Functions in PGPLSQL

functions in psql
create or replace function funcName()
return **s** returntype as **$$**
begin
raise notice 'Hello Birb!';
end;
$$ language langName
The two $ are always necessary. Also note the
returns with an s and the language at the end.
Which **MUST** be a **PROCEDURAL LANGUAGE**,
so c++ doesn't work here.

Parameters are handled like in any language
func(x bigint, y bigint)
you can also define multiple return types
func(variadic a numeric[])
or a generic return
func(param anyelement)

Variable Declaration:
returns void as $$
DECLARE
x bigint; y bigint;
BEGIN ....

Variable manipulation: x := 6 + 4

if: IF n = 0 THEN RETURN 1;
(optional)ELSE RETURN 2; END IF;
ELSIF also possible (note elsif not elseif)

case x when 1, 2 then msg := 'one or two'; end case;
essentially this checks if x is 1 **OR** 2

case when x between 0 and 10 then ....
similar but with a range, both can be
simulated by if else.

Exceptions: BEGIN z:= x / y;
EXCEPTION WHEN division-by-zero
THEN z:= 0; (or error rather) END;
if you want to catch all: WHEN others THEN
often used after exception: RAISE; (show error)

For Loop: For var IN query LOOP
statements END LOOP;
for r in SELECT * FROM ang LOOP
RETURN NEXT r; END LOOP; RETURN; END;
note that the return next doesn't return
you store it in a buffer
and return it at the end of the function.
for infinite loops: FOR i IN 1.._max LOOP;

update and insert: INSERT INTO ANG VALUES(...);
UPDATE ang set salary = salary + 500
where name = 'dashie';
interestingly, after the where name = 'dashie'
you can use if not found then (handle error)
this allows for easier error handling.

queries: **execute** 'SELECT * from ang'
—— into result; return result; END;

comments are done by either – or /* */ for multiline

anonymous function: you can omit
the name and just write do $$ ...

cursor: declare curs CURSOR FOR query;
BEGIN OPEN curs; LOOP do something CLOSE
curs; END;
Cursors are essentially just iterables.
cursors can also be unbound curs1 refcursor
or they can be parameterized curs3 cursor(arg)

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```

arrays: SELECT '1,2,3'::int[]
or SELECT ARRAY[1,2,3]
var int[] only in variable declaration.
!!arrays start with 1 in psql !!

return types: all of the above AND void,
SETOF type (array of a type), TABLE, Trigger

Arrays: Accessoren

```
create table tictactoe as
  (select 1 as id,
          array[
                ['z1 k1', 'z1 k2'],
                ['z2 k1', 'z2 k2'],
                ['z3 k1', 'z3 k2']
               ] as board);
```

Index Query: intuitiv wie eine Koordinate ("1-basiert": Start mit 1 nicht 0):

```
select board[1][1] from tictactoe;
-- z1 k1
```

Slice Query: "Untergrenze:Obergrenze" für jede Dimension:

```
select board[2:3][1:1] from tictactoe;
-- {{z2 k1},{z3 k1}}
```

Max-Bound-Abkürzung "[2]" **vermeiden** (Verwechslungsgefahr), besser [1:2]

```
select board[2:3][2] from tictactoe;  -- [2]=>[1:2]
-- {{z2 k1,z2 k2},{z3 k1,z3 k2}}
```

Suche mit **ANY**:

```
select * from tictactoe where 'z2 k2' = any(board);
-- 1;{{z1 k1,z1 k2},{z2 k1,z2 k2},{z3 k1,z3 k2}}
```

Arrays: Operatoren

- «Is equal»: =
  - SELECT ARRAY[1,2,3] = ARRAY[1,2,3];
    true
  - SELECT ARRAY[3,2,1] = ARRAY[1,2,3];
    false
- «Contains»: @>
  - SELECT ARRAY[1,7,4,2,6] @> ARRAY[2,7];
- «Is contained by»: <@
  - SELECT ARRAY[2,7] <@ ARRAY[1,7,4,2,6];
- «Overlaps»: &&
  - SELECT ARRAY[1,4,3] && ARRAY[2,1]

hstore / map:

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);
```


stored procedures are nothing but a chaining of functions:

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```


some good to know things: plain SQL is more efficient.
write variables lower case for sql
use cast over typename -> not select date '2022-06-07'

Triggers

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```


statement is once, row means once
per row, aka for the entire table.

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```

Create und Insert:

```
CREATE TABLE test (id integer, col2 hstore, col3 text);

INSERT INTO test VALUES (1, 'a=>123, b=>foo, c=>bar'::hstore, NULL);

SELECT * FROM test;
id| col2                                       | col3
-----------------------------------------------------------
1 | "a"=>"123", "b"=>"foo", "c"=>"bar"| null
(1 row)
```


Return types: RETURN NEW -> returns a new table/row
RETURN OLD -> returns the old table/row
(but could change other rows!)
RETURN NULL -> cancel operation.

running order: before statement, before row,
after row, after statement ->
and of course alphabetically.

inside the Trigger functions you can use the
variables that don't matter aka can be ANY -> user
or the entered user from the trigger -> NEW.user
or explicitly the old one -> OLD.user
and last user defined stuff like -> SELECT 'I'
which just places an I as the variable
or something like now() for timestamps.

Triggers make the database slower and harder to maintain.
some databases therefore let you disable them if you want.
On a table basis.

Optimization and indexing

The basics of indexing is that is saves
time on queries, but it uses more space,
and needs to be redone on udpate/insert.

Data stored in Pages / Heap (Collection of Pages)
Indexing on either can be possible.
ex: Page index -> Primary key index
ex: Heap Index -> index on tables
PSQL does table cluster indexing
instead of integrated indexing -> key value (both indexe

Hash Index: just like hashmap in programming
good for single or small multiple queries
bad for entire tables etc -> collision
hashing might take a long time with a lot of data.

Bitmap indexing: Bitmap -> 0 1 stores Booleans/Enums
very fast read / slow update
in postgres **only implicit** use Brin instead.

Brin indexing: