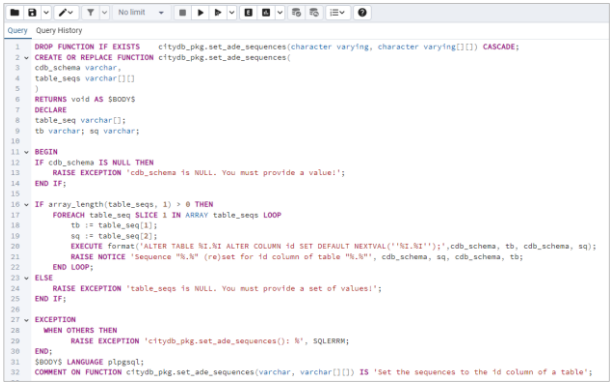
Introduction to PL/pgSQL



```
DROP FUNCTION IF EXISTS     citydb_pkg.set_ade_sequences(character varying, character varying[][]) CASCADE;
CREATE OR REPLACE FUNCTION citydb_pkg.set_ade_sequences(
cdb_schema varchar,
table_seqs varchar[][]
)
RETURNS void AS $BODY$
DECLARE
table_seq varchar[];
tb varchar; sq varchar;

BEGIN
IF cdb_schema IS NULL THEN
    RAISE EXCEPTION 'cdb_schema is NULL. You must provide a value!';
END IF;

IF array_length(table_seqs, 1) > 0 THEN
    FOREACH table_seq SLICE 1 IN ARRAY table_seqs LOOP
        tb := table_seq[1];
        sq := table_seq[2];
        EXECUTE format('ALTER TABLE %I.%I ALTER COLUMN id SET DEFAULT NEXTVAL(''%I.%I'');',cdb_schema, tb, cdb_schema, sq);
        RAISE NOTICE 'Sequence "%.%" (re)set for id column of table "%.%"', cdb_schema, sq, cdb_schema, tb;
    END LOOP;
ELSE
    RAISE EXCEPTION 'table_seqs is NULL. You must provide a set of values!';
END IF;

EXCEPTION
  WHEN OTHERS THEN
        RAISE EXCEPTION 'citydb_pkg.set_ade_sequences(): %', SQLERRM;
END;
$BODY$ LANGUAGE plpgsql;
COMMENT ON FUNCTION citydb_pkg.set_ade_sequences(varchar, varchar[][]) IS 'Set the sequences to the id column of a table';
```

Giorgio Agugiaro

Last update: 9 September 2024

License

Foreword

- This "**PL/pgSQL primer**" is meant to give you a short introduction to the server-side programming language of PostgreSQL called PL/pgSQL

- It is far from being a complete, in-depth guide, as this is not its purpose. It focuses on some most important concepts to extend "standard" (and rather basic) SQL capabilities. These slides take inspiration from the official online documentation of PostgreSQL
  - https://www.postgresql.org/docs/current/plpgsql.html

- Spend some time on reading the original documentation and checking the examples online!

PL/pgSQL: What is it?

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- It is a *procedural language* for PostgreSQL
- In short, it allows to perform more complex operations and computations than "plain" SQL. In particular, you can:
  - Create functions and triggers
  - Add control structures to the SQL language
  - Perform more complex computations
- Additionally:
  - It inherits all user-defined types, functions, and operators
  - It can be defined to be trusted by the server
  - It is rather easy to use
- There are other alternative procedural languages:
  - PL/Tcl, PL/Perl and PL/Python (already installed, but generally not loaded by default)
  - PL/Java, PL/php, PL/R, PL/Ruby, PL/sh, PL/Lua, etc. (by third parties)

TU Delft 3Dgeoinfo

PL/pgSQL: Advantages

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- Generally, every "plain" SQL statement must be executed individually by the database server
  - Client must send query, wait for reply, use data, send further queries, etc.
  - Possible high network overhead, and lots of time needed to send data hence and forth

- PL/pgSQL can improve performance:
  - Groups block of operations (computation and queries) inside the server
  - Avoids data going hence and fort (e.g. no intermediate results being sent over the network)
  - Avoids multiple rounds of query parsing

- Blocks of operations are contained in *functions*
  - https://www.postgresql.org/docs/current/plpgsql-structure.html

PL/pgSQL Functions/Stored procedures

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

Functions:

