Principles of Database Systems (CS307)

Lecture 6: Advanced SQL

Zhong-Qiu Wang

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

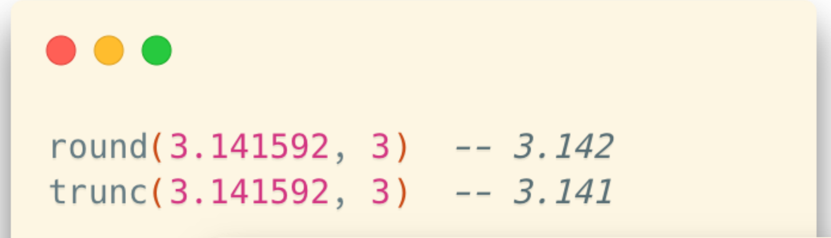
Announcements

- Second assignment is out, due date: 21st Oct., Tuesday, 10pm
  - Do not miss the deadline, or you will receive reduced scores

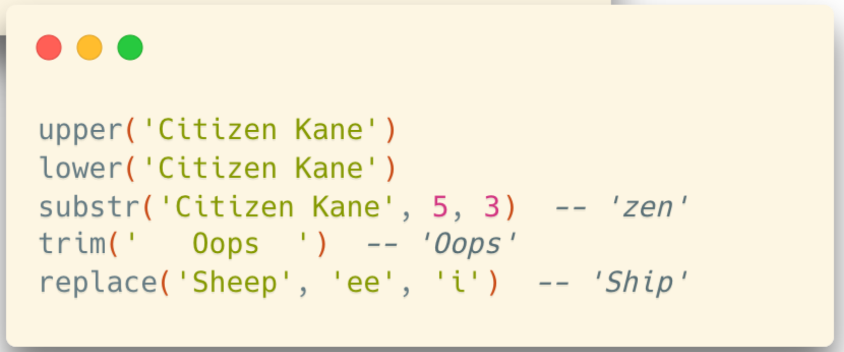
Function

Built-in Functions

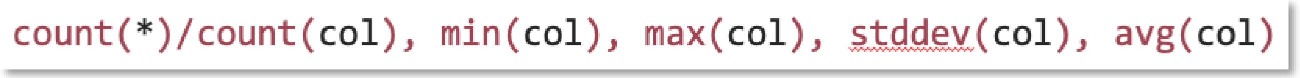
- Most DBMS provides a series of built-in functions
  - E.g., Scalar function, aggregation function, window function



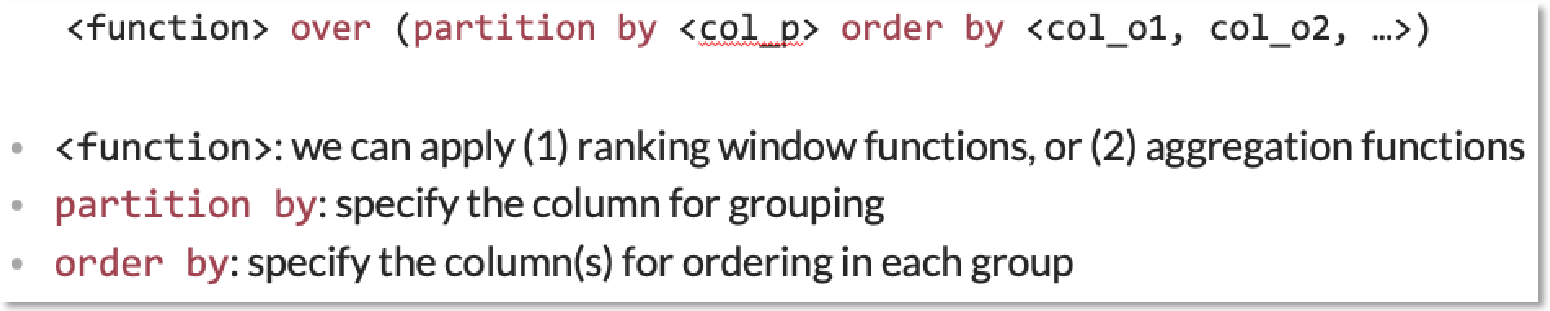
```
round(3.141592, 3)  -- 3.142
trunc(3.141592, 3)  -- 3.141
```



```
upper('Citizen Kane')
lower('Citizen Kane')
substr('Citizen Kane', 5, 3)  -- 'zen'
trim('   Oops   ')  -- 'Oops'
replace('Sheep', 'ee', 'i')  -- 'Ship'
```



```
count(*)/count(col), min(col), max(col), stddev(col), avg(col)
```



```
<function> over (partition by <col_p> order by <col_o1, col_o2, …>)
```

- <function>: we can apply (1) ranking window functions, or (2) aggregation functions
- partition by: specify the column for grouping
- order by: specify the column(s) for ordering in each group

Self-defined Function

- Sometimes the built-in functions cannot fulfill our requirements
  - And the power of declarative language (SQL) is not strong enough

- Most DBMS implement a built-in, SQL-based programming language
  - A procedural extension to SQL

Procedural vs. Declarative

- Two different programming paradigms
  - Imperative programming （命令式编程）
    - Describe the algorithms step-by-step (i.e., how to do)
    - Procedural （过程式）: C (and many other legacy languages)
    - Object-oriented: Java
  - Declarative programming（声明式编程）
    - Describe the result without specifying the detailed steps (i.e., what to do)
    - (Pure) declarative: SQL, Regular Expressions, Markup (HTML, XML), CSS
    - Functional: Scheme, Haskell, Scala, Erlang
    - Logic programming: Prolog

Procedural vs. Declarative

- E.g., How can we get a cup of tea?
  - In a procedural way:

1. Get a cup
2. Get some tea
3. Get some hot water
4. Put tea into the cup
5. Pour hot water into the cup
6. `return tea;`

  - In a declarative way:

<a cup of tea/>

    - You don’t really need to know how to make a cup of tea
    - The system can do it in a black-box manner

Procedural vs. Declarative

- E.g., Find all Chinese movies before 1990 in the movies table?
  - In a procedural way:

```
1. Read the movies table into the memory
2. For each row i in the table, repeat:
2.1 In row i, read the value of the column “country”
2.2 if …
```

  - In a declarative way:

```
select * from movies where country = ‘cn’ and year_released < 1990
```

    - You don’t really need to know how to filter the table
    - The DBMS system can do it in a black-box manner

Procedural vs. Declarative

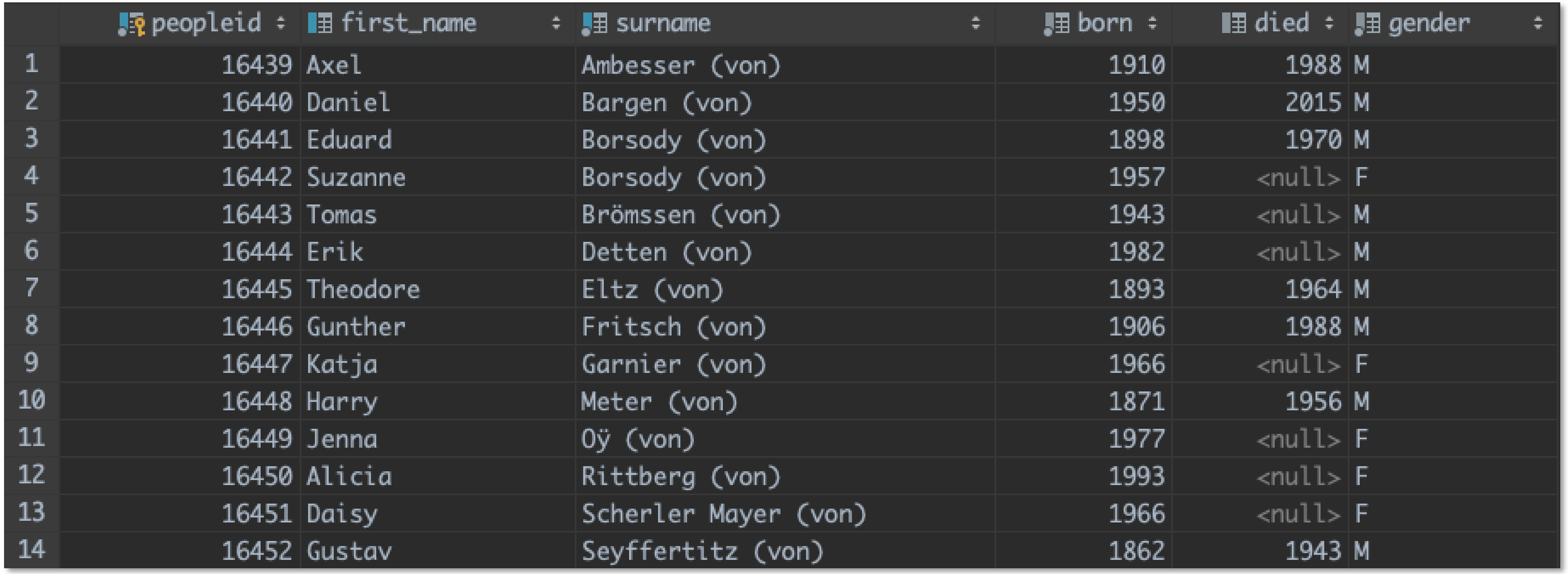
- Benefits in declarative languages
  - No need to understand the details
    - The systems take in charge of all the details
  - Easier to use than imperative programming
    - More user-friendly
- Problems in declarative languages
  - Cannot specify the control flow of a program
    - If there is no such command as <a cup of tea/>, you need to create it by yourself

Procedural Extension to SQL

- Many DBMS products provide a proprietary procedural extension to the standard SQL
  - Transact-SQL (T-SQL) Microsoft SQL Server
  - PL/SQL ORACLE
  - PL/PGSQL PostgreSQL
  - (No specific name) MySQL
  - (Not supported) SQLite

Function in (Postgre)SQL

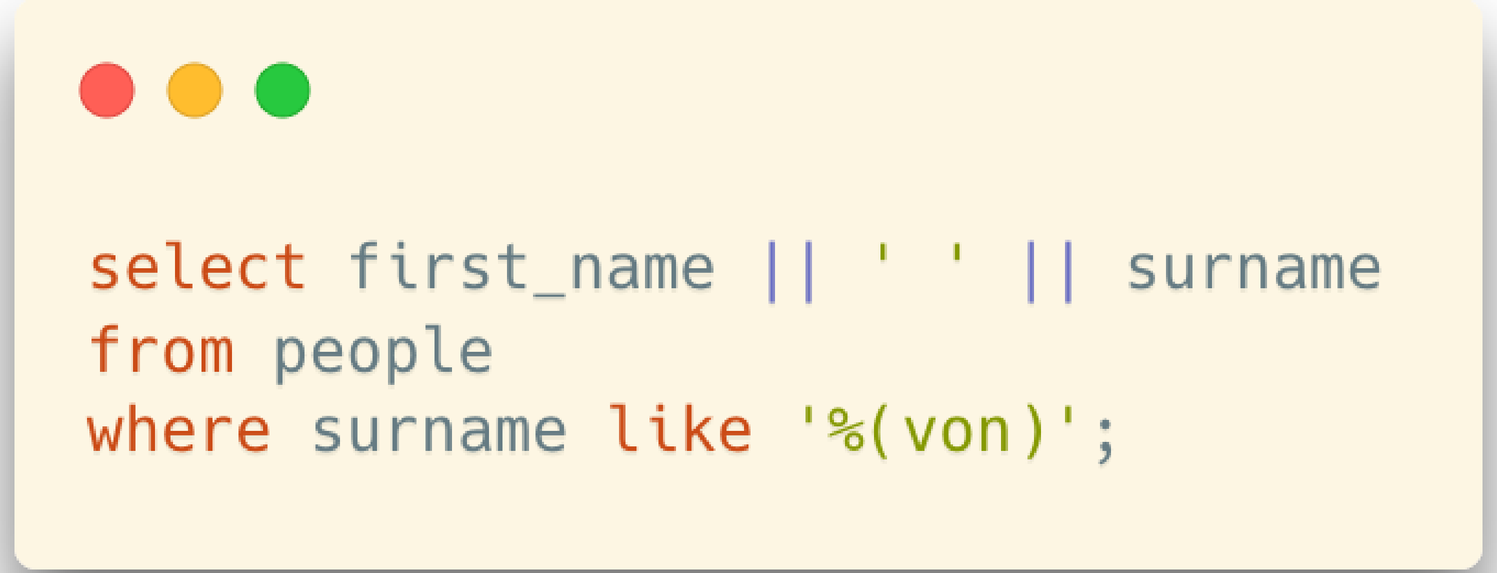
- Example: Display the full name for people with “von”
  - When introducing `update`, we have modified the names starting with “von” into “… (von)” for ordering
    - `von Neumann -> Neumann (von)`



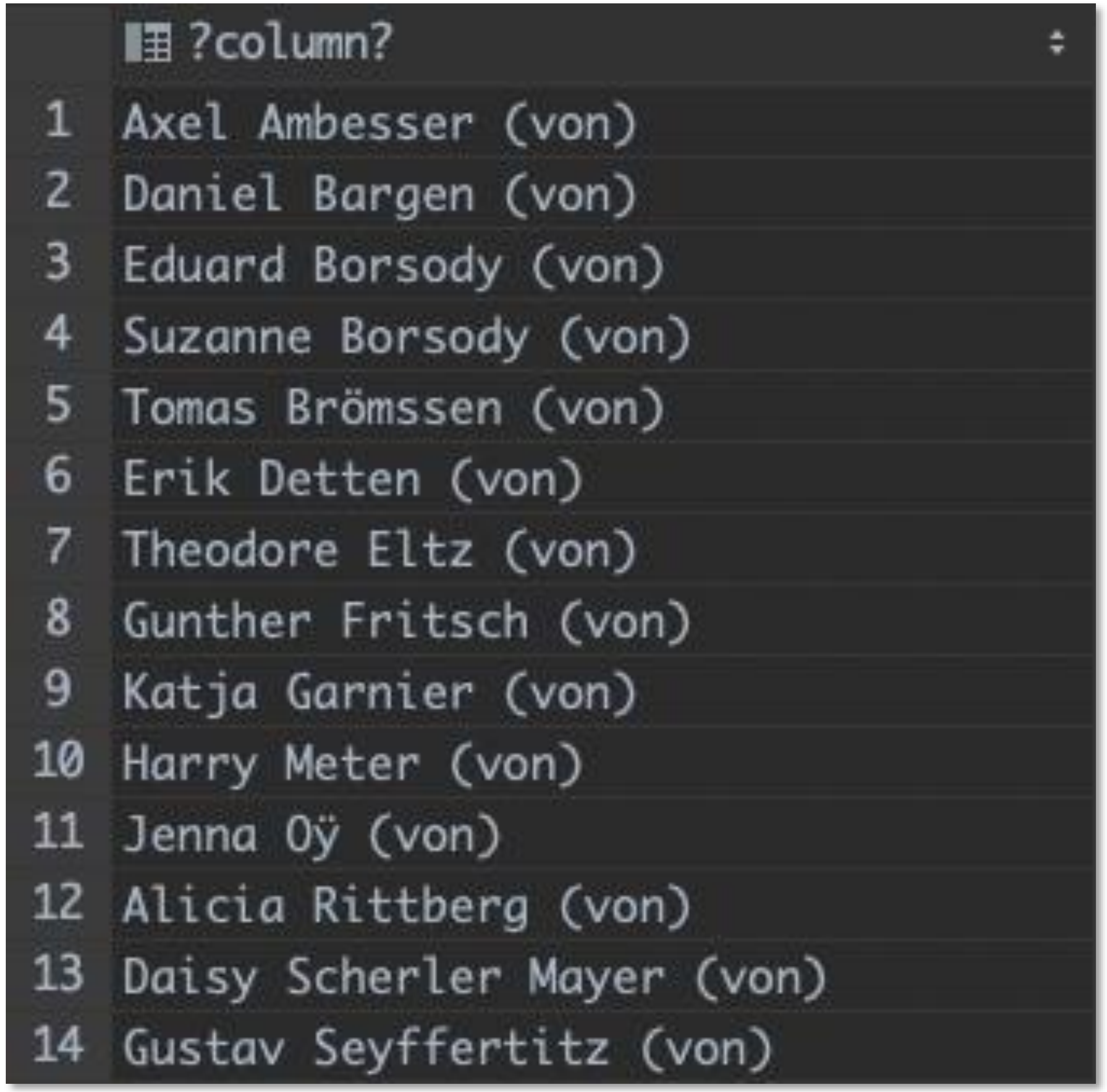
	peopleid	first_name	surname	born	died	gender
1	16439	Axel	Ambesser (von)	1910	1988	M
2	16440	Daniel	Bargen (von)	1950	2015	M
3	16441	Eduard	Borsody (von)	1898	1970	M
4	16442	Suzanne	Borsody (von)	1957	<null>	F
5	16443	Tomas	Brömssen (von)	1943	<null>	M
6	16444	Erik	Detten (von)	1982	<null>	M
7	16445	Theodore	Eltz (von)	1893	1964	M
8	16446	Gunther	Fritsch (von)	1906	1988	M
9	16447	Katja	Garnier (von)	1966	<null>	F
10	16448	Harry	Meter (von)	1871	1956	M
11	16449	Jenna	Oÿ (von)	1977	<null>	F
12	16450	Alicia	Rittberg (von)	1993	<null>	F
13	16451	Daisy	Scherler Mayer (von)	1966	<null>	F
14	16452	Gustav	Seyffertitz (von)	1862	1943	M

Function in (Postgre)SQL

- If we simply concatenate the first name and the last name, it looks like this:
  - A little bit weird format (a trailing “von”)



```sql
select first_name || ' ' || surname
from people
where surname like '%(von)';
```



