



# PL/pgSQL

Jesús García



# PL/pgSQL

Jesús García

27/05/2020

# Índice

Programación de bases de datos.....	4
Estructura de una función .....	7
Declaración básica de variables.....	10
Errores y mensajes.....	11
Declaración avanzada de variables .....	13
SELECT INTO.....	13
%TYPE.....	15
%ROWTYPE .....	16
Operadores y expresiones.....	18
Parámetros .....	21
Parámetros de entrada .....	21
Parámetros de salida.....	26
Estructuras de control.....	35
Condicionales.....	35
Bucles .....	37
Conjuntos de datos (NO ENTRA).....	42
Caso práctico I.....	44
Caso práctico II.....	46
Ejercicios adicionales.....	48
Bibliografía.....	49

## Programación de bases de datos.

Hasta ahora hemos utilizado el lenguaje SQL para la definición y manipulación de datos. SQL es un lenguaje declarativo. Los lenguajes declarativos permiten expresar qué queremos hacer y no el cómo. En el caso del lenguaje SQL la tarea de determinar cómo ejecutar una sentencia la realiza el sistema gestor de bases de datos quien se encarga de interpretar las sentencias SQL y determinar un plan para su ejecución.

La empresa Oracle comercializó en 1979 la primera base de datos relacional basada en el uso del lenguaje SQL. En los años 80 el lenguaje SQL era suficiente para cubrir las necesidades de los programadores, pero con la evolución de la informática y el aumento de la complejidad de los proyectos hizo falta el uso de nuevos mecanismos para desarrollar pequeños programas o *scripts*.

Con SQL no podemos declarar variables, utilizar estructuras como *if*, *for* o *while*; o controlar excepciones. Estas estructuras son propias de los lenguajes imperativos.

Para solucionar este problema en 1991 Oracle decidió añadir a su sistema gestor de base de datos el soporte de un nuevo lenguaje procedural llamado PL/SQL. Los lenguajes procedurales o procedimentales son lenguajes imperativos que permiten organizar el código en módulos (funciones o procedimientos).

PL/SQL, como cualquier lenguaje imperativo nos permite programar haciendo uso de variables, estructuras de control y operaciones aritméticas, relacionales y lógicas. Al tratarse de un lenguaje procedural nos permite organizar el código en procedimientos y funciones que posteriormente pueden ser llamadas desde otros puntos. Lo que realmente da potencia a este lenguaje es la posibilidad de integrar sentencias SQL dentro de nuestro código.

Otros sistemas gestores de bases de datos relacionales como MySQL, PostgreSQL o SQL Server también ofrecen soporte a este tipo de lenguajes y aunque son muy similares entre ellos existen algunas diferencias.

En el caso de PostgreSQL el lenguaje procedural que utiliza se llama PL/pgSQL y es muy similar a PL/SQL.

Hasta ahora, para realizar una consulta a la base de datos simplemente escribimos la sentencia SQL y la ejecutamos. El problema surge cuando queremos realizar una consulta compleja frecuentemente, ya que tendríamos que escribirla de nuevo una y otra vez. Con el uso de PL/pgSQL podemos crear funciones que se almacenarán en la propia base de datos y que contendrán las sentencias SQL que queramos. De esta forma para ejecutar la consulta solo tendremos que llamar a la función. El uso de funciones permite reutilizar el código, ya que cualquier programa podrá hacer uso de las funciones que definamos.

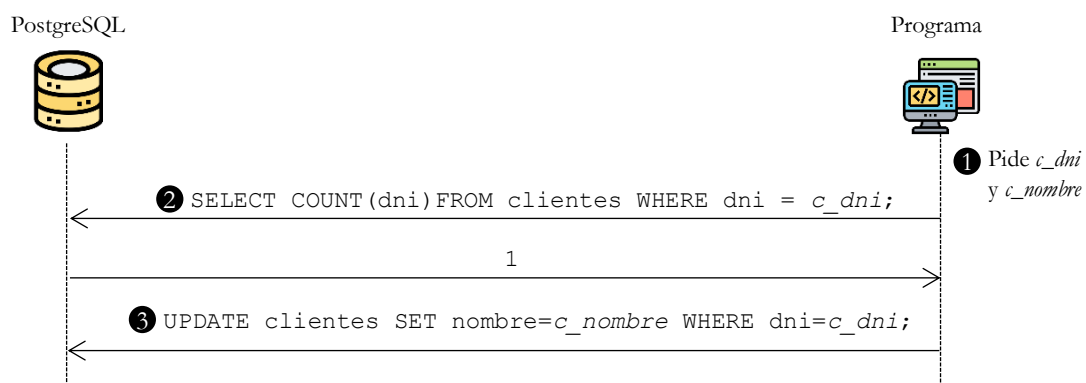
Mediante el uso de funciones también podemos reducir la sobrecarga en la red en los casos en los que los clientes realicen muchas consultas a la base de datos. En vez de hacer que el cliente envíe todas las consultas a la base de datos, podemos hacer que el cliente únicamente tenga que llamar a una función que gestione todas las consultas.

Para ilustrar esto pondremos un ejemplo: imaginemos que tenemos una base de datos muy simple que únicamente tiene una tabla llamada *clientes* con dos columnas: *DNI* y *nombre*. Se quiere desarrollar un programa en C# o Java que pida por pantalla al usuario el DNI y nombre de un cliente. El programa conectará con la base de datos y si el DNI del cliente ya existe actualizará su

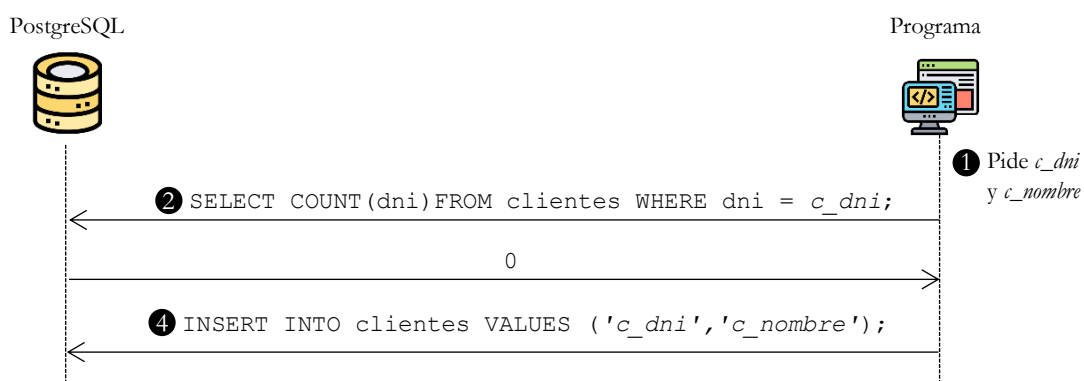
nombre, si no, insertará el nuevo cliente. Los pasos que seguirá nuestro programa serán los siguientes:

1. Pedirá al usuario que escriba el DNI y nombre del cliente y guardará los valores introducidos en las variables *c\_dni* y *c\_nombre*.
2. Realizará una consulta a la base de datos pidiendo el que cuente el número de clientes que existen con el DNI introducido. Si la consulta devuelve un 1 es que el cliente ya existe, si devuelve 0 es que dicho DNI no se ha encontrado.
3. Si el cliente existe enviará una sentencia *update* para actualizar el nombre del cliente.
4. Si no existe enviará una sentencia *insert* para añadir el nuevo cliente.

A continuación, se representa mediante dos diagramas la comunicación que se produce entre nuestro programa y la base de datos. En primer lugar, se muestra el caso en el que el cliente ya existe en la base de datos y se debe realizar una actualización:



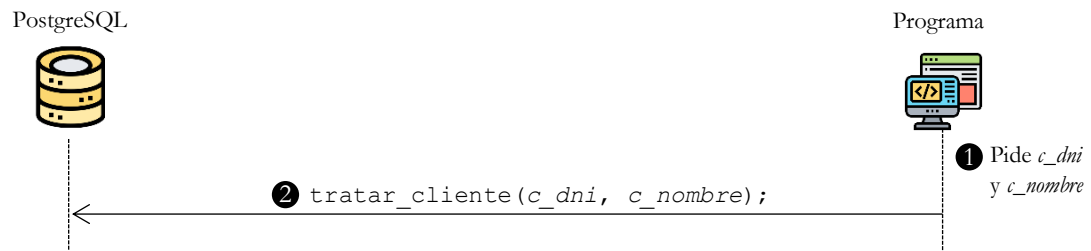
El siguiente diagrama representa la comunicación cuando el cliente no existe en la base de datos y se debe realizar una inserción:



Como vemos, la lógica que determina si se debe hacer un `UPDATE` o un `INSERT` se encuentra en nuestro programa, de forma que independientemente de si existe o no el cliente en la base de datos es necesario realizar siempre dos consultas (un `SELECT` y un `UPDATE` o un `INSERT`). Este proceso se puede simplificar haciendo uso de funciones PL/pgSQL, de forma que la lógica pase a estar en el lado de la base de datos.

Con PL/pgSQL podemos crear una función llamada *tratar\_cliente* que reciba como parámetros el DNI y el nombre del cliente. Esta función tendrá el código necesario para consultar si el cliente ya existe o no y en función de esto realizar una actualización o una inserción. De esta forma nuestro

programa solo tendrá que llamar a la función *tratar\_cliente* pasándole los datos introducidos por el usuario.



De esta forma reduciremos el tráfico de información entre ambas partes ya que nuestro programa únicamente realiza una consulta a la base de datos.

## Estructura de una función

Como ocurre con cualquier lenguaje de programación la mejor forma de empezar su estudio es realizando el programa más básico, el “Hola Mundo”. A continuación, se muestra el código PL/pgSQL que nos permite crear una función que lo único que hará será mostrar por pantalla el mensaje “¡Hola Mundo!”.

```
CREATE OR REPLACE FUNCTION hola_mundo() RETURNS void AS $$  
DECLARE  
    -- Declaración de variables  
BEGIN  
    -- Sentencias SQL y PL/pgSQL  
    RAISE NOTICE '¡Hola Mundo!';  
END;  
$$ LANGUAGE plpgsql;
```

---

*hola\_mundo*

Basándonos en el ejemplo anterior vamos a analizar por partes su contenido.

CREATE OR REPLACE FUNCTION son las cláusulas que se utilizan para indicar que queremos crear o reemplazar una función. Podemos omitir las cláusulas OR REPLACE pero si lo hacemos se producirá un error si intentamos crear una función que ya existe.

A continuación, se escribe el nombre de la función, en el caso del ejemplo *hola\_mundo* y entre paréntesis los parámetros que recibe, en el ejemplo, ninguno.

