Table of Contents

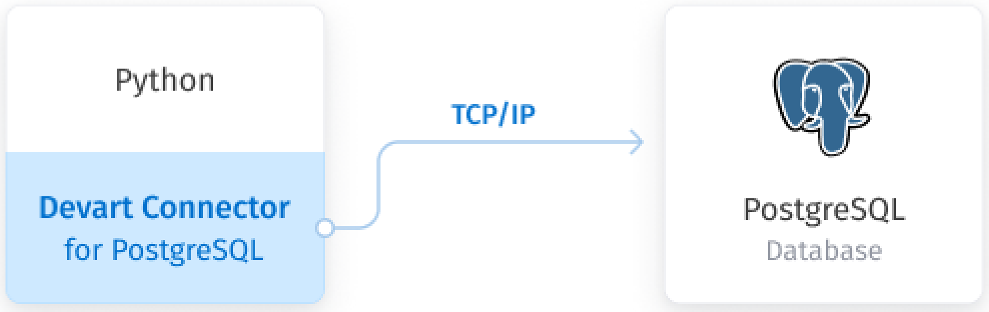
1 Overview

Overview

Python Connector for PostgreSQL is a connectivity solution for accessing PostgreSQL databases from Python applications. It fully implements the Python DB API 2.0 specification. The connector is distributed as a wheel package for Windows, macOS, and Linux.

Direct connection

The connector enables you to establish a direct connection to PostgreSQL from a Python application via TCP/IP, eliminating the need for the database client library. A direct connection increases the speed of data transmission between the application and PostgreSQL database server. It also streamlines the deployment process since you don't have to distribute any client libraries with the application.



Secure communication

The connector supports encrypted communications using SSL/TLS, SSH tunneling, and HTTP/HTTPS tunneling.

2 What's new

Python Connector for PostgreSQL 1.3

- Added support for Python 3.14

- Added support for PostgreSQL 18
- Added support for the Bearer Token authentication when using an HTTP tunnel
- Added support for password-protected private keys for the SSL protocol
- Added support for the COPY ... TO STDOUT statement
- Improved support for array data types
- Improved performance in opening a connection

Python Connector for PostgreSQL 1.2

- Added support for Python 3.13
- Added support for PostgreSQL 17
- Added support for the pgvector extension
- Added Read Only mode for connection

Python Connector for PostgreSQL 1.1

- Added support for PostgreSQL 16
- Added connection pooling
- Added activation with a license key
- Added the subscription license type

Python Connector for PostgreSQL 1.0

- Initial release of Python Connector for PostgreSQL
- Added support for Windows 32-bit and 64-bit
- Added support for Windows Server 32-bit and 64-bit
- Added support for macOS 64-bit and ARM (Apple M1 and M2)
- Added support for Linux 64-bit

3 Compatibility

Compatibility

Python Connector for PostgreSQL supports standard Python tooling and is compatible with major operating systems.

Python versions

The connector supports Python versions 3.7 through 3.14.

PostgreSQL versions

The connector supports the following PostgreSQL versions:

- PostgreSQL 8.0–18

PostgreSQL services

You can also use the connector with the following PostgreSQL-compatible services:

- Microsoft Azure Database for PostgreSQL
- Amazon RDS for PostgreSQL
- Amazon Aurora
- Google Cloud for PostgreSQL
- Heroku Postgres
- AlloyDB

Libraries

You can use the connector with the following Python libraries:

- SQLAlchemy
- pandas
- petl

Platforms

The connector runs on the following platforms:

- Windows (32-bit and 64-bit)
- Windows Server (32-bit and 64-bit)
- macOS (64-bit and ARM—Apple M1 and M2)

- Linux (64-bit)

Note: For more information about supported OS versions, see the compatibility page of your Python version.

4 Installation

4.1 Windows

Install the connector on Windows

You can install the connector from the Python Package Index (PyPI) or a wheel (.whl) file.

Install the connector from PyPI

1. Open Command Prompt.
2. Verify that you have the pip package installer on your system using the `py -m pip --version` command. If you don't have it, run the following command to install pip.

```
python -m ensurepip --upgrade
```

3. Install the package.

```
pip install devart-postgresql-connector
```

Install the connector from a wheel file

1. Download the zip archive.
2. Extract the contents of the archive.
3. Open Command Prompt.
4. Verify that you have the pip package installer on your system using the `py -m pip --version` command. If you don't have it, run the following command to install pip.

```
python -m ensurepip --upgrade
```

5. In Command Prompt, navigate to the directory that contains the extracted wheel packages.
6. Install the package:

   - Windows 32-bit

```
pip install devart_postgresql_connector-1.0.1-cp312-cp312-win32.whl
```

- Windows 64-bit

```
pip install devart_postgresql_connector-1.0.1-cp312-cp312-win_amd64.whl
```

4.2 Linux

Install the connector on Linux

You can install the connector from the Python Package Index (PyPI) or a wheel (.whl) file.

Install the connector from PyPI

1. Open a terminal window.
2. Verify that you have the pip package installer on your system using the `py -m pip --version` command. If you don't have it, run the following command to install pip.

```
python -m ensurepip --upgrade
```

3. Install the package.

```
pip install devart-postgresql-connector
```

Install the connector from a wheel file

1. Download the zip archive.
2. Extract the contents of the archive.
3. Open a terminal window.
4. Verify that you have the pip package installer on your system using the `py -m pip --version` command. If you don't have it, run the following command to install pip.

```
python -m ensurepip --upgrade
```

5. In terminal, navigate to the directory that contains the extracted wheel package.
6. Install the package.

```
pip install devart_postgresql_connector-1.0.1-cp312-cp312-manylinux_2_34_x86
```

4.3 macOS

Install the connector on macOS

You can install the connector from the Python Package Index (PyPI) or a wheel (.whl) file.

Install the connector from PyPI

1. Open a terminal window.
2. Verify that you have the pip package installer on your system using the `py -m pip --version` command. If you don't have it, run the following command to install pip.

```
python -m ensurepip --upgrade
```

3. Install the package.

```
pip install devart-postgresql-connector
```

Install the connector from a wheel file

1. Download the zip archive.
2. Extract the contents of the archive.
3. Open a terminal window.
4. Verify that you have the pip package installer on your system using the `py -m pip --version` command. If you don't have it, run the following command to install pip.

```
python -m ensurepip --upgrade
```

5. In terminal, navigate to the directory that contains the extracted wheel package.
6. Install the package.

```
pip install devart_postgresql_connector-1.0.1-cp312-cp312-macosx_10_9_univer
```

5 Activation

5.1 Activate a license

Activate a license

1. Obtain an activation key using either of the following methods:
   - Copy the activation key that you received in an order confirmation email.
   - Obtain the activation key on the customer portal:
     1. Log in to the customer portal using the login credentials from an order confirmation

email.

2. Click the name of the purchased product on the **Products** page to view the license details.
3. Click **Copy to clipboard** under **Activation key**.

2. Start the Python shell.

3. Import the module.

```
import devart.postgresql
```

4. Specify the activation key using the `activate()` module method.

```
devart.postgresql.license.activate("<your_activation_key>")
```

5. (Optional) View the license details.

```
print(devart.postgresql.license.summary)
```

5.2 View the license details

View the license details

1. Start the Python shell.

2. Import the `devart.postgresql` module.

```
import devart.postgresql
```

3. Print the value of the `summary` module attribute.

```
print(devart.postgresql.license.summary)
```

5.3 Deactivate a license

Deactivate a license

1. Start the Python shell.

2. Import the `devart.postgresql` module.

```
import devart.postgresql
```

3. Deactivate your license using the `deactivate` module method.

```
devart.postgresql.license.deactivate()
```

6 Using the connector

6.1 Connecting to PostgreSQL

Connecting to PostgreSQL

To establish a connection to a PostgreSQL database, import the connector and use the `connect()` method with your connection parameters.