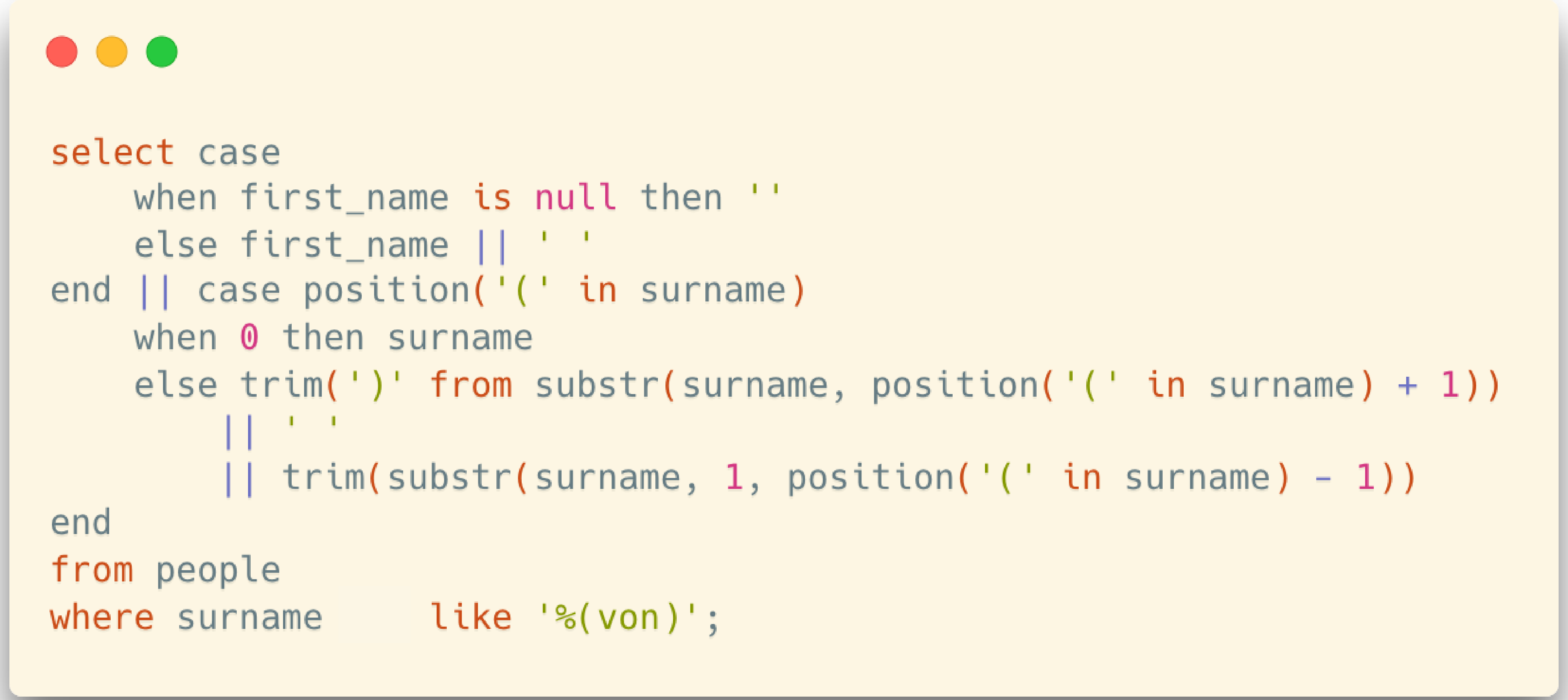
```
    ?column?
1   Axel Ambesser (von)
2   Daniel Bargen (von)
3   Eduard Borsody (von)
4   Suzanne Borsody (von)
5   Tomas Brömssen (von)
6   Erik Detten (von)
7   Theodore Eltz (von)
8   Gunther Fritsch (von)
9   Katja Garnier (von)
10  Harry Meter (von)
11  Jenna Oÿ (von)
12  Alicia Rittberg (von)
13  Daisy Scherler Mayer (von)
14  Gustav Seyffertitz (von)
```

Function in (Postgre)SQL

- Question: How can we restore the format into "**first_name von surname**"?
  - String operations
  - `i.e., Neumann (von) -> von Neumann`

Function in (Postgre)SQL

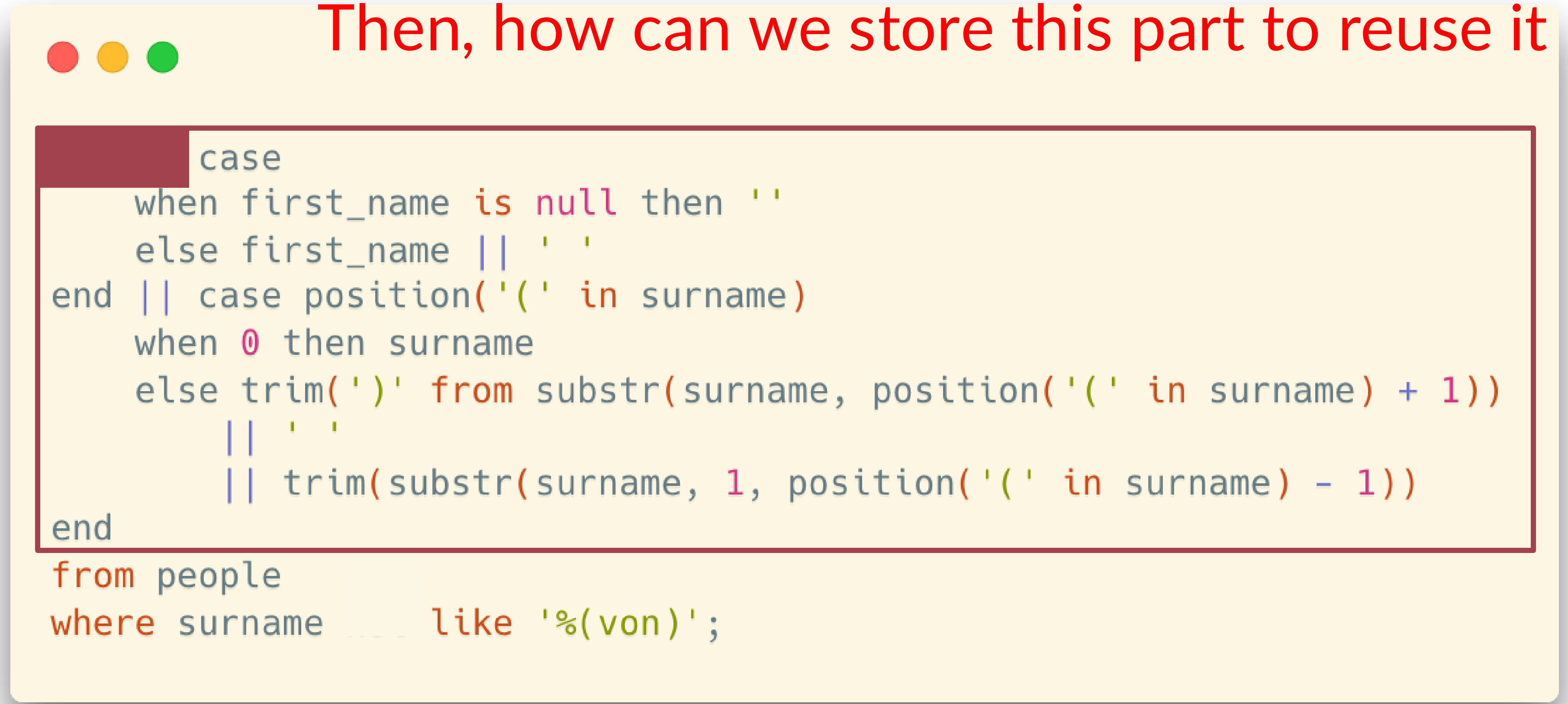
- Question: How can we restore the format into “**first_name von surname**”?
  - String operations
  - i.e., **Neumann (von) -> von Neumann**



```
select case
    when first_name is null then ''
    else first_name || ' '
end || case position('(' in surname)
    when 0 then surname
    else trim(')' from substr(surname, position('(' in surname) + 1))
        || ' '
        || trim(substr(surname, 1, position('(' in surname) - 1))
end
from people
where surname     like '%(von)';
```

Function in (Postgre)SQL

- Question: How can we restore the format into “**first_name von surname**”?
    - String operations

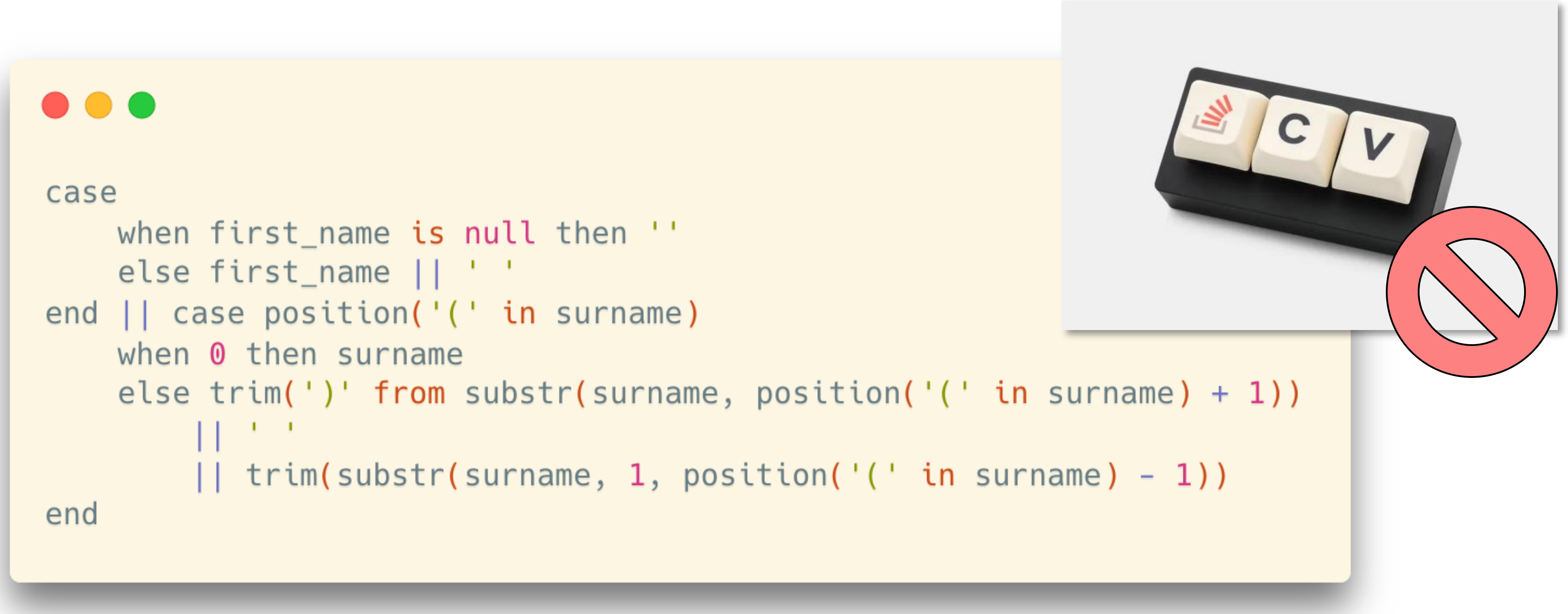


Then, how can we store this part to reuse it in the future?

```
[redacted] case
    when first_name is null then ''
    else first_name || ' '
end || case position('(' in surname)
    when 0 then surname
    else trim(')' from substr(surname, position('(' in surname) + 1))
        || ' '
        || trim(substr(surname, 1, position('(' in surname) - 1))
end
from people
where surname     like '%(von)';
```

Function in (Postgre)SQL

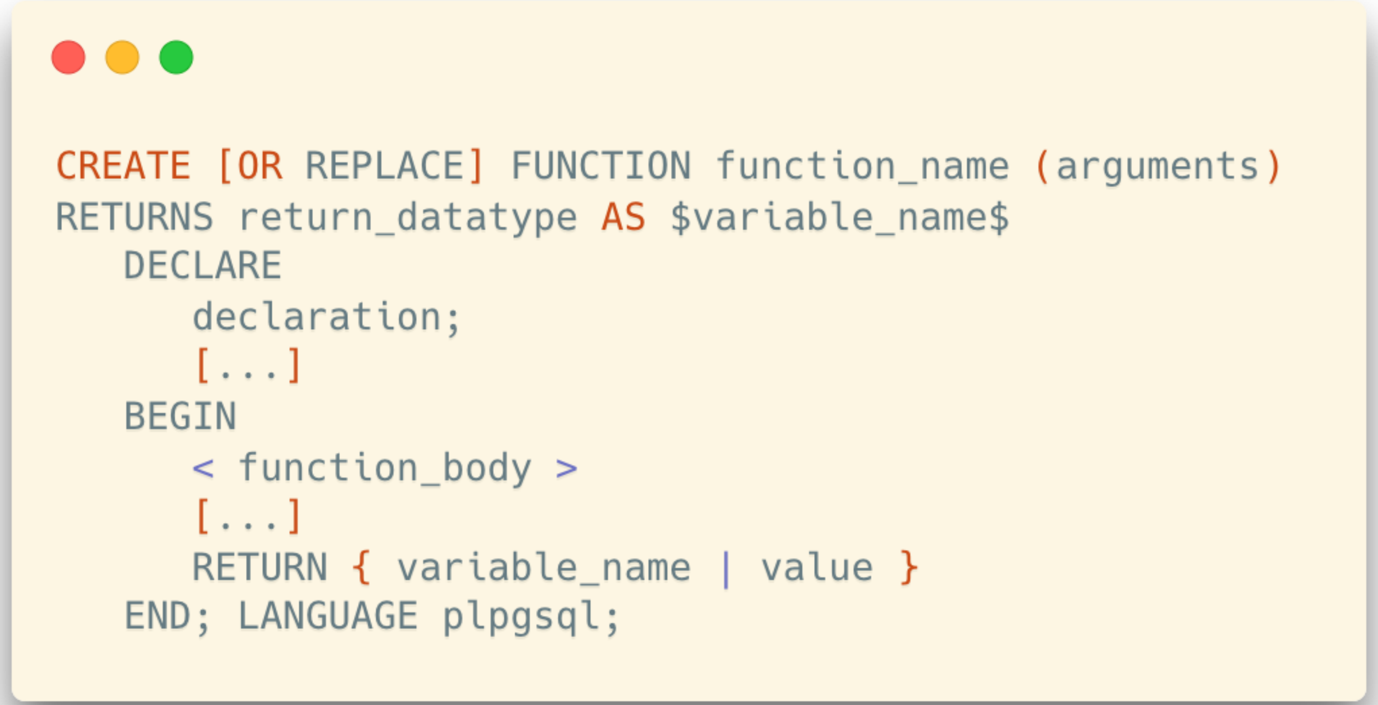
- Copy and paste is not a good habit
  - Whenever you have painfully written something as complicated, which is pretty generic, you'd rather not copy and paste the code every time you need it



```
case
    when first_name is null then ''
    else first_name || ' '
end || case position('(' in surname)
    when 0 then surname
    else trim(')' from substr(surname, position('(' in surname) + 1))
        || ' '
        || trim(substr(surname, 1, position('(' in surname) - 1))
end
```

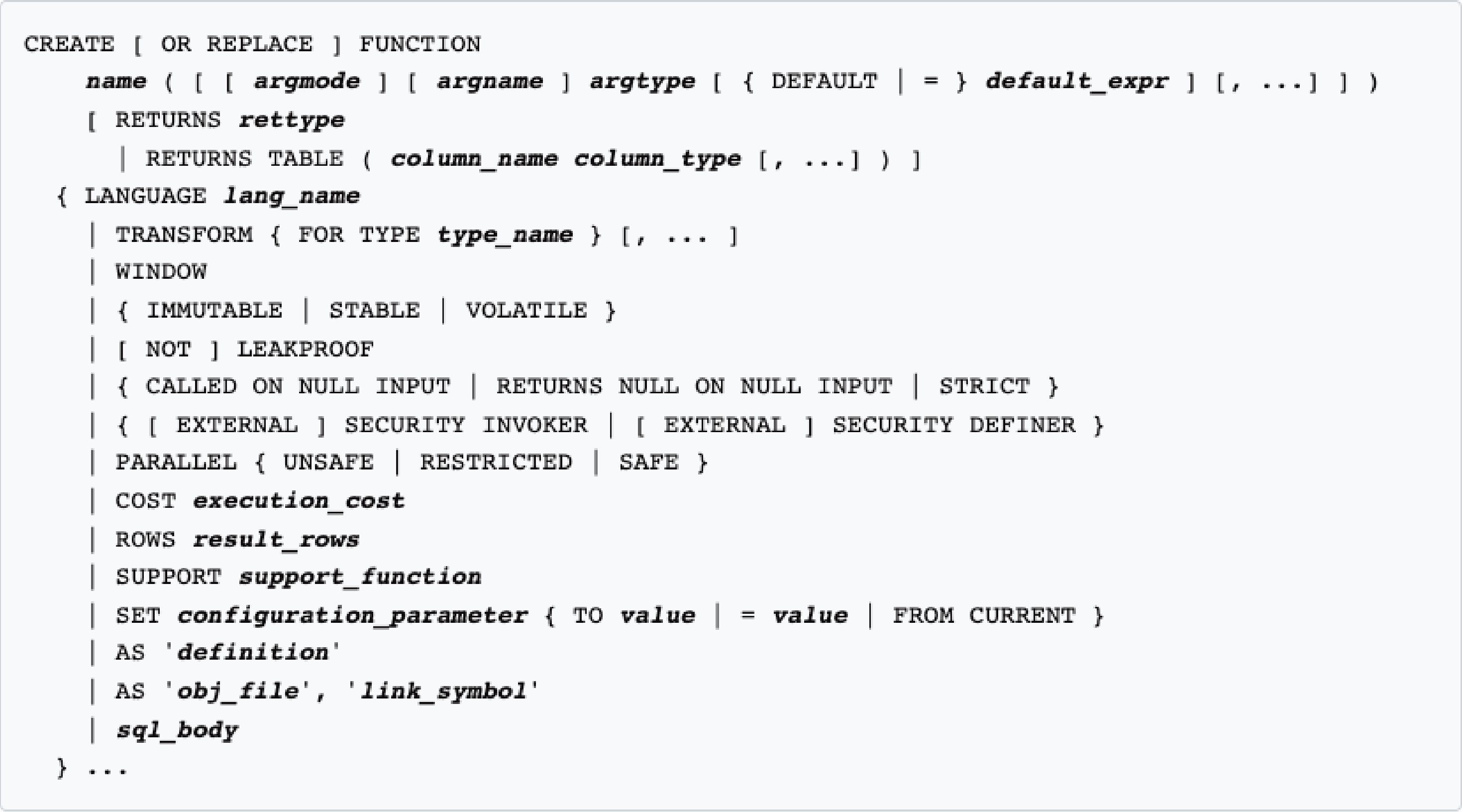
Function in (Postgre)SQL

- Stored for reuse
  - In PostgreSQL, we can store the expression and reuse it in another context
- Self-defined Function
  - `create function`



```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $variable_name$
   DECLARE
      declaration;
      [...]
   BEGIN
      < function_body >
      [...]
      RETURN { variable_name | value }
   END; LANGUAGE plpgsql;
```