Finalmente encontramos la cláusula RETURNS irá seguida del tipo de valor que devolverá nuestra función. De nuevo nos encontramos con que la función no devolverá ningún valor ya que se ha utilizado la palabra reservada *void*.

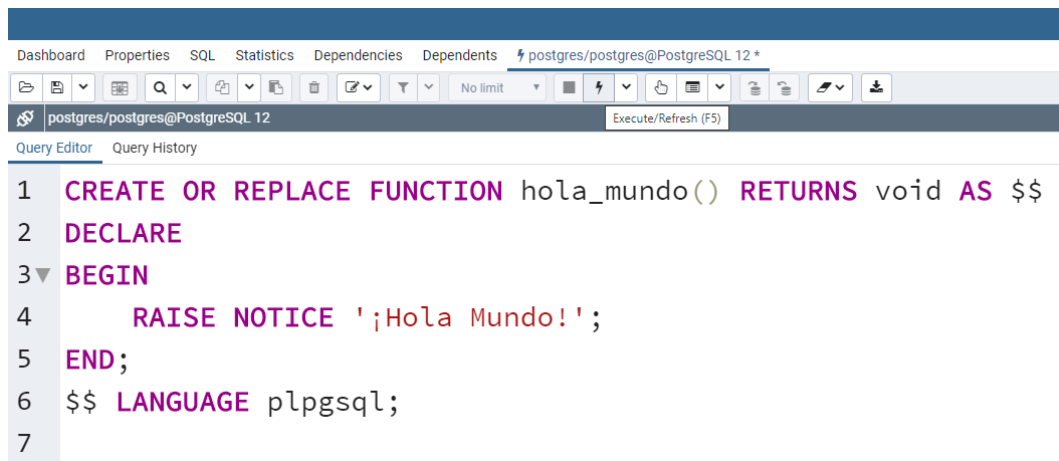
Entre los dos dobles dólares \$\$ escribiremos el cuerpo de la función, dicho de otro modo, lo que debe hacer la función. El cuerpo de la función se divide en dos partes DECLARE y BEGIN – END.

En el apartado DECLARE se definirán todas las variables, constantes, cursores y excepciones que necesite el usuario.

Entre las cláusulas BEGIN – END podremos utilizar tanto sentencias SQL como sentencias PL/pgSQL. En el ejemplo se utiliza la sentencia RAISE para mostrar un mensaje.

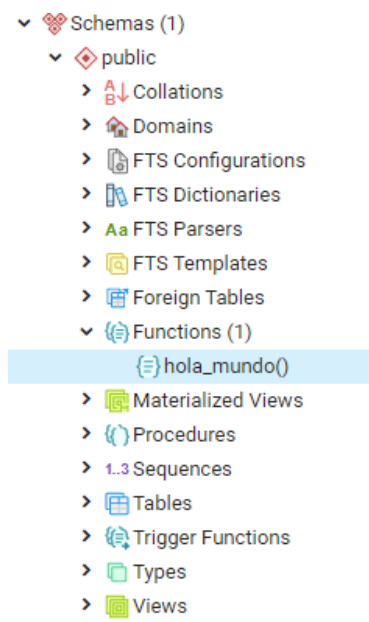
Finalmente, mediante la cláusula LANGUAGE se especifica el lenguaje que se ha utilizado dentro de la función, en este caso plpgsql y es que PL/pgSQL es solo uno de los múltiples lenguajes que podemos utilizar a la hora de crear una función.

Para crear nuestra función simplemente ejecutaremos el código anterior como haríamos con cualquier sentencia SQL. En el caso de utilizar *pgAdmin* abriremos un *Query Tool*, escribiremos la función y pulsaremos el botón de *Ejecutar* o *F5*.



```
1 CREATE OR REPLACE FUNCTION hola_mundo() RETURNS void AS $$
2 DECLARE
3 BEGIN
4     RAISE NOTICE '¡Hola Mundo!';
5 END;
6 $$ LANGUAGE plpgsql;
7
```

Para comprobar que efectivamente se ha creado accederemos al esquema a través del navegador y expandiremos el nodo de funciones, donde encontraremos la función *hola\_mundo*.



La llamada a la función se hace obligatoriamente a través del uso de una sentencia SQL. Por ejemplo:

```
SELECT hola_mundo();
```

O también:

```
SELECT * FROM hola_mundo();
```

Más adelante veremos la diferencia entre ambas opciones y cómo podemos utilizar el resultado devuelto por una función en nuestras consultas SQL.

En cualquier caso, si ejecutamos cualquiera de las dos sentencias anteriores veremos que el resultado obtenido es de 0 filas. Esto es porque los mensajes que escribamos mediante la sentencia `RAISE` se muestran en la consola.

En *pgAdmin* podemos verlos pulsando en la pestaña de mensajes que encontraremos en la parte inferior de la web.



Data Output	Explain	Messages	Notifications
NOTICE: Hola Mundo!			
Successfully run. Total query runtime: 59 msec. 1 rows affected.			

## Declaración básica de variables

Las variables que queramos utilizar en nuestra función deberemos declararlas en la sección DECLARE. La sintaxis para declarar una variable es la siguiente:

```
nombre [ CONSTANT ] tipo [ NOT NULL ] [ = valor ];
```

Donde:

- *nombre* es el identificador de la variable.
- CONSTANT evita que se pueda modificar el valor de la variable tras su inicialización.
- *tipo* es alguno de los tipos válidos que podemos utilizar en PostgreSQL: CHAR, VARCHAR, NUMBER, INT, FLOAT, DATE, etc.
- NOT NULL indica que la variable no puede ser nula y por tanto debe tener un valor.
- = es el operador que nos permite asignar un *valor* inicial a la variable.

### Sintaxis

A lo largo del documento aparecerá la sintaxis de diversas operaciones y estructuras especificada con el formato anterior. Es importante saber interpretarla:

- Las cursivas significan que el programador puede escribir aquí lo que quiera, siempre y cuando sea válido. Por ejemplo, *nombre* y *valor* están en cursiva, eso significa que el programador debe indicar aquí el identificador y el valor que le quiere dar a la variable.
- Los corchetes indican que esa parte de la sentencia es opcional. Por ejemplo, la cláusula CONSTANT es opcional ponerla.
- Las letras en mayúsculas significan que es una palabra clave que se tiene que escribir tal cual. Por ejemplo, si decidimos utilizar la cláusula CONSTANT la escribiremos tal cual.

Algunas declaraciones de variables válidas serían:

```
fecha_nac DATE;  
num_depto NUMERIC(2) NOT NULL = 10;  
localidad_cliente VARCHAR(13) = 'Alcorcón';  
fijo_comision CONSTANT NUMERIC = 500;  
PI CONSTANT NUMERIC(5,4) = 3.1416;
```

## Errores y mensajes

Como hemos visto anteriormente la sentencia `RAISE` nos permite mostrar mensajes por pantalla. En este apartado veremos más detalladamente cómo utilizarla. En primer lugar, analizaremos su sintaxis:

```
RAISE [nivel] 'formato' [, expresión [, ... ]]
```

Donde:

- *nivel* indica el nivel de importancia del mensaje. Posibles valores son `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` y `EXCEPTION`. Elijamos el nivel que elijamos el resultado será el mismo: se mostrará el mensaje por pantalla. El nivel únicamente sirve para indicar a quien llame a nuestra función cuál es la importancia del mensaje. Por norma general en este documento se utiliza el nivel `NOTICE`. Sí que hay que tener en cuenta que si se opta por utilizar `EXCEPTION` al mostrarse el mensaje se abortará la transacción, es decir, se detendrá la función.
- *formato* es el mensaje que se mostrará. En él podemos utilizar el símbolo `%` para indicar el lugar dentro del mensaje donde se mostrarán los valores.
- *expresión* son expresiones aritméticas, alfabéticas o simplemente variables cuyo valor se mostrará en el mensaje sustituyendo los símbolos `%` que hayamos indicado en el formato del mensaje.

### Sintaxis

- Los puntos suspensivos indican repetición. En el caso del `RAISE` nos indica que podemos escribir todas las expresiones que queramos siempre y cuando estén separadas por comas.

Veamos un ejemplo:

```
RAISE NOTICE 'Este es el mensaje número % y es %', 1, 'Hola';
```

Al ejecutar esta sentencia se sustituye el primer `%` por la primera expresión (cuyo valor es 1) y el segundo `%` por la segunda expresión (valor 'Hola'). Por eso el mensaje que se mostrará es:

```
NOTICE: Este es el mensaje número 1 y es Hola
```

A continuación, se muestra una función que utiliza todo lo visto hasta ahora. Es recomendable intentar deducir qué mensajes se mostrarán por pantalla tras su ejecución.

```
CREATE OR REPLACE FUNCTION mensajes() RETURNS void AS $$
DECLARE
    iva CONSTANT NUMERIC (2,2) = 0.21;
    cliente VARCHAR (50) = 'Julián';
    precio NUMERIC(100,2) = 34.50;
BEGIN
    RAISE NOTICE '-----';
    RAISE NOTICE 'El cliente % ha comprado un artículo cuyo precio es %',
        cliente, precio;
    RAISE NOTICE 'El iva es %', iva;
```

```

        RAISE NOTICE 'El precio del artículo con iva es %', precio+iva*precio;
        RAISE NOTICE '-----';
END
$$ LANGUAGE plpgsql;

```

---

*mensajes*

Recuerda que una vez creada una función puedes llamarla de las siguientes formas:

```

SELECT mensajes();
SELECT * FROM mensajes();

```

Como se puede ver en el código de la función podemos utilizar RAISE sin porcentajes o utilizar todos los que queramos, los porcentajes serán sustituidos por el resultado de evaluar las expresiones que indiquemos tras las comas. Si el número de porcentajes y el número de expresiones no coincide se producirá un error. Hay que destacar que en el ejemplo también se utiliza una expresión aritmética `precio+iva*precio` como ejemplo de que también podemos realizar operaciones con las variables o literales. El uso de operadores y expresiones viene explicado más detalladamente en el siguiente apartado.



### Ejercicio 1

Crea una función en la que se declaren las variables necesarias para guardar la siguiente información de una persona:

Nombre: Pedro

Altura: 1.93

Fecha de nacimiento: 23/10/1990

Ten en cuenta que la altura no podrá ser nula. Al ejecutar la función se mostrará el valor de las variables por pantalla.

*Resultado esperado*

---

Pedro nació el 1990-10-23 y mide 1.93



## Declaración avanzada de variables

Hasta ahora hemos visto cómo declarar variables, cómo asignarles valores y cómo mostrarlos por pantalla.