Step 1. Import the connector

First, import the PostgreSQL connector module:

```
import devart.postgresql as postgresql
```

Step 2. Establish a connection

Call the `connect()` method and obtain a `connection` object.

```
my_connection = postgresql.connect(
    Server="your_server",
    Database="your_database",
    UserId="your_username",
    Password="your_password"
)
```

Replace the example values with your actual connection values.

For more information, see Connection parameters.

6.2 Connection pooling

Connection pooling

The connector supports connection pooling, which improves performance and scalability by reusing existing database connections. Instead of repeatedly opening and closing connections, the connector retrieves a connection from a pre-established pool, reducing the time and resources required to establish new connections. This is especially beneficial in scenarios that involve frequent connect/disconnect operations.

When you call the `close()` method on a connection object, the connection remains alive and is returned to the pool. When a new connection is created using the `connect()` method, the module retrieves a connection from the pool—unless the pooler has already detected the connection as severed and marked it as invalid. If the pool is empty or lacks a valid connection, a new one is established.

Enable connection pooling

To enable connection pooling, set the value of the `connection_pool.enabled` module attribute to `True`.

The `connection_pool.enabled` attribute is global. If pooling is enabled, all new connections are pooled. Pooling can be disabled for a particular connection using the `DisablePooling` connection string parameter.

```
my_connection = devart.postgresql.connect("Server=your_server;Database=your_
```

Configure connection pooling

You can configure connection pool behavior using the following attributes:

- `connection_pool.min_size`
- `connection_pool.max_size`
- `connection_pool.lifetime`
- `connection_pool.validate`

For more information about the attributes, see the connection pool class.

The following example sets the attributes for the default connection pool, which implicitly has ID `0`.

```
devart.postgresql.connection_pool.min_size = 0
devart.postgresql.connection_pool.max_size = 1000
devart.postgresql.connection_pool.lifetime = 60000
devart.postgresql.connection_pool.validate = True
devart.postgresql.connection_pool.enabled = True
```

Multiple connection pools

You can define multiple connection pools with different settings. To define settings for a connection pool with a particular ID, use the syntax `connection_pool[pool_id: int]`, where `pool_id` is the ID of the pool. You can also pass the `PoolId` connection string parameter to specify which connection pool needs to be used for a particular connection.

```
devart.postgresql.connection_pool[42].max_size = 100
devart.postgresql.connection_pool[42].lifetime = 120000
devart.postgresql.connection_pool.enabled = True
```

```
my_connection = devart.postgresql.connect("Server=your_server;Database=your_
```

Important: Connections are placed in the same pool when all parameters in the connection string are identical. If the parameters differ, connections are placed in separate pools—even if they share the same pool ID. The connector creates a new pool when a connection uses an existing pool ID but has different connection parameters.

Connection validation

Database connections in a pool are validated every 30 seconds to ensure that a broken connection isn't returned from the pool when a connection object is constructed. Invalid connections are destroyed. The connection pooler also validates connections when they are added or released back into the pool (for example, when you call the `connection.close()` method).

If you set the `validate` attribute to `True`, connections are also validated when they are drawn from the pool. In the event of a network issue, all connections to the database may become broken. Therefore, if a fatal error is detected in one connection from the pool, the pooler validates all connections in the pool.

Idle timeout

The pooler removes a connection from the pool after it's been idle for approximately 4 minutes. If no new connections are added to the pool during this time, it becomes empty to save the resources. If you set the `min_size` attribute to a non-zero value, the pool doesn't destroy all idle connections and become empty unless the remaining connections are marked as invalid.

Maximum pool size

The `max_size` pool attribute limits the number of connections that can be stored in a pool at the same time. When the maximum number of connections in a pool is reached, all future database connections are destroyed once the connection object releases them.

Connection lifetime

You can limit the connection lifetime using the `lifetime` attribute. When a connection object is closed, and a database connection is returned to the pool, the creation time of the connection is compared with the current time. If this timespan exceeds the lifetime value, the connection is destroyed. This technique serves for load balancing.

6.3 Querying data

Querying data

Once connected to PostgreSQL, you can execute SQL queries to retrieve data from your PostgreSQL database.

Execute a query

1. Create a `cursor` object using the `cursor()` connection method.

```
my_cursor = my_connection.cursor()
```

2. Execute a SQL query using the `execute()` cursor method.

```
my_cursor.execute("SELECT * FROM employees")
```

3. Retrieve results using one of the `fetch*()` methods.

```
for row in my_cursor.fetchall():
    print(row)
```

Parameterized queries

You can use parameterized queries to pass variable values to your SQL statements. This allows you to reuse the same query with different data and helps to prevent SQL injection attacks.

Pass parameters as a list or tuple to the `execute()` method:

```
query = "SELECT Id, Name FROM Contact WHERE Name = ? AND Email = ?"
params = ["Jordan Sanders", "jordansanders@example.com"]
my_cursor.execute(query, params)
results = my_cursor.fetchall()
for row in results:
    print(row)
```

Each placeholder `?` in the query is replaced with a corresponding value from the parameter list.

7 Connection parameters

Connection parameters

The following table describes the PostgreSQL connection parameters you can use in the `connect()` module method.

Parameter	Description
`Server`	The server name or IP address
`Port`	The port number. The default value is `5432`.
`UserId`	The name of the database user
`Password`	The password of the database user
`Database`	The name of the database
`Schema`	The schema search path. The default value is `public`.
`Protocol`	The version of the PostgreSQL messaging protocol. The possible values are `pvAuto`, `pv30`, and `pv20`. The default value is `pvAuto`.
`PoolId`	The ID of a connection pool that will be used for a particular connection
`DisablePooling`	Disables connection pooling for a particular connection. The possible values are `True` and `False`. The default value is `False`.

8 Secure connection

8.1 SSL/TLS connection

Connecting to PostgreSQL using SSL/TLS

Transport Layer Security (TLS) is a security protocol for accessing remote machines over untrusted networks. A primary use case of TLS is encrypting the communication between web applications and servers. It runs on top of TCP/IP to secure client-server communications and allows a TLS-enabled client to authenticate itself to a TLS-enabled server and vice versa. TLS evolved from a previous encryption protocol called Secure Sockets Layer (SSL), and the terms TLS and SSL are sometimes used interchangeably.

During server authentication, the client application uses public-key cryptography (PKI) algorithms to verify the server's identity by checking that the server's certificate is issued by a trusted certificate authority (CA) and proves the ownership of the public key. Similarly, TLS client authentication allows the server to validate the client's identity. The client and server can also authenticate each other using self-signed certificates. However, you will only want to use a self-signed certificate for an internal network or a development server.

After establishing a TLS connection, the client and server can exchange symmetrically encrypted messages with a shared secret key. TLS is the recommended method for establishing a secure connection to PostgreSQL due to its more straightforward configuration and higher performance than SSH.

For more information about securing TCP/IP connections with TLS, see Secure TCP/IP connections in the PostgreSQL documentation.

Enable TLS on a connection

1. Import the module.

```
import devart.postgresql
```

2. Connect to a database using the `connect()` module method and obtain a `connection` object.

```
my_connection = devart.postgresql.connect(
    Server="your_server",
    Database="your_database",
    UserId="your_username",
    Password="your_password",
    UseSSL="True",
    SSLCACert="path_to_ca_cert",
    SSLCert="path_to_client_cert",
    SSLKey="path_to_client_key"
)
```

TLS parameters

The following table describes the TLS connection parameters.

Parameter	**Description**
`UseSSL`	Enables TLS connections.