...or, a simpler version

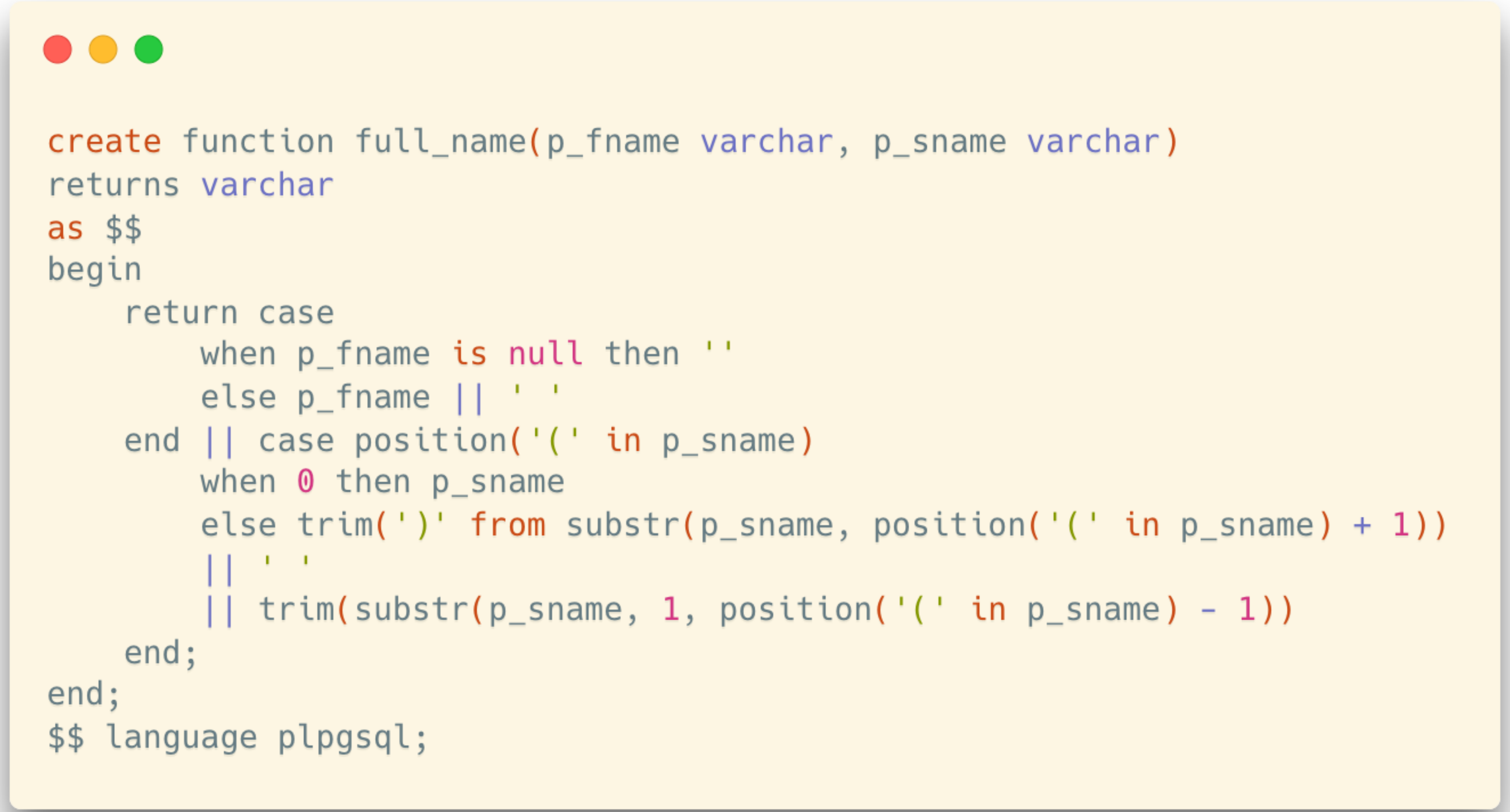


```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } ...
```

https://www.postgresql.org/docs/current/sql-createfunction.html

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?



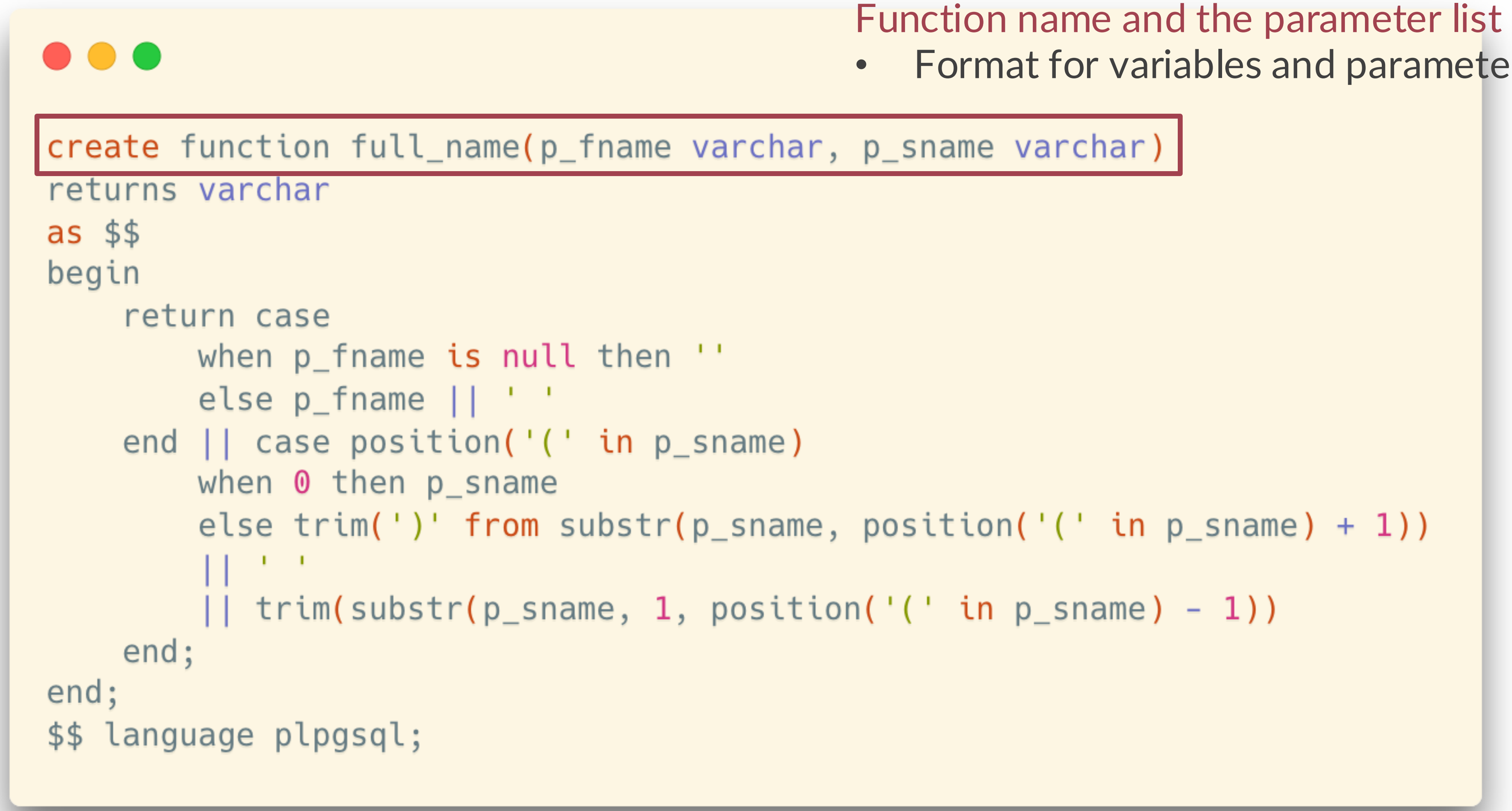
```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

Function name and the parameter list

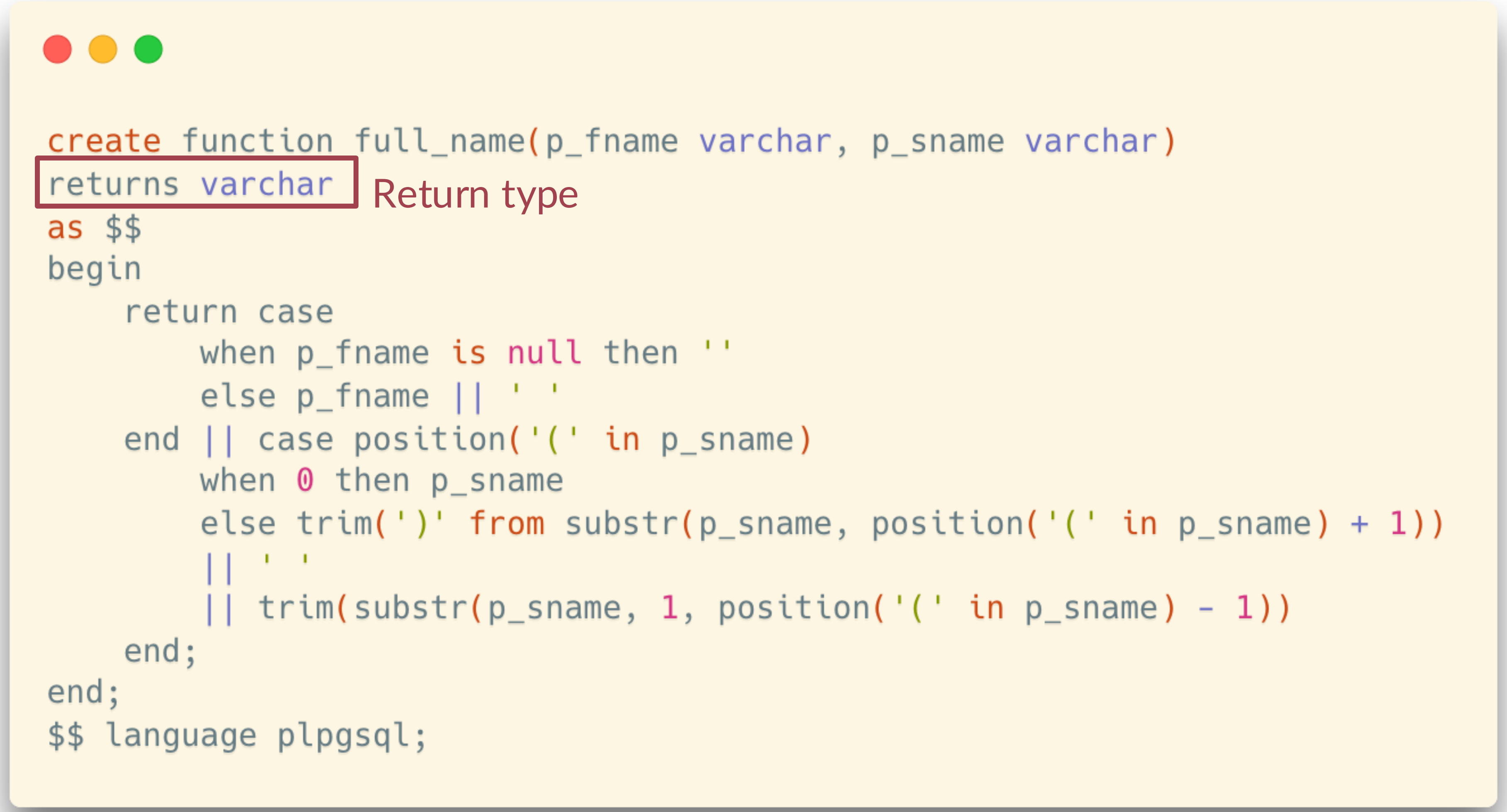
- Format for variables and parameters: [name] [type]



```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

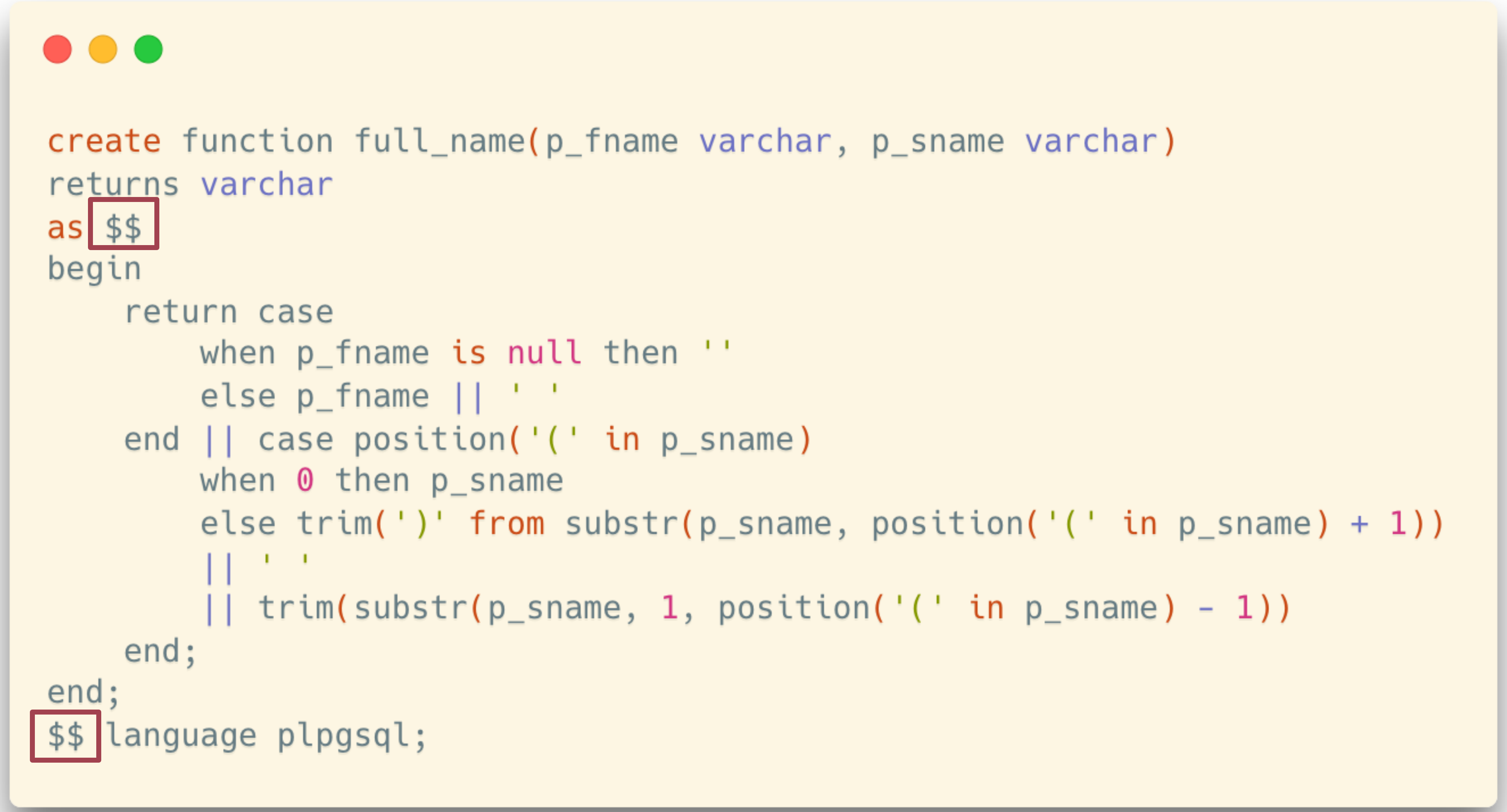


```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Return type

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

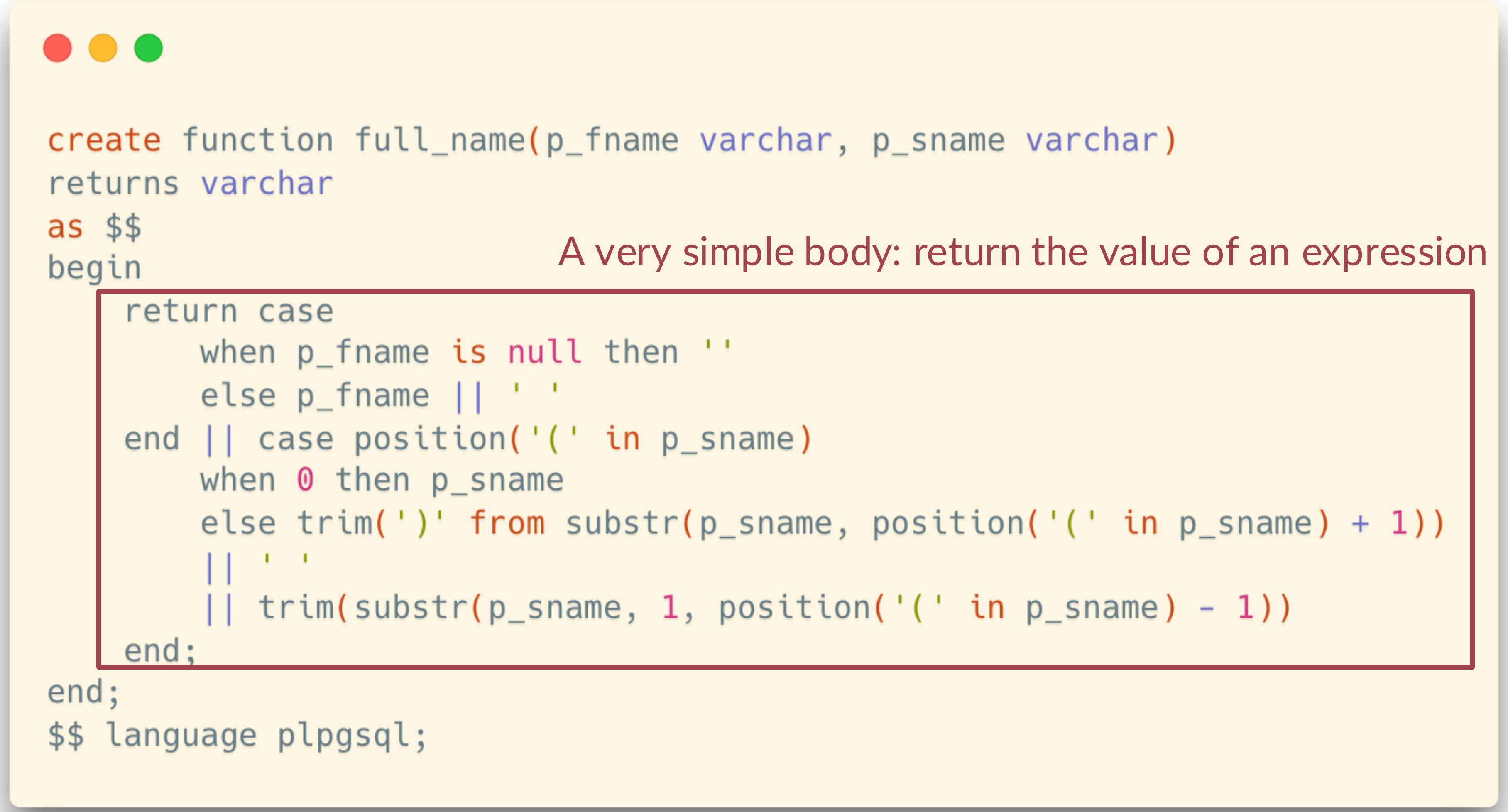


