合肥大學
HEFEI UNIVERSITY

Datenbanken

16. Fabrik Datenbank: von Python auf PostgreSQL zugreifen

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-08-29

Databases

Dies ist ein Kurs über Datenbanken an der Universität Hefei (合肥大学).

Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/databases (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielen finden Sie unter https://github.com/thomasWeise/databasesCode.

Outline

1. Einleitung
2. Daten mit Python aus PostgreSQL Datenbank auslesen
3. Einen Datensatz aus Python zur PostgreSQL Datenbank schicken
4. Exkursion: Systemsicherheit / System Security
5. Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken
6. Überprüfen des Ergebnisses
7. Zusammenfassung

Einleitung

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben.

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.
- Auf den ersten Blick ist das auch ordentlich, denn da gehören sie ja auch hin.

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.
- Auf den ersten Blick ist das auch ordentlich, denn da gehören sie ja auch hin.
- Auf den zweiten Blick erkennen wir ein Problem.

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.
- Auf den ersten Blick ist das auch ordentlich, denn da gehören sie ja auch hin.
- Auf den zweiten Blick erkennen wir ein Problem: Niemand außer uns (den DBAs und Entwicklern) can mit dieser Situation arbeiten.

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.
- Auf den ersten Blick ist das auch ordentlich, denn da gehören sie ja auch hin.
- Auf den zweiten Blick erkennen wir ein Problem: Niemand außer uns (den DBAs und Entwicklern) can mit dieser Situation arbeiten.
- Klar, wir können ein Benutzerkonto `boss` für unsere Chefin erstellen, mit dem sie sich einloggen und mit den Daten arbeiten kann.

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.
- Auf den ersten Blick ist das auch ordentlich, denn da gehören sie ja auch hin.
- Auf den zweiten Blick erkennen wir ein Problem: Niemand außer uns (den DBAs und Entwicklern) can mit dieser Situation arbeiten.
- Klar, wir können ein Benutzerkonto `boss` für unsere Chefin erstellen, mit dem sie sich einloggen und mit den Daten arbeiten kann.
- Aber wird sie wirklich mit SQL herumhantieren?

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.
- Auf den ersten Blick ist das auch ordentlich, denn da gehören sie ja auch hin.
- Auf den zweiten Blick erkennen wir ein Problem: Niemand außer uns (den DBAs und Entwicklern) can mit dieser Situation arbeiten.
- Klar, wir können ein Benutzerkonto `boss` für unsere Chefin erstellen, mit dem sie sich einloggen und mit den Daten arbeiten kann.
- Aber wird sie wirklich mit SQL herumhantieren?
- Können Sie sich vorstellen, dass der Sekretär der Verkaufsabteilung Bestellungen mit `INSERT INTO demand (...` in das System einpflegt?

Wie arbeiten wir mit den Daten?

- Wenn wir uns anschauen, was wir bisher geschafft haben, dann ist das schon ziemlich cool.
- Es gibt aber ein großes Problem, dass wir uns noch gar nicht angeschaut haben: Die Daten liegen vollständig in der Datenbank.
- Auf den ersten Blick ist das auch ordentlich, denn da gehören sie ja auch hin.
- Auf den zweiten Blick erkennen wir ein Problem: Niemand außer uns (den DBAs und Entwicklern) can mit dieser Situation arbeiten.
- Klar, wir können ein Benutzerkonto `boss` für unsere Chefin erstellen, mit dem sie sich einloggen und mit den Daten arbeiten kann.
- Aber wird sie wirklich mit SQL herumhantieren?
- Können Sie sich vorstellen, dass der Sekretär der Verkaufsabteilung Bestellungen mit `INSERT INTO demand (...` in das System einpflegt?
- Wohl eher nicht.

Zugriff auf die Daten

- Die Daten sind in der Datenbank, wo sie hingehören.

Zugriff auf die Daten

- Die Daten sind in der Datenbank, wo sie hingehören.
- Das DBMS beschützt sie und erzwingt, dass die von uns definierten Einschränkungen und Rechte eingehalten werden.

Zugriff auf die Daten

- Die Daten sind in der Datenbank, wo sie hingehören.
- Das DBMS beschützt sie und erzwingt, dass die von uns definierten Einschränkungen und Rechte eingehalten werden.
- Aber nur coole Menschen wie wir können damit arbeiten.

Zugriff auf die Daten

- Die Daten sind in der Datenbank, wo sie hingehören.
- Das DBMS beschützt sie und erzwingt, dass die von uns definierten Einschränkungen und Rechte eingehalten werden.
- Aber nur coole Menschen wie wir können damit arbeiten.
- Weniger kultivierte Leute schauen auf psql wie eine Kuh ins Uhrwerk.

Zugriff auf die Daten

- Die Daten sind in der Datenbank, wo sie hingehören.
- Das DBMS beschützt sie und erzwingt, dass die von uns definierten Einschränkungen und Rechte eingehalten werden.
- Aber nur coole Menschen wie wir können damit arbeiten.
- Weniger kultivierte Leute schauen auf psql wie eine Kuh ins Uhrwerk.
- Wir brauchen also alternative Methoden, damit alle Mitglieder unserer Organisation mit der Datenbank arbeiten können.

Zugriff auf die Daten

- Wir brauchen also alternative Methoden, damit alle Mitglieder unserer Organisation mit der Datenbank arbeiten können.

Zugriff auf die Daten

- Wir brauchen also alternative Methoden, damit alle Mitglieder unserer Organisation mit der Datenbank arbeiten können.
- Dafür gibt es mehrere Möglichkeiten.

Zugriff auf die Daten

- Wir brauchen also alternative Methoden, damit alle Mitglieder unserer Organisation mit der Datenbank arbeiten können.
- Dafür gibt es mehrere Möglichkeiten:
  1. Wir können unser eigenes Klienten-Programm schreiben. Dann können wir den Benutzern ein komfortables Interface zum Eingeben und Auslesen der Daten anbieten. Die Benutzer müssen dann niemals mit SQL arbeiten.

Zugriff auf die Daten

- Wir brauchen also alternative Methoden, damit alle Mitglieder unserer Organisation mit der Datenbank arbeiten können.
- Dafür gibt es mehrere Möglichkeiten:
  1. Wir können unser eigenes Klienten-Programm schreiben. Dann können wir den Benutzern ein komfortables Interface zum Eingeben und Auslesen der Daten anbieten. Die Benutzer müssen dann niemals mit SQL arbeiten.
  2. Wir können auch ein Front-End in Form einer Webapplikation schreiben, vielleicht basierend auf dem Flask Server. Dann können die Nutzer auf die Daten über den Web Browser zugrifen. Unser Programm erledigt dann den eigentlichen Datenbankzugriff im Hintergrund.

Zugriff auf die Daten

- Wir brauchen also alternative Methoden, damit alle Mitglieder unserer Organisation mit der Datenbank arbeiten können.
- Dafür gibt es mehrere Möglichkeiten:
  1. Wir können unser eigenes Klienten-Programm schreiben. Dann können wir den Benutzern ein komfortables Interface zum Eingeben und Auslesen der Daten anbieten. Die Benutzer müssen dann niemals mit SQL arbeiten.
  2. Wir können auch ein Front-End in Form einer Webapplikation schreiben, vielleicht basierend auf dem Flask Server. Dann können die Nutzer auf die Daten über den Web Browser zugrifen. Unser Programm erledigt dann den eigentlichen Datenbankzugriff im Hintergrund.
  3. Oder wir benutzen ein generelles Interface, wie es von LibreOffice Base angeboten wird, und verbinden uns so auf die Datenbank. Mit solch einem Werkzeug können wir bequem Formulare zum Eingeben von Daten und Berichte zum Auslesen und Visualisieren erstellen. Benutzer arbeiten dann nur mit dem Interface.

Zugriff auf die Daten

- Wir brauchen also alternative Methoden, damit alle Mitglieder unserer Organisation mit der Datenbank arbeiten können.
- Dafür gibt es mehrere Möglichkeiten:
  1. Wir können unser eigenes Klienten-Programm schreiben. Dann können wir den Benutzern ein komfortables Interface zum Eingeben und Auslesen der Daten anbieten. Die Benutzer müssen dann niemals mit SQL arbeiten.
  2. Wir können auch ein Front-End in Form einer Webapplikation schreiben, vielleicht basierend auf dem Flask Server. Dann können die Nutzer auf die Daten über den Web Browser zugrifen. Unser Programm erledigt dann den eigentlichen Datenbankzugriff im Hintergrund.
  3. Oder wir benutzen ein generelles Interface, wie es von LibreOffice Base angeboten wird, und verbinden uns so auf die Datenbank. Mit solch einem Werkzeug können wir bequem Formulare zum Eingeben von Daten und Berichte zum Auslesen und Visualisieren erstellen. Benutzer arbeiten dann nur mit dem Interface.
- Hier schauen wir uns die erste Möglichkeit an, in einer späteren Einheit auch noch die dritte.

Was wir jetzt machen werden

- Schreiben wir nun also ein Programm, dass auf unsere PostgreSQL Datenbank zugreift.

Was wir jetzt machen werden

- Schreiben wir nun also ein Programm, dass auf unsere PostgreSQL Datenbank zugreift.
- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.

Was wir jetzt machen werden

- Schreiben wir nun also ein Programm, dass auf unsere PostgreSQL Datenbank zugreift.
- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- In Python können wir beliebige Programme schreiben und Datentypen wir `int`, `float`, und `str` verwenden.

Was wir jetzt machen werden

- Schreiben wir nun also ein Programm, dass auf unsere PostgreSQL Datenbank zugreift.
- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- In Python können wir beliebige Programme schreiben und Datentypen wir `int`, `float`, und `str` verwenden.
- Wir können Kontrollflussstatements wie `if...then...else`, `for`- und `while` Schleifen verwenen.

Was wir jetzt machen werden

- Schreiben wir nun also ein Programm, dass auf unsere PostgreSQL Datenbank zugreift.
- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- In Python können wir beliebige Programme schreiben und Datentypen wir `int`, `float`, und `str` verwenden.
- Wir können Kontrollflussstatements wie `if...then...else`, `for`- und `while` Schleifen verwenen.
- Wir können Funktionen mit Hilfe von `def` definieren.

Was wir jetzt machen werden

- Schreiben wir nun also ein Programm, dass auf unsere PostgreSQL Datenbank zugreift.
- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- In Python können wir beliebige Programme schreiben und Datentypen wir `int`, `float`, und `str` verwenden.
- Wir können Kontrollflussstatements wie `if...then...else`, `for`- und `while` Schleifen verwenen.
- Wir können Funktionen mit Hilfe von `def` definieren.
- Python unterstützt auch objekt-orientiertes Programmieren (OOP).

