QuickStart Guide to Db2 Development with Python

Roger E. Sanders

Contents

CHAPTER 1

What Is IBM Db2?

In 1969, while working at IBM's San Jose Research Laboratory in San Jose, California, Edgar Frank "Ted" Codd introduced a new concept for managing information in a paper titled "*A Relational Model of Data for Large Shared Data Banks*." And, over the next four years, a group of researchers at IBM worked to create a data management system prototype that was based on the relational model described in Codd's paper. (This prototype was named *System R*—short for *System Relational*.) Recognizing that a new language was needed to interact with System R, Codd published a second paper titled "*A Data Base Sublanguage Founded on Relational Calculus*" in 1971; this paper became the foundation for a new language called *DSL/Alpha*, which was quickly renamed *SEQUEL* (for *Structured English QUEry Language*) and later shortened to *SQL*—an acronym for *Structured Query Language*.

Note: Although the SQL language was developed in the early 1970s by IBM researchers Raymond Boyce and Donald Chamberlin, it was not made publicly available until a company called Relational Software—which later became Oracle—commercially released its own relational database management system (RDBMS) and version of SQL called *Oracle V2* in 1979.

In 1986, SQL became an American National Standards Institute (ANSI) standard, and in 1987, it became an International Organization for Standardization (ISO) standard. Over the years, these standards have been revised to include a larger set of features; however, despite the existence of standards, RDBMS vendors often modify the language to some extent. Consequently, SQL code written for one database may have to be altered—or completely rewritten—before it can be used with another.

As part of an effort to port the System R prototype to its System/370 mainframe, IBM began work on a new product called *DATABASE 2* (or *DB2*) in 1980. And, on June 7, 1983, DB2 was made available to a limited number of customers. Two years later it became generally available to customers running the *Multiple Virtual Storage* (*MVS*) operating system on System/370 servers.

Thus, IBM Db2 is an RDBMS that is based on Codd's relational model; it is a data management system that consists of a set of formally described data objects that are used to store and link data values by establishing some type of relationship between them. Typically, data is organized into *tables*, which are user-defined objects that present data as a collection of unordered *rows* with a fixed number of *columns*. (Each column contains values of the same data type, and each row contains a set of values for one or more columns; the representation of a row is called a *record*, the representation of a column is called a *field*, and the intersection of a row and column is referred to as a *value* or *cell*.) This results in a very efficient way to store data, as values only have to be stored once: data in tables can be accessed and assembled in a variety of ways to facilitate any number of operations.

Note: Tables are just one type of object that can exist in an RDBMS. A Db2 database consists of many physical and logical components, all of which aid in the storage, modification, and retrieval of data.

Since its introduction in 1983, new features and functionality have been added in every new release (and many times, with fix packs as well). As a result, Db2 has become an enterprise-level, high-performant RDBMS

that delivers both the four basic functions of persistent storage (create, retrieve, update, and delete, or *CRUD*) *and* the four properties (atomicity, consistency, isolation and durability, or *ACID*) that guarantee data validity, even when unexpected incidents occur.

From the Mainframe to Distributed Platforms and the IBM Cloud

In 1987, DB2 arrived on the personal computer (PC) in the form of an offering called *Database Manager*, which was one of two special add-on products that came with the "Extended Edition" version of OS/2 (a new operating system for PCs that IBM released that same year). A year later, a version for IBM's new AS/400 server emerged in the form of *SQL/400*. And, by 1992, DB2 had become a standalone product for OS/2 (that was renamed *DB2/2*). In 1993, DB2 became available to customers running AIX on IBM RS/6000 series servers. Initially, this port was known as *DB2/6000*, but DB2/2 and DB2/6000 were quickly merged to create a single product that was christened *DB2 for Common Servers*. This flavor of DB2 arrived on HP-UX and Solaris servers in 1994, on Windows servers in 1995, and on Linux servers in 1999. Along the way the name changed yet again (to *DB2 Universal Database* or simply *DB2 UDB*). And, when Version 9 was made generally available in mid-2006, the "Universal Database" moniker was replaced with the names of the predominant operating systems the DB2 software ran on, leaving essentially just two flavors of DB2: *DB2 for z/OS* and *DB2 for Linux, UNIX, and Windows* (otherwise known as *DB2 LUW*).

In 2014, IBM launched a fully managed, cloud-based DB2 data warehouse offering called *dashDB*, along with a hosted database service named *DB2 on Cloud*. And, a fully managed, transactional version of dashDb named *dashDb for Transactions* soon followed.

Db2 Today

On June 22, 2017, as part of the release of Version 11.1, Modification Pack 2, Fix Pack 2 (v11.1.2.2), IBM rebranded all of its DB2 and dashDB offerings to create the following set of products:

- **Db2** (formerly DB2 LUW)
- **Db2 for z/OS** (formerly DB2 for z/OS)
- **Db2 Hosted** (formerly DB2 on Cloud)

- **Db2 on Cloud** (formerly dashDB for Transactions)
- **Db2 Event Store** (a new an in-memory database optimized for event-driven data processing)
- **Db2 Warehouse on Cloud** (formerly dashDB)
- **Db2 Warehouse** (formerly dashDB Local)
- **IBM Integrated Analytics System** (a new a hardware and software platform that combines the analytic performance and functionality of the IBM PureData® System with IBM Netezza®)

Data can be processed by any number of applications running concurrently against these products. And, while the manipulation of data in a Db2 database is still accomplished primarily with SQL—either by embedding it in a C, C++, or Java source code file, or by running it using Db2-specific tools like the Db2 Command Line Processor (CLP)—a variety of driver interfaces can be used to develop Db2 applications, including:

- Google Go
- Python—*the language, the Python Database Interface (Python DBI), Django, and SQLAlchemy*
- Node.js
- Sequelize Object Relational Mapping (ORM) for Node.js
- Java Database Connectivity (JDBC)
- PHP: Hypertext Preprocessor (PHP)
- Ruby on Rails
- Practical Extraction and Report Language Database Interface (Perl DBI)
- Microsoft .NET
- The Open Group Call Level Interface (CLI)
- Microsoft Open Database Connectivity (ODBC)
- Microsoft Active X Data Objects (ADO)
- Microsoft Object Linking and Embedding for Databases (OLE DB)
- Open Data Protocol (OData) Representational State Transfer (REST) application program interface (API)

Because the Db2 offerings available today—with the exception of Db2 for z/OS—share a common SQL engine, applications created for one platform (such as Db2) can be ported to another (for instance, Db2 Warehouse on Cloud) without requiring code modifications.

CHAPTER 2

Structured Query Language (SQL)

Structured Query Language (*SQL*) is a standardized language that is used to work with database objects and the data they contain. SQL consists of several different *statements* that can be used to define, alter, and delete database objects as well as insert, update, delete, and retrieve data values. Like other programming languages, SQL has a specific syntax and its own set of language elements. However, because SQL is non-procedural by design, it is not an actual programming language. (SQL statements are executed by an RDBMS engine, not the operating system.) Consequently, most applications that use SQL are constructed by combining the decision and sequence control of a high-level programming language (interpreted or compiled) with the data storage, manipulation, and retrieval capabilities SQL provides.

SQL statements are frequently categorized according to the function they have been designed to perform; five different categories are typically used:

- **Embedded SQL Application Construct statements**: Used solely for constructing embedded SQL applications. Some of the embedded SQL application construct statements that are recognized by Db2 are **BEGIN DECLARE SECTION**, **END DECLARE SECTION**, **INCLUDE**, and **WHENEVER**.

- **Data Control Language (DCL) statements**: Used to give (grant) and take away (revoke) authorities and privileges. Authorities convey the right to perform high-level administrative and maintenance/utility operations on a Db2 instance or database; privileges convey the right to perform certain actions against specific database objects (like tables, indexes, and views). The most common DCL statements are **GRANT** and **REVOKE**.
- **Data Definition Language (DDL) statements**: Used to create, alter, and delete individual database objects. The DDL statements that are used the most are **CREATE**, **ALTER**, and **DROP**.
- **Data Manipulation Language (DML) statements**: Used exclusively to store data in, modify data in, remove data from, and obtain data from select tables and/or views. The most common DML statements used are **INSERT**, **UPDATE**, **DELETE**, and **SELECT**.
- **Transaction Management statements**: Used to establish and terminate database connections and transactions. A *transaction* (also known as a *unit of work*) is a sequence of one or more SQL operations that are grouped together as a single unit. Such a unit is considered *atomic* (from the Greek word meaning "not able to be cut") because it is indivisible: either all of a transaction's work is carried out, or none of its operations performed are made permanent. The transaction management statements that are recognized by Db2 are **CONNECT**, **ROLLBACK**, and **COMMIT**.

One of the simplest ways to construct applications that require SQL is to use a methodology known as *embedded SQL programming*. As the name implies, embedded SQL applications are built by coding SQL statements directly into high-level programming language source code files. Unfortunately, this approach has some drawbacks. For one thing, high-level programming language compilers do not recognize, and therefore cannot interpret, SQL statements. And, some RDBMS engines cannot work directly with high-level programming language variables: instead, special variables known as *host variables* must be used to move data between an application and the RDBMS engine. (This is the case with Db2.) Therefore, source code files containing embedded SQL statements must be preprocessed with an *SQL precompiler* before they can be compiled.

Note: Db2 ships with an SQL precompiler that will convert SQL statements found in a source code file into comments and generate appropriate runtime API calls to replace them. It also evaluates the data types of declared host variables and determines which data conversion methods are needed to move data in and out of those variables.

Furthermore, embedded SQL applications lack interoperability. Because RDBMS vendors do not strictly adhere to SQL language standards, embedded SQL applications developed for one RDBMS will, in all likelihood, have to be modified before they can be used with another.

Fortunately, there is another way to construct applications that must perform SQL operations. And, while the process may not be as straightforward as embedding SQL statements in a source code file, it is almost as simple.

CHAPTER 3

The Db2 Call Level Interface

To overcome many of the challenges associated with embedded SQL programming, the X/Open Company, together with the SQL Access Group (SAG), which is now a part of X/Open, jointly developed a standard specification for a callable SQL interface in the early 1990s. This interface, known as the *X/Open Call-Level Interface* (or *X/Open CLI*), defined, in a consistent way, how an application should send SQL to a RDBMS for processing and how it should handle any result sets returned. Its primary purpose was to increase the portability of database applications by allowing them to become independent of any one RDBMS's SQL language.

In 1992, Microsoft Corporation developed its own callable SQL interface, known as *Open Database Connectivity* (*ODBC*). Based on a preliminary draft of the X/Open CLI specification but providing more functionality and capability, ODBC utilizes an architecture in which data source-specific ODBC libraries (known as *drivers*) are dynamically loaded and unloaded at application runtime by a component known as the *ODBC Driver Manager*. Each driver is responsible for processing ODBC API function calls, submitting SQL requests to a data source, returning results from that data source, and if appropriate, modifying an application's request so it conforms to the SQL syntax supported by the data source used.

Note: To ODBC, a *data source* consists of the data an application wants to get access to and its associated RDBMS, together with the operating system and network platform (if any) that is used to communicate with the RDBMS.