```sql
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Body

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

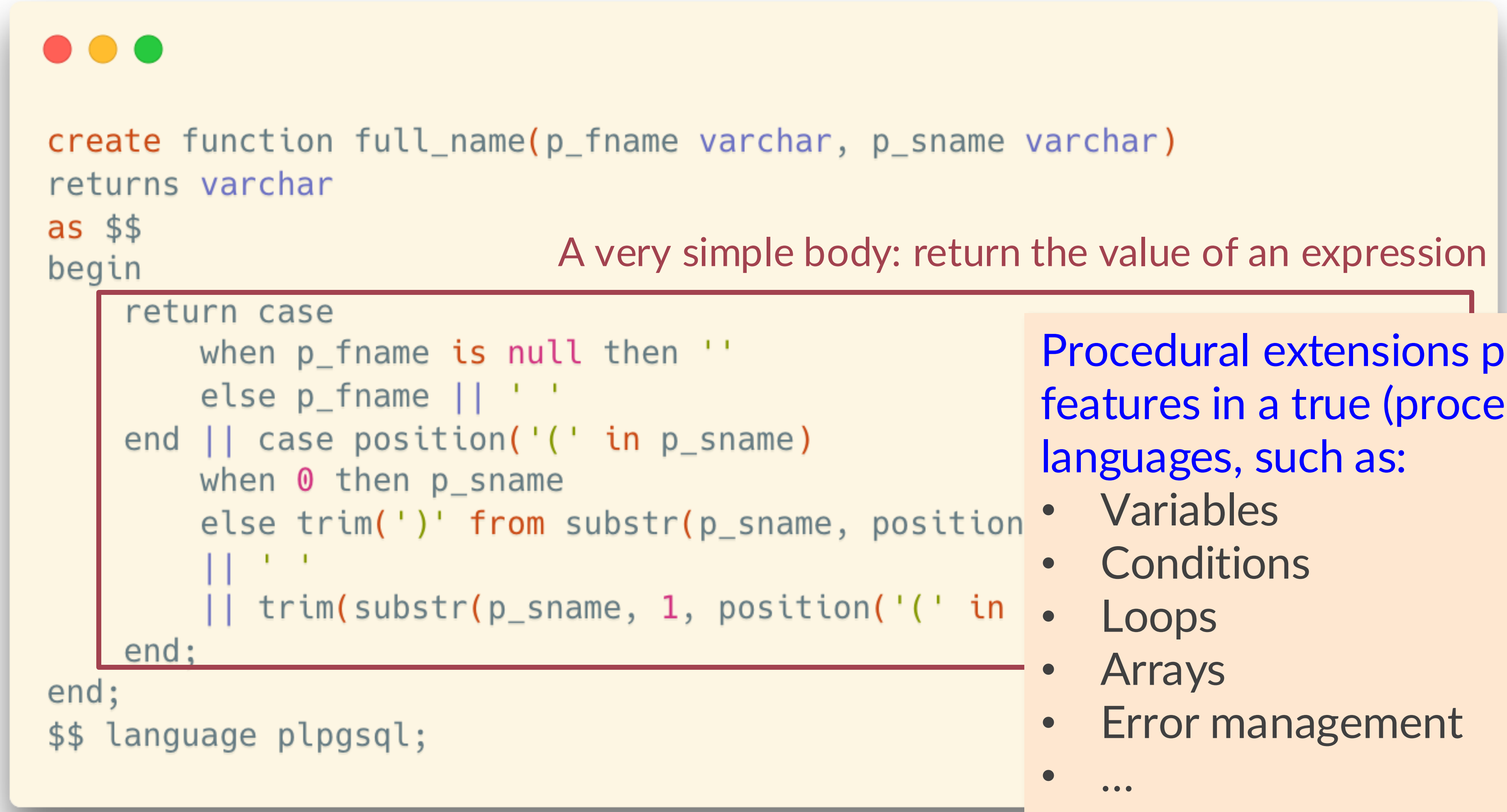


```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

A very simple body: return the value of an expression

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?



```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position
        || ' '
        || trim(substr(p_sname, 1, position('(' in
    end;
end;
$$ language plpgsql;
```

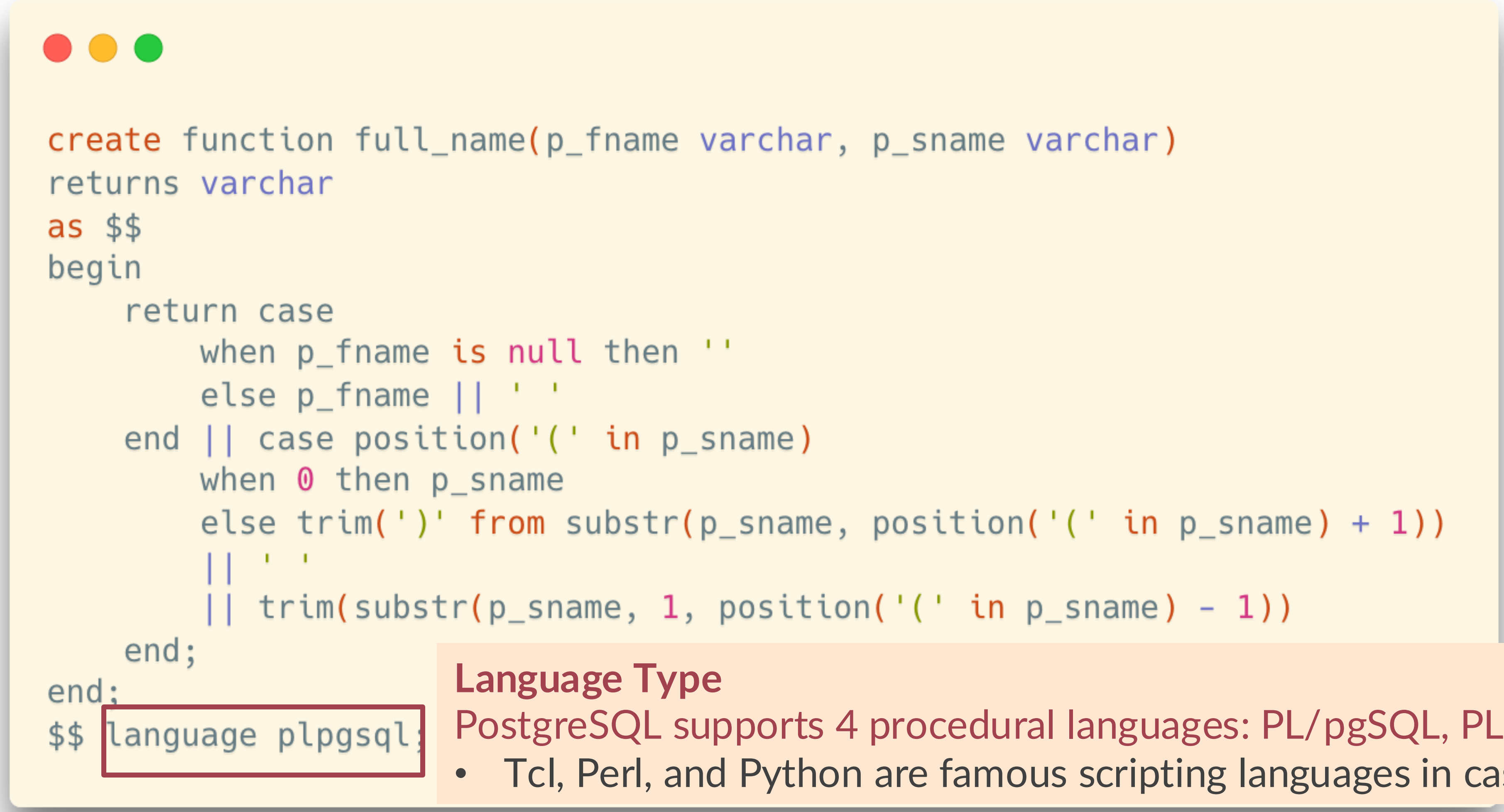
A very simple body: return the value of an expression

Procedural extensions provide all the features in a true (procedural) programming languages, such as:

- Variables
- Conditions
- Loops
- Arrays
- Error management
- …

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?



```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

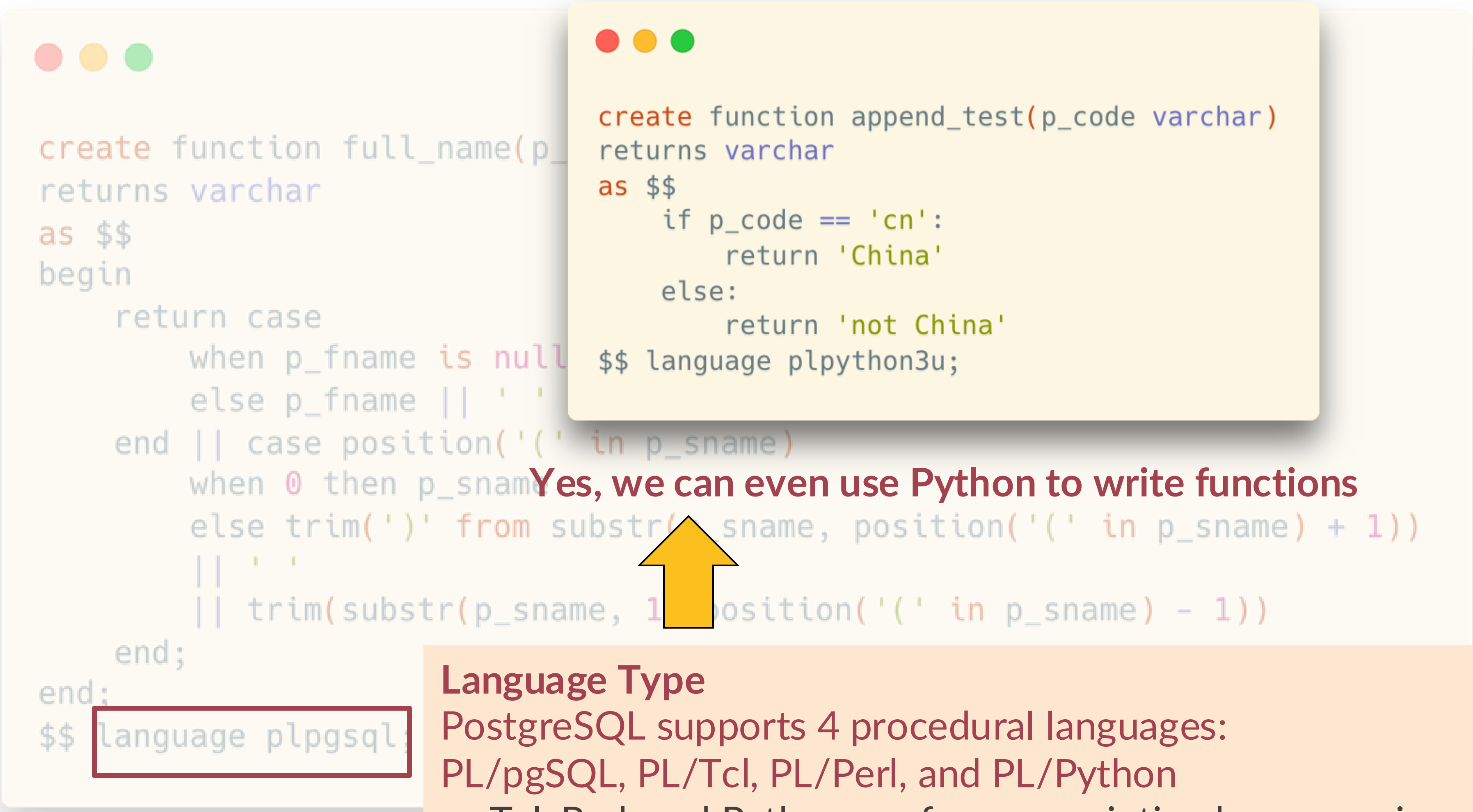
Language Type

PostgreSQL supports 4 procedural languages: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python

- Tcl, Perl, and Python are famous scripting languages in case you don't know

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?



```
create function full_name(p_
returns varchar
as $$
begin
    return case
        when p_fname is null
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_snam
        else trim(')' from substr( sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1 osition('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

