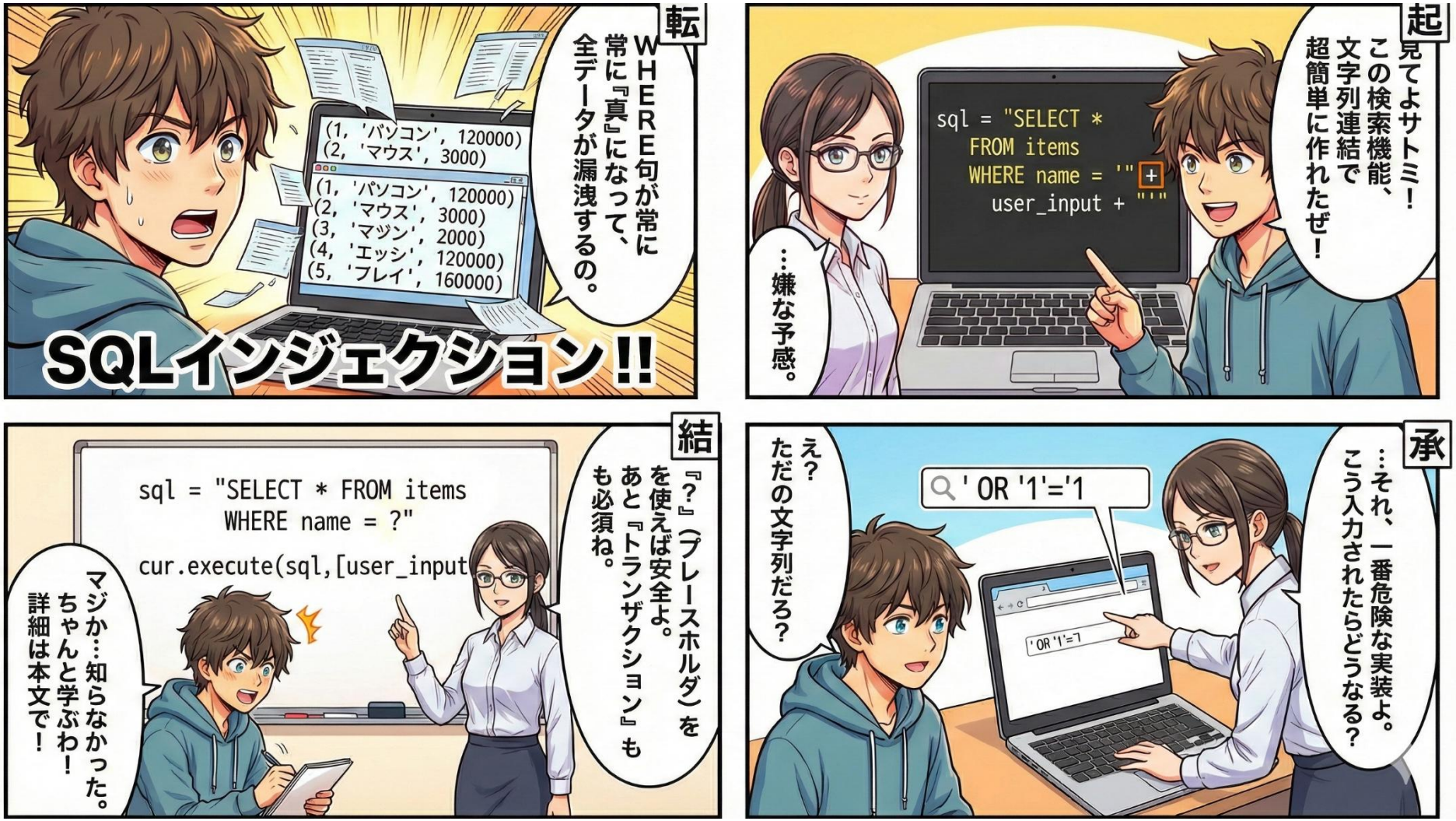
14. データベースアプリケーションの実践

URL: https://www.kkaneko.jp/de/ds/index.html

金子邦彦

謝辞：この資料では「いらすとや」のイラストを使用しています



Nano Banana Pro を用いて作成

プログラム（Python）によるデータベースの自動操作を学ぶ。

1. **仕組みと基本操作**: PythonからSQLを実行する方法
2. **セキュリティ攻撃から守る方法**: プレースホルダ
3. **整合性とトランザクション**: 失敗したときに元に戻す方法（ロールバック）

