Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

PL/pgSQL history

Block structure and declaration of variables

Anonymous blocks

Functions in PL/pgSQL

Conditional operators and loops

Computing expressions

The History of PL/pgSQL

First appeared in version 6.4 in 1998

- is installed by default since version 9.0

Objectives

- create a simple language for user-defined functions and triggers
- add control structures to the SQL language
- keep the opportunity to use any user-defined types, functions, and operators

Ancestors: Oracle PL/SQL, Ada

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

PL/pgSQL extends the SQL functionality, providing such capabilities of procedural languages as using variables and cursors, conditional operators, loops, error handling, etc.

PL/pgSQL is based on the Oracle PL/SQL language, which, in its turn, is derived from a subset of the Ada language. This branch stems from such languages as 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 unusual and excessively verbose (its distinctive feature is using `BEGIN` and `END` keywords instead of curly brackets). Yet its syntax goes well with SQL.

https://postgrespro.com/docs/postgresql/12/plpgsql-overview

Block Structure

A block label

Declaration of variables

- the lifetime of a variable is limited to a block
- the visibility scope can be overridden by a nested block, but a variable can still be referenced by a block label
- any SQL types, references to object types (`%TYPE`) are allowed

Operators

- Control structures
- SQL operators, except for the service ones

Handling exceptions

PL/pgSQL operators are organized into blocks. We can single out the following components in the block structure:

- An optional label that can be used to eliminate naming ambiguities.

- An optional section for *declaration* of local variables and cursors. You can use any types defined in SQL. It is also possible to refer to the type of a table column using the %TYPE construct.

- The main execution section that contains *operators*.

- An optional section for *handling exceptions*.

You can use both PL/pgSQL commands and most of SQL commands as operators, so the two languages are integrated almost seamlessly. It is forbidden to use SQL service commands, such as `VACUUM`. As for transaction control commands (such as `COMMIT` and `ROLLBACK`), they are allowed only in procedures.

A nested PL/pgSQL block can also be used as an operator.

https://postgrespro.com/docs/postgresql/12/plpgsql-structure

https://postgrespro.com/docs/postgresql/12/plpgsql-declarations#PLPGSQL-DECLARATION-TYPE

Anonymous Blocks

One-time execution of procedures

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

`DO` operator of the SQL language

It is possible to use PL/pgSQL without creating routines. The PL/pgSQL code can be written as an anonymous block and executed using the `DO` command of the SQL language.

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 allow passing parameters or returning values. But there are indirect ways to achieve the same outcome, e.g., using tables.

https://postgrespro.com/docs/postgresql/12/sql-do

Anonymous Blocks

A general structure of a PL/pgSQL block:

```
<<label>>
DECLARE
    -- declaration of variables
BEGIN
    -- operators
EXCEPTION
    -- error handling
END label;
```

- All sections except for the operators' one are optional.

The smallest block of PL/pgSQL code:

```
=> DO $$
BEGIN
    -- there can be no operators
END;
$$;

DO
```

One of the implementations of "Hello, World!":

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

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
```

PL/pgSQL Routines

A routine header does not depend on the language

- name, input and output parameters
- for functions: return value and volatility category

Specifying `LANGUAGE plpgsql`

Returning values

- the `RETURN` operator
- assigning values to the output parameters (`INOUT`, `OUT`)

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

- creating, modifying, and deleting routines
- system catalog location (`pg_proc`)
- parameters
- return value and volatility categories (for functions)
- overloading and polymorphism

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

PL/pgSQL Routines

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

```
=> 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 Operators

`IF`

a regular conditional operator

`CASE`

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

Attention: three-valued logic

a condition must be true; false and `NULL` are ignored

PL/pgSQL provides two conditional operators: `IF` and `CASE`.

The first one is an absolutely standard operator, which is available in all languages.

The second one works similar to `CASE` in SQL, but it's a proper operator that does not return a value. It is somewhat analogous to the switch operator in C or Java.