En este apartado vamos a ver cómo podemos obtener datos mediante sentencias SELECT y guardarlos en variables. Además, veremos cómo declarar variables haciendo uso de un mecanismo llamado copia de valores que permitirá que nuestras funciones se adapten a los posibles cambios de tipos que se puedan dar en la definición de nuestra base de datos.

### SELECT INTO

En una variable podemos guardar un dato que obtengamos como resultado de ejecutar una sentencia SELECT, para ello utilizaremos la cláusula INTO.

```
CREATE OR REPLACE FUNCTION select_into_1() RETURNS void AS $$
    DECLARE
        nom_cliente VARCHAR (50);
    BEGIN
        SELECT nombre INTO nom_cliente FROM cliente WHERE codigo = 1;
        RAISE NOTICE 'El nombre es %', nom_cliente;
    END
    $$ LANGUAGE plpgsql;
```

---

*select\_into\_1*

El nombre de la variable no puede ser el mismo que el nombre de la columna seleccionada en la SELECT ya que se producirá un error por ambigüedad. Para comprobarlo crearemos la siguiente función donde la columna seleccionada y la variable se llaman *nombre*.

```
CREATE OR REPLACE FUNCTION select_into_2() RETURNS void AS $$
    DECLARE
        nombre VARCHAR (50);
    BEGIN
        --Hay ambigüedad entre nombre y nombre
        SELECT nombre INTO nombre FROM cliente WHERE codigo = 1;
        RAISE NOTICE 'El nombre es %', nombre;
    END
    $$ LANGUAGE plpgsql;
```

---

*select\_into\_2*

La función se creará sin problemas, pero al ejecutarla nos mostrará el siguiente error.

```
ERROR: la referencia a la columna «nombre» es ambigua
LINE 1: SELECT nombre FROM cliente c WHERE codigo = 1
```

Una posible solución es utilizar un nombre diferente para la variable como es el caso del ejemplo *select\_into\_1* o utilizar un alias a la hora de seleccionar la columna:

```
CREATE OR REPLACE FUNCTION select_into_3() RETURNS void AS $$
DECLARE
    nombre VARCHAR (50);
BEGIN
    SELECT c.nombre INTO nombre FROM cliente c WHERE codigo = 1;
    RAISE NOTICE 'El nombre es %', nombre;
END
$$ LANGUAGE plpgsql;
```

---

*select\_into\_3*

Con SELECT INTO podemos obtener más de una columna y guardar los datos en varias variables, por ejemplo:

```
CREATE OR REPLACE FUNCTION select_into_4() RETURNS void AS $$
DECLARE
    nombre VARCHAR (50);
    cp CHAR (5);
BEGIN
    SELECT c.nombre, c.cp
    INTO nombre, cp FROM cliente c WHERE codigo = 1;
    RAISE NOTICE 'El nombre es % y su CP %', nombre, cp;
END
$$ LANGUAGE plpgsql;
```

---

*select\_into\_4*

Hay que destacar que en los ejemplos que estamos viendo hasta ahora las sentencias SELECT solo devuelven una fila ya que estamos obteniendo únicamente el cliente con código 1. Si la SELECT devolviese más de una fila en las variables solo se guardarían los datos de la primera fila. Podéis comprobarlo quitando la cláusula WHERE y su condición. Más adelante veremos cómo trabajar con sentencias SELECT que devuelven más de una fila.

También hay que destacar que todo lo visto hasta ahora sobre consultas podemos seguir utilizándolo, es decir, nuestra SELECT puede utilizar la cláusula GROUP BY, HAVING, JOIN, subconsultas, etc.

En cualquier caso, es importante tener en cuenta que el tipo del dato obtenido por la SELECT y el tipo de la variable donde lo vamos a guardar deben ser el mismo. En los ejemplos anteriores la columna *nombre* de la tabla *cliente* y la variable donde se guardará su valor son siempre del mismo tipo *VARCHAR(50)*.

Imagina que después de un tiempo nos encontramos con un cliente cuyo nombre supera los 50 caracteres. Para poder guardar su nombre en la base de datos necesitaremos cambiar el tipo de la

columna *nombre* para que pase a ser, por ejemplo *VARCHAR(100)*. El problema lo encontraremos al ejecutar nuestra función ya que se mostrará el error:

```
ERROR: el valor es demasiado largo para el tipo character varying(50)
```

Esto se debe a que en nuestra función la variable *nombre* es del tipo *VARCHAR(50)* y al ejecutarla estaríamos intentando guardar un dato de longitud 100 en una variable de longitud 50. La solución más evidente sería cambiar el tipo de la variable en nuestra función para que tenga la misma longitud.

```
nombre VARCHAR (100);
```

¿Pero qué ocurre si el cambio lo tenemos que hacer en 50 funciones diferentes? Sería muy tedioso y para evitarlo PL/pgSQL dispone de un mecanismo llamado copia de tipos que veremos a continuación.

## %TYPE

Para realizar la copia de tipos se utiliza la cláusula *%TYPE* a la hora de declarar una variable. Al utilizar *%TYPE* no necesitas conocer el tipo del dato al que referencias, y lo más importante, si el tipo cambia no necesitarás cambiar nada en la función. Su sintaxis es:

```
variable tabla.columna%TYPE;
```

Donde

- *variable* es el identificador que le queremos dar a la variable.
- *tabla* y *columna* sirve para indicar de qué columna queremos copiar el tipo y en qué tabla se encuentra.

Aplicando lo aprendido al ejemplo *select\_into\_3* podríamos mejorar nuestra función de la siguiente forma:

```
CREATE OR REPLACE FUNCTION select_into_5() RETURNS void AS $$
    DECLARE
        nombre cliente.nombre%TYPE;
    BEGIN
        SELECT c.nombre INTO nombre FROM cliente c WHERE codigo = 1;
        RAISE NOTICE 'El nombre es %', nombre;
    END
$$ LANGUAGE plpgsql;
```

---

*select\_into\_5*

De esta forma la variable *nombre* será del mismo tipo que la columna *nombre* de la tabla *cliente*. Si en algún momento se decide cambiar el tipo de dicha columna no nos tendremos que preocupar de cambiarle el tipo a las variables de nuestras funciones.



## Ejercicio 2

Realiza una función que muestre por pantalla la descripción del artículo con código *ESC00001*.

*Resultado esperado*

La descripción del artículo es: Escalera aluminio 4 peldaños

## Ejercicio 3

Realiza una función que muestre por pantalla el precio del artículo más caro.

*Resultado esperado*

El precio del artículo más caro es: 975.00

## Ejercicio 4

Realiza una función que muestre por pantalla el nombre del municipio con código BRRN y el nombre de la provincia a la que pertenece haciendo uso de una única sentencia SELECT.

*Resultado esperado*

El municipio es Burriana y su provincia Castellón

## Ejercicio 5

Realiza una función que muestre por pantalla cuántos clientes residen en el municipio con código VLNC.

*Resultado esperado*

En total 1 cliente reside(n) en valencia

## %ROWTYPE

¿Qué ocurre si queremos guardar en una variable el resultado de ejecutar una sentencia SELECT que devuelve todas las columnas de una tabla? Podemos declarar una variable compuesta mediante la cláusula *%ROWTYPE*. Esto nos permite que la variable sea una estructura compuesta por campos, cada campo tendrá el nombre de una de las columnas de la tabla y su mismo tipo. La sintaxis que hay que utilizar es la siguiente:

```
variable tabla%ROWTYPE
```

Donde

- *variable* es el identificador que le queremos dar a la variable.
- *tabla* indica la tabla de la que queremos copiar los tipos de las columnas.

En el siguiente ejemplo se declara una variable llamada *cliente* cuyo tipo está compuesto por todas las columnas de la tabla *cliente*:

```
CREATE OR REPLACE FUNCTION select_into_6() RETURNS void AS $$
DECLARE
    cliente cliente%ROWTYPE;
BEGIN
    SELECT * INTO cliente FROM cliente WHERE codigo = 1;
    RAISE NOTICE 'El nombre es %', cliente.nombre;
```



END

```
$$ LANGUAGE plpgsql;
```

*select\_into\_6*

La sentencia SELECT obtiene todos los datos del cliente con código 1 y los guarda en la variable cliente. Finalmente se muestra el nombre del cliente. Para acceder a los diferentes campos de la variable hay que indicar el nombre de la variable seguida de un punto y el nombre de la columna cuyo dato queremos obtener.

```
cliente.nombre;
```



### Ejercicio 6

Crea una función que obtenga todos los datos del tique con código 10 y los guarde en una variable. La función mostrará por pantalla la fecha, código de cliente y código de vendedor del tique.

*Resultado esperado*

```
Fecha: 2019-08-08, Cliente: 25, Vendedor: 1
```

### Ejercicio 7

Crea una función que obtenga todos los datos del tique con código 10 y todos los datos del tique con código 20 y los guarde en dos variables. La función mostrará por pantalla la fecha, código de cliente y código de vendedor de ambos tiques.

*Resultado esperado*

```
Fecha: 2019-08-08, Cliente: 25, Vendedor: 1
```

```
Fecha: 2019-07-11, Cliente: 23, Vendedor: 2
```

## Operadores y expresiones

Para realizar operaciones con las variables y constantes, se utilizan los operadores, fundamentales en los lenguajes de programación.

El principal operador es el de asignación `=`, que utilizaremos para asignar nuevos valores a las variables. Estas asignaciones podemos hacerlas en el bloque `BEGIN END` de la siguiente forma:

```
variable = expresión;
```

Donde *variable* es el identificador de la variable y *expresión* la expresión cuyo resultado de su evaluación se guardará en la variable. Por ejemplo:

```
edad = 18;
```

En la expresión podemos utilizar operadores aritméticos, relacionales y lógicos.

