

Basic data types and functions

Boolean Type

BOOLEAN – ternary logic: true, false, and “unknown” (null) (1 byte)

Boolean operations

and, or – logical AND (conjunction) and OR (disjunction)

		and		or	
		f	t	f	t
f	f	f	f	f	f
f	t	f	f	f	t
t	f	f	f	f	t
t	t	t	t	t	t
N	f	f	f	f	N
f	N	f	f	f	N
N	t	N	N	t	t
t	N	N	N	t	t
N	N	N	N	N	N

not – logical NOT (negation)

		not	
		f	t
f	t	f	t
t	f	t	f
N	N	N	N

bool_and, bool_or – aggregate variants of AND and OR

```
select bool_and(b) from (values (true), (false)) t(b) → f
select bool_and(b) from (values (true), (null)) t(b) → t - null is ignored
select bool_or(b) from (values (true), (false)) t(b) → t
select bool_or(b) from (values (false), (null)) t(b) → f - null is ignored
```

Comparison operators

Comparison operators return a boolean type value. They are defined for all comparable data types.

<, >, <=, >=, =, <> (!=) – less than, greater than, less than or equal to, greater than or equal to, not equal to

```
'Hello' < 'world' → f - string comparison depends on collation
1 = 1 → t
1 = null → null - most operators return null for undefined arguments
```

is [not] distinct from – (not) different; undefined values are considered identical

```
1 is distinct from null → t
1 is distinct from 1 → f
null is distinct from null → f
```

is [not] null – check if undefined

```
1 is null → f
null is null → t
```

is [not] true, is [not] false – check if true or false

	is true	= true	is not false	is false	= false	is not true
f	f	f	f	t	t	t
t	t	t	t	f	f	f
N	f	N	t	f	N	t

[not] between – check if within range

```
X      between A and B = A <= X and X <= B
X not between A and B = X < A  or  X > B

2 between 1 and 3 → t
2 between 3 and 1 → f
1 between 2 and 3 → f
```

[not] between symmetric – check if within range (any order)

```
2 between symmetric 3 and 1 → t
```

Handling null

coalesce – resolve null

```
coalesce(null, 'no') → no
coalesce('yes', 'no') → yes
coalesce(null, null, 'no', f()) → no – takes any number of arguments; f() is not computed
```

nullif – replace with null

```
nullif('no', 'no') → N
nullif('yes', 'no') → yes
```

Numeric values

Integers

SMALLINT – from -32 768 to +32 767 (2 bytes)

INTEGER, INT – from -2 147 483 648 to +2 147 483 647 (4 bytes)

BIGINT – approximately from -10^{19} to $+10^{19}$ (8 bytes)

SMALLSERIAL	} AUTOINCREMENTING; INSTEAD USE GENERATED ALWAYS AS IDENTITY
SERIAL	
BIGSERIAL	

Floating-point numbers

REAL – approximately 6 decimal digits (4 bytes)

FLOAT, DOUBLE PRECISION – approximately 15 decimal digits (8 bytes)

```
-Infinity  negative infinity
+Infinity  positive infinity
NaN        not-a-number
```

```
0.1::real * 10 = 1.0::real → f – values are not stored precisely, watch out when comparing!
```

Arbitrary precision numbers

NUMERIC, DECIMAL – 131 072 digits before the decimal point and 16 383 digits after (variable length)

NUMERIC(N) – integers up to N digits ($N \leq 1000$)

NUMERIC(N, M) – real numbers up to N digits, of which M after the decimal point

NaN not-a-number

Arithmetics

+, −, *, / – add, subtract, multiply, divide

div – integer quotient

`div(7.0, 2.0) → 3 → trunc(7.0/2.0)`

mod, % – remainder of division

`mod(7, 2) → 7%2 → 1`

power, ^ – exponentiation

`power(2, 3) → 2^3 → 8`

abs – absolute value

`abs(-2.7) → 2.7`

sign – sign of the argument

	-2.7	0	2.7
sign	-1	0	1

trunc, ceil (ceiling), round, floor – roundoff

	-2.7	2.7
trunc	-2	2
ceil	-2	3
round	-3	3
floor	-3	2

Type conversion

to_number – string to numeric (see [formatting codes](#))