`SSLCACert`	The CA certificate
`SSLCert`	The client certificate
`SSLKey`	The client private key
`SSLIgnoreServerCertificateValidity`	Specifies whether to verify the server certificate validity period during a TLS handshake. The possible values are `True` and `False`. The default value is `True`.
`SSLIgnoreServerCertificateConstraints`	Specifies whether to verify the server certificate for compliance with constraints during a TLS handshake. The possible values are `True` and `False`. The default value is `True`.
`SSLTrustServerCertificate`	Specifies whether to verify the server certificate chain during a TLS handshake. By default, the connector verifies the entire certificate chain. The possible values are `True` and `False`. If the parameter is set to `True`, the connector will bypass walking the certificate chain to validate trust.
`SSLIgnoreServerCertificateInsecurity`	Specifes whether to verify the server certificate signature security during a TLS handshake. The possible values are `True` and `False`. The default value is `False`.
`SSLMode`	Specifies whether and with what priority a TLS connection will be negotiated with the

	server. The possible values are: • `smRequire` – (Default) Only TLS connections allowed • `smPrefer` – Negotiates trying first a TLS connection, then if that fails, tries a regular non-TLS connection. • `smAllow` – Negotiates trying first a non-TLS connection, then if that fails, tries a TLS connection. • `smVerifyCA` – Verifies server identity by validating the server certificate chain up to the root certificate installed on the client machine. • `smVerifyFull` – Verifies server identity by validating the server certificate chain up to the root certificate installed on the client machine and validates that the server hostname matches the server certificate.

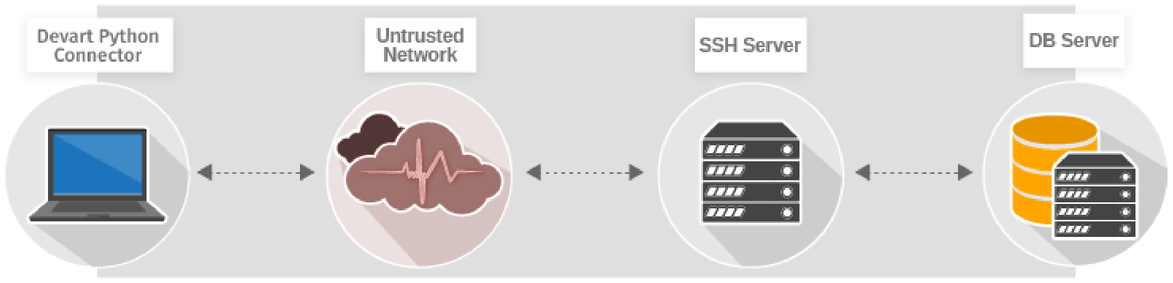
8.2 SSH tunneling

Connecting to PostgreSQL using SSH

Secure Shell (SSH) is a cryptographic protocol for secure remote login, command execution, and file transfer over untrusted networks. It uses a client-server model to authenticate two parties and encrypt the data between them. All user authentication, commands, output, and file transfers are encrypted to protect against attacks in the network. The client and server authenticate to each other and exchange commands and output. SSH uses symmetric encryption, assymetric encryption, and hashing to secure the transferred data.

In symmetric key cryptography, a single key is used by both the sending and receiving parties to encrypt and decrypt messages. Assymetric encryption, on the other hand, requires two associated keys—the private key and the public key. The public key encrypts messages that can only be decrypted with the private key. The public key can be freely shared with any trusted party, while the private key must be kept secret. The client public key is stored in a location that is accessible by the SSH server, allowing the client to authenticate the server. Similarly, the server public key is placed on the client side, enabling the server to authenticate the client. Assymetric encryption is used during the initial key exchange process to generate a shared secret (session key) that encrypts messages for the duration of the session.

Our connector implements the SSH client functionality that enables you to connect to the SSH server on the remote machine. The SSH server listens for incoming TCP connections. The SSH client begins the initial TCP handshake with the server and verifies the server's identity. The client and server agree upon the encryption protocol and negotiate a session key. The server then authenticates the client and spawns a shell. The SSH server authenticates the client and allows the connector to establish a secure direct connection to PostgreSQL.



Enable SSH connections

1. Import the module.

```
import devart.postgresql
```

2. Connect to a database using the `connect()` module method and obtain a `connection` object.

```
my_connection = devart.postgresql.connect(
    Server="your_server",
    Database="your_database",
    UserId="your_username",
    Password="your_password",
    UseSSH="True",
    SSHHostName="your_ssh_host",
    SSHUserName="your_ssh_user",
    SSHClientKey="path_to_priv_client_key",
    SSHServerKey="path_to_pub_host_key",
    SSHStoragePath="path_to_ssh_storage"
)
```

Note: You don't have to install the SSH client on the client machine since the connector already implements the SSH client functionality.

SSH parameters

The following table describes the SSH connection parameters.

Parameter	**Description**
`UseSSH`	Enables SSH connections.
`SSHHostname`	The hostname or IP address of the SSH server
`SSHPort`	The SSH port number. The default port is 22.
`SSHUserName`	The name of the SSH user
`SSHPassword`	The password of the SSH user. It's recommended that you also use an SSH key.
`SSHClientKey`	The path to the client private key
`SSHClientKeyPassword`	The passphrase for the client private key
`SSHServerKey`	The path to the public host key
`SSHStoragePath`	The location where the connector will store its configuration files on the client machine

8.3 HTTP tunneling

Connecting to PostgreSQL using HTTP tunneling

You can use HTTP(s) tunneling when the client needs access to a website database from a remote machine, but direct access to the database server on the specified port is forbidden. In conditions of restricted connectivity, when the database server is hidden behind a firewall or you need to transmit private network data through a public network, you can set up an HTTP tunnel to create a direct network link between the two locations.

If the firewall allows HTTP(S) connections, you can use the connector with a properly configured web server to connect to the database server. The connector comes with a PHP script that enables access to the database server through an HTTP tunnel. The `tunnel.php` script file is located in the `\site-packages\devart\postgresql\http\` directory. You need to deploy it on a web server which will act as an HTTP tunneling server. The script must be accessible through HTTP—you can verify its accessibility in any web browser. The web server must support PHP 5 or later.

You can also use the Secure Shell forwarding, or SSH to tunnel network traffic. However, SSH is designed to encrypt traffic rather than traverse firewalls. Note that traffic tunneling or encryption increases the CPU and bandwidth usage. It is recommended that you use direct connection whenever possible.

Enable HTTP tunneling

1. Import the module.

```
import devart.postgresql
```

2. Connect to a database using the `connect()` module method and obtain a `connection` object.

```
    my_connection = devart.postgresql.connect(
    Server="your_server",
    Database="your_database",
    UserId="your_username",
    Password="your_password",
    UseHttp="True",
    HttpUrl="https://hostname/tunnel.php",
    HttpTrustServerCertificate="True"
)
```

HTTP tunneling parameters

The following table describes the HTTP tunneling parameters.

Parameter	Description
`UseHttp`	Enables HTTP tunneling.
`HttpUrl`	The URL of the PHP script for HTTP tunneling
`HttpUserName`	The username for the password-protected directory that contains the HTTP tunneling script
`HttpPassword`	The password for the password-protected directory that contains the HTTP tunneling script
`HttpTrustServerCertificate`	Specifies whether to verify the server certificate chain during a TLS handshake. By default, the connector verifies the entire certificate chain. The possible values are `True` and `False`. If the parameter is set to `True`, the connector will bypass walking the certificate chain to validate trust.

HTTP proxy parameters

If the HTTP tunneling server isn't directly accessible from the client machine, you can connect through a proxy server.

The following table describes the HTTP proxy parameters.

Parameter	Description
`ProxyHostName`	The hostname or IP address of the proxy server
`ProxyPort`	The proxy port
`ProxyUserName`	The username for proxy authentication
`ProxyPassword`	The password for proxy authentication

9 Data types

Data types

The following table describes the supported PostgreSQL data types and their mapping to the Python data types. The type codes returned in the `description` cursor attribute can be used in the `addtypecast()` cursor method.

PostgreSQL data type	Type code	Python data type
character	507	str
character varying	508	str
text	517	str
smallint	515	int
integer	511	int
bigint	501	int
serial	516	int
bigserial	502	int
real	514	float
double precision	510	float
numeric	513	numeric
date	509	datetime.date
time	518	datetime.time
time with time zone	519	datetime.time
timestamp	520	datetime.datetime
timestamp with time zone	521	datetime.datetime
bytea	506	binary
point	562	point
lseg	559	lseg
line	558	line
path	561	path

polygon	563	polygon
box	553	box
circle	555	circle

10 Class reference

10.1 Module class

Module class

The `module` class provides methods, global properties, exceptions, constructors, and type objects to be used by all connections created in the module.