### Operadores aritméticos

<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>^</code>	Suma, resta, multiplicación, división y potencia.
--	---

### Operadores relacionales

<code>=</code> <code>&lt;</code> <code>&gt;</code> <code>&lt;=</code> <code>&gt;=</code> <code>&lt;&gt;</code> <code>!=</code>	Igual, mayor que, menor que, menor o igual que, mayor o igual que, distinto (podemos utilizar tanto <code>&lt;&gt;</code> como <code>!=</code> )
--	--

<code>IS NULL</code> , <code>LIKE</code> , <code>BETWEEN</code> , <code>IN</code>	Operadores relacionales utilizados en SQL: no nulo, como, entre y en.
--	---

### Operadores lógicos

<code>NOT</code> <code>AND</code> <code>OR</code>	Negación, y, o
---	----------------

### Operadores con cadenas

<code>  </code>	Permite concatenar dos cadenas de texto.
-----------------	--

El siguiente ejemplo muestra cómo podemos utilizar los diferentes operadores:

```
CREATE OR REPLACE FUNCTION operadores() RETURNS void AS $$  
    DECLARE  
        nombre VARCHAR(50) = 'Julián';  
        apellido VARCHAR(50) = 'Gutierrez';  
        nombre_completo VARCHAR(101);  
        peso NUMERIC(5,2) = 73.2;  
        estatura NUMERIC(2,1) = 1.70;  
        imc NUMERIC(2,0);  
        peso_valido BOOL;  
  
    BEGIN  
        nombre_completo = nombre || ' ' || apellido;  
        peso_valido = peso BETWEEN 0 AND 100;  
        RAISE NOTICE '¿El peso está entre 0 y 100? %', peso_valido;
```

```

        imc = peso / (estatura ^ 2);

        RAISE NOTICE 'IMC de % es %', nombre_completo, imc;

    END

$$ LANGUAGE plpgsql;

```

---

*operadores*

Al llamar la función anterior se mostrará por pantalla:

```

NOTICE: ¿El peso está entre 0 y 100? t
NOTICE: IMC de Julián Gutierrez es 25

```

Hay que destacar que el resultado de utilizar un operador relacional o lógico es un booleano (es decir, o verdadero o falso). En el ejemplo, la expresión `peso BETWEEN 0 AND 100` da como resultado o cierto o falso y por eso su resultado se guarda en la variable `peso_valido` que es de tipo `BOOL`. Al mostrar el contenido de `peso_valido` en el mensaje aparece la letra `t` (indicando que su valor es *true*). En el caso de que fuese falsa mostraría una `f`. Normalmente los operadores relacionales y lógicos estamos acostumbrados a utilizarlos en la condición de las estructuras de control (*if*, *while*, etc.) pero es importante saber que también se pueden utilizar para guardar su resultado en una variable.

Mencionar por último que la asignación

```
peso_valido = peso BETWEEN 0 AND 100;
```

También se podría haber escrito de la siguiente forma:

```
peso_valido = peso >= 0 AND peso <=100;
```



### Ejercicio 8

Realiza una función que declare dos variables *num1* y *num2* inicializadas con los valores que quieras. La función mostrará por pantalla el resultado de sumar, restar, dividir y multiplicar ambos números.

*Resultado esperado*

---

```

-- Dependerá de num1 y num2.
-- Se muestra el resultado esperado siendo num1=10 y num2=5
Suma: 15
Resta: 5
Multiplicación: 50
División: 2

```

### Ejercicio 9

Realiza una función que obtenga el precio y descuento del artículo CUB00004, los datos se guardarán en dos variables diferentes. Finalmente se mostrará por pantalla el precio del artículo aplicando el descuento.

*Resultado esperado*

---

```
El precio una vez aplicado el descuento es 1.7
```

### Ejercicio 10

Realiza una función que muestre por pantalla la suma de los precios de los artículos SEG00002 y SEG00003.

*Resultado esperado*

---

El precio total es 881.00



# Parámetros

Los parámetros permiten que las funciones se puedan comunicar con el exterior, ya sea para recibir información (parámetros de entrada) o devolverla (parámetros de salida).

## Parámetros de entrada

Los parámetros de entrada se especifican entre los paréntesis de la cabecera de la función. Una función puede recibir múltiples parámetros de entrada, declarándose uno detrás de otro, separados por comas:

```
[nombre] tipo [,...]
```

Donde *nombre* es el identificador del parámetro y *tipo* su tipo de dato. El tipo puede ser un tipo simple o compuesto como veremos a continuación.

### Típos simples

Por tipos simples entendemos los tipos utilizados por SQL (NUMERIC, INT, VARCHAR, CHAR, DATE, etc.) que solo pueden almacenar un tipo de dato.

A continuación, se muestra el código de una función que recibe como parámetros dos números (tipos simples) y muestra el resultado de multiplicarlos:

```
CREATE OR REPLACE FUNCTION multiplica(num1 NUMERIC, num2 NUMERIC)
    RETURNS void AS $$
DECLARE
    res NUMERIC;
BEGIN
    res = num1*num2;
    RAISE NOTICE 'El resultado es %', res;
END
$$ LANGUAGE plpgsql;
```

---

*multiplica*

Como se puede ver podemos utilizar los parámetros dentro de la función ya que no dejan de ser variables cuyo valor nos viene dado desde fuera. Para llamar a la función tendremos que indicar obligatoriamente los valores que queremos pasar como argumentos de la siguiente forma:

```
SELECT multiplica(4,3);
```

Que nos dará como resultado:

```
El resultado es 12
```

A la hora de llamar a la función con la sentencia SELECT no solo podemos pasar valores literales como argumentos si no que podemos pasar valores obtenidos por la propia SELECT.

Por ejemplo, podríamos utilizar la función para calcular el precio total de una línea de un tique. En la tabla *linea\_ticket* existe una columna *cant* que determina el número de artículos que figuran en la línea y una columna *precio* con el precio de dicho artículo. El precio total de la línea no es más que el

resultado de multiplicar la cantidad de artículos por su precio. Podríamos utilizar nuestra función para realizar dicha multiplicación de la siguiente forma:

```
SELECT multiplica(cant,precio) FROM linea_ticket;
```

Al ejecutar la función veremos que muestra muchos mensajes, en realidad, uno por cada línea de la tabla *linea\_ticket*. No se incluyen todos los que aparecen por pantalla ya que ocuparían mucho espacio:

```
El resultado es 5.53
El resultado es 6.00
El resultado es 5.53
El resultado es 8.40
...
```

El motivo por el que se muestran tantos mensajes es que al ejecutar la *SELECT* esta recorre todas las filas de *linea\_ticket* y para cada una de ellas se llama a la función *multiplica*, pasando el valor de la columna cantidad y precio.

Podemos declarar parámetros utilizando la copia de tipos mediante *%TYPE*. A continuación, se muestra el ejemplo de una función que recibe como parámetros de entrada una cantidad y un precio y calcula el precio total:

```
CREATE OR REPLACE FUNCTION calcula_total(cantidad linea_ticket.cant%TYPE,
                                          precio linea_ticket.precio%TYPE)
    RETURNS void AS $$

DECLARE
    res linea_ticket.precio%TYPE;
BEGIN
    res = cantidad*precio;
    RAISE NOTICE 'El resultado es %', res;
END
$$ LANGUAGE plpgsql;
```

---

*calcula\_total\_simple*

En este caso se ha indicado que los parámetros de entrada deben ser del tipo de las columnas *cant* y *precio* respectivamente. También se ha modificado el tipo de la variable *res* para que sea del mismo tipo que la columna *precio*. De este modo, si se modifican los tipos de la tabla, automáticamente la función se adaptará al cambio.

### ***Tipos compuestos***

Los tipos compuestos son aquellos que permiten almacenar en una misma variable diferentes valores, pudiendo ser éstos de tipos diferentes. Hasta ahora hemos visto como mediante *%ROWTYPE* podemos declarar una variable donde almacenar una fila de una tabla, este tipo de variables decimos que es compuesta.

Podemos indicar que el tipo de un parámetro de entrada es una fila (un tipo compuesto por las columnas de una tabla). Para ello, en contra de lo que nos parecería más intuitivo, no se utiliza *%ROWTYPE*, si no que indicamos como tipo el nombre de la tabla. A continuación, se muestra

una modificación del ejemplo *calcula\_total\_simple* para que, en vez de recibir la cantidad y el precio como parámetros, reciba una fila completa de la tabla *linea\_ticket*.

```
CREATE OR REPLACE FUNCTION calcula_total(lt linea_ticket) RETURNS void AS
$$
DECLARE
    res linea_ticket.precio%TYPE;
BEGIN
    res = lt.cant*lt.precio;
    RAISE NOTICE 'El resultado es %', res;
END
$$ LANGUAGE plpgsql;
```

---

*calcula\_total\_compuesta*

Es importante tener claro los cambios que se han hecho sobre la función:

- El parámetro se ha declarado como *lt linea\_ticket*, eso significa que la variable *lt* es una variable compuesta por las columnas de la tabla *linea\_ticket*.
- Para acceder a las columnas de la variable compuesta se utiliza un punto seguido del nombre de la columna a la que queremos acceder. Esto podemos verlo en la línea *lt.cant\*lt.precio*.

Si queremos llamar a la función deberemos de hacerlo de una forma diferente a la vista hasta ahora:

```
SELECT calcula_total(linea_ticket.*) FROM linea_ticket;
```

*linea\_ticket.\** indica que queremos pasar a la función únicamente las columnas correspondientes a *linea\_ticket*. La *SELECT* recorrerá todas las filas de *linea\_ticket* y para cada fila llamará a la función pasándole los valores de todas sus columnas.

Hay que mencionar que si intentásemos ejecutar la siguiente consulta:

```
SELECT calcula_total(*) FROM linea_ticket;
```

Se produciría un error, ya que estamos pasando a la función todas las columnas que devuelve la *SELECT* y puede que la *SELECT* devuelva columnas que no son de *linea\_ticket* porque se haya hecho, por ejemplo, un *JOIN* con otra tabla.

### **Insertión, modificación y actualización**

Hasta ahora solo estamos utilizando consultas *SELECT* dentro de nuestras funciones, pero también podemos utilizar sentencias *INSERT*, *UPDATE* y *DELETE*. Esto tiene sentido sobre todo ahora que sabemos cómo recibir parámetros de entrada.

### **Ejercicios resueltos**

Los siguientes ejercicios se basan en una tabla *emp* para almacenar datos de empleados, la definición de la tabla es la siguiente:

```
CREATE TABLE emp (
    codigo INT,
    nombre TEXT,
```

```
salario NUMERIC,  
edad INT);
```



### Ejercicio resuelto 1

Crea una función que reciba como parámetro el nombre y el salario de un empleado y lo inserte en la tabla *emp*. El código del empleado deberá ser un número que se irá incrementando cada vez que se inserte un nuevo empleado, empezando por 1. Llama a la función para crear un empleado llamado José con un salario de 2300€.

```
CREATE OR REPLACE FUNCTION crea_empleado (nombre emp.nombre%TYPE,  
                                           salario emp.salario%TYPE)  
    RETURNS VOID AS $$  
  
    DECLARE  
        codigo emp.codigo%TYPE;  
  
    BEGIN  
        -- Contamos el número de filas en emp  
        -- y guardamos el resultado en codigo  
        SELECT COUNT(*) INTO codigo FROM emp;  
        -- Incrementamos codigo en 1  
        codigo=codigo+1;  
        -- Guardamos los valores en la base de datos  
        INSERT INTO emp (codigo, nombre, salario)  
            VALUES (codigo, nombre, salario);  
  
    END  
  
    $$ language plpgsql;  
  
    SELECT crea_empleado('José', 2300);
```

### Ejercicio resuelto 2

Crea una función que reciba como parámetro el código de un empleado y su edad. La función modificará la edad del empleado indicado. Llama a la función para modificar la edad de José de forma que pase a ser 20 años.

```
CREATE OR REPLACE FUNCTION actualiza_empleado (c emp.codigo%TYPE,  
                                              e emp.edad%TYPE)  
    RETURNS VOID AS $$  
  
    DECLARE  
        BEGIN  
            UPDATE emp SET edad = e WHERE codigo = c;  
  
        END  
  
    $$ language plpgsql;  
  
    SELECT actualiza_empleado(1, 20);
```



En este caso UPDATE no soporta el uso de alias, por lo que, para evitar la ambigüedad, se ha optado por cambiar el identificador de los parámetros.