9	digit	0	digit with a leading zero
.	decimal point	D	period or comma (depending on locale)
,	thousands separator	G	thousands separator (depending on locale)
MI	minus (<0)	PL	plus (>0)
		SG	plus or minus

`to_number('3,1416', '99D00') → 3.14`

`to_number('3,1416', '99D000000') → 3.1416`

`to_number('123,45', '99D00') → numeric field overflow`

String types

CHAR(N) – fixed-length string, padded with spaces

VARCHAR(N) – string with a maximum length limit

TEXT, VARCHAR – string with no length limit

```
'What's up?'
$$What's up?$$
$function$BEGIN; RETURN $$What's up?$$; END;$function$
E'col1\tcol2\nval1\tval2'
```

Segmentation

length, char_length – string length in characters

```
length('Hello, world!') → char_length('Hello, world!') → 13
```

octet_length – string length in bytes

```
octet_length('Hello, world!') → 13 – depending on encoding
```

position, strpos – substring position

```
position('world' in 'Hello, world!') → strpos('Hello, world!', 'world') → 8
```

substring – get substring

```
substring('Hello, world!' from 8 for 5) → substr('Hello, world!', 8, 5) → world
substring('Hello, world!' from 8) → substr('Hello, world!', 8) → world!
```

left, right – substring at the beginning or at the end

```
left('Hello, world!', 5) → Hello
right('Hello, world!', 6) → world!
```

Modification

overlay – replace substring

```
overlay('Hello, world!' placing 'PostgreSQL' from 8 for 5) → Hello, PostgreSQL!
```

replace – replace every instance of substring

```
replace('Hello, world!', 'o', 'ooo') → Helloooo, wooorld!
```

translate – replace characters by dictionary

```
translate('Hello, world!', 'elo', '310') → H3110, w0r1d!
```

lower, upper, initcap – convert case (depending on CTYPE)

```
lower('Hello, world!') → hello, world!
upper('Hello, world!') → HELLO, WORLD!
initcap('Hello, world!') → Hello, World!
```

trim, ltrim, rtrim, btrim – trim characters at string edges (spaces by default)

```
trim( leading 'Held!' from 'Hello, world!') → ltrim('Hello, world!', 'Held!') → lo, world!
trim(trailing 'Held!' from 'Hello, world!') → rtrim('Hello, world!', 'Held!') → Hello, wor
trim( both 'Held!' from 'Hello, world!') → btrim('Hello, world!', 'Held!') → lo, wor
```

lpad, rpad – append to left or right (spaces by default)

```
lpad('Hello, world!', 17, ' ') → . . .Hello, world!
rpad('Hello, world!', 17, ' ') → Hello, world! . . .
```

reverse – reverse character order

```
reverse('Hello, world!') → !dlrow ,olleH
```

Construction

|| – merge two strings

```
'Hello, ' || 'world!' → Hello, world!
```

concat, concat_ws – concatenate strings (any number of arguments)

```
concat('Hello,', ' ', 'world!') → Hello, world!  
concat_ws(' ', 'Hello', 'oh', 'world!') → Hello, oh, world!
```

string_agg – aggregate strings (see [aggregate functions](#))

```
string_agg(s, ' ' order by id) from (values (2,'world!'), (1,'Hello')) v(id,s)  
→ Hello, world!
```

repeat – repeat string

```
repeat('Hello', 2) → HelloHello
```

chr – get character by code (depending on encoding)

```
chr(34) → "
```

Quoting and escaping in dynamic SQL

quote_ident – return string as an identifier

```
quote_ident('id') → id  
quote_ident('foo bar') → "foo bar"
```