It's important to always remember that boolean expressions in SQL (and, consequently, in PL/pgSQL) can take *three* values: true, false, and `NULL`. A condition is triggered only if it is true; it won't be triggered if it is false *or undefined*. It is equally applicable to both `WHERE` conditions in SQL and conditional operators in PL/pgSQL.

https://postgrespro.com/docs/postgresql/12/plpgsql-control-structures#PLPGSQL-CONDITIONALS

Conditional Operators

A generic form of the IF operator:

```
IF condition THEN
    -- operators
ELSIF condition THEN
    -- operators
ELSE
    -- operators
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 operators corresponding to the first true condition will be executed.
- If none of the conditions is true, the operators of the ELSE section are executed (if available).

Consider an example of a function that uses a conditional operator 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 operator (by condition):

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

- There can be several WHEN sections.

- There can be no ELSE section.
- The operators corresponding to the first true condition will be executed.
- If none of the conditions is true, ELSE operators 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 operator (by expression):

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

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

If conditions are similar, this form of the CASE operator 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
```

Loops

A `FOR` loop over a range of numbers

A `WHILE` loop with a precondition

An infinite loop

A loop can have its own label, just like any block

Control

- exiting a loop (`EXIT`)
- initiating a new iteration (`CONTINUE`)

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

- a `FOR` loop over a range of numbers

- a `WHILE` loop with a precondition

- an infinite loop

A loop is a special kind of a block; it can have its own label.

You can additionally control loop execution by initiating a new iteration or terminating the loop.

https://postgrespro.com/docs/postgresql/12/plpgsql-control-structures#PLPGSQL-CONTROL-STRUCTURES-LOOPS

In addition to working with ranges of numbers, `FOR` loops can iterate through query results and arrays. These flavors of the `FOR` loop will be discussed in the next lectures.

Loops

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

```
LOOP
    -- operators
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
    -- operators
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
    -- operators
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
    -- operators
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 operator.

```
EXIT label WHEN condition;
```

- The label is optional; if it is not specified, the most inner 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 operator, 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
```

Computing Expressions

Any expression is computed in the context of SQL

- an expression is automatically converted into a query
- a query is getting prepared
- PL/pgSQL variables become implicit parameters

Distinctive features

- you can use all SQL capabilities, including subqueries
- the execution speed is quite low,
  although the parsed query (and sometimes the query plan) is cached
- naming ambiguities have to be taken care of

All expressions that occur in the PL/pgSQL code are computed as SQL queries to the database. The interpreter builds the required query by substituting PL/pgSQL variables with parameters, prepares the operator (while the parsed query is being cached, as it is usually done for prepared operators), and executes it.

Although it's not good for PL/pgSQL performance, it ensures very close integration with SQL. In fact, expressions can use any SQL functionality without limitations, including calling built-in and user-defined functions, running subqueries, etc.

https://postgrespro.com/docs/postgresql/12/plpgsql-expressions

Computing Expressions

Any PL/pgSQL expression is computed using database queries. The easiest way to verify it is to make a mistake and check the message:

```
=> DO $$
BEGIN
    RAISE NOTICE '%', 2 + 'a';
END;
$$;
```

```
ERROR:  invalid input syntax for type integer: "a"
LINE 1: SELECT 2 + 'a'
                   ^
QUERY:  SELECT 2 + 'a'
CONTEXT:  PL/pgSQL function inline_code_block line 3 at RAISE
```

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 operator, 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
```

Summary

PL/pgSQL is installed by default and is integrated with SQL; it is a convenient and easy-to-use language

The process of managing routines in PL/pgSQL is the same as in 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 operators and loops

Practice

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 shortened by a full word.

   Check your implementation.

Task 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:

- This function can be used somewhere else.

- Each function will perform exactly one task.

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 well if the title consists of a single long word without spaces?

Task 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. It won't complicate the code and will allow us to do without magic numbers.

```
=> CREATE OR REPLACE 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
```

Let's 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
AS $$
SELECT shorten(book_name.title) ||
       '. ' ||
       string_agg(
           author_name(a.last_name, a.first_name, a.middle_name), ', '
           ORDER BY ash.seq_num
)
FROM   authors a
       JOIN authorship ash ON a.author_id = ash.author_id
WHERE  ash.book_id = book_name.book_id;
$$ STABLE LANGUAGE sql;
```

```
CREATE FUNCTION
```