### Ejercicio resuelto 3

Crea una función que reciba como parámetro un empleado y muestre por pantalla su código y doble de su salario. Llama la función solo para aquellos empleados con un salario superior a 1000€.

```
CREATE OR REPLACE FUNCTION dobla_salario (empleado emp)
    RETURNS VOID AS $$

    DECLARE
        doble_salario emp.salario%TYPE;
    BEGIN
        doble_salario = empleado.salario * 2;
        RAISE NOTICE 'Empleado: % Sueldo: %', empleado.codigo,
            doble_salario;
    END
    $$ language plpgsql;

SELECT dobla_salario(emp.*) FROM emp WHERE salario > 1000;
```

En este caso la sentencia SELECT obtiene todos los empleados con un salario superior a 1000€ y posteriormente llama a la función tantas veces como filas cumplan la condición.



### Ejercicio 11

Realiza una función que reciba como parámetro el código de un artículo. La función mostrará por pantalla el precio de dicho artículo aplicándole su descuento.

*Resultado esperado*

```
SELECT ejercicio11('SEG00002');
El precio del artículo con código SEG00002 es 654.75
```

### Ejercicio 12

Realiza una función que reciba como parámetro dos códigos de artículos. La función mostrará por pantalla la suma de los precios de ambos artículos.

*Resultado esperado*

```
SELECT ejercicio12('SEG00001', 'SEG00002');
La suma de los precios es 1650.00
```

### Ejercicio 13

Realiza una función que reciba como parámetro el código de un artículo y una cantidad en euros. La función actualizará el precio del artículo sumándole la cantidad indicada.

*Resultado esperado*

Si el precio del artículo con código SEG00001 es 975.0, después de llamar a la siguiente función:

```
SELECT * FROM ejercicio13('SEG00001', 10);
```

Su precio pasará a ser 985.0

### Ejercicio 14

Realiza una función que reciba como parámetro el código de un municipio. La función mostrará por pantalla el número de tickets cuyo vendedor y comprador residen en dicho municipio.

*Resultado esperado*

---

```
SELECT ejercicio14 (codigo) FROM municipio;
```

Par el municipio CSTLL el número de tickets es 3

Par el municipio VLLRL el número de tickets es 3

Par el municipio BRRN el número de tickets es 0

Par el municipio VLLDX el número de tickets es 0

Par el municipio VNRZ el número de tickets es 0

...

### Ejercicio 15

Realiza una función que reciba un código de artículo y un descuento. La función modificará el descuento en aquellas líneas de ticket en las que se venda el artículo indicado.

*Resultado esperado*

---

El artículo de los tickets 74 y 100 es 'SEG00004' y su descuento es del 0%. Tras ejecutar la función:

```
SELECT ejercicio15('SEG00004', 3);
```

El descuento en ambos tickets pasará a ser del 3%.

## Parámetros de salida

Los parámetros de salida permiten que, al llamar a una función, ésta nos devuelva el resultado de su ejecución. Que una función devuelva el resultado de su ejecución nos permite poder “recoger” el dato y procesarlo según nos interese. Podríamos pensar que con RAISE NOTICE ya podemos “devolver” un resultado. En realidad, lo que nos permite RAISE NOTICE es mostrar información por pantalla, quien ha llamado a la función no recibe nada como resultado de su ejecución.

Para intentar aclarar este concepto pondremos un ejemplo. Anteriormente hemos visto cómo utilizar la función *multiplica* para calcular el precio total de una línea.

```
SELECT multiplica(cant,precio) FROM linea_ticket;
```

Recordemos que conforme está implementada la función ésta no devuelve nada, únicamente muestra un mensaje por pantalla con el precio total de cada línea. Si quisiéramos calcular cuál sería la mitad del precio total de cada línea podríamos pensar que es suficiente con realizar una división de la siguiente forma:

```
SELECT multiplica(cant,precio) / 2 FROM linea_ticket;
```

Si ejecutamos la sentencia anterior, veremos que los mensajes que muestra la función son los mismos que antes de añadir la división por 2.

```
El resultado es 5.53
```

```
El resultado es 6.00
```

```
El resultado es 5.53
```

Esto es porque lo que estamos haciendo es dividir entre 2 el valor devuelto por la función y ésta no devuelve nada. Así pues, nuestra función *multiplica* es poco flexible y reutilizable, ya que al no devolvernos el resultado de realizar la multiplicación no podemos utilizarlo para lo que queramos.

### Diferencia entre devolver un resultado y mostrarlo por pantalla

Cuando empezamos a programar cuesta entender la diferencia y la implicación que tiene devolver un resultado o mostrarlo por pantalla. Parece que es lo mismo, pero no. Pudiendo pecar de ser repetitivo es importante aclarar este concepto, porque no solo es importante en base de datos si no en la programación en general.

Mostrar el resultado por pantalla solo sirve para que el usuario pueda verlo. Si únicamente lo mostramos por pantalla estamos decidiendo qué se hace con el dato: mostrarlo por pantalla. Sin embargo, si devolvemos el resultado estamos dando, a quien llame a la función, la posibilidad de utilizar el resultado con el fin que quiera (mostrarlo por pantalla, hacer un cálculo, o lo que sea).

Para que una función devuelva un resultado tenemos dos opciones: utilizar la cláusula RETURNS o utilizar parámetros de salida mediante la cláusula OUT.

### Cláusula RETURNS

Hasta ahora, tras la cláusula RETURNS siempre hemos escrito la palabra clave *void* para indicar que la función no devuelve ningún resultado. Para indicar que queremos devolver un resultado simplemente hay que sustituir *void* por el tipo de datos que queramos. El tipo devuelto puede ser simple (NUMERIC, VARCHAR, etc) o compuesto (una fila de una tabla).

Hay que tener en cuenta que al utilizar la cláusula RETURNS solo podremos devolver un parámetro o valor, aunque éste pueda ser compuesto. Dentro del bloque BEGIN END tendremos que indicar mediante la cláusula RETURN qué variable queremos devolver.

#### Tipos simples

Los tipos son los que ya conocemos y que se han descrito en el apartado de Parámetros de entrada.

Como ejemplo veremos cómo modificar la función *multiplica* para que devuelva el resultado de la multiplicación.

```
CREATE OR REPLACE FUNCTION multiplica_returns(num1 NUMERIC, num2 NUMERIC)
                                RETURNS NUMERIC AS $$

DECLARE
    res NUMERIC;
BEGIN
    res = num1*num2;
    RAISE NOTICE 'El resultado es %', res;
    RETURN res;
END
$$ LANGUAGE plpgsql;
```

---

*multiplica\_returns*

Los cambios realizados son los siguientes:

- Sustituir *void* por NUMERIC como tipo devuelto tras la cláusula RETURNS.
- Añadir la cláusula RETURN seguida de la variable que se quiere devolver.

Si llamamos a la función de la siguiente forma:

```
SELECT multiplica_returns(cant,precio) FROM linea_ticket;
```

Veremos que el resultado de cada multiplicación no se muestra únicamente mediante un mensaje, si no que obtenemos una tabla con los resultados:

	<b>multiplica_returns</b> numeric	
1	5.53	
2	6.00	
3	5.53	
4	8.40	
5	3.96	

Ahora podemos trabajar con el resultado de la función desde nuestra SELECT para, por ejemplo, calcular cuál sería la mitad del precio total de cada línea de un tique.

```
SELECT multiplica_returns(cant,precio) / 2 AS mitad FROM linea_ticket;
```

	<b>mitad</b> numeric	
1	2.7650000000000000	
2	3.0000000000000000	
3	2.7650000000000000	
4	4.2000000000000000	
5	1.9800000000000000	

Nótese que en la SELECT se ha utilizado un alias (*mitad*) para cambiarle el nombre a la columna ya que por defecto le asigna el nombre de la función (*multiplica\_returns*). Hay que recordar que toda SELECT es una consulta en la que podemos seguir utilizando todas las cláusulas vistas hasta ahora (WHERE, JOIN, GROUP BY, HAVING, etc.), el único cambio es que ahora somos capaces de añadir llamadas a funciones definidas por nosotros mismos.

Otro ejemplo de uso de la función sería:

```
SELECT ticket, articulo, multiplica_returns(cant,precio) AS total  
FROM linea_ticket  
WHERE dto > 0;
```

	<b>ticket</b> integer		<b>articulo</b> character (8)		<b>total</b> numeric	
1	3	TEN00002		90.30		
2	5	SEG00002		1350.00		
3	12	CUB00002		190.00		
4	12	CUB00004		12.25		
5	15	SEG00002		675.00		

Nuestra función *multiplica\_returns* devuelve un único valor (el resultado de la multiplicación) pero se llama una vez por cada fila de *línea\_ticket*.

Por último, hay que recordar que también podemos utilizar la copia de tipos %TYPE a la hora de declarar el tipo de valor devuelto.

*Tipos compuestos: filas*

Podemos indicar que el tipo devuelto por una función es una fila (un tipo compuesto por las columnas de una tabla). Como ocurre con los parámetros de entrada no utilizaremos %ROWTYPE tras el RETURNS, si no que indicaremos como tipo devuelto el nombre de la tabla.