quote_literal, quote_nullable – return argument as a string literal

```
quote_literal('id') → 'id'  
quote_nullable('id') → 'id'  
quote_literal($$What's up?$$) → 'What''s up?'  
quote_nullable($$What's up?$$) → 'What''s up?'  
quote_literal(null) → N  
quote_nullable(null) → NULL
```

format – string formatting (like sprintf in C)

```
format('Hello, %s!', 'world') → Hello, world!  
format('UPDATE %I SET s = %L', 'tbl', $$What's up?$$)  
→ 'UPDATE '||quote_ident('tbl')||' SET s = '||quote_nullable($$What's up?$$)  
→ UPDATE tbl SET s = 'What''s up?'
```

Type conversion

to_char – numeric to string (see [formatting codes](#))

9	digit	0	digit with a leading zero
.	decimal point	D	period or comma (depending on locale)
,	thousands separator	G	thousands separator (depending on locale)
RN	roman numerals		
EEEE	exponential notation		
MI	minus (<0)	PL	plus (>0)
FM	no leading zeros and spaces	SG	plus or minus


```
to_char(3.1416, 'FM99D00') → 3.14  
to_char(3.1416, 'FM99D000000') → 3.141600  
to_char(56789, '999G999G999') → 56,789  
to_char(123456789, '999G999G999') → 123,456,789  
to_char(-123456789, '999G999G999') → -123,456,789
```

to_char – date to string (see [formatting codes](#))

YYYY	year		
MM	month (01-12)	MON	month (abbr.)
DD	day (01-31)		MONTH month in full
D	day of the week as number (1-7)	DY	day name (abbr.)
HH	hours (01-12)		DAY day name
MI	minutes	HH24	hours (00-23)
SS	seconds		
TZ	time zone	OF	time zone offset
FM	no leading spaces	TM	translate day and month names

```
to_char(now(), 'DD.MM.YYYY HH24:MI:SSOF') → 05.10.2016 10:51:08+03
to_char(now(), 'FMDD TMonth YYYY, day') → 5 october 2016, wednesday
```

Pattern matching

like (~), not like (!~) – match pattern

```
–      any one character
%      ≥0 characters

'Hello, world!' like 'Hello_ %'          → t
'Hello_world!' like 'Hello\_%' escape '\' → t
```

ilike (~*), not ilike (!~*) – match pattern (case-insensitive)

```
'Hello, world!' ilike 'hello, world!' → t
```

SQL regular expressions

similar to – match

```
–      any one character
%      ≥0 character
*      repeat last character ≥0 times      {m}    repeat m times
+      repeat ≥1 time                      {m,}   repeat ≥m times
?      repeat 0 or 1 time                  {m,n}  repeat m to n times

|      alternatives
(...)  grouping
[...]  character class

'Hello, world!' similar to 'Hello_ %'          → t
'-3.14' similar to '(\+|-)?[0-9]+(\.[0-9]*)?' escape '#' → t
'24.sep.2016' similar to '_{1,2}._{3}._{4}'      → t
```

substring – get substring

```
substring('Hello, world!' from 'Hello, \"%\"!' for '\') → world
```

split_part – split off one word by separator

```
split_part('Hello, world!', ' ', 1) → Hello
split_part('Hello, world!', ' ', 2) → world!
```

string_to_table – split string into a set of rows by separator

```
string_to_table('one two three', ' ') → one
                                         two
                                         three – 3 rows
```

