL Databases bundle operations together in Transactions

kubrick

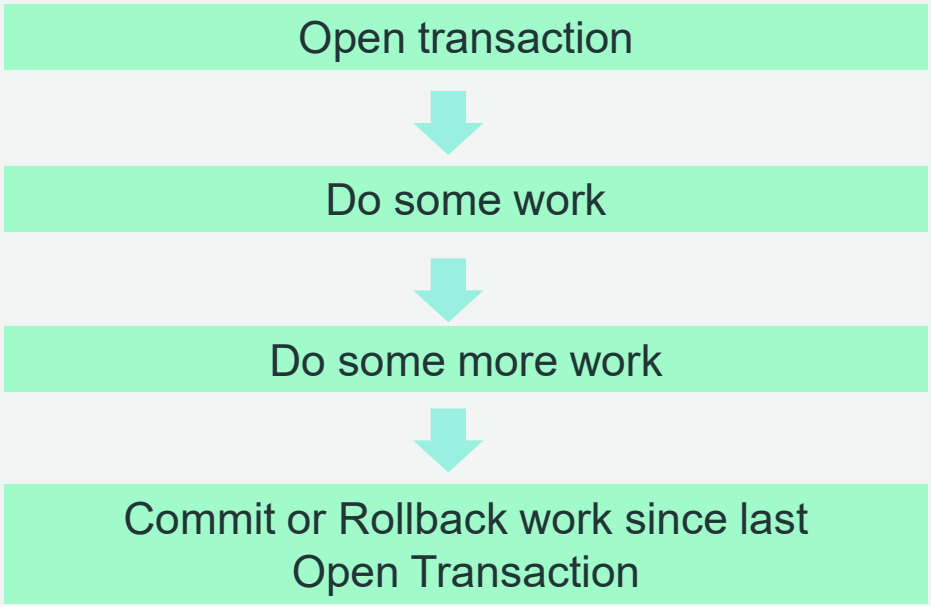
- Most SQL Database systems support Database Transactions
- Transactions are a way in which databases can bundle together related operations

- Consider a bank needing to transfer money from one account to another
  They might perform this as 2 updates to the accounts table
  1st update reduces balance on account 1 by the amount of money to be transferred
  2nd update increases balance on account 2 by the amount of money to be transferred
- The bank would want either both updates to occur OR neither
  If either of the updates occur on their own – then the bank has a problem
- Transactions allow the grouping of these changes together so they all succeed OR they all fail

- You may hear of people referring to **ACID** transaction properties
  Atomic – the transaction is an indivisible unit of work that all happens or none of it does
  Consistent – the database system moves from one state where whole transactions have been applied to another – with no visible state where parts of transactions have been applied
  Isolated – while one transaction of work is being applied it doesn't affect ant others in progress
  Durable – when a transaction has succeeded it is recorded securely in the database

Setup and commit for transactions

kubrick



How pyodbc Manages Transactions

kubrick

- The main mechanism that pyodbc uses to control how it handles transactions is the **connection** autocommit property
- You can set autocommit to True – in which case pyodbc will commit each statement as it is submitted (and you effectively have no control over transactions in pyodbc)
- If you set autocommit to False (which is the default) then
  - when you open a cursor there is an implicit “begin transaction”
  - you can perform some work in the cursor in a bundle
  - you can issue a commit statement to save all of your work bundle to the database
  - if you detect an error you can issue a rollback statement to rollback all of the bundle of work

There is a feature of the way pyodbc handles transactions that could catch you out. Transactions are handled at the **connection** level but there are cusor.commit() and cursor.rollback() methods. These cursor method affect database transactions at the connection level and they are functionally identical to the connection commit and rollback methods. If you have several cursors generated from one connection and you issue a commit – the work in ALL of the cursors gets committed (or rolled back if you issue a rollback). This might not be what you were expecting!