Was wir jetzt machen werden

- Schreiben wir nun also ein Programm, dass auf unsere PostgreSQL Datenbank zugreift.
- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- In Python können wir beliebige Programme schreiben und Datentypen wir `int`, `float`, und `str` verwenden.
- Wir können Kontrollflussstatements wie `if...then...else`, `for`- und `while` Schleifen verwenen.
- Wir können Funktionen mit Hilfe von `def` definieren.
- Python unterstützt auch objekt-orientiertes Programmieren (OOP).
- Für Python gibt es unglaublich viele Pakete, die verschiedene zusätzliche Funktionalität anbieten, z. B. NumPy, Pandas, Scikit-learn, SciPy, TensorFlow, oder PyTorch.

Was wir jetzt machen werden

- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- Für Python gibt es unglaublich viele Pakete, die verschiedene zusätzliche Funktionalität anbieten, z. B. NumPy, Pandas, Scikit-learn, SciPy, TensorFlow, oder PyTorch.
- Diese Pakete liegen im PyPI repository und werden via pip installiert.

Was wir jetzt machen werden

- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- Für Python gibt es unglaublich viele Pakete, die verschiedene zusätzliche Funktionalität anbieten, z. B. NumPy, Pandas, Scikit-learn, SciPy, TensorFlow, oder PyTorch.
- Diese Pakete liegen im PyPI repository und werden via pip installiert.
- Python hat keine native Verbindung zu PostgreSQL.

Was wir jetzt machen werden

- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- Für Python gibt es unglaublich viele Pakete, die verschiedene zusätzliche Funktionalität anbieten, z. B. NumPy, Pandas, Scikit-learn, SciPy, TensorFlow, oder PyTorch.
- Diese Pakete liegen im PyPI repository und werden via pip installiert.
- Python hat keine native Verbindung zu PostgreSQL.
- Zum Glück gibt es das Paket psycopg, welches eine nach PEP 249[54] standardisierte API zum Zugriff auf PostgreSQL anbietet.

Was wir jetzt machen werden

- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- Für Python gibt es unglaublich viele Pakete, die verschiedene zusätzliche Funktionalität anbieten, z. B. NumPy, Pandas, Scikit-learn, SciPy, TensorFlow, oder PyTorch.
- Diese Pakete liegen im PyPI repository und werden via pip installiert.
- Python hat keine native Verbindung zu PostgreSQL.
- Zum Glück gibt es das Paket psycopg, welches eine nach PEP 249[54] standardisierte API zum Zugriff auf PostgreSQL anbietet.
- Mit psycopg können wir SQL-Anfragen in einem Python-Programm erstellen, an PostgreSQL schicken, und die Antworten auslesen.

Was wir jetzt machen werden

- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- Für Python gibt es unglaublich viele Pakete, die verschiedene zusätzliche Funktionalität anbieten, z. B. NumPy, Pandas, Scikit-learn, SciPy, TensorFlow, oder PyTorch.
- Diese Pakete liegen im PyPI repository und werden via pip installiert.
- Python hat keine native Verbindung zu PostgreSQL.
- Zum Glück gibt es das Paket psycopg, welches eine nach PEP 249[54] standardisierte API zum Zugriff auf PostgreSQL anbietet.
- Mit psycopg können wir SQL-Anfragen in einem Python-Programm erstellen, an PostgreSQL schicken, und die Antworten auslesen.
- Das ist sehr gut, weil wir ja schon einiges an SQL kennen.

Was wir jetzt machen werden

- Wir verwenden die Programmiersprache Python, die Sie in der Schwestervorlesung *Programming with Python*[102] lernen.
- Für Python gibt es unglaublich viele Pakete, die verschiedene zusätzliche Funktionalität anbieten, z. B. NumPy, Pandas, Scikit-learn, SciPy, TensorFlow, oder PyTorch.
- Diese Pakete liegen im PyPI repository und werden via pip installiert.
- Python hat keine native Verbindung zu PostgreSQL.
- Zum Glück gibt es das Paket psycopg, welches eine nach PEP 249[54] standardisierte API zum Zugriff auf PostgreSQL anbietet.
- Mit psycopg können wir SQL-Anfragen in einem Python-Programm erstellen, an PostgreSQL schicken, und die Antworten auslesen.
- Das ist sehr gut, weil wir ja schon einiges an SQL kennen.
- Somit müssen wir uns nur mit Python auseinandersetzen.

Python und PsycoPG

- Wir wollen also nun von Python aus auf unsere Datenbank zugreifen.

Python und PsycoPG

- Wir wollen also nun von Python aus auf unsere Datenbank zugreifen.
- Dafür verwenden wir das Paket psycopg[97].

Python und PsycoPG

- Wir wollen also nun von Python aus auf unsere Datenbank zugreifen.
- Dafür verwenden wir das Paket psycopg[97].
- Dieses haben wir in der letzten Einheit installiert.

Python und PsycoPG

- Wir wollen also nun von Python aus auf unsere Datenbank zugreifen.
- Dafür verwenden wir das Paket psycopg[97].
- Dieses haben wir in der letzten Einheit installiert.
- Es kann also losgehen.

Daten mit Python aus PostgreSQL Datenbank auslesen

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wir wollen mit einem Python-Programm Daten aus unserer Datenbank auslesen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Unser Programm muß dazu zwei Funktionen vom Paket psycopg importieren, nämlich `connect` ind `dict_row`.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Unser Programm muß dazu zwei Funktionen vom Paket psycopg importieren, nämlich `connect` ind `dict_row`.
- Wir brauchen diese, um uns mit dem PostgreSQL DBMS zu verbinden und um zu definieren, wie wir Anfrageergebnsse empfangen wollen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wir wollen die Daten mit einer `SELECT`-Anweisung auslesen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wir wollen die Daten mit einer `SELECT`-Anweisung auslesen.
- Diese schicken wir NICHT mit dem psql Klient an das DBMS, sondern via Python.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Dafür müssen wir zuerst eine Verbindung zum DBMS herstellen.

```python
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Dafür müssen wir zuerst eine Verbindung zum DBMS herstellen.
- Diese öffnen wir mit einem `with`-Statement, dass Sie hoffentlich aus einer Python-Vorlesung kennen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Dafür müssen wir zuerst eine Verbindung zum DBMS herstellen.
- Diese öffnen wir mit einem `with`-Statement, dass Sie hoffentlich aus einer Python-Vorlesung kennen.
- Datenbank-Sessions und Cursors sind als *context managers*[104] implementiert, was hier nur bedeutet, dass sie am Ende des `with`-Blocks automatisch geschlossen werden.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Am Anfang des Blocks wird mit der `connect` Funktion eine Verbindung `conn` zum PostgreSQL DBMSgeöffnet[95].

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Am Anfang des Blocks wird mit der `connect` Funktion eine Verbindung `conn` zum PostgreSQL DBMSgeöffnet[95].
- Wir geben dafür die selbe Verbindungs-URI an, die wir auch mit psql verwenden würden.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Am Anfang des Blocks wird mit der `connect` Funktion eine Verbindung `conn` zum PostgreSQL DBMSgeöffnet[95].
- Wir geben dafür die selbe Verbindungs-URI an, die wir auch mit psql verwenden würden.
- Damit wird der PostgreSQL Server gefunden und der Nutzer mit dem Passwort eingelogg.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Dann öffnen wir mit der Methode `cursor` der Verbindung einen so genannten Cursor[54,96].

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Dann öffnen wir mit der Methode `cursor` der Verbindung einen so genannten Cursor[54,96].
- Cursors sind Objekte mit denen wir die Kommandos an den Server schicken und die Antwort des Servers empfangen können.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Der Cursor liefert uns die Ergebnisse von SQL-Anfragen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Der Cursor liefert uns die Ergebnisse von SQL-Anfragen.
- Beim Erstellen des Cursos kann man optional einen Parameter `row_factory` spezifizieren.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Der Cursor liefert uns die Ergebnisse von SQL-Anfragen.
- Beim Erstellen des Cursos kann man optional einen Parameter `row_factory` spezifizieren.
- Hier nutzen wir dafür die `dict_row`-Function, die dafür sort, dass wir Anfrageergebnisse als `dict` zurückbekommen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Das Cursor-Objekt `cur` hat die Methode `execute`[54].

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Das Cursor-Objekt `cur` hat die Methode `execute`[54].
- Der erste Parameter dieser Methode ist ein SQL-Statement, dass vom DBMS ausgeführt werden soll.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Das Cursor-Objekt `cur` hat die Methode `execute`[54].
- Der erste Parameter dieser Methode ist ein SQL-Statement, dass vom DBMS ausgeführt werden soll.
- Der optionale zweite Parameter erlaubt uns, Anfrageparameter zu spezifizieren. Wir brauchen ihn diesmal nicht.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Was wir wollen ist alle Bestellungen des Kunden Mr. Bebbo (ID 2) auslesen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Was wir wollen ist alle Bestellungen des Kunden Mr. Bebbo (ID 2) auslesen.
- Wir nutzen also `cur` um das Kommando `"SELECT * FROM demand WHERE customer=2"` mit der `execute`-Methode abzusenden.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Nachdem das Kommando abgeschickt ist, können wir den Cursor als `Iterator`[106] in einer `for`-Schleife verwenden.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Nachdem das Kommando abgeschickt ist, können wir den Cursor als `Iterator`[106] in einer `for`-Schleife verwenden.
- Wir können die Anfrageergebnisse Zeile für Zeile ausgeben.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Jede Zeile wird uns dabei als `dict` geliefert.

```python
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Jede Zeile wird uns dabei als `dict` geliefert.
- Dieses `dict` hat die Spaltennamen als Schlüssel und die Attributwerte als, nunja, Werte.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Nachdem alle diese `dicts` zum stdout geschrieben wurden ended die `for`-Schleife.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Nachdem alle diese `dicts` zum stdout geschrieben wurden ended die `for`-Schleife.
- Der `with`-Block ist vorbei und der Cursor und die Verbindung wird geschlossen.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wir erinnern uns, dass Mr. Bebbo vier Bestellungen abgegeben hatte.

```
Now performing a SELECT request for customer Bebbo (id 2).
{'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
    ↪ date(2024, 12, 9)}
{'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
    ↪ date(2024, 12, 30)}
{'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
    ↪ date(2025, 1, 12)}
{'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
    ↪ date(2025, 2, 5)}
All done.
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wir erinnern uns, dass Mr. Bebbo vier Bestellungen abgegeben hatte.
- Und diese werden ausgegeben.

