

PL/pgSQL Overview and Programming Structures



Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

PL/pgSQL History

Block Structure and Declaration of Variables

Anonymous Blocks

Routines in PL/pgSQL

Conditional Statements and Loops

Expression Computing

Introduced in PostgreSQL 6.4 in 1998

comes out of the box since PostgreSQL 9.0

Objectives

- create a simple language for custom functions and triggers

- add control structures to the SQL language

- keep the ability to use any custom types, functions, and statements

Inspired by: Oracle PL/SQL, Ada

PL/pgSQL is one of the first procedural languages for PostgreSQL. It first appeared in 1998 in PostgreSQL 6.4, and since 9.0, it has been installed by default when a database is created.

PL/pgSQL extends the SQL functionality, providing variables and cursors, conditional statements, loops, error handling, and other features commonly seen in procedural languages.

PL/pgSQL is based on the Oracle PL/SQL language, which, in turn, is derived from a subset of the Ada language, with its roots going back to Algol and Pascal. Most of the modern programming languages belong to another branch of the C-like languages, that's why PL/pgSQL can at first seem unconventional and excessively verbose (its distinctive feature is using BEGIN and END keywords instead of curly brackets). However, this syntax goes perfectly with the SQL syntax.

<https://postgrespro.com/docs/postgresql/16/plpgsql-overview>

Block Structure



Block label

Variable declarations

- variable scope is a block

- the name can be overridden by a nested block,

- but a variable can still be qualified by a block label

- any SQL types and references to object types (%TYPE) are allowed

Statements

- control structures

- SQL statements, except for the service ones

- nested blocks

Exception handling

4

PL/pgSQL statements are organized into blocks. A block structure comprises several components:

- An optional label that can be used to eliminate naming ambiguities.
- An optional section for *declaration* of local variables and cursors. Any types defined in SQL are allowed. You can also use the %TYPE construct to refer to the type of a table column or other object.
- The main execution section that contains *statements*.
- An optional section for handling *exceptions*.

You can use both PL/pgSQL commands and most of SQL commands as statements, so the two languages are integrated almost seamlessly. Exceptions are SQL service commands, such as VACUUM, which are not allowed, and transaction control commands, such as COMMIT and ROLLBACK, which are allowed only in procedures.

Another (nested) PL/pgSQL block can also be used as a statement.

<https://postgrespro.com/docs/postgresql/16/plpgsql-structure>

<https://postgrespro.com/docs/postgresql/16/plpgsql-declarations#PLPGSQL-DECLARATION-TYPE>

Anonymous Blocks

Ad-hoc execution of procedures

- without creating a stored routine
- with no parameters
- with no return values

The DO command in the SQL language

You can use PL/pgSQL without creating routines. The PL/pgSQL code can be written as an anonymous block and executed using the SQL's DO command.

This command can be used with various server languages, but if you do not specify the language explicitly, it will be assumed that PL/pgSQL is used.

The code of anonymous blocks is not saved on the server. Anonymous blocks do not take arguments or return any values (but there are ways to circumvent that: for example, by using tables).

<https://postgrespro.com/docs/postgresql/16/sql-do>

Anonymous Blocks

A general structure of a PL/pgSQL block:

```
<<label>>
DECLARE
    -- declarations of variables
BEGIN
    -- statements
EXCEPTION
    -- error handling
END label;
```

- All sections except for the statements section are optional.

The smallest block of PL/pgSQL code:

```
=> DO $$
BEGIN
    -- there may be no statements
END
$$;

DO
```

One of the implementations of “Hello, World!”:

```
=> DO $$
DECLARE
    -- This is a single-line comment.
    /* This is a multiline comment.
       Each declaration is ended by a semicolon ';'.
       A semicolon is also placed after each statement.
    */
    foo text;
    bar text := 'World'; -- you can also use = or DEFAULT
BEGIN
    foo := 'Hello'; -- this is an assignment operation
    RAISE NOTICE '%, %!', foo, bar; -- message output
END
$$;

NOTICE: Hello, World!
DO
```

- There must be no semicolon after BEGIN!