今日身につける知識、スキル

【対象】Pythonを用いて**データベースを操作**する**アプリケーション開発の基礎**を学びたい人。

　特に、**セキュリティ**（SQLインジェクション対策）や**データの整合性**（トランザクション処理）といった、安全なシステム構築を習得したい

- **SQLインジェクション**: **ユーザーからの入力データ**が**SQL命令の一部として不正に実行**されてしまう脆弱性。
- **プレースホルダ**: SQL文の中で、後から**データが入る場所**を**あらかじめ「?」記号で確保**しておく仕組み（Python 言語では「?」）。SQL インジェクション対策にもなる

14-1 仕組みと基本操作

イントロダクション ～手動から自動へ～

- これまでの授業：**人間がSQLを手入力していた**
- 現実のシステム：**プログラムがSQLを自動実行している**
- 本日のテーマ：**プログラム（Python）にSQLを実行させる**
- ゴール：**Pythonからデータベースに安全にデータを渡す方法**を習得する

ドライバとは何か

- **Python**と**データベース**は**別のシステム**であり、そのままでは**通信できない**
- **ドライバ：両者の間の通信を担うソフトウェア**
  例）プリンタを使うには「プリンタドライバ」が必要
- データベースを使うには「**データベースドライバ**」が必要

  （今回は **sqlite3** を使用）

[Python] ←——— sqlite3 ———→ [データベース]

ドライバ

- **ドライバ＝異なるシステム間の橋渡しをするソフトウェア**の総称

sqlite3 ドライバの接続と切断のルール

- 接続：connect() で通信を開始する
- 切断：close() で通信を終了する

  close() は必ず実行すること

プログラム例

```
conn = sqlite3.connect('shop.db')  # 接続
# ... 処理 ...
conn.close()                       # 切断
```

カーソルとは何か

- **接続**しても、**データ**は**自動的に流れない**
- **カーソル：SQLを実行**し、**結果を取得するオブジェクト**
- 画面上の「カーソル」が現在位置を示すように、データベースのカーソルも、**検索結果の中での現在位置**を管理する

[Python] —— conn（道路） —— [データベース]

↑

cur（トラック）

sqlite3 のカーソルの使い方

- 行き：**SQLをデータベースシステムに送る**

→ execute()

- 帰り：結果を戻す

→ fetchone() / fetchall()

プログラム例

```
cur = conn.cursor() # カーソルを作成
cur.execute("SELECT ...") # SQLを実行 result =
cur.fetchall() # 結果を取得
```

fetchone() ・・・ *1*件だけ必要なとき

fetchall() ・・・ 全件必要なとき

データは「リスト」に入れて渡す

- データは**リスト（角括弧 []）**に入れて渡す
- **SQL文の？と、リストの要素が順番に対応する**

プログラム例

```
data_list = [1, 'パソコン', 120000] cur.execute("INSERT INTO items VALUES (?, ?, ?)", data_list)
```

? の位置	対応する値
1番目　1	（id）
2番目　'パソコン'	（name）
3番目　120000	（price）

14-2 セキュリティ

やってはいけないこと

- 文字列連結（+）でSQLを組み立ててはならない
- **ユーザーの入力**が**SQL命令の一部**になってしまう

```
# 危険なコード（使用しないこと）
sql = "SELECT * FROM items WHERE name = '" + user_input + "'"
```

正常な動作と攻撃の比較

正常な場合：**ユーザーが「マウス」と入力**

SELECT * FROM items WHERE name = 'マウス’