```
1 Now performing a SELECT request for customer Bebbo (id 2).
2 {'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
    ↪ date(2024, 12, 9)}
3 {'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
    ↪ date(2024, 12, 30)}
4 {'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
    ↪ date(2025, 1, 12)}
5 {'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
    ↪ date(2025, 2, 5)}
6 All done.
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wie uns der Output zeigt, werden die Werte auch in vernünftigen Datentypen geliefert.

```
1 Now performing a SELECT request for customer Bebbo (id 2).
2 {'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
    ↪ date(2024, 12, 9)}
3 {'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
    ↪ date(2024, 12, 30)}
4 {'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
    ↪ date(2025, 1, 12)}
5 {'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
    ↪ date(2025, 2, 5)}
6 All done.
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wie uns der Output zeigt, werden die Werte auch in vernünftigen Datentypen geliefert.
- Ganzzahlige Werte wie IDs kommen als `ints` an.

```
Now performing a SELECT request for customer Bebbo (id 2).
{'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
    ↪ date(2024, 12, 9)}
{'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
    ↪ date(2024, 12, 30)}
{'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
    ↪ date(2025, 1, 12)}
{'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
    ↪ date(2025, 2, 5)}
All done.
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Wie uns der Output zeigt, werden die Werte auch in vernünftigen Datentypen geliefert.
- Ganzzahlige Werte wie IDs kommen als `ints` an.
- Die Datums kommen als `datetime.date`.

```
Now performing a SELECT request for customer Bebbo (id 2).
{'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
    ↪ date(2024, 12, 9)}
{'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
    ↪ date(2024, 12, 30)}
{'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
    ↪ date(2025, 1, 12)}
{'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
    ↪ date(2025, 2, 5)}
All done.
```

Daten mit Python aus PostgreSQL Datenbank auslesen

- Sehr schön.

```
Now performing a SELECT request for customer Bebbo (id 2).
{'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
    ↪ date(2024, 12, 9)}
{'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
    ↪ date(2024, 12, 30)}
{'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
    ↪ date(2025, 1, 12)}
{'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
    ↪ date(2025, 2, 5)}
All done.
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir wollen mit einem Python-Programm einen Datensatz in unsere Datenbank schreiben.

```
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Am Anfang des Programms importieren wir den Datentyp `LiteralString`.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Am Anfang des Programms importieren wir den Datentyp `LiteralString`.
- Dieser Datentyp is speziell für String-Konstanten.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Am Anfang des Programms importieren wir den Datentyp `LiteralString`.
- Dieser Datentyp is speziell für String-Konstanten.
- Wir diskutieren das später.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir wollen Datensätze in die Datenbank einfügen.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir wollen Datensätze in die Datenbank einfügen.
- Das geht mit dem `INSERT INTO`-Statement.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir wollen Datensätze in die Datenbank einfügen.
- Das geht mit dem `INSERT INTO`-Statement.
- Weil dieses Statement lang ist, schreiben wir es erstmal als Variable hin.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir geben der Variable den Type Hint `LiteralString`, womit wir ausdrücken, dass das ein String ist, den wir direkt so hingeschrieben haben und nicht das Ergebnis von irgendwelchen String operationen.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir geben der Variable den Type Hint `LiteralString`, womit wir ausdrücken, dass das ein String ist, den wir direkt so hingeschrieben haben und nicht das Ergebnis von irgendwelchen String operationen.
- Warum erklären wir später. Machen wir erstmal weiter.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir beginnen mit dem selben `with`-Block wie vorhin.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir beginnen mit dem selben `with`-Block wie vorhin.
- Erst wird eine Verbindung `conn` zum PostgreSQL DBMS geöffnet[95].

```
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir beginnen mit dem selben `with`-Block wie vorhin.
- Erst wird eine Verbindung `conn` zum PostgreSQL DBMS geöffnet[95].
- Dann wird ein Cursor `cur` in der Verbindung erstellt[96].

```
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir benutzen nun wieder die Methode `execute` des Cursors.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir benutzen nun wieder die Methode `execute` des Cursors.
- Der erste Parameter ist wieder das SQL statement.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Wir benutzen nun wieder die Methode `execute` des Cursors.
- Der erste Parameter ist wieder das SQL statement.
- Diesmal brauche wir auch den zweiten Parameter: Eine Sequenz mit den Parametern für das Statement.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Unser SQL statement ist
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES (%s,%s,%s,%s)"`.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Unser SQL statement ist `"INSERT INTO demand (customer, product, amount, ordered)VALUES (%s,%s,%s,%s)"`.
- Der erste Teil ist klar: Wir wollen einen neuen Datensatz in die Tabelle `demand` via `INSERT INTO` einfügen.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Dafür geben wir den Name der Tabelle (`demand`) sowie die Namen der Spalten an, deren Werte wir schreiben wollen, nämlich i.e., `customer`, `product`, `demand`, and sqlilordered.

```
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Dafür geben wir den Name der Tabelle (`demand`) sowie die Namen der Spalten an, deren Werte wir schreiben wollen, nämlich i.e., `customer`, `product`, `demand`, and sqlilordered.
- In einem richtigen SQL-Kommando würden wir nun die zu schreibenden Werte in Klammern nach dem Schlüsselwort `VALUES` angeben.

```
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Stattdessen steht hier `"VALUES (%s,%s,%s,%s)"`.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Stattdessen steht hier `"VALUES (%s,%s,%s,%s)"`.
- Diese `%s` werden, eins nach dem anderen, mit den Parameterwerten im zweiten Argument der Methode ersetzt.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Diese werden sich nach SQL konvertiert, gleichgültig welchem Datentyp sie angehören.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Diese werden sich nach SQL konvertiert, gleichgültig welchem Datentyp sie angehören.
- `cur.execute(statement, (3, 4, 5, "2025-03-05"))` fügt eine neuen Datensatz in die Tabelle `demand` ein, bei dem `3` der Wert für `customer`, `4` der Wert für `product`, `5` der Wert für `amount`, und `"2025-03-05"` der Wert für `ordered` sind.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Daher ist die customer ID des Datensatzes 3, die product ID ist 4, die Produktmenge ist 5, und das Bestelldatum ist der 5. März 2025.

```python
"""Connect to our factory database and insert one record."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing a single INSERT statement.")
    cur.execute(statement,  # Insert one new demand record.
                (3, 4, 5, "2025-03-05" ))

print("All done.")
```

Einen Datensatz aus Python zur PostgreSQL Datenbank schicken

- Und das Program läuft auch problemlos durch.

```
Executing a single INSERT statement.
All done.
```

Exkursion: Systemsicherheit / System Security

- Datenbanken sind einige der wichtigsten Komponenten des IT-Ökosystems von Organisationen.

- Datenbanken sind einige der wichtigsten Komponenten des IT-Ökosystems von Organisationen.
- Wir müssen sie vor Angriffen sowohl von außen als auch von innen schützen.

Systemsicherheit / System Security

- Datenbanken sind einige der wichtigsten Komponenten des IT-Ökosystems von Organisationen.
- Wir müssen sie vor Angriffen sowohl von außen als auch von innen schützen.
- Wir müssen den Diebstahl, die Manipulation, und die Zerstörung von unseren Daten verhinden.

Systemsicherheit / System Security

- Datenbanken sind einige der wichtigsten Komponenten des IT-Ökosystems von Organisationen.
- Wir müssen sie vor Angriffen sowohl von außen als auch von innen schützen.
- Wir müssen den Diebstahl, die Manipulation, und die Zerstörung von unseren Daten verhinden.
- Wenn Programme auf unsere Datenbank zugreifen, dann müssen wir verhindern, dass sie zum Angriff auf unser System misbraucht werden.

Systemsicherheit / System Security

- Datenbanken sind einige der wichtigsten Komponenten des IT-Ökosystems von Organisationen.
- Wir müssen sie vor Angriffen sowohl von außen als auch von innen schützen.
- Wir müssen den Diebstahl, die Manipulation, und die Zerstörung von unseren Daten verhinden.
- Wenn Programme auf unsere Datenbank zugreifen, dann müssen wir verhindern, dass sie zum Angriff auf unser System misbraucht werden.
- Programme bekommen ihren Input oft entweder von Benutzereingaben, aus Dateien, aus dem Internet, oder von anderen Programmen.

Systemsicherheit / System Security

- Datenbanken sind einige der wichtigsten Komponenten des IT-Ökosystems von Organisationen.
- Wir müssen sie vor Angriffen sowohl von außen als auch von innen schützen.
- Wir müssen den Diebstahl, die Manipulation, und die Zerstörung von unseren Daten verhinden.
- Wenn Programme auf unsere Datenbank zugreifen, dann müssen wir verhindern, dass sie zum Angriff auf unser System misbraucht werden.
- Programme bekommen ihren Input oft entweder von Benutzereingaben, aus Dateien, aus dem Internet, oder von anderen Programmen.
- Wenn dieser Input ungeprüft an die Datenbank gesendet wird, dann kann das katastrophale Konsequenzen haben.

- Datenbanken sind einige der wichtigsten Komponenten des IT-Ökosystems von Organisationen.
- Wir müssen sie vor Angriffen sowohl von außen als auch von innen schützen.
- Wir müssen den Diebstahl, die Manipulation, und die Zerstörung von unseren Daten verhinden.
- Wenn Programme auf unsere Datenbank zugreifen, dann müssen wir verhindern, dass sie zum Angriff auf unser System misbraucht werden.
- Programme bekommen ihren Input oft entweder von Benutzereingaben, aus Dateien, aus dem Internet, oder von anderen Programmen.
- Wenn dieser Input ungeprüft an die Datenbank gesendet wird, dann kann das katastrophale Konsequenzen haben.
- Die Manipulation der Eingabedaten von Programmen, die auf Datenbanken zugreifen, ist ein sehr alter und sehr häufiger Angriffsvektor[77].

Programm sah komisch aus

- Und deshalb sah unser Programm komisch aus.

Programm sah komisch aus

- Und deshalb sah unser Programm komisch aus.
- Zum Beispiel haben wir in Python ja f-Strings, die genau dazu da sind, Zeichenketten aus Parameterwerten zu konstruieren.

Programm sah komisch aus

- Und deshalb sah unser Programm komisch aus.
- Zum Beispiel haben wir in Python ja f-Strings, die genau dazu da sind, Zeichenketten aus Parameterwerten zu konstruieren.
- Man könnte also fragen: Warum haben wir unsere `INSERT INTO`-Anfrage so komisch zusammengebaut, anstatt die Standardwerkzeuge unserer Programmiersprache zu verwenden?

Programm sah komisch aus

- Und deshalb sah unser Programm komisch aus.
- Zum Beispiel haben wir in Python ja f-Strings, die genau dazu da sind, Zeichenketten aus Parameterwerten zu konstruieren.
- Man könnte also fragen: Warum haben wir unsere `INSERT INTO`-Anfrage so komisch zusammengebaut, anstatt die Standardwerkzeuge unserer Programmiersprache zu verwenden?
- Dann haben wir auch noch den komischen Datentyp `LiteralString` verwendet.