Variables can have modifiers:

- CONSTANT — once a variable is initialized, its value must not change;
- NOT NULL — undefined values are not allowed.

```
=> DO $$
DECLARE
    foo integer NOT NULL := 0;
    bar CONSTANT text := 42;
BEGIN
    bar := bar + 1; -- error
END
$$;
```

```
ERROR: variable "bar" is declared CONSTANT
LINE 6:     bar := bar + 1; -- error
          ^
```

Here is an example of nested blocks. A variable in the inner block overrides the one declared in the outer block, but you can refer to any of them using labels:

```
=> DO $$
<<outer_block>>
DECLARE
    foo text := 'Hello';
BEGIN
    <<inner_block>>
    DECLARE
        foo text := 'World';
    BEGIN
        RAISE NOTICE '%, %!', outer_block.foo, inner_block.foo;
        RAISE NOTICE 'An inner variable, without a label: %', foo;
    END inner_block;
END outer_block
$$;
```

NOTICE: Hello, World!

NOTICE: An inner variable, without a label: World

DO

A routine header is language-agnostic

name, input and output parameters

for functions: the return value and volatility category

LANGUAGE plpgsql clause

Returning values

RETURN statement

assigning values to output parameters (INOUT, OUT)

We have already learned about stored functions and procedures, using the SQL language as an example. Most of the information related to creation and management of routines applies to PL/pgSQL routines as well:

- Creating, modifying, and deleting routines
- Location in the system catalog (pg_proc)
- Parameters
- Return value and volatility categories (for functions)
- Overloading and polymorphism
- Etc.

While SQL routines return a value produced by the last SQL command, PL/pgSQL routines either have to assign return values to INOUT or OUT parameters, or use a special RETURN statement (which is available for functions).

PL/pgSQL Routines

Here is an example of a function that returns a value using the RETURN statement:

```
=> CREATE FUNCTION sqr_in(IN a numeric) RETURNS numeric
AS $$
BEGIN
    RETURN a * a;
END
$$ LANGUAGE plpgsql IMMUTABLE;

CREATE FUNCTION
```

Now let's take a look at the same function with the OUT parameter. The return value is assigned to this parameter:

```
=> CREATE FUNCTION sqr_out(IN a numeric, OUT retval numeric)
AS $$
BEGIN
    retval := a * a;
END
$$ LANGUAGE plpgsql IMMUTABLE;

CREATE FUNCTION
```

Here is the same function with the INOUT parameter. This parameter is used for both providing input values and returning the function value:

```
=> CREATE FUNCTION sqr_inout(INOUT a numeric)
AS $$
BEGIN
    a := a * a;
END
$$ LANGUAGE plpgsql IMMUTABLE;

CREATE FUNCTION
```

```
=> SELECT sqr_in(3), sqr_out(3), sqr_inout(3);
```

sqr_in		sqr_out		sqr_inout
9		9		9

(1 row)

Conditional Statements



IF

a regular conditional statement

CASE

similar to CASE in the SQL language, but does not return a value

Note: three-valued logic!

the condition must be true; false and NULL are ignored

9

PL/pgSQL provides two conditional statements: IF and CASE.

The first one is the bread and butter statement available in all languages.

The second one is similar to the SQL CASE construct, but it is a statement and so it does not return a value. It is unlike the `switch` statements in C or Java.

Remember that boolean expressions in SQL (and, consequently, in PL/pgSQL) can take three values: *true*, *false*, and *NULL*. A condition is triggered only when it is *true*, and is not triggered when it is *false* or undefined. This is equally applicable to both WHERE conditions in SQL and conditional statements in PL/pgSQL.

<https://postgrespro.com/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-CONDITIONALS>

Conditional Statements

A generic form of the IF statement:

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

- The ELSIF section can be used several times, or there can be no such section at all.
- There can be no ELSE section.
- The statements corresponding to the first true condition will be executed.
- If none of the conditions is true, the statements of the ELSE section are executed (if available).