- Methods
  - connect()
  - activate()
  - deactivate()
- Globals
  - apilevel
  - threadsafety
  - paramstyle
  - connection_pool
  - summary
- Exceptions
  - Warning
  - Error
  - InterfaceError
  - DatabaseError
  - DataError
  - OperationalError

- IntegrityError
- InternalError
- ProgrammingError
- NotSupportedError

- Constructors
  - Date()
  - Time()
  - Timestamp()
  - DateFromTicks()
  - TimeFromTicks()
  - TimestampFromTicks()
  - Binary()
  - PostgreSQL-specific constructors
- Type objects
  - STRING
  - BINARY
  - NUMBER
  - DATETIME
  - ROWID
  - binary
  - NaN
  - Infinity
  - PostgreSQL-specific type objects

Methods

`connect(connection string|connection parameters)`

Creates a new connection to the database.

Arguments

`connection string`

A string literal of form `"parameter=value;parameter=value"`

`connection parameters`

A sequence of named parameters

Connection parameters

For the full list of supported connection parameters, see Connection parameters.

Return value

Returns a `connection` object.

Code sample

```
# establising a connection using a connection string
connection1 = devart.postgresql.connect("Server=your_server;Database=your_da
# establising a connection using named parameters
connection2 = devart.postgresql.connect(
    Server="your_server",
    Database="your_database",
    UserId="your_username",
    Password="your_password"
)
```

`license.activate(activation key)`

Activates a license.

Arguments

`activation key`

A string literal that contains the activation key.

Remarks

See Activate a license for activation instructions.

`license.deactivate()`

Deactivates a license.

Arguments

This method has no arguments.

Remarks

See Deactivate a license for deactivation instructions.

Globals

`apilevel`

The DB API level supported by the module. Returns a string value `"2.0"`.

`threadsafety`

The thread safety level of the module. Returns an integer value `2` meaning threads may share the module and connections.

`paramstyle`

The type of parameter marker formatting expected by the module. Returns a string value `"named"` indicating that the module supports named style parameters, such as `...WHERE name=:name`.

`connection_pool`

Returns the connection pooling configuration.

`license.summary`

Returns the license details.

Exceptions

The module provides the following exceptions to make all error information available.

`Warning`

This exception is raised for important warnings like data truncations while inserting, etc. The `Warning` exception is a subclass of the Python `Exception` class.

`Error`

This exception is the base class of all error exceptions. You can use it to catch all errors with a single `except` statement. The `Error` exception is a subclass of the Python

`Exception` class.

`InterfaceError`

This exception is raised for errors that are related to the database interface rather than the database itself. The `InterfaceError` exception is a subclass of `Error`.

`DatabaseError`

This exception is raised for errors that are related to the database. The `DatabaseError` exception is a subclass of `Error`.

`DataError`

This exception is raised for errors caused by issues with the processed data like division by zero, numeric value out of range, etc. The `DataError` exception is a subclass of `DatabaseError`.

`OperationalError`

This exception is raised for errors that are related to the database operation and not necessarily under the control of the developer, for example, an unexpected disconnect occurs, the data source name isn't found, a transaction couldn't be processed, a memory allocation error occurred during processing, etc. The OperationalError exception is a subclass of `DatabaseError`.

`IntegrityError`

This exception raised when the relational integrity of the database is affected, for example, a foreign key check fails. The `IntegrityError` exception is a subclass of `DatabaseError`.

`InternalError`

This exception is raised when the database encounters an internal error, for example, the cursor isn't valid anymore, the transaction is out of sync, etc. The `InternalError` exception is a subclass of `DatabaseError`.

`ProgrammingError`

This exception is raised for programming errors, for example, table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. The `ProgrammingError` exception is a subclass of `DatabaseError`.

`NotSupportedError`

This exception is raised when a method or database API isn't supported by the database, for example, requesting a `rollback()` on a connection that doesn't support transactions or has transactions turned off. The `NotSupportedError` exception is a subclass of `DatabaseError`.

The complete exception inheritance tree:
`Exception`

`Warning`

`Error`

`InterfaceError`

`DatabaseError`

`DataError`

`OperationalError`

`IntegrityError`

`InternalError`

`ProgrammingError`

`NotSupportedError`

Constructors

The module provides the following constructors for creating date/time objects. The created date/time objects are implemented as Python `datetime` module objects.

`Date(year, month, day)`

Creates an object that holds a date value.

Arguments

`year`

`month`

`day`

Values of type `int` that specify the year, month, and day.

Return value

Returns a `datetime.date` object.

`Time(hour, minute, second[, timezone])`

Creates an object that holds a time value.

Arguments

`hour`

`minute`

Values of type `int` that specify hours and minutes.

`second`

An `int` value that specifies seconds or a `float` value that specifies seconds and microseconds.

`timezone`

(Optional) A value of type `datetime.tzinfo` that specifies a timezone. The value can be `None`.

Return value

Returns a `datetime.time` object.

`Timestamp(year, month, day[, hour[, minute[, second[, timezone]]]])`

Creates an object that holds a timestamp value.

Arguments

`year`

`month`

`day`

Values of type `int` that specify the year, month, and day.

`hour`

`minute`

(Optional) Values of type `int` that specify hours and minutes.

`second`

(Optional) An `int` value that specifies seconds or a `float` value that specifies seconds and microseconds.

`timezone`

(Optional) A value of type `datetime.tzinfo` that specifies a timezone. The value can be `None`.

Return value

Returns a `datetime.datetime` object.

`DateFromTicks(ticks)`

Creates an object that holds a date value from the given ticks value (the number of seconds since the Unix epoch). For more information, see the `time` module in the standard Python documentation.

Arguments

`ticks`

A value of type `float` that specifies number of seconds since the Unix epoch.

Return value

Returns a `datetime.date` object.

`TimeFromTicks(ticks)`

Creates an object that holds a time value from the given ticks value (number of seconds since the Unix epoch). For more information, see the `time` module in the standard Python documentation.

Arguments

`ticks`

A value of type `float` that specifies number of seconds since the Unix epoch.

Return value

Returns a `datetime.time` object.

`TimestampFromTicks(ticks)`

Creates an object that holds a timestamp value from the given ticks value (number of seconds since the Unix epoch). For more information, see the `time` module in the standard Python documentation.

Arguments

`ticks`

A value of type `float` that specifies number of seconds since the Unix epoch.

Return value

Returns a `datetime.datetime` object.

The module provides the following additional constructors.

`Binary(value)`

Creates an object that holds binary data.

Arguments

`value`

A value of type `str`, `bytes`, `bytearray`, `array.array`, or a `binary` object.

Return value

Returns a binary object.

PostgreSQL-specific constructors

Additional constructors for handling PostgreSQL-specific database types.

Type objects

The module provides the following type objects to create mapping between the PostgreSQL database types and Python types. You can use these type objects as arguments for the `addtypecast()` cursor method to define a data type cast rule to use when fetching data from the cursor. They can also be used to determine the Python types of the result columns returned by the `execute*()` cursor methods.

STRING

This type object describes string-based columns in a database.

BINARY

This type object describes binary columns in a database.

NUMBER

This type object describes numeric columns in a database.

DATETIME

This type object describes date/time columns in a database.

ROWID

This type object describes the `row ID` column in a database.

Code sample

```
cursor.execute("select column1 from table1")
# check if the first column in the result set is string-based so that its va
if cursor.description[0].type_code in postgresql.STRING:
    # do something
```

The module provides the following additional type objects.

`binary`

This type object describes an object that holds binary data. By default, this type object is used to fetch BLOB-based columns from the cursor. You can also create a `binary` object using the `Binary()` constructor.

Attributes

`value`

A value of type `bytes` that represents binary data. This is a read/write attribute that accepts values of type `str`, `bytes`, `bytearray`, `array.array`, and `binary`.

`NaN`

This type object describes a special numeric value `not-a-number` that can be used in operations with `numeric` objects.

