PostgreSQL

GnuGroup - ILGLabs
Www.gnugroup.org

PostgreSQL Stored Procedures

In these slides we will learn about PostgreSQL stored procedures for developing functions in PostgreSQL.

PostgreSQL allows you to **extend** the **database functionality** with user-defined functions by using various procedural languages, which often referred to as stored procedures.

Procedural Languages

Procedural languages allow developers to extend the database with custom subroutines (functions), often called stored procedures. These functions can be used to build triggers (functions invoked upon modification of certain data) and custom aggregate functions.

- Procedural languages can also be invoked without defining a function, using the "DO" command at SQL level.
- Languages are divided into two groups: "Safe" languages are sandboxed and can be safely used by any user.
- Procedures written in "unsafe" languages can only be created by superusers, because they allow bypassing the database's security restrictions, but can also access sources external to the database.
- Some languages like Perl provide both safe and unsafe versions.

PostgreSQL has built-in support for three procedural languages:

- Plain SQL (safe). Simpler SQL functions can get expanded inline into the calling (SQL) query, which saves function call overhead and allows the query optimizer to "see inside" the function.
- PL/pgSQL (safe), which resembles Oracle's PL/SQL procedural language and SQL/PSM.
- C (unsafe), which allows loading custom shared libraries into the database. Functions written in C offer the best performance, but bugs in code can crash and potentially corrupt the database. Most built-in functions are written in C.

In addition, PostgreSQL allows procedural languages to be loaded into the database through extensions.

Three language extensions are included with PostgreSQL to support

Perl, Python and Tcl.

There are external projects to add support for many other languages, including Java, JavaScript (PL/V8), R.

Advantages of using PostgreSQL stored procedures

1) *Reduce the number of round trips* between application and database servers.

All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call.

2*) Increase application performance* because user-defined functions pre-compiled and stored in the PostgreSQL database server.

3) Be able to *reuse in many applications*. Once you develop a function, you can reuse it in any applications.

Disadvantages of using PostgreSQL stored procedures

1) Slow in software development because it requires specialized skills that many developers do not possess.

2) Make it difficult to manage versions and hard to debug.

3) May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server.

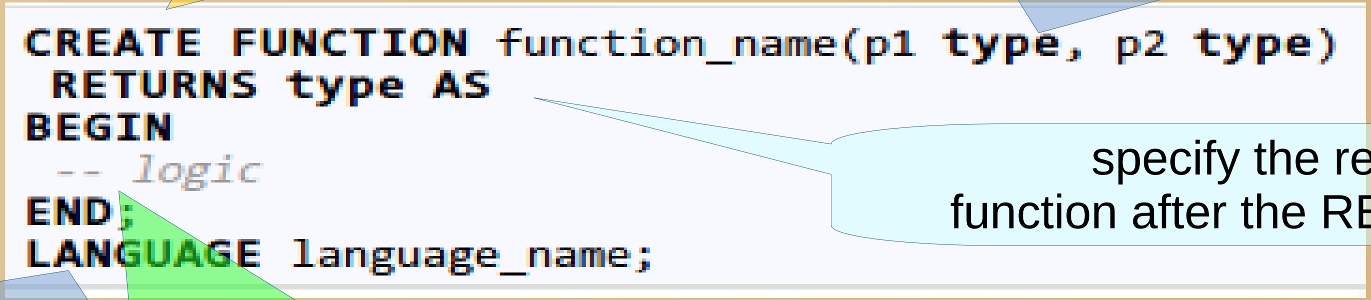
Developing User-defined Functions Using PostgreSQL CREATE FUNCTION Statement

Introduction to **CREATE FUNCTION** statement

To create a new user-defined function in PostgreSQL, we use the CREATE FUNCTION statement as follows:

specify the name of function followed by the CREATE FUNCTION clause.

put a comma-separated list of parameters inside the parentheses followed the function name.



```
CREATE FUNCTION function_name(p1 type, p2 type)
 RETURNS type AS
BEGIN
 -- logic
END;
LANGUAGE language_name;
```

specify the return type of function after the RETURNS keyword.

indicate the procedural language of the function e.g., plpgsql in case PL/pgSQL is used.

place the code inside the BEGIN and END block.
The function always ends with a semicolon (;) followed by the END keyword.

Examples

We are going to develop a very simple function named **inc** that increases an integer by 1 and returns the result.

1) We will launch psql program and login as postgres user to connect to the dvdrental sample database.

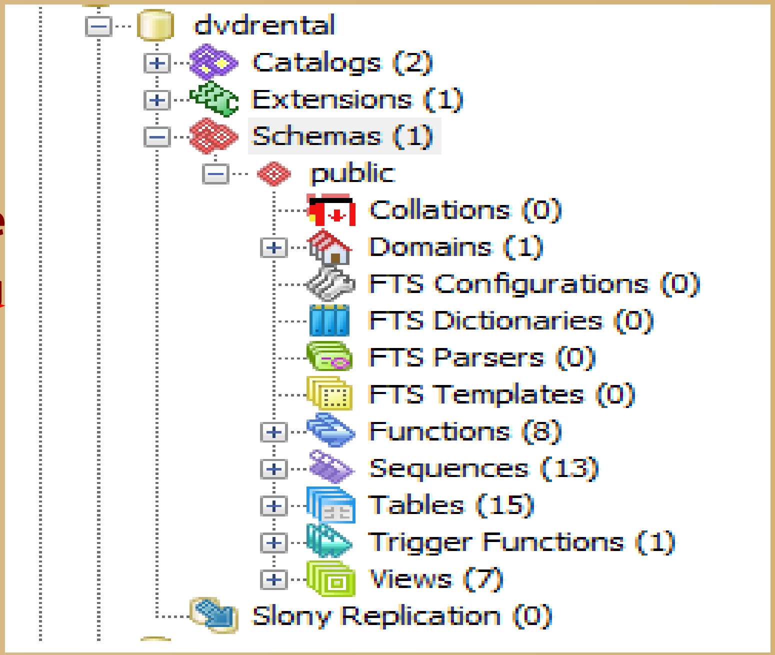
a) **CREATE DATABASE dvdrental;**

b) Load the DVD rental database

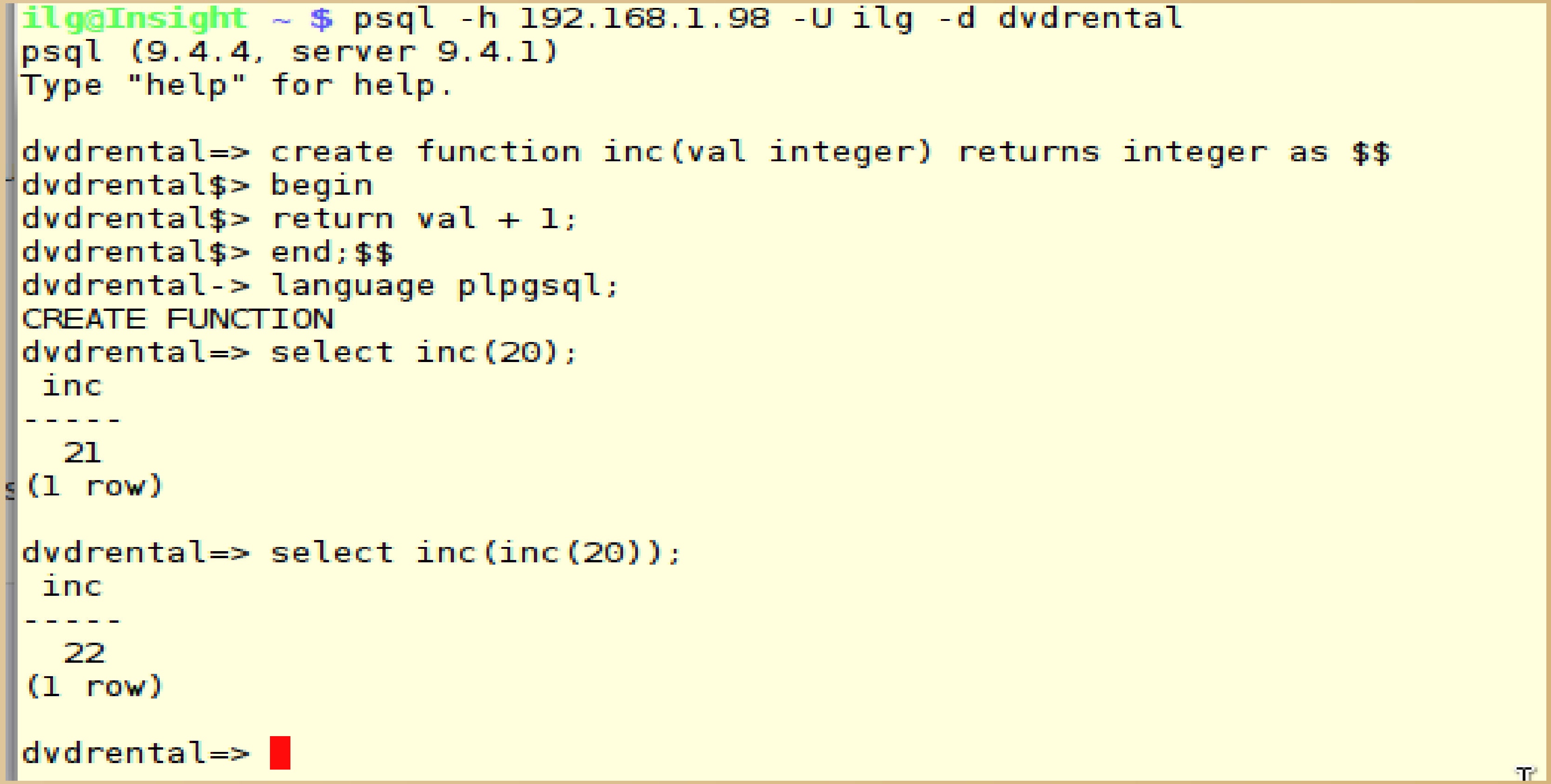
1) copy the DVD rental database to a folder e.g. **/home/ilg/postgre**

2) **pg_restore -h 192.168.1.98 -U ilg -d dvdrental /home/ilg/postg**

c) Verify the loaded sample database.



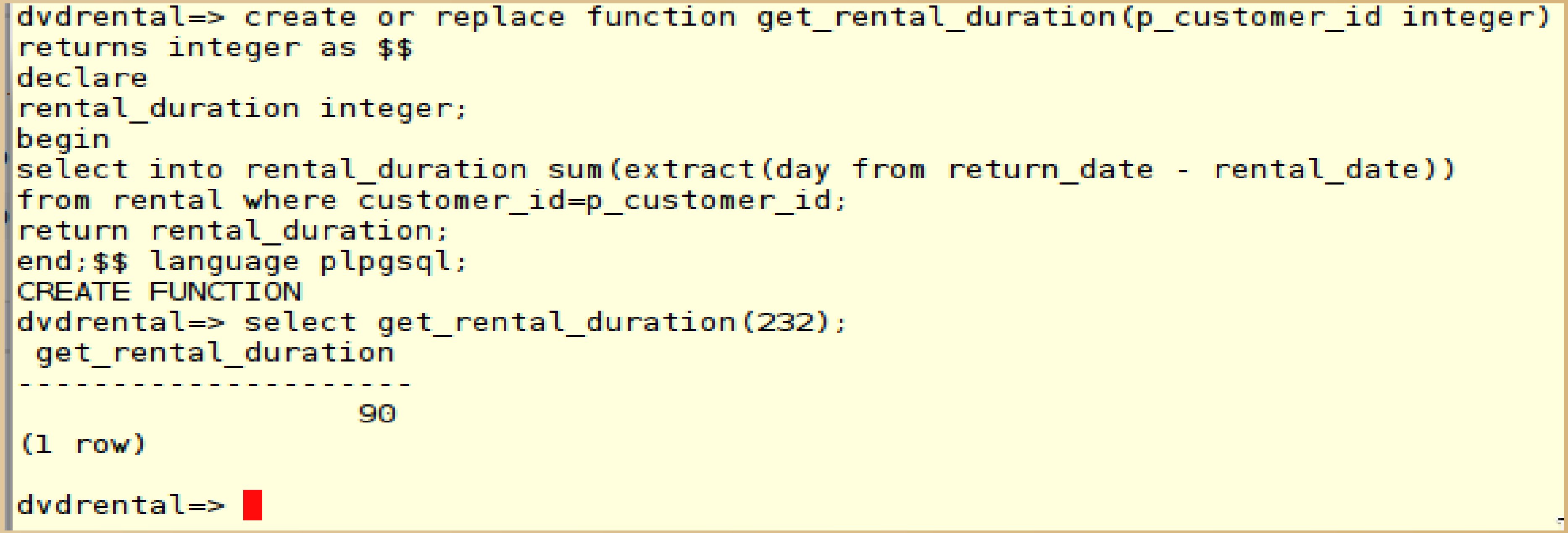
```
dvdrental
  Catalogs (2)
  Extensions (1)
  Schemas (1)
    public
      Collations (0)
      Domains (1)
      FTS Configurations (0)
      FTS Dictionaries (0)
      FTS Parsers (0)
      FTS Templates (0)
      Functions (8)
      Sequences (13)
      Tables (15)
      Trigger Functions (1)
      Views (7)
  Slony Replication (0)
```


```
ilg@Insight ~ $ psql -h 192.168.1.98 -U ilg -d dvdrental
psql (9.4.4, server 9.4.1)
Type "help" for help.

dvdrental=> create function inc(val integer) returns integer as $$
dvdrental$> begin
dvdrental$> return val + 1;
dvdrental$> end;$$
dvdrental-> language plpgsql;
CREATE FUNCTION
dvdrental=> select inc(20);
 inc
-----
  21
(1 row)

dvdrental=> select inc(inc(20));
 inc
-----
  22
(1 row)

dvdrental=> 
```

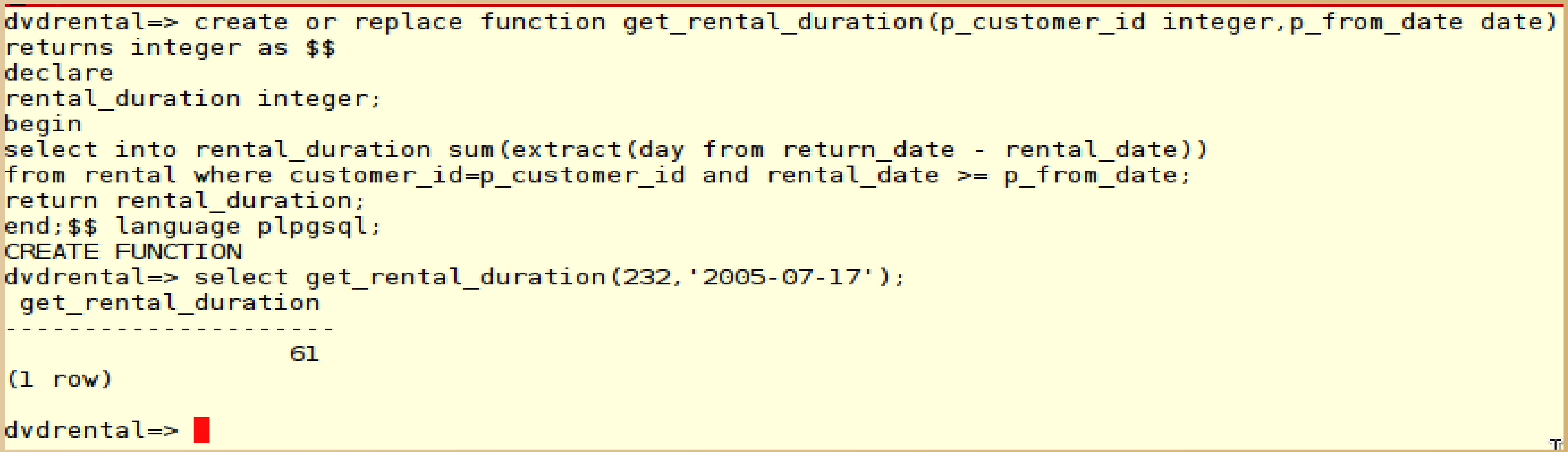
PL/pgSQL Function Overloading

PostgreSQL allows more than one function to have the same name, so long as the arguments are different. If more than one function has the same name, we say those functions are overloaded. When a function is called, PostgreSQL determines exact function is being called based on the input arguments.

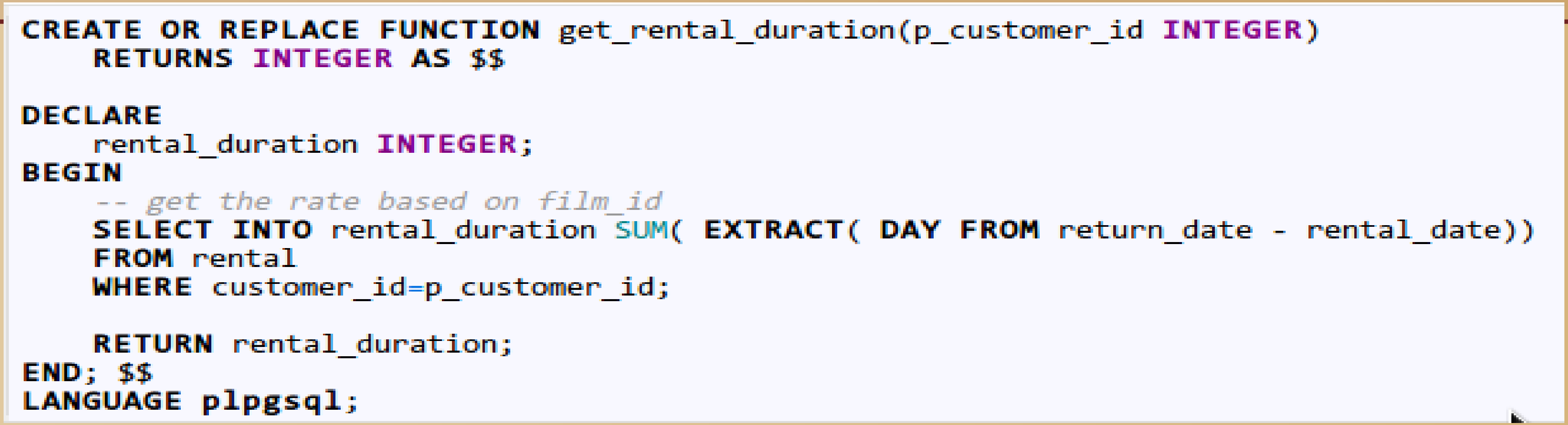
Let's take a look at the following **get_rental_duration()** function.