Consider an example of a function that uses a conditional statement for decoding an ISBN-10 number. The function returns three values:

```
=> CREATE FUNCTION decode_isbn(
  IN isbn text,
  OUT country text,
  OUT publisher_and_book text,
  OUT check_digit integer
) AS $$
DECLARE
  country_len integer;
BEGIN
  IF left(isbn,1)::integer IN (0,1,2,3,4,5,7) THEN
    country_len := 1;
  ELSIF left(isbn,2)::integer BETWEEN 80 AND 94 THEN
    country_len := 2;
  ELSIF left(isbn,3)::integer BETWEEN 600 AND 649 THEN
    country_len := 3;
  ELSIF left(isbn,3)::integer BETWEEN 950 AND 993 THEN
    country_len := 3;
  ELSIF left(isbn,4)::integer BETWEEN 9940 AND 9989 THEN
    country_len := 4;
  ELSE
    country_len := 5;
  END IF;
  country := left(isbn, country_len);
  publisher_and_book := substr(isbn, country_len+1, 12);
  check_digit := right(isbn, 1);
END
$$ LANGUAGE plpgsql IMMUTABLE;
```

CREATE FUNCTION

```
=> SELECT * FROM decode_isbn('1484268849');
```

country	publisher_and_book	check_digit
1	484268849	9

(1 row)

```
=> SELECT * FROM decode_isbn('8845210669');
```

country	publisher_and_book	check_digit
88	45210669	9

(1 row)

A generic form of the CASE statement (by condition):

```
CASE
  WHEN condition THEN
    -- statements
  ELSE
    -- statements
END CASE;
```

- There can be several WHEN sections.
- There can be no ELSE section.
- The statements corresponding to the first true condition will be executed.
- If none of the conditions is true, ELSE statements are executed (it is an error to have no ELSE in this case).

Usage example:

```
=> DO $$
DECLARE
    country text := (decode_isbn('1484268849')).country;
BEGIN
    CASE
        WHEN country IN ('0','1') THEN
            RAISE NOTICE '% - English-speaking area', country;
        WHEN country = '7' THEN
            RAISE NOTICE '% - Russia', country;
        WHEN country = '88' THEN
            RAISE NOTICE '% - Italy', country;
        ELSE
            RAISE NOTICE '% - Other', country;
    END CASE;
END
$$;

NOTICE:  1 - English-speaking area
DO
```

A generic form of the CASE statement (by expression):

```
CASE expression
    WHEN value, ... THEN
        -- statements
    ELSE
        -- statements
END CASE;
```

- There can be several WHEN sections.
- There can be no ELSE section.
- The statements corresponding to the first true condition “expression = value” will be executed.
- If none of the conditions is true, ELSE statements are executed (it is an error to have no ELSE in this case).

If conditions are similar, this form of the CASE statement can turn out to be shorter:

```
=> DO $$
DECLARE
    country text := (decode_isbn('8845210669')).country;
BEGIN
    CASE country
        WHEN '0', '1' THEN
            RAISE NOTICE '% - English-speaking area', country;
        WHEN '7' THEN
            RAISE NOTICE '% - Russia', country;
        WHEN '88' THEN
            RAISE NOTICE '% - Italy', country;
        ELSE
            RAISE NOTICE '% - Other', country;
    END CASE;
END
$$;

NOTICE:  88 - Italy
DO
```

FOR loop over a range of integers

WHILE loop with a precondition

Infinite loop

Loop can have its own label, just like any block

Loop controls

exit a loop (EXIT)

begin a new iteration (CONTINUE)

For repeated execution of a set of statements, PL/pgSQL offers several types of loops:

- A FOR loop over a range of integers
- A WHILE loop with a precondition
- An infinite loop

A loop is a type of a block; it can have its own label. You can additionally control loop execution by initiating a new iteration or exiting the loop.

<https://postgrespro.com/docs/postgresql/16/plpgsql-control-structures#PLPGSQL-CONTROL-STRUCTURES-LOOPS>

In addition to working with integer ranges, FOR loops can iterate through query results and arrays. Such FOR loops will be discussed later on.

Loops

In PL/pgSQL, all loops have the same structure:

```
LOOP
    -- statements
END LOOP;
```

It can be extended by a header that defines the exit condition for the loop.