Demonstration with connection autocommit True and False

kubrick

- Here we show the effect on a session when the connection autocommit is set to True and when autocommit is set to False

demo_db_txs.py > ...

```
25  # We can use a python context manager with pyodbc.connect()
26  with  pyodbc.connect(connectString) as cnxn:
27      # Create a cursor object from the connection
28      cursor = cnxn.cursor()
29      cursor.setinputsizes([(pyodbc.SQL_WVARCHAR, 50, 0)])
30      #autocommit False by default
31      print(f"Autocommit is {str(cnxn.autocommit)}\n")
32
33      # Create a new table with special offer items in it
34      cr_tab = '''CREATE TABLE [production].[promo_products](
35      [promo_id] [int] IDENTITY(1,1) NOT NULL,
36      [product_id] [int] NOT NULL,
37      [product_name] [varchar](255) NOT NULL,
38      [brand_id] [int] NOT NULL,
39      [category_id] [int] NOT NULL,
40      [model_year] [smallint] NOT NULL,
41      [list_price] [decimal](10, 2) NOT NULL,
42      PRIMARY KEY CLUSTERED
43      (
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
About to creat table Press return to continue

About to poulate table Press return to continue

About to Rollback Press return to continue

About to DROP table Press return to continue

About to Rollback Press return to continue

Press return to End
PS Microsoft.PowerShell.Core\FileSystem::\\wsl$\Ubuntu-20.04\home\djbadm\src\SQL_DB_Access_Python>
```

Setting autocommit to True – one last Gotcha !

kubrick

If you set the autocommit connection property to True there is one further thing to be aware of. If you use the executemany() method on a cursor and the connection fast_executemany is set to False (as is the default). Then each parameter set is sent as a separate SQL statement (with its own implicit commit as autocommit is set to True)

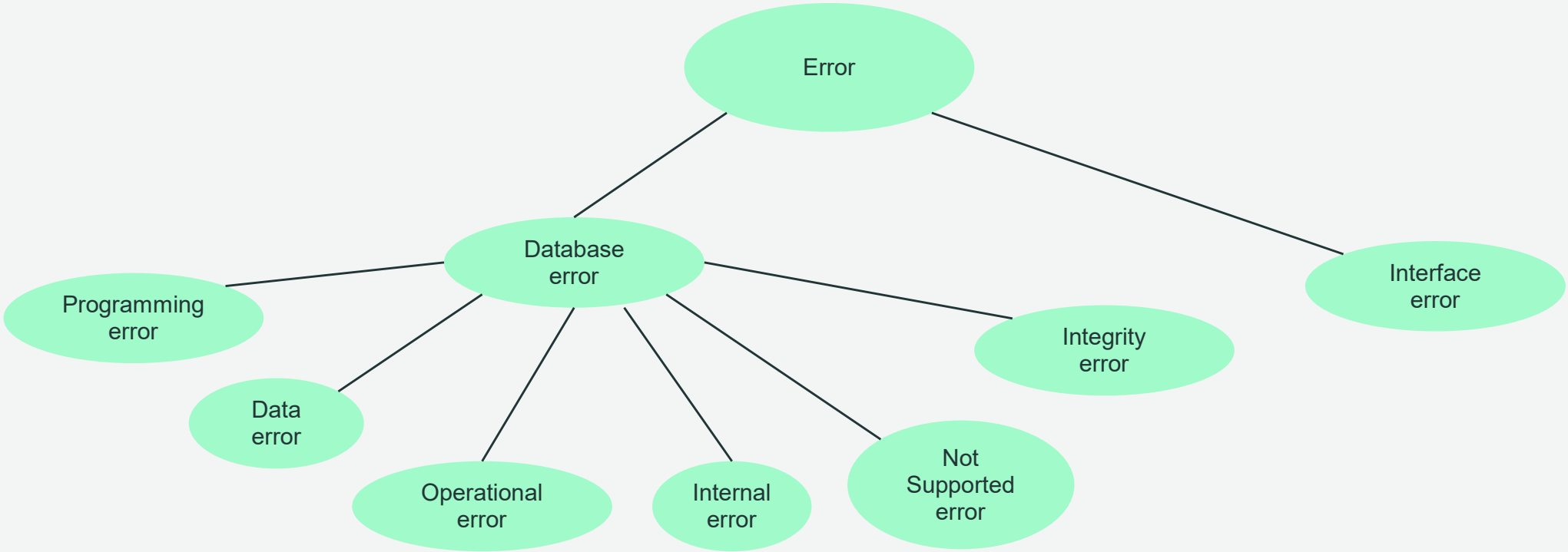
If a database failure occurs during the operation – it can be very difficult to tell what has been committed and what part of the batch has not.

