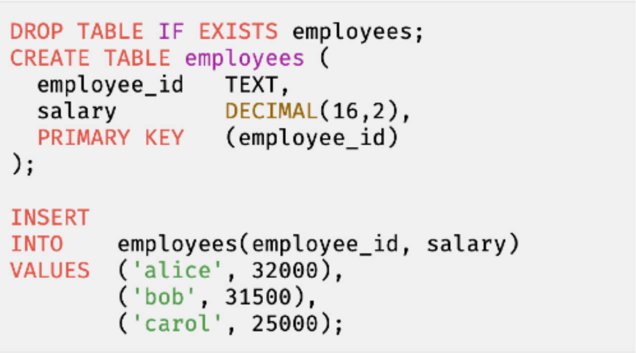
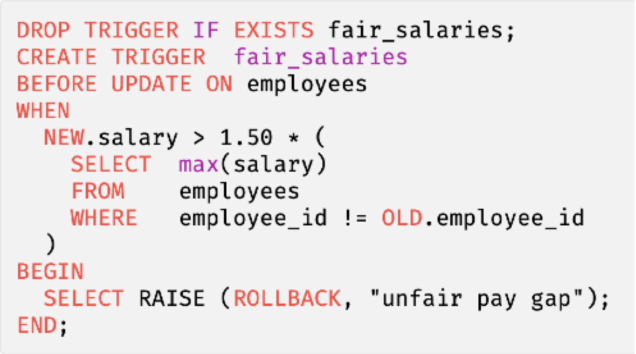
```
import sqlite3


db = sqlite3.connect("course-database.sqlite")


def find_available_groups(assignment_number):
    c = db.cursor()
    c.execute(
        """
        SELECT    group_number
        FROM      group_memberships
        WHERE     assignment_number = ?
        GROUP BY  group_number
        HAVING    count() < 4
        """,
        [assignment_number]
    )
    return [group_number for group_number, in c]


def main():
    assignment_number = int(input("Assignment number: "))
    print(f"Looking for a group for assignment {assignment_number}")
    for group_number in find_available_groups(assignment_number):
        print(f"+ {group_number}")


main()
```

Skapa och föra in värden:



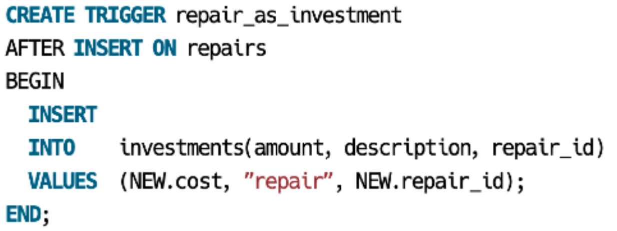
```
DROP TABLE IF EXISTS employees;
CREATE TABLE employees (
  employee_id   TEXT,
  salary        DECIMAL(16,2),
  PRIMARY KEY   (employee_id)
);

INSERT
INTO      employees(employee_id, salary)
VALUES    ('alice', 32000),
          ('bob', 31500),
          ('carol', 25000);
```

Before trigger:



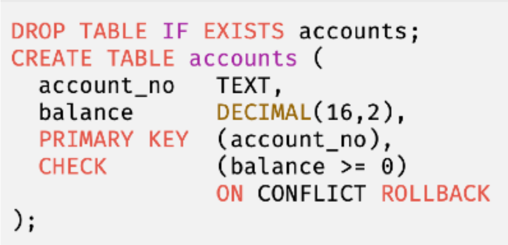
```
DROP TRIGGER IF EXISTS fair_salaries;
CREATE TRIGGER  fair_salaries
BEFORE UPDATE ON employees
WHEN
  NEW.salary > 1.50 * (
    SELECT  max(salary)
    FROM    employees
    WHERE   employee_id != OLD.employee_id
  )
BEGIN
  SELECT RAISE (ROLLBACK, "unfair pay gap");
END;
```

After trigger:



```
CREATE TRIGGER repair_as_investment
AFTER INSERT ON repairs
BEGIN
  INSERT
  INTO    investments(amount, description, repair_id)
  VALUES  (NEW.cost, ”repair”, NEW.repair_id);
END;
```

Check before insert:



```
DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
  account_no    TEXT,
  balance       DECIMAL(16,2),
  PRIMARY KEY   (account_no),
  CHECK         (balance >= 0)
                ON CONFLICT ROLLBACK
);
```

Atomicity: transaction either happens or dont.
Consistency: if all constraints hold when we begin a transaction, they hold when finished. We go from one consistent state to another.
Isolation: run without interference from other transactions.
Durability: if a crash happens after a commit, all effects remain.

Isolation levels:

Read uncommitted: we read whatever value is currently presented to the database, committed or not. Very high concurrency, almost no lock-up.

Read committed: we read the latest value that has been committed, meaning that reads in the same transaction might give different results.

Repeatable read: the same read will give the same value in the transaction, but different values may be taken from different values. Might get phantom reads, meaning that if we add rows it will affect the next read of same value

Serializable: transactions happen as if they were ordered in a queue, no one messes with the database during your turn.