Programm sah komisch aus

- Und deshalb sah unser Programm komisch aus.
- Zum Beispiel haben wir in Python ja f-Strings, die genau dazu da sind, Zeichenketten aus Parameterwerten zu konstruieren.
- Man könnte also fragen: Warum haben wir unsere `INSERT INTO`-Anfrage so komisch zusammengebaut, anstatt die Standardwerkzeuge unserer Programmiersprache zu verwenden?
- Dann haben wir auch noch den komischen Datentyp `LiteralString` verwendet.
- Wofür soll das gut sein?

Programm sah komisch aus

- Und deshalb sah unser Programm komisch aus.
- Zum Beispiel haben wir in Python ja f-Strings, die genau dazu da sind, Zeichenketten aus Parameterwerten zu konstruieren.
- Man könnte also fragen: Warum haben wir unsere `INSERT INTO`-Anfrage so komisch zusammengebaut, anstatt die Standardwerkzeuge unserer Programmiersprache zu verwenden?
- Dann haben wir auch noch den komischen Datentyp `LiteralString` verwendet.
- Wofür soll das gut sein?
- Die Antwort auf beide Fragen ist: Systemsicherheit (systems security).

LiteralString

- `LiteralString` ist ein Sonderfall des Datentyps `str`.

LiteralString

- `LiteralString` ist ein Sonderfall des Datentyps `str`.
- Er existiert für String-Literale, also Strings, die wir "genau so in den Quelltext hingeschrieben haben."

LiteralString

- `LiteralString` ist ein Sonderfall des Datentyps `str`.
- Er existiert für String-Literale, also Strings, die wir "genau so in den Quelltext hingeschrieben haben."
- Eine Variable kann diesen Datentyp haben. Dann ist sie immer noch ein String.

LiteralString

- `LiteralString` ist ein Sonderfall des Datentyps `str`.
- Er existiert für String-Literale, also Strings, die wir "genau so in den Quelltext hingeschrieben haben."
- Eine Variable kann diesen Datentyp haben. Dann ist sie immer noch ein String.
- Aber wir haben explizit gesagt: Dieser String ist NICHT das Ergebnis von irgendeiner String-Operation, nicht das Ergebnis von String-Konkatenation (`+`), nicht das Ergebnis von String Interpolation, und auch kein f-String.

LiteralString

- `LiteralString` ist ein Sonderfall des Datentyps `str`.
- Er existiert für String-Literale, also Strings, die wir "genau so in den Quelltext hingeschrieben haben."
- Eine Variable kann diesen Datentyp haben. Dann ist sie immer noch ein String.
- Aber wir haben explizit gesagt: Dieser String ist NICHT das Ergebnis von irgendeiner String-Operation, nicht das Ergebnis von String-Konkatenation (`+`), nicht das Ergebnis von String Interpolation, und auch kein f-String.
- Wir benutzen also nicht nur keine f-Strings in unserem Programm, wir sagen sogar noch explizit das wir das nicht tun.

LiteralString

- `LiteralString` ist ein Sonderfall des Datentyps `str`.
- Er existiert für String-Literale, also Strings, die wir "genau so in den Quelltext hingeschrieben haben."
- Eine Variable kann diesen Datentyp haben. Dann ist sie immer noch ein String.
- Aber wir haben explizit gesagt: Dieser String ist NICHT das Ergebnis von irgendeiner String-Operation, nicht das Ergebnis von String-Konkatenation (`+`), nicht das Ergebnis von String Interpolation, und auch kein f-String.
- Wir benutzen also nicht nur keine f-Strings in unserem Programm, wir sagen sogar noch explizit das wir das nicht tun.
- Warum machen wir das, wo f-Strings doch eigentlich so cool sind[102]?

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.
- Jetzt könnte jemand kommen, und den Wert für `s` so setzen:
  `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`.

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.
- Jetzt könnte jemand kommen, und den Wert für `s` so setzen:
  `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`.
- Das `);` in `s` würde die Einfügeanfrage beenden.

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.
- Jetzt könnte jemand kommen, und den Wert für `s` so setzen:
  `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`.
- Das `);` in `s` würde die Einfügeanfrage beenden.
- Der Rest von `s` wäre dann eine neue SQL-Anfrage.

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.
- Jetzt könnte jemand kommen, und den Wert für `s` so setzen:
  `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`.
- Das `);` in `s` würde die Einfügeanfrage beenden.
- Der Rest von `s` wäre dann eine neue SQL-Anfrage.
- Das `DROP TABLE...` würde die gesamte Tabelle `demand` löschen (das Kommado lernen wir später).

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.
- Jetzt könnte jemand kommen, und den Wert für `s` so setzen:
  `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`.
- Das `);` in `s` würde die Einfügeanfrage beenden.
- Der Rest von `s` wäre dann eine neue SQL-Anfrage.
- Das `DROP TABLE...` würde die gesamte Tabelle `demand` löschen (das Kommado lernen wir später).
- Das ist nur *ein* Beispiel.

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.
- Jetzt könnte jemand kommen, und den Wert für `s` so setzen:
  `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`.
- Das `);` in `s` würde die Einfügeanfrage beenden.
- Der Rest von `s` wäre dann eine neue SQL-Anfrage.
- Das `DROP TABLE...` würde die gesamte Tabelle `demand` löschen (das Kommado lernen wir später).
- Das ist nur *ein* Beispiel.
- Andere typische Beispiele sind das Umgehen von Passwortabfragen um Zugriff auf sensible Daten zu erhalten.

SQL Injection

- SQL Injection Attacks (SQLi)[21,49,69,77,101] sind Angriffe auf SQL Server die dadurch ermöglicht werden, dass SQL Anfragen mit Hilfe von String-Koncatenation (Python+) der String-Interpolation zusammengebaut werden.
- Stellen Sie sich vor, wir würden unsere Anfrage so zusammenbauen:
  `"INSERT INTO demand (customer, product, amount, ordered)VALUES ("+ s + ")"`.
- Jetzt könnte jemand kommen, und den Wert für `s` so setzen:
  `s = "1, 2, 3, '2023-02-02'); DROP TABLE demand;"`.
- Das `);` in `s` würde die Einfügeanfrage beenden.
- Der Rest von `s` wäre dann eine neue SQL-Anfrage.
- Das `DROP TABLE...` würde die gesamte Tabelle `demand` löschen (das Kommado lernen wir später).
- Das ist nur *ein* Beispiel.
- Andere typische Beispiele sind das Umgehen von Passwortabfragen um Zugriff auf sensible Daten zu erhalten.
- Ein Angreifer könnte also alle möglichen fiesen Dinge tun, wenn wir unsere Datenbank nicht vor dieser Art Angriff schützen.

Darum LiteralString

- `LiteralStrings` gibt es in Python also aus Sicherheitsgründen[84].

Darum LiteralString

- `LiteralStrings` gibt es in Python also aus Sicherheitsgründen[84].
- Es soll dabei helfen SQLi-Angriff zu verhindern.

Darum LiteralString

- `LiteralStrings` gibt es in Python also aus Sicherheitsgründen[84].
- Es soll dabei helfen SQLi-Angriff zu verhindern.
- Die Idee ist, das statische Typ-Checker in Zukunft festellen so können, ob ein String ein Literal ist oder das Ergebnis von String-Operationen und Fehler in Sicherheitskritischen Situation anzeigen können[84,99].

Darum LiteralString

- `LiteralStrings` gibt es in Python also aus Sicherheitsgründen[84].
- Es soll dabei helfen SQLi-Angriff zu verhindern.
- Die Idee ist, das statische Typ-Checker in Zukunft festellen so können, ob ein String ein Literal ist oder das Ergebnis von String-Operationen und Fehler in Sicherheitskritischen Situation anzeigen können[84,99].
- Es ist in psycopg also verboten, irgendeine Art von String-Operation zu verwenden, um ein SQL-Kommando zusammenzubauen.

Darum LiteralString

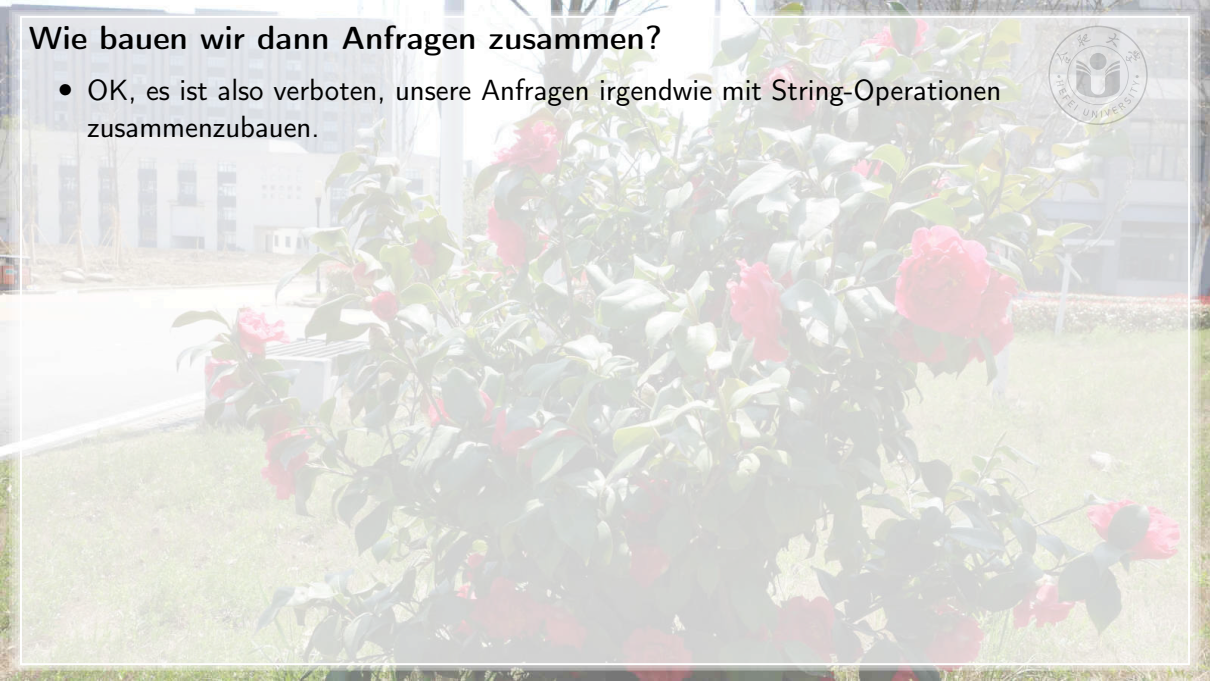
- `LiteralStrings` gibt es in Python also aus Sicherheitsgründen[84].
- Es soll dabei helfen SQLi-Angriff zu verhindern.
- Die Idee ist, das statische Typ-Checker in Zukunft festellen so können, ob ein String ein Literal ist oder das Ergebnis von String-Operationen und Fehler in Sicherheitskritischen Situation anzeigen können[84,99].
- Es ist in psycopg also verboten, irgendeine Art von String-Operation zu verwenden, um ein SQL-Kommando zusammenzubauen.
- SQL-Kommandos müssen vom Typ `LiteralString` sein.