POSIX regular expressions

~, !~ – match

```
.      any one character
*      repeat last character ≥0 times      {m}    repeat m times
+      repeat ≥1 time                      {m,}   repeat ≥m times
?      repeat 0 or 1 time                  {m,n}  repeat m to n times
^      string start                        $      string end

|      alternatives
(...)  grouping
[...]  character class

'Hello, world!' ~ 'Hello'          → t
'Hello, world!' ~ '^Hello$'        → f
'Hello, world!' ~ '^Hello. *$'     → t
'-3.14' ~ '(\+|-)?[0-9]+(\.[0-9]*)?' → t
'24.sep.2016' ~ '.{1,2}\.{3}\.{4}' → t

\m    word start                      \M    word end
\d    digit                          \D    non-digit
\s    whitespace                     \S    non-whitespace
\w    letter or digit                \W    non-letter non-digit
\n    end of line                    \t    tab

'Hello, world!' ~ '\mworld\M' → t
'Helloworld!' ~ '\mworld\M' → f
'-3.14' ~ '[+]?[0-9]+(\.[0-9]*)?' → t
'24.sep.2016' ~ '\d{1,2}\.\w{3}\.\d{4}' → t
```

~*, !~* – case-insensitive match

```
'Hello, world!' ~* 'hello'      → t
'Hello, world!' ~ '(?i)hello' → t  - another format with (?i) at pattern start
```

substring – get substring

```
?      the "non-greedy" quantifier (for *, +, ?, {})

substring('Hello, world!' from '.*o') → Hello, wo
substring('Hello, world!' from '.*?o') → Hello

(?:=...) positive lookahead
(?:!...) negative lookahead

substring('Hello, world!' from '\w+',)      → Hello,
substring('Hello, world!' from '\w+(?=,)') → Hello
substring('Hello, world!' from '\w+(?!,)') → world
```

regexp_matches – match substring

```
regexp_matches('Hello, world!', '\w+')      → {Hello}
regexp_matches('Hello, world!', '(\w+).*?(\w+)') → {Hello,world}
regexp_matches('Hello, world!', '\w+', 'g')  → {Hello}
                                             {world}    - 2 rows
regexp_matches('Hello, world!', '\d')        - 0 rows
```

regexp_replace – replace

```
regexp_replace('Hello, world!', '\w+', 'Hi')      → Hi, world!
regexp_replace('Hello, world!', '(\w+).*?(\w+)', '\2, \1') → world, Hello!
regexp_replace('Hello, world!', '\w+', '_, 'g')   → _, _!
```

regexp_split_to_array – split to array

```
regexp_split_to_array('Hello, world!', ',\s+') → {hello,world!}
```

regexp_split_to_table – split to table

```
regexp_split_to_table('Hello, world!', ',\s+') → Hello
                                              world!  - 2 rows
```

Dates and times

DATE – date without time of day (4 bytes)

```
date '2016-12-31'
make_date(2016,12,31)
```

TIME – time without date (8 bytes)

```
time '23:59:59.999'
make_time(23,59,59.999)
```

TIME WITH TIME ZONE, TIMETZ – time with time zone (12 bytes)

TIMESTAMP – date and time (8 bytes)

```
timestamp '2016-12-31 23:59:59.999'
make_timestamp(2016,12,31,23,59,59.999)
```

TIMESTAMP WITH TIME ZONE, TIMESTAMPTZ – date and time with time zone (8 bytes)

```
timestamptz '2016-12-31 23:59:59.999 MSK'
make_timestamptz(2016,12,31,23,59,59.999, 'MSK')
```

INTERVAL – time interval (16 bytes)

```
interval '1 year 4 months 12 days 03:17:23 ago'
make_interval(-1,-4,0/*weeks*/,-12,-3,-17,-23)
```

Arithmetics

+, **-** – increase (decrease) the date/time by an interval

```
date '2016-12-31' + 1 → 2017-01-01 – for DATE, the interval is an integer number of days
timestamp '2016-12-31 23:50:01' + interval '9 minutes 59 seconds' → 2017-01-01 00:00:00
```

- – the interval between two dates/times

```
date '2016-12-31' - date '2016-12-01' → 30 – for DATE, the interval is an integer number of days
timestamp '2016-12-31 23:59:59' - timestamp '2016-12-01 23:50:00' → 30 days 00:09:59
```

*****, **/** – increase (decrease) the interval a certain number of times

```
5 * interval '1 day' → 5 days
interval '5 day' / 2 → 2 days 12:00:00
```