→ マウスだけが検索される（意図通り）

攻撃された場合：**ユーザーが ' OR '1'='1 と入力**

SELECT * FROM items WHERE name = '' OR '1'='1’

→ ‘1’=‘1’ は常に真なので、全商品が取得されてしまう。情報漏洩。

SQLインジェクション

- **SQLインジェクション**：ユーザーの入力がSQL命令の一部として解釈される脆弱性
- **なぜ全データが漏洩するのか**
- WHERE句の条件を分解すると：
  - name = ''（名前が空文字）**または**
  - '1'='1'（常に真）
- WHERE句全体が常に真となり、全データが取得される
- **結論**：ユーザーの入力を信用してSQL文と連結してはならない

解決策 ～プレースホルダ～

- SQL文には？（**プレースホルダ**）を記述
- **データ**は別途**リストで渡す**

<u>プログラム例</u>

```
sql = "SELECT * FROM items WHERE name = ?“
data_list = [user_input]
cur.execute(sql, data_list)
```

1. ？を含むSQL文を使用
2. ？の位置にデータが埋め込まれる

なぜプレースホルダは安全なのか

- 「WHERE句には条件が1つある」という構造が確定する

SELECT * FROM items WHERE name = ?

- ' OR '1'='1 という文字列が渡されても：
  - 「name列と比較する1つの文字列データ」として扱われる
  - SQL命令として解釈されることはない

  （データが命令としてSQL命令として解釈される可能性がない）

14-3 整合性とトランザクション

トランザクション

- **複数の操作**を「**1つのまとまり**」として扱う仕組み
- 「**すべて成功**」か「**すべて取り消し**」のどちらかを保証する

具体例：銀行の振込（AさんからBさんに1万円を送金）

操作1. Aさんの口座から1万円を引く

操作2. Bさんの口座に1万円を足す

この2つの操作は「**セット**」で成功しなければならない。

トランザクションの必要性

操作1. Aさんの口座から1万円を引く

操作2. Bさんの口座に1万円を足す

問題：操作1の後にシステム障害が発生した場合

- Aさんの口座：1万円が減る
- Bさんの口座：1万円が増えない

⇒　**1万円が消失する**

解決：トランザクションを使用する

- **コミット（Commit）**：すべて成功したら、まとめて確定
- **ロールバック（Rollback）**：途中で失敗したら、すべて取り消し
- これにより、データの整合性（矛盾のない状態）が保たれる。

コミット

- プログラムからの変更は、最初は「**仮保存**」状態
- sqlite3 ドライバでは、**conn.commit()** を実行するとコミット（データベースに確定）

類似例：Wordで「保存」ボタンを押すまでファイルに反映されないのと同じ

なぜ仮保存状態にするのか

- すべての操作が成功したことを確認してから、**まとめて確定**できる
- **途中で失敗**しても、**確定前なら元に戻せる**

ロールバックとは

- sqlite3 ドライバで conn.rollback() を実行すると

  ⇒　操作がすべて取り消される

類似例：ワードで「保存せずに閉じる」と同じ

トランザクション、コミット／ロールバック

トランザクション

- 複数のデータベース操作を「**1つのまとまり**」として扱う。
- 「**すべて成功するか、すべて取り消すか**」を保証し、データの不整合を防ぐ。

コミット / ロールバック

- トランザクション処理において、変更を**データベースに確定**する操作（**コミット**）
- **途中までの変更を破棄**して**元に戻す**操作（**ロールバック**）。

まとめ ～ Python での 3つの原則 ～

1. **データ**は**リスト** [] に**格納**して渡す
2. **SQL**には？**を使用**し、**文字列連結（+）を行わない**
3. エラーが発生したら **rollback でデータを復旧**する

