POSTGRE SQL :

CREATE TABLE products (

product\_no integer,

name text,

price numeric

);

If you no longer need a table, you can remove it using the [DROP TABLE](https://www.postgresql.org/docs/current/static/sql-droptable.html) command. For example:

DROP TABLE my\_first\_table;

DROP TABLE products;

**Default Values :**

CREATE TABLE products (

product\_no integer,

name text,

price numeric **DEFAULT 9.99**

);

If no default value is declared explicitly, the default value is the null value.

CREATE TABLE products (

product\_no integer **DEFAULT nextval('products\_product\_no\_seq')**,

...

);

CREATE TABLE products (

product\_no **SERIAL**,

...

);

CREATE SEQUENCE — define a new sequence generator

## Synopsis

CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] *name*

[ AS *data\_type* ]

[ INCREMENT [ BY ] *increment* ]

[ MINVALUE *minvalue* | NO MINVALUE ] [ MAXVALUE *maxvalue* | NO MAXVALUE ]

[ START [ WITH ] *start* ] [ CACHE *cache* ] [ [ NO ] CYCLE ]

[ OWNED BY { *table\_name*.*column\_name* | NONE } ]

TEMPORARY or TEMP ::

If specified, the sequence object is created only for this session, and is automatically dropped on session exit.

CYCLE  
NO CYCLE::

* The CYCLE option allows the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.
* If NO CYCLE is specified, any calls to nextval after the sequence has reached its maximum value will return an error. If neither CYCLE or NO CYCLE are specified, NO CYCLE is the default.

## Constraints::

### Check Constraints

CREATE TABLE products (

product\_no integer,

name text,

price numeric **CHECK (price > 0)**

);

CREATE TABLE products (

product\_no integer,

name text,

price numeric **CONSTRAINT positive\_price** CHECK (price > 0)

);

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

CHECK (price > 0),

discounted\_price numeric,

CHECK (discounted\_price > 0),

CHECK (price > discounted\_price)

);

### Not-Null Constraints

CREATE TABLE products (

product\_no integer NOT NULL,

name text NOT NULL,

price numeric NOT NULL CHECK (price > 0)

);

### Unique Constraints

CREATE TABLE products (

product\_no integer **UNIQUE**,

name text,

price numeric

);

when written as a column constraint, and:

CREATE TABLE products (

product\_no integer,

name text,

price numeric,

**UNIQUE (product\_no)**

);

CREATE TABLE products (

product\_no integer **CONSTRAINT must\_be\_different** UNIQUE,

name text,

price numeric

);

### Primary Keys

CREATE TABLE products (

product\_no integer UNIQUE NOT NULL,

name text,

price numeric

);

CREATE TABLE products (

product\_no integer **PRIMARY KEY**,

name text,

price numeric

);

### Foreign Keys

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

product\_no integer **REFERENCES products (product\_no)**,

quantity integer

);

## A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form.

CREATE TABLE t1 (

a integer PRIMARY KEY,

b integer,

c integer,

**FOREIGN KEY (b, c) REFERENCES other\_table (c1, c2)**

);

## \*

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

shipping\_address text,

...

);

CREATE TABLE order\_items (

product\_no integer REFERENCES products,

order\_id integer REFERENCES orders,

quantity integer,

PRIMARY KEY (product\_no, order\_id)

);

## 

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric

);

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

shipping\_address text,

...

);

CREATE TABLE order\_items (

product\_no integer REFERENCES products **ON DELETE RESTRICT**,

order\_id integer REFERENCES orders **ON DELETE CASCADE**,

quantity integer,

PRIMARY KEY (product\_no, order\_id)

);

### Exclusion Constraints

CREATE TABLE circles (

c circle,

EXCLUDE USING gist (c WITH &&)

);

## Exclusion constraints ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return false or null.

## Modifying Tables :

### Adding a Column:

ALTER TABLE products ADD COLUMN description text;

ALTER TABLE products ADD COLUMN description text CHECK (description <> '');

### Removing a Column

ALTER TABLE products DROP COLUMN description;

ALTER TABLE products DROP COLUMN description CASCADE;

### Removing a Constraint

ALTER TABLE products DROP CONSTRAINT some\_name;

### Changing a Column's Default Value

ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;

ALTER TABLE products ALTER COLUMN price DROP DEFAULT;

### Renaming a Column

ALTER TABLE products RENAME COLUMN product\_no TO product\_number;

### Changing a Column's Data Type

ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);

### Renaming a Table

To rename a table:

ALTER TABLE products RENAME TO items;

## Row Security Policies

## <https://www.postgresql.org/docs/current/static/ddl-rowsecurity.html>

## Inserting Data

CREATE TABLE products (

product\_no integer,

name text,

price numeric

);

INSERT INTO products VALUES (1, 'Cheese', 9.99);

INSERT INTO products (product\_no, name, price) VALUES (1, 'Cheese', 9.99);

INSERT INTO products (name, price, product\_no) VALUES ('Cheese', 9.99, 1);

INSERT INTO products (product\_no, name, price)

