In PostgreSQL, both CREATE DOMAIN and CREATE TYPE are used to define **user-defined data types**, but they serve different purposes and offer different capabilities.

**🔹 1. CREATE DOMAIN (User-Defined Domains)**

**✅ Purpose:**

Used to define a **custom data type with constraints** based on an existing built-in type. It is essentially a named data type with built-in validation rules.

**✅ Key Features:**

* Based on a **single built-in type**.
* Allows **constraints** like NOT NULL, CHECK, etc.
* Mainly used to **enforce data consistency** and reuse logic.

**✅ Syntax:**

sql

CopyEdit

CREATE DOMAIN domain\_name AS base\_type

[ DEFAULT default\_value ]

[ CONSTRAINT constraint\_name CHECK (expression) ];

**✅ Example:**

sql

CopyEdit

CREATE DOMAIN email AS TEXT

CHECK (VALUE ~\* '^[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$');

CREATE TABLE users (

user\_id SERIAL PRIMARY KEY,

user\_email email NOT NULL

);

**✅ Use Case:**

To ensure all email entries follow a pattern, without writing the regex every time.

**🔹 2. CREATE TYPE (User-Defined Composite or Enum Types)**

**✅ Purpose:**

Used to create **entirely new types** such as:

* Composite types (like a struct or record)
* Enumerated types (ENUM)
* Range types (less common)

**✅ Types Under CREATE TYPE:**

1. **Composite Type** – like a table row.
2. **ENUM Type** – set of allowed values.
3. **Range Type** – defines a range of values (advanced use).

**🔸 A. Composite Type**

**✅ Syntax:**

sql

CopyEdit

CREATE TYPE full\_name AS (

first\_name TEXT,

last\_name TEXT

);

**✅ Example Usage:**

sql

CopyEdit

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name full\_name

);

**🔸 B. ENUM Type**

**✅ Syntax:**

sql

CopyEdit

CREATE TYPE job\_role AS ENUM ('Manager', 'Analyst', 'Clerk');