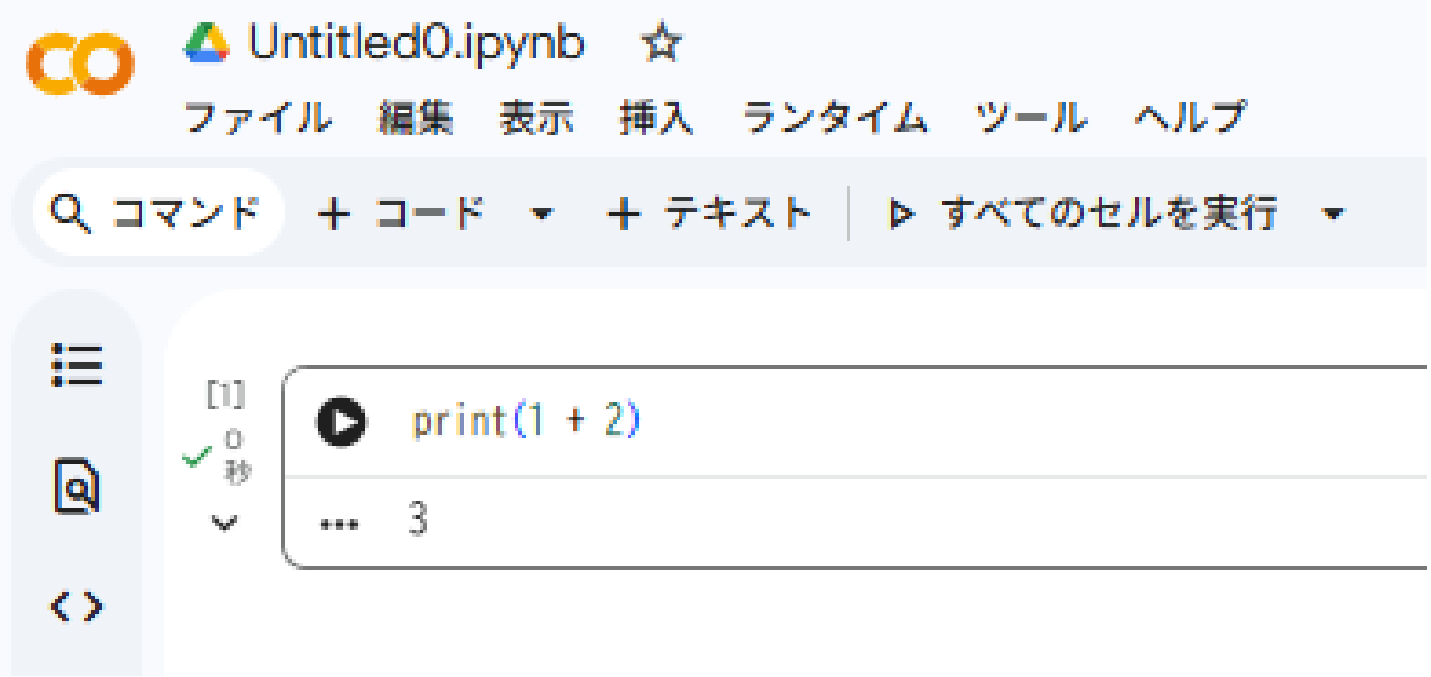
用語集

- ドライバ：プログラミング言語とデータベースの間の通信を担うソフトウェア
- カーソル：SQL文を実行し、結果を取得するためのオブジェクト
- プレースホルダ：SQL文中でデータの挿入位置を示す記号（?）
- SQLインジェクション：悪意のある入力によってSQL文を改ざんする攻撃手法
- トランザクション：複数の操作を1つのまとまりとして扱い、整合性を保証する仕組み
- コミット：仮保存状態の変更をデータベースに確定する操作
- ロールバック：変更を取り消す操作