Remarks

`NaN` can be interpreted as an undefined or non-representable value. It doesn't support comparison operators except `==` and `!=`.

`Infinity`

This type object describes a special numeric value `positive infinity` that is larger than any natural number. It can be used in operations with `numeric` objects.

Remarks

The connector doesn't provide any special type object for `negative infinity` that represents the smallest number. For negative infinity, use a `-Infinity` value.

`PostgreSQL-specific type objects`

Additional type objects for handling PostgreSQL-specific database types.

10.2 Module additions

Module additions

The module provides several additional constructors and type objects for handling PostgreSQL-specific database types.

- Constructors
  - Numeric()
  - Point()
  - Lseg()
  - Line()
  - Path()
  - Polygon()
  - Box()
  - Circle()
  - Vector()
  - Halfvec()
  - Sparsevec()
- Type objects
  - numeric
  - point
  - lseg
  - line
  - path
  - polygon
  - box
  - circle
  - vector
  - halfvec

- sparsevec

Constructors

`Numeric(weight, scale, sign, digits|value)`

Creates an object that holds a number with a very large number of digits.

Arguments

`weight, scale, sign, digits`

Corresponding values for the `numeric` object attributes.

`value`

A value of type `str`, `int`, `float`, or a `numeric` object.

Return value

Returns a `numeric` object.

Remarks

The `value` argument can also contain special values `Infinity`, `-Infinity`, and `NaN` (or their string representations). These are adapted from the IEEE 754 standard, and represent the `positive infinity`, `negative infinity`, and `not-a-number`, respectively.

`Point(x, y|value)`

Creates an object that holds coordinates of a point on a plane.

Arguments

`x, y`

Values of type `int` or `float` that specify the coordinates of the point.

`value`

A string literal of form `(x, y)` where`x` and `y` are the respective coordinates, or a `point` object.

Return value

Returns a `point` object.

`Lseg(start, end|startx, starty, endx, endy|value)`

Creates an object that holds information about a line segment.

Arguments

`start, end`

Values of type `point` that specify the endpoints of the segment.

`startx, starty, endx, endy`

Values of type `int` or `float` that specify the coordinates of the endpoints.

`value`

A string literal of form `[(x1, y1), (x2, y2)]` where `(x1, y1)` and `(x2, y2)` are the respective endpoints, or a `lseg` object.

Return value

Returns a `lseg` object.

`Line(a, b, c|value)`

Creates an object that holds information about a line.

Arguments

`a, b, c`

Values of type `int` or `float` that specify the coefficients of the linear equation `ax + by + c = 0` that describes the line.

`value`

A string literal of form `{a, b, c}` where `a`, `b`, and `c` are the respective coefficients, or a [`line`] object.

Return value

Returns a `line` object.

Remarks

The `a` and `b` arguments must not both be zero.

`Path(points|value)`

Creates an object that holds an array of connected line segments.

Arguments

`points`

A `list` of `point` objects that specify the endpoints of the segments that form the path.

`value`

A string literal of form `[(x1, y1), ..., (xn, yn)]` where `(xn, yn)` are sequential endpoints of all the segments, or a `path` object.

Return value

Returns a `path` object.

`Polygon(points|value)`

Creates an object that holds information about a polygon.

Arguments

`points`

A `list` of `point` objects that specify the vertexes of the polygon.

`value`

A string literal of form `((x1, y1), ..., (xn, yn))` where `(xn, yn)` are sequential coordinates of all the vertexes of the polygon, or a `polygon` object.

Return value

Returns a `polygon` object.

`Box(upperright, lowerleft|upperrightx, upperrighty, lowerleftx, lowerlefty|value)`

Creates an object that holds information about the rectangle.

Arguments

`upperright, lowerleft`

Values of type `point` that specify the upper right and lower left corners of the rectangle.

`upperrightx, upperrighty, lowerleftx, lowerlefty`

Values of type `int` or `float` that specify the coordinates of the rectangle corners.

`value`

A string literal of form `((x1, y1), (x2, y2))` where `(x1, y1)` and `(x2, y2)` are coordinates of two opposite corners of the rectangle, or a `box` object.

Return value

Returns a `box` object.

Remarks

There are no strict requirements that the input values must be exactly upper right and lower left corners. Any two opposite corners can be supplied, but you should keep in mind that when saved into a database, the values will be reordered as needed to store the upper right and lower left corners, in that order. Therefore, on subsequent reading of the stored value, you may get an object that doesn't match the one that was written.

`Circle(center, radius|centerx, centery, radius|value)`

Creates an object that holds information about a circle.

Arguments

`center`

A value of type `point` that specifies the center point of the circle.

`radius`

A value of type `int` or `float` that specifies the radius of the circle.

`centerx, centery, radius`

Values of type `int` or `float` that specify coordinates of the center point of the circle and it's radius, respectively.

`value`

A string literal of form `<(x, y), r>` where `x` and `y` are the center point coordinates and `r` is the radius, or a `circle` object.

Return value

Returns a `circle` object.

`Vector(item)`

Creates an object that holds information about vectors.

Arguments

`items`

A list of floating-point numbers that make up vectors.

Return value

Returns a `vector` object.

`Halfvec(item)`

Creates an object that holds information about half-precision vectors.

Arguments

`items`

A list of floating-point numbers that form half-precision vectors.

Return value

Returns a `halfvec` object.

`Sparsevec(count, elementcount, items)`

Creates an object that holds information about sparse vectors.

Arguments

`count`

The dimensions of a vector.

`elementcount`

The number of non-zero items in a vector.

`items`

A dictionary of index value pairs for each non-zero item.

Return value

Returns a `sparsevec` object.

Type objects

`numeric`

This type object describes an object that holds numbers with a very large number of digits. By default, this type object is used to fetch `numeric` or `decimal` columns from the cursor. You can also create a `numeric` object using the `Numeric()` constructor.

Attributes

`digits`

A `tuple` of integers that specifies all the digits of a `numeric` object. Each element of the `tuple` contains four digits.

`weight`

The index of the last `digits` element that refers to the integer part of a `numeric` object.

`scale`

The number of digits in the fractional part of a `numeric` object.

`sign`

Specifies whether a `numeric` object is positive or negative, or has a special value. The possible values are:

- `0x0000` – The `numeric` value is positive.
- `0x4000` – The `numeric` value is negative.
- `0xC000` – The `numeric` is NaN.
- `0xD000` – The `numeric` value is `Infinity`.
- `0xF000` – The `numeric` value is `-Infinity`.

Examples

```
num1 = devart.postgresql.Numeric('-12345678.1234')
repr(num1)
'<devart.postgresql.numeric object at 0x...; weight=1, scale=4, sign=16384,
```

The value `12345678.1234` is split into four-digit groups, which are then converted to integers. In this example, there are three `digits` elements: `1234`, `5678`, and `1234`. The first two `digits` elements contain the integer part of the value, so the `weight` is `1`. The fractional part of the value consists of four digits, so the `scale` is `4`. Since the value is negative, the `sign` is `0x4000`.

```
>>> num2 = devart.postgresql.Numeric('NaN')
repr(num2)
'<devart.postgresql.numeric object at 0x...; weight=0, scale=0, sign=49152,
```

Since the value is NaN, the `numeric` object contains no data other than the `sign` attribute, which is `0xC000`.

`point`

This type object describes an object that holds coordinates of a point on a plane. By default, this type object is used to fetch `point` columns from the cursor. You can also create a `point` object using the `Point()` constructor.

Attributes

`x`

`y`

The coordinates of the point.

`lseg`

This type object describes an object that holds information about a line segment. By default, this type object is used to fetch `lseg` columns from the cursor. You can also create a `lseg` object using the `Lseg()` constructor.

Attributes

`start`

`end`

The endpoints of the line segment.

`line`

This type object describes an object that holds information about a line. By default, this type object is used to fetch `line` columns from the cursor. You can also create a `line` object using the `Line()` constructor.