A continuación, se muestra un ejemplo de una función que recibe como parámetro el código de un artículo y devuelve todos los datos de dicho artículo:

```
CREATE OR REPLACE FUNCTION devuelve_articulo (cod articulo.codigo%TYPE)
    RETURNS articulo AS $$
DECLARE
    art articulo%ROWTYPE;
BEGIN
    SELECT * INTO art FROM articulo WHERE codigo = cod;
    RETURN art;
END
$$ LANGUAGE plpgsql;
```

*devuelve\_articulo*

Podemos ver que el tipo devuelto es de tipo articulo (RETURNS articulo), es decir, la función va a devolver una variable compuesta que tendrá las columnas de la tabla artículo o, dicho de otro modo, el dato devuelto es una fila de la tabla artículo. Dentro del bloque de declaraciones se declara una variable llamada *art* (art articulo%ROWTYPE;), ahora sí, con %ROWTYPE, para indicar que la variable es compuesta y que puede guardar una fila de la tabla artículo. Dentro del cuerpo de la función se obtiene, mediante la sentencia SELECT, todos los datos del artículo con el código recibido como parámetro. El resultado obtenido se guarda en la variable *art* que finalmente es devuelta (RETURN art;). Hay que tener claro que podemos devolver la variable *art* porque su tipo (fila de un artículo) es el mismo que el tipo indicado tras la cláusula RETURNS (también, la fila de un artículo).

Podemos llamar a la función del siguiente modo:

```
SELECT * FROM devuelve_articulo('SEG00004');
```

Que nos devolverá el siguiente resultado:

	 codigo character (8)	 descripcion character varying (100)	 precio numeric (7,2)	 dto numeric (3,1)	 stock integer	 stock_min integer
1	SEG00004	Caja Fuerte Camuflada GRID ...	85.00	0.0	15	10

Si queremos podemos seleccionar una o más columnas específicas, por ejemplo:

```
SELECT precio, dto FROM devuelve_articulo('SEG00004');
```

Si ejecutamos la función del siguiente modo:

```
SELECT devuelve_articulo('SEG00004');
```

Veremos que todo el valor devuelto por la función se muestra en una única columna cuyo nombre es el de la función:

	devuelve_articulo	
	articulo	
1	(SEG00004,"Caja Fuerte Camuflada GRID 13000W-S0",85.00,0.0,15,10)	

En este caso, si queremos seleccionar una columna en particular tendremos que utilizar una sintaxis particular consistente en encerrar la llamada de la función entre paréntesis y utilizando un punto como separador indicar el nombre de la columna a seleccionar:

```
SELECT (devuelve_articulo('SEG00004')).precio;
```

Con este método, si queremos seleccionar más de una columna no nos queda otro remedio que llamar dos veces a la función:

```
SELECT (devuelve_articulo('SEG00004')).precio,  
       (devuelve_articulo('SEG00004')).dto;
```

### **Cláusula OUT**

En este apartado veremos cómo utilizar parámetros de salida mediante la cláusula OUT. Los parámetros de salida se declaran en el mismo lugar que los de entrada: entre los paréntesis de nuestra función. Para indicar que son de salida añadiremos la cláusula OUT justo antes de su declaración. Utilizar este método de devolver valores tiene una serie de implicaciones como veremos a continuación.

#### *Tipos simples*

Para demostrar cómo utilizar la cláusula OUT realizaremos una función que recibe dos números y devuelve el resultado de multiplicarlos.

```
CREATE OR REPLACE FUNCTION multiplicacion_out (num1 NUMERIC,  
                                              num2 NUMERIC,  
                                              OUT res NUMERIC)  
  RETURNS NUMERIC AS $$  
  
DECLARE  
  
BEGIN  
    res = num1 * num2;  
  
END  
  
$$ LANGUAGE plpgsql;
```

*multiplicacion\_out*

Este ejemplo nos sirve para explicar las principales diferencias entre utilizar la cláusula OUT y la cláusula RETURN.

- En el código podemos ver que el tercer parámetro es de salida (OUT res NUMERIC) lo que nos permitirá devolver el resultado de la operación.
- Cuando la función solo tiene un parámetro de salida, como es el caso, la cláusula RETURNS deberá ir seguida del mismo tipo que el parámetro de salida, en este caso NUMERIC (RETURNS NUMERIC).

- Si la función tiene más de un parámetro de salida el tipo devuelto tras la cláusula RETURNS será RECORD (RETURNS RECORD).
- Cuando utilizamos parámetros de salida no se utiliza la cláusula RETURN para devolver una variable, si no que asignamos al parámetro de salida el valor que queramos devolver (res = num1 \* num2;).

Si ejecutamos la función del siguiente modo:

```
SELECT multiplicacion_out(cant,precio) FROM linea_ticket;
```

	<b>multiplicacion_out</b>	
	numeric	
1	5.53	
2	6.00	
3	5.53	
4	8.40	
5	3.96	

Vemos que efectivamente se está multiplicando la cantidad por el precio por cada una de las líneas del tique. El nombre de la columna es el nombre de la función (podríamos utilizar un alias para cambiarle el nombre).

También podemos llamar a la función de la siguiente forma:

```
SELECT * FROM multiplicacion_out(6,5);
```

	<b>res</b>	
	numeric	
1	30	

En este caso el nombre de la columna es el nombre del parámetro de salida.

*Tipos compuestos: filas*

También podemos devolver tipos compuestos haciendo uso de la cláusula OUT. En primer lugar, explicaremos cómo devolver una fila (el único tipo compuesto que hemos visto hasta ahora). Para devolver una fila indicaremos como tipo del parámetro de salida el nombre de la tabla.

En el siguiente ejemplo la función recibe por parámetro de entrada el código de un artículo y devuelve todos los datos del artículo mediante un parámetro de salida.

```
CREATE OR REPLACE FUNCTION devuelve_articulo_out (
                                cod articulo.codigo%TYPE, OUT art articulo)
RETURNS articulo AS $$
DECLARE
BEGIN
    SELECT * INTO art FROM articulo WHERE codigo = cod;
END
$$ LANGUAGE plpgsql;
```

*devuelve\_articulo\_out*



El parámetro de salida se ha declarado de tipo *artículo* lo que hace que *art* sea un tipo compuesto (es una fila de la tabla artículo). Además, tras la cláusula RETURNS se indica que el tipo devuelto es *artículo*.

Recuerda que podemos llamar a la función de dos formas:

```
SELECT devuelve_articulo_out('SEG00004');
```

En este caso se obtendrá toda la fila devuelta (todos los datos del artículo) en una única columna llamada *devuelve\_articulo\_out*.

```
SELECT * FROM devuelve_articulo_out('SEG00004');
```

Si llamamos a la función en la cláusula FROM se obtendrán las columnas de forma separada.

### *Tipos compuestos: registros*

Existe otro tipo compuesto llamado RECORD. Una variable de tipo RECORD puede guardar múltiples valores. Éste es el tipo que indicaremos tras la cláusula RETURNS cuando queramos utilizar más de un parámetro de salida.

La siguiente función devuelve el precio mínimo y máximo entre todos los artículos.

```
CREATE OR REPLACE FUNCTION precios_out (OUT min articulo.precio%TYPE,  
                                         OUT max articulo.precio%TYPE)  
    RETURNS RECORD AS $$  
  
DECLARE  
  
BEGIN  
  
    SELECT MAX(precio), MIN(precio) INTO max, min FROM articulo;  
  
END  
  
$$ LANGUAGE plpgsql;
```

*precios\_out*

Puesto que la función tiene más de un parámetro de salida, tras la cláusula RETURNS se ha indicado que el tipo es RECORD (RETURNS RECORD). En realidad, la función devuelve un único valor de tipo RECORD que está compuesto por dos precios (*min* y *max*).

Además, se ha utilizado la cláusula SELECT INTO para guardar directamente los valores obtenidos en los parámetros de salida *min* y *max*. Como hemos mencionado anteriormente, al utilizar parámetros de salida ya no tenemos que retornar valores mediante la cláusula RETURN.

Podemos llamar a la función anterior del siguiente modo:

```
SELECT * FROM precios_out();
```

	 min numeric	 max numeric
1	0.19	975.00

Como vemos, devuelve una única fila con dos columnas, una por cada parámetro de salida. Los nombres de las columnas son los nombres de los parámetros.

La otra forma de llamarla es:

```
SELECT precios_out();
```

	precios_out record
1	(0.19,975.00)

En este caso el resultado se devuelve en una única columna cuyo nombre es el de la función. Al igual que ocurría anteriormente, si queremos seleccionar una columna en particular deberemos utilizar la siguiente sintaxis:

```
SELECT (precios_out()).min;
```

	min numeric
1	0.19

Los parámetros de salida pueden ser a su vez compuestos, por ejemplo, podemos tener varios parámetros de salida cuyo tipo sea una fila de una tabla.

```
CREATE OR REPLACE FUNCTION articulos_out (OUT art_barato articulo,
                                           OUT art_caro articulo)
    RETURNS RECORD AS $$

DECLARE

BEGIN

    SELECT * INTO art_barato FROM articulo
    WHERE precio = (SELECT MIN(precio) FROM articulo);

    SELECT * INTO art_caro FROM articulo
    WHERE precio = (SELECT MAX(precio) FROM articulo);

END

$$ LANGUAGE plpgsql;
```

*articulos\_out*

En el ejemplo anterior tenemos dos parámetros de salida que son filas de la tabla artículo. Puesto que la función tiene más de un parámetro de salida la cláusula RETURNS va seguida de RECORD. Es importante entender que en realidad la función devuelve un único valor de tipo RECORD que está compuesto por dos campos o valores *art\_barato* y *art\_caro* que son, a su vez, compuestas.

Para llamar a la función lo haremos de la misma forma que hemos descrito anteriormente:

```
SELECT * FROM articulos_out();

SELECT articulos_out();
```



### Ejercicio 16

Realiza una función que reciba por parámetro el código de dos artículos. La función devolverá la suma de sus precios. Haz una versión con la cláusula RETURN y otra con la cláusula OUT.

*Resultado esperado*

```
SELECT ejercicio16('SEG00001','SEG00002');

1650.00
```

### Ejercicio 17

Realiza una función que reciba un precio y un descuento y devuelva el precio aplicándole el descuento. Llama la función en una consulta que muestre el código de todos los artículos, su descripción y su precio aplicando el descuento. Haz una versión con la cláusula RETURN y otra con la cláusula OUT.

*Resultado esperado (no se muestran todas las filas por cuestiones de espacio)*

	codigo [PK] character (8)	descripcion character varying (100)	total numeric
1	SEG00001	Armero Athenas Arma Corta ...	975.00
2	SEG00002	Armero Gredos 100-7 Arma L...	654.75
3	SEG00003	Caja Fuerte Camuflada Box-In...	176.40
4	SEG00004	Caja Fuerte Camuflada GRID ...	85.00

### Ejercicio 18

Realiza una función que reciba todos los datos de un ticket y devuelva la cantidad total de artículos que se han vendido en dicho ticket. Haz una versión con la cláusula RETURN y otra con la cláusula OUT.