Applications call ODBC APIs to submit SQL statements and retrieve results, and API calls are sent to the ODBC Driver Manager, where they are examined and routed to the appropriate ODBC driver for processing. It's important to note that ODBC APIs are used in two places: between an application and the ODBC Driver Manager, and between the ODBC Driver Manager and a driver. Consequently, an application written for one RDBMS can be executed against another without having to be altered: it simply has to establish a connection to the new RDBMS (which will cause the driver for the new RDBMS to be loaded, dynamically). Because multiple drivers and data sources can exist, a single application can interact with multiple data sources, simultaneously. And, since ODBC is not limited to Microsoft operating systems (other implementations are available on a variety of platforms), applications running on one platform can easily work with an RDBMS that is running on another.

The Db2 *Call Level Interface* (or *Db2 CLI*) is IBM's callable SQL interface to its Db2 product offerings. Written in C and C++, it is an application programming interface (API) for database access that uses function calls to pass dynamic SQL statements (as arguments) to a Db2 data source for processing. The Db2 CLI is based on both the Microsoft ODBC specification and the International Standard for SQL/CLI (specifically, ISO/IEC 9075-3:2003). These specifications were chosen in an effort to follow industry standards and to provide a shorter learning curve for application developers who are already familiar with using one of these interfaces. Thus, the Db2 CLI driver provides support for all of the ODBC 3.51 core APIs (except the **SQLDrivers()** API), all ODBC Level 1 and Level 2 functions, some X/Open CLI-specific functions that are not supported by ODBC, and some Db2-specific APIs that enable application developers to take advantage of features and functionality that are only available with Db2.

Applications that only work with Db2 data sources can link directly to the Db2 CLI driver. And, any ODBC Driver Manager can dynamically load this library as if it were an ODBC driver. However, it is important to note that when an application uses the Db2 CLI driver independently, it cannot communicate with anything other than Db2 data sources. Figure 3.1 illustrates how applications use the Db2 CLI driver in a Db2-only environment; Figure 3.2 illustrates how applications use the Db2 CLI driver in an ODBC Driver Manager environment.

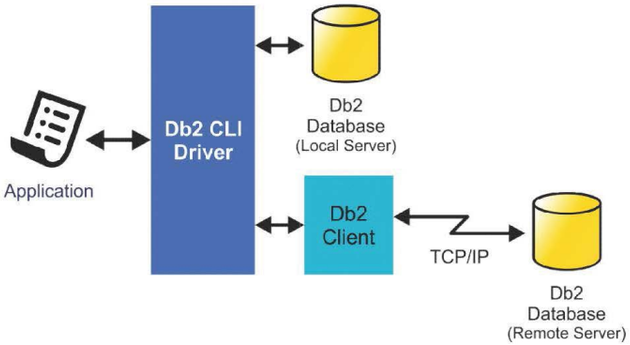

Figure 3.1: How the Db2 CLI driver is used in a Db2-only environment

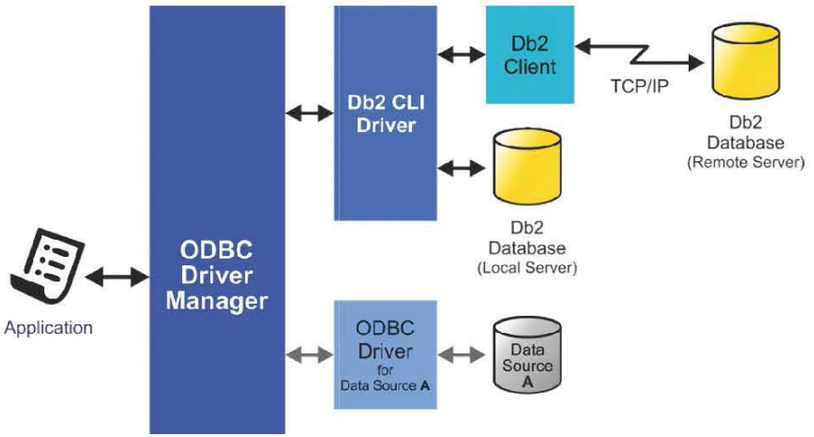

Figure 3.2: How the Db2 CLI driver is used in an ODBC environment

Because Db2 CLI/ODBC applications rely on a standardized set of APIs to execute SQL statements and perform database-related operations, the way in which they are constructed differs from when embedded SQL is used:

- Db2 CLI/ODBC applications do not require the explicit declaration and use of host variables; any variable can be used to send data to or retrieve data from a connected data source.
- Cursors do not have to be explicitly created (declared) by Db2 CLI/ODBC applications. Instead, they are automatically generated whenever they are needed. (When a **SELECT** statement is executed from within an application program, Db2 uses a mechanism known as a *cursor* to retrieve data values from the result data set produced. The name "cursor" probably originated from the blinking cursor found on early computer screens. And just as that cursor indicated the current position on the screen, a Db2 cursor points to the current position in a result data set—i.e., the current row).
- Cursors do not have to be explicitly opened in Db2 CLI/ODBC applications; they are implicitly opened the moment they are created.
- Db2 CLI/ODBC functions treat environment (server), connection, and SQL statement-related information as abstract data objects. This eliminates the need to use RDBMS product-specific data structures (such as the Db2 SQLCA data structure).
- Db2 CLI/ODBC applications inherently have the ability to establish multiple connections to multiple data sources or to the same data source, simultaneously.
- Because **ROLLBACK** and **COMMIT** statements can be dynamically prepared by some data sources but not others, they are not used in Db2 CLI/ODBC applications. Instead, Db2 CLI/ODBC applications must use an API to end active transactions (unless AUTOCOMMIT behavior has been enabled).

Despite these differences, one important common concept exists between embedded SQL applications and Db2 CLI/ODBC applications: *Db2 CLI/ODBC applications can execute any Db2 SQL statement that can be dynamically prepared and executed.*

The Db2 CLI driver serves as the foundation for many open source programming languages, including Python. Consequently, many of the APIs found in IBM's **ibm_db** Python library are strikingly similar to those found in the Db2 CLI driver. (We'll look at the **ibm_db** library shortly.) And, you will discover that when a call to an API in this library fails, any error message returned will begin with the text "[IBM] [CLI Driver]".

CHAPTER 4

Python and Db2

Python is a high-level, general purpose, scripting language that can be used for a wide variety of programming tasks. Named after the British comedy television show *Monty Python's Flying Circus*, Python was created by Guido Van Rossum at the National Research Institute for Mathematics and Computer Science in 1989. (The first Python interpreter was developed as a hobby; the second was made generally available on October 16, 2000.) Since then, its following has steadily grown, and according to the March 2019 TIOBE Programming Community Index, it is the third most popular programming language in use today.

Note: The TIOBE Programming Community index, updated once a month, is a popularity indicator of programming languages. Ratings are based on the number of skilled engineers (worldwide) who claim to be using a language, the number of courses available, and the number of third-party vendors that provide related offerings. The frequency at which a language is searched for on popular websites like Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube, and Baidu are used to calculate popularity ratings as well.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and UNIX shells like ksh and bash. Consequently, it offers a variety of basic data types: numbers (floating point, complex, and unlimited-length long integers); strings (ASCII and Unicode); lists; tuples; and dictionaries. It also provides a number of built-in functions, as well as several constructs (for example, a loop construct that can iterate over items in a collection is available). In addition, Python comes with a broad standard library that consists of hundreds of modules that can be used to perform many common programming operations, such as connecting to a Web server, searching text using regular expressions, and reading or modifying files.

Other features that Python provides include:

- Support for functional and structured programming methods as well as object-oriented programming (i.e., classes, objects, and inheritance)
- Automatic memory management (i.e., "garbage collection")
- Support for raising and catching exceptions (which makes error handling cleaner)
- Strong, dynamic data types and data type checking. (The mixing of incompatible data types—for example, trying to add a string and a number—will cause an exception to be raised.)
- Ability to group chunks of code into separate modules and packages

Because Python is an interpreted language, it is processed at runtime by a Python interpreter. From a coder perspective, applications written in Python do not have to be compiled before they can be executed. However, it's important to note that Python programs are automatically compiled into bytecode before execution, and in most cases, this bytecode is saved to disk so that compilation only needs to happen again if the program's source code is altered. The Python interpreter itself can be run in interactive mode, making it easy to test short code snippets, build prototypes, and perform other types of ad hoc programming. In addition, users can add their own low level-modules to the Python interpreter, if desired.

So, why is Python so popular? For one thing, it's open source. This means that anyone can download it for free (from *www.python.org*) and use it to develop applications. This also means that the source code for the Python interpreter can be accessed and modified as desired. Furthermore, this can be done on a wide variety of platforms: Python can be used on Microsoft Windows, macOS, and most Linux distributions. As a result, applications written on one platform can easily be executed on another.

Another thing that makes the language appealing is that Python code is easy to read, which makes it easy to learn and maintain. Python has a relatively simple structure, and its syntax is clearly defined; it uses English keywords where other languages rely on punctuation, so there are no curly braces to balance or semicolons to forget. (Some programming languages require lines to be terminated with semicolons.) Instead, indentation is used to mark where code blocks begin and end. Python also uses libraries to provide functionality, which helps keep the core language simple and lightweight. Developers only need to add a basic set of libraries to their code to get the functionality they desire.

Finally, Python is flexible. It can be used with functional, object-oriented, and imperative coding styles, making it useful to different types of programmers. And, it can be used to build a wide variety of applications: it can be used solely as a scripting language, it can be embedded, or it can be compiled and integrated with C, C++, COM, ActiveX, CORBA, and/or Java. Coders trying to choose the best programming tool for the job find that Python is flexible enough to be considered in many situations.

Installing the Python Interpreter

To program in Python, you need the Python interpreter. And, if you are using Linux, there's a good chance the interpreter is already installed. However, if you are using a different operating system, you may have to install it yourself.

To determine whether the Python interpreter is already installed on a Linux workstation, open a terminal window and type the following command:

```
python --version
```

If you see something like `Python 2.7.5`, the Python interpreter is installed on your system, and Python 2.7 is the default version used. If that is the case, you should also check to see if Python 3.*x* is installed. You can do this by executing the following command instead:

```
python3 --version
```

If you see something like Python 3.4 (or earlier), an older version of the 3.*x* interpreter is installed and you should probably upgrade to the latest version.

Important: Python version 3.0 was released on December 3, 2008. However, the Python developers decided **NOT** to make this version **backward-compatible** with the version 2.*x* line of releases, as certain features like the handling of Unicode strings were deemed too unwieldy or broken to be carried forward. However, version 2.*x* flavors of the Python interpreter are still around. Considering that at the time of this writing, the latest version of the Python interpreter is 3.7, you should consider upgrading your applications and infrastructure if you are still using version 2.*x*.

If the Python interpreter cannot be found on your system, you can install it by going to *https://www.python.org/downloads/* and following the directions provided for the platform you are using. (If you download and install the ready-made Python interpreter for your platform, it's a good idea to download the source code as well. This will allow you to browse the standard library and take advantage of the collection of demos and tools that come with the code. There's a lot you can learn from the source!)

IBM Db2 Support for Python

Several resources are available to help developers build Python applications that interact with IBM Db2 servers and databases:

- **The ibm_db driver/library**: This Python driver uses the IBM Data Server Driver for ODBC/CLI driver to connect to and interact with IBM Db2 and Informix servers and databases. APIs in this library can be used to perform advanced operations that cannot be done with the other Python drivers available.
- **The ibm_db_dbi driver/library**: This Python driver provides functionality that conforms to the *PEP 249 — Python Database API Specification v2.0*. Consequently, it does not offer some of the more advanced features that are available with the **ibm_db** driver.