**✅ Example Usage:**

sql

CopyEdit

CREATE TABLE staff (

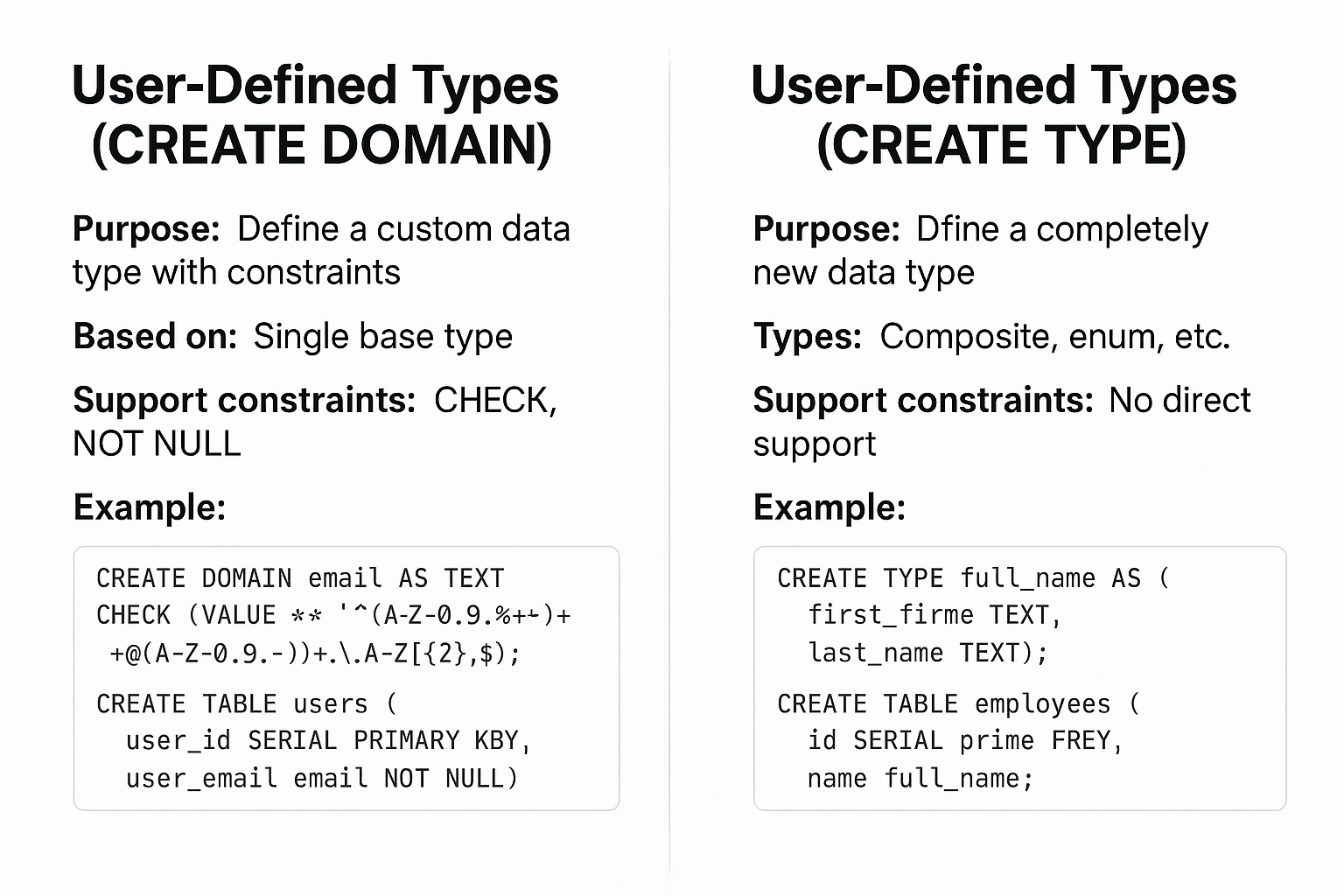
id SERIAL PRIMARY KEY,

role job\_role

);

**🔄 Summary Comparison Table:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **CREATE DOMAIN** | **CREATE TYPE** |
| Based on | Existing base type | Can be completely custom (composite, enum) |
| Supports Constraints | ✅ Yes (CHECK, NOT NULL) | ❌ Not directly (unless ENUM or logic-based) |
| Use Case | Enforce format/constraints | Represent structured or finite data |
| Enum Support | ❌ No | ✅ Yes |
| Composite Type Support | ❌ No | ✅ Yes |
| Simplicity | ✅ Simple | ❌ More complex |
| Reusability | ✅ Reusable data type with constraints | ✅ Highly reusable, flexible types |



In PostgreSQL, **User-Defined Types** allow you to define your own data types. There are two main ways to define them:

1. **User-Defined Types using DOMAIN**
2. **User-Defined Types using TYPE (also called composite types)**

Both serve different purposes and are used in different contexts. Here's a **detailed comparison** with **examples**:

**🔸 1. User-Defined Type using DOMAIN**

**✅ Purpose:**

* To create a **custom data type based on an existing type**, with **constraints**.
* Acts like an alias with validation rules.

**✅ Use Case:**

* Useful when you want to enforce **business rules** (e.g., email format, phone length, positive age, etc.)

**✅ Syntax:**

CREATE DOMAIN domain\_name AS base\_type

[ CONSTRAINT constraint\_name CHECK (expression) ];

**✅ Example:**

-- Create a domain for positive age

CREATE DOMAIN positive\_age AS INTEGER

CHECK (VALUE > 0);

-- Using the domain in a table

CREATE TABLE employee (

name TEXT,

age positive\_age

);

**✅ Key Features:**

* Simple constraints (e.g., length, range, format)
* Easy reuse across tables
* Cannot hold multiple columns (i.e., not composite)

**🔸 2. User-Defined Type using TYPE (Composite Type)**

**✅ Purpose:**

* To define a **custom composite type**, which is a structure containing **multiple fields** (like a table row or a struct).

**✅ Use Case:**

* Useful when you want to **group multiple values** together (e.g., a full address, product details, etc.)