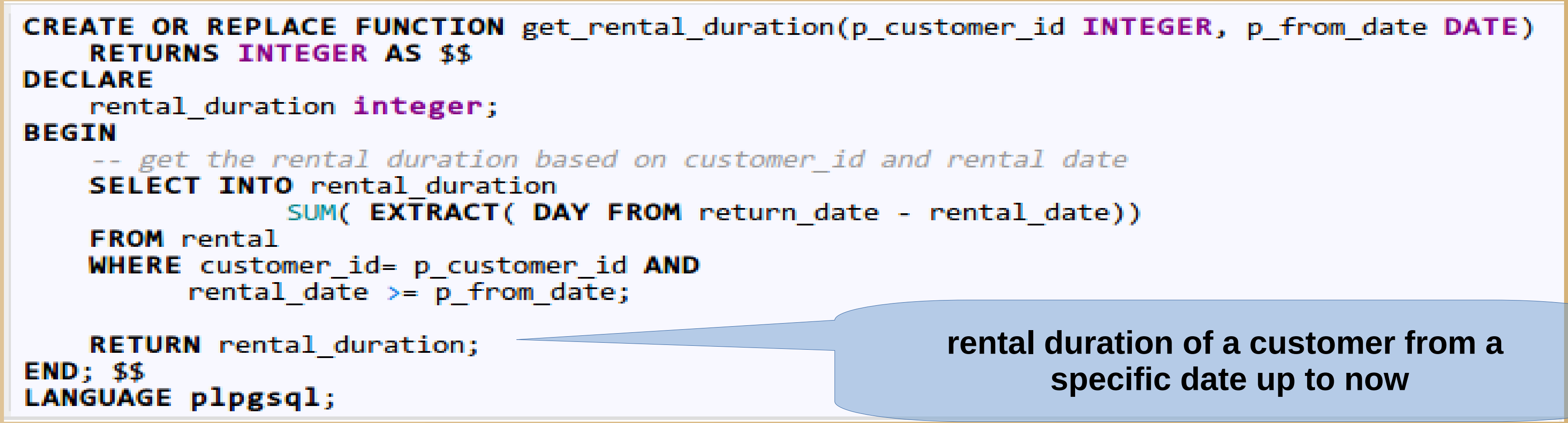
```
dvdrental=> create or replace function get_rental_duration(p_customer_id integer)
returns integer as $$
declare
rental_duration integer;
begin
select into rental_duration sum(extract(day from return_date - rental_date))
from rental where customer_id=p_customer_id;
return rental_duration;
end;$$ language plpgsql;
CREATE FUNCTION
dvdrental=> select get_rental_duration(232);
 get_rental_duration
---------------------
                  90
(1 row)

dvdrental=> 
```



```
dvdrental=> create or replace function get_rental_duration(p_customer_id integer,p_from_date date)
returns integer as $$
declare
rental_duration integer;
begin
select into rental_duration sum(extract(day from return_date - rental_date))
from rental where customer_id=p_customer_id and rental_date >= p_from_date;
return rental_duration;
end;$$ language plpgsql;
CREATE FUNCTION
dvdrental=> select get_rental_duration(232,'2005-07-17');
 get_rental_duration
---------------------
                  61
(1 row)

dvdrental=>
```



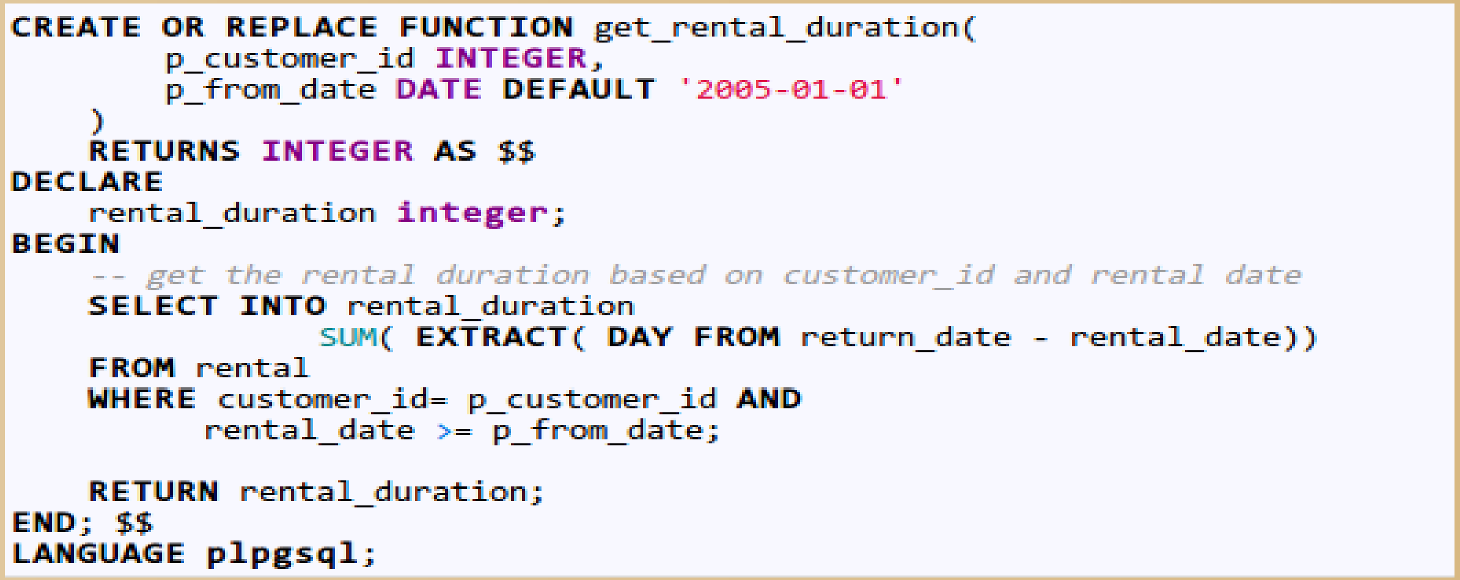
```
CREATE OR REPLACE FUNCTION get_rental_duration(p_customer_id INTEGER)
    RETURNS INTEGER AS $$

DECLARE
    rental_duration INTEGER;
BEGIN
    -- get the rate based on film_id
    SELECT INTO rental_duration SUM( EXTRACT( DAY FROM return_date - rental_date))
    FROM rental
    WHERE customer_id=p_customer_id;

    RETURN rental_duration;
END; $$
LANGUAGE plpgsql;
```



```
CREATE OR REPLACE FUNCTION get_rental_duration(p_customer_id INTEGER, p_from_date DATE)
    RETURNS INTEGER AS $$
DECLARE
    rental_duration integer;
BEGIN
    -- get the rental duration based on customer_id and rental date
    SELECT INTO rental_duration
                SUM( EXTRACT( DAY FROM return_date - rental_date))
    FROM rental
    WHERE customer_id= p_customer_id AND
          rental_date >= p_from_date;

    RETURN rental_duration;
END; $$
LANGUAGE plpgsql;
```

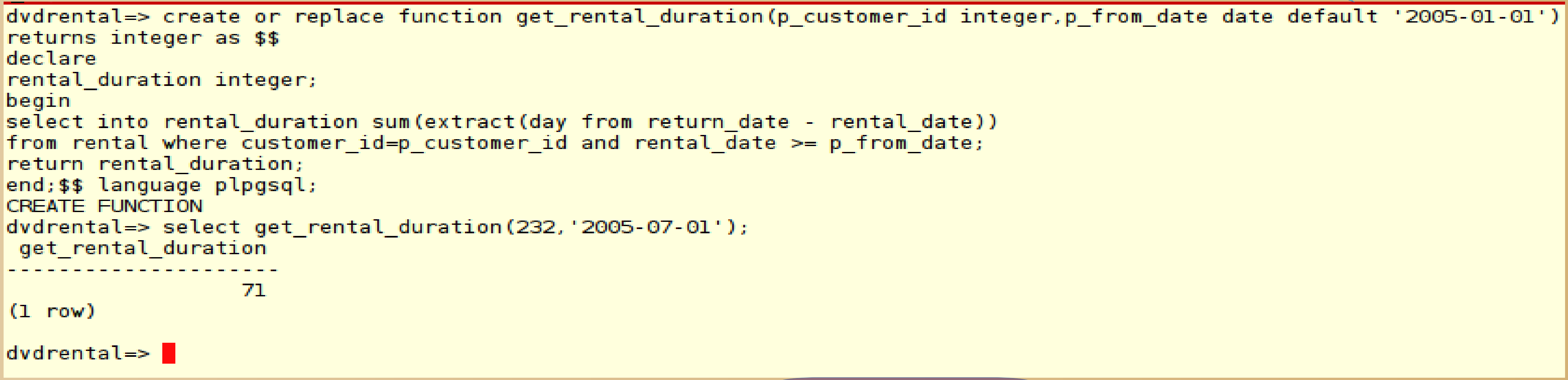
rental duration of a customer from a specific date up to now

PL/pgSQL function overloading and default value



```
CREATE OR REPLACE FUNCTION get_rental_duration(
        p_customer_id INTEGER,
        p_from_date DATE DEFAULT '2005-01-01'
    )
    RETURNS INTEGER AS $$
DECLARE
    rental_duration integer;
BEGIN
    -- get the rental duration based on customer_id and rental date
    SELECT INTO rental_duration
                SUM( EXTRACT( DAY FROM return_date - rental_date))
    FROM rental
    WHERE customer_id= p_customer_id AND
          rental_date >= p_from_date;

    RETURN rental_duration;
END; $$
LANGUAGE plpgsql;
```

Default date



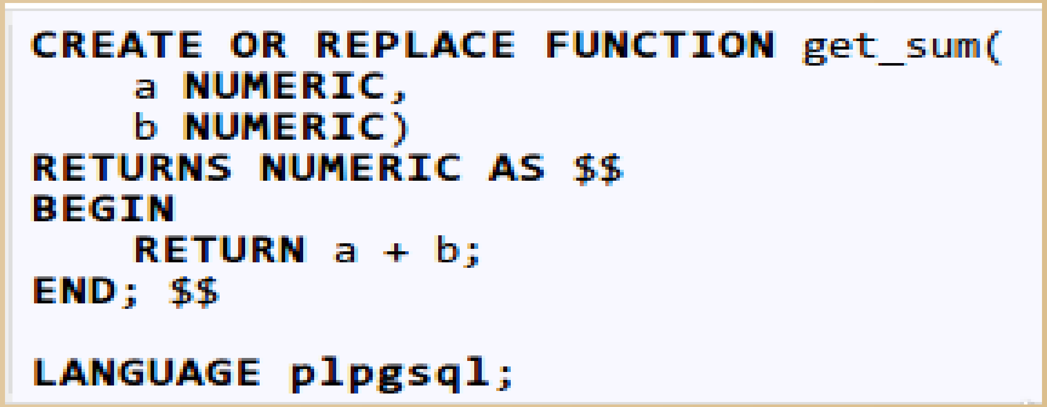
```
dvdrental=> create or replace function get_rental_duration(p_customer_id integer,p_from_date date default '2005-01-01')
returns integer as $$
declare
rental_duration integer;
begin
select into rental_duration sum(extract(day from return_date - rental_date))
from rental where customer_id=p_customer_id and rental_date >= p_from_date;
return rental_duration;
end;$$ language plpgsql;
CREATE FUNCTION
dvdrental=> select get_rental_duration(232,'2005-07-01');
 get_rental_duration
---------------------
                  71
(1 row)

dvdrental=> 
```

I am dropped

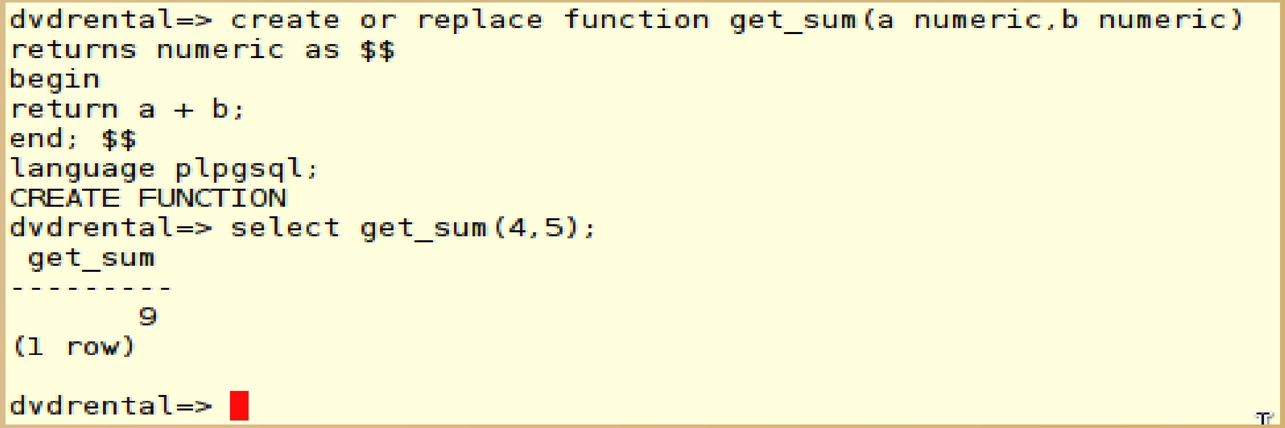
DROP FUNCTION get_rental_duration(INTEGER,DATE);

PL/pgSQL Function Parameters

PL/pgSQL IN parameters



```
CREATE OR REPLACE FUNCTION get_sum(
    a NUMERIC,
    b NUMERIC)
RETURNS NUMERIC AS $$
BEGIN
    RETURN a + b;
END; $$

LANGUAGE plpgsql;
```



```
dvdrental=> create or replace function get_sum(a numeric,b numeric)
returns numeric as $$
begin
return a + b;
end; $$
language plpgsql;
CREATE FUNCTION
dvdrental=> select get_sum(4,5);
 get_sum
---------
       9
(1 row)

dvdrental=> 
```

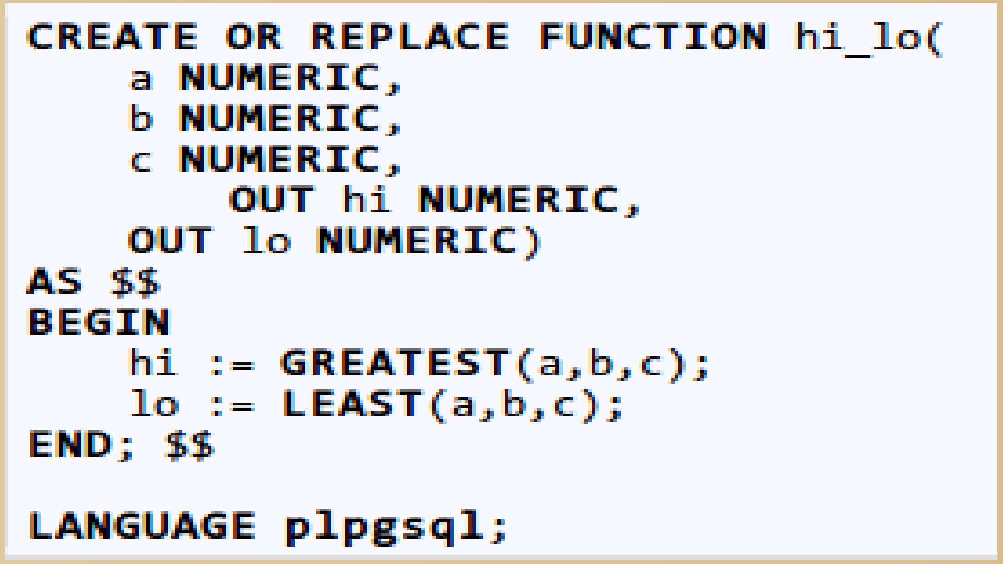
By default, the parameter's type of any parameter in PostgreSQL is IN parameter.

You can pass the IN parameters to the function but you cannot get them back as a part of result.

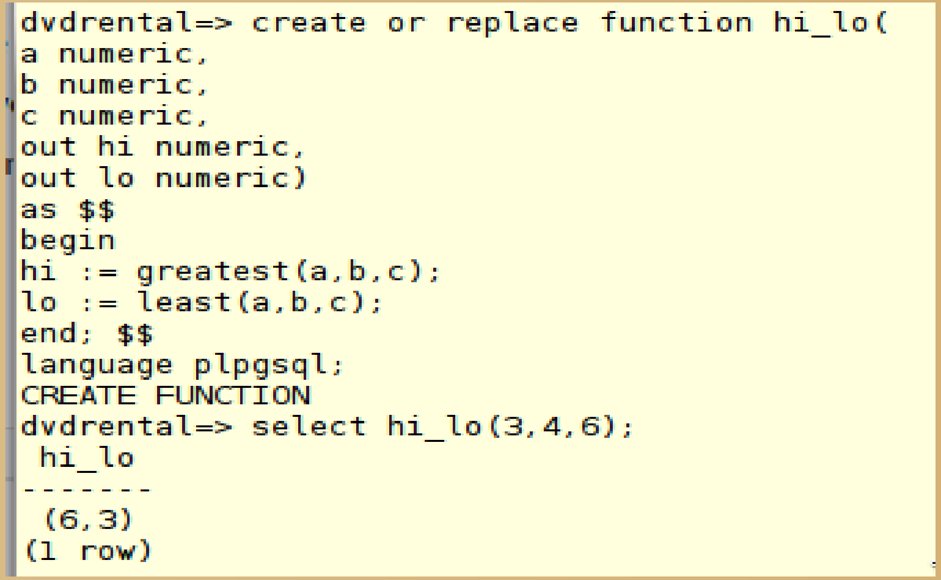
PL/pgSQL OUT parameters

The **OUT** parameters is a part of the function arguments list and you can get the result back as the part of the result.

To define OUT parameters, you use OUT keyword.



```
CREATE OR REPLACE FUNCTION hi_lo(
    a NUMERIC,
    b NUMERIC,
    c NUMERIC,
        OUT hi NUMERIC,
    OUT lo NUMERIC)
AS $$
BEGIN
    hi := GREATEST(a,b,c);
    lo := LEAST(a,b,c);
END; $$

LANGUAGE plpgsql;
```



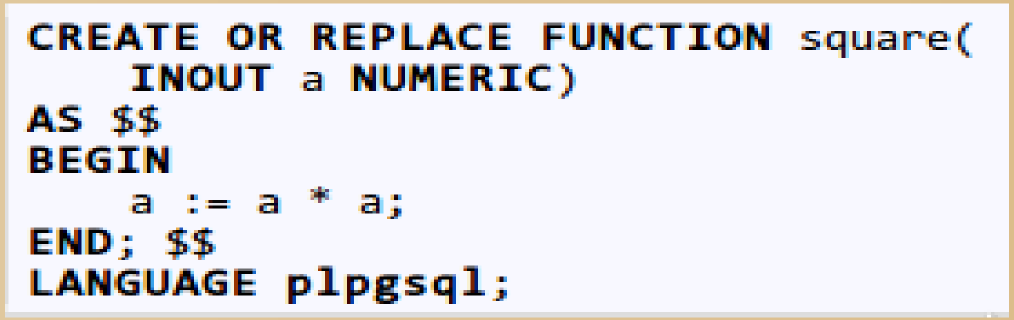
```
dvdrental=> create or replace function hi_lo(
a numeric,
b numeric,
c numeric,
out hi numeric,
out lo numeric)
as $$
begin
hi := greatest(a,b,c);
lo := least(a,b,c);
end; $$
language plpgsql;
CREATE FUNCTION
dvdrental=> select hi_lo(3,4,6);
 hi_lo
-------
 (6,3)
(1 row)
```

PL/pgSQL INOUT parameters

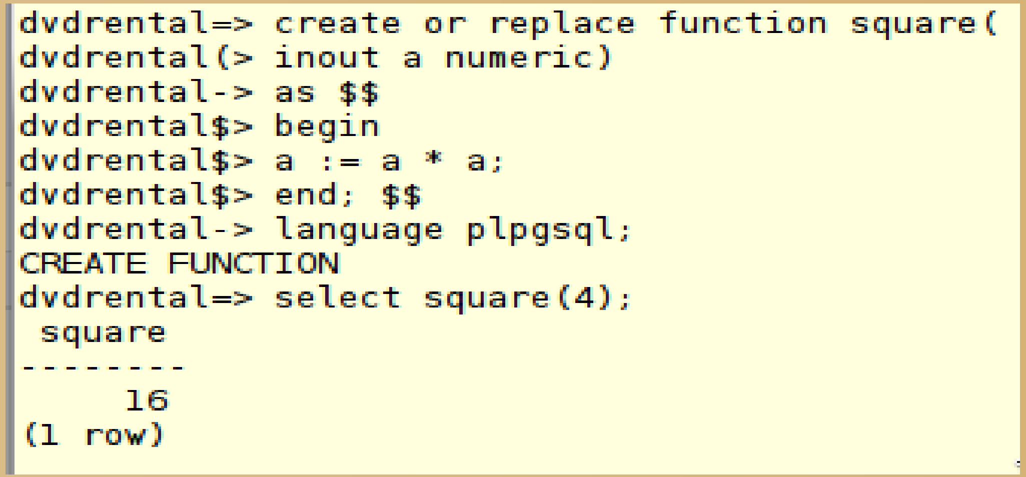
The INOUT parameter is the combination IN and OUT parameters.

It means that the caller can pass the value to the function.