```
create function append_test(p_code varchar)
returns varchar
as $$
    if p_code == 'cn':
        return 'China'
    else:
        return 'not China'
$$ language plpython3u;
```

Yes, we can even use Python to write functions

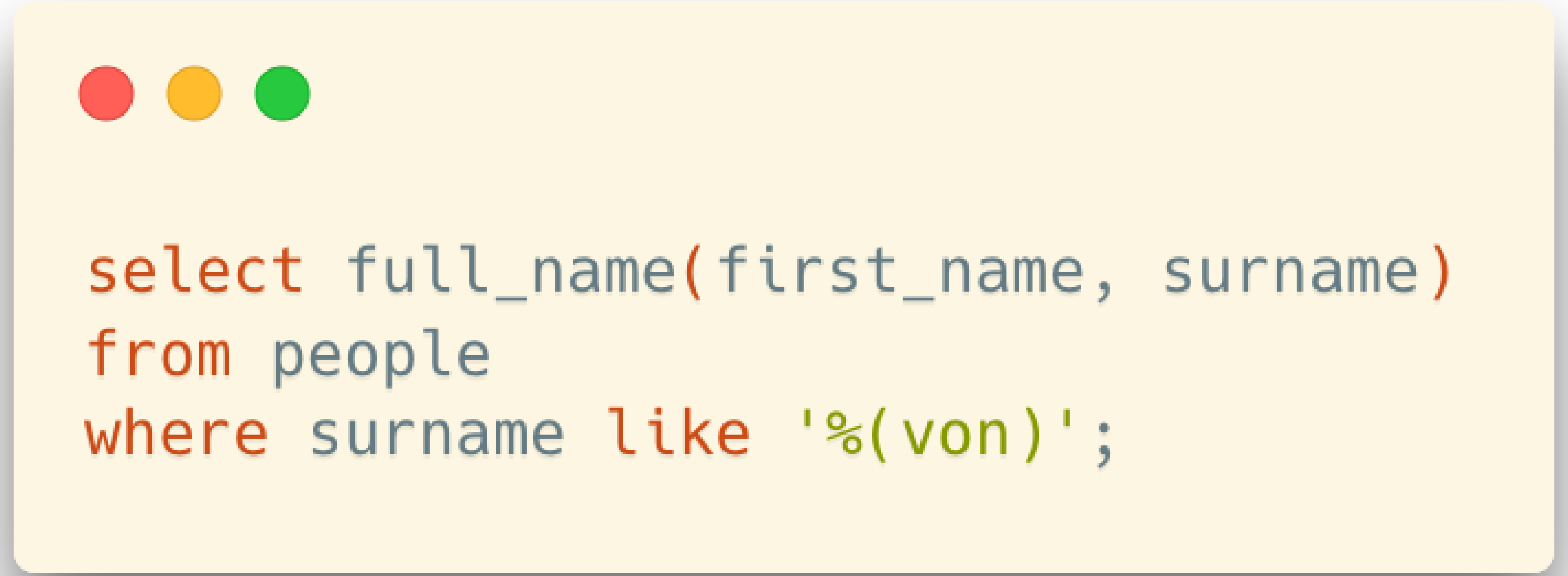
Language Type

PostgreSQL supports 4 procedural languages:
PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python

- Tcl, Perl, and Python are famous scripting languages in case you don’t know

Function in (Postgre)SQL

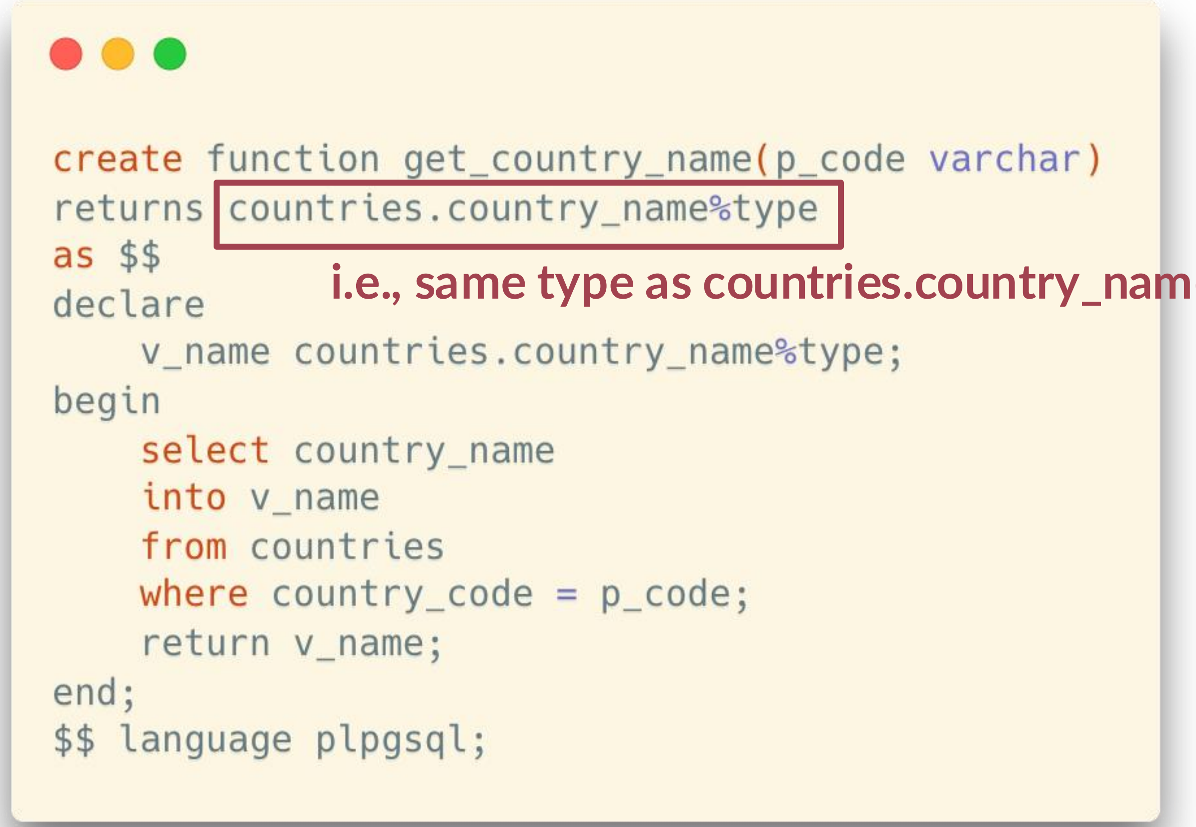
- Once your function is created, you can use it as if it were any built-in function.



```sql
select full_name(first_name, surname)
from people
where surname like '%(von)';
```

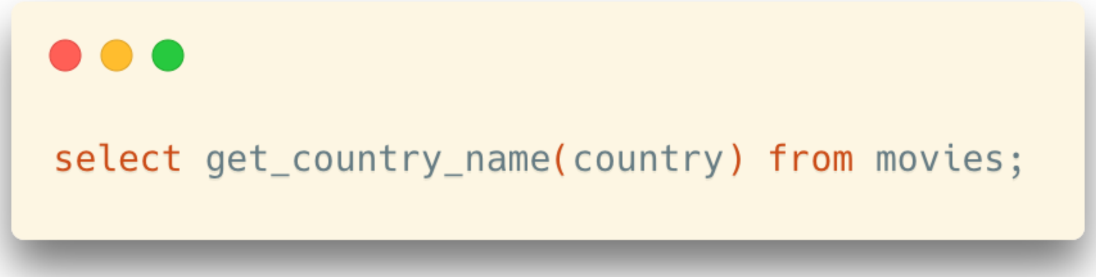
Function in (Postgre)SQL

- We can run `select` queries in functions
  - Example: design a function “get_country_name” to transform the country codes into country names based on the `countries` table



```
create function get_country_name(p_code varchar)
returns countries.country_name%type
as $$
declare
    v_name countries.country_name%type;
begin
    select country_name
    into v_name
    from countries
    where country_code = p_code;
    return v_name;
end;
$$ language plpgsql;
```

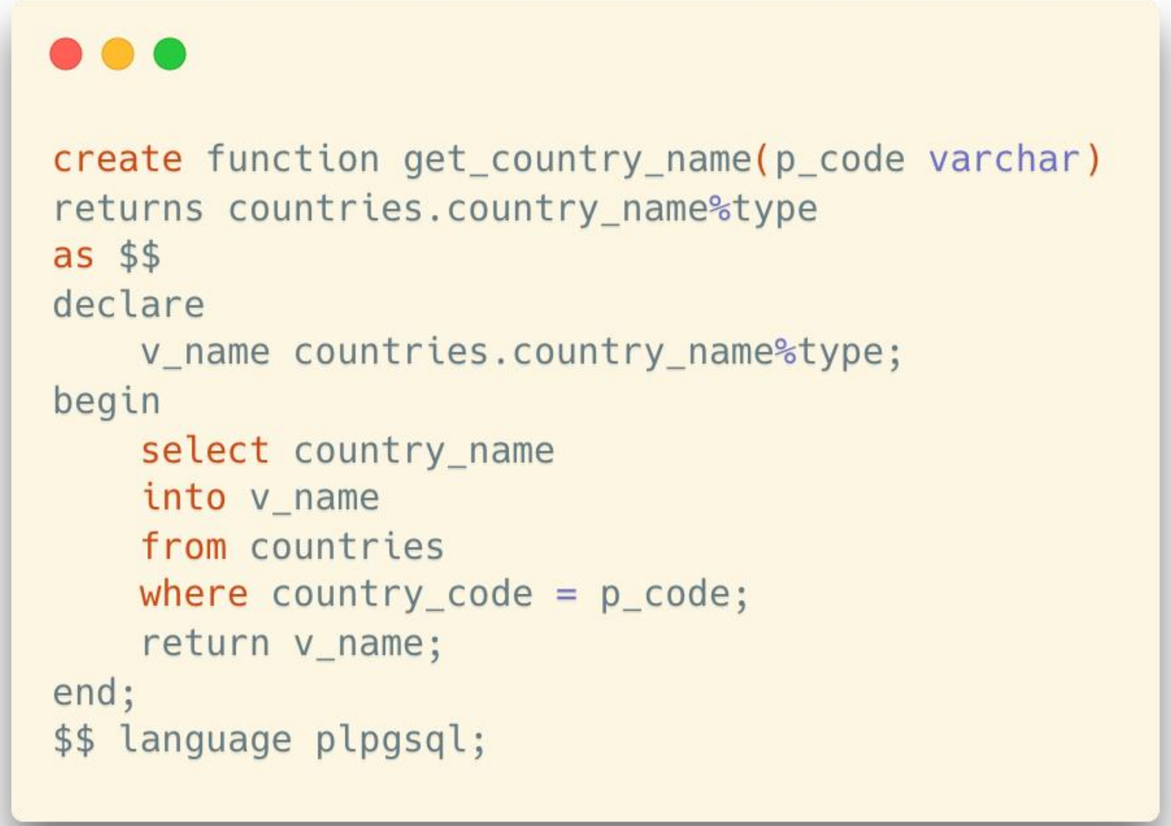
i.e., same type as countries.country_name



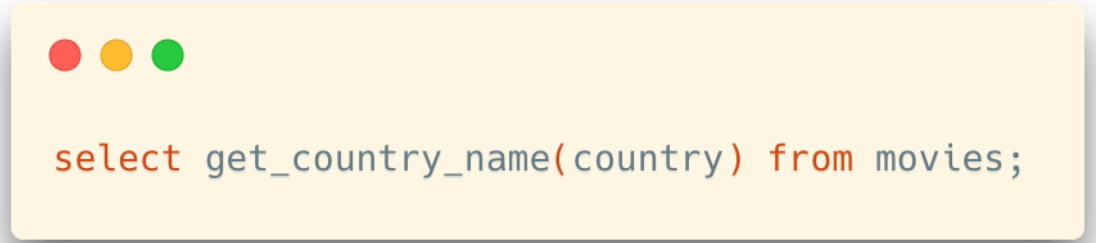
```
select get_country_name(country) from movies;
```

Function in (Postgre)SQL

- We can run `select` queries in functions
  - Example: design a function “get_country_name” to transform the country codes into country names based on the `countries` table

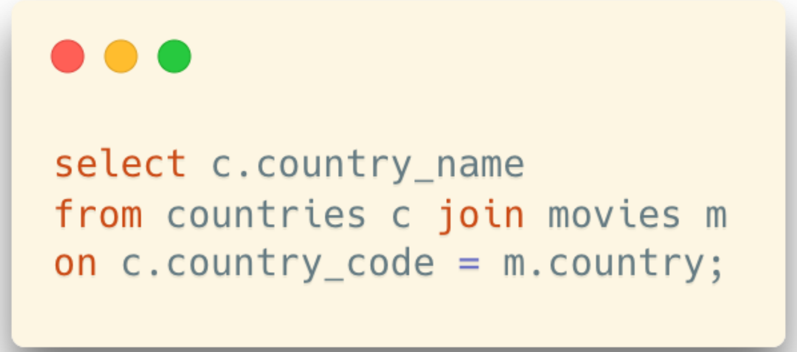


```
create function get_country_name(p_code varchar)
returns countries.country_name%type
as $$
declare
    v_name countries.country_name%type;
begin
    select country_name
    into v_name
    from countries
    where country_code = p_code;
    return v_name;
end;
$$ language plpgsql;
```



```
select get_country_name(country) from movies;
```

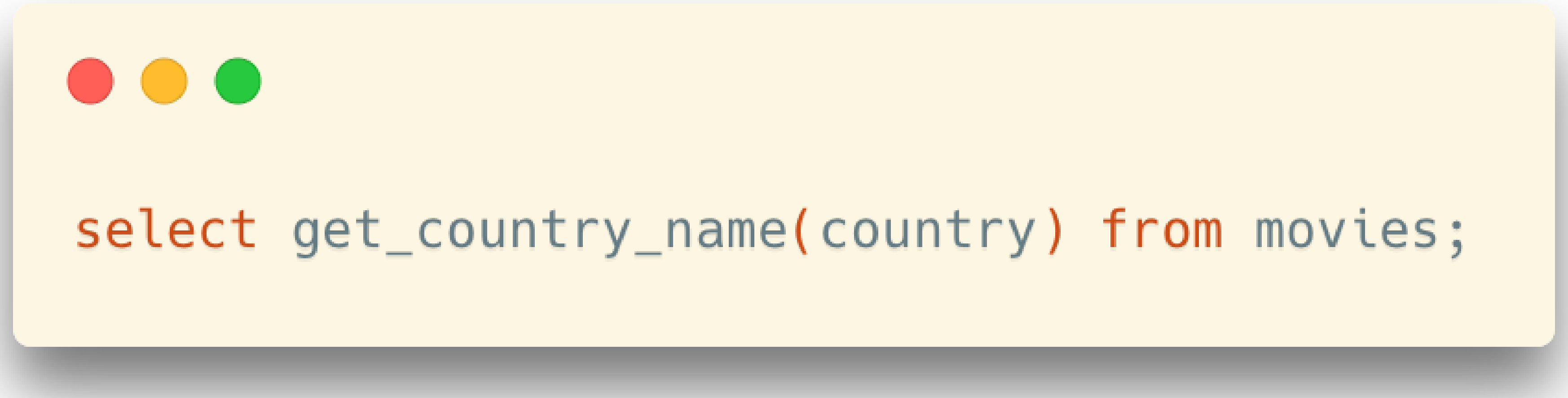
… seems to be an easy way to get rid of `join` operations?



```
select c.country_name
from countries c join movies m
on c.country_code = m.country;
```

Function in (Postgre)SQL

- A “look-up function” forces a “one row at a time” join which in most cases will be costly