However, if you have an application that uses this driver, you can easily switch to the **ibm_db** driver since both the **ibm_db** and the **ibm_db_dbi** drivers are packaged together. (You can learn more about the PEP 249 specification at https://www.python.org/dev/peps/pep-0249/.)

- **The ibm_db_sa adapter**: This adapter supports SQLAlchemy, which is a popular open source Python SQL toolkit and object-to-relational mapper (ORM). (Only SQLAlchemy 0.7.3 and later are supported.)
- **The ibm_db_django adapter**: This adapter provides access to IBM Db2 and Informix servers and databases from Django. Django is a popular Web framework that can be used to build high-performing, elegant Web applications quickly.

Figure 4.1 illustrates how APIs in each of these resources can be used to interact with IBM Db2 and Informix servers and databases. In Chapter 5, "Building a Python-Db2 Development Environment," we'll see how to install the **ibm_db** and **ibm_db_dbi** drivers/libraries; in Chapter 6, "Building Db2 Applications with Python," we'll see how they are used.

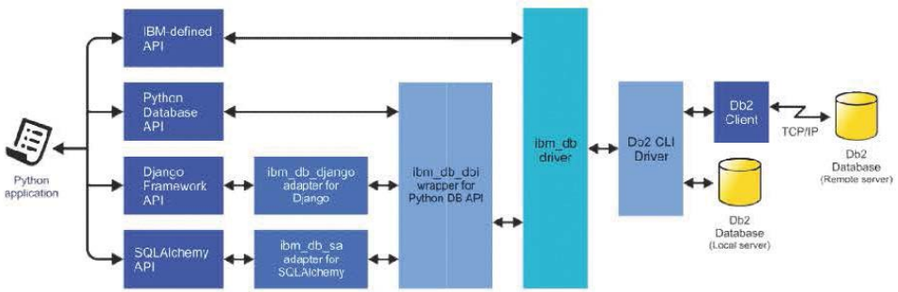

Figure 4.1: IBM Db2 support for Python

CHAPTER 5

Setting Up a Python-Db2 Development Environment

The process of setting up an application development environment is frequently overlooked because the environment used is often dictated by the company a developer works for. Even developers who don't work with an environment that is governed by their company may not have to do much to get started; they may only need to install their integrated development environment (IDE) of choice (such as Xcode or Visual Studio Code). But creating a new development environment from the ground up can be frustrating because not everybody's computer is the same. Consequently, a setup tutorial for one computer—or a process used by one developer—may not work for another.

For this reason, this chapter does not cover the exact links and clicks that are required to set up a computer for application development—the right Google searches can help you with that. Instead, it focuses on how to get Db2 up and running on a Linux server (Linux was chosen because I find it to be an easy operating system to use for Python development), how to create a populated database to work with, and how to install the libraries necessary to build Python applications that can interact with Db2.

A Word About *My* Development Platform: the UDOO X86

Before we look at the development environment setup process, I want to spend a few moments talking about the hardware and software I have been using over the past two years to create application development environments: a UDOO X86 Ultra single-board computer (SBC) running Windows 10, Red Hat Enterprise Linux (RHEL7.5), and Ubuntu Linux 18.04, together with Db2 Developer-C (now Db2 Community).

On June 22, 2017, with the release of Db2 v11.1.2.2 (Version 11.1, Modification Pack 2, Fix Pack 2), IBM introduced a new, free, full-featured version of Db2 known as *Db2 Developer-C Edition*. Before that, a free version of Db2 called Db2 Express-C was available, but that offering did not allow users to take advantage of features like range-partitioned tables, high-availability disaster recovery (HADR), and BLU Acceleration (columnar, in-memory tables). Like Db2 Express-C, Db2 Developer-C has some restrictions: no more than 4 cores/CPUs and 16 GB of RAM can be used, and databases cannot exceed 100 GB (compressed) in size. Unlike Db2 Express-C, these restrictions can be lifted simply by applying a new license key.

Note: Initially, Db2 Developer-C could only be used in non-production environments. However, that restriction was removed when v11.1.3.3 was released. Unfortunately, at the time of this writing, the Db2 Knowledge Center (which contains the Db2 product documentation) and many Db2 websites haven't been updated to reflect this change.

Six weeks before Db2 Developer-C was made generally available, I started looking for an SBC with an x86 CPU I could use to create a simple Internet of Things (IoT) project. The goal was to create something like the **IBM TJBot** to tempt new developers to try Db2 Developer-C. (I considered using a Raspberry Pi, but needed a platform with a CPU the Db2 software is compiled for.) That's when I discovered the UDOO X86 (*https://www.udoo.org/udoo-x86/*). Not only was I able to build something that got everyone's attention at the 2017 International Db2 Users Group (IDUG) conference in Lisbon, Portugal (where my project was unveiled),

but I also discovered what has become, at least to me, the ideal application development platform. With a quad-core Intel® Pentium processor, 8 GB of RAM, Intel dual-band wireless Wi-Fi, Bluetooth, and a 128 GB M.2 Solid State Disk (SSD) drive, my UDOO X86 Ultra is a very powerful computer for its size (which is 4.72" x 3.35" or 120mm x 85mm). I have found that by putting install images for different operating systems on a set of color-coded USB thumb drives, I can quickly reimage this machine in a variety of ways to build/rebuild a variety of development environments. (One nice feature of the UDOO X86 is its built-in 32GB eMMC storage. I keep the install code for Db2 and development tools like Anaconda and Visual Studio Code there so they do not have to be reloaded each time my machine is reimaged.) And, in just a few hours I can create a complex, live-demo platform that can be taken anywhere. The UDOO X86 that was used to create the Python examples referenced throughout the remainder of this book can be seen in Figure 5.1.

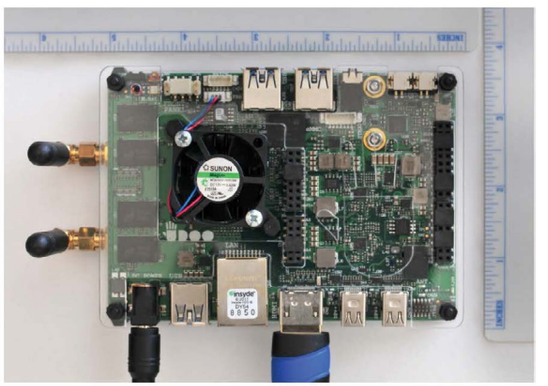

Figure 5.1: My UDOO X86 Ultra Db2-Python development platform (Note: I created a custom mount bracket and added the external Wi-Fi antennas; the acrylic case is an add-on accessory.)

Downloading the Db2 Software

It has been my experience that IBM websites are constantly changing, so I'm hesitant to provide the link to a specific URL where the Db2 Community software can be found. At the time of this writing, the easiest way to get the software is to do a Google search for "Db2 Community download". One of the first links provided in the list returned *should* take

you to the IBM Db2 for Developers Marketplace. (Today, the URL for that website is *https://www.ibm.com/us-en/marketplace/ibm-db2-direct-and-developer-editions*). Select the link that takes you to the Marketplace website; then, locate and click the push button labeled **Download now** (or something similar). Follow the instructions provided—ultimately, you want to arrive at a Web page that looks something like the one shown in Figure 5.2 (this was the download page for Db2 Developer-C).

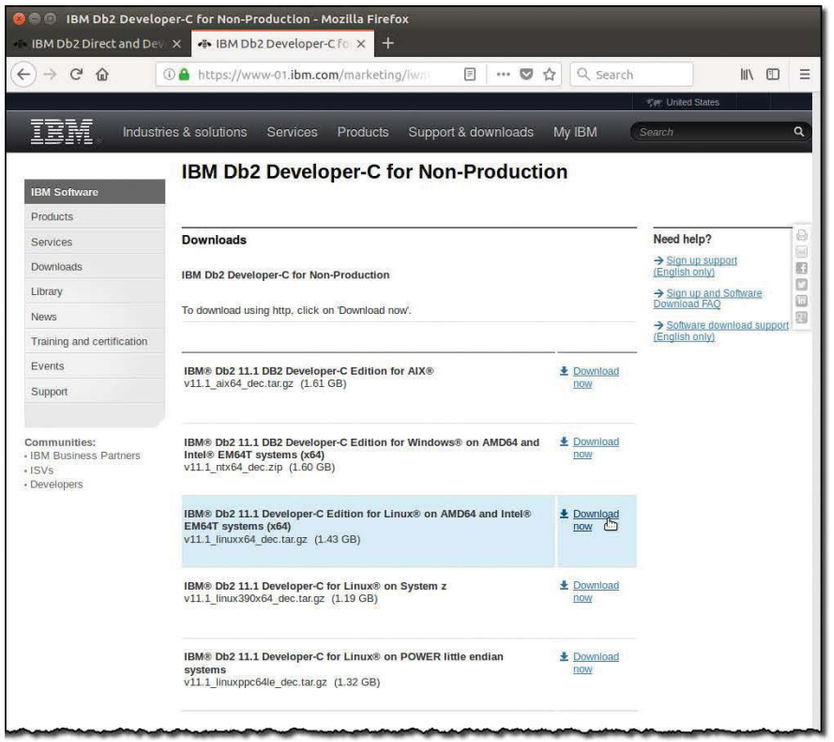

Figure 5.2: IBM Db2 Developer-C for Non-Production Downloads Web page

When you arrive at this page, look for the **⬇ Download now** link located just to the left of text that looks like "**IBM® Db2 11.1 for Linux® on AMD64 and Intel® EM64T systems (x64) v11.1_linuxx64_dec.tat.gz (1.43 GB)**" (similar to the entry highlighted in Figure 5.2). Left-click this link to download the software, taking note of the location where the software is stored.

Important: At the time of this writing, IBM is in the process of renaming Db2 Developer-C to **Db2 Community**. So, you may have to Google the new name to locate the necessary software, and you may find that the Marketplace Web pages have been changed to reflect this new name.

Enabling root User Access (*Ubuntu Linux Setup Only*)

When Ubuntu Linux is installed on a server, **root** login is disabled by default. But ideally, Db2 should be installed under the root user account. So, to enable root user access on an Ubuntu server, perform the following steps:

1. Log in with a user account that has administrative privileges. Usually, this is the account that was created during Ubuntu installation. (I typically use the account name *ibm_admin*.)
2. Open a terminal window. (Right-click anywhere on the desktop and select **Open Terminal** from the menu shown or press **Ctrl + Alt + T**).
3. Execute the following command (from the terminal window):

   ```
   sudo passwd root
   ```

4. When the prompt **[sudo] password for [***AdminUser***]**: (where *AdminUser* is the user account used in Step 1) appears, enter the password for the administrative user account and press **Enter**.
5. When the prompt **Enter new Unix Password:** appears, type a new password for the root user and press **Enter**.
6. When the prompt **Retype new Unix Password:** appears, type the new password again and press **Enter**. You should see the message **passwd: password updated successfully**; if this message does not appear, go through the steps again, making any corrections suggested by the error message(s) returned.