Attributes

`a`

`b`

`c`

The coefficients of the linear equation `ax + by + c = 0` that describes the line.

`path`

This type object describes an object that holds an array of connected line segments. By default, this type object is used to fetch `path` columns from the cursor. You can also create a `path` object using the `Path()` constructor.

Attributes

`points`

A `list` of endpoints of all the segments of the path.

`polygon`

This type object describes an object that holds information about a polygon. By default, this type object is used to fetch `polygon` columns from the cursor. You can also create a `polygon` object using the `Polygon()` constructor.

Attributes

`points`

A `list` of coordinates of all the vertexes of the polygon.

`box`

This type object describes an object that holds information about a rectangle. By default, this type object is used to fetch `box` columns from the cursor. You can also create a `box` object using the `Box()` constructor.

Attributes

`upperright`

`lowerleft`

The opposite corners of the rectangle.

`circle`

This type object describes an object that holds information about a circle. By default, this type object is used to fetch `circle` columns from the cursor. You can also create a `circle` object using the `Circle()` constructor.

Attributes

`center`

The center point of the circle.

`radius`

The radius of the circle.

`vector`

This type object describes an object that holds information about a vector. By default, this type object is used to fetch `vector` columns from the cursor. You can also create a `vector` object using the `Vector()` constructor.

Attributes

`items`

A list of floating point numbers that make up a vector.

`halfvec`

This type object describes an object that keeps information about half-precision vectors. By default, this type object is used to fetch `halfvec` columns from the cursor. You can also create a `halfvec` object using the `Halfvec()` constructor.

Attributes

`items`

A list of floating-point numbers that form half-precision vectors.

`sparsevec`

This type object describes an object that stores information about sparse vectors. By default, this type object is used to fetch `sparsevec` columns from the cursor. You can also create a `sparsevec` object using the `Sparsevec()` constructor.

Attributes

`count`

The dimensions of a vector.

10.3 Connection class

Connection class

The `connection` class encapsulates a database session. It provides methods for creating cursors, type casting, and transaction handling. Connections are created using the `connect()` module method.

- Methods
  - cursor()
  - commit()
  - rollback()
  - addtypecast()
  - cleartypecast()

Methods

`cursor()`

Creates a new cursor object, which is used to manage the context of fetch operations.

Arguments

This method has no arguments.

Return value

Returns a `cursor` object.

`commit()`

Commits any pending transaction to the database.

Arguments

This method has no arguments.

`rollback()`

Causes the database to roll back any pending transaction.

Arguments

This method has no arguments.

Remarks

Closing a connection without first committing changes causes an implicit rollback.

`addtypecast(database type|module type object|column name|description|dictionary[, Python type])`

Defines a data type cast rule to use when fetching data from the cursor.

Arguments

`database type`

An `int` value that specifies the database data type code. You can also pass multiple data type codes in a `tuple` or `list`.

`module type object`

A `module type object` that specifies the family of the database data types.

`column name`

A string literal that specifies the name of the database column. You can also pass multiple string literals in a `tuple` or `list`.

`description`

A `description` object that describes the column in a rowset. You can also pass multiple objects in a `tuple` or `list`.

`dictionary`

A `dictionary` of pairs `column name:Python type` that specifies individual cast rules for a set of columns. The method argument `Python type` can be omitted.

`Python type`

A Python type object that specifies the target type to which to cast the database type, or an `int` value which means that the column will be of type `str` and defines its maximum length.

Code sample

```
connection = devart.postgresql.connect("Server=your_server;Database=your_dat
# all database columns with data type code 511 (PostgreSQL database type int
connection.addtypecast(511, int)
# all numeric database columns will be fetched as strings
connection.addtypecast(devart.postgresql.NUMBER, str)
# data of "column1" will be fetched as a string
connection.addtypecast("column1", str)
# data of "column2" will be fetched as `int` and data of "column3" will be f
connection.addtypecast({"column2":int, "column3":50})
```

Remarks

The cast rule affects all cursors created within the connection. To define a cast rule for a particular cursor, use the `addtypecast()` cursor method. The type code of a database column can be obtained from the `type_code` attribute of the corresponding element of the `description` cursor attribute.

`cleartypecast()`

Removes all data type cast rules defined for the connection.

Arguments

This method has no arguments.

Remarks

This method doesn't remove cast rules defined for a particular cursor using the `addtypecast()` cursor method.

`close()`

Closes the connection.

Arguments

This method has no arguments.

Remarks

The connection becomes unusable after calling this method. The `InterfaceError` exception is raised if any operation is attempted with the connection. The same applies to all cursor objects trying to use the connection. Closing a connection prior to committing changes causes an implicit rollback.

Attributes

`connectstring`

A read-only attribute that returns a string literal of the form `"parameter=value;parameter=value"` that contains the parameters for the current connection.

Exceptions

The `connection` class provides a set of exception classes that exactly match the module exceptions. This simplifies error handling in environments with multiple connections.

10.4 Cursor class

Cursor class

The `cursor` class represents a database cursor, which is used to manage the context of fetch operations. This class provides methods for executing SQL statements and operating rowsets. Cursors are created using the `cursor()` connection method.

- Methods
  - setinputsizes()
  - execute()
  - executemany()
  - fetchone()
  - fetchmany()
  - fetchall()
  - next()
  - scroll()
  - addtypecast()
  - cleartypecast()
  - close()
  - setoutputsize()
- Attributes
  - connection
  - arraysize
  - rowtype
  - description
  - rowcount
  - rownumber
  - lastrowid

Methods

`setinputsizes([sizes])`

Defines parameter types for subsequent calls to the `execute*()` method.

Arguments

`sizes`

(Optional) A sequence (`list` or `tuple`) with one item for each input parameter. The item should be a type object that defines the type of the input parameter, or an integer value specifying the maximum length of the string parameter. If the item is `None`, the parameter type is determined by the value provided in the `execute*()` method.

Code sample

```
cursor = connection.cursor()
# in the further call to cursor.execute() the supplied parameters will be tr
cursor.setinputsizes(int, float, 20)
```

Remarks

Once set, the types of parameters are retained on subsequent calls to the `execute*()` method until the cursor is closed by calling `close()`. To clear the set parameter types, call the method with no arguments.

`execute(operation[, parameters])`

Prepares and executes a database operation.

Arguments

`operation`

A string literal that specifies the database command (SQL statement) to be executed.

`parameters`

(Optional) Can be specified as either:

- A sequence (`list` or `tuple`) of values, or

- A dictionary of `"parameter_name": parameter_value` pairs,

to be bound to the corresponding parameters of the operation.

Code sample

```
cursor = connection.cursor()
  cursor.execute("create table test_table(column1 integer, column2 integer)"
  cursor.execute("insert into test_table(column1, column2) values(:parameter
  cursor.execute("insert into test_table(column1, column2) values(:parameter
```

Remarks

The types of the input parameters can be pre-specified using the `setinputsizes()` method. To execute a batch operation that affects multiple rows in a single operation, use the `executemany()` method.

`executemany(operation[, sequence of parameters])`

Prepares and executes a batch database operation.

Arguments

`operation`

A string literal that specifies the database command (SQL statement) to be executed.

`parameters`

(Optional) A sequence (`list` or `tuple`) of parameter sets. Each parameter set can be:

- A sequence (`list` or `tuple`) of values, or
- A dictionary of `"parameter_name": parameter_value` pairs,

to be bound to the corresponding parameters of the operation.

Code sample

```
cursor = connection.cursor()
 cursor.execute(
  "create table test_table(column1 integer, column2 integer)")
 cursor.executemany(
  "insert into test_table(column1, column2) values(:parameter1, :parameter2)
```

```
cursor.executemany("insert into test_table(column1, column2) values(:parame
        [
         {"parameter1": 4, "parameter2": 4},
         {"parameter1": 5, "parameter2": 5},
         {"parameter1": 6, "parameter2": 6}
        ])
```

Remarks

The types of the input parameters can be pre-specified using the `setinputsizes()` method. This method is significantly faster than executing the `execute()` method in a loop.