The function then changes the argument and passes the value back as a part of the result.



```
CREATE OR REPLACE FUNCTION square(
    INOUT a NUMERIC)
AS $$
BEGIN
    a := a * a;
END; $$
LANGUAGE plpgsql;
```

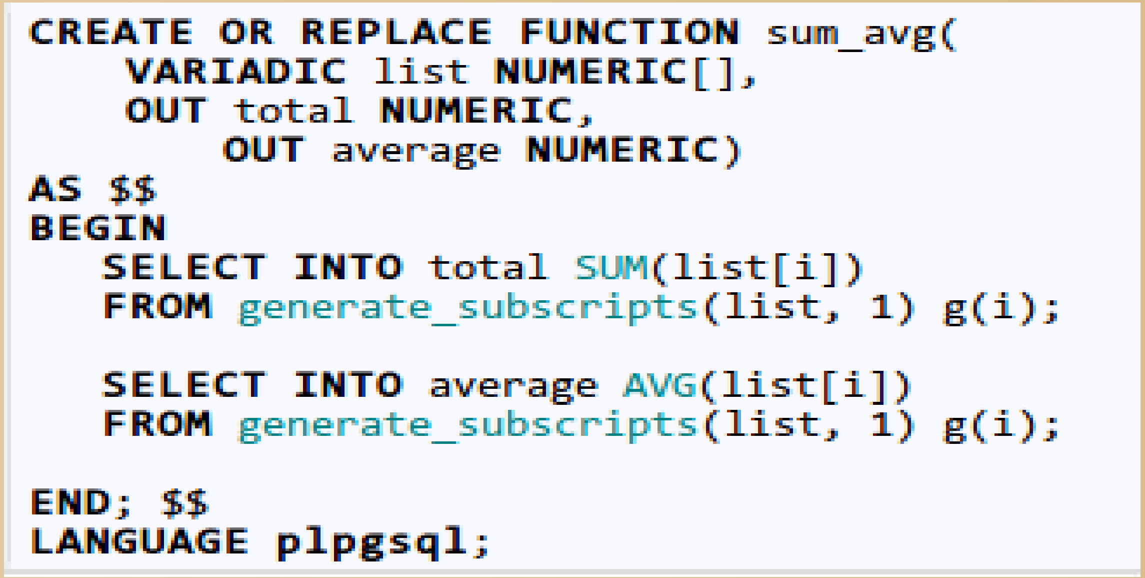


```
dvdrental=> create or replace function square(
dvdrental(> inout a numeric)
dvdrental-> as $$
dvdrental$> begin
dvdrental$> a := a * a;
dvdrental$> end; $$
dvdrental-> language plpgsql;
CREATE FUNCTION
dvdrental=> select square(4);
 square
--------
     16
(1 row)
```

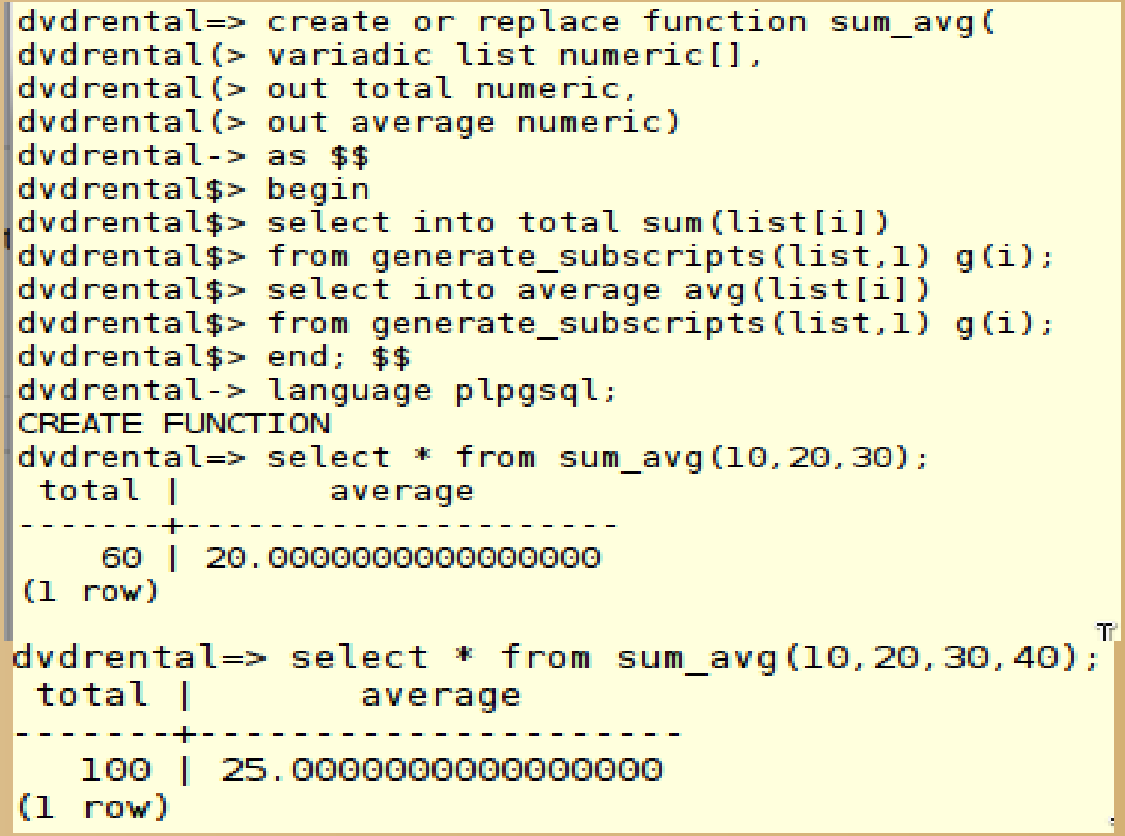
PL/pgSQL VARIADIC parameters

A PostgreSQL function can ***accept a variable numbers of arguments*** with one condition that all arguments have the same data type.

The arguments are passed to the function as an array.



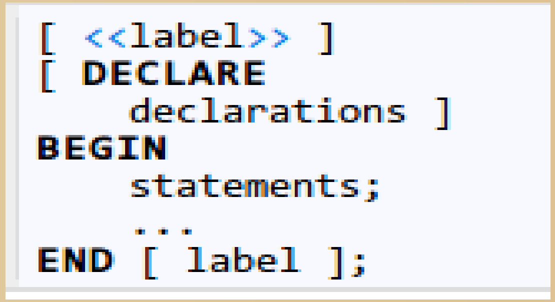
```
CREATE OR REPLACE FUNCTION sum_avg(
    VARIADIC list NUMERIC[],
    OUT total NUMERIC,
        OUT average NUMERIC)
AS $$
BEGIN
   SELECT INTO total SUM(list[i])
   FROM generate_subscripts(list, 1) g(i);

   SELECT INTO average AVG(list[i])
   FROM generate_subscripts(list, 1) g(i);

END; $$
LANGUAGE plpgsql;
```



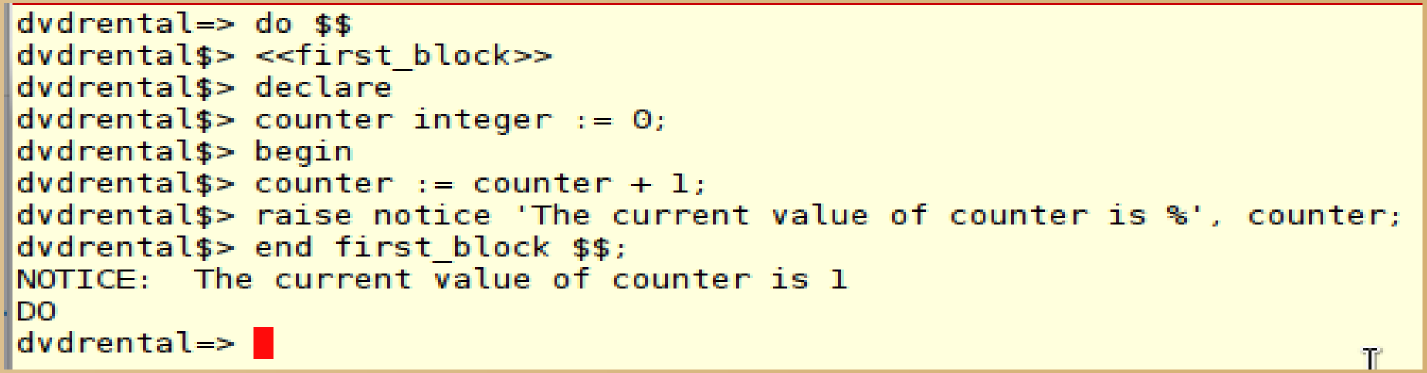
```
dvdrental=> create or replace function sum_avg(
dvdrental(> variadic list numeric[],
dvdrental(> out total numeric,
dvdrental(> out average numeric)
dvdrental-> as $$
dvdrental$> begin
dvdrental$> select into total sum(list[i])
dvdrental$> from generate_subscripts(list,1) g(i);
dvdrental$> select into average avg(list[i])
dvdrental$> from generate_subscripts(list,1) g(i);
dvdrental$> end; $$
dvdrental-> language plpgsql;
CREATE FUNCTION
dvdrental=> select * from sum_avg(10,20,30);
 total |       average
-------+---------------------
    60 | 20.0000000000000000
(1 row)

dvdrental=> select * from sum_avg(10,20,30,40);
 total |       average
-------+----------------------
   100 | 25.0000000000000000
(1 row)
```

PL/pgSQL Block Structure

A PL/pgSQL function is organized into blocks.

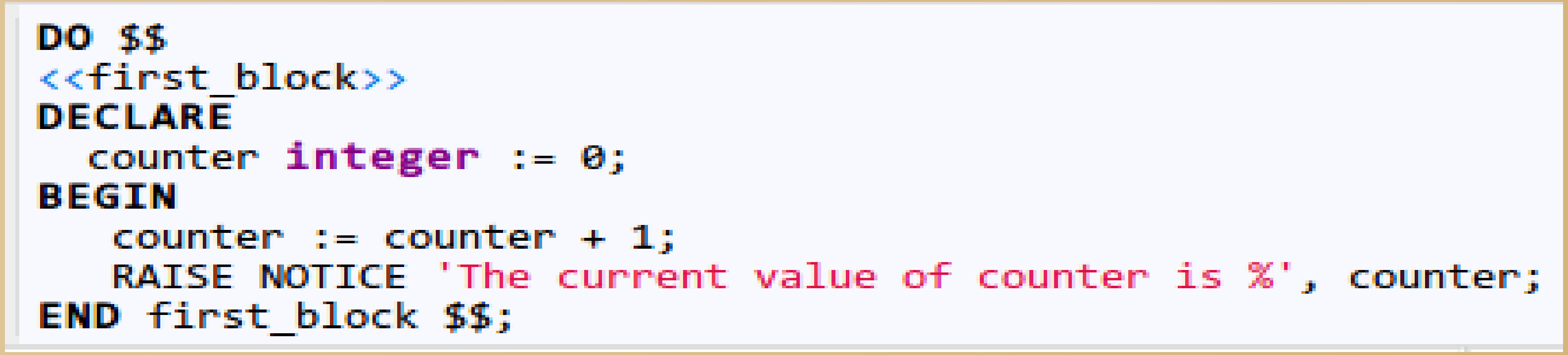


```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements;
    ...
END [ label ];
```



```
dvdrental=> do $$
dvdrental$> <<first_block>>
dvdrental$> declare
dvdrental$> counter integer := 0;
dvdrental$> begin
dvdrental$> counter := counter + 1;
dvdrental$> raise notice 'The current value of counter is %', counter;
dvdrental$> end first_block $$;
NOTICE:  The current value of counter is 1
DO
dvdrental=>
```

PL/pgSQL block structure example



```
DO $$
<<first_block>>
DECLARE
  counter integer := 0;
BEGIN
   counter := counter + 1;
   RAISE NOTICE 'The current value of counter is %', counter;
END first_block $$;
```

DO statement does not belong to the block.

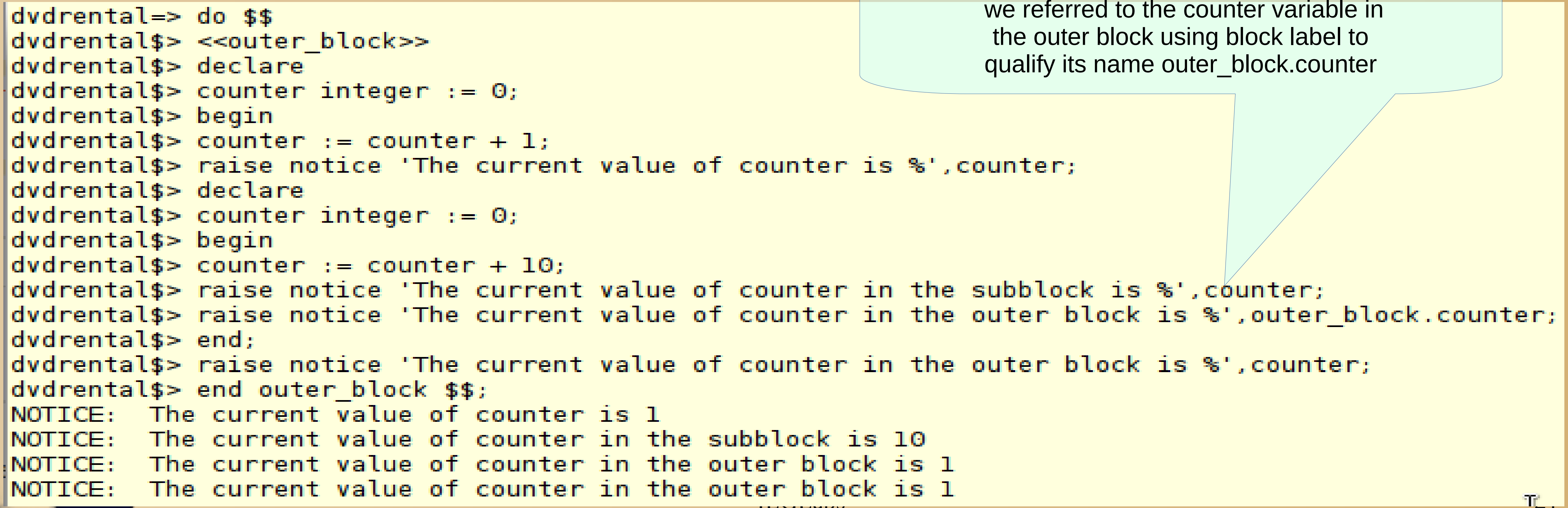
It is used to execute an anonymous block.

PostgreSQL introduced the DO statement since version 9.0.

Same name blocks

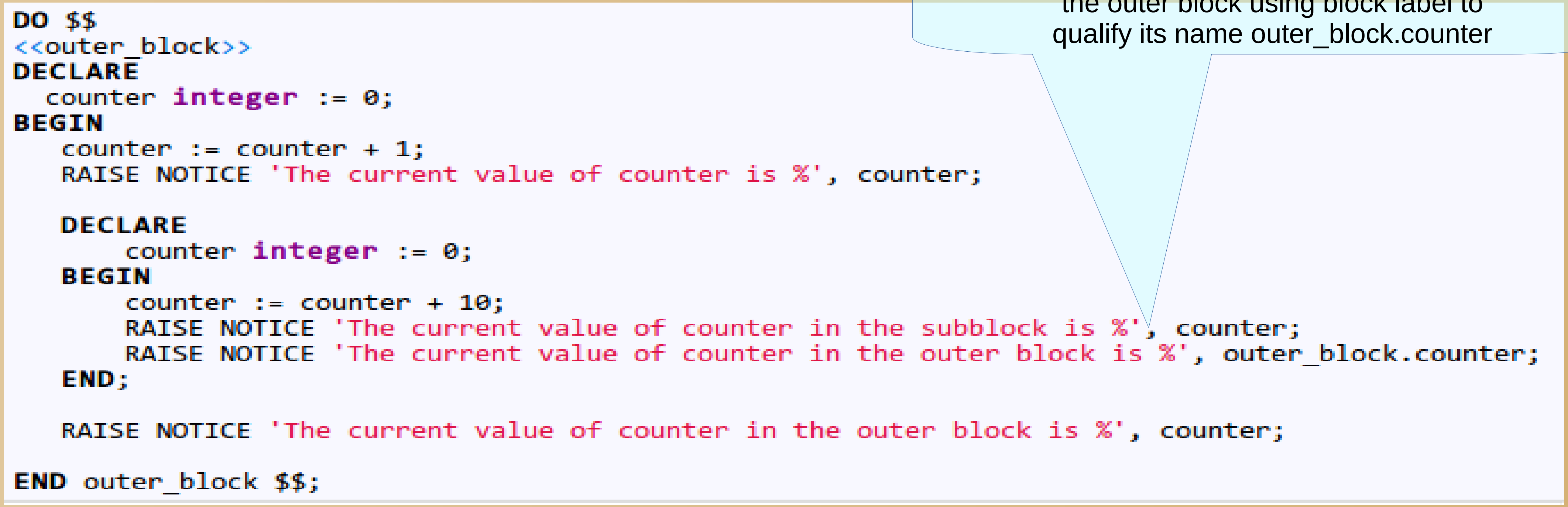
When you define a variable within subblock with the same name as the one in the outer block, the variable in the outer block is hidden in the subblock. In case you want to access a variable in the outer block, you use block label to qualify its name;

we referred to the counter variable in the outer block using block label to qualify its name outer_block.counter



```
dvdrental=> do $$
dvdrental$> <<outer_block>>
dvdrental$> declare
dvdrental$> counter integer := 0;
dvdrental$> begin
dvdrental$> counter := counter + 1;
dvdrental$> raise notice 'The current value of counter is %',counter;
dvdrental$> declare
dvdrental$> counter integer := 0;
dvdrental$> begin
dvdrental$> counter := counter + 10;
dvdrental$> raise notice 'The current value of counter in the subblock is %',counter;
dvdrental$> raise notice 'The current value of counter in the outer block is %',outer_block.counter;
dvdrental$> end;
dvdrental$> raise notice 'The current value of counter in the outer block is %',counter;
dvdrental$> end outer_block $$;
NOTICE:  The current value of counter is 1
NOTICE:  The current value of counter in the subblock is 10
NOTICE:  The current value of counter in the outer block is 1
NOTICE:  The current value of counter in the outer block is 1
```

we referred to the counter variable in the outer block using block label to qualify its name outer_block.counter



```
DO $$
<<outer_block>>
DECLARE
  counter integer := 0;
BEGIN
   counter := counter + 1;
   RAISE NOTICE 'The current value of counter is %', counter;

   DECLARE
       counter integer := 0;
   BEGIN
       counter := counter + 10;
       RAISE NOTICE 'The current value of counter in the subblock is %', counter;
       RAISE NOTICE 'The current value of counter in the outer block is %', outer_block.counter;
   END;

   RAISE NOTICE 'The current value of counter in the outer block is %', counter;

END outer_block $$;
```

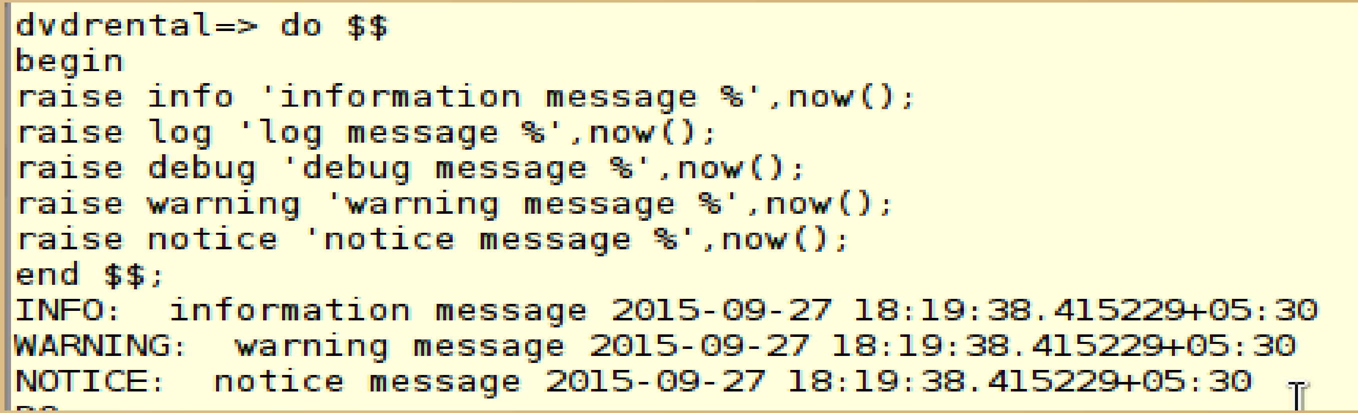
PL/pgSQL Errors and Messages

PL/pgSQL reporting messages

To raise a message, you use the RAISE statement as follows:

RAISE level format;

DEBUG
LOG
NOTICE
INFO
WARNING
EXCEPTION



```
dvdrental=> do $$
begin
raise info 'information message %',now();
raise log 'log message %',now();
raise debug 'debug message %',now();
raise warning 'warning message %',now();
raise notice 'notice message %',now();
end $$;
INFO:  information message 2015-09-27 18:19:38.415229+05:30
WARNING:  warning message 2015-09-27 18:19:38.415229+05:30
NOTICE:  notice message 2015-09-27 18:19:38.415229+05:30
```

Format is a string that specifies the message.