Preparing a Linux Server for Db2 Installation

Before you can successfully install the Db2 software, you need to install several Linux shared library files that Db2 requires. To install these

libraries, as well as update the Linux packages that are currently installed on the system (*Ubuntu Linux only*), perform the following steps:

1. Open a terminal window. (Right-click anywhere on the desktop and select **Open Terminal** from the menu shown.)
2. Log in as the root user by executing the following command:

   ```
   su - root
   ```

   When the **Password:** prompt appears, provide the appropriate password for the root user. The command-line prompt should change from **$** to **#**.
3. If you are using Ubuntu Linux, execute the following commands to install the prerequisite packages:

   ```
   apt-get -y upgrade
   apt-get -y install libaio1
   apt-get -y install binutils
   apt-get -y install zlib1g-dev
   apt-get -y install libpam0g:i386
   apt-get -y install libstdc++6:i386
   apt-get -y install ksh
   apt-get -y install liblogger-syslog-perl
   apt-get update
   ```

   If you are using Red Hat Enterprise Linux, execute the following commands instead:

   ```
   subscription-manager repos –enable rhel-7-server-optional-rpms
   yum -y --noplugins install pam-devel.i686
   yum -y install libstdc++.so.6
   yum -y install ksh
   yum -y install perl-Sys-Syslog
   ```

Important: You must register and activate your Red Hat subscription before you can download and install any additional libraries needed by Db2. (This can be done any time after installation.)

Installing the Db2 Software

Once the required Linux shared library files have been installed and the tarball containing the Db2 software has been downloaded, you need to extract the software from the tarball, and then use the installation program provided (**db2setup**) to install and configure Db2. This can be done by performing the following steps:

1. Open a terminal window (if you don't already have one open).
2. Log in as the root user by executing the following command:

   ```
   su - root
   ```

   When the **Password:** prompt appears, provide the appropriate password for the root user. (Again, the command line prompt should change from **$** to **#**.)
3. Create a directory named "*software*" in the */home* directory and make the directory accessible to everyone by executing the following commands:

   ```
   mkdir /home/software
   chmod 777 /home/software
   ```
4. Move the Db2 Community tarball file that was downloaded earlier to the directory created in the previous step by executing a command that looks something like this:

   mv /*DownloadDir***/v11*.gz /home/software**

   (where *DownloadDir* is the name of the directory the tarball file was stored in).

   In my case, the Db2 Community tarball file is usually stored in the directory */home/ibm_admin/Downloads*, so I execute a command that looks like this:

   ```
   mv /home/ibm_admin/Downloads/v11*.gz /home/software
   ```
5. Go to the */home/software* directory and execute the following commands to unzip, and then untar the file that was just copied there:

   ```
   cd /home/software
   gunzip v11*.gz
   tar -xvf v11.1_linuxx64_dec.tar
   ```

When these commands are executed, a subdirectory named *server_dec* is created in the */home/software* directory, and the files that are used to install Db2 are extracted and stored there.

6. Rename the subdirectory that was created in the previous step (*server_dec*) to *ibm-db2* and make it accessible to everyone by executing the following commands:

```
mv server_dec ibm-db2
chmod -R 777 ibm-db2
```

At this point, you can delete the *.tar* file that was created earlier since it's no longer needed; to remove the file, execute the following command:

```
rm -f v11*.tar
```

7. Move to the */home/software/ibm-db2* directory and verify that the server has everything necessary to install and run Db2 by executing the following commands:

```
cd ibm-db2
./db2prereqcheck -i -v 11.1.4.4
```

Several messages will be generated as different prerequisites for Db2 are checked, and when all checks are completed, you should see the message: **DBT3533I The db2prereqcheck utility has confirmed that all installation prerequisites were met.** If you do not see this message, review the output of the db2prereqcheck utility and resolve any problems identified.

Important: If you are installing a version other than **11.1.4.4**, provide that version number with the db2prereqcheck command instead.

8. Start the Db2 Setup/Installation program by executing the following command:

```
./db2setup
```

When an ***IBM DB2 Welcome*** window like the one shown in Figure 5.3 appears, click **New Install**.

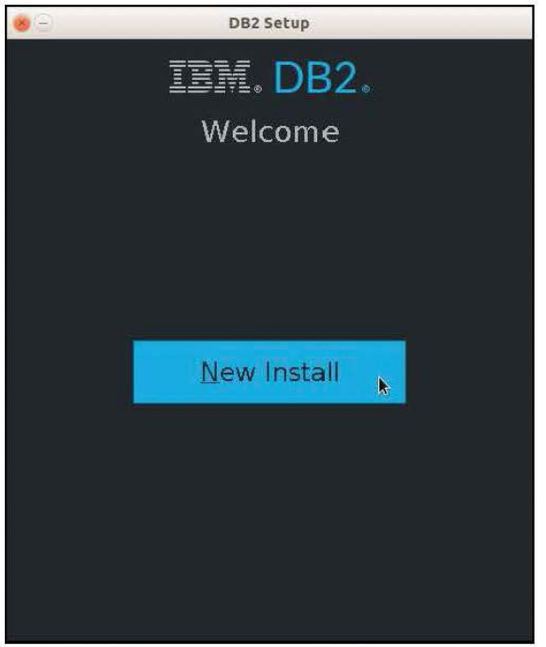

Figure 5.3: IBM Db2 Welcome window

9. A ***Choose a Product*** window will appear (Figure 5.4). Select **DB2 Version 11.1.4.4 Server Editions**, then click **Next**. This launches the Db2 Setup/Install Wizard.

Figure 5.4: IBM Db2 Setup program product selection window

10. On the ***Configuration*** window of the Db2 Setup/Install Wizard, (Figure 5.5), click on the **Click to view link** to see the IBM terms and conditions for using Db2. Then, check the **I agree to the IBM terms** checkbox. Finally, click **Next**.

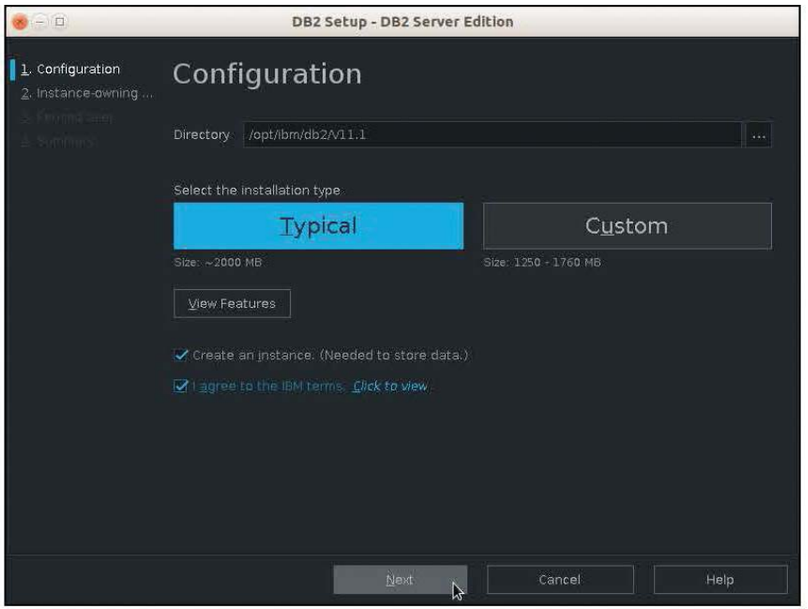

Figure 5.5: IBM Db2 Setup/Install Wizard Configuration window

11. On the ***Instance Owner*** window (Figure 5.6), enter a password for the Db2 instance owner (whose name is *db2inst1*, by default) in the **Password** field. Then, enter the password again in the **Confirm Password** field. (The instance owner username and password provided here is the user ID and password you will use to work with Db2). When finished, click **Next**.

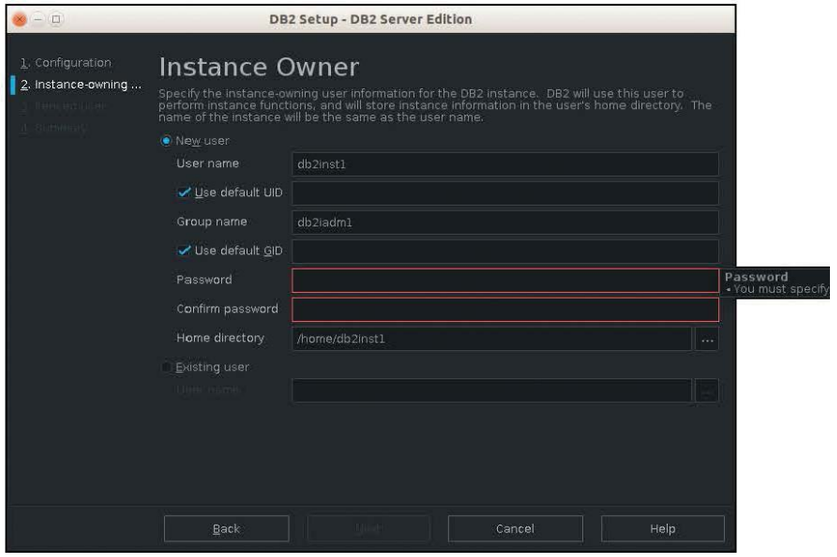

Figure 5.6: IBM Db2 Setup/Install Wizard Instance Owner window

12. On the ***Fenced User*** window (Figure 5.7), enter a password for the Db2 fenced user (whose name is *db2fenc1*, by default) in the **Password** field. Then, enter the password again in the **Confirm Password** field. (The fenced user is used to run user-defined functions and stored procedures outside of the address space that is used by a Db2 database). When finished, click **Next**.

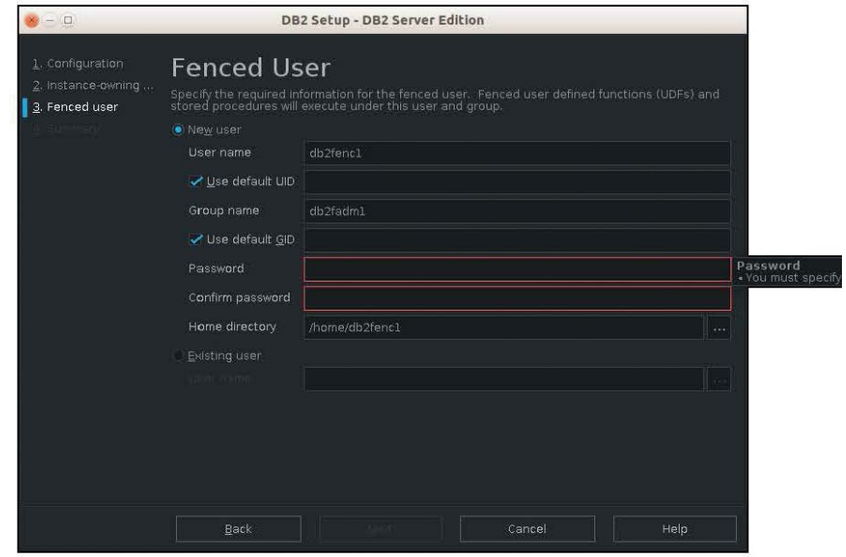

Figure 5.7: IBM Db2 Setup/Install Wizard Fenced User window

13. On the ***Response File and Summary*** window (Figure 5.8), verify that the **Install DB2 Server Edition on this computer and save my settings in a response file** radio button is selected. Then click **Finish** to close the IBM Db2 Setup/Install Wizard and start the Db2 installation process.