```
select get_country_name(country) from movies;
```

For each row in movies, the `select` query in `get_country_name()` is executed once

More to Read

- We may not cover all the details in functions in the theoretical session, so here are some more materials on procedural programming in PostgreSQL:
  - Lab tutorial on Functions
    - Please read it before your next lab sessions
  - Chapter 5.2 “Functions and Procedures,” Database System Concepts (7th Edition)
  - Chapter 43 “PL/pgSQL,” PostgreSQL Documentation
    - https://www.postgresql.org/docs/current/plpgsql.html

Procedures

Functions vs. Procedures

- Generally,
  - “Function” comes from mathematics
    - … which calculates a value with a given input (or to say, map a value to another)
    - Thus, functions always have a return value
  - ”Procedure” comes from programming
    - … which is used to describe a set of instructions that will be executed in order
    - … and does NOT (necessarily) have a return value
- However,
  - Sometimes, the two terms are interchangeably (used for representing the same thing)
    - e.g., procedures are called functions as well
  - Be careful when seeing both terms
    - Always identify the exact meaning of each term and see whether they have different or the same meaning(s)

Functions and Procedures in (Postgre)SQL

- It follows the general definition of functions and procedures
  - Function: return a value
  - Procedure: return NO value
- However,
  - For some historical reasons, PostgreSQL actually has no implementation specifically for procedures
    - It shares the same mechanism with functions
    - Treats procedures as void functions
  - * But for some other database systems, there are separate implementations for functions and procedures

When to Use Procedures

- For business logics
  - One requirement may need a series of SQL querys and statements
  - Transactions may be used
  - Example: Insert a new movie into the databases
    - `movies` table
      - Basic information for the movie
    - `countries` table
      - Transformation between country names and codes
    - `people` table
      - new actors / directors
    - `credits` table
      - new credit information
    - Problem: Update all the tables? Input validation? Code reuse? Security?

When to Use Procedures

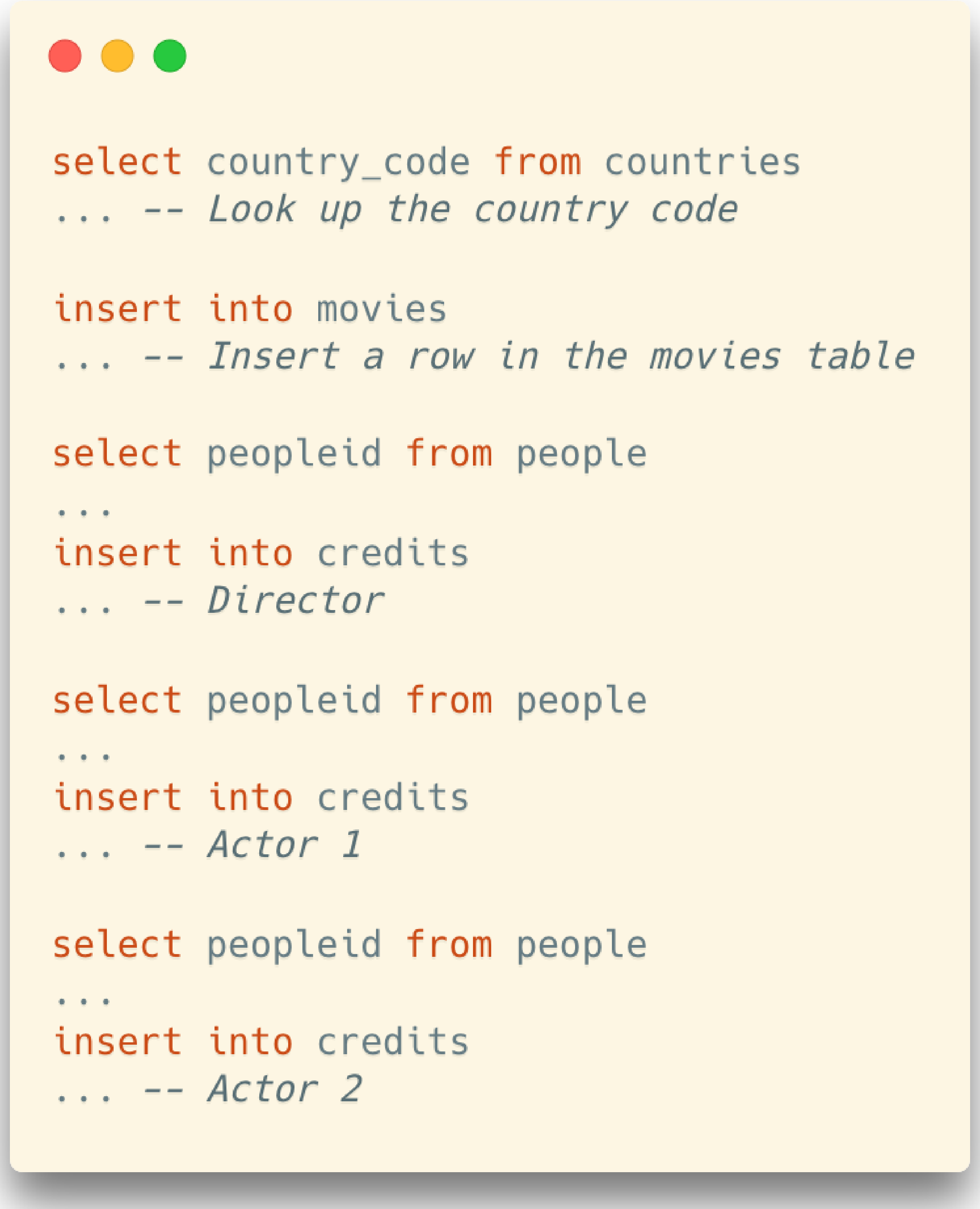
- To add a movie:
  - We may have a series queries to execute when inserting only one movie
  - How about one call for all the processes?
    - Benefit 1: Network overhead
      - When running multiple queries, you are going to waste time chatting over the network with the remote server
    - Benefit 2: Security
      - Prevent users from modifying data otherwise than by calling carefully written and well tested procedures
      - Ensure that users can only modify data via carefully written and well tested procedures

Example: Adding a New Movie

- The information provided for a new movie:
  - Title
  - Year
  - Country Name
    - Note: Name, not code. Country codes are not user-friendly
      - E.g. Which country does “at” represent? ”al”? “ma”? “li”?
  - Director
  - Actor 1
  - Actor 2
    - Let’s assume that only one director and at most two actors are allowed
    - It will be more difficult when the number of people are flexible

A Typical Process

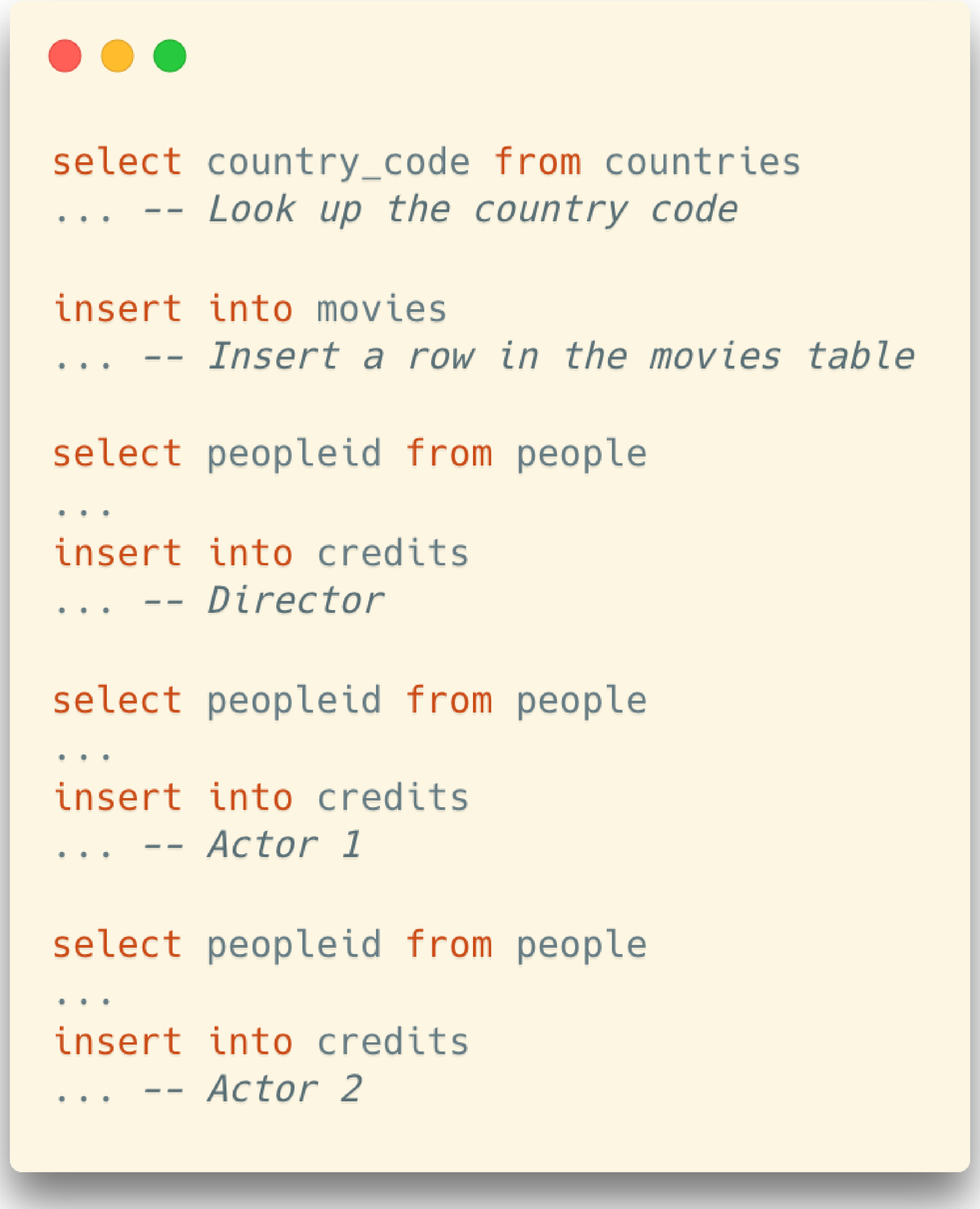
- Insert and check the values, constraints, existence, duplicates, etc
  - A series of inter-related statements



```
select country_code from countries
... -- Look up the country code

insert into movies
... -- Insert a row in the movies table

select peopleid from people
...
insert into credits
... -- Director

select peopleid from people
...
insert into credits
... -- Actor 1

select peopleid from people
...
insert into credits
... -- Actor 2
```

A Typical Process

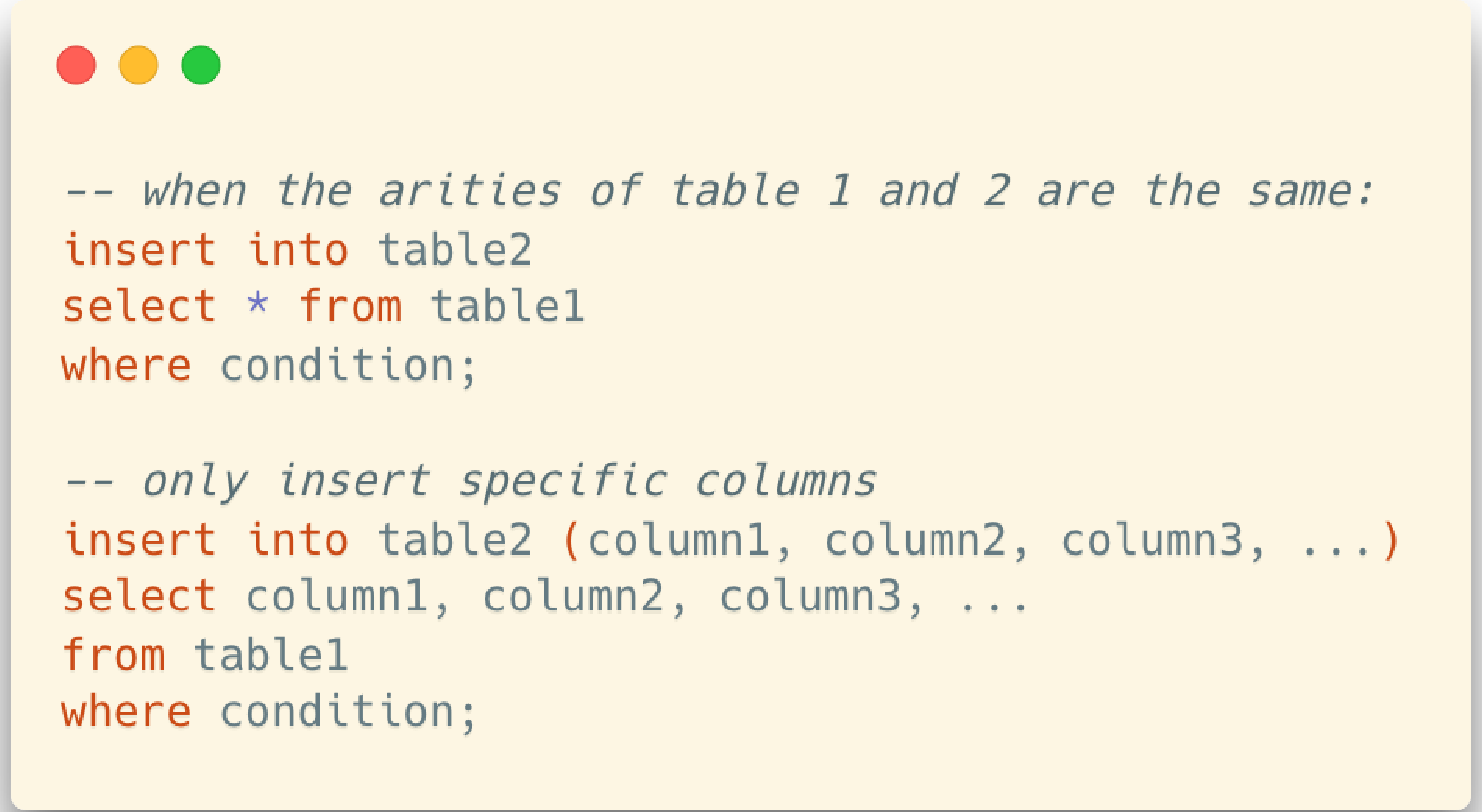
- How can we pack them into a single execution unit?
  - Minimize communication between client program and database server
    - Client program = DataGrip, `psql`, …



```
select country_code from countries
... -- Look up the country code

insert into movies
... -- Insert a row in the movies table

select peopleid from people
...
insert into credits
... -- Director

select peopleid from people
...
insert into credits
... -- Actor 1

select peopleid from people
...
insert into credits
... -- Actor 2
```

insert into select

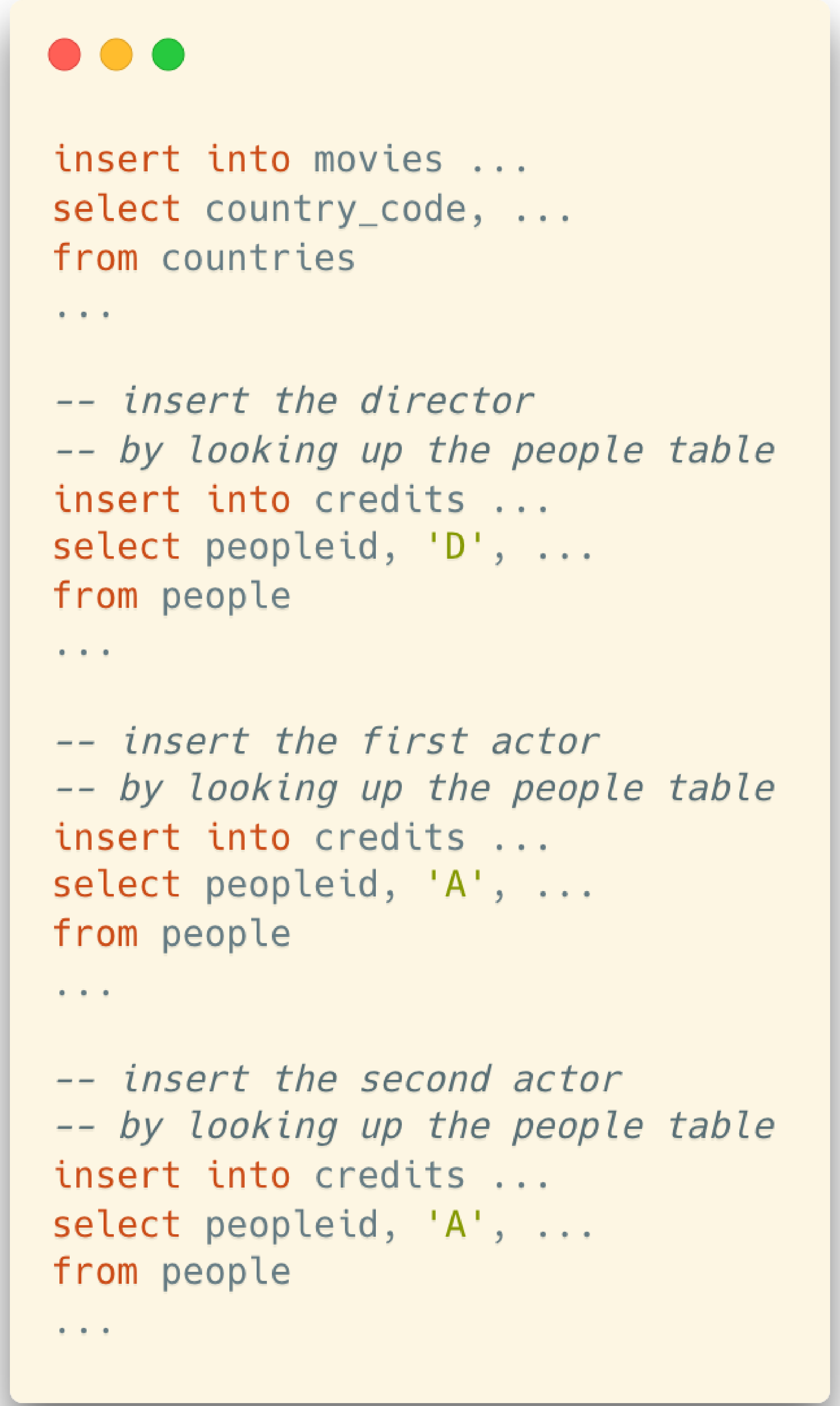
- One thing to optimize: `insert into … select`



```
-- when the arities of table 1 and 2 are the same:
insert into table2
select * from table1
where condition;

-- only insert specific columns
insert into table2 (column1, column2, column3, ...)
select column1, column2, column3, ...
from table1
where condition;
```

Optimize the Insertion

- Use `insert into select`

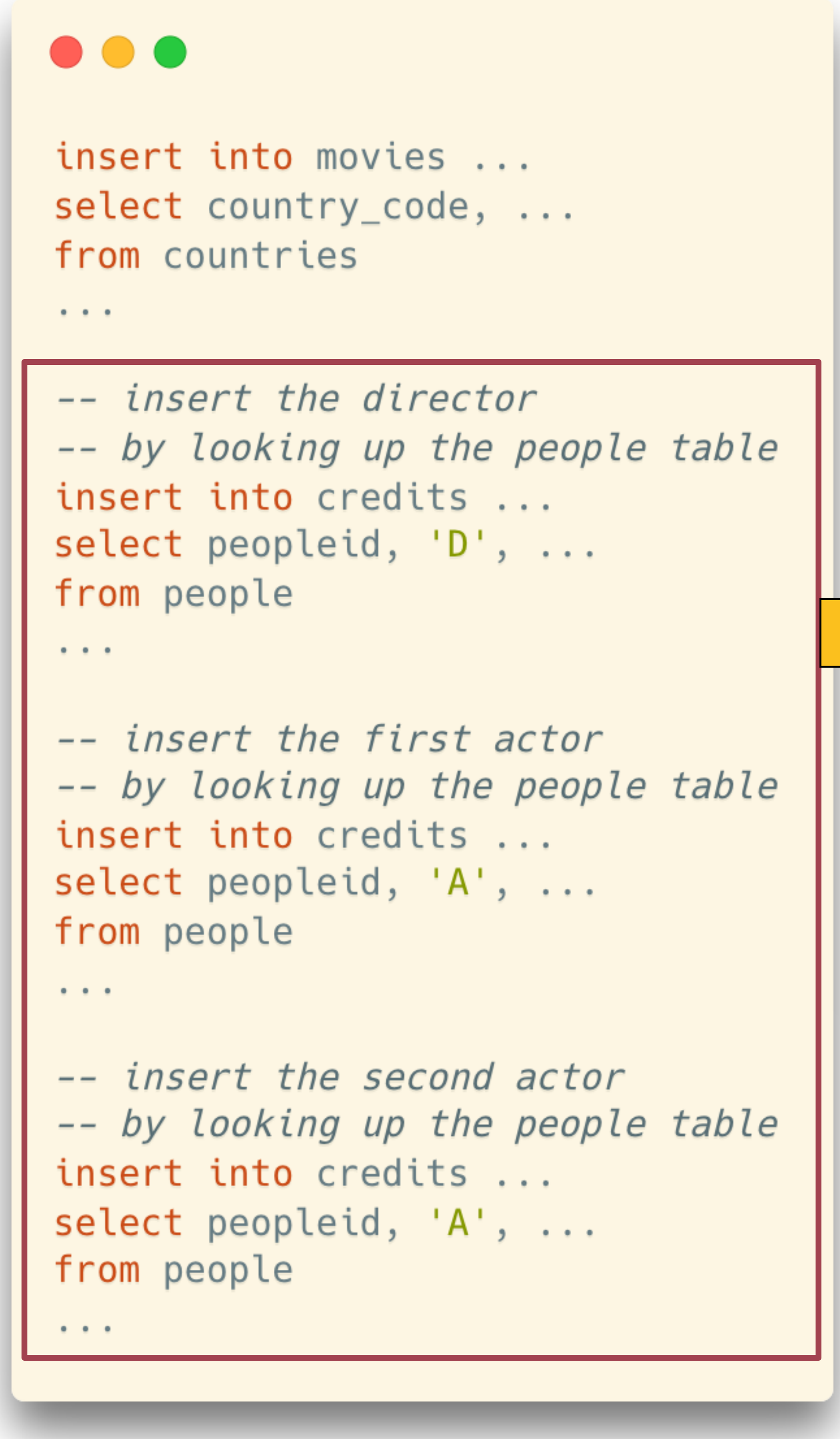


```
insert into movies ...
select country_code, ...
from countries
...