*Resultado esperado (no se muestran todas las filas por cuestiones de espacio)*

	codigo [PK] integer	cantidad numeric
1	1	19.00
2	2	4.00
3	3	7.00
4	4	7.00
5	5	6.00
6	6	2.00

### Ejercicio 19

Realiza una función que no reciba ningún parámetro y que devuelva todos los datos del artículo más caro. Llama a la función en una consulta que solo obtenga la descripción del artículo y su precio. Haz una versión con la cláusula RETURN y otra con la cláusula OUT.

*Resultado esperado*


	descripcion character varying (100)	precio numeric (7,2)
1	Armero Athenas Arma Corta Alta	975.00

### Ejercicio 20

Realiza una función que reciba como parámetro el código de un ticket y devuelva el nombre del municipio donde reside su cliente y vendedor.

Resultado esperado

```
SELECT * FROM ejercicio20(1);
```

	 mun_cli character varying		mun_ven character varying	
1	Castellón de la Plana		Villarreal	

## Estructuras de control

En este apartado veremos la sintaxis de las estructuras condicionales y bucles que nos permitirán manipular y trabajar con los datos. Estas estructuras son muy similares a las utilizadas en cualquier lenguaje de programación.

### Condicionales

Las sentencias condicionales *IF* nos permiten ejecutar bloques de instrucciones en función de si una condición se cumple o no. En *PL/pgSQL* podemos escribir una sentencia *IF* de tres formas diferentes:

**Condición simple:** se ejecuta un bloque de sentencias si se cumple una determinada condición. En caso contrario se continua con la ejecución de las instrucciones que se encuentren después de la condición.

```
IF condicion THEN
    sentencias;
END IF;
```

**Condición doble:** se ejecuta un bloque de sentencias si se cumple una determinada condición. En caso contrario se ejecuta un bloque de sentencias diferente.

```
IF condicion THEN
    sentencias;
ELSE
    sentencias;
END IF;
```

**Condiciones anidadas:** podemos anidar condiciones unas dentro de otras simplemente escribiendo un *IF* dentro de otro. *PL/pgSQL* dispone de una sintaxis más compacta mediante la cláusula *ELSIF*:

```
IF condicion THEN
    sentencias;
[ ELSIF condicion THEN
    sentencias;
[ ELSIF condicion THEN
    sentencias;
...]
```

```

]
[ ELSE
    sentencias]
END IF;

```

Los corchetes indican las partes que son opcionales y los puntos suspensivos indican que la estructura *ELSIF* se puede repetir múltiples veces. Estos símbolos no se deben escribir.

La *condicion* debe ser una expresión cuyo resultado al evaluarla es un *booleano* por lo que los operadores que podemos utilizar son los relacionales y los lógicos:

### Operadores relacionales

= < > <= >= <> !=

Igual, mayor que, menor que, menor o igual que, mayor o igual que, distinto (podemos utilizar tanto <> como !=)

IS NULL, LIKE,  
BETWEEN, IN

Operadores relacionales utilizados en SQL: no nulo, como, entre y en.

### Operadores lógicos

NOT AND OR

Negación, y, o



### Ejercicio 21

Realiza una función que reciba por parámetro el código de dos artículos. La función devolverá el precio de aquel artículo más caro. La función debe utilizar una sentencia IF y la cláusula OUT para el parámetro devuelto.

*Resultado esperado*

```
SELECT ejercicio21('SEG00001','SEG00002');
```

	ejercicio20 numeric
1	975.00

### Ejercicio 22

Partiendo del ejercicio anterior, modifícalo para que la función devuelva todos los datos del artículo más caro (toda la fila). Haz uso de la cláusula RETURN. Realiza una consulta que llame a la función pasándole como códigos 'SEG00001' y 'SEG00002' y que obtenga el código y el precio del artículo devuelto por la función.

*Resultado esperado*

	codigo character (8)	precio numeric (7,2)
1	SEG00001	975.00

### Ejercicio 23

Realiza una función que reciba por parámetro el código de un artículo y dos precios: *p1* y *p2*. La función devolverá el texto “barato” si el precio del artículo es inferior o igual a *p1*. Devolverá “normal” si el precio está entre *p1* y *p2*. Devolverá “caro” si es superior o igual a *p2*.

Realiza una consulta que muestre la descripción de todos los artículos, su precio y si es un artículo barato, normal o caro.

Resultado esperado (no se muestran todas las filas por cuestiones de espacio)

	descripcion character varying (100)	precio numeric (7,2)	ejercicio22 character varying
1	Caja Fuerte Camuflada Box-In 22100-S1	196.00	caro
2	Caja Fuerte Camuflada GRID 13000W-S0	85.00	normal
3	Caja Fuerte Camuflada GRID 13000-WS1	110.00	caro
4	Caja Fuerte Camuflada Libro	18.90	barato
5	Escalera aluminio 4 peldaños	28.95	barato
6	Escalera articulada multifunción acero	49.95	barato
7	Escalera industrial 2x10 aluminio	124.55	caro

## Bucles

Los bucles nos permiten ejecutar un bloque de instrucciones repetidas veces. En *PL/pgSQL* existen tres tipos de bucles: *LOOP*, *WHILE* y *FOR*.

### **LOOP**

La sintaxis de un bucle *LOOP* es la siguiente:

```
LOOP
    sentencias;
END LOOP;
```

En este tipo de bucles el bloque de sentencias se ejecuta indefinidamente hasta que se ejecute una sentencia *EXIT* o *RETURN*. La sentencia *EXIT* sirve para finalizar la ejecución del bucle, de forma que se ejecutarán las sentencias que hayan después del bucle. La sentencia *RETURN*, como hemos visto anteriormente, sirve para devolver una variable y por tanto finaliza la ejecución de la función.

El siguiente ejemplo muestra por pantalla los números del 1 al 10.

```
CREATE OR REPLACE FUNCTION loop_contador() RETURNS VOID AS $$
DECLARE
    cont INTEGER = 1;
BEGIN
    LOOP
        RAISE NOTICE '%', cont;
        cont=cont+1;
        IF cont > 10 THEN
            EXIT;
        END IF;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

*loop\_contador*

## WHILE

La sintaxis de un bucle *WHILE* es:

```
WHILE condicion LOOP
    sentencias;
END LOOP;
```

El equivalente al ejemplo del contador de 1 a 10 haciendo uso de un bucle *WHILE* sería:

```
CREATE OR REPLACE FUNCTION while_contador() RETURNS VOID AS $$
DECLARE
    cont INTEGER = 1;
BEGIN
    WHILE cont <= 10 LOOP
        RAISE NOTICE '%', cont;
        cont=cont+1;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

---

*while\_contador*

## FOR

La sintaxis de los bucles *FOR* es la siguiente:

```
FOR variable IN [ REVERSE ] inicial .. final [ BY incremento ] LOOP
    sentencias;
END LOOP;
```

Donde *variable* es el identificador que le queremos dar a la variable que irá tomando los valores entre el rango *inicial* y *final*. Esta variable se declara automáticamente como *integer* por lo que no hay que declararla en el bloque *declare* de la función. Las expresiones *inicial* y *final* sirven para especificar el rango de números a evaluar de forma que *variable* comenzará teniendo el valor indicado en *inicial* y finalizará al alcanzar el valor *final*. Por defecto el incremento de la variable en cada iteración es 1 aunque se puede modificar haciendo uso de la cláusula *BY*. La cláusula *REVERSE* se utiliza para indicar si el valor de *incremento* se debe sumar o restar a la *variable* en cada iteración.

La siguiente función muestra los números del 1 al 10:

```
CREATE OR REPLACE FUNCTION for_contador() RETURNS VOID AS $$
DECLARE
BEGIN
    FOR cont IN 1 .. 10 LOOP
        RAISE NOTICE '%', cont;
    END LOOP;
```



```
END
$$ LANGUAGE plpgsql;
```

---

*for\_contador*

Haciendo uso de la cláusula *BY* y *REVERSE* podemos mostrar los números pares existentes entre 1 y 10 de forma descendente.

```
CREATE OR REPLACE FUNCTION for_pares() RETURNS VOID AS $$
DECLARE
BEGIN
    FOR cont IN REVERSE 10 .. 1 BY 2 LOOP
        RAISE NOTICE '%', cont;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

---

*for\_pares*

*Iterando sobre el resultado de una consulta*

Podemos utilizar un bucle *for* para iterar sobre los resultados de una consulta SQL. La sintaxis que utilizaremos será:

```
FOR variable IN consulta LOOP
    sentencias;
END LOOP;
```

Donde *consulta* es una consulta SELECT. En este caso sí que habrá que declarar *variable* en el bloque *declare* de la función. Si la consulta obtiene la fila de una tabla, la variable deberá de ser del tipo *%ROWTYPE* de la tabla correspondiente. Si la consulta obtiene un conjunto de columnas que no se corresponde con las mismas columnas de una tabla, la variable deberá declararse del tipo *RECORD*.

Por ejemplo, la siguiente función recorre todos los artículos, mostrando el código de aquellos cuyo precio es superior al indicado en el parámetro de entrada.

```
CREATE OR REPLACE FUNCTION for_consulta_1(precio articulo.precio%TYPE)
    RETURNS VOID AS $$
DECLARE
    art articulo%ROWTYPE;
BEGIN
    FOR art IN SELECT * FROM articulo LOOP
        IF art.precio > precio THEN
            RAISE NOTICE 'Código: % Precio: %', art.codigo,
                art.precio;
        END IF;
    END LOOP;
END;
```

```

        END LOOP;

    END

    $$ LANGUAGE plpgsql;

```

---

*for\_consulta\_1*

En este caso la `SELECT` obtiene todas las columnas de la tabla *articulo*, por eso se ha declarado la variable *art* del tipo *articulo%ROWTYPE*.

Sin embargo, en el siguiente ejemplo la consulta obtiene únicamente las columnas *codigo* y *precio*. Por lo que la variable *datos* se ha declarado del tipo compuesto `RECORD`.