A FOR loop over a range is executed while the loop counter goes over the values from bottom to top. Each iteration increases the counter by 1 (but the increment can be changed in the optional BY clause).

```
FOR name IN bottom .. top BY increment
LOOP
    -- statements
END LOOP;
```

- The variable used as a counter is declared implicitly and exists only within the LOOP — END LOOP block.
-

If REVERSE is specified, the counter value is reduced with each iteration, and the top and bottom of the loop have to be swapped:

```
FOR name IN REVERSE top .. bottom BY increment
LOOP
    -- statements
END LOOP;
```

An example of using a FOR loop is a function that reverses a string:

```
=> CREATE FUNCTION reverse_for (line text) RETURNS text
AS $$
DECLARE
    line_length CONSTANT int := length(line);
    retval text := '';
BEGIN
    FOR i IN 1 .. line_length
    LOOP
        retval := substr(line, i, 1) || retval;
    END LOOP;
    RETURN retval;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

CREATE FUNCTION

As you might remember, a STRICT function returns NULL right away if at least one of the input parameters is undefined. The function body is not executed in this case.

A WHILE loop is executed while the condition is true:

```
WHILE condition
LOOP
    -- statements
END LOOP;
```

Here is the same function that reverses a string using a WHILE loop:

```
=> CREATE FUNCTION reverse_while (line text) RETURNS text
AS $$
DECLARE
    line_length CONSTANT int := length(line);
    i int := 1;
    retval text := '';
BEGIN
    WHILE i <= line_length
    LOOP
        retval := substr(line, i, 1) || retval;
        i := i + 1;
    END LOOP;
    RETURN retval;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

CREATE FUNCTION

A LOOP without a header runs infinitely. To terminate it, use the EXIT statement.

EXIT label **WHEN** condition;

- The label is optional; if it is not specified, the innermost loop will be terminated.
 - The WHEN condition is also optional; if it is not specified, the loop is exited unconditionally.
-

LOOP usage example:

```
=> CREATE FUNCTION reverse_loop (line text) RETURNS text
AS $$
DECLARE
    line_length CONSTANT int := length(reverse_loop.line);
    i int := 1;
    retval text := '';
BEGIN
    <<main_loop>>
    LOOP
        EXIT main_loop WHEN i > line_length;
        retval := substr(reverse_loop.line, i,1) || retval;
        i := i + 1;
    END LOOP;
    RETURN retval;
END
$$ LANGUAGE plpgsql IMMUTABLE STRICT;
```

CREATE FUNCTION

- The function body is placed into an implicit block, with the function name used as the block label. So you can access parameters using the “function_name.parameter” notation.
-

Let's make sure that all functions work correctly:

```
=> SELECT reverse_for('AMBULANCE') as "for",
         reverse_while('AMBULANCE') as "while",
         reverse_loop('AMBULANCE') as "loop";
```

```
   for   |   while   |   loop
-----+-----+-----
ECNALUBMA | ECNALUBMA | ECNALUBMA
(1 row)
```

Note: PostgreSQL has a built-in reverse function.

It is sometimes useful to apply the CONTINUE statement, which starts a new iteration of the loop:

```
=> DO $$
DECLARE
    s integer := 0;
BEGIN
    FOR i IN 1 .. 100
    LOOP
        s := s + i;
        CONTINUE WHEN mod(i, 10) != 0;
        RAISE NOTICE 'i = %, s = %', i, s;
    END LOOP;
END
$$;
```

```
NOTICE: i = 10, s = 55
NOTICE: i = 20, s = 210
NOTICE: i = 30, s = 465
NOTICE: i = 40, s = 820
NOTICE: i = 50, s = 1275
NOTICE: i = 60, s = 1830
NOTICE: i = 70, s = 2485
NOTICE: i = 80, s = 3240
NOTICE: i = 90, s = 4095
NOTICE: i = 100, s = 5050
DO
```

Any expression is computed in the context of SQL

- an expression is automatically converted into a query

- the query is prepared

- PL/pgSQL variables are substituted as parameters

Features

- you can use all SQL capabilities, including subqueries

- the execution speed is lower

- although the parsed query (and sometimes the query plan) is cached

- naming ambiguities are an issue

All expressions in PL/pgSQL code are computed as SQL database queries. The interpreter builds the query by composing a `SELECT <expr>` statement and putting parameters in place of PL/pgSQL variables. Next, the statement is prepared (parsed query is cached as usually) and planned. When the statement is executed, specific values are bound to the parameters, and planning is redone (if PostgreSQL has the query plan cached as well, this step may be skipped).