Task 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 $$
DECLARE
    suffix_len integer := length(suffix);
    short text := suffix;
    pos integer;
BEGIN
    IF length(s) < max_len THEN
        RETURN s;
    END IF;
    FOR pos in 1 .. least(max_len-suffix_len+1, length(s))
    LOOP
        IF substr(s,pos-1,1) != ' ' AND substr(s,pos,1) = ' ' THEN
            short := left(s, pos-1) || suffix;
        END IF;
    END LOOP;
    RETURN short;
END;
$$ IMMUTABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Let's 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)
```

Practice

1. Create a PL/pgSQL function that returns a string of random characters of the specified length.
2. A shell game problem.

   One of the three shells contains a pea.
   A player selects one of the shells. The operator removes one of the two remaining shells (which must be empty) and gives the player an opportunity to change the choice, i.e., select the other shell from the remaining two.

   Does it make sense to change the choice, or is it better to keep the initial one?

   Assignment: using PL/pgSQL, estimate the probability of the win for both the first and second choices.

You can first create the `rnd_integer` function that returns a random integer within the specified range. This function will be useful for solving both problems.

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

Task 1. Apart from the string length, you can also provide the list of allowed characters as an input parameter. By default, it can be all alphabetic characters, digits, and some other special characters. To select random characters from the list, you can use the `rnd_integer` function. A function declaration can look as follows:

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

An example of the function call: `rnd_text(10) → 'LjdabF_OOJ'`

Task 2. You can use an anonymous block in your solution.

First, you have to develop an individual game run and check which choice has won: the initial or the modified one. For setting and guessing the winning shell you can use `rnd_integer(1,3)`.

Then place the game into a loop and iterate through it, e.g., 1000 times, counting wins for each choice. Finally, use `RAISE NOTICE` to display the counter values and determine the winner (or lack thereof).

Task 1. A Random String of the Specified Length

First, let's declare an auxiliary function that returns a random integer from the specified range. It's easy to write such a function in pure SQL, but here we'll use PL/pgSQL:

```
=> 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
```

Let's check this implementation:

```
=> 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
-------+---------+-----------
     0 |      55 |       -29
     0 |     141 |        -3
     1 |     252 |        21
     0 |       7 |       -19
     0 |     204 |       -25
     0 |     349 |       -15
     0 |      65 |         6
     1 |     336 |         1
     1 |      74 |       -28
     0 |     292 |        -4
(10 rows)
```

The function guarantees uniform distribution of random numbers through the whole range, including boundary values:

```
=> SELECT rnd_value, count(*)
FROM (
    SELECT rnd_integer(1,5) AS rnd_value
    FROM generate_series(1,100000)
) AS t
GROUP BY rnd_value ORDER BY rnd_value;

 rnd_value | count
-----------+-------
         1 | 20002
         2 | 19938
         3 | 19829
         4 | 20233
         5 | 19998
(5 rows)
```

Now we can get down to the function that returns a string of the specified length. We'll 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
```

Check the result:

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

          rnd_text
-----------------------------
 qb7kITIleu71aPvGmu
 B7K_Qaj8F56NfPTqRwn3Q
 oU1eP07AnFLUMQtrksw
 AOpib4IR9B2X9GtkrbdTuyJ
 RZhnMgtltrpgByQHmYzKCVNzkZ
 gm6BMbFsfbs
 fNB11
 f
 g
 c09Edr6jWYxQVfRzO_oyjCp
(10 rows)
```

Task 2. The Shell Game

For setting and guessing the winning shell, we are going to 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
        -- Setting the winning shell
        x := rnd_integer(1,3);

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

        -- We remove one wrong option, other than the player's choice
        FOR i IN 1 .. 3
        LOOP
            IF i NOT IN (x, choice) THEN
                remove := i;
                EXIT;
            END IF;
        END LOOP;

        -- Should the player change the choice?

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

            END IF;
        END LOOP;

        -- Either the initial or the modified choice is bound to win
        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 first choice has won:  % of %',
        old_choice_win_counter, total_games;
    RAISE NOTICE 'The second choice has won: % of %',
        new_choice_win_counter, total_games;
END;
$$;

NOTICE:  The first choice has won:  309 of 1000
NOTICE:  The second choice has won: 691 of 1000
DO
```

At first, we select one shell out of three, so the probability of the win is 1/3. If the choice is changed, the probability changes to 2/3.

Thus, the probability of the win is higher for the new choice. So it makes sense to change the choice.