Specifically, for Microsoft SQL Server, if you have to use the connection autocommit to true then you should also set fast_executemany to True in the connection

Handling Errors in pyodbc

kubrick

The pyodbc driver uses the standard pattern of raising exceptions for various errors that may occur when connecting to and interacting with the database. The hierarchy of error classes is as follows



Handling Errors try except finally pattern

kubrick

Errors can be captured and handled using the standard Python try, except catch blocks with optional finally blocks to clear up after failures

The pattern is generally as follow

```
try:
        some database operation

except pyodbc.ErrorType1 as pyodbcerror:
        take some suitable action

except pyodbc.ErrorType2 as pyodbcerror:
        take some suitable action

except pyodbc.ErrorType2 as pyodbcerror:
        take some suitable action

finally:
        optional clean up (if you have to terminate etc)
```

Might want to have multiple except blocks to cope with multiple classes of error

Handling Errors Demo

kubrick

We will look at how we use the try, catch blocks to handle errors coming back from the database

demo_local_error_handling.py > ...

```
database = "BikeShop"

connectString = 'DRIVER={ODBC Driver 18 for SQL Server};SERVER='+\
hostname+';DATABASE='+database+';UID='+\
user+';PWD='+psw+';TrustServerCertificate=yes'

input(" shutdown the local SQL database!")
try:
    cnxn = pyodbc.connect(connectString)
except pyodbc.OperationalError as pyodbcerror:
    print(f"Error {str(pyodbcerror)}")
    input("\nStart the database now  Press return to continue")

# startup the local database and get the connection

cnxn = pyodbc.connect(connectString)
# Create a cursor object from the connection
cursor = cnxn.cursor()
cursor.setinputsizes([(pyodbc.SQL_WVARCHAR, 50, 0)])
#autocommit False by default
print(f"Autocommit is {str(cnxn.autocommit)}\n")
input("\nEstablished a connection  Press return to run disconnect demo\n")
print("Database Disconnect Demo")
sql = "WAITFOR DELAY '00:01:30';"
try:
    cursor.execute(sql)
    rows = cursor.fetchall()
```

Exercise Ex4
Handling Errors and managing transactions

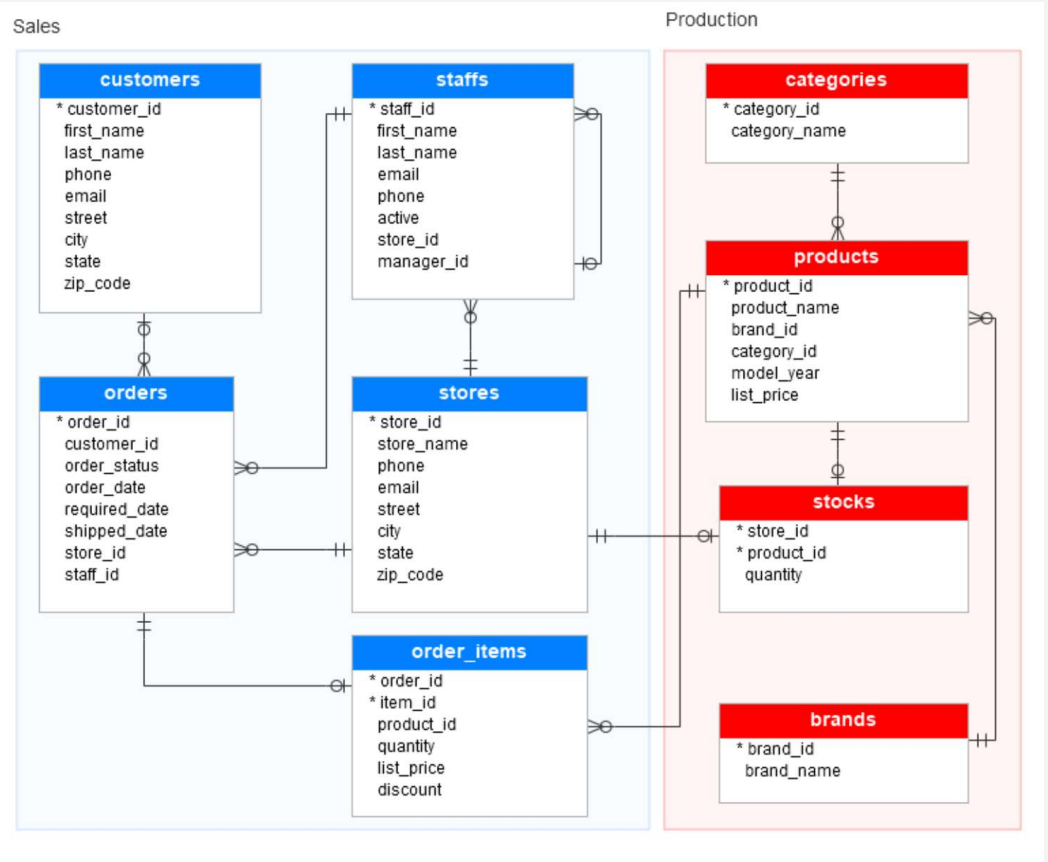
Work in your pods and use the bikestore tables we created earlier

