>>

PLpgSQL (ii)

- PLpgSQL Functions (recap)
- Debugging Output
- Returning Multiple Values
- INSERT ... RETURNING
- Exceptions

^ >>

❖ PLpgSQL Functions (recap)

Defining PLpgSQL functions:

```
CREATE OR REPLACE
    funcName(param1, param2, ....)
    RETURNS rettype
AS $$
DECLARE
    variable declarations
BEGIN
    code for function
END;
$$ LANGUAGE plpgsql;
```

Setting *rettype* to `void` means "no return value"

<< ^ >>

❖ Debugging Output

Printing info about intermediate states is critical for debugging

Depending on how PostgreSQL is configured

- `raise notice` allows you to display info from a function
- displayed in `psql` window during the function's execution
- usage: `raise notice 'FormatString',Value1,...Valuen;`

Example:

```
-- assuming x==3, y==3.14, z='abc'
raise notice 'x+1 = %, y = %, z = %', x+1, y, z;
-- displays "NOTICE:  x+1 = 4, y = 3.14, z = abc"
```

<< ^ >>

❖ Debugging Output (cont)

Example: a simple function with debugging output

Function

```
create or replace function
   seq(_n int) returns setof int
as $$
declare i int;
begin
   for i in 1..n loop
      raise notice 'i=%',i;
      return next i;
   end loop;
end;
$$ language plpgsql;
```

Output

```
db=# select * from seq(3);
NOTICE:  i=1
NOTICE:  i=2
NOTICE:  i=3
 seq
-----
   1
   2
   3
(3 rows)
```

Replacing `notice` by `exception` causes function to terminate in first iteration

<< ^ >>

❖ Returning Multiple Values

PLpgSQL functions can return a set of values (`setof` *Type*)

- effectively a function returning a table
- *Type* could be atomic ⇒ like a single column
- *Type* could be tuples ⇒ like a full table

Atomic types, e.g.

```
integer, float, numeric, date, text, varchar(n), ...
```

Tuple types, e.g.

```
create type Point as (x float, y float);
```

<< ^ >>

❖ Returning Multiple Values (cont)

Example function returning a set of tuples

```
create type MyPoint as (x integer, y integer);

create or replace function
   points(n integer, m integer) returns setof MyPoint
as $$
declare
   i integer;  j integer;
   p MyPoint;  -- tuple variable
begin
   for i in 1 .. n loop
      for j in 1 .. m loop
         p.x := i;  p.y := j;
         return next p;
      end loop;
   end loop;
end;
$$ language plpgsql;
```

<< ^ >>

❖ Returning Multiple Values (cont)

Functions returning `setof` *Type* are used like tables

```
db=# select * from points(2,3);
 x | y
---+---
 1 | 1
 1 | 2
 1 | 3
 2 | 1
 2 | 2
 2 | 3
(6 rows)
```

<< ^ >>

❖ INSERT ... RETURNING

Can capture values from tuples inserted into DB:

```
insert into Table(...) values
(Val1, Val2, ... Valn)
returning ProjectionList into VarList
```

Useful for recording id values generated for `serial` PKs:

```
declare newid integer; colour text;
...
insert into T(id,a,b,c) values (default,2,3,'red')
returning id,c into newid,colour;
-- id contains the primary key value
-- for the new tuple T(?,2,3,'red')
```

❖ Exceptions

PLpgSQL supports exception handling via

```
begin
    Statements...
exception
    when Exceptions1 then
        StatementsForHandler1
    when Exceptions2 then
        StatementsForHandler2
    ...
end;
```

Each *Exceptions*$_i$ is an `OR` list of exception names, e.g.

```
division_by_zero OR floating_point_exception OR ...
```

A list of exceptions is in Appendix A of the PostgreSQL Manual.

<< ^ >>

❖ Exceptions (cont)

When an exception occurs:

- control is transferred to the relevant exception handling code
- all database changes so far in this transaction are undone
- all function variables retain their current values
- handler executes and then transaction aborts (and function exits)

If no handler in current scope, exception passed to next outer level.

Default exception handlers, at outermost level, exit and log error.

<< ^ >>

❖ Exceptions (cont)

Example: exception handling:

```
-- table T contains one tuple ('Tom','Jones')
declare
   x integer := 3;
   y integer;
begin
   update T set firstname = 'Joe'
   where lastname = 'Jones';
   -- table T now contains ('Joe','Jones')
   x := x + 1;
   y := x / 0;
exception
   when division_by_zero then
      -- update on T is rolled back to ('Tom','Jones')
      raise notice 'caught division_by_zero';
      return x; -- value returned is 4
end;
```

<< ^

❖ Exceptions (cont)

The `raise` operator can generate server log entries, e.g.

```
raise debug1 'Simple message';
raise notice 'User = %',user_id;
raise exception 'Fatal: value was %',value;
```

There are several levels of severity:

- `DEBUG1`, `LOG`, `INFO`, `NOTICE`, `WARNING`, and `EXCEPTION`
- not all severities generate a message to the client (`psql`)

Your CSE server log is the file `/srvr/YOU/pgsql/Log`

Server logs can grow *very* large; delete when you shut your server down

Produced: 27 Feb 2021