14-4 Google Colaboratory

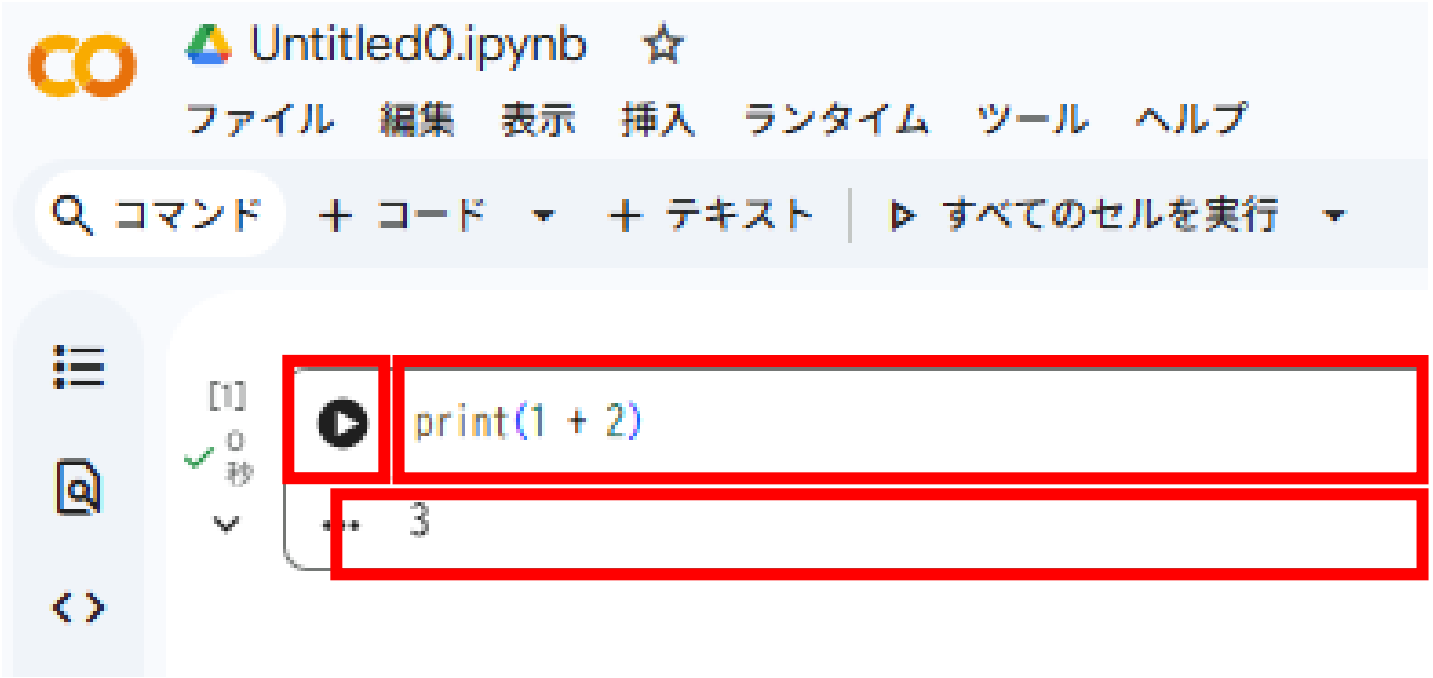
Google Colaboratory のノートブックとは

- **プログラム**と**実行結果**を**一緒に保存できるファイル**
- スマホの**メモ帳**に近いが、**書いた内容を「実行」できる点**が違う
- Google Driveに保存可能



画面の見方

- **コードセル**：グレーの枠。ここに**プログラムを書く**
- **実行ボタン**：セルの左側にある**再生ボタン**のような**三角形。クリックで実行**
- **出力エリア**：セルの下。**結果がここに出る**