SELECT product\_no, name, price FROM new\_products

WHERE release\_date = 'today';

## Updating Data

PDATE products SET price = 10 WHERE price = 5;

## Deleting Data

DELETE FROM products WHERE price = 10;

DELETE FROM products;

## Returning Data From Modified Rows

UPDATE products SET price = price \* 1.10

WHERE price <= 99.99

RETURNING name, price AS new\_price;

DELETE FROM products

WHERE obsoletion\_date = 'today'

RETURNING \*;

## Enumerated Types

CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

CREATE TABLE person (

name text,

current\_mood mood

);

INSERT INTO person VALUES ('Moe', 'happy');

SELECT \* FROM person WHERE current\_mood = 'happy';

name | current\_mood

------+--------------

Moe | happy

(1 row)

### Ordering

INSERT INTO person VALUES ('Larry', 'sad');

INSERT INTO person VALUES ('Curly', 'ok');

SELECT \* FROM person WHERE current\_mood > 'sad';

name | current\_mood

-------+--------------

Moe | happy

Curly | ok

(2 rows)

SELECT \* FROM person WHERE current\_mood > 'sad' ORDER BY current\_mood;

name | current\_mood

-------+--------------

Curly | ok

Moe | happy

(2 rows)

SELECT name

FROM person

WHERE current\_mood = (SELECT MIN(current\_mood) FROM person);

name

-------

Larry

(1 row)

### Type Safety

CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');

CREATE TABLE holidays (

num\_weeks integer,

happiness happiness

);

INSERT INTO holidays(num\_weeks,happiness) VALUES (4, 'happy');

INSERT INTO holidays(num\_weeks,happiness) VALUES (6, 'very happy');

INSERT INTO holidays(num\_weeks,happiness) VALUES (8, 'ecstatic');

INSERT INTO holidays(num\_weeks,happiness) VALUES (2, 'sad');

ERROR: invalid input value for enum happiness: "sad"

SELECT person.name, holidays.num\_weeks FROM person, holidays

WHERE person.current\_mood = holidays.happiness;

ERROR: operator does not exist: mood = happiness

## If you really need to do something like that, you can either write a custom operator or add explicit casts to your query

SELECT person.name, holidays.num\_weeks FROM person, holidays

WHERE person.current\_mood::text = holidays.happiness::text;

name | num\_weeks

------+-----------

Moe | 4

(1 row)

### SQL Functions on Base Types

CREATE FUNCTION one() RETURNS integer AS $$

SELECT 1 AS result;

$$ LANGUAGE SQL;

-- Alternative syntax for string literal:

CREATE FUNCTION one() RETURNS integer AS '

SELECT 1 AS result;

' LANGUAGE SQL;

SELECT one();

one

-----

1

CREATE FUNCTION add\_em(x integer, y integer) RETURNS integer AS $$

SELECT x + y;

$$ LANGUAGE SQL;

SELECT add\_em(1, 2) AS answer;

answer

--------

3

CREATE FUNCTION add\_em(integer, integer) RETURNS integer AS $$

SELECT $1 + $2;

$$ LANGUAGE SQL;

SELECT add\_em(1, 2) AS answer;

answer

--------

3

CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$

UPDATE bank

SET balance = balance - debit

WHERE accountno = tf1.accountno;

SELECT 1;

$$ LANGUAGE SQL;

SELECT tf1(17, 100.0);

CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$

UPDATE bank

SET balance = balance - debit

WHERE accountno = tf1.accountno;

SELECT balance FROM bank WHERE accountno = tf1.accountno;

$$ LANGUAGE SQL;

CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$

UPDATE bank

SET balance = balance - debit

WHERE accountno = tf1.accountno

RETURNING balance;

$$ LANGUAGE SQL;

CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$

UPDATE bank

SET balance = balance - debit

WHERE accountno = tf1.accountno

RETURNING balance;

$$ LANGUAGE SQL;

### SQL Functions on Composite Types

CREATE TABLE emp (

name text,

salary numeric,

age integer,

cubicle point

);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double\_salary(emp) RETURNS numeric AS $$

SELECT $1.salary \* 2 AS salary;

$$ LANGUAGE SQL;

SELECT name, double\_salary(emp.\*) AS dream

FROM emp

WHERE emp.cubicle ~= point '(2,1)';

name | dream

------+-------

Bill | 8400

SELECT name, double\_salary(emp) AS dream

FROM emp

WHERE emp.cubicle ~= point '(2,1)';

## 

CREATE FUNCTION new\_emp() RETURNS emp AS $$

SELECT text 'None' AS name,

1000.0 AS salary,

25 AS age,

point '(2,2)' AS cubicle;

$$ LANGUAGE SQL;

ERROR: function declared to return emp returns varchar instead of text at column 1

CREATE FUNCTION new\_emp() RETURNS emp AS $$

SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;

$$ LANGUAGE SQL;

SELECT new\_emp();

new\_emp

--------------------------

(None,1000.0,25,"(2,2)")

SELECT \* FROM new\_emp();

name | salary | age | cubicle

------+--------+-----+---------

None | 1000.0 | 25 | (2,2)