While executing SQL queries impacts PL/pgSQL performance, it ensures close integration with SQL. In fact, expressions can leverage any SQL functionality without limitations, including calling built-in or custom functions, running subqueries, etc.

Starting with PostgreSQL 14, the execution of simple expressions (at least those that do not access any tables) has been optimized: such expressions are processed by the server's parser directly, without using the planner at all.

<https://postgrespro.com/docs/postgresql/16/plpgsql-expressions>

Computing Expressions

Any PL/pgSQL expression is computed using the SQL engine. Thus, PL/pgSQL provides exactly the same features as SQL. For example, since SQL allows using CASE, the same construct will also work in PL/pgSQL code (as an expression; it should not be confused with the CASE ... END CASE statement, which is available only in PL/pgSQL):

```
=> DO $$
BEGIN
    RAISE NOTICE '%', CASE 2+2 WHEN 4 THEN 'Everything is OK' END;
END
$$;

NOTICE: Everything is OK
DO
```

You can also use subqueries in expressions:

```
=> DO $$
BEGIN
    RAISE NOTICE '%', (
        SELECT code
        FROM (VALUES (1, 'One'), (2, 'Two')) t(id, code)
        WHERE id = 1
    );
END
$$;

NOTICE: One
DO
```

Another PL/pgSQL expression computation example: how many string reverse functions did we have in total?

```
=> DO $$
DECLARE
    s integer;
BEGIN
    s := count(*) FROM pg_proc WHERE proname LIKE 'reverse%';
    RAISE NOTICE 'Total "reverse" functions : %', s;
END
$$;

NOTICE: Total "reverse" functions : 4
DO
```

Takeaways



PL/pgSQL is an easy-to-use language that comes with the system by default, integrated with SQL

Managing routines in PL/pgSQL is similar to other languages

DO is an SQL command for executing anonymous blocks

PL/pgSQL variables can use any SQL types

PL/pgSQL supports regular control structures, such as conditional statements and loops

1. Modify the *book_name* function, so that the length of the return value does not exceed 47 characters.
If the book title gets truncated, it must be concluded with an ellipsis.
Check your implementation in SQL and in the application. Add more books with long titles if required.
2. Modify the *book_name* function again, so that an excessively long title gets cut off at the end of a full word.
Check the implementation.

1. For example:

Travels into Several Remote Nations of the World. In Four Parts.
By Lemuel Gulliver, First a Surgeon, and then a Captain of Several
Ships →

→ Travels into Several Remote Nations of the W...

Here are some cases that are worth checking for:

- The title length is less than 47 characters (should not change).
- The title length is exactly 47 characters (should not change).
- The title length is 48 characters (four characters have to be truncated because three dots will be added).

It is recommended to implement and debug a separate function for truncation, and then use it in *book_name*. It is useful for other reasons as well:

- It may come in handy somewhere else.
- Each function will perform exactly one task.

2. For example:

Travels into Several Remote Nations of the World. In Four Parts.
By Lemuel Gulliver, First a Surgeon, and then a Captain of Several
Ships →

→ Travels into Several Remote Nations of the...

Will your implementation work properly if the title consists of a single long word without spaces?

1. Truncating Book Titles

Let's create a more general function that accepts the following parameters: the string to truncate, the maximum length, and the suffix to be used in case of truncation. This does not complicate the code and avoids magic numbers.

```
=> CREATE FUNCTION shorten(  
    s text,  
    max_len integer DEFAULT 47,  
    suffix text DEFAULT '...'  
)  
RETURNS text AS $$  
DECLARE  
    suffix_len integer := length(suffix);  
BEGIN  
    RETURN CASE WHEN length(s) > max_len  
        THEN left(s, max_len - suffix_len) || suffix  
        ELSE s  
    END;  
END  
$$ IMMUTABLE LANGUAGE plpgsql;
```

CREATE FUNCTION

Check the result:

```
=> SELECT shorten(  
    'Travels into Several Remote Nations of the World. In Four Parts. '  
'By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships'  
);
```

```
              shorten  
-----  
Travels into Several Remote Nations of the W...  
(1 row)
```

```
=> SELECT shorten(  
    'Travels into Several Remote Nations of the World. In Four Parts. '  
'By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships',  
    34  
);
```

```
              shorten  
-----  
Travels into Several Remote Nat...  
(1 row)
```