```

CREATE OR REPLACE FUNCTION for_consulta_2(precio articulo.precio%TYPE)
    RETURNS VOID AS $$

    DECLARE

        datos RECORD;

    BEGIN

        FOR datos IN SELECT a.codigo, a.precio FROM articulo a LOOP

            IF datos.precio > precio THEN

                RAISE NOTICE 'Código: % Precio: %', datos.codigo,
                                datos.precio;

            END IF;

        END LOOP;

    END

    $$ LANGUAGE plpgsql;

```

---

*for\_consulta\_2*

Si utilizáramos alias para renombrar las columnas de la consulta, a la hora de acceder a los campos de la variable habría que utilizar el nombre del alias.



### Ejercicio 24

Realiza una función que reciba por parámetro dos números *n1* y *n2*. La función mostrará por pantalla los números pares que hay entre *n1* y *n2* (inclusive).

*Resultado esperado*

---

```

SELECT ejercicio24(5,8);

NOTICE: 6
NOTICE: 8

```

### Ejercicio 25

Realiza una función que reciba por parámetro dos números *n1* y *n2*. La función mostrará por pantalla las tablas de multiplicar del 1 al 5 de todos los números entre *n1* y *n2* (inclusive).

*Resultado esperado*

---

```

SELECT ejercicio25(1,2);

```

```
NOTICE: 1 x 1 = 1
NOTICE: 1 x 2 = 2
NOTICE: 1 x 3 = 3
NOTICE: 1 x 4 = 4
NOTICE: 1 x 5 = 5
NOTICE: 2 x 1 = 2
NOTICE: 2 x 2 = 4
NOTICE: 2 x 3 = 6
NOTICE: 2 x 4 = 8
NOTICE: 2 x 5 = 10
```

### Ejercicio 26

Realiza una función que reciba por parámetro dos precios *precio1* y *precio2*. La función mostrará por pantalla la descripción de aquellos artículos cuyo precio esté entre *precio1* y *precio2*. Utiliza un bucle FOR.

*Resultado esperado*

---

```
SELECT ejercicio26(600,1000);
NOTICE: Armero Gredos 100-7 Arma Larga
NOTICE: Armero Athenas Arma Corta Alta
```

### Ejercicio 27

Realiza una función que reciba por parámetro dos códigos de tickets *cod1* y *cod2*. La función mostrará por pantalla el número de líneas que tiene cada uno de los tickets cuyo código esté entre *cod1* y *cod2*. Utiliza un bucle WHILE.

*Resultado esperado*

---

```
SELECT ejercicio27(1,2);
NOTICE: El ticket 1 tiene 3 líneas
NOTICE: El ticket 2 tiene 1 líneas
```

## Conjuntos de datos (NO ENTRA)

Hasta ahora hemos visto dos formas de devolver valores: con la cláusula RETURN y con parámetros de salida (cláusula OUT). Con la cláusula RETURN podemos devolver una variable simple (NUMERIC, VARCHAR, etc.) o una variable compuesta (una fila de una tabla). Con los parámetros de salida (cláusula OUT) podríamos pensar que podemos devolver varias variables, ya que podemos tener múltiples parámetros de salida, pero en realidad lo que estamos devolviendo es una única variable compuesta de tipo RECORD que contiene tantos campos como parámetros de salida. Es decir, todos los parámetros de salida están contenidos en una única variable del tipo compuesto RECORD. Así pues, hasta ahora nuestras funciones, como mucho, devolvían una variable ya fuese simple o compuesta, pero solo una.

Con las cláusulas RETURN NEXT y SETOF podemos hacer que nuestra función devuelva múltiples resultados.

El siguiente ejemplo devuelve todos los números enteros entre 1 y 5 (inclusive):

```
CREATE OR REPLACE FUNCTION multiples () RETURNS SETOF NUMERIC AS $$
DECLARE
    cont NUMERIC = 1;
BEGIN
    WHILE cont <= 5 LOOP
        RETURN NEXT cont;
        cont=cont+1;
    END LOOP;
END
$$ LANGUAGE plpgsql;
```

*multiples*

La cláusula SETOF después de RETURNS indica que nuestra función va a devolver múltiples números. La cláusula RETURNS NEXT hace que, cada vez que se ejecute, se devuelva un valor. Una vez se ejecuta no finaliza la ejecución de la función, si no que continúa ejecutando las siguientes líneas. En el ejemplo se devuelve la variable *cont* en cada iteración del bucle (RETURN NEXT *cont*) . Por eso al llamar a la función se obtienen 5 filas (una por cada dato devuelto).

	<b>multiples</b> numeric	
1	1	
2	2	
3	3	
4	4	
5	5	

El equivalente haciendo uso de parámetros de salida mediante la cláusula OUT sería el siguiente:

```
CREATE OR REPLACE FUNCTION multiples_out (OUT cont NUMERIC) RETURNS SETOF
NUMERIC AS $$
```

```

DECLARE

BEGIN
    cont=1;

    WHILE cont <= 10 LOOP
        RETURN NEXT;
        cont=cont+1;
    END LOOP;

END

$$ LANGUAGE plpgsql;

```

---

*multiples\_out*

En este caso hemos añadido el parámetro de salida *cont*. La cláusula `RETURNS SETOF NUMERIC` se mantiene igual, ya que nos sirve para indicar que no vamos a devolver un único valor numérico, si no un conjunto de números. La principal diferencia es que, en este caso, al utilizar `RETURN NEXT` no indicamos qué variable se va a devolver (no escribimos *cont*). Nuestra función ya sabe que esa es la variable que tiene que devolver, porque está declarada como parámetro de salida. Si ejecutamos la función anterior veremos que el resultado obtenido es el mismo.

## Caso práctico I



### Ejercicio 28

En nuestra empresa disponemos de una base de datos con una única tabla llamada *empleados* donde se almacena la información de los empleados. Su modelo relacional es el siguiente:

```
empleados (dni, nombre, fecha_contratacion, sueldo, tipo)
```

Donde *tipo* nos indica si el empleado es un comercial o un técnico.

Las sentencias necesarias para crear la tabla e insertar los empleados las encontraréis en el archivo *empleados.sql*.

Nos han pedido que creemos dos tablas para separar los técnicos de los comerciales. También que creemos una tabla *ventas* para guardar los datos de las ventas realizadas por los comerciales. Por último, en la tabla *comerciales* se incluirá una columna *total\_importe* en la que se guardará la suma de los importes de las ventas realizadas por cada comercial. El modelo relacional quedará de la siguiente forma:

```
tecnicos(dni, nombre, fecha_contratacion, sueldo)
comerciales(dni, nombre, fecha_contratacion, sueldo, total_importe)
      ↑
      |
ventas(id, fecha, importe, comercial)
```

En el archivo *empleados-modificado.sql* encontraréis las sentencias necesarias para crear las nuevas tablas y sus restricciones.

Se pide crear una función encargada de migrar los datos de la tabla *empleados* a las tablas *tecnicos* y *comerciales*. Es decir, la función recorrerá las filas de la tabla *empleados* y los insertará en la tabla *tecnicos* o *comerciales* según corresponda.

### Ejercicio 29

Implementa una función que reciba como parámetros de entrada una fecha *f* y una letra. Si la letra es *T* (de técnico) la función devolverá el número de técnicos cuya fecha de contratación es posterior a la fecha *f*. Si la letra es *C* (de comercial), la función devolverá el número de comerciales cuya fecha de contratación es posterior a la fecha *f*. Si la letra no es *T* ni *C* la función devolverá 0 y se mostrará un mensaje de error por pantalla. El valor se devolverá mediante un parámetro de salida (cláusula OUT).

*Resultado esperado*

```
SELECT ejercicio29('10/10/2019','T');
```

ejercicio29	
integer	
1	1

### Ejercicio 30

Implementa una función que reciba como parámetros un sueldo *s* y un porcentaje *p*. La función deberá comprobar que el sueldo es superior a 0 y que el porcentaje está entre el 0% y el 50% (inclusive). Si se cumplen dichas condiciones, aquellos técnicos y comerciales cuyo sueldo sea inferior al sueldo *s* incrementarán su sueldo el porcentaje determinado por *p*.

#### Resultado esperado

---

El comercial con DNI 30000000C tiene un sueldo de 1000€. Tras ejecutar la siguiente sentencia:

```
SELECT ejercicio30(1200,0.5);
```

Su sueldo pasará a ser 1500€.





### Ejercicio 31

En el archivo *empleados-ventas.sql* encontrarás las sentencias para insertar los datos de las ventas realizadas por los comerciales. Una vez insertados los datos se pide programar una función que calcule el *importe* total de ventas de cada comercial y lo guarde en la columna *total\_importe* de la tabla comerciales.

#### Resultado esperado

---

Tras ejecutar esta función la tabla *comerciales* quedará de la siguiente forma:

	 dni [PK] character (9)	 nombre character varying (50)	 fecha_contratacion date	 sueldo numeric	 total_importe numeric
1	20000000B	Sofía	2019-10-25	1500	6200
2	30000000C	Andrés	2017-04-02	1500.0	6600





En el archivo *ajedrez-partidas.sql* encontrarás las sentencias necesarias para crear las tablas e insertar los datos.

Lamentablemente el diseño de estas tablas no es el más adecuado ya que permite partidas con un único jugador y partidas con más de dos jugadores.

Se pide desarrollar una función que no reciba ningún parámetro ni devuelva ningún valor. La función detectará partidas de uno o más de dos jugadores y mostrará un mensaje de error indicando el identificador de la partida y por qué es inválida.

*Resultado esperado*

---

```
SELECT ejercicio34();  
  
NOTICE: La partida con id 2 tiene más de dos jugadores  
NOTICE: La partida con id 4 tiene más de dos jugadores  
NOTICE: La partida con id 5 tiene un único jugador
```

### **Ejercicio 35**

Otro posible error que se puede dar en los datos es que un mismo jugador participe más de una vez en una partida.

Desarrolla una función que no reciba ningún parámetro ni devuelva ningún valor. La función buscará partidas en las que un mismo jugador aparezca más de una vez. Por cada caso que encuentre, la función mostrará por pantalla el identificador de la partida, el nombre del jugador implicado y cuántas veces aparece en la partida.

*Resultado esperado*

---

```
SELECT ejercicio35();  
  
NOTICE: En la partida con id 3 aparece 2 veces el jugador Vachier-  
Lagrave, Maxime  
  
NOTICE: En la partida con id 4 aparece 2 veces el jugador Vachier-  
Lagrave, Maxime
```

## Ejercicios adicionales



### Ejercicio 36

Realiza una función en la base de datos instituto que reciba como parámetro el nombre de un edificio y devolverá el número de asignaturas que se imparten en dicho edificio.

*Resultado esperado*

```
SELECT ejercicio32('Letras') AS num_asignaturas;
```

	num_asignaturas integer	
1	8	

## Bibliografía

Group, T. P. (2020). *PostgreSQL 12.2 Documentation*.

*PostgreSQL Tutorial*. (4 de Abril de 2020). Obtenido de <https://www.postgresqltutorial.com/>

### Atribuciones

- Portada de *Berti Weber* descargada de [www.pexels.com](http://www.pexels.com)
- Iconos diseñados por *Encahy* descargados de [www.flaticon.es](http://www.flaticon.es)