- Accept as arguments (or return) any scalar or array data type supported by the server
- Can accept or return any composite type (row type)
- Can return a specific record (the result is a row type whose columns are determined by specification in the calling query)
- Can be declared to return *void*
- Can return a “set” (or table) of records
- Can accept and return polymorphic types (*anyelement*, anyarray, anynonarray, anyenum, and *anyrange*)
- Can accept a variable number of arguments by using the VARIADIC marker

Structure of PL/pgSQL

Functions written are defined by executing CREATE (OR REPLACE) FUNCTION command

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION function_name(param1_name integer, param_name2 text)
RETURNS integer
AS
$$ (or:  $label$)
function body text as block
$$ (or:  $label$)
LANGUAGE plpgsql;
```

```
[DECLARE
   declaration of variables]
BEGIN
   statements
END;
```

function body as block

Structure of PL/pgSQL

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- Within the block:
  - You (may) declare variables in the DECLARE section
  - Every statement is ended with semi-colon ; (as in SQL)
  - Comments are
    - -- for single line
    - /* ..... */ for multi-line comments
  - You can send messages to the console via

```
RAISE NOTICE 'Message to the console, generated by % on %', var1, var2;
```

  - If the function returns something, you need to add RETURN inside the block

Example

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE OR REPLACE FUNCTION func_name()
RETURNS integer
AS $$
DECLARE
    -- Here you declare your internal variables (this is a comment line)
   quantity integer := 30;
BEGIN
   RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
   quantity := 50;
   RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50
   RETURN quantity;
END;
$$
LANGUAGE plpgsql;
```

Declarations

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- All variables used in a block must be declared in the declaration section of the block
- PL/pgSQL variables can have any SQL data type, such as integer, varchar, and char, and much more
- The general syntax of a variable declaration is

```
name [CONSTANT] type [COLLATE collation_name] [NOT NULL] [{DEFAULT | := | = } expression];
```

Did you know?
Collation specifies how data is sorted and compared in a database. Collation provides the sorting rules, case, and accent sensitivity properties for the data in the database

Declarations

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

Examples:

- user_id integer;
- quantity numeric(5);
- url varchar;
- genericrow RECORD;
- quantity integer DEFAULT 32;
- url varchar := 'http://mysite.com';
- user_id CONSTANT integer := 10;
- myfield tablename.columnname%TYPE;
- myrow tablename%ROWTYPE;

TU Delft 3Dgeoinfo

Declarations

- Function parameters are named
  - With **$1, $2, $3, … $n** identifiers,
  - Or by defining an alias directly in the CREATE FUNCTION
  - Or by defining an alias in the DECLARE

- In general: if possible, avoid **$n** identifiers, use explicit names!

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION func_prod(integer, integer)
RETURNS integer
AS $$
DECLARE
BEGIN
   RETURN $1 * $2
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION func_name2(varchar, integer)
RETURNS integer
AS $$
DECLARE
   v_string ALIAS FOR $1;
   index ALIAS FOR $2;
BEGIN
   -- some computations using v_string and index here
END;
$$ LANGUAGE plpgsql;
```

Declarations

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION func_name3(v_string varchar, index integer)
RETURNS integer
AS $$
DECLARE
BEGIN
   -- some computations using v_string and index here
END;
$$ LANGUAGE plpgsql;
```

Declarations

- There are several ways to define output parameters
  - Normally in DECLARE
  - Or directly in the CREATE FUNCTION
  - Refer to the on-line documentation for an exhaustive list of possible cases
    - https://www.postgresql.org/docs/current/plpgsql-declarations.html
  - You can also return a RECORD, or a SETOF records (i.e. a table)

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION func_prod2(number1 integer, number2 integer)
RETURNS integer
AS $$
DECLARE
  product integer;
BEGIN
   product := number1 * number2;
   RETURN  product
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION func_prod2(number1 integer, number2 integer,
OUT product integer)
AS $$
DECLARE
BEGIN
   product := number1 * number2;
END;
$$ LANGUAGE plpgsql;
```

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION func_prod2(number1 integer, number2 integer,
OUT product integer, OUT sum integer)
AS $$
DECLARE
BEGIN
    product := number1 * number2;
    sum := number1 + number2;
END;
$$ LANGUAGE plpgsql;
```