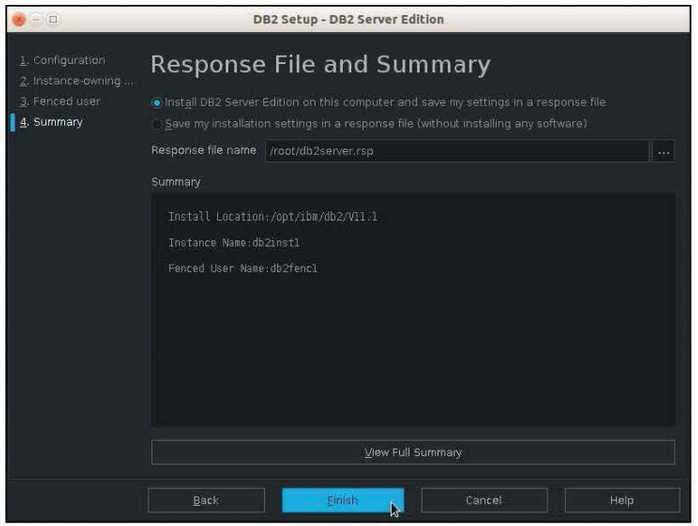

Figure 5.8: IBM Db2 Setup/Install Wizard Response File and Summary window

14. The ***Installing DB2 Server Edition*** window should appear (Figure 5.9). This window contains two status bars, which show the progress of individual tasks and the overall progress of the installation.

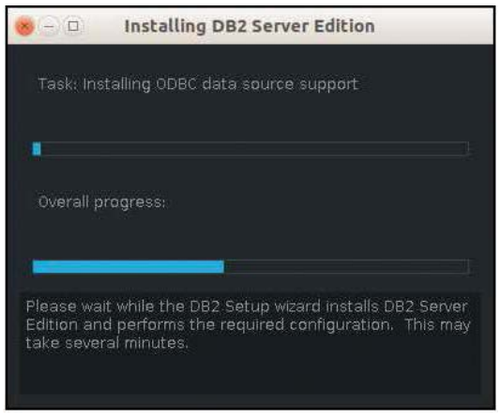

Figure 5.9: IBM Db2 Setup/Installation program window

15. When the Db2 installation process has finished, a ***Setup Complete*** window like the one shown in Figure 5.10 will appear.

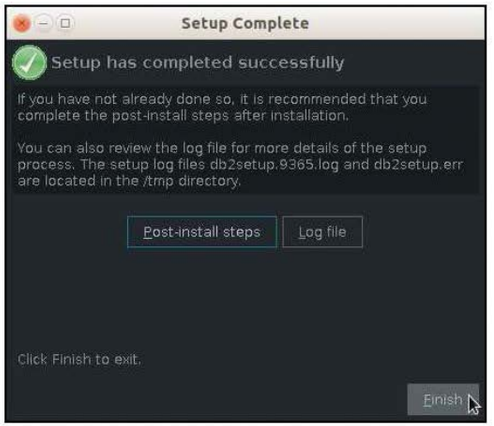

Figure 5.10: IBM Db2 Setup/Installation Program Setup Complete window

16. If you would like to see some recommended actions to take now that Db2 has been installed, click the **Post-install steps** button (refer to Figure 5.10). This will cause a separate ***Post-install steps*** window to appear; information shown in this window will tell you how to:

    - Verify that the Db2 software was installed correctly.
    - View your Db2 license entitlements.
    - Start using Db2.
    - Access the online Db2 documentation.

    (After reviewing the information provided, click **Close** to return to the ***Setup Complete*** window.)

17. If you would like to review the contents of the log file that was generated during the Db2 installation process, click **Log file**. This will open a separate ***Log file*** window that contains log file information; when you have finished reviewing this information in this window, click **Close** (to return to the ***Setup Complete*** window.)
18. Close the ***Setup Complete*** window and terminate the Db2 Setup/Installation program by clicking **Finish** (see Figure 5.10).

Note: With Ubuntu Linux version 17.10 and later, the Db2 instance owner and fenced user that are created during the Db2 installation process are assigned the Dash shell by default. (Unfortunately, this can cause shell scripts that begin with the line **#!/bin/bash** to stop working). If you want Bash to be the default shell used when you log in as either of these two users, execute the command **chsh -s /bin/bash** *UserName* (where *UserName* is the name of the Db2 instance owner or fenced user) before doing anything else.

Building the SAMPLE Database

Db2 comes with a sample database (named SAMPLE) that, when created, can be used for a variety of purposes, including the development and testing of Db2 applications. Consequently, most of the sample programs that were created to supplement this book have been designed to work with the SAMPLE database. (You can learn more about the Db2 sample database here: *https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.apdv.samptop.doc/doc/r0001094.html*.)

To create the Db2 sample database and run a simple query against it, perform the following steps:

1. Open a terminal window (if you don't already have one open).
2. Log in as the Db2 instance user (which is *db2inst1*, by default) by executing the following command:

   su – db2inst1

   When the **Password:** prompt appears, provide the appropriate password and press **Enter**.

Important: If you specified a different Db2 instance username when installing Db2, replace `db2inst1` in this command with the appropriate instance owner name.

3. Start the Db2 database manager background processes by executing the following command:

   db2start

   If the appropriate background processes were started successfully, you should see a message that looks like this: **SQL1063N DB2START processing was successful.** (If the Db2 database manager background processes are already running, the message will look like this instead: **SQL1026N The database manager is already active**.)

4. Create the Db2 sample database by executing a command that looks like this:

   db2sampl -force -sql

   The **-force** option tells Db2 to re-create the database if it already exists; the **-sql** option tells Db2 not to include XML data in the database it creates.

   When this operation is completed (usually within 2 to 5 minutes), you should see a message that says **'db2sampl' processing complete.**

5. Connect to the database that was just created by executing the following command:

   db2 connect to sample

   When this command is executed, you should see a message that looks something like this:

   ```
   Database Connection Information
   Database server        = DB2/LINUXX866411.1.4.4
   SQL authorization ID   = DB2INST1
   Local database alias   = SAMPLE
   ```

6. Once a database connection is established, retrieve the contents of a table named DEPARTMENT by executing the following command:

   db2 "select * from department"

Important: Be sure to include quotes around the text that follows **db2**; if the quotes are omitted, the operating system will try to process the asterisk as a wildcard and an error will result.

You should be presented with fourteen rows of data along with the message **14 record(s) selected.**

7. Terminate the database connection by executing the following command:

   ```
   db2 connect reset
   ```

   You should see the message **DB20000I The SQL command completed successfully.**

Installing the Db2 Python Library (Driver)

Before you can build Python applications that work with a Db2 server or database, you must first install the **ibm_db** Python library (driver). This is done by performing the following steps:

1. Open a terminal window (if you don't already have one open).
2. Log in as the root user by executing the following command:

   ```
   su - root
   ```

   When the **Password:** prompt appears, provide the appropriate password for the root user. The command line prompt should change from **$** to **#**.

3. If you are using Ubuntu Linux, execute the following commands to install packages that will aid in Python application development and make the installation of the **ibm_db** library easier:

   ```
   apt-get -y install git
   apt-get -y install python3-setuptools
   apt-get -y install python-pip
   apt-get -y install python3-pip
   pip install --upgrade pip
   pip3 install -U testresources
   pip3 install -U pytz
   pip3 install -U ipython
   pip3 install -U ibm_db
   apt-get update
   ```

   If you are using Red Hat Enterprise Linux, execute the following commands instead:

   ```
   subscription-manager repos –enable rhel-server-rhscl-7-rpms
   yum -y install @development
   yum -y install rh-python36
   yum -y install rh-python36-python-tools
   yum -y install rh-python36-python-six
   chmod -R 777 /opt/rh/rh-python36
   ```


4. (**Red Hat Enterprise Linux only**) Use your editor of choice to add the following lines to the *.bashrc* file for the Db2 instance user (which is *db2inst1*, by default):

   ```
   # Add RHSCL Python 3 to my login environment
   source scl_source enable rh-python36
   ```

   If you want other users to be able to execute Python 3 applications, you need to add these lines to the *.bashrc* file for those users as well.

Note: The *.bashrc* file is a hidden shell script file that is located in the home directory assigned to a user; it's used to save and load environment variables and terminal preferences whenever a bash (Bourne-Again Shell) terminal window is started. Thus, to add lines to the *.bashrc* file for a user named **ibm_admin**, you would need to edit the file */home/ibm_admin/.bashrc*. The user **ibm_admin** would then have to open a new terminal window for the changes to take effect.

5. (**Red Hat Enterprise Linux only**) Open a new terminal window and use the **su** command to log in with a non-root administrator account that has had the *.bashrc* file modified. For example, to log in under a user account that was assigned the name **ibm_admin**, you would execute a command that looks like this:

   ```
   su – ibm_admin
   ```

   (If the **Password:** prompt appears, provide the appropriate password and press **Enter**.)

6. (**Red Hat Enterprise Linux only**) Install the **ibm_db** Python library software by executing the following commands:

   ```
   pip install --upgrade pip
   pip3 install -U ipython
   pip3 install -U ibm_db
   ```

Important: If you plan to use Anaconda to build and run Jupyter Notebooks (iPython programs) on Ubuntu or Red Hat Enterprise Linux, you should install the latest version of the Anaconda software *before* you install the **ibm_db** Python library (Steps 3 to 6). Refer to the *readme.md* file at *https://github.com/IBM/db2-python/tree/master/Jupyter_Notebooks* for information on how to configure Anaconda to use the **ibm_db** library.

Summary

After you have successfully completed the steps outlined in this chapter, you should have a Linux environment that can be used to develop Python 3.6 applications that interact with the **SAMPLE** database provided with Db2. You should also be able to create other databases and database objects to work with, as well as explore the different features that has Db2 to offer. (To learn more about these features, refer to the *IBM Db2 Version 11.1 Knowledge Center*.)

CHAPTER 6

Building Python Applications That Work with Db2

At the most basic level, an application is a computer program that has been designed to perform a group of coordinated operations to solve a particular problem. Consequently, all applications are constructed around five basic elements:

1. Input
2. Logic (decision control)
3. Memory (data storage and retrieval)
4. Arithmetic operations (calculations or processing)
5. Output

Input is defined as the way an application receives the information it needs to produce solutions for the problems it has been designed to solve. Once the appropriate input is received, *logic* takes over and controls what information (data) should be placed in or taken out of *memory* and what *arithmetic operations* should be performed on that information. (Because data placed into or taken out of memory is not persistent and can be lost if not physically stored somewhere else, applications often interact with the operating system to move data to and from simple character or byte-oriented files.) Finally, when the application has generated a solution to

the problem it was designed to solve, it provides *output* in the form of an answer or specific action.

Applications that work with Db2 still contain these basic elements; the only real difference is the way in which persistent data is stored and retrieved, and in some cases, the way logic is exercised. File input/output (I/O) operations are replaced with SQL operations, and in some cases, decision control can be built directly into a Db2 database in the form of a *trigger*, *stored procedure*, or *constraint*. But, because of this difference, Python applications that use a Db2 database for persistent storage must perform three distinct tasks that are not required by more traditional applications:

- Establish a connection to a Db2 server or database
- Perform any transaction processing required (using SQL)
- Terminate the connection when it is no longer needed

Additionally, Python applications that work with Db2 often perform other tasks like obtaining information about a specific database or retrieving Db2-specific error messages when a desired Db2 server or database operation fails.