Darum LiteralString

- `LiteralStrings` gibt es in Python also aus Sicherheitsgründen[84].
- Es soll dabei helfen SQLi-Angriff zu verhindern.
- Die Idee ist, das statische Typ-Checker in Zukunft festellen so können, ob ein String ein Literal ist oder das Ergebnis von String-Operationen und Fehler in Sicherheitskritischen Situation anzeigen können[84,99].
- Es ist in psycopg also verboten, irgendeine Art von String-Operation zu verwenden, um ein SQL-Kommando zusammenzubauen.
- SQL-Kommandos müssen vom Typ `LiteralString` sein.
- Somit wird SQLi-Angriffen ein Riegel vorgeschoben (wenn man sich an die Vorgaben hält).

Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.



Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.
- Es würde aber wenig Sinn ergeben, wenn wir nur SQL-Kommandos verwenden dürften, die hart-kodierte String-Literale sind.

Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.
- Es würde aber wenig Sinn ergeben, wenn wir nur SQL-Kommandos verwenden dürften, die hart-kodierte String-Literale sind.
- Wie bekommen wir Parameter in unsere Anfragen rein?

Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.
- Es würde aber wenig Sinn ergeben, wenn wir nur SQL-Kommandos verwenden dürften, die hart-kodierte String-Literale sind.
- Wie bekommen wir Parameter in unsere Anfragen rein?
- Dafür gibt es die Platzhalter `"%s"` in psycopg.

Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.
- Es würde aber wenig Sinn ergeben, wenn wir nur SQL-Kommandos verwenden dürften, die hart-kodierte String-Literale sind.
- Wie bekommen wir Parameter in unsere Anfragen rein?
- Dafür gibt es die Platzhalter `"%s"` in psycopg.
- Wir stellen die Werte für die Platzhalter zur Verfügung und das System kann dann z. B. String-Escaping anwenden, also alle "gefährlichen" Zeichen durch Escape-Sequenzen ersetzen und in harmlosen Text umwandeln.

Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.
- Es würde aber wenig Sinn ergeben, wenn wir nur SQL-Kommandos verwenden dürften, die hart-kodierte String-Literale sind.
- Wie bekommen wir Parameter in unsere Anfragen rein?
- Dafür gibt es die Platzhalter `"%s"` in psycopg.
- Wir stellen die Werte für die Platzhalter zur Verfügung und das System kann dann z. B. String-Escaping anwenden, also alle "gefährlichen" Zeichen durch Escape-Sequenzen ersetzen und in harmlosen Text umwandeln.
- Egal welche Werte für Query-Parameter bereitgestellt werden, es kann kein bösartiger String für eine Attacke konstruiert werden.

Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.
- Es würde aber wenig Sinn ergeben, wenn wir nur SQL-Kommandos verwenden dürften, die hart-kodierte String-Literale sind.
- Wie bekommen wir Parameter in unsere Anfragen rein?
- Dafür gibt es die Platzhalter `"%s"` in psycopg.
- Wir stellen die Werte für die Platzhalter zur Verfügung und das System kann dann z. B. String-Escaping anwenden, also alle "gefährlichen" Zeichen durch Escape-Sequenzen ersetzen und in harmlosen Text umwandeln.
- Egal welche Werte für Query-Parameter bereitgestellt werden, es kann kein bösartiger String für eine Attacke konstruiert werden.
- SQLi-Angriffe werden verhindert.

Wie bauen wir dann Anfragen zusammen?

- OK, es ist also verboten, unsere Anfragen irgendwie mit String-Operationen zusammenzubauen.
- Dafür gibt es die Platzhalter `"%s"` in psycopg.
- Wir stellen die Werte für die Platzhalter zur Verfügung und das System kann dann z. B. String-Escaping anwenden, also alle "gefährlichen" Zeichen durch Escape-Sequenzen ersetzen und in harmlosen Text umwandeln.
- Egal welche Werte für Query-Parameter bereitgestellt werden, es kann kein bösartiger String für eine Attacke konstruiert werden.
- SQLi-Angriffe werden verhindert.

Gute Praxis

Gleichgültig, welche Programmiersprache und welche Werkzeuge Sie zum Zugriff auf eine Datenbank verwenden, konstrieren Sie NIEMALS SQL-Kommandos mit String-Operationen. Verwenden Sie immer die entsprechenden Methoden wie Anfrageparameter, die Ihnen das System anbietet. Öffnen Sie niemals die Tür für SQLi-Angriffe.

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Wir wollen mit einem Python-Programm mehrere Datensätze in unsere Datenbank schreiben.

```python
"""Connect to our factory database and insert some records."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing three INSERT statements at once.")
    cur.executemany(statement, (  # Insert three new demand records.
        (3,  5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
        (1, 10, 5, "2025-04-05")))

print("All done.")
```

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Das geht fast ganz genauso als wenn wir nur einen Datensatz schreiben würden.

```python
"""Connect to our factory database and insert some records."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing three INSERT statements at once.")
    cur.executemany(statement, (  # Insert three new demand records.
        (3,  5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
        (1, 10, 5, "2025-04-05")))

print("All done.")
```

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Das geht fast ganz genauso als wenn wir nur einen Datensatz schreiben würden.
- Es gibt nur zwei kleine Unterschiede.

```python
"""Connect to our factory database and insert some records."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing three INSERT statements at once.")
    cur.executemany(statement, (  # Insert three new demand records.
        (3,  5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
        (1, 10, 5, "2025-04-05")))

print("All done.")
```

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Erstens verwenden wir die Methode `executemany` anstelle von `execute`.

```python
"""Connect to our factory database and insert some records."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing three INSERT statements at once.")
    cur.executemany(statement, (  # Insert three new demand records.
        (3,  5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
        (1, 10, 5, "2025-04-05")))

print("All done.")
```

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Erstens verwenden wir die Methode `executemany` anstelle von `execute`.
- Zweitens bekommt diese Methode eine Squence von Sequenzen von Parameterwerten, wobei das SQL-Kommando dann für jede einzelne Sequenz aufgerufen wird.

```python
"""Connect to our factory database and insert some records."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing three INSERT statements at once.")
    cur.executemany(statement, (  # Insert three new demand records.
        (3,  5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
        (1, 10, 5, "2025-04-05")))

print("All done.")
```

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Wir benutzen das selbe SQL-Kommando also, um drei Datensätze in unsere Tabelle einzufügen.

```python
"""Connect to our factory database and insert some records."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing three INSERT statements at once.")
    cur.executemany(statement, (  # Insert three new demand records.
        (3,  5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
        (1, 10, 5, "2025-04-05")))

print("All done.")
```

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Wir benutzen das selbe SQL-Kommando also, um drei Datensätze in unsere Tabelle einzufügen.
- Davor wird wie immer die Verbindung zur Datenbank am Anfang des `with`-Blocks aufgebaut und an dessen Ende wieder geschlossen.

```
"""Connect to our factory database and insert some records."""

from typing import LiteralString

from psycopg import connect

# The insert statement has to be a literal string.
statement: LiteralString = "INSERT INTO demand (customer, product,
    ↪ amount, ordered) VALUES (%s,%s,%s,%s)"

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor() as cur):
    print("Executing three INSERT statements at once.")
    cur.executemany(statement, (  # Insert three new demand records.
        (3,  5, 2, "2025-03-16"), (2, 7, 1, "2025-03-29"),
        (1, 10, 5, "2025-04-05")))

print("All done.")
```

Mehrere Datensätze aus Python zur PostgreSQL Datenbank schicken

- Es scheint ohne Fehler zu klappen.

```
Executing three INSERT statements at once.
All done.
```

Überprüfen des Ergebnisses

Überprüfen des Ergebnisses

- Überprüfen wir mal, ob die Daten wirklich zur Datenbank durchgekommen sind.

Überprüfen des Ergebnisses

- Führen wir nochmal unser Beispielprogramm von vorhin aus.

```
"""Connect to our factory database and select some records."""

from psycopg import connect
from psycopg.rows import dict_row

# Connect to the database and create a cursor to interact with the db:
with (connect("postgres://boss:superboss123@localhost/factory") as conn,
      conn.cursor(row_factory=dict_row) as cur):  # SELECT returns dicts

    print("Now performing a SELECT request for customer Bebbo (id 2).")
    cur.execute("SELECT * FROM demand WHERE customer=2")
    for record in cur:  # Iterate over the records in the cursor.
        print(record)  # Print them as-is.

print("All done.")
```

Überprüfen des Ergebnisses

- Führen wir nochmal unser Beispielprogramm von vorhin aus.
- Mr. Bebbo hat nun fünf Bestellung anstatt von vier, so wie es nach dem Hinzufügen der Daten seien soll.

```
Now performing a SELECT request for customer Bebbo (id 2).
{'id': 2, 'customer': 2, 'product': 3, 'amount': 2, 'ordered': datetime.
    ↪ date(2024, 12, 9)}
{'id': 4, 'customer': 2, 'product': 5, 'amount': 7, 'ordered': datetime.
    ↪ date(2024, 12, 30)}
{'id': 6, 'customer': 2, 'product': 6, 'amount': 4, 'ordered': datetime.
    ↪ date(2025, 1, 12)}
{'id': 8, 'customer': 2, 'product': 3, 'amount': 6, 'ordered': datetime.
    ↪ date(2025, 2, 5)}
{'id': 13, 'customer': 2, 'product': 7, 'amount': 1, 'ordered': datetime
    ↪ .date(2025, 3, 29)}
All done.
```

Überprüfen des Ergebnisses

- Führen wir doch nochmal unsere Sicht (View) `sale` von Einheit 12 aus.

```sql
/* Extract some information from our database using the view 'sale'. */

-- Get the total sales per customer.
SELECT customer_name, SUM(amount * price) AS customer_sale FROM sale
    GROUP BY customer_name ORDER BY customer_sale DESC;

-- Get the total sales per product.
SELECT product_name, SUM(amount) AS total_amount,
    SUM(amount * price) AS product_sale FROM sale
    GROUP BY product_name ORDER BY product_sale DESC;
```

Überprüfen des Ergebnisses

- Führen wir doch nochmal unsere Sicht (View) `sale` von Einheit 12 aus.
- Auch hier haben sich die Daten korrekt geändert.