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION extended_sales(p_item_no integer)
RETURNS TABLE(quantity int, total numeric)
AS $$
DECLARE
BEGIN
   RETURN QUERY
            SELECT s.quantity, s.quantity * s.price
            FROM sales AS s
            WHERE s.item_no = p_item_no;
END;
$$
LANGUAGE plpgsql;
```

Basic Statements

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- Assignments are:
  - *variable* **:=** *expression;* OR, alternatively, *variable* **=** *expression;*
- To execute a command with no results
  - Just write it (e.g. INSERT, UPDATE, etc.)
  - With a SELECT statement, use **PERFORM** (it will discard the result)
- To execute a command with a single-row result (*target* variable can be a single value or a record of values)
  - SELECT select_expressions **INTO [STRICT]** ***target*** FROM ...;
  - INSERT ... **RETURNING expressions INTO [STRICT]** ***target***;
  - UPDATE ... **RETURNING expressions INTO [STRICT]** ***target***;
  - DELETE ... **RETURNING expressions INTO [STRICT]** ***target***;

If STRICT is specified, the query must return exactly one row!

ATTENTION! In PL/pgSQL SELECT … INTO is quite different from regular SELECT INTO command, wherein the INTO target is a newly created table. If you want to create a table from a SELECT result inside a PL/pgSQL function, use the syntax CREATE TABLE ... AS SELECT.

Basic Statements

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- To execute a dynamic command, use the EXECUTE command

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND
inserted <= $2'
  INTO c
  USING checked_user, checked_date;

EXECUTE 'SELECT count(*) FROM '
   || quote_ident(tabname)
   || ' WHERE inserted_by = $1 AND inserted <= $2'
  INTO c
  USING checked_user, checked_date;
```

Basic Statements

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- To execute a dynamic command, use the EXECUTE command

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

BETTER: use **format()**'s **%I** specifications for table or column names, and **%L** for other variables

```
EXECUTE format('SELECT count(*) FROM %I WHERE inserted_by = %L AND
inserted <= %L', tab_name, checked_user, checked_date)
  INTO c
```

Basic Statements

- Doing nothing
  - Use the NULL;
  - Or just omit a command

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
BEGIN
   y := x / 0;
EXCEPTION
   WHEN division_by_zero THEN
      NULL;  -- ignore the error
END;

BEGIN
   y := x / 0;
EXCEPTION
   WHEN division_by_zero
    THEN  -- do nothing, ignore the error
END;
```

Control Structures

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

To return from a function, there are two commands available to return data from a function: RETURN and RETURN NEXT/QUERY

```
RETURN expression;
```

RETURN terminates the function and returns the value of expression. This form is used for PL/pgSQL functions that do not return a set.

```
RETURN NEXT expression;
RETURN QUERY query;
```

RETURN NEXT and RETURN QUERY append zero or more rows to the function's result set. Execution then continues with the next statement in the PL/pgSQL function. As successive RETURN NEXT or RETURN QUERY commands are executed, the result set is built up. A final RETURN, with no argument, causes control to exit the function.

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION get_all_table1()
RETURNS SETOF table1
AS $$
DECLARE
   r table1%rowtype;
BEGIN
   FOR r IN SELECT * FROM table1 WHERE id > 0 LOOP
      -- can do some processing here
      RETURN NEXT r; -- return current row of SELECT
   END LOOP;
   RETURN;
END
$$ LANGUAGE plpgsql;

SELECT * FROM get_all_table1();
```

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION get_available_flight_id(date)
RETURNS SETOF integer
AS $BODY$
BEGIN
   RETURN QUERY SELECT flight_id FROM flight_table
                WHERE flight_date >= $1 AND flight_date < ($1 + 1);
   -- Since execution is not finished, we can check whether rows were returned
   -- and raise exception if not.
   IF NOT FOUND THEN
      RAISE EXCEPTION 'No flight at %.', $1;
   END IF;
   RETURN;
 END
$BODY$ LANGUAGE plpgsql;

SELECT * FROM get_available_flight_id(CURRENT_DATE);
```

Control Structures

- Conditionals: There are IF-THEN-ELSE and CASE commands
  - IF ... THEN ... END IF;
  - IF ... THEN ... ELSE ... END IF;
  - IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF;
  - CASE ... WHEN ... THEN ... ELSE ... END CASE;
  - CASE WHEN ... THEN ... ELSE ... END CASE;

- Refer to the online documentation for a complete list of examples
  - https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-CONDITIONALS

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
IF v_user_id <> 0 THEN
   UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