Functionality Provided by the ibm_db and ibm_db_dbi Python Libraries

In Chapter 4, "Python and Db2," we saw that there are two Python libraries that are used primarily to build Python applications that interact with Db2 servers and databases: the **ibm_db** library and the **ibm_db_dbi** library. And, in Chapter 3, "The Db2 Call Level Interface," we learned that many of the APIs found in the **ibm_db** library are similar to those found in the Db2 CLI driver. Table 6.1 shows the APIs available with this library, along with their purpose.

Table 6.1: APIs available with the ibm_db Python library

API	Purpose
Data Source Connection Management	
ibm_db.connect()	Establish a new connection to an IBM Db2 server or database.
ibm_db.pconnect()	Establish a persistent connection to an IBM Db2 server or database.

Table 6.1: APIs available with the ibm_db Python library

API	Purpose
ibm_db.close()	Close an open IBM Db2 server or database connection and if appropriate, return it to a connection pool.
ibm_db.active()	Determine whether the Db2 server or database connection specified is active.
Database Creation and Deletion	
ibm_db.createdb()	Create a new database using the name and code set specified.
ibm_db.createdbNX()	Create a new database using the name and code set specified; if the database already exists, do not return an error.
ibm_db.recreatedb()	Drop and re-create a database, using the name and code set specified.
ibm_db.dropdb()	Delete (drop) the database specified.
SQL Statement Processing	
ibm_db.bind_param()	Associate (bind) parameter markers coded in a prepared SQL statement with application variables.
ibm_db.prepare()	Send an SQL statement to a Db2 server or database to have it prepared for execution.
ibm_db.execute()	Execute an SQL statement that has been prepared by the **ibm_db.prepare()** API.
ibm_db.execute_many()	Execute an SQL statement that has been prepared by the **ibm_db.prepare()** API, using the parameter sequences or mappings specified.
ibm_db.exec_immediate()	Prepare and execute an SQL statement, using values supplied for parameter markers that were coded in the statement (if any).
ibm_db.num_rows()	Determine how many rows were inserted, updated, or deleted by an SQL statement.
ibm_db.callproc()	Invoke (execute) a stored procedure.
Query Results Retrieval	
ibm_db.fetch_assoc()	Retrieve a row from a result set and copy its data to a dictionary.
ibm_db.fetch_tuple()	Retrieve a row from a result set and copy its data to a tuple.
ibm_db.fetch_both()	Retrieve a row from a result set and copy its data to both a dictionary *and* a tuple.
ibm_db.fetch_row()	Move a cursor to a specific row in a result set *or* advance a cursor to the next row in a result set.
ibm_db.result()	Retrieve a value from a column in the current row of a result set.

Table 6.1: APIs available with the **ibm_db** Python library	
API	**Purpose**
ibm_db.next_result()	Retrieve the next result set returned by a stored procedure.
ibm_db.free_result()	Free all system resources associated with an **IBM_DBStatement** object.
Query Result Set Information	
ibm_db.num_fields()	Determine the number of columns that exist in a result set.
ibm_db.field_name()	Determine the name of a column in a result set.
ibm_db.field_num()	Determine the position of a column in a result set.
ibm_db.field_type()	Determine the data type of a column in a result set.
ibm_db.field_width()	Determine the width (size) of a column in a result set.
ibm_db.field_display_size()	Determine the maximum number of bytes needed to display a column in a result set.
ibm_db.field_precision()	Determine the precision (i.e., total number of digits) of a column in a result set.
ibm_db.field_scale()	Determine the scale (i.e., number of decimal digits) of a column in a result set.
Transaction Management	
ibm_db.autocommit()	Determine or set the AUTOCOMMIT behavior of a Db2 database connection.
ibm_db.commit()	Terminate an in-progress transaction and make the effects of all operations performed by that transaction permanent.
ibm_db.rollback()	Terminate an in-progress transaction and back out (roll back) the effects of all operations performed by that transaction.
Error Handling	
ibm_db.conn_error()	Return an SQLSTATE value associated with an **IBM_DBConnection** object.
ibm_db.conn_errormsg()	Return an SQLCODE and corresponding error message associated with an **IBM_DBConnection** object.
ibm_db.stmt_error()	Return an SQLSTATE value associated with an **IBM_DBStatement** object.
ibm_db.stmt_errormsg()	Return an SQLCODE and corresponding error message associated with an **IBM_DBStatement** object.
Data Source Information	
ibm_db.server_info()	Obtain information about an IBM Db2 server.
ibm_db.client_info()	Obtain information about an IBM Db2 client.

Table 6.1: APIs available with the **ibm_db** Python library	
API	**Purpose**
ibm_db.tables()	Retrieve a list of tables and their associated metadata.
ibm_db.table_privileges()	Retrieve a list of tables and their associated privileges.
ibm_db.columns()	Retrieve a list of columns and their associated metadata, for a given table.
ibm_db.column_privileges()	Retrieve a list of columns and their associated privileges, for a given table.
ibm_db.special_columns()	Retrieve a list of unique row identifier columns and their associated metadata, for a table.
ibm_db.statistics()	Retrieve statistical information for a table and its associated indexes.
ibm_db.primary_keys()	Retrieve information about the columns that make up the primary key for a table.
ibm_db.foreign_keys()	Retrieve information about the columns that participate in foreign keys that reference another table.
ibm_db.procedures()	Retrieve a list of stored procedures that have been registered in a database.
Ibm_db.procedure_columns()	Retrieve information about the parameters that have been defined for one or more stored procedures.
IBM_DBConnection *and* IBM_DBStatement *Object Attribute Management*	
ibm_db.cursor_type()	Retrieve information about the type of cursor currently being used.
ibm_db.get_option()	Retrieve the current value of a connection or statement option (attribute).
ibm_db.set_option()	Assign a value to a connection or statement option (attribute).

Because the **ibm_db_dbi** library adheres to the *PEP 249 — Python Database API Specification v2.0*, it contains only one API: the **ibm_db_dbi.connect()** API. Once executed, attributes and methods associated with the Connection object this API returns can be used to perform basic operations against the connected Db2 server or database.

Table 6.2 provides Information about the **ibm_db_dbi.connect()** API, along with a list of the object attributes and methods that are available with the **ibm_db_dbi** library.

Table 6.2: The API, object attributes, and object methods available with the ibm_db_dbi Python library	
API, Object Attribute, or Object Method	**Purpose**
Data Source Connection Management	
ibm_db_dbi.connect()	Establish a new connection to an IBM Db2 server or database and return a *Connection* object.
.close() Connection object method	Close an open IBM Db2 server or database connection.
SQL Statement Processing	
.cursor() Connection object method	Return a new ***Cursor*** object for the current Connection object.
.execute() Cursor object method	Prepare and execute an SQL statement, using values supplied for parameter markers that were coded in the statement (if any).
.executemany() Cursor object method	Execute an SQL statement, using all of the parameter sequences or mappings specified for parameter markers that were coded in the statement (if any).
.rowcount Cursor object attribute	Contains information about the number of rows that were inserted, updated, deleted, or returned by an SQL operation.
.callproc() Cursor object method	Invoke (execute) a stored procedure.
Query Results Retrieval	
.fetchone() Cursor object method	Retrieve a row from a result set and copy its data to a tuple.
.fetchmany() Cursor object method	Retrieve a specific number of rows from a result set and copy the data to a tuple.
.fetchall() Cursor object method	Retrieve every row from a result set and copy the data to a tuple.
.arraysize Cursor object attribute	Specify the number of rows to retrieve data from a result set for, at one time (when **.fetchmany()** is used).
.nextset() Cursor object method	Retrieve the next result set returned by a stored procedure.
.close() Cursor object method	Close the cursor object specified.
Query Result Set Information	
.description Cursor object attribute	Contains the following information for every column found in a result set, stored in a tuple: column name, data type(s), display size, internal size, precision, scale, and nullability.
Transaction Management	
.commit() Connection object method	Terminate an in-progress transaction and make the effects of all operations performed by that transaction permanent.

Table 6.2: The API, object attributes, and object methods available with the ibm_db_dbi Python library

API, Object Attribute, or Object Method	Purpose
.rollback() Connection object method	Terminate an in-progress transaction and back out (roll back) the effects of all operations performed by that transaction.

Important: You can find sample programs (and in the case of the **ibm_db** library, Jupyter Notebooks) that illustrate how to use every API, object attribute, and object method shown in Tables 6.1 and 6.2 in the IBM Db2-Python GitHub repository (*https://github.com/IBM/db2-python*).

Special Objects Used by Db2-Python Applications

Db2 CLI/ODBC applications rely on special data storage areas to interact with Db2 servers and databases—these storage areas are identified by unique *handles*, which are simply pointer variables that refer to data objects controlled by Db2 CLI or the ODBC Driver Manager. Use of these storage areas frees Db2 CLI/ODBC applications from having to allocate and manage global variables and Db2-specific data structures.

Because the Db2 CLI driver serves as the foundation for the **ibm_db** and **ibm_db_dbi** Python libraries, both of these libraries rely on similar special data objects to interact with Db2 servers and databases. With the **ibm_db** library, IBM_DBConnection and IBM_DBStatement objects are used; with the **ibm_db_dbi** library, Connection and Cursor objects are used instead. As the name implies, IBM_DBConnection and Connection objects are used to store information about a Db2 server or database connection, such as:

- The current state of the connection being managed
- The current value of each connection attribute available
- Diagnostic information about the connection the object refers to

On the other hand, IBM_DBStatement and Cursor objects are used to store specific information about a single SQL statement (and its associated cursor, if any), such as:

- The current value of each SQL statement attribute available
- The addresses of application variables that have been bound to (associated with) parameter markers coded in the SQL statement the object refers to
- Diagnostic information about the SQL statement the object refers to

Establishing a Db2 Server or Database Connection

Before any type of operation can be performed against a Db2 server or database, a connection to that server or database must first be established. Python applications that use the **ibm_db** library can establish Db2 server and database connections by executing the **ibm_db.connect()** or the **ibm_db.pconnect()** API. Python applications using the **ibm_db_dbi** library must execute the **ibm_db_dbi.connect()** API instead.

Note: The **ibm_db.connect()** API is used to establish a single connection, whereas the **ibm_db.pconnect()** API is used to establish a pool of persistent connections. Persistent connections are not closed by executing the **ibm_db.close()** API; instead, they are returned to a process-wide connection pool. The next time the **ibm_db.pconnect()** API is called, the connection pool is searched for a matching connection; if one is found, it is returned to the application. New connections are only created when there are no available connections in the pool.

You can find the syntax for these APIs, (as well as the other APIs in the **ibm_db** library) in the IBM Db2-Python GitHub repository. That said, the most common way of executing these APIs is by providing a connection string for the first input parameter. The connection string used must adhere to the following format:


```
DRIVER={IBM DB2 ODBC DRIVER};
ATTACH=connType;
DATABASE=dbName;
HOSTNAME=hostName;
PORT=port;
PROTOCOL=TCPIP;
UID=userName;
PWD=password
```

where:

connType	Specifies whether the connection is to be made to a Db2 server or a database; **TRUE** indicates the connection is to be made to a Db2 server, and **FALSE** indicates the connection is to be made to a database.
dbName	The name of the Db2 server or database the connection is to be made to. *This option is only required when connecting to a Db2 database.*
hostName	The host name or IP address of the Db2 server the connection is to be made to. (The host name is the name of the Db2 server, as it is known to the TCP/IP network.) *This option is only required when connecting to a Db2 server.*
port	The port number that receives Db2 connections on the server the connection is to be made to. (Port number **50000** is used by default.) *This option is only required when connecting to a Db2 server.*
userName	The username/ID that is to be used for authentication purposes when the connection is first established.
password	The password that corresponds to the username/ID specified in the *userName* parameter.