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf select_from_view_sale_1.sql
    customer_name     | customer_sale
----------------------+---------------
 Bebba, 33333333333 |       3673.86
 Bebbo, 55555555555 |       3159.80
 Bibbo, 99999999999 |       3032.85
(3 rows)

 product_name   | total_amount | product_sale
----------------+--------------+--------------
 Shoe, Size 42 |           13 |      2118.87
 Shoe, Size 40 |           12 |      1907.88
 Large Purse    |           10 |      1500.00
 Shoe, Size 38 |            8 |      1239.92
 Shoe, Size 37 |            7 |      1070.93
 Shoe, Size 39 |            5 |       784.95
 Shoe, Size 41 |            4 |       643.96
 Medium Purse   |            5 |       600.00
(8 rows)

# psql 16.10 succeeded with exit code 0.
```

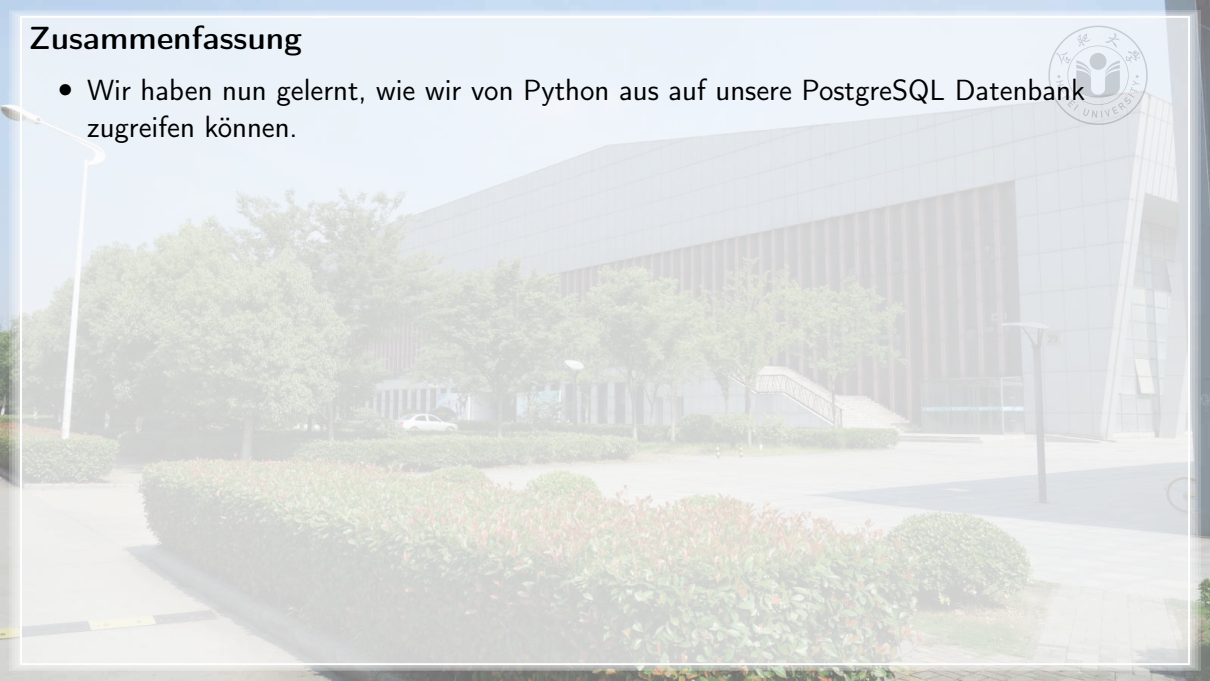
Überprüfen des Ergebnisses

- Überprüfen wir mal, ob die Daten wirklich zur Datenbank durchgekommen sind.
- Führen wir nochmal unser Beispielprogramm von vorhin aus.
- Mr. Bebbo hat nun fünf Bestellung anstatt von vier, so wie es nach dem Hinzufügen der Daten seien soll.
- Führen wir doch nochmal unsere Sicht (View) `sale` von Einheit 12 aus.
- Auch hier haben sich die Daten korrekt geändert.
- Es hat also alles gut funktioniert.

Zusammenfassung

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.



Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.
- Wir haben hier aber nur an der Oberfläche gekratzt.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.
- Wir haben hier aber nur an der Oberfläche gekratzt.
- Wir können nun theoretisch Datenströme in z. B. dem Extensible Markup Language (XML) oder dem comma-separated values (CSV)-Format einlesen und an die Datenbank schicken.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.
- Wir haben hier aber nur an der Oberfläche gekratzt.
- Wir können nun theoretisch Datenströme in z. B. dem Extensible Markup Language (XML) oder dem comma-separated values (CSV)-Format einlesen und an die Datenbank schicken.
- Oder wir können die vielen ML, AI, und Visualisierungs-Bibliotheken von Python, wie Scikit-learn, TensorFlow, SciPy, PyTorch, oder Matplotlib benutzen, um Daten zu verarbeiten.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.
- Wir haben hier aber nur an der Oberfläche gekratzt.
- Wir müssen aber immer vorsichtig sein: Es kann sehr schnell passieren, dass wir aus Versehen das Tor für z. B. SQLi-Angriffe öffnen.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.
- Wir haben hier aber nur an der Oberfläche gekratzt.
- Wir müssen aber immer vorsichtig sein: Es kann sehr schnell passieren, dass wir aus Versehen das Tor für z. B. SQLi-Angriffe öffnen.
- Deshalb dürfen wir niemals SQL-Anfragen aus Zeichenketten zusammenbauen.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.
- Wir haben hier aber nur an der Oberfläche gekratzt.
- Wir müssen aber immer vorsichtig sein: Es kann sehr schnell passieren, dass wir aus Versehen das Tor für z. B. SQLi-Angriffe öffnen.
- Deshalb dürfen wir niemals SQL-Anfragen aus Zeichenketten zusammenbauen.
- Stattdessen müssen diese hart im Quelltext kodiert sein und wir dürfen nur Anfrage-Parameter verwenden, um dynamisch Anfragen zu generieren.

Zusammenfassung

- Wir haben nun gelernt, wie wir von Python aus auf unsere PostgreSQL Datenbank zugreifen können.
- Wir können Daten in die Datenbank einfügen und Daten aus der Datenbank auslesen.
- Wir brauchen dazu nur grundlegende Python Kenntnisse, denn der Zugriff funktioniert wieder über SQL.
- Wir haben hier aber nur an der Oberfläche gekratzt.
- Wir müssen aber immer vorsichtig sein: Es kann sehr schnell passieren, dass wir aus Versehen das Tor für z. B. SQLi-Angriffe öffnen.
- Deshalb dürfen wir niemals SQL-Anfragen aus Zeichenketten zusammenbauen.
- Stattdessen müssen diese hart im Quelltext kodiert sein und wir dürfen nur Anfrage-Parameter verwenden, um dynamisch Anfragen zu generieren.
- Was wir überhaupt niemals machen sollten, sind Benutzernamen und Passwörter hart in unseren Programmquelltext zu schreiben. Das habe ich hier nur der Einfachheit halber gemacht...

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu und Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'2016)*. 2.–4. Nov. 2016, Savannah, GA, USA. Hrsg. von Kimberly Keeton und Timothy Roscoe. Berkeley, CA, USA: USENIX Association, 2016, S. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi (besucht am 2024-06-26) (siehe S. 190).

[2] Olatunde Adedeji. *Full-Stack Flask and React*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-844-8 (siehe S. 188).

[3] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 188, 190).

[4] David M. Beazley. "Data Processing with Pandas". *;login: Usenix Magazin* 37(6), Dez. 2012. Berkeley, CA, USA: USENIX Association. ISSN: 1044-6397. URL: https://www.usenix.org/publications/login/december-2012-volume-37-number-6/data-processing-pandas (besucht am 2024-06-25) (siehe S. 189).

[5] Ben Beitler. *Hands-On Microsoft Access 2019*. Birmingham, England, UK: Packt Publishing Ltd, März 2020. ISBN: 978-1-83898-747-3 (siehe S. 188).

[6] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt (besucht am 2025-02-05) (siehe S. 191).

[7] Tim Berners-Lee, Larry Masinter und Mark P. McCahill. *Uniform Resource Locators (URL)*. Request for Comments (RFC) 1738. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Dez. 1994. URL: https://www.ietf.org/rfc/rfc1738.txt (besucht am 2025-02-05) (siehe S. 191).

[8] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 187).

References II

[9] Bernard Obeng Boateng. *Data Modeling with Microsoft Excel*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-028-2 (siehe S. 188).

[10] Ethan Bommarito und Michael Bommarito. *An Empirical Analysis of the Python Package Index (PyPI)*. arXiv.org: Computing Research Repository (CoRR) abs/1907.11073. Ithaca, NY, USA: Cornell Universiy Library, 26. Juli 2019. doi:10.48550/arXiv.1907.11073. URL: https://arxiv.org/abs/1907.11073 (besucht am 2024-08-17). arXiv:1907.11073v2 [cs.SE] 26 Jul 2019 (siehe S. 189).

[11] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 188).

[12] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 187).

[13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen und Eve Maler, Hrsg. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. Wakefield, MA, USA: World Wide Web Consortium (W3C), 26. Nov. 2008–7. Feb. 2013. URL: http://www.w3.org/TR/2008/REC-xml-20081126 (besucht am 2024-12-15) (siehe S. 191).

[14] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 190).

[15] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: https://cacm.acm.org/research/50-years-of-queries (besucht am 2025-01-09) (siehe S. 190).

[16] Aakash Choudhury, Arjav Choudhury, Umashankar Subramanium und S. Balamurugan. "HealthSaver: A Neural Network based Hospital Recommendation System Framework on Flask Webapplication with Realtime Database and RFID based Attendance System". *Journal of Ambient Intelligence and Humanized Computing* 13(10):4953–4966, Okt. 2022. London, England, UK: Springer Nature Limited. ISSN: 1868-5137. doi:10.1007/S12652-021-03232-7 (siehe S. 188).

[17] Christmas, FL, USA: Simon Sez IT. *Microsoft Access 2021 – Beginner to Advanced*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2023. ISBN: 978-1-83546-911-8 (siehe S. 188).

References III

[18] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 190).

[19] Edgar Frank "Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (besucht am 2025-01-05) (siehe S. 189).

[20] Timothy W. Cole und Myung-Ja K. Han. *XML for Catalogers and Metadata Librarians (Third Millennium Cataloging)*. 1. Aufl. Dublin, OH, USA: Libraries Unlimited, 23. Mai 2013. ISBN: 978-1-59884-519-8 (siehe S. 191).

[21] Ignacio Samuel Crespo-Martínez, Adrián Campazas Vega, Ángel Manuel Guerrero-Higueras, Virginia Riego-Del Castillo, Claudia Álvarez-Aparicio und Camino Fernández Llamas. "SQL Injection Attack Detection in Network Flow Data". *Computers & Security* 127:103093, Apr. 2023. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0167-4048. doi:10.1016/J.COSE.2023.103093 (siehe S. 123–131).

[22] "`csv` – CSV File Reading and Writing". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/csv.html (besucht am 2024-11-14) (siehe S. 187).

[23] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 190).

[24] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: https://dataschool.com/how-to-teach-people-sql (besucht am 2025-02-27) (siehe S. 190).

[25] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 190).

[26] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide Web: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 191).

[27] Justin Dennison, Cherokee Boose und Peter van Rysdam. *Intro to NumPy*. Centennial, CO, USA: ACI Learning. Birmingham, England, UK: Packt Publishing Ltd, Juni 2024. ISBN: 978-1-83620-863-1 (siehe S. 189).

References IV

[28] Pooyan Doozandeh und Frank E. Ritter. "Some Tips for Academic Writing and Using Microsoft Word". *XRDS: Crossroads, The ACM Magazine for Students* 26(1):10–11, Herbst 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1528-4972. doi:10.1145/3351470 (siehe S. 189).

[29] Phillip J. Eby. *Python Web Server Gateway Interface v1.0.1*. Python Enhancement Proposal (PEP) 3333. Beaverton, OR, USA: Python Software Foundation (PSF), 26. Sep.–4. Okt. 2010. URL: https://peps.python.org/pep-3333 (besucht am 2025-03-04) (siehe S. 188).

[30] Steve Fanning, Vasudev Narayanan, "flywire", Olivier Hallot, Jean Hollis Weber, Jenna Sargent, Pulkit Krishna, Dan Lewis, Peter Schofield, Jochen Schiffers, Robert Großkopf, Jost Lange, Martin Fox, Hazel Russman, Steve Schwettman, Alain Romedenne, Andrew Pitonyak, Jean-Pierre Ledure, Drew Jensen und Randolph Gam. *Base Guide 7.3. Revision 1. Based on LibreOffice 7.3 Community*. Berlin, Germany: The Document Foundation, Aug. 2022. URL: https://books.libreoffice.org/en/BG73/BG73-BaseGuide.pdf (besucht am 2025-01-13) (siehe S. 188).

[31] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 189).

[32] Jonas Gamalielsson und Björn Lundell. "Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice". In: *8th IFIP WG 2.13 International Conference on Open Source Systems: Long-Term Sustainability OSS'2012*. 10.–13. Sep. 2012, Hammamet, Tunisia. Hrsg. von Imed Hammouda, Björn Lundell, Tommi Mikkonen und Walt Scacchi. Bd. 378. IFIP Advances in Information and Communication Technology (IFIPAICT). Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2012, S. 29–47. ISSN: 1868-4238. ISBN: 978-3-642-33441-2. doi:10.1007/978-3-642-33442-9_3 (siehe S. 188).