The format uses percentage (%) placeholders that will be substituted by the next arguments.

The number of placeholders must match the number of arguments,
otherwise PostgreSQL will report the following error message:

[Err] ERROR: too many parameters specified for RAISE

PL/pgSQL raising errors

To raise errors, you use the EXCEPTION level after the RAISE statement. Note that RAISE statement uses EXCEPTION level by default.

Besides raising an error, you can add more detailed information using the following clause with the RAISE statement:

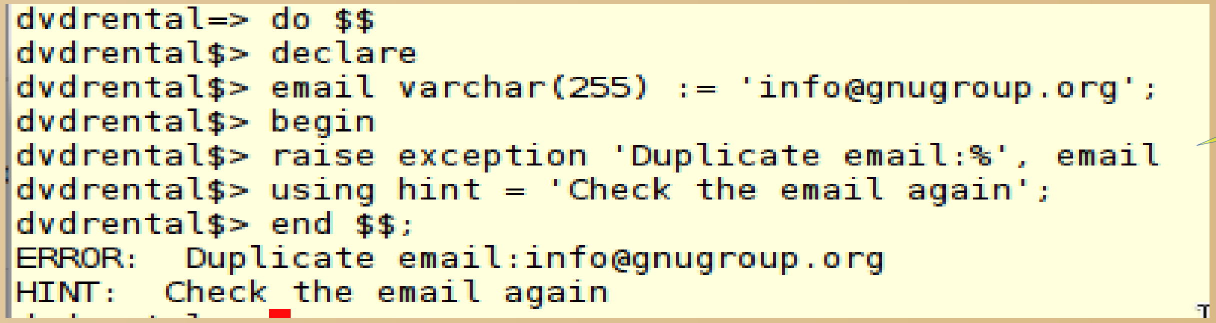
USING option = expression

MESSAGE: to set error message text

HINT: to provide the hint message so that the root cause of the error is easier to be discovered.

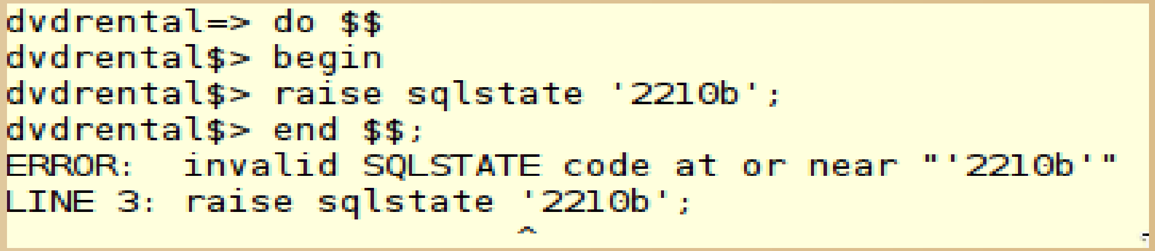
DETAIL: to give detailed information about the error.

ERRCODE: to identify the error code, which can be either by condition name
or directly five-character SQLSTATE code.



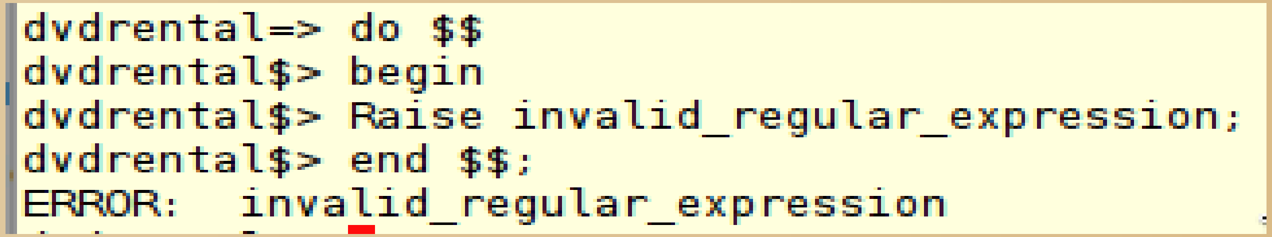
```
dvdrental=> do $$
dvdrental$> declare
dvdrental$> email varchar(255) := 'info@gnugroup.org';
dvdrental$> begin
dvdrental$> raise exception 'Duplicate email:%', email
dvdrental$> using hint = 'Check the email again';
dvdrental$> end $$;
ERROR:  Duplicate email:info@gnugroup.org
HINT:  Check the email again
```

raises a duplicate email error message:



```
dvdrental=> do $$
dvdrental$> begin
dvdrental$> raise sqlstate '2210b';
dvdrental$> end $$;
ERROR:  invalid SQLSTATE code at or near "'2210b'"
LINE 3: raise sqlstate '2210b';
                       ^
```

how to raise a SQLSTATE and its corresponding condition



```
dvdrental=> do $$
dvdrental$> begin
dvdrental$> Raise invalid_regular_expression;
dvdrental$> end $$;
ERROR:  invalid_regular_expression
```

PL/pgSQL putting debugging checks using ASSERT statement

Notice that PostgreSQL introduces the ASSERT statement since version 9.5. *Check your PostgreSQL version before using it.*

****
******

Sometimes, a PL/pgSQL function is so big that make it more difficult to detect the bugs.

To facilitate this, PostgreSQL provides you with the ASSERT statement for adding debugging checks into a PL/pgSQL function.

The condition is a boolean expression. If the condition evaluates to TRUE, ASSERT statement does nothing. If the condition evaluates to FALSE or NULL, the ASSERT_FAILURE is raised.

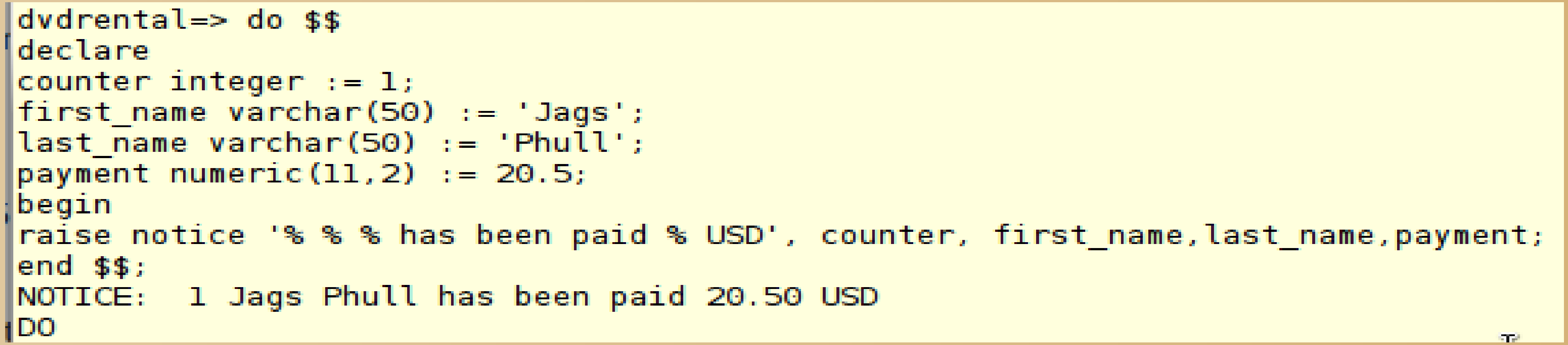
If you don’t provide the message, PL/pgSQL uses ***” assertion failed” message by default.*** If the message is provided, the ASSERT statement will use it to replace the default message.

PL/pgSQL Variables

A PL/pgSQL variable is a **meaningful name of a memory location.**

A variable **holds a value** that can be changed through the block or function.

A variable is always associated with a **particular data type**.



```
dvdrental=> do $$
declare
counter integer := 1;
first_name varchar(50) := 'Jags';
last_name varchar(50) := 'Phull';
payment numeric(11,2) := 20.5;
begin
raise notice '% % % has been paid % USD', counter, first_name,last_name,payment;
end $$;
NOTICE:  1 Jags Phull has been paid 20.50 USD
DO
```

PL/pgSQL Constants

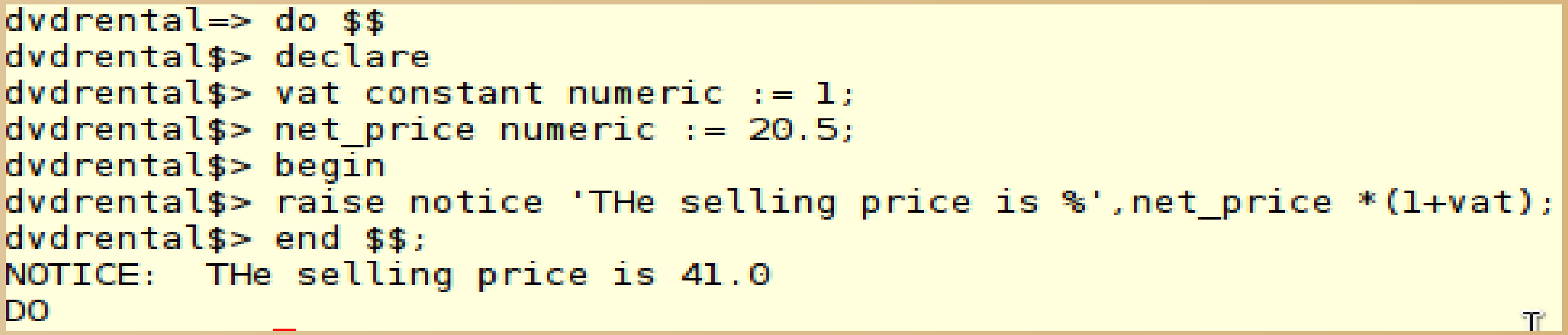
Values of constants cannot be changed once they are initialized.

Declaring constant syntax

constant_name CONSTANT data_type := expression;

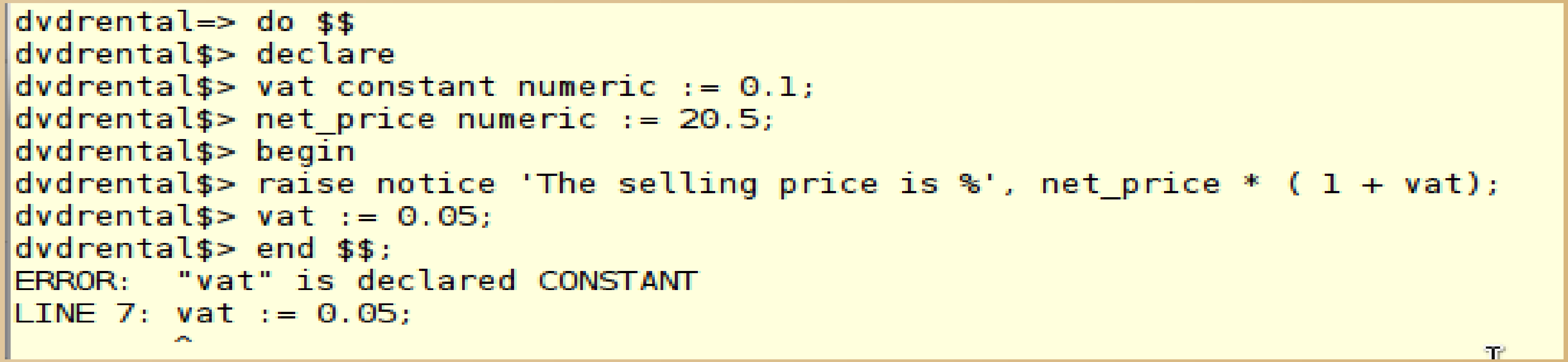
PL/pgSQL constants example

This e.g declares a constant named VAT for valued added tax, and calculates the selling price from the net price:



```
dvdrental=> do $$
dvdrental$> declare
dvdrental$> vat constant numeric := 1;
dvdrental$> net_price numeric := 20.5;
dvdrental$> begin
dvdrental$> raise notice 'THe selling price is %',net_price *(1+vat);
dvdrental$> end $$;
NOTICE:  THe selling price is 41.0
DO
```

Now, if you try to change the value of the constant as follows:

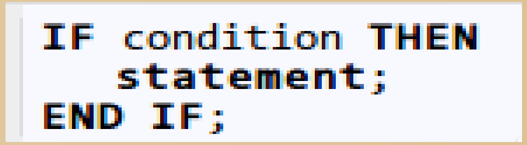


```
dvdrental=> do $$
dvdrental$> declare
dvdrental$> vat constant numeric := 0.1;
dvdrental$> net_price numeric := 20.5;
dvdrental$> begin
dvdrental$> raise notice 'The selling price is %', net_price * ( 1 + vat);
dvdrental$> vat := 0.05;
dvdrental$> end $$;
ERROR:  "vat" is declared CONSTANT
LINE 7: vat := 0.05;
        ^
```

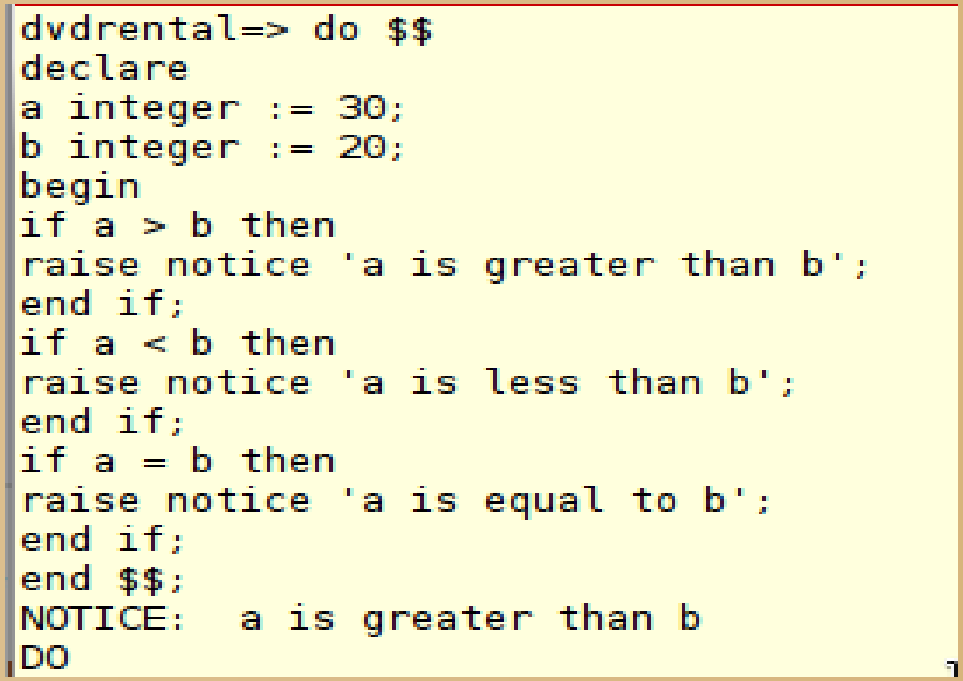
Notice that PostgreSQL evaluates the value for the constant when the block is entered at run-time, not compiled-time.

PL/pgSQL IF Statement

The simplest form of PL/pgSQL IF statement

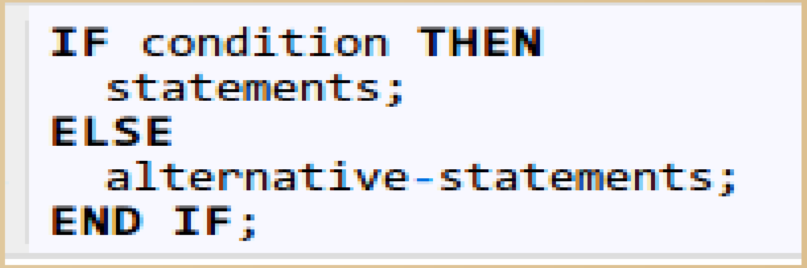


```
IF condition THEN
   statement;
END IF;
```

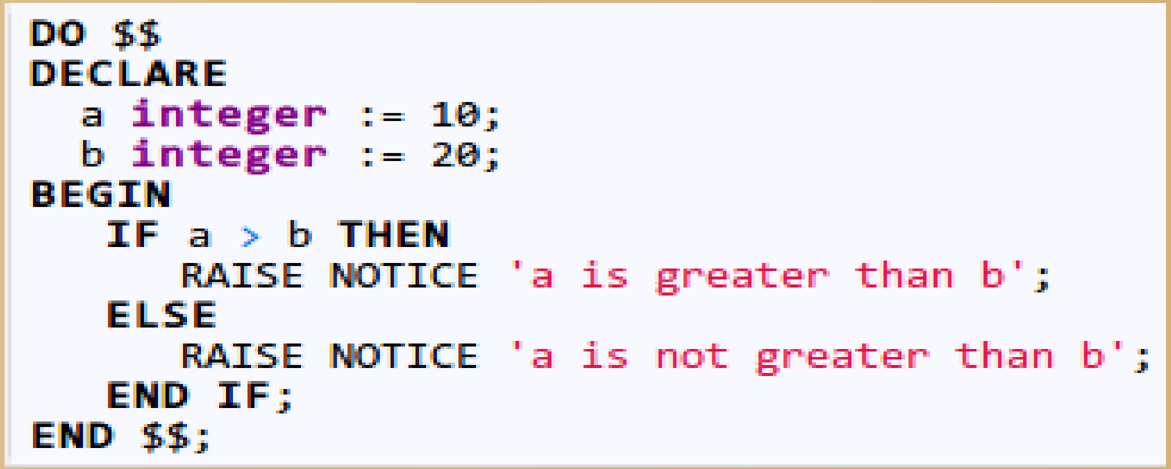


```
dvdrental=> do $$
declare
a integer := 30;
b integer := 20;
begin
if a > b then
raise notice 'a is greater than b';
end if;
if a < b then
raise notice 'a is less than b';
end if;
if a = b then
raise notice 'a is equal to b';
end if;
end $$;
NOTICE:  a is greater than b
DO
```

PL/pgSQL IF THEN ELSE statement

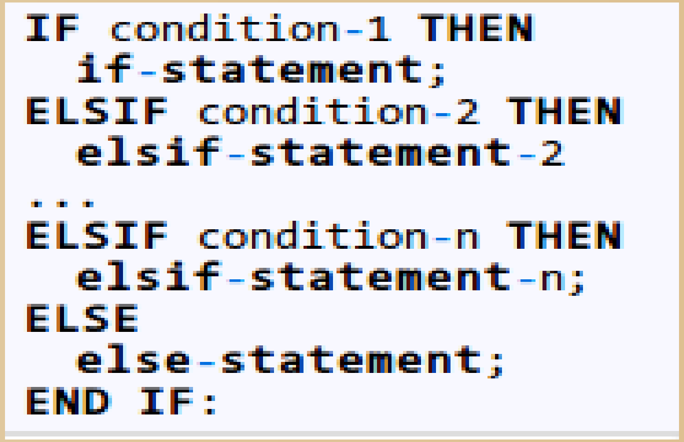


```
IF condition THEN
  statements;
ELSE
  alternative-statements;
END IF;
```