Thus, if the **ibm_db** library is used, the Python code needed to establish a connection to a database named SAMPLE with the user ID "*db2inst1*" and the password "*Passw0rd*" would look something like this:

```
#! /usr/bin/python3
# Load The Appropriate Python Modules
```

```
import ibm_db

# Define And Initialize The Appropriate Variables
dbName = "SAMPLE"
userID = "db2inst1"
passWord = "PasswOrd"
connID = None

# Construct The String That Will Be Used To Establish
# A Database Connection
connString = "DRIVER={IBM DB2 ODBC DRIVER}"
connString += ";ATTACH=FALSE"
connString += ";DATABASE=" + dbName
connString += ";PROTOCOL=TCPIP"
connString += ";UID=" + userID
connString += ";PWD=" + passWord

# Attempt To Establish A Connection
connID = ibm_db.connect(connString, '', '')
if connID is None:
    print("\nERROR: Unable to connect.")
exit(-1)

# Display A Status Message
print("Connected!")
...
```

If the **ibm_db_dbi** library is used, the Python code needed to establish the exact same connection would be very similar; the following code snippet highlights the changes that would be required:

```
#! /usr/bin/python3
# Load The Appropriate Python Modules
import ibm_db_dbi
...
# Attempt To Establish A Connection
connID = ibm_db_dbi.connect(connString, '', '')
```



```
if connID is None:
    print("\nERROR: Unable to connect.")
exit(-1)

# Display A Status Message
print("Connected!")
...
```

Once a Db2 server or database connection is established, it remains in effect until it is explicitly terminated or until the application that established the connection ends.

Transaction Processing: Executing SQL Statements

The bulk of the work that is performed by most Python applications that interact with Db2 servers and databases revolves around transaction processing, which is where SQL statements are passed to the appropriate server or database, via API calls (**ibm_db** library) or object methods (**ibm_db_dbi** library), for execution. This is also where operations performed against a Db2 database are made permeant or backed out, and where the results of SQL queries are retrieved. Depending upon which Db2 Python library is used, there can be two ways in which SQL statements are executed:

- **Prepare and then Execute.** This approach separates the preparation of an SQL statement from its actual execution and is typically used when a statement is to be executed repeatedly (often with different values being supplied for parameter markers with each execution). This method is also used when an application needs information about the columns that will exist in the result set that will be produced when the statement is executed, in advance. The **ibm_db.prepare()** and **ibm_db.execute()** APIs found in the **ibm_db** library are used to process SQL statements in this manner.
- **Execute immediately.** This approach combines the preparation and execution of an SQL statement into a single step and is typically used when a statement only needs to be executed once. The **ibm_db.exec_immediate()** API (**ibm_db** library) and the **.execute()** Cursor object method (**ibm_db_dbi** library) are used to process SQL statements in this manner.

Both methodologies allow the use of *parameter markers* in place of constants or expressions in the SQL statements used. Parameter markers are represented by question mark (?) characters and indicate positions in an SQL statement where the value of one or more application variables is to be substituted when the statement is executed. If an application variable has been associated with a specific parameter marker in an SQL statement, that variable is said to be "bound" to the parameter marker; such bindings can be carried out by executing the **ibm_db.bind_param()** API—provided the **ibm_db** library is being used.

Note: With Python, application variables do not have to be "bound" in order to be used to provide values to parameter markers during SQL statement execution. Values can also be passed, via a tuple, when the **ibm_db.execute()** or **ibm_db.exec_immediate()** API is called (**ibm_db** library) or when the **.execute()** Cursor object method is invoked (**ibm_db_dbi** library). However, binding gives you more control over the parameter type (input, output, or input/output); SQL data type; precision; and scale that are used for the parameter marker values provided.

The following pseudo-code example illustrates how an application variable would be bound to a parameter marker that has been coded in a simple **SELECT** statement. It also illustrates the way in which a value would be provided for the bound parameter before the statement is executed:

```
#! /usr/bin/python3
# Load The Appropriate Python Modules
import ibm_db

# Define And Initialize The Appropriate Variables
...
deptID = ['B01', 'C01', 'D01', 'E01']

# Construct The String That Will Be Used To Connect
# To A Database, Then Establish A Connection
...

# Define The SQL Statement To Be Executed
sqlStmt = "SELECT projname FROM project "
sqlStmt += "WHERE deptno= ?"

# Prepare The SQL Statement Just Defined
prepStmt = ibm_db.prepare(connID, sqlStmt)

# If The SQL Statement Could Not Be Prepared
# Display An Error Message And Exit
if prepStmt is False:
    print("Unable to prepare the statement.")

# For Every Value Specified In The deptID List, ...
for x in range(0, 4):

    # Assign A Value To The Appropriate Variable
    pValue = deptID[x]

    # Bind The Variable To The Parameter Marker
    retCode = ibm_db.bind_param(prepStmt, 1, pValue,
        ibm_db.SQL_PARAM_INPUT, ibm_db.SQL_CHAR)

    # Execute The Prepared Statement
    rc = ibm_db.execute(prepStmt)

    # Retrieve And Print The Records Produced
...
```


Transaction Processing: Retrieving Query Results

When an SQL statement other than the **SELECT** statement is executed, the only additional processing that might be performed is a check of the return code produced to verify the statement executed as expected. Or, a check to determine how many rows were affected by the operation. However, when

a **SELECT** statement (i.e., a query) is executed, any results that might have been produced will typically need to be retrieved and processed. With the **ibm_db** library, all the query results produced can be copied to a dictionary, tuple, or both (by calling the **ibm_db.fetch_assoc()**, **ibm_db.fetch_tuple()**, or **ibm_db.fetch_both()** API). Alternately, the **ibm_db.fetch_row()** API can be used to advance the cursor to the next row (or move the cursor to a specific row) in a result set, and the **ibm_db.result()** API can be used to copy data from individual columns in that row to application variables. When the **ibm_db_dbi** library is used, one or more records can be retrieved and copied to a tuple by executing the **.fetchone()**, **.fetchmany()**, or **.fetchall()** Cursor object method.

The following pseudo-source code example illustrates how **ibm_db.fetch_tuple()** API in the **ibm_db** Python library might be used to retrieve data from a result set:

```
#! /usr/bin/python3
# Load The Appropriate Python Modules
import ibm_db

# Define And Initialize The Appropriate Variables
...
record = False

# Construct The String That Will Be Used To Connect
# To A Database, Then Establish A Connection
...

# Define The SQL Statement To Be Executed
sqlStmt = "SELECT deptname FROM department "
sqlStmt += "WHERE admrdept = 'A00'"

# Execute The SQL Statement Just Defined
results = ibm_db.exec_immediate(connID, sqlStmt)

# As Long As There Are Records In The Result Set, ...
noData = False
```



```
while noData is False:

    # Retrieve A Record And Store It In A Tuple
    record = ibm_db.fetch_tuple(results)

    # If There Are No More Records, Exit The Loop
    if record is False:
        noData = True

    # Otherwise, Display The Data Retrieved
    else:
        print(record[0])
...
```

The next pseudo-source code example illustrates how the **.fetchall()** Cursor object method could be used to retrieve data from a result set if the **ibm_db_dbi** Python library were used instead:

```
#! /usr/bin/python3
# Load The Appropriate Python Modules
import ibm_db_dbi

# Define And Initialize The Appropriate Variables
...
record = False

# Construct The String That Will Be Used To Connect
# To A Database, Then Establish A Connection
...

# Retrieve The Cursor Object That Was Created For
# The Connection Object
if not connID is None:
    cursorID = connID.cursor()

# Define The SQL Statement To Be Executed
sqlStmt = "SELECT deptname FROM department "
```

```
sqlStmt += "WHERE admrdept = 'A00'"

# Execute The SQL Statement Just Defined
if not cursorID is None:
    cursorID.execute(sqlStmt)

    # Retrieve All Of The Records Returned
    # And Store Them In A Python Tuple
    results = cursorID.fetchall()

    # Print Every Record Returned
    for value in results:
        print("{:24}" .format(value[0]))
...
```

Transaction Processing: Obtaining Result Set Information

If a **SELECT** statement is hard-coded into an application, the structure of the result set that will be produced is typically known in advance. However, if the **SELECT** statement used is created at application run time, you may need to acquire this information before you can process the results. With the **ibm_db** library, information about a result set can be obtained by executing any of the following APIs:

- **ibm_db.num_fields()**
- **ibm_db.field_name()**
- **ibm_db.field_num()**
- **ibm_db.field_type()**
- **ibm_db.field_width()**
- **ibm_db.field_display_size()**
- **ibm_db.field_precision()**
- **ibm_db.field_precision()**

When the **ibm_db_dbi** library is used, similar information can be obtained by examining the contents of the **.description** Cursor object attribute. The following pseudo-source code example illustrates how this attribute can be examined to obtain result set column information:


```
#! /usr/bin/python3
# Load The Appropriate Python Modules
import ibm_db_dbi

# Define And Initialize The Appropriate Variables
...

# Construct The String That Will Be Used To Connect
# To A Database, Then Establish A Connection
...

# Retrieve The Cursor Object That Was Created For
# The Current Connection Object
if not connID is None:
    cursorID = connID.cursor()

# Define The SQL Statement To Be Executed
sqlStmt = "SELECT * FROM department "
sqlStmt += "WHERE admrdept = 'A00'"

# Execute The SQL Statement Just Defined
if not cursorID is None:
    cursorID.execute(sqlStmt)

# Retrieve Attribute Information For Every Column
# In The Result Set Produced By The Statement
colInfo = cursorID.description

# Display Column Name And Data Type Information
x = 0
for record in colInfo:
    print("Column name          : ", end="")
    print(colInfo[x][0])
    dataTypes = colInfo[x][1]
    typeNames = []
```

```
    for type in dataTypes:
        typeNames.append(type)
    print("Data type name(s)          : ", end="")
    print(typeNames[0])
    del typeNames[0]
    for type in typeNames:
        print(' ' * 27, end="")
        print("{}" .format(type))

    # Increment The x Variable And Print A Blank Line
    x += 1
    print()

...
```

Transaction Processing: Terminating the Current Transaction

You may recall that in Chapter 2, "Structured Query Language," we saw that a *transaction* (also known as a *unit of work*) is a sequence of one or more SQL operations that are grouped together as a single unit, usually within an application process. Transactions are important because the initiation and termination of a single transaction defines points of data consistency within a database. Either the effects of all operations performed within a transaction are applied to the database and made permanent (committed), or, they are backed out (rolled back) and the database is returned to the state it was in before the transaction began.

When it comes to managing transactions, Python applications can be configured such that they run in one of two modes: *auto-commit* mode or *explicit-commit* mode. When auto-commit mode is used, every SQL operation performed is treated as a complete, individual transaction that is automatically committed as soon as it is successfully executed. When explicit-commit mode is used, transactions are started implicitly the first time an application connects to a data source or the first time an SQL operation is performed *after* a previously running transaction has been terminated. With Python applications, such transactions are explicitly ended when the **ibm_db.commit()** API or **ibm_db.rollback()** API is

executed (**ibm_db** library) or the **.commit()** or **.rollback()** Connection object method is invoked (**ibm_db_dbi** library).