Functions

overlaps – do the intervals overlap

```
(time '12:00', time '14:00') overlaps (time '13:00', time '15:00') → t
(time '12:00', interval '2 hours') overlaps (time '13:00', interval '2 hours') → t
```

date_trunc – truncate date, time, or interval

```
year      hour
month     minute
day       second

date_trunc('month', timestamp '2016-12-31 23:59:59') → 2016-12-01 00:00:00
date_trunc('minutes', interval '9 minutes 59 seconds') → 00:09:00
```

Current time

current_date, **localtime**, **localtimestamp** – transaction start (without time zone)

current_time, **current_timestamp** = **transaction_timestamp()** = **now()** – transaction start (with time zone)

statement_timestamp() – statement start (with time zone)

clock_timestamp() – current time (with time zone)

Extract part

extract, **date_part** – extract a specific field from date or time (as a double precision type value)

```
year      hour      isodow
month     minute    timezone
day       second    timezone_hour

extract(year from timestamp '2016-12-31 23:59:59')
→ date_part('year', timestamp '2016-12-31 23:59:59') → 2016
extract(isodow from timestamp '2016-12-31 23:59:59') → 6 – 1..7 (Mon..Sun)
```


Type conversion

to_date – string to date (see [formatting codes](#))

```
YYYY year
MM month (01-12)
DD day (01-31)
MON month (abbr.)
DY day name (abbr.)
MONTH month in full
DAY day name (full)

to_date('31.12.2016', 'DD.MM.YYYY') → 2016-12-31
```

to_timestamp – string to date and time with time zone (see [formatting codes](#))

```
YYYY year
MM month (01-12)
DD day (01-31)
HH hours (01-12)
MI minutes
SS seconds
MON month (abbr.)
DY day name (abbr.)
HH24 hours (00-23)
MONTH month in full
DAY day name (full)

to_timestamp('31.12.2016 23:59:59', 'DD.MM.YYYY HH24:MI:SS') → 2016-12-31 23:59:59+03
```

Composite types

CREATE TYPE AS () – composite type as a database object

```
CREATE TYPE monetary AS (amount numeric, currency text);
CREATE TABLE uom(units text, value numeric); - implicitly defines a composite type of the same name

monetary '(25.10,"USD")'
ROW(25.10,'USD')::monetary      (25.10,'USD')::monetary

((25.10,'USD')::monetary).amount → 25.10
((25.10,'USD')::monetary).currency → USD
```

Define a variable storing a table row or a composite type value (PL/pgSQL)

```
DECLARE
    m monetary;
    u uom;
BEGIN
    m := (25.10,'USD')::monetary;
    u := ('inch', 12.77)::uom;
END;
```

RECORD – an anonymous composite type without a predefined structure (PL/pgSQL)

```
DECLARE
    r record;
BEGIN
    r := (25.10,'USD')::monetary;
    SELECT * INTO r FROM uom LIMIT 1;
END;
```

Arrays

TYPE[] – an array of elements of the specified type

```
text[]
integer[][] - two-dimensional array

{'Hello','world!'}
ARRAY['Hello','world!']
'{{1,2,3},{10,20,30}}'
ARRAY[[1,2,3], [10,20,30]]
```

```

(ARRAY['Hello', 'world!'])[1] → Hello - array indexing starts from 1 by default
(ARRAY['Hello', 'world!'])[2] → world!
(ARRAY['Hello', 'world!'])[3] → N - not an error
(ARRAY[[ 1, 2, 3],
        [10,20,30]])[2][1] → 10

'[-1:1]={10,20,30}'

('[-1:1]={5,6,7}':int[])[-1] → 5 - any index value is allowed
('[-1:1]={5,6,7}':int[])[-1:0] → {5,6} - array slice
(ARRAY[[ 1, 2, 3],
        [10,20,30]])[1:2][2:3] → {{2,3},{20,30}}