-- insert the director
-- by looking up the people table
insert into credits ...
select peopleid, 'D', ...
from people
...

-- insert the first actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...

-- insert the second actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...
```

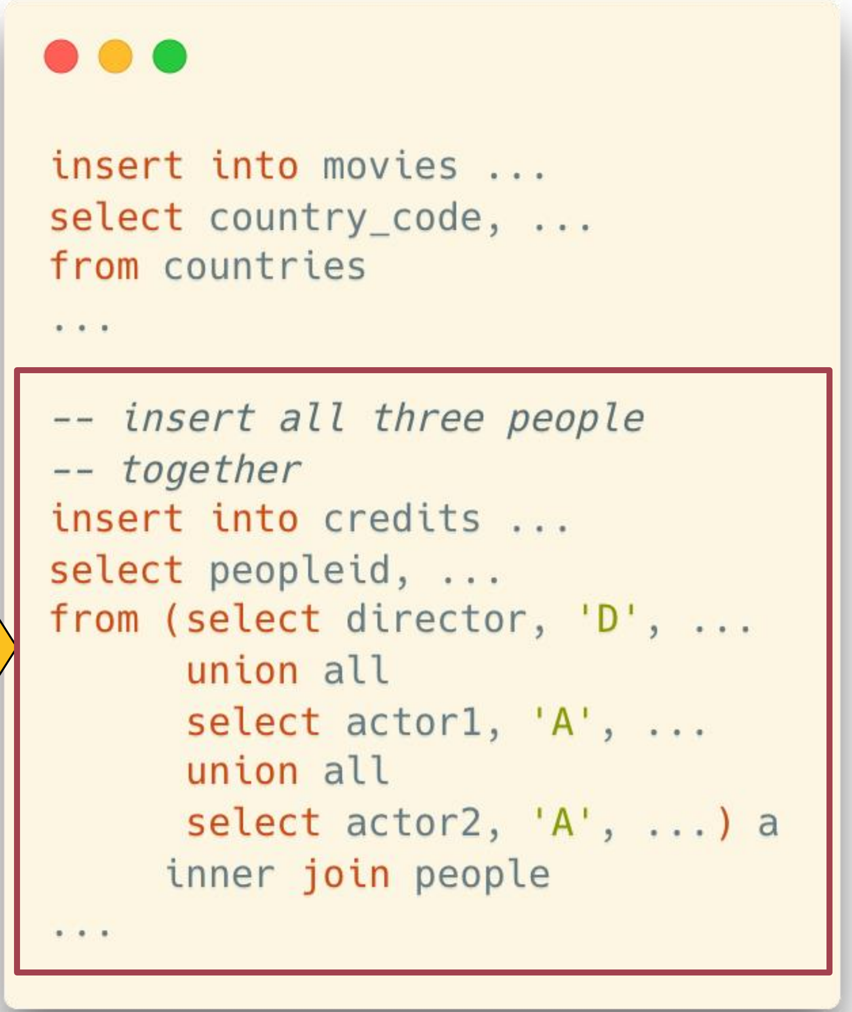
Further Optimize the Insertion

- Use `insert into select`

- Combine the queries of people



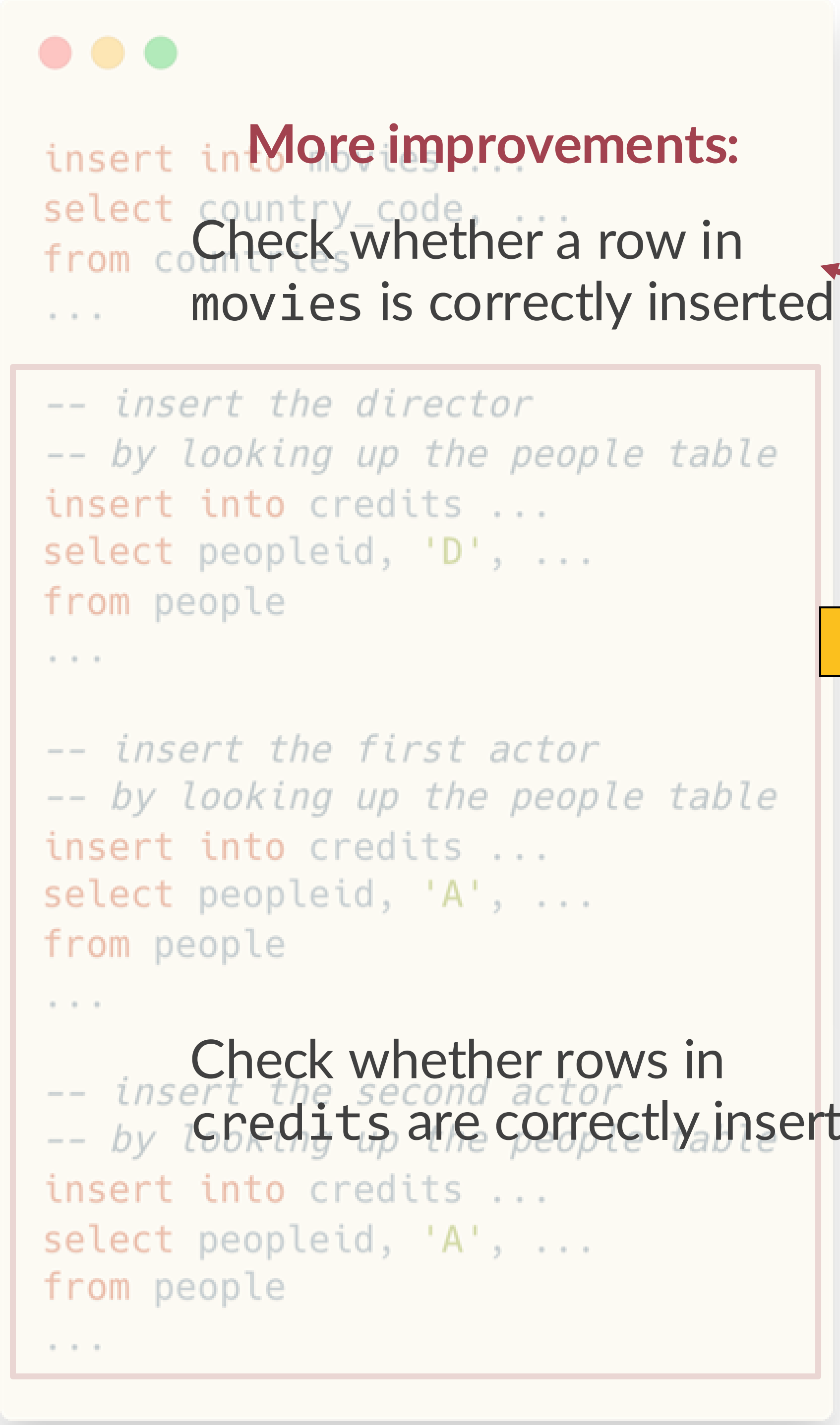
```
insert into movies ...
select country_code, ...
from countries
...
-- insert the director
-- by looking up the people table
insert into credits ...
select peopleid, 'D', ...
from people
...

-- insert the first actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...

-- insert the second actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...
```



```
insert into movies ...
select country_code, ...
from countries
...
-- insert all three people
-- together
insert into credits ...
select peopleid, ...
from (select director, 'D', ...
      union all
      select actor1, 'A', ...
      union all
      select actor2, 'A', ...) a
     inner join people
...
```

Further Optimize the Insertion

- Use `insert into select`
- Combine the queries of `people`

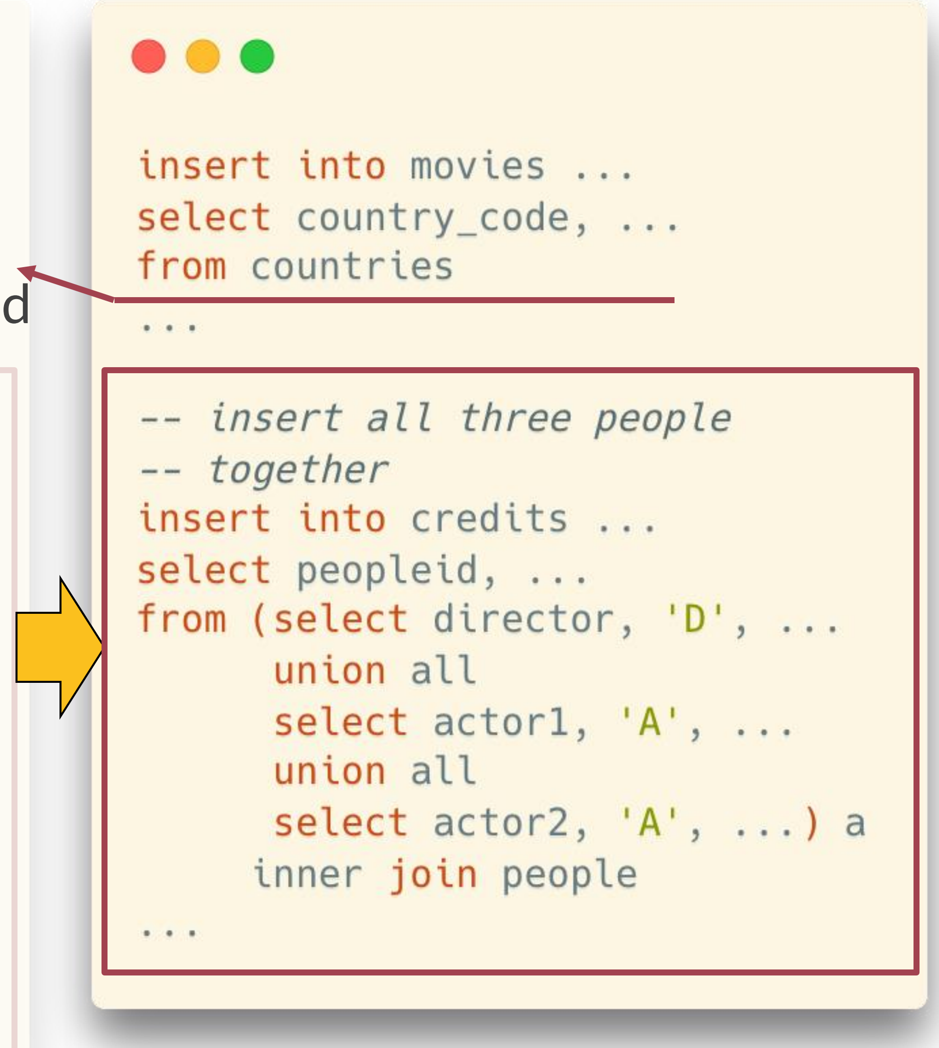


```
insert into movies ...
select country_code, ...
from countries
...

-- insert the director
-- by looking up the people table
insert into credits ...
select peopleid, 'D', ...
from people
...

-- insert the first actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...

-- insert the second actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...
```

More improvements:

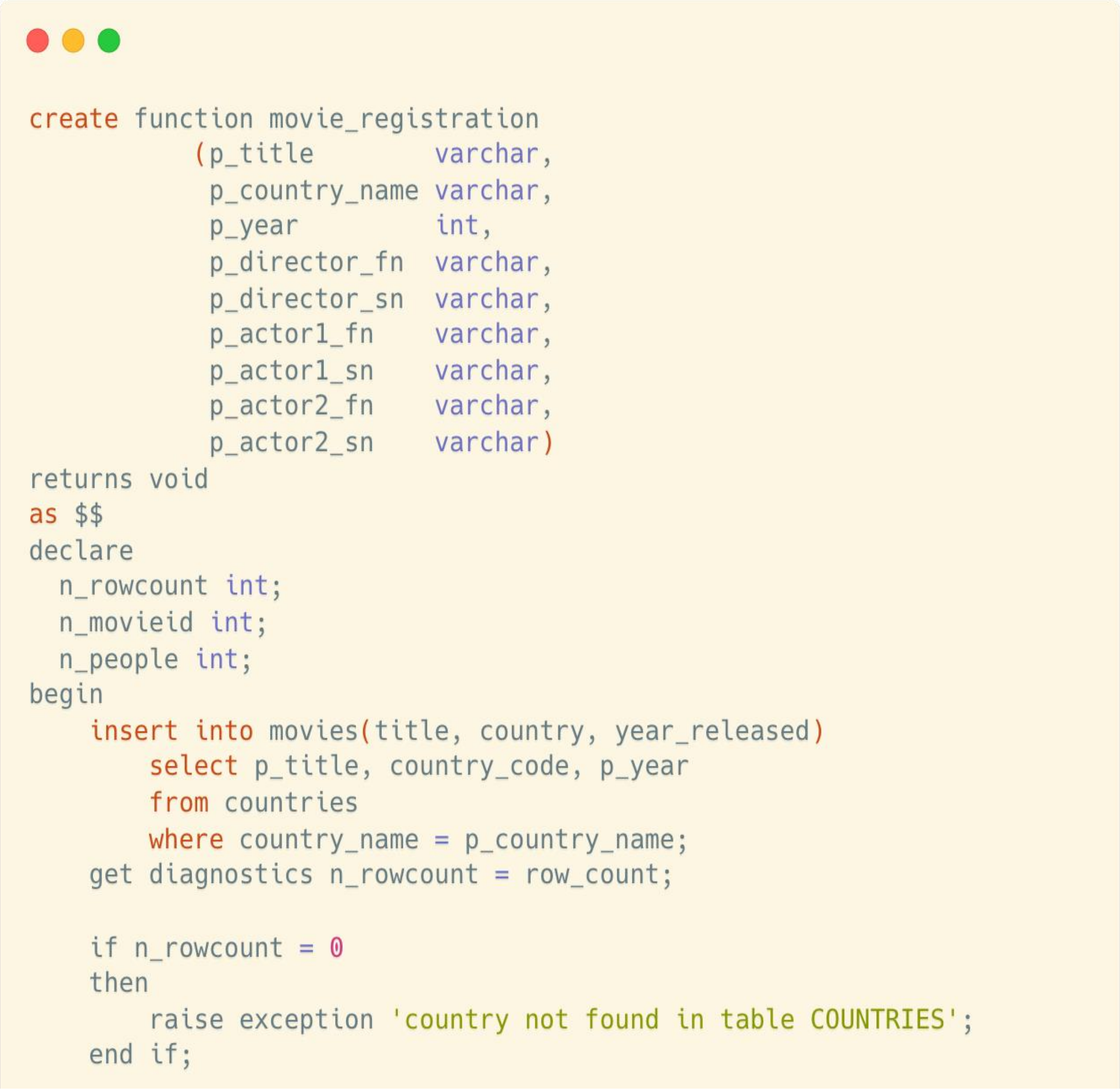
Check whether a row in movies is correctly inserted

Check whether rows in credits are correctly inserted

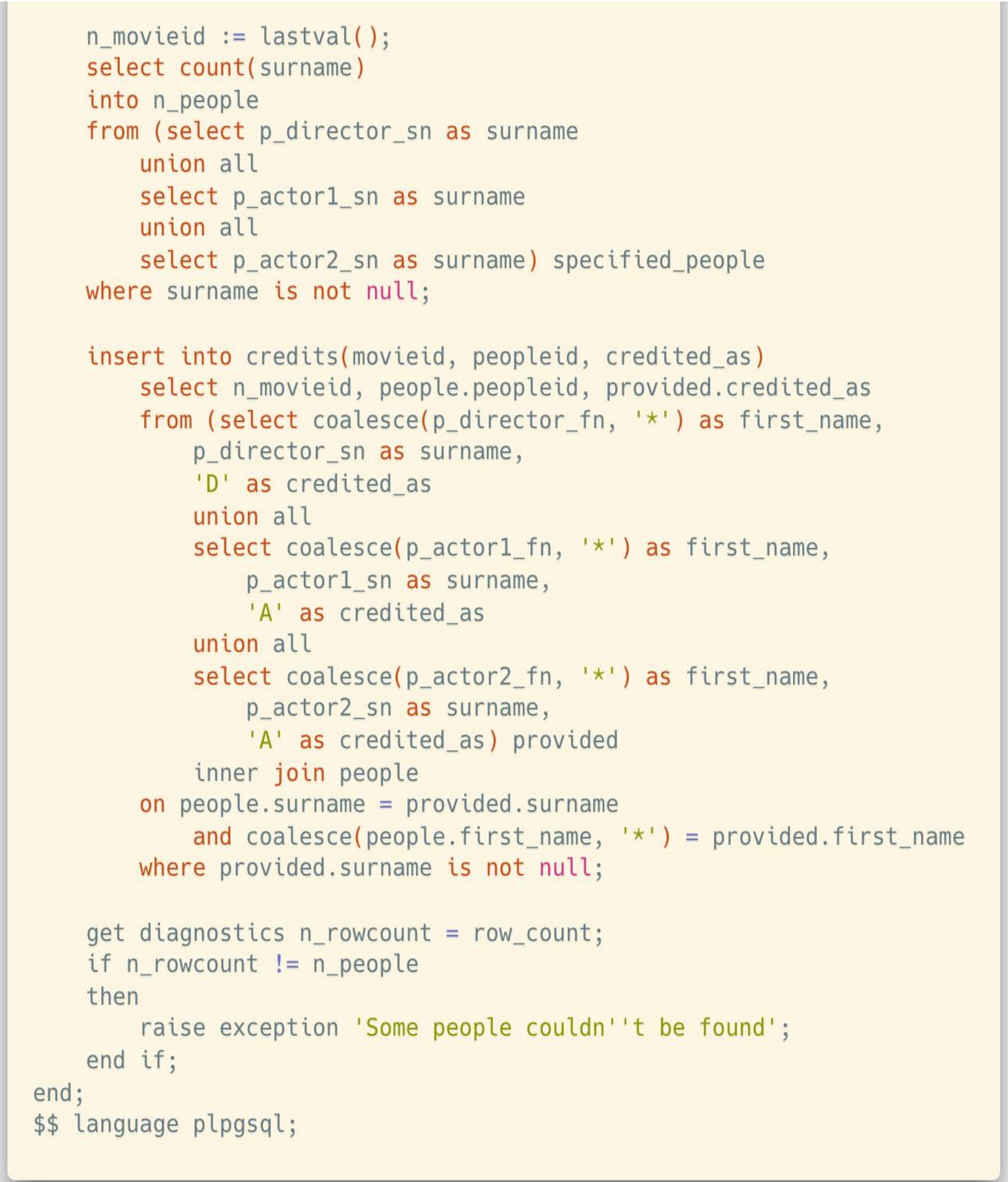


```
insert into movies ...
select country_code, ...
from countries
...