`fetchone()`

Fetches the next row of a query result set.

Arguments

This method has no arguments.

Return value

Returns a single sequence (`tuple`, `list` or `dict` according to the `rowtype` value) that contains values for each queried database column, or `None` when no more data is available.

Remarks

The `ProgrammingError` exception is raised if the previous call to the `execute*()` method didn't produce any result set, or no call has been made yet.

`fetchmany([size=cursor.arraysize])`

Fetches the next set of rows of a query result.

Arguments

`size`

(Optional) The number of rows to fetch per call. If the number isn't specified, the `arraysize` attribute determines the number of rows to be fetched.

Return value

Returns a `list` of sequences (`tuples`, `lists` or `dicts` according to the `rowtype` value) for each result row. Each sequence contains values for each queried database column. An empty `list` is returned when no more rows are available.

Remarks

The `ProgrammingError` exception is raised if the previous call to the `execute*()` method didn't produce any result set, or no call has been made yet.

`fetchall()`

Fetches all remaining rows of a query result.

Arguments

This method has no arguments.

Return value

Returns a `list` of sequences (`tuples`, `lists` or `dicts` according to the `rowtype` value) for each result row. Each sequence contains values for each queried database column. An empty `list` is returned when no more rows are available.

Remarks

This method returns as many rows as are left in the result set, regardless of the `arraysize` value. The `ProgrammingError` exception is raised if the previous call to the `execute*()` method didn't produce any result set or no call has been made yet.

`next()`

Returns the next row from the currently executed SQL statement.

Arguments

This method has no arguments.

Return value

Returns a single `tuple` that contains values for each queried database column.

Remarks

This method uses the same semantics as `fetchone()`, except that the standard `StopIteration` exception is thrown if no more rows are available.

`scroll(value[, mode='relative'])`

Scrolls the cursor in the result set to a new position.

Arguments

`value`

An `int` value that specifies the new cursor position.

`mode`

(Optional) The value can be either `relative` or `absolute`. If the `mode` is `relative` (the default value), the `value` is taken as offset to the current position in the result set. If the `mode` is set to `absolute`, the `value` states an absolute target position.

Remarks

The `IndexError` exception is raised in case a scroll operation attempts to access an item beyond the bounds of the result set. In this case, the cursor position is left unchanged.

`addtypecast(database type|module type object|column name|description|dictionary[, Python type])`

Defines a data type cast rule to use when fetching data from the cursor.

Arguments

`database type`

An `int` value that specifies the database data type code. You can also pass multiple data type codes in a `tuple` or `list`.

`module type object`

A `module type object` that specifies the family of the database data types.

`column name`

A string literal that specifies the name of the database column. You can also pass multiple string literals in a `tuple` or `list`.

`description`

A `description` object that describes the column in a rowset. You can also pass multiple objects in a `tuple` or `list`.

`dictionary`

A `dictionary` of pairs `column name:Python type` that specifies individual cast rules for a set of columns. The method argument `Python type` can be omitted.

`Python type`

A Python type object that specifies the target type to which to cast the database type, or an `int` value, which means that the column will be treated as a `string` with the specified maximum length.

Code sample

```
cursor = connection.cursor()
# all columns with the data type code 511 (PostgreSQL type `integer`) will b
cursor.addtypecast(511, int)
# all numeric columns will be fetched as strings
cursor.addtypecast(postgresql.NUMBER, str)
# data of "column1" will be fetched as a string
cursor.addtypecast("column1", str)
# data of "column2" will be fetched as `int` and data of "column3" will be f
cursor.addtypecast({"column2":int, "column3":50})
```

Remarks

The cast rule affects only the current cursor. To define the cast rule for all cursors created within the connection, use the `addtypecast()` connection method. The type code of a database column can be obtained from the `type_code` attribute of the corresponding element of the `description` attribute.

`cleartypecast()`

Removes all data type cast rules defined for the cursor.

Arguments

This method has no arguments.

Remarks

This method doesn't remove cast rules defined for the entire connection using the `addtypecast()` connection method.

`close()`

Closes the cursor.

Arguments

This method has no arguments.

Remarks

After calling this method, the cursor can no longer be used. The `InterfaceError` exception is raised if any operation is attempted with the cursor.

`setoutputsize(int size[, int column])`

This method is provided for compatibility with the DB API 2.0 specification. It currently does nothing but is safe to call.

Attributes

`connection`

A read-only attribute that specifies the `connection` object to which the cursor belongs.

`arraysize`

A read/write attribute that specifies the number of rows to fetch at a time with the `fetchmany()` method.

Remarks

The default value of the attribute is `1`, meaning to fetch a single row at a time.

`rowtype`

A read/write attribute that specifies the type of rows fetched with the `fetch*()` method.

The possible attribute values are `tuple`, `list`, and `dict`.

Remarks

The default value of the attribute is `tuple`.

`description`

A read-only attribute that describes the columns in a rowset returned by the cursor.

Return value

Returns a `tuple` of `description` objects with the following attributes:

`name`

The name of the column in the rowset

`type_code`

The database type code that corresponds to the type of the column

`display_size`

The actual length of the column in characters for a character column, `None` otherwise

`internal size`

The size in bytes used by the connector to store the column data

`precision`

The total number of significant digits for a numeric column, `None` otherwise

`scale`

The number of digits in the fractional part for a numeric column, `None` otherwise

`null_ok`

`Py_True` if the corresponding database column accepts `NULL` values, `Py_False` otherwise

Remarks

The attribute is `None` for operations that don't return rows or if no operation has been invoked for the cursor via the `execute()` method yet. The `type_code` attribute can be used in the `addtypecast()` method to define a data type cast rule for the corresponding column.

`rowcount`

A read-only attribute that specifies the number of rows that the last `execute()` call produced by a `SELECT` statement or affected by `UPDATE` or `INSERT` statements.

Remarks

The value of this attribute is `-1` if no `execute()` call has been made on the cursor or the rowcount of the last operation cannot be determined.

`rownumber`

A read-only attribute that specifies the current 0-based index of the cursor in the result set.

Remarks

The next `fetch*()` retrieves rows starting from the current `rownumber` index. The attribute initial value is always `0`, regardless of whether the `execute()` call returned a rowset or not.

`lastrowid`

This read-only attribute is provided for compatibility with the DB API 2.0 specification. It currently returns `None`.

10.5 Connection pool class

Connection pool class

The `connection_pool` class is used to manage the connection pooling mechanism. This class provides properties for enabling and configuring pooling.

- Properties
  - enabled
  - max_size
  - min_size
  - lifetime
  - validate

Properties

`enabled`

Enables connection pooling.

Syntax

```
enabled = False | True
```

Remarks

Set `enabled` to `True` to enable connection pooling. The default value is `False`.

`max_size`

The maximum number of connections allowed in the pool

Syntax

```
max_size = int
max_size[pool_id: int] = int
```

Remarks

When the maximum number of connections in the pool is reached, new database connections will be destroyed instead of released back into the pool after you close them. The default value of `max_size` is `100`.

If no pool ID (`pool_id`) is specified, the maximum number of connections is set for the default connection pool. If the pool ID is specified, the maximum number of connections is set for the pool with the given ID.

`min_size`

The minimum number of connections maintained in the pool

Syntax

```
min_size = int
min_size[pool_id: int] = int
```

Remarks

Set this property to a non-zero value to prevent removing all connections from the pool after they have been idle for a long time. The default value of `min_size` is `0`.

If no pool ID (`pool_id`) is specified, the minimum number of connections is set for the default connection pool. If the pool ID is specified, the minimum number of connections is set for the pool with the given ID.

`lifetime`

The maximum time (in milliseconds) during which a database connection will be kept in the connection pool

Syntax

```
lifetime = int
lifetime[pool_id: int] = int
```

Remarks

The creation time of a connection is compared with the current time, and the connection is destroyed if that timespan exceeds the lifetime. If `lifetime` is set to `0` (by default), the lifetime of a connection is infinite.