```

Dimensions

array_ndims – number of dimensions

```

\set A '[1:2][-1:1]={10,20,30},{40,50,60}}'
                                -1  0  1
                                1  10 20 30
                                2  40 50 60

array_ndims(:'A':int[][]) → 2

```

array_length, cardinality – number of elements

```

array_length(:'A':int[][], 1) → 2
array_length(:'A':int[][], 2) → 3
array_length(:'A':int[][]) → 6 = 2*3

```

array_lower, array_upper – the index of the first (last) element

```

array_lower(:'A':int[][], 2) → -1
array_upper(:'A':int[][], 2) → 1

```

Occurrence and search

@>, <@ – is one array contained in the other (GIN index support)

```

ARRAY[1,2,3,4] @> ARRAY[2,3] → t
ARRAY[2,3] <@ ARRAY[1,2,3,4] → t

```

&& – is there at least one common element (GIN index support)

```

ARRAY[1,3,5] && ARRAY[3,6,9] → t

```

array_position, array_positions – element position(s)

```

array_position (ARRAY[7,8,9,8,7], 8) → 2
array_position (ARRAY[7,8,9,8,7], 8, 3) → 4
array_positions(ARRAY[7,8,9,8,7], 8) → {2,4}
array_positions(ARRAY[null,null], null) → {1,2} - (is not distinct from)-like semantics

```

Modification

array_remove – remove element

```

array_remove(ARRAY[7,8,9,8,7], 8) → {7,9,7}

```

array_replace – replace element

```

array_replace(ARRAY[7,8,9,8,7], 8, 0) → {7,0,9,0,7}

```

trim_array – remove a number of elements from array tail

```

trim_array(ARRAY[1,2,3,4,5,6], 2) → {1,2,3,4}

```

Construction

array_fill – initialize array with values

```
array_fill(0, ARRAY[2,3]) → {{0,0,0},{0,0,0}}
array_fill(0, ARRAY[2,3], ARRAY[1,-1]) → [1:2][-1:1]={{0,0,0},{0,0,0}}
```

||, array_append, array_prepend, array_cat – concatenation

```
ARRAY[1,2,3] || 4 → array_append(ARRAY[1,2,3], 4) → {1,2,3,4}
1 || ARRAY[2,3,4] → array_prepend(1, ARRAY[2,3,4]) → {1,2,3,4}
ARRAY[1,2] || ARRAY[3,4] → array_cat(ARRAY[1,2], ARRAY[3,4]) → {1,2,3,4}
```

Type conversion

array_to_string – concatenate array elements

```
array_to_string(ARRAY[1,2,3], ' ', ' ') → 1, 2, 3
array_to_string(ARRAY[1,2,null,4], ' ', ' ', 'N/A') → 1, 2, N/A, 4
```

string_to_array – split string into array by separator

```
string_to_array('one two three', ' ') → {one,two,three}
string_to_array('Hello', null) → {H,e,l,l,o}
string_to_array('1;2;N/A;4', ';', 'N/A') → {1,2,NULL,4}
```

array_agg – table to array

```
select array_agg(a) from (values (1),(2)) t(a) → {1,2}
select array_agg(a) from (values (ARRAY[1,2]),(ARRAY[3,4])) t(a) → {{1,2},{3,4}}
```

unnest – array to table

```
unnest(ARRAY[1,2]) → 1
                     2 – 2 rows
```

Miscellaneous

array_sample – return an array of n random elements of the input array (for multidimensional arrays, elements are taken from the first dimension)

```
array_sample(ARRAY[1,2,3,4,5,6], 3) → {2,6,1}
array_sample(ARRAY[[1,2],[3,4],[5,6]], 2) → {{5,6},{1,2}}
```

array_shuffle – shuffle the elements of the first dimension of the array

```
array_shuffle(ARRAY[1, 2, 3]) → {2,3,1}
array_shuffle(ARRAY[[1,2],[3,4],[5,6]]) → {{5,6},{1,2},{3,4}}
```