Let's use the created function:

```
=> CREATE OR REPLACE FUNCTION book_name(book_id integer, title text)  
RETURNS text  
STABLE LANGUAGE sql  
BEGIN ATOMIC  
SELECT shorten(book_name.title) ||  
    CASE WHEN (right(shorten(book_name.title), 3) != '...')  
        THEN '. '::text  
        ELSE ''  
    END ||  
    string_agg(  
        author_name(a.last_name, a.first_name, a.middle_name), ', '  
        ORDER BY ash.seq_num  
    )  
FROM authors a  
JOIN authorship ash ON a.author_id = ash.author_id  
WHERE ash.book_id = book_id;  
END;
```

CREATE FUNCTION

2. Truncating Book Titles by Full Words

```

=> CREATE OR REPLACE FUNCTION shorten(
    s text,
    max_len integer DEFAULT 47,
    suffix text DEFAULT '...'
)
RETURNS text
AS $$
DECLARE
    suffix_len integer := length(suffix);
    short text := suffix;
BEGIN
    IF length(s) < max_len THEN
        RETURN s;
    END IF;
    FOR pos in 1 .. least(max_len-suffix_len+1, length(s))
    LOOP
        IF substr(s,pos-1,1) != ' ' AND substr(s,pos,1) = ' ' THEN
            short := left(s, pos-1) || suffix;
        END IF;
    END LOOP;
    RETURN short;
END
$$ IMMUTABLE LANGUAGE plpgsql;

```

CREATE FUNCTION

Check the result:

```

=> SELECT shorten(
    'Travels into Several Remote Nations of the World. In Four Parts. '
    'By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships'
);

```

```

              shorten
-----
Travels into Several Remote Nations of the...
(1 row)

```

```

=> SELECT shorten(
    'Travels into Several Remote Nations of the World. In Four Parts. '
    'By Lemuel Gulliver, First a Surgeon, and then a Captain of Several Ships',
    34
);

```

```

              shorten
-----
Travels into Several Remote...
(1 row)

```

1. Create a PL/pgSQL function that returns a string of random characters of the specified length.
2. The Monty Hall problem.

There are three doors. Behind one of the doors is a prize. The player selects one of the doors. The game host opens one of the two remaining doors that does not contain the prize, and gives the player an opportunity to change the choice — that is, to select the other door from the remaining two.

Should the player switch the choice, or is it better to stick with the initial one?

Task: using PL/pgSQL, calculate the probability of winning for both situations: keeping the initial choice and switching doors.

17

You can first create the *rnd_integer* function that returns a random integer within the specified range. You can use the function for both problems.

For example: `rnd_integer(30, 1000) → 616`

1. In addition to specifying the string length, you can also provide the list of allowed characters as an argument. By default, the list can be all Latin letters, digits, and common special characters. To select random characters from the list, you can use the *rnd_integer* function. The function declaration can look as follows:

```
CREATE FUNCTION rnd_text(
  len int,
  list_of_chars text DEFAULT
  'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_0123456789'
) RETURNS text AS ...
```

Function call example: `rnd_text(10) → 'LjdabF_00J'`

2. You can use an anonymous block for this one.

First, implement one round of the game and see which option won: the initial one or the modified one. You can use `rnd_integer(1, 3)` for setting and guessing the winning door.

Then run the game in a loop for a thousand times or so, counting wins for each strategy. Finally, use `RAISE NOTICE` to display the counter values and determine the optimal choice (or lack thereof).

1. A Random String of a Given Size

```
=> CREATE DATABASE plpgsql_introduction;
```

CREATE DATABASE

```
=> \c plpgsql_introduction
```

You are now connected to database "plpgsql_introduction" as user "student".