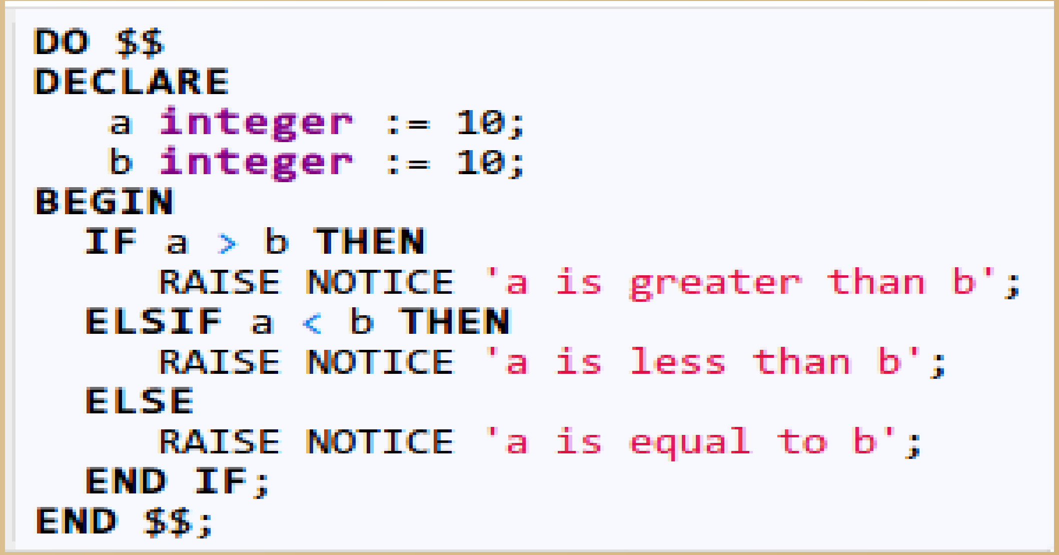


```
DO $$
DECLARE
  a integer := 10;
  b integer := 20;
BEGIN
   IF a > b THEN
      RAISE NOTICE 'a is greater than b';
   ELSE
      RAISE NOTICE 'a is not greater than b';
   END IF;
END $$;
```

PL/pgSQL IF THEN ELSIF THEN ELSE statement



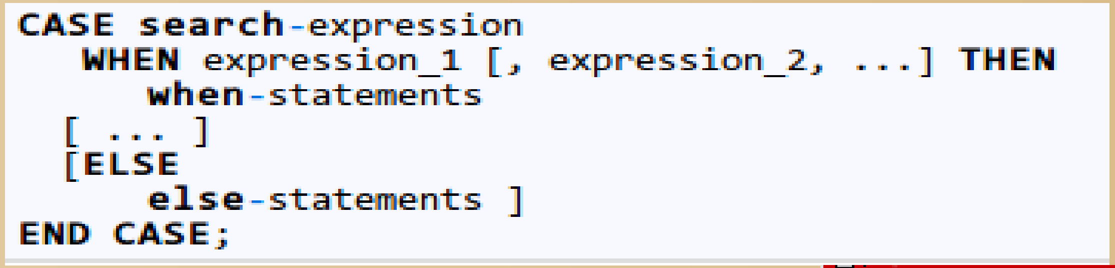
```
IF condition-1 THEN
  if-statement;
ELSIF condition-2 THEN
  elsif-statement-2
...
ELSIF condition-n THEN
  elsif-statement-n;
ELSE
  else-statement;
END IF:
```



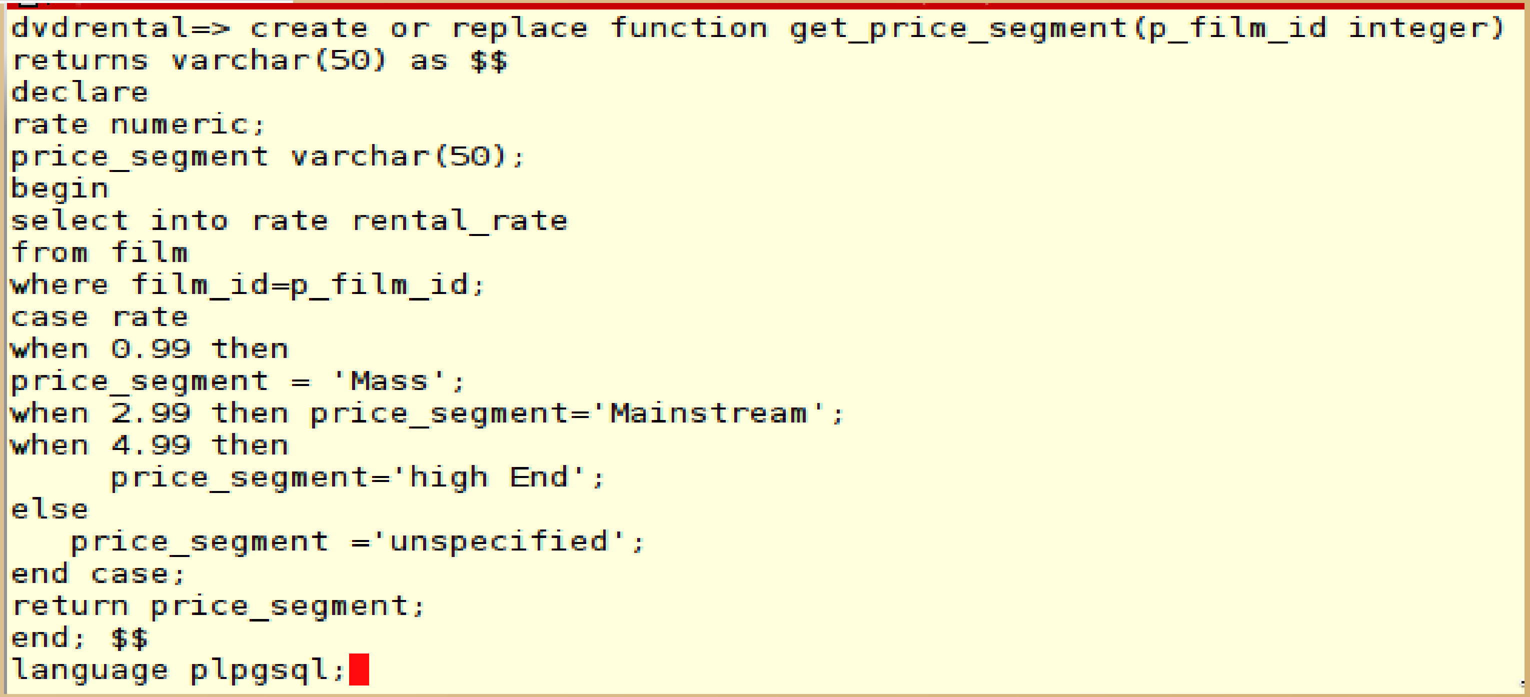
```
DO $$
DECLARE
   a integer := 10;
   b integer := 10;
BEGIN
  IF a > b THEN
     RAISE NOTICE 'a is greater than b';
  ELSIF a < b THEN
     RAISE NOTICE 'a is less than b';
  ELSE
     RAISE NOTICE 'a is equal to b';
  END IF;
END $$;
```

PL/pgSQL CASE Statement

Simple CASE statement

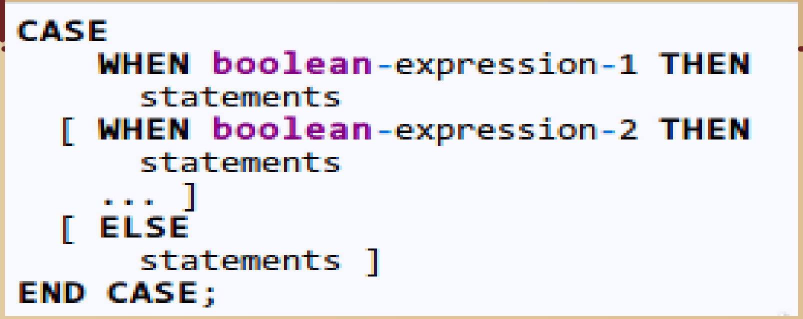


```
CASE search-expression
   WHEN expression_1 [, expression_2, ...] THEN
      when-statements
  [ ... ]
  [ELSE
      else-statements ]
END CASE;
```



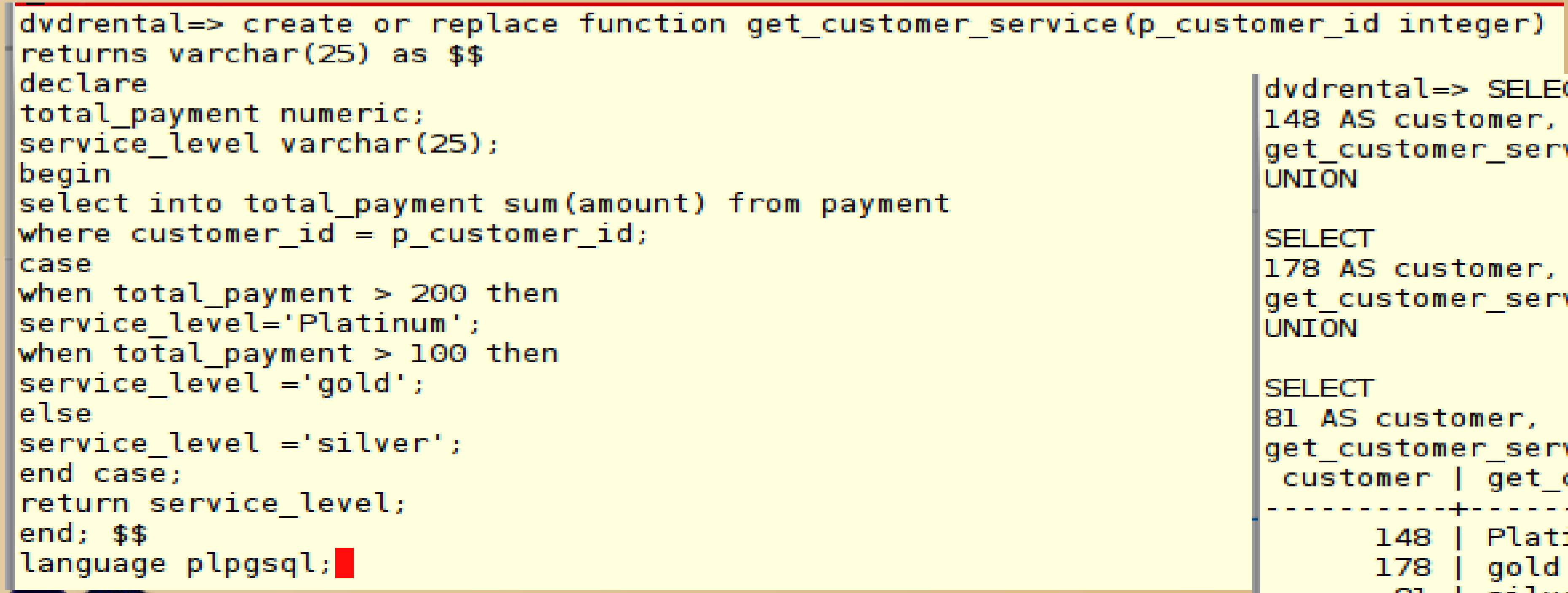
```
dvdrental=> create or replace function get_price_segment(p_film_id integer)
returns varchar(50) as $$
declare
rate numeric;
price_segment varchar(50);
begin
select into rate rental_rate
from film
where film_id=p_film_id;
case rate
when 0.99 then
price_segment = 'Mass';
when 2.99 then price_segment='Mainstream';
when 4.99 then
     price_segment='high End';
else
   price_segment ='unspecified';
end case;
return price_segment;
end; $$
language plpgsql;
```

Searched CASE statement

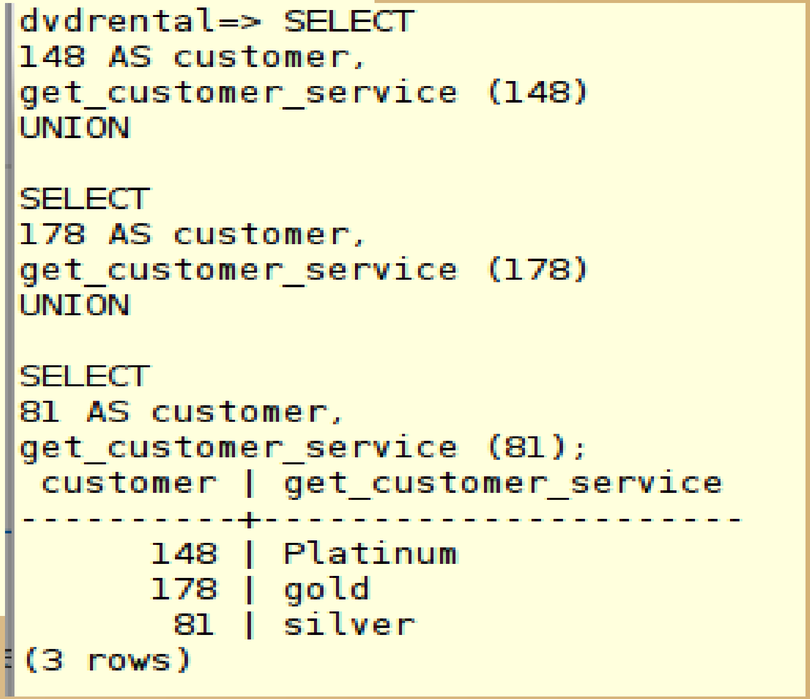


```
CASE
    WHEN boolean-expression-1 THEN
      statements
  [ WHEN boolean-expression-2 THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Searched CASE statement executes statements based on the result of Boolean expressions in each WHEN clause. PostgreSQL evaluates the Boolean expressions sequentially from top to bottom until one expression is true. Then the evaluation stops and the corresponding statement are executed. The control is passed to the next statement after END CASE.



```
dvdrental=> create or replace function get_customer_service(p_customer_id integer)
returns varchar(25) as $$
declare
total_payment numeric;
service_level varchar(25);
begin
select into total_payment sum(amount) from payment
where customer_id = p_customer_id;
case
when total_payment > 200 then
service_level='Platinum';
when total_payment > 100 then
service_level ='gold';
else
service_level ='silver';
end case;
return service_level;
end; $$
language plpgsql;
```

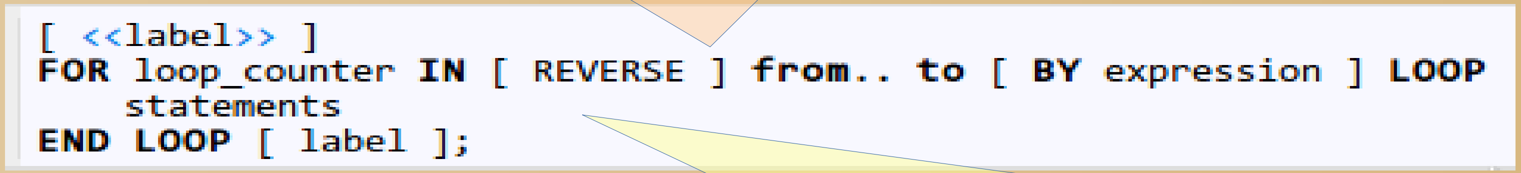


```
dvdrental=> SELECT
148 AS customer,
get_customer_service (148)
UNION

SELECT
178 AS customer,
get_customer_service (178)
UNION

SELECT
81 AS customer,
get_customer_service (81);
 customer | get_customer_service
----------+----------------------
      148 | Platinum
      178 | gold
       81 | silver
(3 rows)
```

PL/pgSQL FOR loop statement

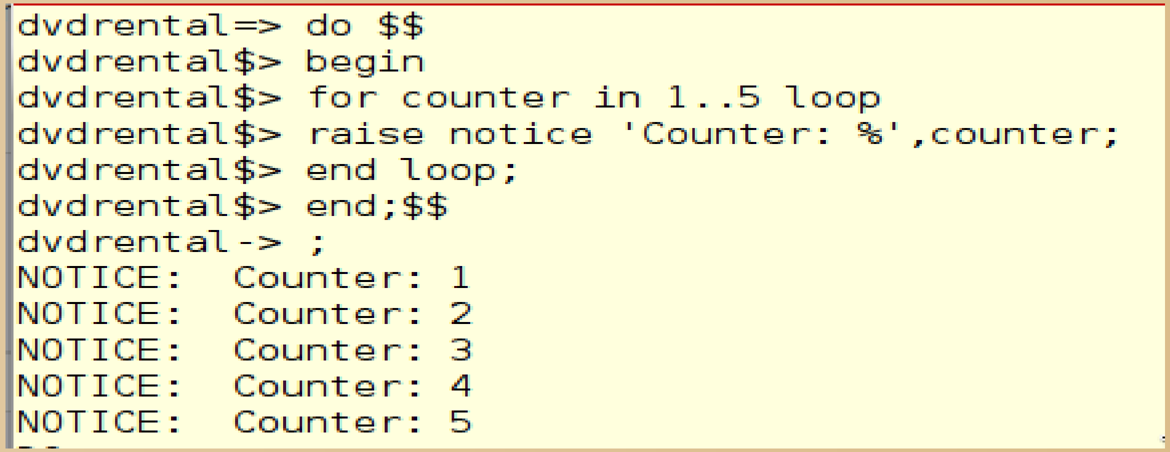
FOR loop for looping through a ranges of integers

from and **to** are expressions that specify the lower and upper bound of the range.



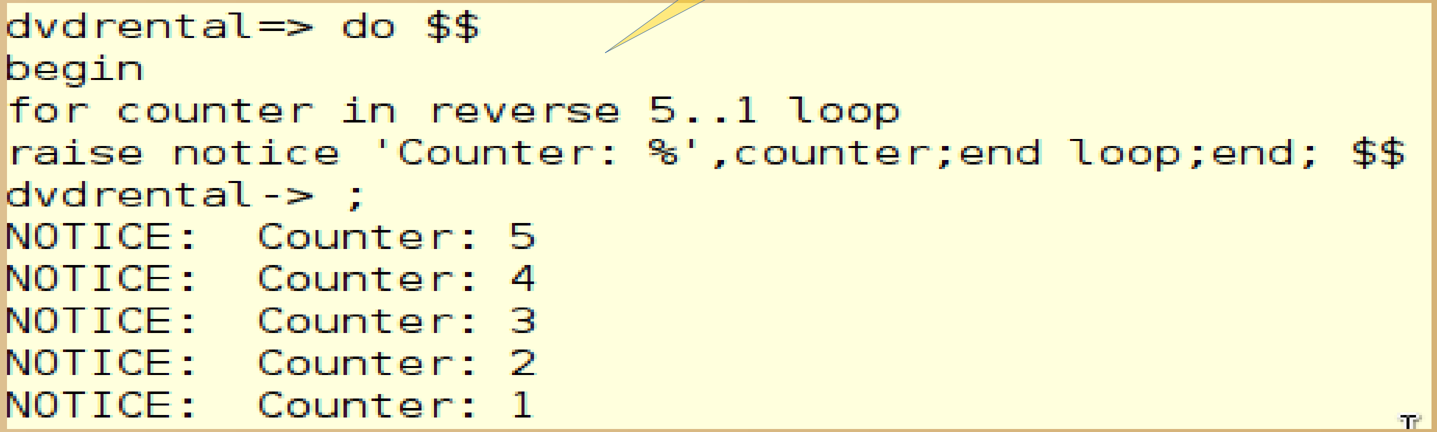
```
[ <<label>> ]
FOR loop_counter IN [ REVERSE ] from.. to [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

REVERSE keyword, PostgreSQL will subtract the loop counter.