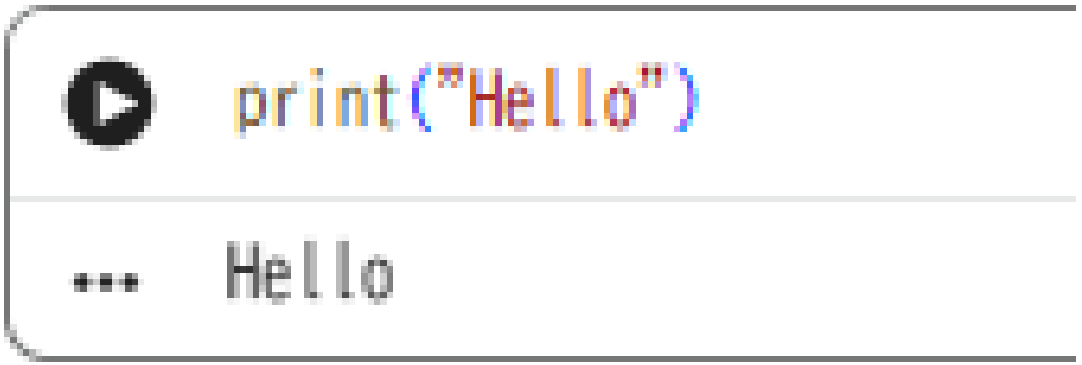
Google Colaboratory でのプログラムの動作法

①コードセルに次を入れる

```
print("Hello")
```

②入力できたら、実行

- 方法1：**三角形の実行ボタン**を**クリック**
- 方法2：**Shiftキーを押しながらEnterキー**
- セルの下に Hello と表示されたら成功



プログラミングの基本サイクル

入力

```
print("Hello")
```

→ 実行

→ 結果を見る

```
Hello
```

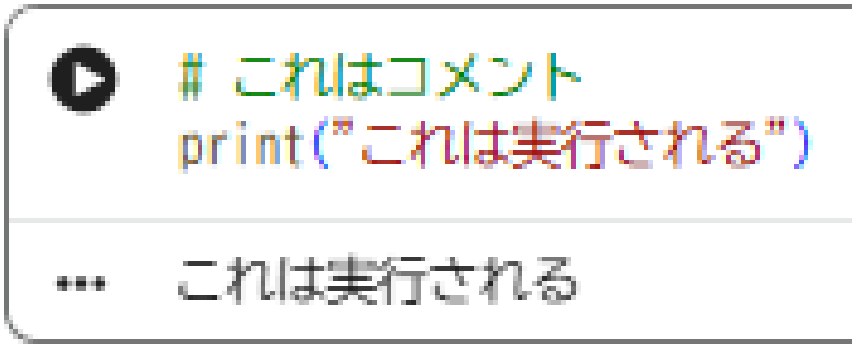
このサイクルの繰り返しがプログラミングの基本

コメントとは

#（シャープ）で始まる行は「コメント」

 # これはコメント

print("これは実行される")



- **コメント**は**プログラムとして実行されない**
- **メモ書き**として使う
- **Google ColaboratoryではAIへの指示にも使える**
- 日本語で書ける

14-5 データベースアプリケーションの実践

本日の演習：データベースの「攻撃」と「防御」を体験する

1. お店のデータベースを作る（全体の**準備**）
2. SQLインジェクション攻撃で**全データが漏洩**する瞬間を見る
3. プレースホルダで**攻撃を防御**する
4. **データを変更**して「**確定**」する
5. **データを削除した直後**に**プログラムが停止**する
6. ロールバックで**削除したデータが復活**する
7. 切断

演習の手順

Google Colaboratory を使用

- ブラウザ上で動作するPython実行環境

<u>演習の進め方</u>

- 各セルを**上から順番に**実行する
- **セル5**では**エラー**が発生するが、**それは正常な動作**である
- **セル7で接続を切断した後**に**再度演習を行う**場合は、**セル1から実行し直す**

警告

- 本演習で学ぶSQLインジェクションは、**防御方法を理解するため**の学習である。
- 他者が管理するシステムに対してSQLインジェクション攻撃を試みることは、**不正アクセス禁止法に違反する犯罪行為**である。
- 本演習の知識は、自身が開発するシステムの安全性向上のためにのみ使用すること。

演習１　～お店を作る～

演習で使用するデータベースを準備する。

1. Google Colabのノートブックを開く
   https://colab.research.google.com/drive/18F9rdskdANKrWkB9CTbzyi8e8Gpuj18V?usp=drive_link
2. セル1を見つける
3. セル1を実行する（Shift + Enter または再生ボタン）