If no pool ID (`pool_id`) is specified, the connection lifetime is set for the default connection pool. If the pool ID is specified, the maximum number of connections is set for the pool with the given ID.

`validate`

Specifies whether to validate a connection when it's returned from the pool.

Syntax

```
validate[pool_id: int] = False | True
```

Remarks

If the value of `validate` is `False`, the pool will validate a connection only when it's added to the pool. If the value is `True`, the pool will validate a connection when it's added or drawn from the pool. The default value is `False`.

If no pool ID (`pool_id`) is specified, the validation rule is set for the default connection pool. If the pool ID is specified, the rule is set for the pool with the given ID.

11 Support

Support

This page describes the support options and programs available for users of Python Connector for PostgreSQL.

Support options

The following support options are available for users of Python Connector for PostgreSQL:

- Annual maintenance and support service through the Python Connector for PostgreSQL Subscription program
- Community assistance and technical support through the community forum.
- Advanced technical support from the product developers through the Python Connector for PostgreSQL Priority Support program.

Subscriptions

The Python Connector for PostgreSQL Subscription program is an annual maintenance and support service that provides the following benefits:

- Support through the Priority Support program
- Access to new versions of the product
- Access to nightly builds with hotfixes (on demand)
- Notifications about new product versions

Priority Support

Python Connector for PostgreSQL Priority Support is an advanced product support service from the product developers. Devart staff will provide a response to the customer via email within two business days from the date of receipt. Priority Support is available for users with an active subscription.

If you need assistance with our product, send us an email at support@devart.com with the following details:

- The license number of your product

- The version and edition of your product
- The version of your PostgreSQL server
- A detailed description of the issue
- (Optional) Scripts for creating and populating the database objects

If you have any questions regarding licensing or subscriptions, send us an email at sales@devart.com

12 Licensing

Licensing

Python Connector for PostgreSQL License Agreement

PLEASE READ THIS LICENSE AGREEMENT CAREFULLY. BY INSTALLING OR USING THIS SOFTWARE, YOU INDICATE ACCEPTANCE OF AND AGREE TO BECOME BOUND BY THE TERMS AND CONDITIONS OF THIS LICENSE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENSE, DO NOT INSTALL OR USE THIS SOFTWARE AND PROMPTLY RETURN IT TO DEVART.

INTRODUCTION

This Devart end-user license agreement ("Agreement") is a legal agreement between you (either an individual person or a single legal entity) and Devart, for the use of the Python Connector software application, demos, intermediate files, printed materials, and online or electronic documentation contained in this installation file. For the purpose of this Agreement, the software program(s) and supporting documentation will be referred to as the "Software".

LICENSE

1. GRANT OF LICENSE

The enclosed Software is licensed, not sold. You have the following rights and privileges, subject to all limitations, restrictions, and policies specified in this Agreement.

1.1. If you are a legally licensed user, depending on the License Type specified in the registration letter you have received from Devart upon purchase of the Software:

- The "Single License" allows you to install and use the Software on one or more computers, provided it is used by 1 (one) user for the sole purposes of developing, testing, and deploying scripts or applications in a single company at one physical address in accordance with this Agreement.

- The "Team License" allows you to install and use the Software on one or more computers, provided it is used by up to 4 (four) users for the sole purposes of developing, testing, and deploying scripts or applications in a single company at one physical address in accordance with this Agreement.

- The "Site License" allows you to install and use the Software on one or more computers, provided it is used by unlimited number of users for the sole purposes of developing, testing, and deploying scripts or applications in a single company at one physical address in accordance with this Agreement.

- The "OEM License" allows you to install and use the Software as part of a licensee's script or application that can be deployed to web servers, application servers, batch servers, desktops, and other end-user devices. This definition includes the ability to install and use the script or application containing the Software an unlimited number of times, without any additional fees in favor of the licensor.

1.2. If you are a legally licensed user, depending on the License Type specified in the registration letter you have received from Devart upon purchase of the Software:

- The "Subscription-based License" allows you to install and use the Software on a single computer only during the subscription term specified at purchase. An Internet connection is required to activate the license and check the license status when the Software is used. Once the subscription term is over, you will be able to either stop using the Software or renew the license for a new subscription term.

- The "Perpetual License" allows you to install and use the specific Software product version on a single computer without an active subscription. A subscription provides access to new product releases, regular upgrades, and support for new server versions provided during the subscription term.

1.3. If you are a legally licensed user of the Software, you are also entitled to:

- Make one copy of the Software for archival purposes only, or copy the Software onto the hard disk of your computer and retain the original for archival purposes

- Develop and test Applications with the Software, subject to the Limitations below.

1.4. If you have the "OEM License ", you are also entitled to:

- Make any number of copies of the Software to deploy it to your end-user

- Deploy the Software to your end-user as a Software installation package or integrate it into your Applications.

1.5. You are allowed to use evaluation versions of the Software as specified in the Evaluation section.

No other rights or privileges are granted in this Agreement.

2. LIMITATIONS

Only legally registered users are licensed to use the Software, subject to all of the conditions of this Agreement. Usage of the Software is subject to the following restrictions.

2.1. You may not reverse engineer, decompile, or disassemble the Software.

2.2. You may not build any other Python packages through inheritance for public distribution or commercial sale.

2.3. You may not reproduce or distribute any Software documentation without express written permission from Devart.

2.4. You may not distribute and sell any portion of the Software integrating it into your Applications.

2.5. You may not transfer, assign, or modify the Software in whole or in part. In particular, the Software license is non-transferable, and you may not transfer the Software installation package.

2.6. You may not remove or alter any Devart's copyright, trademark, or other proprietary rights notice contained in any portion of Devart files.

3. REDISTRIBUTION

The license grants you a non-exclusive right to reproduce any new software programs (Applications) created using the Software. You cannot distribute the Software integrated into your Applications unless you are an "OEM License" holder. Any Devart's files remain Devart's exclusive property.

4. TRANSFER

You may not transfer the Software to any individual or entity without express written permission from Devart. In particular, you may not share copies of the Software under "Single License" and "Team License" with other co-developers without obtaining proper license of these copies for each individual.

5. TERMINATION

Devart may immediately terminate this Agreement without notice or judicial resolution in the event of any failure to comply with any provision of this Agreement. Upon such termination you must destroy the Software, all accompanying written materials, and all copies.

6. EVALUATION

Devart may provide evaluation ("Trial") versions of the Software. You may transfer or distribute Trial versions of the Software as an original installation package only. If the Software you have obtained is marked as a "Trial" version, you may install and use the Software for a period of up to 30 calendar days from the date of installation (the ""Trial Period"), subject to the additional restriction that it is used solely for evaluation of the Software and not in conjunction with the development or deployment of any application in production. You may not use Applications developed using Trial versions of the Software for any commercial purposes. Upon expiration of the Trial Period, the Software must be uninstalled, all its copies and all accompanying written materials must be destroyed.

7. WARRANTY

The Software and documentation are provided "AS IS" without warranty of any kind. Devart makes no warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose or use.

8. SUBSCRIPTION AND SUPPORT

The Software is sold on a subscription basis. The Software subscription entitles you to download improvements and enhancement from Devart's web site as they become available, during the active subscription period. The initial subscription period is one year from the date of purchase of the license. The subscription is automatically activated upon purchase, and may be subsequently renewed by Devart, subject to receipt applicable fees. Licensed users of the Software with an active subscription may request technical assistance with using the Software over email from the Software development. Devart shall use its reasonable endeavors to answer queries raised, but does not guarantee that your queries or problems will be fixed or solved.

Devart reserves the right to cease offering and providing support for legacy Python and Database versions.

9. COPYRIGHT

The Software is confidential and proprietary copyrighted work of Devart and is protected by international copyright laws and treaty provisions. You may not remove the copyright notice from any copy of the Software or any copy of the written materials, accompanying the Software.

This Agreement contains the total agreement between the two parties and supersedes any other agreements, written, oral, expressed, or implied.

13 Uninstall the connector

Uninstall the connector

To uninstall the connector, run the following command.

```
pip uninstall devart-postgresql-connector
```