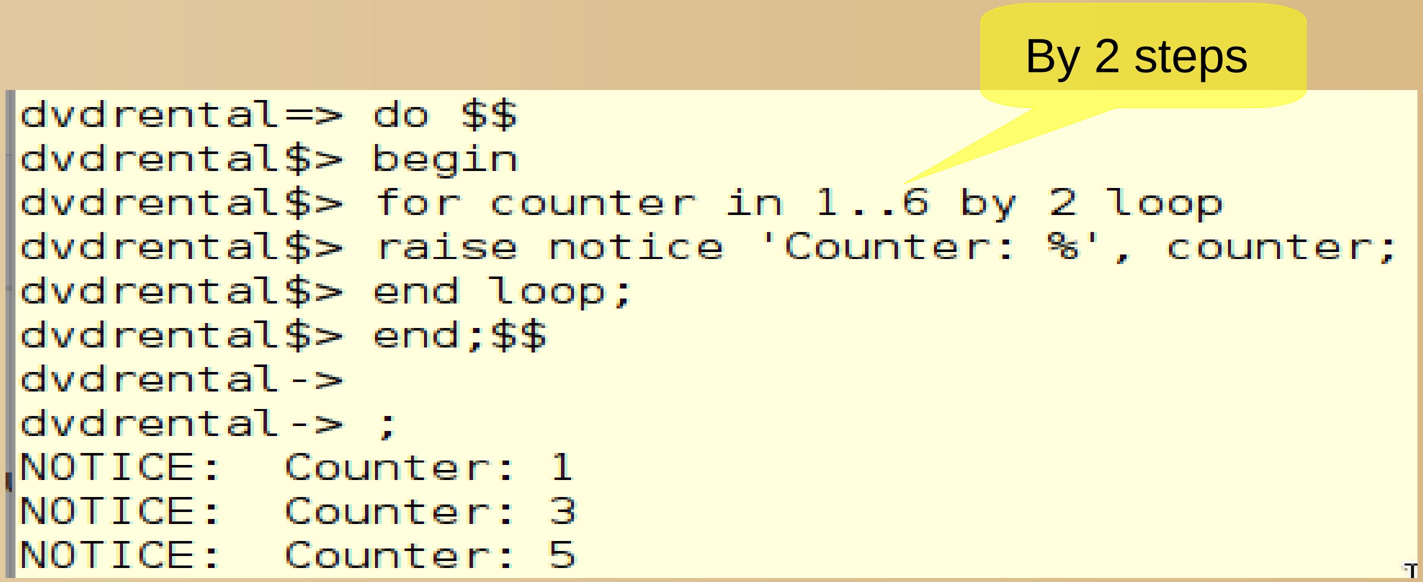


```
dvdrental=> do $$
dvdrental$> begin
dvdrental$> for counter in 1..5 loop
dvdrental$> raise notice 'Counter: %',counter;
dvdrental$> end loop;
dvdrental$> end;$$
dvdrental-> ;
NOTICE:  Counter: 1
NOTICE:  Counter: 2
NOTICE:  Counter: 3
NOTICE:  Counter: 4
NOTICE:  Counter: 5
```

Reverse

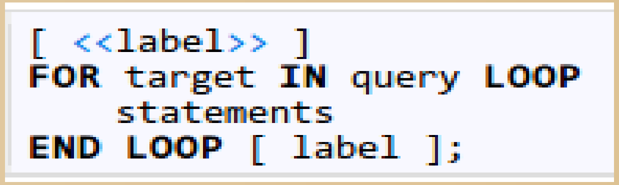


```
dvdrental=> do $$
begin
for counter in reverse 5..1 loop
raise notice 'Counter: %',counter;end loop;end; $$
dvdrental-> ;
NOTICE:  Counter: 5
NOTICE:  Counter: 4
NOTICE:  Counter: 3
NOTICE:  Counter: 2
NOTICE:  Counter: 1
```

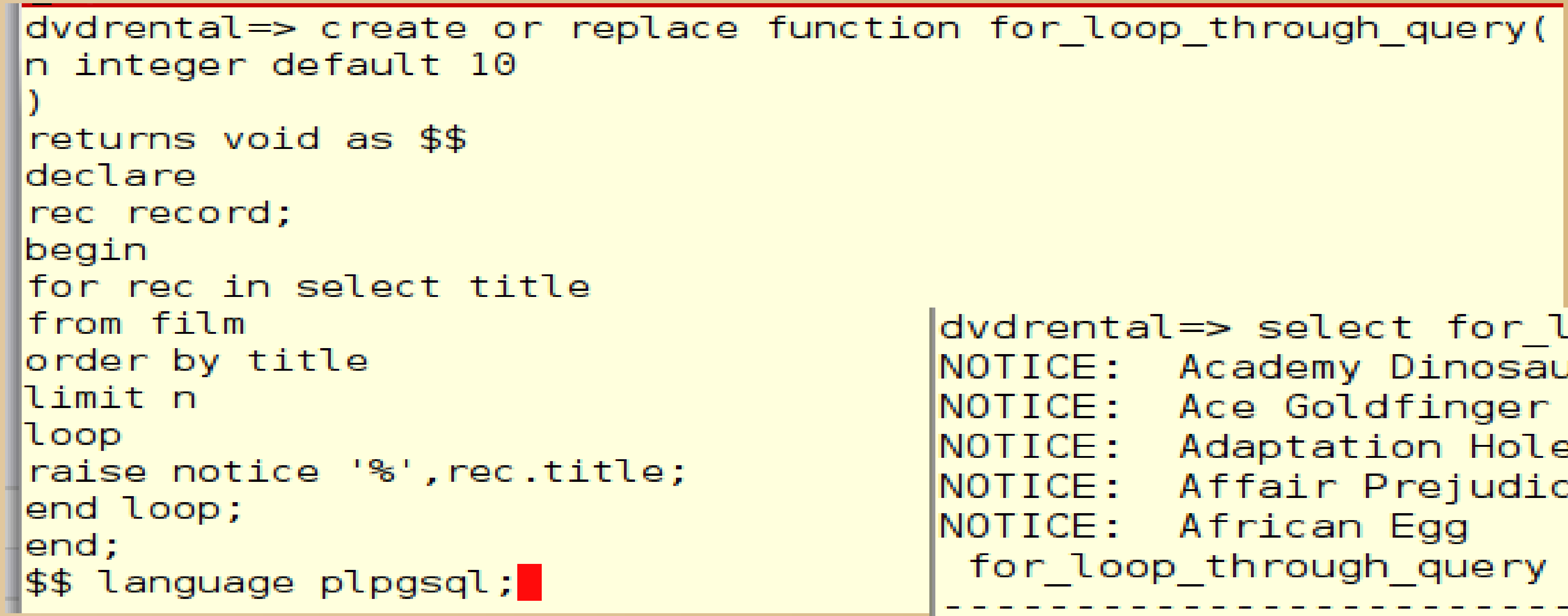


```
dvdrental=> do $$
dvdrental$> begin
dvdrental$> for counter in 1..6 by 2 loop
dvdrental$> raise notice 'Counter: %', counter;
dvdrental$> end loop;
dvdrental$> end;$$
dvdrental->
dvdrental-> ;
NOTICE:  Counter: 1
NOTICE:  Counter: 3
NOTICE:  Counter: 5
```

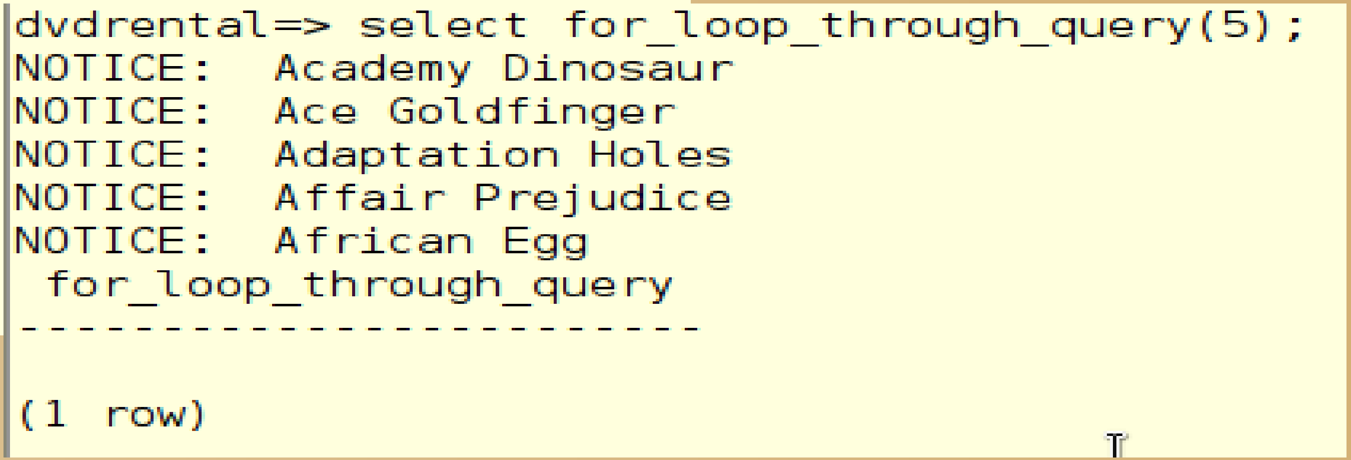
FOR loop for looping through a query result



```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

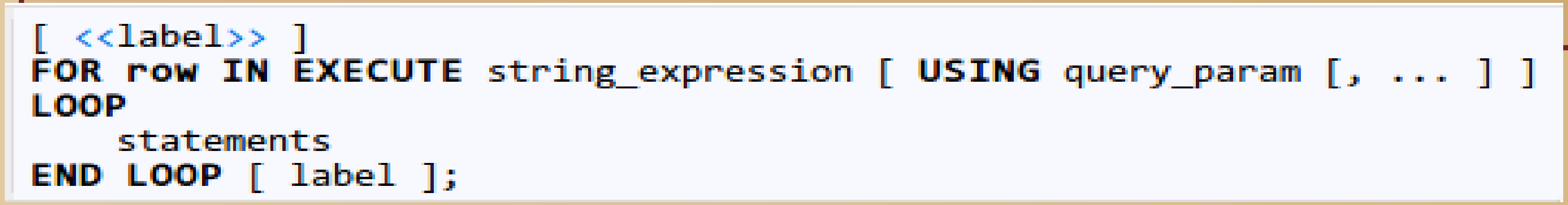


```
dvdrental=> create or replace function for_loop_through_query(
n integer default 10
)
returns void as $$
declare
rec record;
begin
for rec in select title
from film
order by title
limit n
loop
raise notice '%',rec.title;
end loop;
end;
$$ language plpgsql;
```

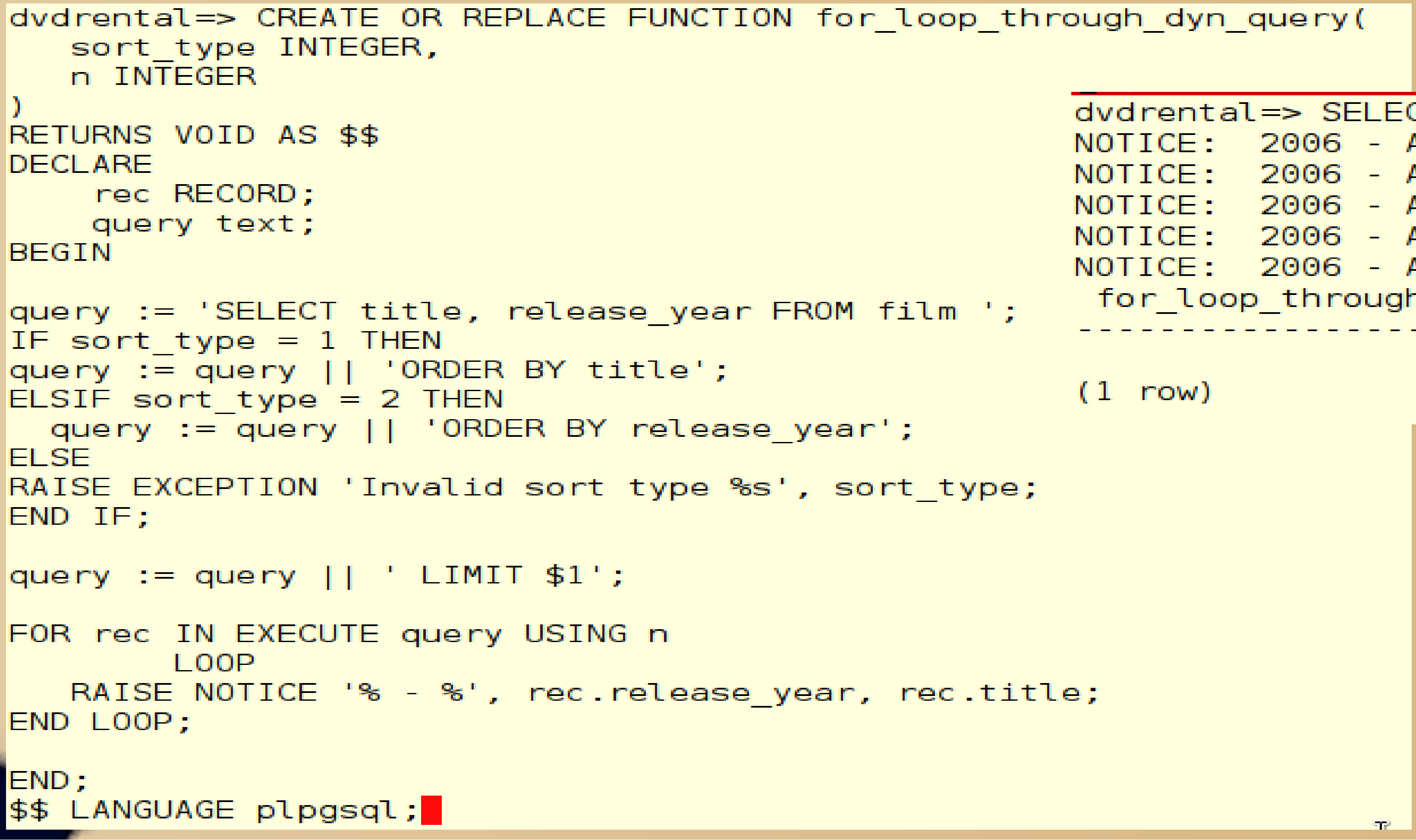


```
dvdrental=> select for_loop_through_query(5);
NOTICE:  Academy Dinosaur
NOTICE:  Ace Goldfinger
NOTICE:  Adaptation Holes
NOTICE:  Affair Prejudice
NOTICE:  African Egg
 for_loop_through_query
------------------------

(1 row)
```

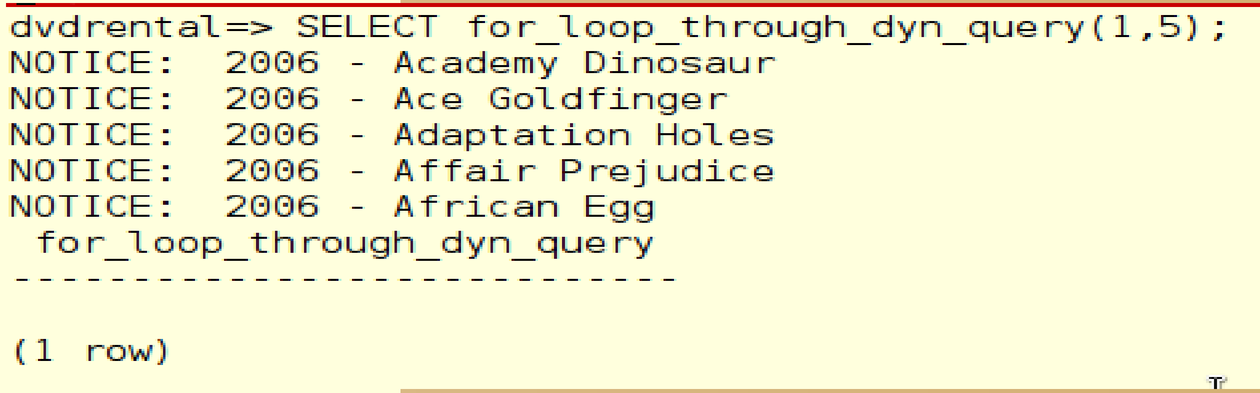
FOR loop for looping through a query result of a dynamic query



```
[ <<label>> ]
FOR row IN EXECUTE string_expression [ USING query_param [, ... ] ]
LOOP
    statements
END LOOP [ label ];
```

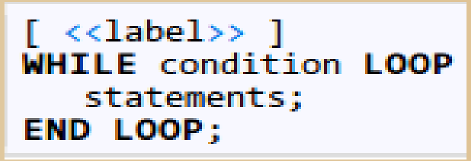


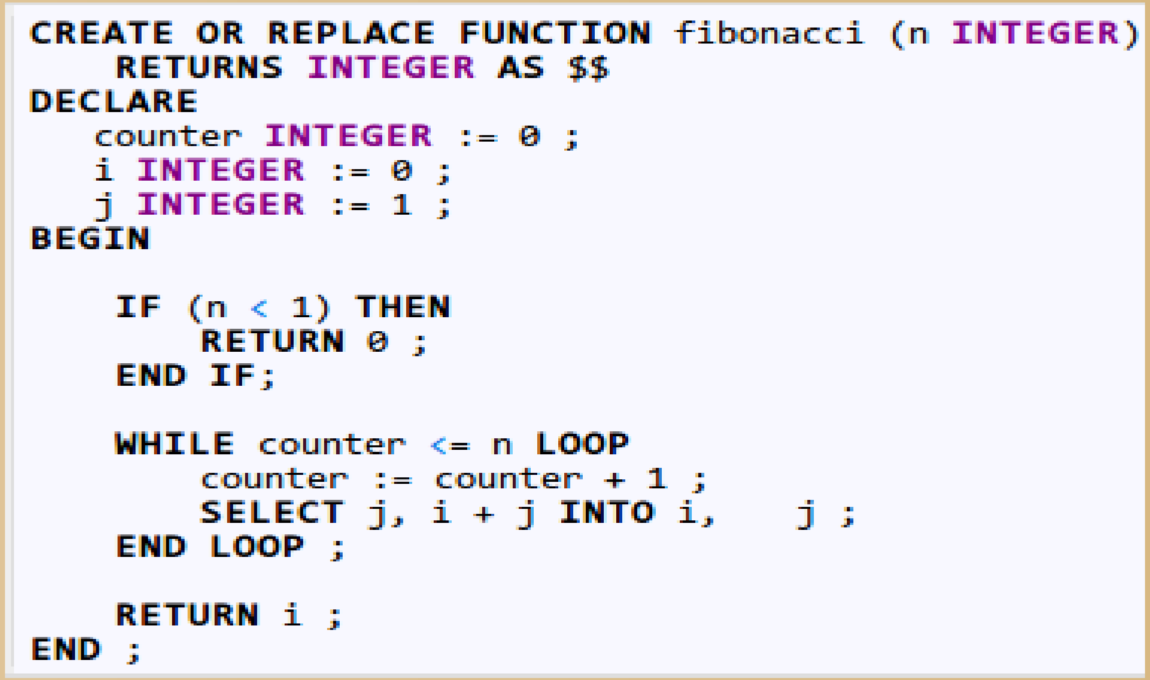
```
dvdrental=> CREATE OR REPLACE FUNCTION for_loop_through_dyn_query(
   sort_type INTEGER,
   n INTEGER
)
RETURNS VOID AS $$
DECLARE
    rec RECORD;
    query text;
BEGIN

query := 'SELECT title, release_year FROM film ';
IF sort_type = 1 THEN
query := query || 'ORDER BY title';
ELSIF sort_type = 2 THEN
  query := query || 'ORDER BY release_year';
ELSE
RAISE EXCEPTION 'Invalid sort type %s', sort_type;
END IF;

query := query || ' LIMIT $1';

FOR rec IN EXECUTE query USING n
        LOOP
   RAISE NOTICE '% - %', rec.release_year, rec.title;
END LOOP;