Write a customer transaction simulation: The simulation should behave like an order processing system.

Create new orders for existing goods in the production.products tables. The orders can come from existing customers (to make things a little simpler) and are for existing products.
When new orders come in you should update the number of items from the stocks table and you should allocate a salesperson and a store to the order.
You should record the new orders in the sales.orders and order items tables.

Use the order_status field to signify whether an order has been placed , fulfilled , cancelled etc – you may have to create a reference table for this.

Simulate some orders failing due to lack of stock, payment declined, etc . Simulate failures and use transaction rollbacks to tidy up after failing transactions.
Decide as a team which parts of the solution to perform in python and which in SQL



SQL Alchemy

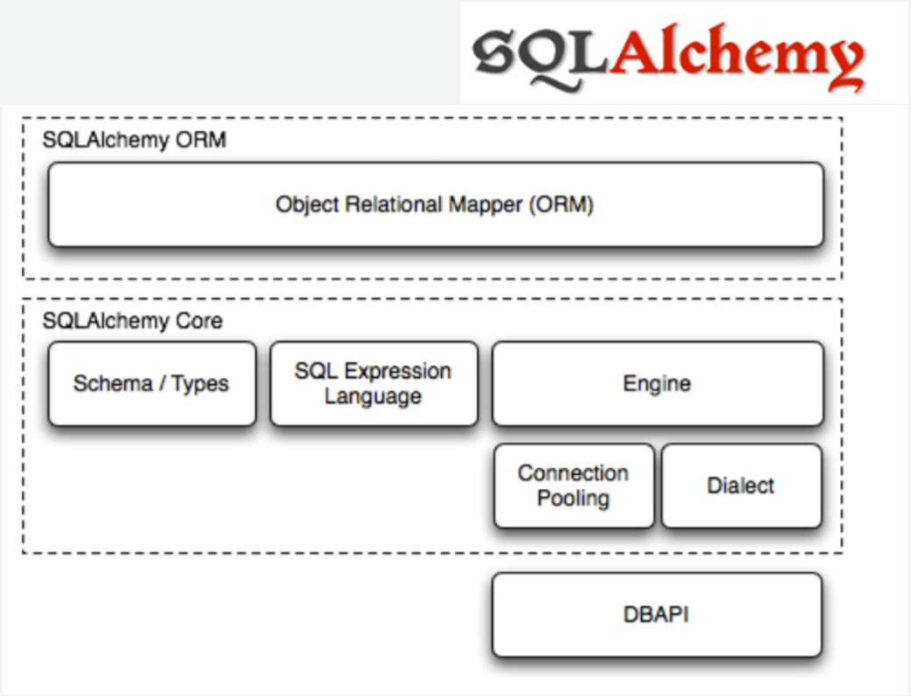
kubrick

SQL Alchemy is a higher abstracted API than pyodbc and it sits on top of the DBI 2.0 specification.

It and provides more insulation against differing Database vendors than is possible with pyodbc. It has dialects to cope with the differing features of SQL databases. It has a metadata language to describe database objects and queries to facilitate this.

SQL Alchemy provides connection pooling and allows for larger applications requiring multiple concurrent connections to manage their database access more easily

SQL Alchemy engine can be used by Pandas to rapidly load dataframes from database tables and vice-versa.



Further Reading

kubrick

Michael Kleehammer's wiki pages https://github.com/mkleehammer/pyodbc/wiki

SQLAlchemy – Python relational toolkit and Object Relational Mapper
https://www.sqlalchemy.org

kubrick

Questions

11 April 2023