[33] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: https://www.ibm.com/topics/api (besucht am 2024-12-12) (siehe S. 187).

[34] Dawn Griffiths. *Excel Cookbook – Receipts for Mastering Microsoft Excel*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2024. ISBN: 978-1-0981-4332-9 (siehe S. 188).

[35] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 189).

References V

[36] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 189).

[37] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli "pv" Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke und Travis E. Oliphant. "Array programming with NumPy". *Nature* 585:357–362, 2020. London, England, UK: Springer Nature Limited. ISSN: 0028-0836. doi:10.1038/S41586-020-2649-2 (siehe S. 189).

[38] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 188).

[39] Christian Heimes. "`defusedxml` 0.7.1: XML Bomb Protection for Python stdlib Modules". In: 8. März 2021. URL: https://pypi.org/project/defusedxml (besucht am 2024-12-15) (siehe S. 191).

[40] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 190).

[41] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 189).

[42] John D. Hunter. "Matplotlib: A 2D Graphics Environment". *Computing in Science & Engineering* 9(3):90–95, Mai–Juni 2007. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2007.55 (siehe S. 188).

[43] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom und The Matplotlib Development Team. *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: https://matplotlib.org (besucht am 2025-02-02) (siehe S. 188).

References VI

[44] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 190).

[45] *Information Technology – Universal Coded Character Set (UCS)*. International Standard ISO/IEC 10646:2020. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Dez. 2020 (siehe S. 186).

[46] *Python 3 Documentation. Installing Python Modules*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/installing (besucht am 2024-08-17) (siehe S. 189).

[47] Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*. New York, NY, USA: Apress Media, LLC, Dez. 2018. ISBN: 978-1-4842-4246-9 (siehe S. 188–190).

[48] Katie Kodes. *Intro to XML, JSON, & YAML*. London, England, UK: Payhip, 2019–4. Sep. 2020 (siehe S. 191).

[49] Animesh Kumar, Sandip Dutta und Prashant Pranav. "Analysis of SQL Injection Attacks in the Cloud and in WEB Applications". *Security and Privacy* 7(3), Mai–Juni 2024. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 2475-6725. doi:10.1002/SPY2.370 (siehe S. 123–131).

[50] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 190).

[51] Joan Lambert und Curtis Frye. *Microsoft Office Step by Step (Office 2021 and Microsoft 365)*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Juni 2022. ISBN: 978-0-13-754493-6 (siehe S. 188).

[52] Charles Landau. *TensorFlow Deep Dive: Build, Train, and Deploy Machine Learning Models with TensorFlow*. Sebastopol, CA, USA: O'Reilly Media, Inc., Dez. 2023 (siehe S. 190).

References VII

[53] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 189).

[54] Marc-André Lemburg. *Python Database API Specification v2.0*. Python Enhancement Proposal (PEP) 249. Beaverton, OR, USA: Python Software Foundation (PSF), 12. Apr. 1999. URL: https://peps.python.org/pep-0249 (besucht am 2025-02-02) (siehe S. 26–38, 44–62, 189).

[55] Reuven M. Lerner. *Pandas Workout*. Shelter Island, NY, USA: Manning Publications, Juni 2024. ISBN: 978-1-61729-972-8 (siehe S. 189).

[56] *LibreOffice – The Document Foundation*. Berlin, Germany: The Document Foundation, 2024. URL: https://www.libreoffice.org (besucht am 2024-12-12) (siehe S. 188).

[57] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: https://www.britannica.com/technology/client-server-architecture (besucht am 2025-01-20) (siehe S. 187).

[58] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 189).

[59] Ron McFadyen und Cindy Miller. *Relational Databases and Microsoft Access*. 3. Aufl. Palatine, IL, USA: Harper College, 2014–2019. URL: https://harpercollege.pressbooks.pub/relationaldatabases (besucht am 2025-04-11) (siehe S. 188).

[60] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: 978-1-55860-456-8 (siehe S. 190).

[61] *Microsoft Word*. Redmond, WA, USA: Microsoft Corporation, 2024. URL: https://www.microsoft.com/en-us/microsoft-365/word (besucht am 2024-12-12) (siehe S. 189).

[62] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 187).

References VIII

[63] NumPy Team. *NumPy*. San Francisco, CA, USA: GitHub Inc und Austin, TX, USA: NumFOCUS, Inc. URL: https://numpy.org (besucht am 2025-02-02) (siehe S. 189).

[64] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 189).

[65] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 187).

[66] Ashwin Pajankar. *Hands-on Matplotlib: Learn Plotting and Visualizations with Python 3*. New York, NY, USA: Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7410-1 (siehe S. 188).

[67] Pandas Developers. *Pandas*. Austin, TX, USA: NumFOCUS, Inc. und Montreal, QC, Canada: OVHcloud. URL: https://pandas.pydata.org (besucht am 2025-02-02) (siehe S. 189).

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai und Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS'2019)*. 8.–14. Dez. 2019, Vancouver, BC, Canada. Hrsg. von Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox und Roman Garnett. San Diego, CA, USA: The Neural Information Processing Systems Foundation (NeurIPS), 2019, S. 8024–8035. ISBN: 978-1-7138-0793-3. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html (besucht am 2024-07-18) (siehe S. 189).

[69] Alan Paul, Vishal Sharma und Oluwafemi Olukoya. "SQL Injection Attack: Detection, Prioritization & Prevention". *Journal of Information Security and Applications* 85:103871, Sep. 2024. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 2214-2126. doi:10.1016/J.JISA.2024.103871 (siehe S. 123–131).

References IX

[70] Fabian Pedregos, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot und Edouard Duchesnay. "Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research (JMLR)* 12:2825–2830, Okt. 2011. Cambridge, MA, USA: MIT Press. ISSN: 1532-4435. doi:10.5555/1953048.2078195 (siehe S. 190).

[71] pip Developers. *pip Documentation v24.3.1*. Beaverton, OR, USA: Python Software Foundation (PSF), 27. Okt. 2024. URL: https://pip.pypa.io (besucht am 2024-12-25) (siehe S. 189).

[72] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 189).

[73] Sebastian Raschka, Yuxi Liu und Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-learn*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2022. ISBN: 978-1-80181-931-2 (siehe S. 189, 190).

[74] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 187).

[75] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 187).

[76] Stuart J. Russell und Peter Norvig. *Artificial Intelligence: A Modern Approach (AIMA)*. 4. Aufl. Hoboken, NJ, USA: Pearson Education, Inc. ISBN: 978-1-292-40113-3. URL: https://aima.cs.berkeley.edu (besucht am 2024-06-27) (siehe S. 187).

[77] Stephan Scheuermann. "SQL Injection". In: *1st Kassel Student Workshop on Security in Distributed Systems (KaSWoSDS'2008)*. 13. Feb. 2008, Kassel, Hessen, Germany. Hrsg. von Thomas Weise (汤卫思) und Philipp Andreas Baer. Bd. 2008-1 der Reihe Kasseler Informatikschriften (KIS). Kassel, Hessen, Germany: Universität Kassel, Universitätsbibliothek, 14. Apr. 2008. Kap. 3, S. 35–49. URL: https://kobra.uni-kassel.de/items/a6134d61-d5c3-4cb8-804f-bbb89a3f59a7 (besucht am 2025-08-23) (siehe S. 104–110, 123–131).

[78] Winfried Seimert. *LibreOffice 7.3 – Praxiswissen für Ein- und Umsteiger*. Blaufelden, Schwäbisch Hall, Baden-Württemberg, Germany: mitp Verlags GmbH & Co. KG, Apr. 2022. ISBN: 978-3-7475-0504-5 (siehe S. 188).

References X

[79] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Request for Comments (RFC) 4180. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Okt. 2005. URL: https://www.ietf.org/rfc/rfc4180.txt (besucht am 2025-02-05) (siehe S. 187).

[80] Shai Shalev-Shwartz und Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, England, UK: Cambridge University Press & Assessment, Juli 2014. ISBN: 978-1-107-05713-5. URL: http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning (besucht am 2024-06-27) (siehe S. 189).

[81] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 188).

[82] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 189).

[83] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-commands.html (besucht am 2025-02-25) (siehe S. 190).

[84] Pradeep Kumar Srinivasan und Graham Bleaney. *Arbitrary Literal String Type*. Python Enhancement Proposal (PEP) 675. Beaverton, OR, USA: Python Software Foundation (PSF), 30. Nov. 2021–7. Feb. 2022. URL: https://peps.python.org/pep-0675 (besucht am 2025-03-04) (siehe S. 132–137).

[85] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 183, 190).

[86] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of[85] (siehe S. 190).

[87] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 189, 190).

References XI

[88] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 189).

[89] *The Python Package Index (PyPI)*. Beaverton, OR, USA: Python Software Foundation (PSF), 2024. URL: https://pypi.org (besucht am 2024-08-17) (siehe S. 189).

[90] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 188).

[91] Sherwin John C. Tragura. *Mastering Flask Web and API Development*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83763-322-7 (siehe S. 188).

[92] Laurie A. Ulrich und Ken Cook. *Access For Dummies*. Hoboken, NJ, USA: For Dummies (Wiley), Dez. 2021. ISBN: 978-1-119-82908-9 (siehe S. 188).

[93] Marat Valiev, Bogdan Vasilescu und James D. Herbsleb. "Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem". In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'2018)*. 4.–9. Nov. 2018, Lake Buena Vista, FL, USA. Hrsg. von Gary T. Leavens, Alessandro F. Garcia und Corina S. Păsăreanu. New York, NY, USA: Association for Computing Machinery (ACM), 2018, S. 644–655. ISBN: 978-1-4503-5573-5. doi:10.1145/3236024.3236062 (siehe S. 189).

[94] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 188).