**✅ Syntax:**

CREATE TYPE type\_name AS (

column1 data\_type1,

column2 data\_type2,

...

);

**✅ Example:**

-- Create a composite type for address

CREATE TYPE address\_type AS (

street TEXT,

city TEXT,

zip\_code TEXT

);

-- Use it in a table

CREATE TABLE customer (

id SERIAL PRIMARY KEY,

name TEXT,

address address\_type

);

**✅ Key Features:**

* Can be used in table definitions, functions, and more
* Supports row-like structures
* Cannot enforce constraints like CHECK directly (needs separate validation)

DIFFERNCE BETWEEN User-Defined Types ( DOMAIN) AND User-Defined Types ( TYPE) IN POSTGRE SQL EXPLAIN IN DETAIL WITH THEIR RESPECTIVE EXAMPLES.

**ChatGPT said:**

In PostgreSQL, **User-Defined Types** allow you to define your own data types. There are two main ways to define them:

1. **User-Defined Types using DOMAIN**
2. **User-Defined Types using TYPE (also called composite types)**

Both serve different purposes and are used in different contexts. Here's a **detailed comparison** with **examples**:

**🔸 1. User-Defined Type using DOMAIN**

**✅ Purpose:**

* To create a **custom data type based on an existing type**, with **constraints**.
* Acts like an alias with validation rules.

**✅ Use Case:**

* Useful when you want to enforce **business rules** (e.g., email format, phone length, positive age, etc.)

**✅ Syntax:**

sql

CopyEdit

CREATE DOMAIN domain\_name AS base\_type

[ CONSTRAINT constraint\_name CHECK (expression) ];

**✅ Example:**

sql

CopyEdit

-- Create a domain for positive age

CREATE DOMAIN positive\_age AS INTEGER

CHECK (VALUE > 0);

-- Using the domain in a table

CREATE TABLE employee (

name TEXT,

age positive\_age

);

**✅ Key Features:**

* Simple constraints (e.g., length, range, format)
* Easy reuse across tables
* Cannot hold multiple columns (i.e., not composite)

**🔸 2. User-Defined Type using TYPE (Composite Type)**

**✅ Purpose:**

* To define a **custom composite type**, which is a structure containing **multiple fields** (like a table row or a struct).

**✅ Use Case:**

* Useful when you want to **group multiple values** together (e.g., a full address, product details, etc.)

**✅ Syntax:**

sql

CopyEdit

CREATE TYPE type\_name AS (

column1 data\_type1,

column2 data\_type2,

...

);

**✅ Example:**

sql

CopyEdit

-- Create a composite type for address

CREATE TYPE address\_type AS (

street TEXT,

city TEXT,

zip\_code TEXT

);

-- Use it in a table

CREATE TABLE customer (

id SERIAL PRIMARY KEY,

name TEXT,

address address\_type

);

**✅ Key Features:**

* Can be used in table definitions, functions, and more
* Supports row-like structures
* Cannot enforce constraints like CHECK directly (needs separate validation)

**🔍 Summary Table**

|  |  |  |
| --- | --- | --- |
| **Feature** | **DOMAIN** | **TYPE (Composite)** |
| Based on existing type | ✅ Yes | ❌ No, defines multiple fields |
| Supports multiple fields | ❌ No | ✅ Yes |
| Constraint enforcement | ✅ Built-in (CHECK, etc.) | ❌ Not directly (use functions/triggers) |
| Usage | Columns with validation rules | Grouping fields (e.g., structs) |
| Function return type | ✅ Yes | ✅ Yes |
| Table usage | ✅ Yes | ✅ Yes |
| Example use | positive\_age INTEGER | address\_type (street, city) |

In PostgreSQL, **User-Defined Types**—DOMAIN and TYPE—can be used in **various ways beyond just table columns**. Here's a detailed explanation of their **additional use cases**, **practical applications**, and **examples**:

**🔷 USER-DEFINED TYPE: DOMAIN**

**✅ 1. As a Column in a Table**

* Most common use case.
* Enforces business rules at the data type level.