期待される出力

準備完了：お店を開店しました。

注目ポイント

- この出力は、**3つの商品**が登録されたことを示す

パソコン（120,000円）、マウス（3,000円）、キーボード（8,000円）

- `commit()` により、データが確定された
- この時点のデータを、後の演習で使う

演習2　～SQLインジェクション攻撃～

文字列連結でSQLを組み立てると、情報漏洩が発生することを確認する。

1. セル1を実行済みであることを確認する
2. セル2を見つける
3. セル2を実行する（Shift + Enter または再生ボタン）

期待される出力

実行される危険なSQL: SELECT * FROM items WHERE name = '' OR '1'='1'

--- 検索結果 ---

[(1, 'パソコン', 120000), (2, 'マウス', 3000), (3, 'キーボード', 8000)]

注目ポイント

- **全商品**が表示されてしまった
- 本来は特定の商品名で検索する処理であった
- `' OR '1'='1` という入力により、WHERE句が常に真になった
- これが「SQLインジェクション攻撃」による情報漏洩である

演習3 ～プレースホルダで守る～

プレースホルダ（`?`）を使うと、同じ攻撃が防御できることを確認する。

1. セル2を実行済みであることを確認する
2. セル3を見つける
3. セル3を実行する（Shift + Enter または再生ボタン）

期待される出力

--- 安全な検索結果 ---

[]

注目ポイント

- **何も表示されなかった（攻撃失敗）**
- 同じ攻撃データ `' OR '1'='1` を送信したのに、結果が異なる
- プレースホルダ `?` を使ったため、攻撃文字列は「ただの検索文字列」として処理された
- `?` を使うだけで、**SQLインジェクションを防御**できる

演習4 ～更新とコミット～

データを変更し、`commit()` で確定する操作を体験する。

Google Colabのノートブックを開く

1. セル3を実行済みであることを確認する
2. セル4を見つける
3. セル4を実行する（Shift + Enter または再生ボタン）

期待される出力

値上げを確定しました。

[(2, 'マウス', 3500)]

注目ポイント

- マウスの価格が3,000円から3,500円に変更された
- `commit()` により、この変更が**確定**された
- この時点が、**次の演習以降で「戻る先（ロールバック先）」**になる
- 演習6で、この状態に戻ることを確認する

演習5 ～エラー体験～

`commit()` の前にエラーが発生すると、変更が確定されないことを確認する。

1. セル4を実行済みであることを確認する
2. セル5を見つける
3. セル5を実行する（Shift + Enter または再生ボタン）

期待される出力

--- ここからトラブル実験 ---

パソコンを削除してしまいました...（まだ未確定）

その後、赤い背景で以下のエラーが表示される：
ZeroDivisionError: division by zero ：0で割り算を行ったときに発生するエラー

注目ポイント

- パソコンのデータは削除されたように見える
- しかし、エラーにより **`commit()` が実行されなかった**
- つまり、削除は「**仮保存**」のまま、確定されていない

演習6 ～手動ロールバック～

`rollback()` を実行すると、最後の確定状態に戻ることを確認する。

1. セル5を実行済みであることを確認する
2. セル6を見つける
3. セル6を実行する（Shift + Enter または再生ボタン）

期待される出力

ロールバックを実行しました。

--- 在庫確認 ---

[(1, 'パソコン', 120000), (2, 'マウス', 3500), (3, 'キーボード', 8000)]

注目ポイント

- **パソコンのデータが復旧した**
- `rollback()` により、「最後の確定状態」（演習4でコミットした直後）に戻った
- マウスは3,500円のまま（演習4で確定済みのため）
- **これがトランザクションの機能である**

演習の総括: 体験したこと

1. データベースを作成
2. 全データが漏洩。文字列連結は危険である
3. 攻撃を防御した。`?` を使えば安全である
4. データを変更して確定した。コミット
5. エラーでプログラムが停止した
6. 削除したデータが復旧した。`rollback()` で元に戻せる