First, define an auxiliary function to obtain a random integer within a given range. Such a function is trivial in pure SQL, but here is a PL/pgSQL version instead:

```
=> CREATE FUNCTION rnd_integer(min_value integer, max_value integer)
RETURNS integer
AS $$
DECLARE
    retval integer;
BEGIN
    IF max_value <= min_value THEN
        RETURN NULL;
    END IF;

    retval := floor(
        (max_value+1 - min_value)*random()
    )::integer + min_value;
    RETURN retval;
END
$$ STRICT LANGUAGE plpgsql;
```

CREATE FUNCTION

Test it:

```
=> SELECT rnd_integer(0,1) as "0 - 1",
        rnd_integer(1,365) as "1 - 365",
        rnd_integer(-30,30) as "-30 - +30"
FROM generate_series(1,10);
```

0 - 1	1 - 365	-30 - +30
1	316	-22
1	269	-11
0	15	27
0	118	13
0	142	26
1	153	-11
1	316	-12
1	235	29
0	83	29
0	94	24

(10 rows)

The function ensures an even distribution of random values over the entire range, including boundaries:

```
=> SELECT rnd_value, count(*)
FROM (
    SELECT rnd_integer(1,5) AS rnd_value
    FROM generate_series(1,100_000)
)
GROUP BY rnd_value ORDER BY rnd_value;
```

rnd_value	count
1	19697
2	20007
3	20058
4	20227
5	20011

(5 rows)

Now, start working on a function to obtain a random string of a given size. We will use the rnd_integer function to get a random character from the list.

```
=> CREATE FUNCTION rnd_text(
    len int,

    list_of_chars text DEFAULT 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_0123456789'
) RETURNS text
AS $$
DECLARE
    len_of_list CONSTANT integer := length(list_of_chars);
    i integer;
    retval text := '';
BEGIN
    FOR i IN 1 .. len
    LOOP
        -- add a random character to the string
        retval := retval ||
            substr(list_of_chars, rnd_integer(1,len_of_list),1);
    END LOOP;
    RETURN retval;
END
$$ STRICT LANGUAGE plpgsql;

CREATE FUNCTION
```

Test it:

```
=> SELECT rnd_text(rnd_integer(1,30)) FROM generate_series(1,10);
```

```

      rnd_text
-----
x6N8vbAZYNIIDfvjP
GTb2tmPD2b
MV73VeLiJ5YRkwqNlx10C3NgBMm
xUZnTgBFko5HLIJevxKGZnK4oB1
7EC
XtIsGPVCP9_teegBrsN_nEpvI3
ClQY070vZ58g0q67y2KY
yKKq8FuSC0r7x0
hRmi3z3yVUfZKde
s5BhEGhhr7gWEAYliM4xGg
(10 rows)
```

2. The Monty Hall Problem

To set and guess the door, use `rnd_integer(1,3)`.


```

=> DO $$
DECLARE
    x integer;
    choice integer;
    new_choice integer;
    remove integer;
    total_games integer := 1000;
    old_choice_win_counter integer := 0;
    new_choice_win_counter integer := 0;
BEGIN
    FOR i IN 1 .. total_games
    LOOP
        -- Set the prize door
        x := rnd_integer(1,3);

        -- Player makes a choice
        choice := rnd_integer(1,3);

        -- Remove one wrong door not selected by the player
        FOR i IN 1 .. 3
        LOOP
            IF i NOT IN (x, choice) THEN
                remove := i;
                EXIT;
            END IF;
        END LOOP;

        -- Should the player switch their choice?
        -- Is it better to stay or to switch?

        -- Switch
        FOR i IN 1 .. 3
        LOOP
            IF i NOT IN (remove, choice) THEN
                new_choice := i;
                EXIT;
            END IF;
        END LOOP;

        -- Either the original or the new choice always wins
        IF choice = x THEN
            old_choice_win_counter := old_choice_win_counter + 1;
        ELSIF new_choice = x THEN
            new_choice_win_counter := new_choice_win_counter + 1;
        END IF;
    END LOOP;

    RAISE NOTICE 'The original wins: % out of %',
        old_choice_win_counter, total_games;
    RAISE NOTICE 'The switch wins: % out of %',
        new_choice_win_counter, total_games;
END
$$;

```

NOTICE: The original wins: 325 out of 1000

NOTICE: The switch wins: 675 out of 1000

DO

Initially, we select 1 out of 3, so the chance to win is 1/3. If we switch, the probability switches to the opposite, 2/3.

Therefore, the chance to win doubles when we switch.

```

=> \c postgres

```

You are now connected to database "postgres" as user "student".

```

=> DROP DATABASE plpgsql_introduction;

```

DROP DATABASE