SELECT (new\_emp()).name;

name

------

None

SELECT name(new\_emp());

name

------

None

### SQL Functions with Output Parameters

CREATE FUNCTION add\_em (IN x int, IN y int, OUT sum int)

AS 'SELECT x + y'

LANGUAGE SQL;

SELECT add\_em(3,7);

add\_em

--------

10

(1 row)

CREATE FUNCTION sum\_n\_product (x int, y int, OUT sum int, OUT product int)

AS 'SELECT x + y, x \* y'

LANGUAGE SQL;

SELECT \* FROM sum\_n\_product(11,42);

sum | product

-----+---------

53 | 462

(1 row)

CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$

SELECT min($1[i]) FROM generate\_subscripts($1, 1) g(i);

$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);

mleast

--------

-1

(1 row)

SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);

SELECT mleast(VARIADIC ARRAY[]::numeric[]);

### SQL Functions with Default Values for Arguments

CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)

RETURNS int

LANGUAGE SQL

AS $$

SELECT $1 + $2 + $3;

$$;

SELECT foo(10, 20, 30);

foo

-----

60

(1 row)

SELECT foo(10, 20);

foo

-----

33

(1 row)

SELECT foo(10);

foo

-----

15

(1 row)

SELECT foo(); -- fails since there is no default for the first argument

ERROR: function foo() does not exist

### SQL Functions as Table Sources

CREATE TABLE foo (fooid int, foosubid int, fooname text);

INSERT INTO foo VALUES (1, 1, 'Joe');

INSERT INTO foo VALUES (1, 2, 'Ed');

INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$

SELECT \* FROM foo WHERE fooid = $1;

$$ LANGUAGE SQL;

SELECT \*, upper(fooname) FROM getfoo(1) AS t1;

fooid | foosubid | fooname | upper

-------+----------+---------+-------

1 | 1 | Joe | JOE

(1 row)

### SQL Functions Returning Sets

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$

SELECT \* FROM foo WHERE fooid = $1;

$$ LANGUAGE SQL;

SELECT \* FROM getfoo(1) AS t1;

Then we would get:

fooid | foosubid | fooname

-------+----------+---------

1 | 1 | Joe

1 | 2 | Ed

(2 rows)

CREATE TABLE tab (y int, z int);

INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum\_n\_product\_with\_tab (x int, OUT sum int, OUT product int)

RETURNS SETOF record

AS $$

SELECT $1 + tab.y, $1 \* tab.y FROM tab;

$$ LANGUAGE SQL;

SELECT \* FROM sum\_n\_product\_with\_tab(10);

sum | product

-----+---------

11 | 10

13 | 30

15 | 50

17 | 70

(4 rows)

SELECT \* FROM nodes;

name | parent

-----------+--------

Top |

Child1 | Top

Child2 | Top

Child3 | Top

SubChild1 | Child1

SubChild2 | Child1

(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$

SELECT name FROM nodes WHERE parent = $1

$$ LANGUAGE SQL STABLE;

SELECT \* FROM listchildren('Top');

listchildren

--------------

Child1

Child2

Child3

(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;

name | child

--------+-----------

Top | Child1

Top | Child2

Top | Child3

Child1 | SubChild1

Child1 | SubChild2

(5 rows)

### SQL Functions Returning TABLE

CREATE FUNCTION sum\_n\_product\_with\_tab (x int)

RETURNS TABLE(sum int, product int) AS $$

SELECT $1 + tab.y, $1 \* tab.y FROM tab;

$$ LANGUAGE SQL;

### Polymorphic SQL Functions

CREATE FUNCTION make\_array(anyelement, anyelement) RETURNS anyarray AS $$

SELECT ARRAY[$1, $2];

$$ LANGUAGE SQL;

SELECT make\_array(1, 2) AS intarray, make\_array('a'::text, 'b') AS textarray;

intarray | textarray

----------+-----------

{1,2} | {a,b}

(1 row)

CREATE FUNCTION is\_greater(anyelement, anyelement) RETURNS boolean AS $$

SELECT $1 > $2;

$$ LANGUAGE SQL;

SELECT is\_greater(1, 2);

is\_greater

------------

f

(1 row)

CREATE FUNCTION invalid\_func() RETURNS anyelement AS $$

SELECT 1;

$$ LANGUAGE SQL;

ERROR: cannot determine result data type

DETAIL: A function returning a polymorphic type must have at least one polymorphic argument.

CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)

AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT \* FROM dup(22);

f2 | f3

----+---------

22 | {22,22}

(1 row)

CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$

SELECT min($1[i]) FROM generate\_subscripts($1, 1) g(i);

$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);

anyleast

----------

-1

(1 row)

SELECT anyleast('abc'::text, 'def');

anyleast

----------

abc

(1 row)

CREATE FUNCTION concat\_values(text, VARIADIC anyarray) RETURNS text AS $$

SELECT array\_to\_string($2, $1);

$$ LANGUAGE SQL;

SELECT concat\_values('|', 1, 4, 2);

concat\_values

---------------

1|4|2

(1 row)