Chapter 3

October 27, 2019

```
In [1]: %load_ext sql
```

```
In [59]: %sql postgresql://postgres:postgres@localhost:5432/analysis
```

```
Out[59]: 'Connected: postgres@analysis'
```

1 Understanding Data Types

```
In [5]: %%sql

        CREATE TABLE char_data_types(
        varchar_column varchar(10),
        char_column char(10),
        text_column text
        );
```

```
 * postgresql://postgres:***@localhost:5432/analysis
(psycopg2.errors.DuplicateTable) relation "char_data_types" already exists

[SQL: CREATE TABLE char_data_types(
varchar_column varchar(10),
char_column char(10),
text_column text
);]
(Background on this error at: http://sqlalche.me/e/f405)
```

- we first make a table with three different char types

```
In [13]: %%sql

         INSERT INTO char_data_types
         VALUES
         ('abc', 'abc', 'abc'),
         ('defghi', 'defghi', 'defghi');
```

```
 * postgresql://postgres:***@localhost:5432/analysis
(psycopg2.errors.UndefinedTable) relation "char_data_types" does not exist
```

```
LINE 1: INSERT INTO char_data_types
                    ^

[SQL: INSERT INTO char_data_types
VALUES
('abc', 'abc', 'abc'),
('defghi', 'defghi', 'defghi');]
(Background on this error at: http://sqlalche.me/e/f405)
```

- we are not specifying the names of the columns

```
In [11]: %%sql

         COPY char_data_types TO '/Users/ugurtigu/Documents/Learn/Docs/SQL/typetest.txt'
         WITH (FORMAT CSV, HEADER, DELIMITER '|');
```

```
 * postgresql://postgres:***@localhost:5432/analysis
2 rows affected.
```

```
Out[11]: []
```

- we use the COPY keyword to export the data to a text file names typetest.txt
  - COPY table_name FROM is the import function
  - COPY table_name TO is the export function
  - the WITH keyword will format each column separeted by a pipe character (|)
- even though you specified 10 characters for both the varchar and char columns, only the char columns outputs 10 characters every time, padding unused characters with whitespace

```
In [20]: f = open('typetest.txt', 'r')
         print(f.read())
```

```
varchar_column|char_column|text_column
abc|abc       |abc
defghi|defghi    |defghi
```

1.1 Numbers

- **Integers** whole numbers, both postive and negative
  - smallint
  - integer
  - bigint
- **Auto-Incrementing Integers**

  - smallserial
  - serial
  - bigserial

- theese are corresponding to the **Integer** types
- the special thing about them is, that they are incrementing starting with 1
- this can be handy for id when creating a table

```
In [21]: %%sql

         CREATE TABLE peope(
         id serial,
         person_name varchar(100)
         );
```

```
 * postgresql://postgres:***@localhost:5432/analysis
Done.
```

```
Out[21]: []
```

- now every time a new person_name is added to the table, the id column will increment by 1

- **Decimal Numbers**
- Decimal Numbers represent a whole number plus a fraction of a whole number, which is represented by digits following a decimal point
- **fixed-point** and **floating-point**
  - **fixed-point**
  - numeric(precision,scale)
  - you give the argument precision as the maximum number of digits to the lest and right of the decial point
  - and the argument scale as the number of digits allowable on the right of the decimal point
  - if you omit the a scale value, the scale will be set to zero, which will create an integer
  - if you omit both the maximum will be taken
  - for example 1.47 and 1.00 and 121.50
  - **floating-point**
  - **real** and **double precision**
  - difference between the both is the precision, real allows 6 decimal digits and double 15
  - the floating-point is calles variable-precision
  - the decimal point in a given column "float" depending on the number

```
In [25]: %%sql

         CREATE TABLE number_data_types(
             numeric_column numeric(20,5),
             real_column real,
             double_column double precision
         );
```

```
 * postgresql://postgres:***@localhost:5432/analysis
Done.
```

```
Out[25]: []
```

- first we create a table

```
In [26]: %%sql

         INSERT INTO number_data_types
         VALUES
         (.7, .7, .7),
         (2.13579, 2.13579, 2.13579),
         (2.1357987654, 2.1357987654, 2.1357987654);
```

```
 * postgresql://postgres:***@localhost:5432/analysis
3 rows affected.
```

```
Out[26]: []
```

- we insert our values into the tables columns

```
In [27]: %%sql

         SELECT * FROM number_data_types;
```

```
 * postgresql://postgres:***@localhost:5432/analysis
3 rows affected.
```

```
Out[27]: [(Decimal('0.70000'), 0.7, 0.7),
          (Decimal('2.13579'), 2.13579, 2.13579),
          (Decimal('2.13580'), 2.1357987, 2.1357987654)]
```