-- insert all three people
-- together
insert into credits ...
select peopleid, ...
from (select director, 'D', ...
      union all
      select actor1, 'A', ...
      union all
      select actor2, 'A', ...) a
     inner join people
...
```

The Procedure

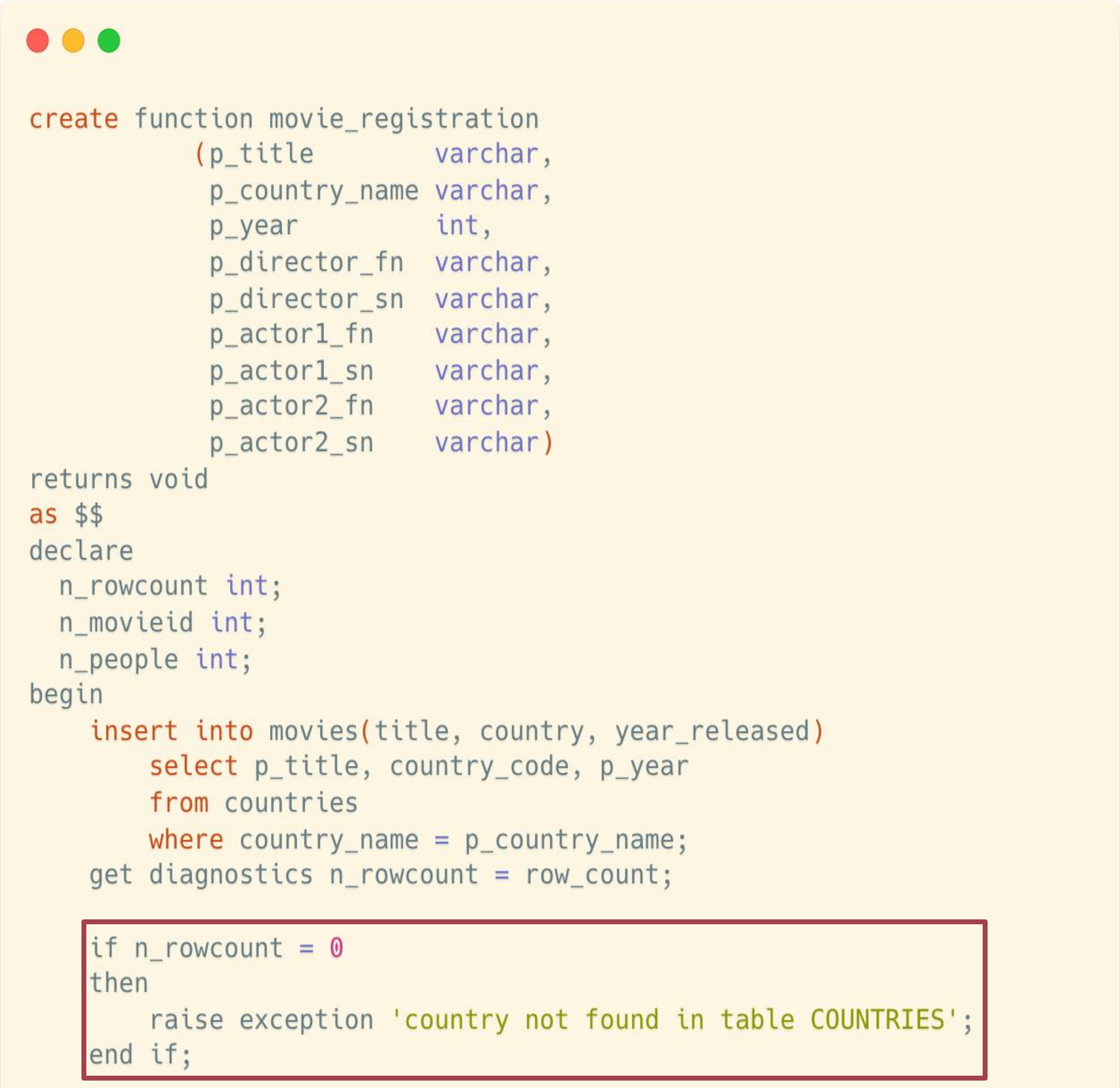


```
create function movie_registration
           (p_title        varchar,
            p_country_name varchar,
            p_year         int,
            p_director_fn  varchar,
            p_director_sn  varchar,
            p_actor1_fn    varchar,
            p_actor1_sn    varchar,
            p_actor2_fn    varchar,
            p_actor2_sn    varchar)
returns void
as $$
declare
  n_rowcount int;
  n_movieid int;
  n_people int;
begin
    insert into movies(title, country, year_released)
        select p_title, country_code, p_year
        from countries
        where country_name = p_country_name;
    get diagnostics n_rowcount = row_count;

    if n_rowcount = 0
    then
        raise exception 'country not found in table COUNTRIES';
    end if;
```



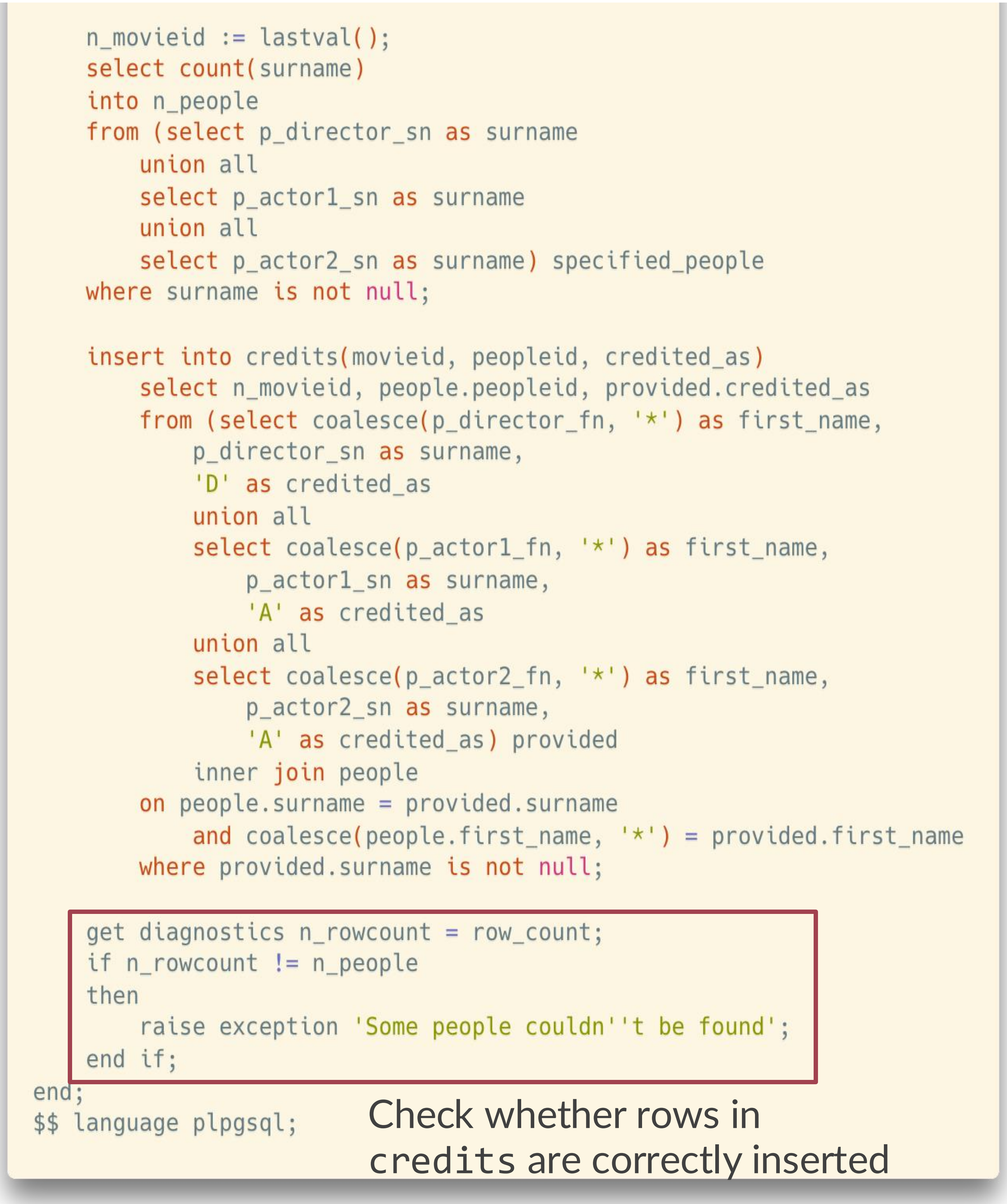
```
    n_movieid := lastval();
    select count(surname)
    into n_people
    from (select p_director_sn as surname
        union all
        select p_actor1_sn as surname
        union all
        select p_actor2_sn as surname) specified_people
    where surname is not null;

    insert into credits(movieid, peopleid, credited_as)
        select n_movieid, people.peopleid, provided.credited_as
        from (select coalesce(p_director_fn, '*') as first_name,
            p_director_sn as surname,
            'D' as credited_as
            union all
            select coalesce(p_actor1_fn, '*') as first_name,
                p_actor1_sn as surname,
                'A' as credited_as
            union all
            select coalesce(p_actor2_fn, '*') as first_name,
                p_actor2_sn as surname,
                'A' as credited_as) provided
            inner join people
        on people.surname = provided.surname
            and coalesce(people.first_name, '*') = provided.first_name
        where provided.surname is not null;

    get diagnostics n_rowcount = row_count;
    if n_rowcount != n_people
    then
        raise exception 'Some people couldn''t be found';
    end if;
end;
$$ language plpgsql;
```

The Procedure



```
create function movie_registration
           (p_title          varchar,
            p_country_name varchar,
            p_year           int,
            p_director_fn   varchar,
            p_director_sn   varchar,
            p_actor1_fn     varchar,
            p_actor1_sn     varchar,
            p_actor2_fn     varchar,
            p_actor2_sn     varchar)
returns void
as $$
declare
  n_rowcount int;
  n_movieid int;
  n_people int;
begin
    insert into movies(title, country, year_released)
        select p_title, country_code, p_year
        from countries
        where country_name = p_country_name;
    get diagnostics n_rowcount = row_count;

    if n_rowcount = 0
    then
        raise exception 'country not found in table COUNTRIES';
    end if;
```

Check whether a row in movies is correctly inserted

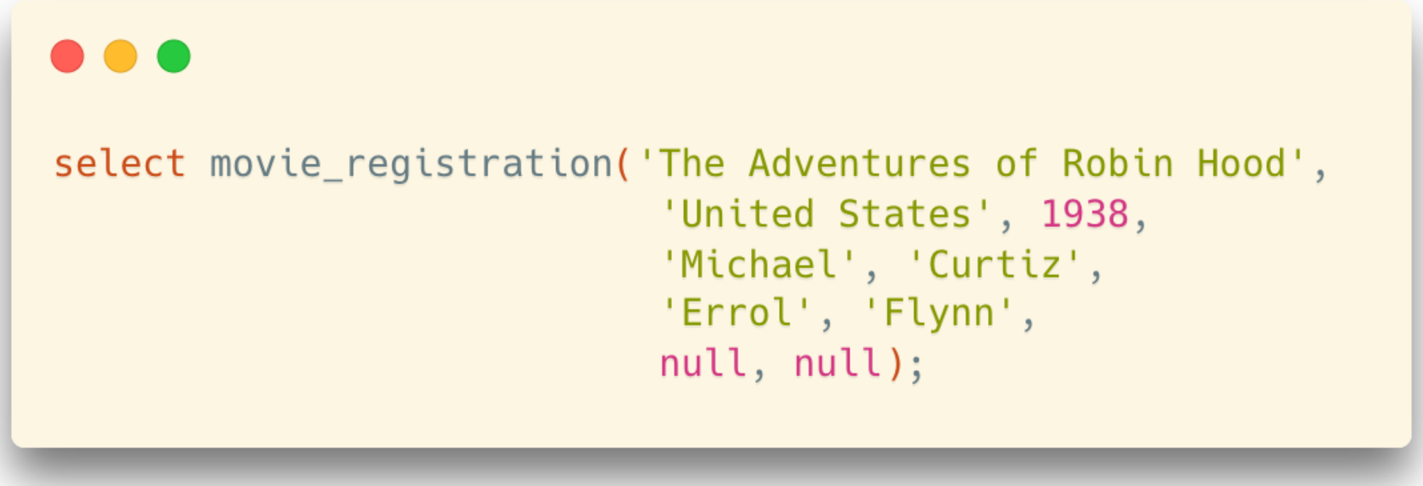


```
     n_movieid := lastval();
     select count(surname)
     into n_people
     from (select p_director_sn as surname
         union all
         select p_actor1_sn as surname
         union all
         select p_actor2_sn as surname) specified_people
     where surname is not null;

     insert into credits(movieid, peopleid, credited_as)
         select n_movieid, people.peopleid, provided.credited_as
         from (select coalesce(p_director_fn, '*') as first_name,
             p_director_sn as surname,
             'D' as credited_as
             union all
             select coalesce(p_actor1_fn, '*') as first_name,
                 p_actor1_sn as surname,
                 'A' as credited_as
             union all
             select coalesce(p_actor2_fn, '*') as first_name,
                 p_actor2_sn as surname,
                 'A' as credited_as) provided
             inner join people
         on people.surname = provided.surname
             and coalesce(people.first_name, '*') = provided.first_name
         where provided.surname is not null;

     get diagnostics n_rowcount = row_count;
     if n_rowcount != n_people
     then
         raise exception 'Some people couldn''t be found';
     end if;
end;
$$ language plpgsql;
```

Check whether rows in credits are correctly inserted

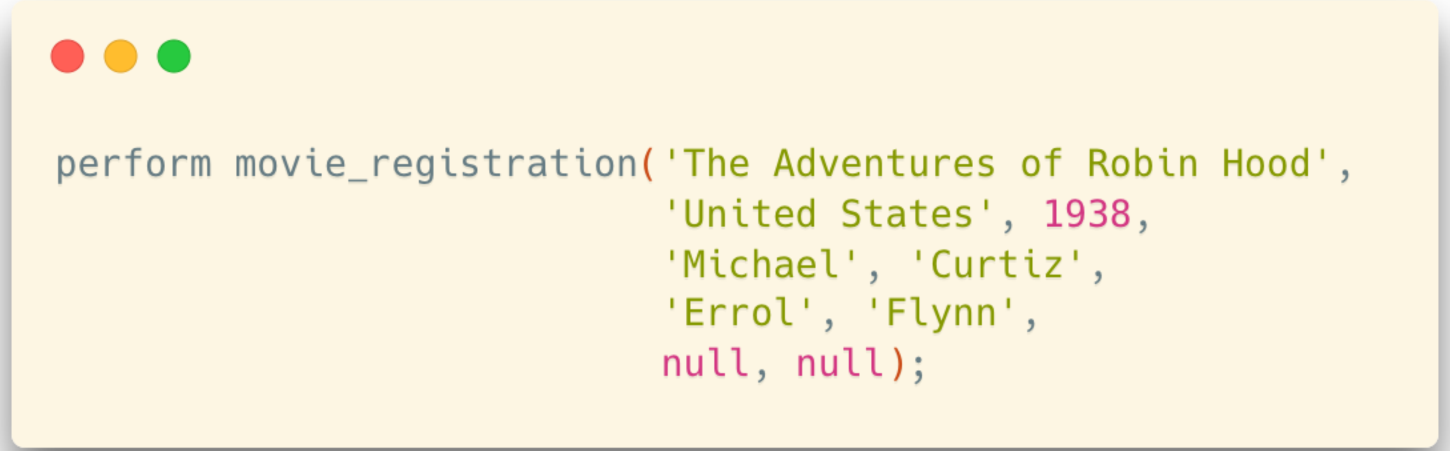
Calling Procedures

- In PostgreSQL
  - We can call the procedure interactively by calling it from a SELECT statement (that will return nothing)



```
select movie_registration('The Adventures of Robin Hood',
                           'United States', 1938,
                           'Michael', 'Curtiz',
                           'Errol', 'Flynn',
                           null, null);
```

- We can also call a procedure from another procedure



```
perform movie_registration('The Adventures of Robin Hood',
                           'United States', 1938,
                           'Michael', 'Curtiz',
                           'Errol', 'Flynn',
                           null, null);
```