END;
$$ LANGUAGE plpgsql;
```

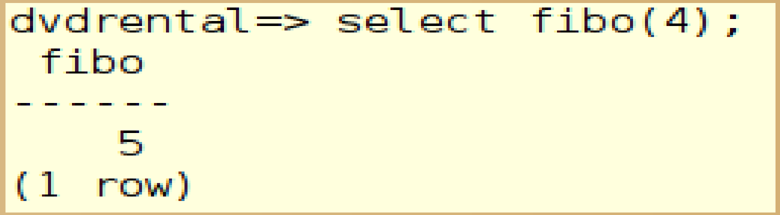


```
dvdrental=> SELECT for_loop_through_dyn_query(1,5);
NOTICE:  2006 - Academy Dinosaur
NOTICE:  2006 - Ace Goldfinger
NOTICE:  2006 - Adaptation Holes
NOTICE:  2006 - Affair Prejudice
NOTICE:  2006 - African Egg
 for_loop_through_dyn_query
----------------------------

(1 row)
```

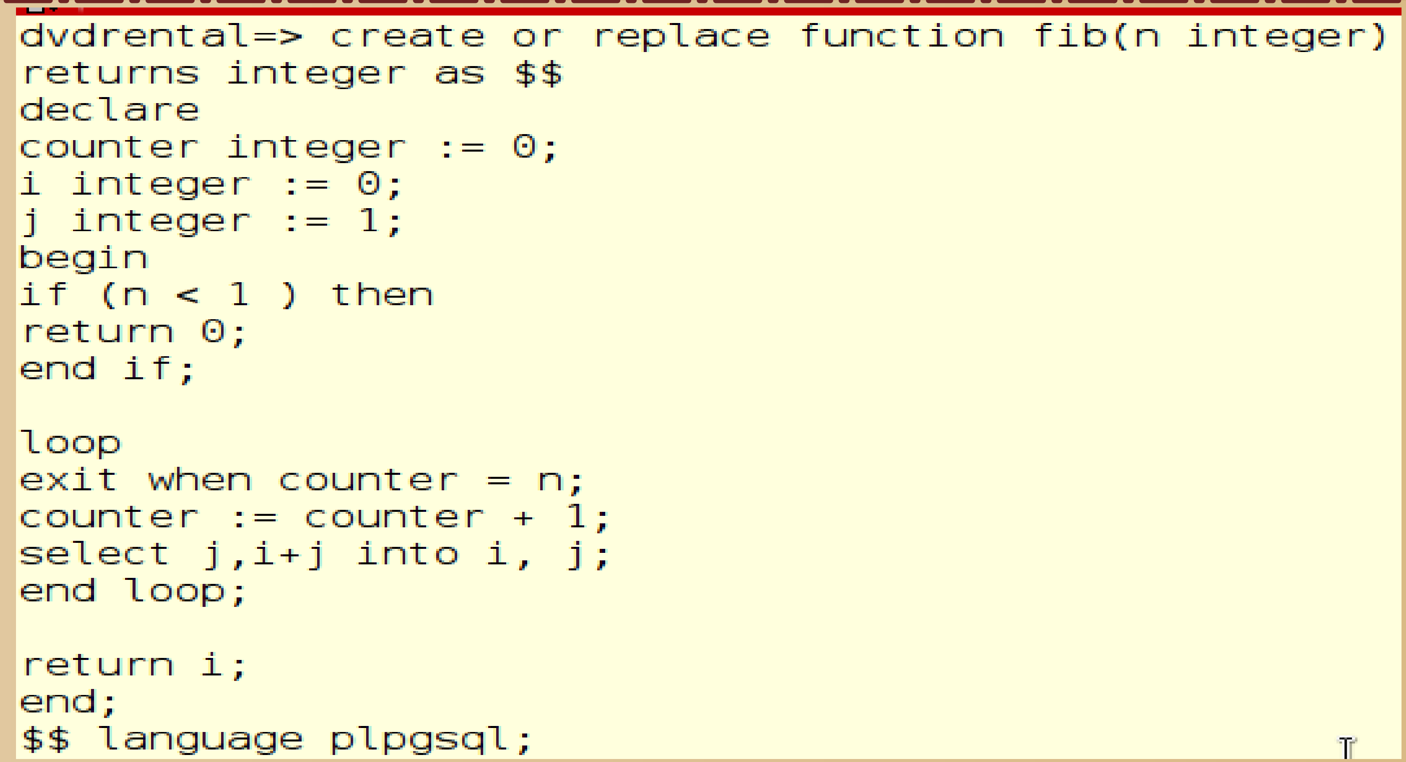
PL/pgSQL WHILE loop


```
[ <<label>> ]
WHILE condition LOOP
   statements;
END LOOP;
```

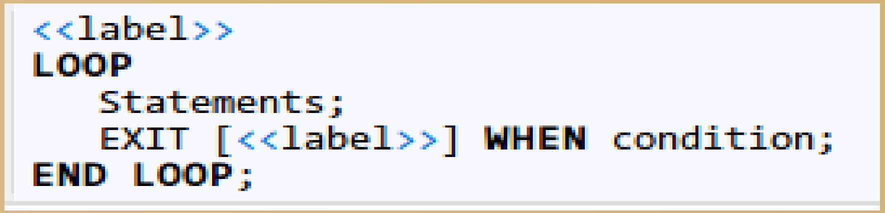

```
CREATE OR REPLACE FUNCTION fibonacci (n INTEGER)
    RETURNS INTEGER AS $$
DECLARE
   counter INTEGER := 0 ;
   i INTEGER := 0 ;
   j INTEGER := 1 ;
BEGIN

    IF (n < 1) THEN
        RETURN 0 ;
    END IF;

    WHILE counter <= n LOOP
        counter := counter + 1 ;
        SELECT j, i + j INTO i,    j ;
    END LOOP ;

    RETURN i ;
END ;
```


```
dvdrental=> select fibo(4);
 fibo
------
    5
(1 row)
```

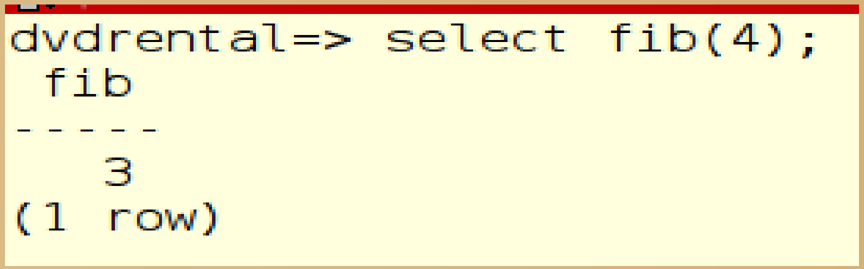
PL/pgSQL LOOP example



```
dvdrental=> create or replace function fib(n integer)
returns integer as $$
declare
counter integer := 0;
i integer := 0;
j integer := 1;
begin
if (n < 1 ) then
return 0;
end if;

loop
exit when counter = n;
counter := counter + 1;
select j,i+j into i, j;
end loop;

return i;
end;
$$ language plpgsql;
```



```
<<label>>
LOOP
   Statements;
   EXIT [<<label>>] WHEN condition;
END LOOP;
```



```
dvdrental=> select fib(4);
 fib
-----
   3
(1 row)
```

Cursors

When to use cursor's

A PL/pgSQL cursor allows us to **encapsulate a query** and process each individual row at a time.

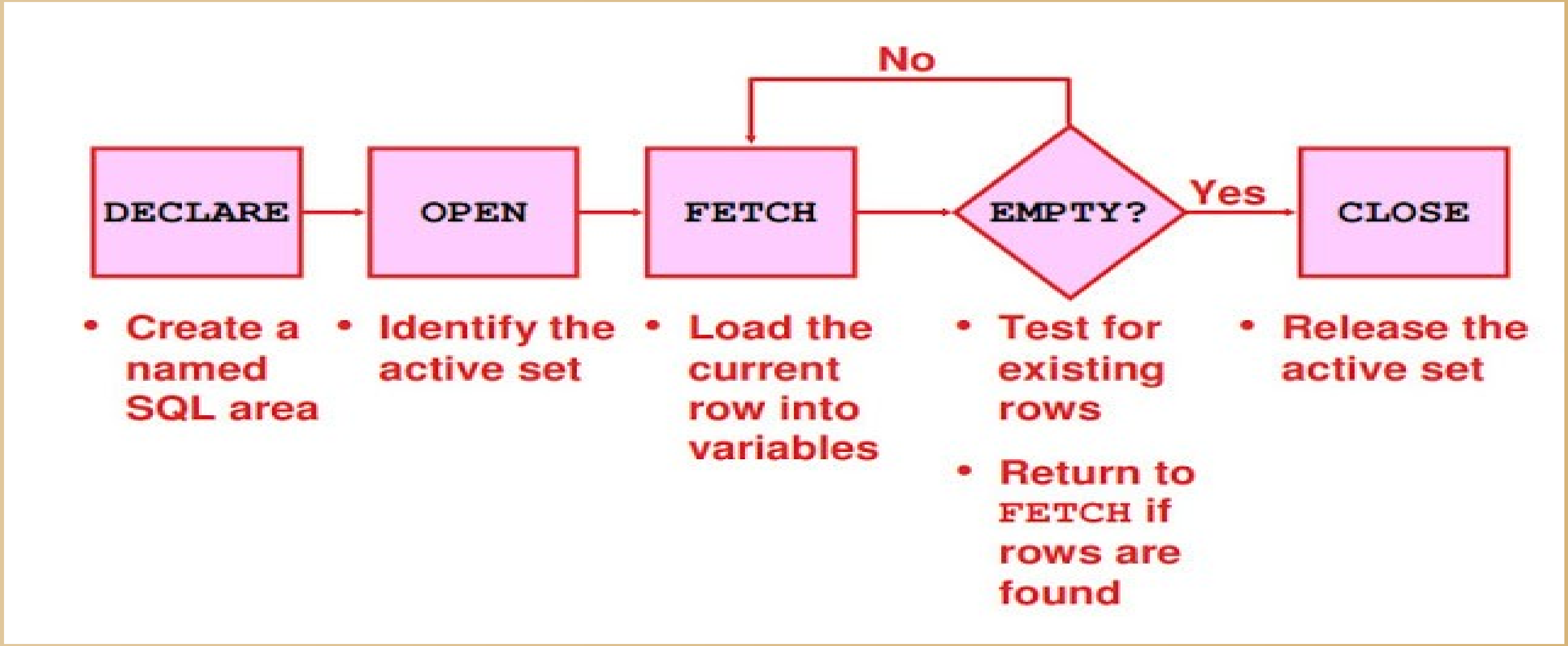
We use cursors when we want to *divide a large result set into parts and process each part individually.*

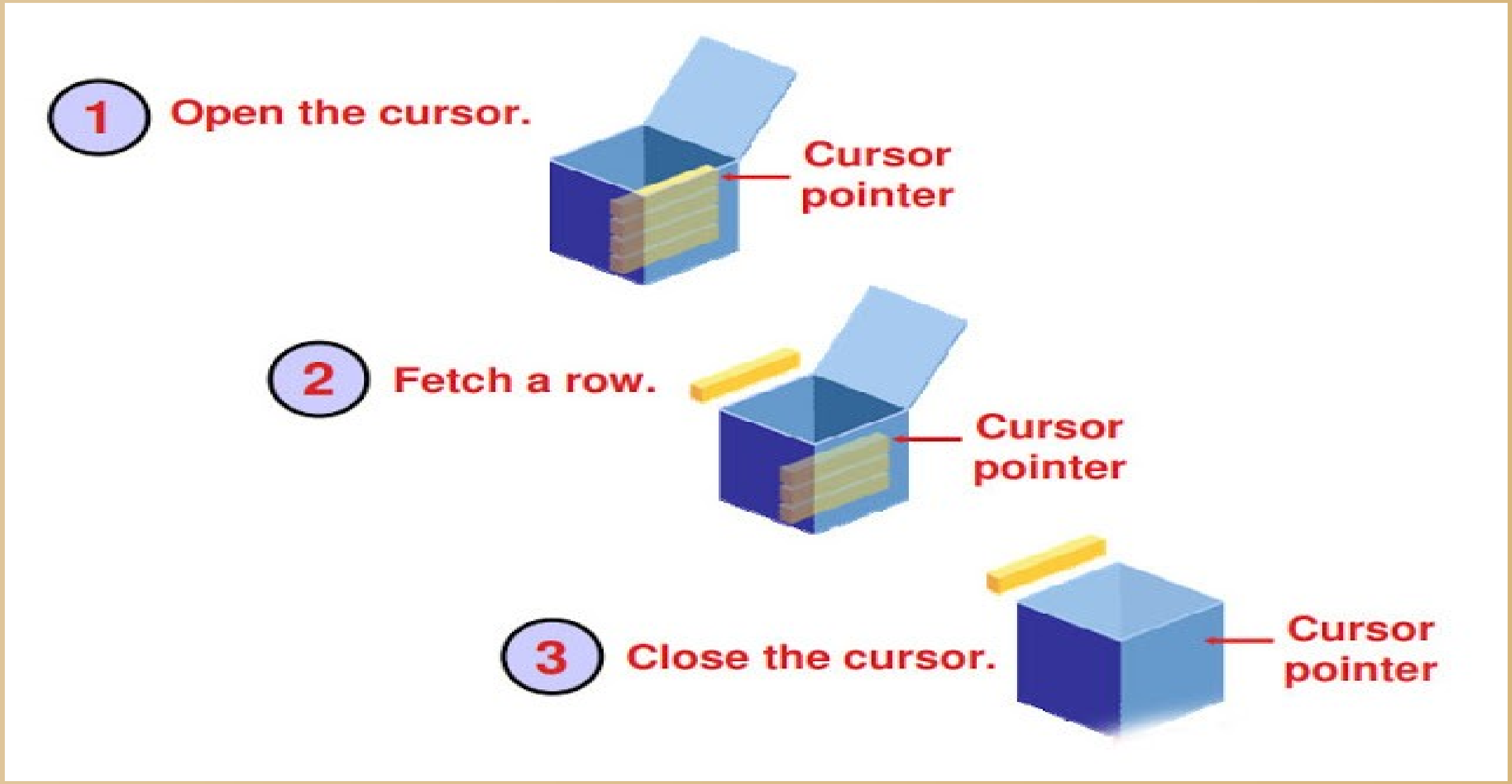
If we process it at once, we may have a memory overflow error.

we can **develop a function** that **returns a reference to a cursor.**

The caller of the function can process the result set based on the cursor reference.

This is an efficient way to return a large result set from a function.


No
DECLARE
OPEN
FETCH
EMPTY?
Yes
CLOSE
• Create a named SQL area
• Identify the active set
• Load the current row into variables
• Test for existing rows
• Return to FETCH if rows are found
• Release the active set


1 Open the cursor.
Cursor pointer
2 Fetch a row.
Cursor pointer
3 Close the cursor.
Cursor pointer

Declaring Cursors

To access to a cursor, you need to declare a cursor variable at the declaration section of a block.

PostgreSQL provides us with a special type called REFCURSOR to declare a cursor variable.

```
DECLARE
      my_cursor REFCURSOR;
```

Another way to declare a cursor that bounds to a query is using the following syntax:

```
cursor_name [ [NO] SCROLL ] CURSOR [( name datatype, name data type, ...)] FOR query;
```

E.g

```
DECLARE
      cur_films  CURSOR FOR SELECT * FROM film;
       cur_films2 CURSOR (year integer) FOR SELECT * FROM film WHERE release_year = year;
```

Opening cursors

Opening unbound cursors

We open an unbound cursor using the following syntax:

```
OPEN unbound_cursor_variable [ [ NO ] SCROLL ] FOR query;
```

Because unbound cursor variable is not bounded to any query when we declared it, we have to specify the query when we open it.

```
OPEN my_cursor FOR SELECT * FROM city WHERE counter = p_country;
```

PostgreSQL allows us to open a cursor and bound it to a dynamic query.

```
OPEN unbound_cursor_variable[ [ NO ] SCROLL ]
FOR EXECUTE query_string [USING expression [, ... ] ];
```

E.g

```
query := 'SELECT * FROM city ORDER BY $1';
OPEN cur_city FOR EXECUTE query USING sort_field;
```

Opening bound cursors

Because a bound cursor already bounds to a query when we declared it, so when we open it, we just need to pass the arguments to the query if necessary.

```
OPEN cursor_variable[ (name:=value,name:=value,...)];
```

We open bound cursors **cur_films** and **cur_films2** that we declared above:

```
OPEN cur_films;

OPEN cur_films2(year:=2005);
```

E.g

Using cursors

After opening a cursor, we can manipulate it using

FETCH, MOVE, UPDATE, or DELETE statement.

Fetching the next row

FETCH [direction { FROM | IN }] cursor_variable INTO target_variable;

FETCH statement gets the **next row f**rom the cursor and **assign it a target_variable,** which could be a :-

record, a row variable, or a comma-separated list of variables.

If no more row found, the target_variable is set to **NULL(s).**

By default, a cursor gets the next row if you don't specify the direction explicitly. The following is the valid for the cursor:

- **NEXT**
- **LAST**
- **PRIOR**
- **FIRST**
- **ABSOLUTE count**
- **RELATIVE count**
- **FORWARD**
- **BACKWARD**

Note that FORWARD and BACKWARD directions are only for cursors declared with SCROLL option.

E.g

```
FETCH cur_films INTO row_film;

FETCH LAST FROM row_film INTO title, release_year;
```

Moving the cursor

MOVE [direction { FROM | IN }] cursor_variable;

If you want to move the cursor only without retrieving any row, you use the MOVE statement. The direction accepts the same value as the FETCH statement.

Eg

```
MOVE cur_films2;

MOVE LAST FROM cur_films;

MOVE RELATIVE -1 FROM cur_films;

MOVE FORWARD 3 FROM cur_films;
```

Deleting or updating row

Once a cursor is positioned, we can delete or update row identifying by the cursor using **DELETE WHERE CURRENT OF** or **UPDATE WHERE CURRENT OF** statement as follows:

```
UPDATE table_name

SET column = value, ... WHERE CURRENT OF cursor_variable;