CREATE DOMAIN email\_domain AS TEXT

CHECK (VALUE ~\* '^[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$');

CREATE TABLE users (

id SERIAL,

email email\_domain

);

**✅ 2. In Function Parameters and Returns**

* Ensures inputs/outputs follow specific constraints.

CREATE FUNCTION register\_user(username TEXT, email email\_domain)

RETURNS VOID AS $$

BEGIN

-- Insert logic

END;

$$ LANGUAGE plpgsql;

**✅ 3. As Variables in Functions**

* Enforces rules even in internal logic.

DO $$

DECLARE

user\_age positive\_age; -- DOMAIN used here

BEGIN

user\_age := 25; -- Will fail if value doesn't satisfy the domain check

END;

$$;

**✅ 4. Used in Composite Types**

* You can include domain types inside composite TYPE.

CREATE TYPE person\_type AS (

name TEXT,

age positive\_age -- DOMAIN used in TYPE

);

**🔷 USER-DEFINED TYPE: TYPE (COMPOSITE)**

**✅ 1. As a Column in a Table**

* Store multiple logically grouped values in one column.

CREATE TYPE contact\_info AS (

phone TEXT,

email TEXT

);

CREATE TABLE customers (

id SERIAL,

contact contact\_info

);

**✅ 2. As a Function Return Type**

* Functions can return multiple values via a custom type.

CREATE TYPE user\_profile AS (

name TEXT,

age INTEGER,

city TEXT

);

CREATE FUNCTION get\_user\_profile(user\_id INT)

RETURNS user\_profile AS $$

BEGIN

RETURN ('Alice', 30, 'Delhi');

END;

$$ LANGUAGE plpgsql;

**✅ 3. As a Function Parameter**

* Accept multiple related values as one argument.

CREATE FUNCTION print\_address(addr address\_type)

RETURNS VOID AS $$

BEGIN

RAISE NOTICE 'Address: %, %, %', addr.street, addr.city, addr.zip\_code;

END;

$$ LANGUAGE plpgsql;

Usage:

SELECT print\_address(ROW('MG Road', 'Mumbai', '400001')::address\_type);

**✅ 4. Table of a Composite Type (Used with RETURNS SETOF)**

* Useful in functions that return multiple rows of structured data.

CREATE FUNCTION list\_addresses()

RETURNS SETOF address\_type AS $$

BEGIN

RETURN QUERY

SELECT 'Park Street', 'Kolkata', '700016'

UNION ALL

SELECT 'Brigade Road', 'Bangalore', '560001';

END;

$$ LANGUAGE plpgsql;

-- Usage

SELECT \* FROM list\_addresses();

**✅ 5. Row Type in PL/pgSQL Blocks**

* For creating row variables inside code blocks.

DECLARE

addr\_row address\_type;

BEGIN

addr\_row := ROW('Station Rd', 'Chennai', '600001');

END;

**🔍 Summary Table of Applications**

|  |  |  |
| --- | --- | --- |
| **Usage Area** | **DOMAIN** | **TYPE (Composite)** |
| Table Column | ✅ Yes | ✅ Yes |
| Function Parameter | ✅ Yes | ✅ Yes |
| Function Return Type | ✅ Yes | ✅ Yes (can return structured values) |
| PL/pgSQL Variable | ✅ Yes | ✅ Yes |
| Inside Composite Type | ✅ Yes (can be nested inside TYPE) | ❌ Not applicable |
| Used with RETURNS SETOF | ❌ Not applicable | ✅ Yes (returns table-like result set) |
| Constraint Enforcement | ✅ Strong constraint checking | ❌ Manual validation required |

**✅ Real-World Applications**

**🏦 Banking:**

* **DOMAIN**: Enforce account number format, positive balance
* **TYPE**: Store address, transaction summary (amount, type, time)

**🛍️ E-commerce:**

* **DOMAIN**: Validate email, price > 0
* **TYPE**: Group order details (product ID, qty, price)

**🧑‍💼 HR Systems:**

* **DOMAIN**: Validate employee age, contact number format
* **TYPE**: Represent full employee record or education qualification