When the **ibm_db** library is used, auto-commit mode is enabled by default. However, the actual commit mode used can be controlled by assigning the value **ibm_db.SQL_AUTOCOMMIT_ON** or **ibm_db.SQL_AUTOCOMMIT_OFF** to the **ibm_db.SQL_ATTR_AUTOCOMMIT** connection option at the time a connection to a Db2 database is made. Alternatively, the **ibm_db.autocommit()** API can be used to change the commit mode used after a database connection has been established. When the **ibm_db_dbi** library is used, explicit-commit is the only commit mode available. Consequently, every transaction must be terminated by executing the **.commit()** or **.rollback()** method of the Connection object that was returned when the **ibm_db_dbi.connect()** API was executed. Regardless of which type of commit mode is used, all transactions associated with a specific database connection should be completed before that connection is terminated.

When running in explicit-commit mode, the following should be taken into consideration:

- Only the current, active transaction can be committed or rolled back; therefore, all dependent SQL operations should be performed within the same transaction.
- Various table-level and row-level locks can be held by a single transaction and these locks are not released until the transaction is terminated. Consequently, other concurrently running transactions may be prevented from getting access to the locked data until the transaction holding the locks ends.
- Any transaction that has not been committed or rolled back before an application ends will be "lost" and its effects will be discarded. The same is true if a system or application failure occurs. Therefore, transactions should be ended as soon as reasonably possible.

When defining transaction boundaries, keep in mind that all resources associated with a transaction—with the exception of those coupled with a held cursor—are released. However, prepared SQL statements and parameter marker bindings are maintained from one transaction to the next. Therefore, once an SQL statement has been prepared, it does not need to be prepared

again—even after a commit or rollback operation is performed—provided it remains associated with the same IBM_DBStatement or Cursor object.

Calling Stored Procedures

If an application has one or more transactions that perform a relatively large amount of work with little or no user interaction, those transactions can be encapsulated and stored on the database server as a *stored procedure*. Stored procedures make it possible to perform data processing operations directly at the server, which typically is a high-performant computer that can provide quick, coordinated data access. More importantly, because a stored procedure is invoked by a single SQL statement (or API/Cursor object method), fewer messages have to be transmitted across the network—only the data that is actually needed at the client has to be sent across the wire.

Once a stored procedure has been created and registered with a Db2 database (using the **CREATE PROCEDURE** SQL statement), that procedure can be invoked, either interactively using a utility like the Db2 CLP, or from an application. Typically, stored procedures are invoked by executing the **CALL** SQL statement. However, with Python applications, the preferred way to invoke a stored procedure is by executing the **ibm_db.callproc()** API (if the **ibm_db** library is used), or the **.callproc()** Cursor object attribute (if the **ibm_db_dbi** library is used).

The following pseudo-source code example illustrates how **ibm_db.callproc()** API in the **ibm_db** Python library might be used to execute a stored procedure:

```
#! /usr/bin/python3
# Load The Appropriate Python Modules
import ibm_db

# Define And Initialize The Appropriate Variables
...
pVals = (0.0, 0.0)

# Construct The String That Will Be Used To Connect
# To A Database, Then Establish A Connection
...
```



```
# Define An SQL Statement To Create A
# Stored Procedure
sqlStmt = "CREATE OR REPLACE PROCEDURE salary_stats "
sqlStmt += "(OUT maxSal DOUBLE, OUT minSal DOUBLE) "
sqlStmt += "DYNAMIC RESULT SETS 0 "
sqlStmt += "BEGIN"
sqlStmt += "SELECT MAX(salary) INTO maxSal "
sqlStmt += "FROM employee; "
sqlStmt += "SELECT MIN(salary) INTO minSal "
sqlStmt += "FROM employee; "
sqlStmt += "END"

# Execute The SQL Statement Just Defined
retCode = ibm_db.exec_immediate(connID, sqlStmt)

# Execute The Stored Procedure Just Created
results = ibm_db.callproc(connID, "salary_stats", pVals)

# Display The Values Returned By The Stored Procedure
print("Highest salary : $ ", end="")
print(results[1])
print("Lowest salary: $ ", end="")
print(results[2])
...
```

It's important to note that a single stored procedure is capable of returning zero, one, or more result sets, depending on how it was designed. (If you look closely at the pseudo-source code example just presented, you will see a line that reads "**DYNAMIC RESULT SETS 0**"; this line tells Db2 the procedure will not return any result sets.) The **ibm_db.next_result()** API (if the **ibm_db** library is used), or the **.nextset()** Cursor object method (if the **ibm_db_dbi** library is used), can be used to initialize processing of the next result set available if a stored procedure returns more than one result set.

Terminating a Db2 Server or Database Connection

The last thing every Db2-based application must do before returning control to the operating system is terminate any Db2 server or database

connections that have been established. When the **ibm_db** Python library is used, connections can be terminated by calling the **ibm_db_dbi.close()** API; when the **ibm_db_dbi** Python library is used, connections can be terminated by executing the **.close()** method of the Connection object that was returned when the **ibm_db_dbi.connect()** API was invoked.

Obtaining Information About a Data Source and Setting Driver Options

There may be times when it is necessary to obtain information about a Db2 client, server, or database an application is connected to. For this reason, all CLI/ODBC drivers must support a set of functions that can provide information about the capabilities of a driver and the driver's underlying data source. And, because the Db2 CLI driver serves as the foundation for the **ibm_db** and **ibm_db_dbi** libraries, similar capability is available to Python applications that use the **ibm_db** library. (*This functionality is NOT available with the* **ibm_db_dbi** *library*.)

The **ibm_db.client_info()** and **ibm_db.server_info()** APIs can be used to obtain information about a Db2 client or server being used. And, metadata from a Db2 database's system catalog can be obtained by executing any of the following APIs:

- **ibm_db.tables()**
- **ibm_db.table_privileges()**
- **ibm_db.columns()**
- **ibm_db.column_privileges()**
- **ibm_db.special_columns()**
- **ibm_db.statistics()**
- **ibm_db.primary_keys()**
- **ibm_db.foreign_keys()**
- **ibm_db.procedures()**
- **ibm_db.procedure_columns()**

Most data source drivers contain additional information that can be changed to alter the way in which a driver behaves for a particular application. This updatable information is referred to as *driver attributes* or *options*. And with the **ibm_db** library, two types of driver options are available: connection options and SQL statement options. Python applications can retrieve the value of the connection and statement options available by executing the **ibm_db.get_option()** API. And, they can change the value of select connection and statement options by calling the **ibm_db.set_option()** API.

Diagnostics and Error Handling

Error handling is an important part of any application, and Python applications that interact with Db2 are no exception. At a minimum, a Db2-Python application should always check to see if an API or object method that was invoked was able to execute successfully. Often, this can be done by examining the API or object method's return code. (For the most part, APIs in the **ibm_db** library return **False** or **None** if they fail to execute. If an object method in the **ibm_db_dbi** library fails to execute, an Error or subclass exception will typically be raised.) In either case, users should be notified that an error or warning condition has occurred and, whenever possible, they should be provided with sufficient diagnostic information to help them identify and correct the problem.

Although return codes can indicate that an error or warning condition occurred, they do not always provide an application (or developer or user) with specific information about what caused the error or warning to be generated. Because additional information about an error or warning condition is usually needed to resolve a problem, Db2 (as well as other relational database products) use a set of error message codes known as SQLSTATEs to provide supplementary diagnostic information for warnings and errors. SQLSTATEs are alphanumeric strings that are five characters (bytes) in length and have the format *ccsss*, where *cc* indicates the error message class and *sss* indicates the error message subclass. Any SQLSTATE that has a class of **01** corresponds to a warning; any SQLSTATE that has a class of **HY** corresponds to an error that was generated by the Db2 CLI; and any SQLSTATE that has a class of **IM** corresponds to an error that was generated by the ODBC Driver Manager. (Because different database servers often have different diagnostic message codes, SQLSTATEs follow standards that are outlined in the X/Open CLI standard specification. This standardization of SQLSTATE values enables application developers to process errors and warnings consistently across different relational database products.)

Unlike return codes, SQLSTATEs are often treated as guidelines, and drivers are not required to return them. Consequently, most applications simply display them, along with any corresponding diagnostic message and native error code available. Loss of functionality rarely occurs with this approach because applications normally do not base programming logic on SQLSTATE values.

When the **ibm_db** library is used, SQLSTATE values associated with Db2 server or database connections can be obtained by executing the **ibm_db.conn_error()** API. On the other hand, SQLSTATE values associated with SQL statements can be retrieved by executing the **ibm_db.stmt_error()** API. Developers using the **ibm_db_dbi** library, however, will find that a different set of error codes are used to conform to the PEP 249 — Python Database API Specification. Consequently, SQLSTATE values are not available when this library is used.

SQLSTATE values alone may not be enough to help resolve a problem if an application user is unfamiliar with them. Therefore, two additional APIs that can be used to obtain Db2-specific error messages associated with connections and SQL statements are provided with the **ibm_db** library. These APIs are **ibm_db.conn_errormsg()** and **ibm_db.conn_errormsg()**.

For examples of how to perform error handling using API return codes, as well as the error handling APIs just discussed, refer to the sample programs found in the IBM Db2-Python GitHub repository.

QuickStart Guide to Db2 Development with Python

Python has grabbed the attention of application developers and is fast becoming one of the world's most popular programming languages, according to a 2019 Stack Overflow poll of nearly 90,000 developers. Python joins the ranks of the top five programming languages, holding a respectable fourth place behind JavaScript, HTML/CSS, and SQL. It's an open source, easy-to-read, flexible language that's a great fit for many types of applications.

IBM Db2 is an enterprise-level family of hybrid data management products used in Fortune 500 companies worldwide. It's designed to make storing and managing structured and unstructured data easy, no matter where the data resides—on-premises or in a private or public cloud. Released in 1983, Db2 continues to deliver innovation as an industry-leading platform for efficient data management. It delivers high-impact data insights, seamless business continuity, and real business transformation for many organizations.

This book brings Python, SQL, and Db2 application development together as never before, to show how these three technologies can successfully be used with each other. By reading this book, you will receive:

- An introduction to Db2
- An overview of SQL and how it is used
- An introduction to Python and the Python libraries/drivers available for Db2 application development
- A step-by-step guide for setting up a Python-Db2 development environment (on RedHat or Ubuntu Linux)
- In-depth information on how to structure and build Python applications that interact with Db2 (along with the link to a GitHub site that contains over 70 sample programs and 60 Jupyter Notebooks)

Whether you're a Python developer who wants to build applications that work with Db2, or you're a Db2 user who wants to know how to build Python applications that interact with Db2 servers and databases, you'll find this book a must-read.

About the Author

ROGER E. SANDERS is a Field Enablement Professional focused on Machine Learning on IBM Z Systems at IBM. He has worked with Db2 since it was first introduced and is the author of 25 books on relational database technology (six focused on application development), and many articles and tutorials on database topics. He lives in Fuquay Varina, North Carolina.

MC PRESS

MC Press Online, LLC
3695 W. Quail Heights Court
Boise, ID 83703-3861

ISBN 978-1-58347-888-2
50995 >
EAN
9 781583 478882