[95] Daniele "dvarrazzo" Varrazzo, Federico "fogzot" Di Gregorio und Jason "jerickso" Erickson. "`Connection` Classes". In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, 21. Juni 2022. URL: https://www.psycopg.org/psycopg3/docs/api/connections.html (besucht am 2025-03-04) (siehe S. 44–54, 78–89).

[96] Daniele "dvarrazzo" Varrazzo, Federico "fogzot" Di Gregorio und Jason "jerickso" Erickson. "`Cursor` Classes". In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, 21. Juni 2022. URL: https://www.psycopg.org/psycopg3/docs/api/cursors.html (besucht am 2025-03-04) (siehe S. 44–56, 78–89).

References XII

[97] Daniele "dvarrazzo" Varrazzo, Federico "fogzot" Di Gregorio und Jason "jerickso" Erickson. *Psycopg*. London, England, UK: The Psycopg Team, 2010–2023. URL: https://www.psycopg.org (besucht am 2025-02-02) (siehe S. 39–42, 189).

[98] Daniele "dvarrazzo" Varrazzo, Federico "fogzot" Di Gregorio und Jason "jerickso" Erickson. *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, 21. Juni 2022. URL: https://www.psycopg.org/psycopg3/docs (besucht am 2025-03-04).

[99] Daniele "dvarrazzo" Varrazzo, Federico "fogzot" Di Gregorio und Jason "jerickso" Erickson. "Static Typing". In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, 21. Juni 2022. URL: https://www.psycopg.org/psycopg3/docs/advanced/typing.html (besucht am 2025-03-04) (siehe S. 132–137).

[100] Pauli "pv" Virtanen, Ralf Gommers, Travis E. Oliphant, Matt "mdhaber" Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan "ilayn" Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregos, Paul van Mulbregt und SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". *Nature Methods* 17:261–272, 2. März 2020. London, England, UK: Springer Nature Limited. ISSN: 1548-7091. doi:10.1038/s41592-019-0686-2. URL: http://arxiv.org/abs/1907.10121 (besucht am 2024-06-26). See also arXiv:1907.10121v1 [cs.MS] 23 Jul 2019. (Siehe S. 190).

[101] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2025. URL: https://thomasweise.github.io/databases (besucht am 2025-01-05) (siehe S. 123–131, 187–189).

[102] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 26–38, 117–122, 189).

[103] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: https://www.ibm.com/think/topics/relational-databases (besucht am 2025-01-05) (siehe S. 189).

References XIII

[104] "With Statement Context Managers". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3.3.9. URL: https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers (besucht am 2024-12-15) (siehe S. 44–51).

[105] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: https://www.techtarget.com/searchdatamanagement/definition/database-management-system (besucht am 2025-01-11) (siehe S. 187).

[106] Ka-Ping Yee und Guido van Rossum. *IteratorS*. Python Enhancement Proposal (PEP) 234. Beaverton, OR, USA: Python Software Foundation (PSF), 30. Jan.–30. Apr. 2001. URL: https://peps.python.org/pep-0234 (besucht am 2025-02-02) (siehe S. 44–66).

[107] François Yergeau. *UTF-8, A Transformation Format of ISO 10646*. Request for Comments (RFC) 3629. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Nov. 2003. URL: https://www.ietf.org/rfc/rfc3629.txt (besucht am 2025-02-05). See Unicode and[45] (siehe S. 190).

[108] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 187).

Glossary (in English) I

AI Artificial Intelligence, see, e.g.,[76]

API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another[33].

Bash is a the shell used under Ubuntu Linux, i.e., the program that "runs" in the terminal and interprets your commands, allowing you to start and interact with other programs[12,62,108]. Learn more at `https://www.gnu.org/software/bash`.

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples for clients are web browsers in the internet as well as clients for DBMSes, such as psql.

client-server architecture is a system design where a central server receives requests from one or multiple clients[8,57,65,74,75]. These requests and responses are usually sent over network connections. A typical example for such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

CSV *Comma-Separated Values* is a very common and simple text format for exchanging tabular or matrix data[79]. Each row in the text file represents one row in the table or matrix. The elements in the row are separated by a fixed delimiter, usually a comma (","), sometimes a semicolon (";"). Python offers some out-of-the-box CSV support in the `csv` module[22].

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*[101].

DBA A *database administrator* is the person or group responsible for the effective use of database technology in an organization or enterprise.

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[105].

Glossary (in English) II

Flask is a lightweight Python framework that allows developers to quickly and easily build web applications[2,16,91]. It is based on the Python WSGI standard[29]. Learn more at https://flask.palletsprojects.com.

LibreOffice is on open source office suite[32,56,78] which is a good and free alternative to Microsoft Office. It offers software such as LibreOffice Calc and LibreOffice Base. See[101] for more information and installation instructions.

LibreOffice Base is a DBMSes that can work on stand-alone files but also connect to other popular relational databases[30,78]. It is part of LibreOffice[32,56,78] and has functionality that is comparable to Microsoft Access[5,17,92].

LibreOffice Calc is a spreadsheet software that allows you to arrange and perform calculations with data in a tabular grid. It is a free and open source spread sheet software[56,78], i.e., an alternative to Microsoft Excel. It is part of LibreOffice[32,56,78].

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows[3,38,81,90,94]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.

Matplotlib is a Python package for plotting diagrams and charts[42,43,47,66]. Learn more at at https://matplotlib.org[43].

Microsoft Access is a DBMSes that can work on DBs stored in single, stand-alone files but also connect to other popular relational databases[5,17,59,92]. It is part of Microsoft Office. A free and open source alternative to this commercial software is LibreOffice Base.

Microsoft Excel is a spreadsheet program that allows users to store, organize, manipulate, and calculate data in tabular structures[9,34,51]. It is part of Microsoft Office. A free alternative to this commercial software is LibreOffice Calc[56,78].

Microsoft Office is a commercial suite of office software, including Microsoft Excel, Microsoft Word, and Microsoft Access[51]. LibreOffice is a free and open source alternative.

Microsoft Windows is a commercial proprietary operating system[11]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

Glossary (in English) III

Microsoft Word	is one of the leading text writing programs[28,61] and part of Microsoft Office. A free alternative to this commercial software is the LibreOffice Writer.
ML	Machine Learning, see, e.g.,[80]
NumPy	is a fundamental package for scientific computing with Python, which offers efficient array datastructures[27,37,47]. Learn more at https://numpy.org[63].
Pandas	is a Python data analysis and manipulation library[4,55]. Learn more at https://pandas.pydata.org[67].
pip	is the standard tool to install Python software packages from the PyPI repository[46,71]. To install a package `thepackage` hosted on PyPI, type `pip install thepackage` into the terminal. Learn more at https://packaging.python.org/installing.
PostgreSQL	An open source object-relational DBMS[31,64,72,88]. See https://postgresql.org for more information.
psql	is the client program used to access the PostgreSQL DBMS server.
psycopg	or, more exactly, `psycopg 3`, is the most popular PostgreSQL adapter for Python, implementing the Python DB API 2.0 specification[54]. Learn more at https://www.psycopg.org[97].
PyPI	The Python Package Index (PyPI) is an online repository that provides the software packages that you can install with pip[10,89,93]. Learn more at https://pypi.org.
Python	The Python programming language[41,53,58,102], i.e., what you will learn about in our book[102]. Learn more at https://python.org.
PyTorch	is a Python library for deep learning and AI[68,73]. Learn more at https://pytorch.org.
relational database	A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other[19,35,36,82,87,101,103].

Glossary (in English) IV

Scikit-learn is a Python library offering various machine learning tools[70,73]. Learn more at https://scikit-learn.org.

SciPy is a Python library for scientific computing[47,100]. Learn more at https://scipy.org.

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers[14] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the "server computer"[50].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[15,23–25,44,60,83,85–87]. It is understood by many DBMSes. You find the SQL commands supported by PostgreSQL in the reference[83].

TensorFlow is a Python library for implementing machine learning, especially suitable for training of neural networks[1,52]. Learn more at https://www.tensorflow.org.

terminal A terminal is a text-based window where you can enter commands and execute them[3,18]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

Ubuntu is a variant of the open source operating system Linux[18,40]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

URI A *Uniform Resource Identifier* is an identifier for an abstract or physical resource in the internet[107]. It can be a Uniform Resource Locator (URL), a name, or both. URIs are supersets of URLs. The connection strings of the PostgreSQL DBMS are examples for URIs.

Glossary (in English) V

URL A *Uniform Resource Locator* identifies a resource in the WWW and a way to obtain it by describing a network access mechanism. The most notable example of URLs is the text you write into web browsers to visit websites[7]. URLs are subsets of URIs.

WWW World Wide Web[6,26]

XML The *Extensible Markup Language* is a text-based language for storing and transporting of data[13,20,48]. It allows you to define elements in the form `<myElement myAttr="x">...text..</myElement>`. Different from CSV, elements in XML can be hierarchically nested, like `<a><b><c>test</c></b><b>bla</b></a>`, and thus easily represent tree structures. XML is one of most-used data interchange formats. To process XML in Python, use the `defusedxml` library[39], as it protects against several security issues.