```
IF number = 0 THEN
   result := 'zero';
ELSIF number > 0 THEN
   result := 'positive';
ELSIF number < 0 THEN
   result := 'negative';
ELSE
   result := 'NULL';
END IF;
```

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CASE x
   WHEN 1, 2 THEN
      msg := 'one or two';
   ELSE
      msg := 'other value than one or two';
END CASE;
```

```
CASE
   WHEN x BETWEEN 0 AND 10 THEN
      msg := 'value is between zero and ten';
   WHEN x BETWEEN 11 AND 20 THEN
      msg := 'value is between eleven and twenty';
END CASE;
```

Control Structures

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- For loops, there are several commands: LOOP, EXIT, CONTINUE, WHILE, FOR, and FOREACH

- **Remember:**
  - **FOR** loops through integers and query results,
  - **FOREACH** loops through ARRAYS!

- Refer to the online documentation for a complete list of examples
  - https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-CONTROL-STRUCTURES-LOOPS

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
   -- some computations here
END LOOP;

WHILE NOT done LOOP
   -- some computations here
END LOOP;
```

```
FOR i IN 1..10 LOOP
   -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
   -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;
```

FOR loops through integers

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

FOR loops through query results

```
CREATE FUNCTION refresh_mviews()
RETURNS integer
AS $$
DECLARE
   mviews RECORD;
BEGIN
   RAISE NOTICE 'Refreshing all materialized views...';
   FOR mviews IN SELECT n.nspname AS mv_schema, c.relname AS mv_name,
            pg_catalog.pg_get_userbyid(c.relowner) AS owner
            FROM pg_catalog.pg_class c
            LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
            WHERE c.relkind = 'm'  ORDER BY 1 LOOP
      -- Now "mviews" has one record with information about the materialized view
      EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I',
                     mviews.mv_schema, mviews.mv_name);
   END LOOP;
   RAISE NOTICE 'Done refreshing materialized views.';
   RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
CREATE FUNCTION sum(int[])
RETURNS int8
AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1 LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

FOREACH loops through arrays

- If you do not remember how arrays are defined in PostgreSQL, refresh your memory here:
  - https://www.postgresql.org/docs/current/arrays.html

Control Structures

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- Exceptions (errors) can be trapped with an EXCEPTION clause
  - If no error occurs, this form of block simply executes all the statements, and then control passes to the next statement after END. But if an error occurs within the statements, further processing of the statements is abandoned, and control passes to the EXCEPTION list

```
BEGIN
   statements
EXCEPTION
   WHEN condition [ OR condition ... ] THEN handler_statements
   [WHEN condition [ OR condition ... ] THEN handler_statements ... ]
END;
```

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

```
INSERT INTO mytab(firstname, lastname) VALUES ('Tom', 'Jones');

BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
            RAISE NOTICE 'caught division_by_zero';
            RETURN x;
END;
```

More details here:
https://www.postgresql.org/docs/current/plpgsql-control-structures.html#PLPGSQL-ERROR-TRAPPING

Trigger functions

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- PL/pgSQL can be used to define trigger functions on data changes or database events
- Triggers are used to perform automatic operations upon certain conditions (data insert, update, etc.), but can be costly in terms of performance
- A trigger automatically executes its associated function
- A trigger function is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger (for data change triggers) or event_trigger (for database event triggers).
- Special local variables named TG_*something* are automatically defined to describe the condition that triggered the call

- Anyway, triggers (and trigger functions) are beyond the scope of this course
- For more information, please refer to the online documentation and examples
  - https://www.postgresql.org/docs/current/plpgsql-trigger.html

Some coding tips and hints for PL/pgSQL

Intro
Structure
Declarations
Basic statements
Control structures
Triggers
Tips and hints

- It is a good idea to write the function using CREATE **OR REPLACE** FUNCTION. In that way you can just reload the file to update the function definition

- These slides are meant to be a sort of crash course on PL/pgSQL. They are taken from the official online documentation of PostgreSQL
  - https://www.postgresql.org/docs/current/plpgsql.html

- Again: Spend some time on reading the original documentation and checking the examples online!

Dr. Giorgio Agugiaro
g.agugiaro@tudelft.nl
3D Geoinformation Group
TU Delft
The Netherlands
https://3d.bk.tudelft.nl/gagugiaro