DELETE FROM table_name

WHERE CURRENT OF cursor_variable;
```

Eg

```
UPDATE film SET release_year = p_year WHERE CURRENT OF cur_films;
```

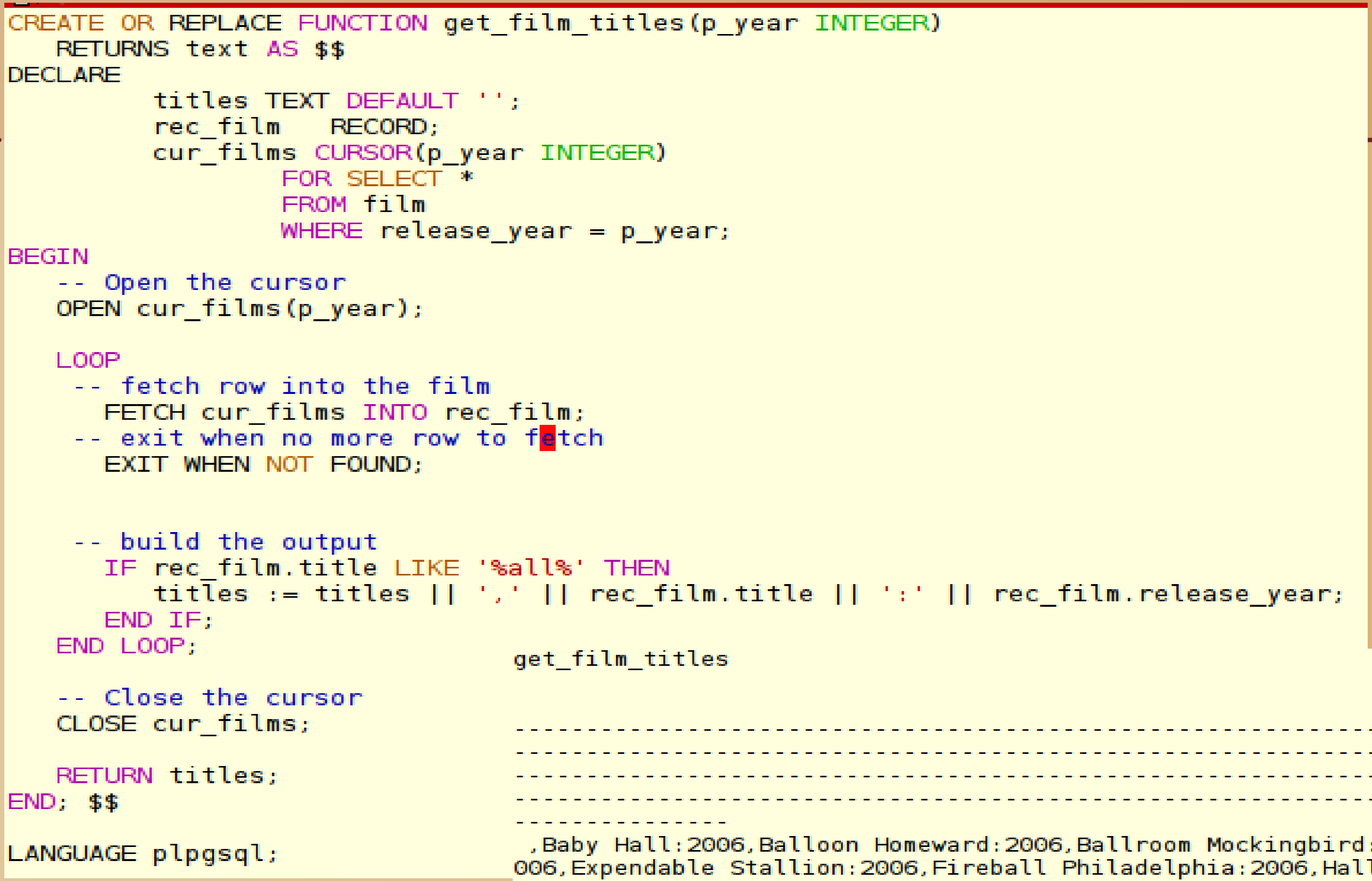
Closing cursors

To close an opening cursor, we use CLOSE statement as follows:

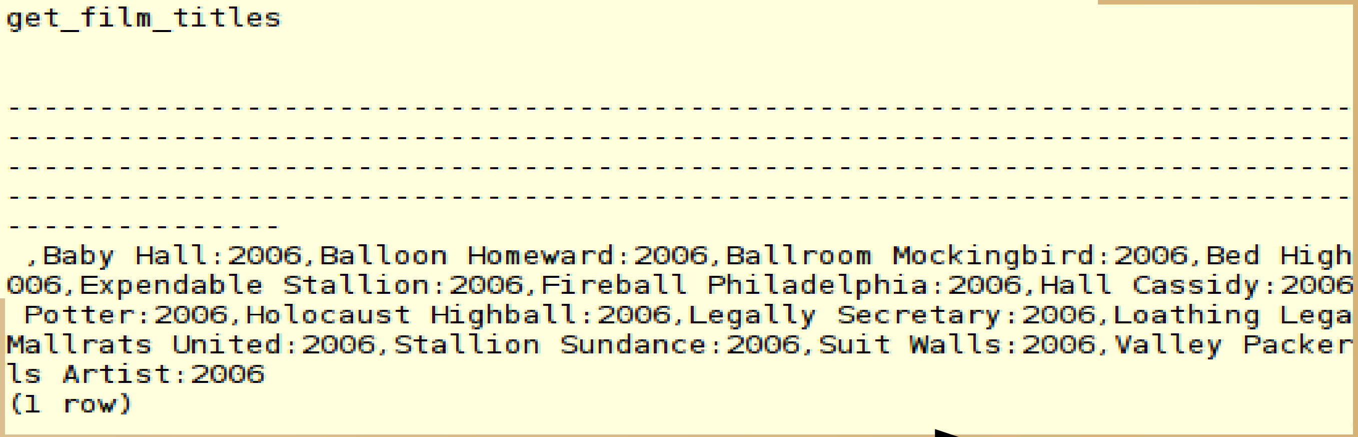
```
CLOSE cursor_variable;
```

The CLOSE statement releases resources or frees up cursor variable to allow it to be opened again using OPEN statement.

What we have built



```
CREATE OR REPLACE FUNCTION get_film_titles(p_year INTEGER)
   RETURNS text AS $$
DECLARE
         titles TEXT DEFAULT '';
         rec_film   RECORD;
         cur_films CURSOR(p_year INTEGER)
                 FOR SELECT *
                 FROM film
                 WHERE release_year = p_year;
BEGIN
   -- Open the cursor
   OPEN cur_films(p_year);

   LOOP
    -- fetch row into the film
      FETCH cur_films INTO rec_film;
    -- exit when no more row to fetch
      EXIT WHEN NOT FOUND;


    -- build the output
      IF rec_film.title LIKE '%all%' THEN
         titles := titles || ',' || rec_film.title || ':' || rec_film.release_year;
      END IF;
   END LOOP;

   -- Close the cursor
   CLOSE cur_films;

   RETURN titles;
END; $$

LANGUAGE plpgsql;
```



```
get_film_titles


-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--------------
 ,Baby Hall:2006,Balloon Homeward:2006,Ballroom Mockingbird:2006,Bed High
006,Expendable Stallion:2006,Fireball Philadelphia:2006,Hall Cassidy:2006
 Potter:2006,Holocaust Highball:2006,Legally Secretary:2006,Loathing Lega
Mallrats United:2006,Stallion Sundance:2006,Suit Walls:2006,Valley Packer
ls Artist:2006
(1 row)
```

Triggers

What is trigger ?

- PostgreSQL trigger is a function invoked automatically whenever an event associated with a table occurs.
- An event could be any of the following: **INSERT**, **UPDATE**, **DELETE** or **TRUNCATE**.

- A trigger is a special user-defined function that binds to a table.

Creating trigger

- To create a new trigger, you must define trigger function first, and then bind this trigger function to a table.

Difference – Trigger & Functions

The difference between a trigger and a user-defined function is that a trigger is automatically invoked when an event occurs.

2 Types - Row & statement level triggers

Differences between the two are how many times the trigger is invoked and at what time.

E.g

if you issue an UPDATE statement that affects 20 rows, the row level trigger will be invoked 20 times, while the statement level trigger will be invoked one time.

When to invoke

We can specify whether the trigger is invoked **before** or **after** an event.

If the trigger is invoked before the event,

- it can skip the operation for the current row or even change the row being updated or inserted.
- In case the trigger is invoked after the event, all changes are available to the trigger.

Usefulness

Triggers are useful in case the database is accessed by various applications, and you want to keep the **cross-functionality within database** that runs automatically whenever the data of the table is modified.

E.g

if you want to keep history of data without requiring application to have logic to check for every event such as **INSERT** or **UDPATE**

You can also use triggers to **maintain complex data integrity rules** which you cannot implement elsewhere except in the database level.

E.g

when a new record is added into customer table, new records must be also created in tables of banks and credits.

Drawback

Main drawback of using trigger is that you must know the trigger exists and understand its logic in order to *figure it out the effects when data changes*.

Postgresql Specifics

- PostgreSQL fires trigger for TRUNCATE.
- PostgreSQL allows you to define **statement-level trigger on views.**
- PostgreSQL requires you to define a user-defined function as the action of the trigger, while the SQL standard allows you to use any number of SQL commands.

Creating the First Trigger in PostgreSQL

To create a new trigger in PostgreSQL you need to:

1) Create a trigger function using CREATE FUNCTION statement.

2) Bind this trigger function to a table using CREATE TRIGGER statement.

Creating the trigger function

A trigger function is similar to an ordinary function, except that it does not take any arguments and has return value type trigger as follows:

```
CREATE FUNCTION trigger_function() RETURN trigger AS
```

<u>**Notice**</u> that you can create trigger functions using any languages supported by PostgreSQL.

Trigger function receives data about their calling environment through a special structure called TriggerData, which contains a set of local variables.

E.g

OLD and NEW represent the states of row in the table before or after the triggering event.

PostgreSQL provides other local variables starting with TG_ as the prefix such as TG_WHEN, TG_TABLE_NAME, etc.

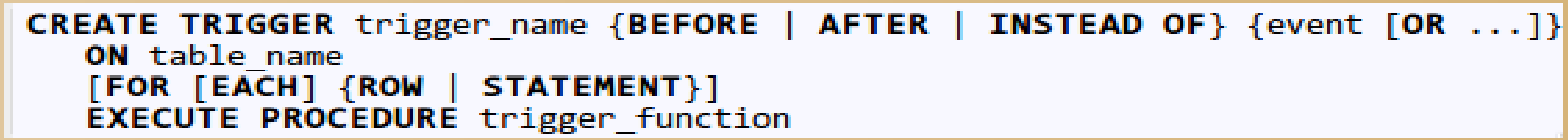
Once the trigger function is defined, you can bind it to specific actions on a table.

Creating the trigger

To create a new trigger, you use the CREATE TRIGGER statement.

The complete syntax of the CREATE TRIGGER is complex with many options.

for the sake of demonstration, we will use the simple form of the CREATE TRIGGER syntax as follows:



```
CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
   ON table_name
   [FOR [EACH] {ROW | STATEMENT}]
   EXECUTE PROCEDURE trigger_function
```

The **event** could be **INSERT**, **UPDATE**, **DELETE** or **TRUNCATE**.

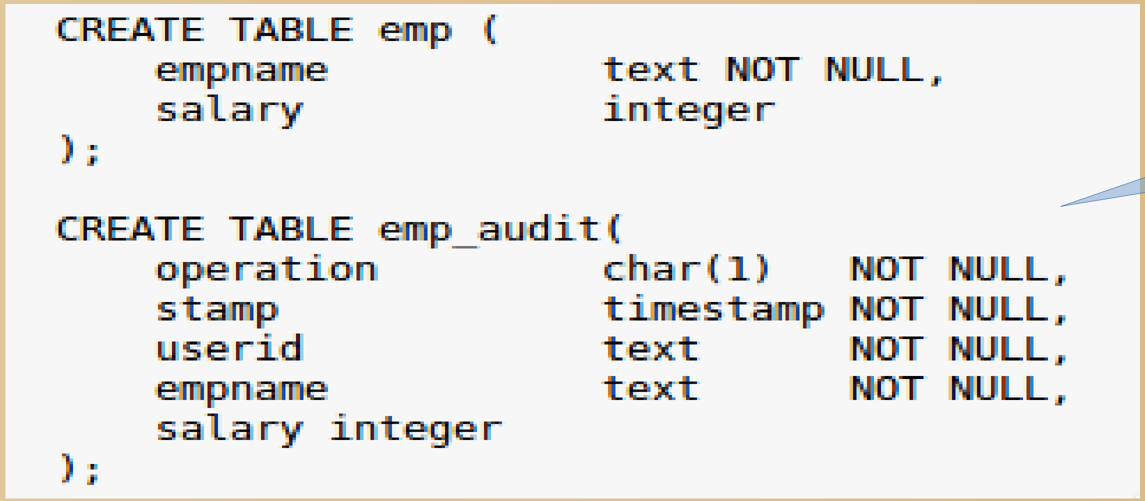
You can define trigger that fires before (BEFORE) or after (AFTER) event.

The INSTEAD OF is used only for INSERT, UPDATE, or DELETE on the views.

PostgreSQL Trigger example

A PL/pgSQL Trigger Procedure For Auditing

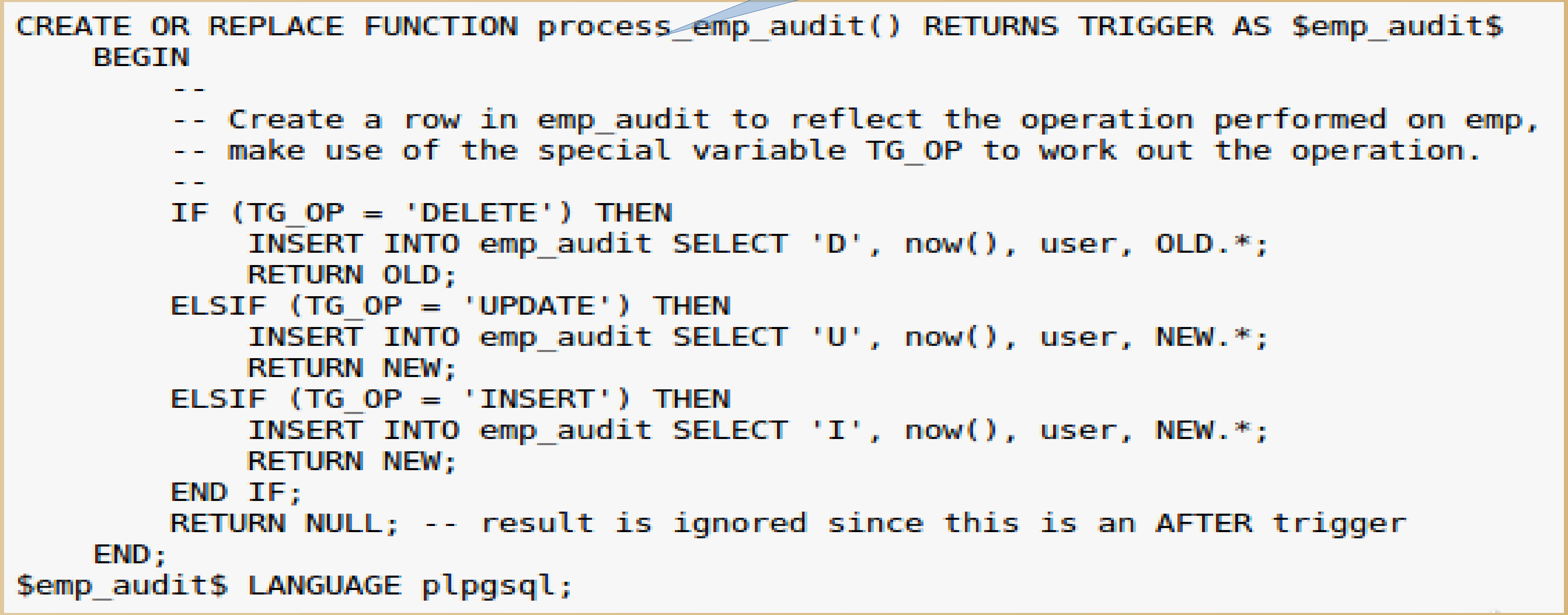
This example trigger ensures that any insert, update or delete of a row in the emp table is recorded (i.e., audited) in the emp_audit table. The current time and user name are stamped into the row, together with the type of operation performed on it.



```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);
```

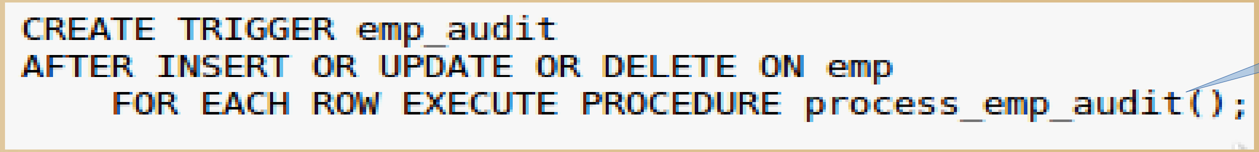
Creating 2 tables
1) emp
2) emp_audit

Process audit function for Record update



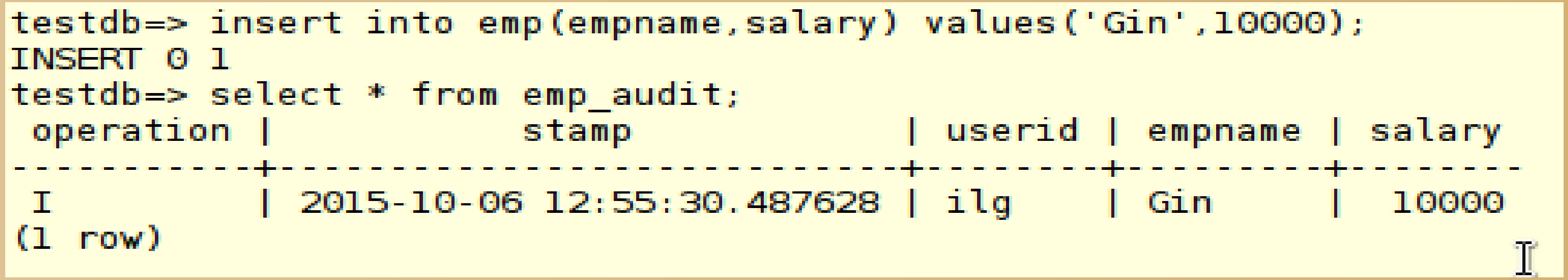
```
CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;
```

trigger

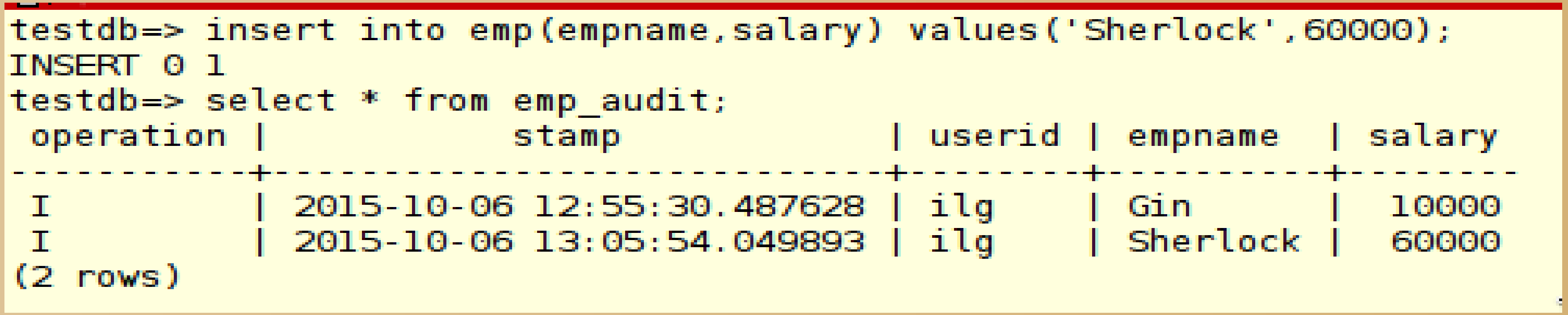


```
CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Execution & Results - Audit trail



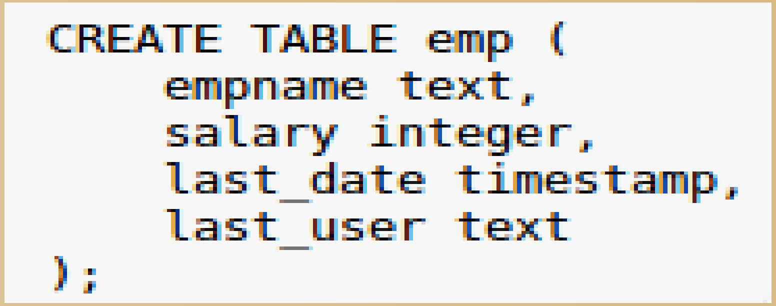
```
testdb=> insert into emp(empname,salary) values('Gin',10000);
INSERT 0 1
testdb=> select * from emp_audit;
 operation |           stamp            | userid | empname | salary
-----------+----------------------------+--------+---------+--------
 I         | 2015-10-06 12:55:30.487628 | ilg    | Gin     |  10000
(1 row)
```



```
testdb=> insert into emp(empname,salary) values('Sherlock',60000);
INSERT 0 1
testdb=> select * from emp_audit;
 operation |           stamp            | userid | empname  | salary
-----------+----------------------------+--------+----------+--------
 I         | 2015-10-06 12:55:30.487628 | ilg    | Gin      |  10000
 I         | 2015-10-06 13:05:54.049893 | ilg    | Sherlock |  60000
(2 rows)
```

Example 2

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it checks that an employee's name is given and that the salary is a positive value.



```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);
```

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

tests

```
testdb=> insert into empl(empname,salary) values('William',-1);
ERROR:  Williamcannot have a negaive salary
testdb=> insert into empl(empname,salary) values('William',0);
INSERT 0 1
```

????
&
Practice