- and we have our table
- the numeric column, with a scale of five stores five digits after the decimal point whether or not you inserted that many
- and if more than 10 digits we give, it rounds it after the decimal
- the real and double precision columns store only the number of digits present with no padding
- the real row also rounds it because it has maximum of six digits of precision
- the double precision can hold up to 15 digits, so it stores the entire number

```
In [30]: %%sql

         SELECT
         numeric_column * 10000000 AS "Fixed",
         real_column * 10000000 AS "Float"
         FROM number_data_types
         WHERE numeric_column = 0.7;
```

```
 * postgresql://postgres:***@localhost:5432/analysis
1 rows affected.
```

```
Out[30]: [(Decimal('7000000.00000'), 6999999.88079071)]
```

- **trouble with floating-point math**
- we are multiplying the numeric_column and the real_column by 10 million
- float is inaccurate

- **Choosing your number data type**
  - use integers when possible
  - if you are working with decimals
    - choose numeric or decimal
    - with whole numbers use bigint
    - with small ones smallint or integer

1.2 Dates and Times

- **timestamp**
  - date and time
  - with time zone
- **date**
  - just date
- **time**
  - just time
- **interval**
  - only the lenght of the time

```
In [32]: %%sql

         CREATE TABLE date_time_types (
         timestamp_column timestamp with time zone,
         interval_column interval
         );
```

```
 * postgresql://postgres:***@localhost:5432/analysis
Done.
```

```
Out[32]: []
```

```
In [33]: %%sql

         INSERT INTO date_time_types
         VALUES
             ('2018-12-31 01:00 EST', '2 days'),
             ('2018-12-31 01:00 -8', '1 month'),
             ('2018-12-31 01:00 Australia/Melbourne', '1 century'),
             (now(), '1 week');
```

```
 * postgresql://postgres:***@localhost:5432/analysis
4 rows affected.
```

```
Out[33]: []
```

```
In [34]: %%sql

         SELECT * FROM date_time_types;
```

```
 * postgresql://postgres:***@localhost:5432/analysis
4 rows affected.
```

```
Out[34]: [(datetime.datetime(2018, 12, 31, 7, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offset=
          (datetime.datetime(2018, 12, 31, 10, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offse
          (datetime.datetime(2018, 12, 30, 15, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offse
          (datetime.datetime(2019, 10, 25, 16, 49, 26, 801488, tzinfo=psycopg2.tz.FixedOffsetTi
```

- we created a table with timestamps and intervals
- we inserted our values
- we show our values

```
In [41]: %%sql

         SELECT
         timestamp_column,
         interval_column,
         timestamp_column - interval_column AS new_date
         FROM date_time_types;
```

```
 * postgresql://postgres:***@localhost:5432/analysis
4 rows affected.
```

```
Out[41]: [(datetime.datetime(2018, 12, 31, 7, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offset=
          (datetime.datetime(2018, 12, 31, 10, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offse
          (datetime.datetime(2018, 12, 30, 15, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offse
          (datetime.datetime(2019, 10, 25, 16, 49, 26, 801488, tzinfo=psycopg2.tz.FixedOffsetTi
```

- a typical SELECT statement which contains also a new column called **new_date**
- which is a refult of our previous defined timestamp_column and interval_colum
- we do a arithemtic operation with both, this is a computed column which is called **expressions**

1.3 Miscellaneous Types

- a boolean type that stores a value of TRUE or FALSE
- geometric types (points, lines, circles)
- network adresses such as IP or MAC adress
- UUID, sometimes used as unqique key value
- XML or JSON data types

1.4 Transforming Values from one type to another with CAST

- the CAST() function transorms one type to another
- casting an integer to a string is possible, because strings contains integer too
- casting text with letters of the alphabet as a number is not

```
In [54]: %%sql

         SELECT timestamp_column, CAST(timestamp_column AS varchar(10))
         FROM date_time_types;

 * postgresql://postgres:***@localhost:5432/analysis
4 rows affected.


Out[54]: [(datetime.datetime(2018, 12, 31, 7, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offset=
          (datetime.datetime(2018, 12, 31, 10, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offset
          (datetime.datetime(2018, 12, 30, 15, 0, tzinfo=psycopg2.tz.FixedOffsetTimezone(offset
          (datetime.datetime(2019, 10, 25, 16, 49, 26, 801488, tzinfo=psycopg2.tz.FixedOffsetTi
```

- this SELECT statement will return the timestamp_column value as a varchar
- we set the lenght of the new varchar with 10
- the first 10 numbers will be casted
- this is handy in this case, because we exclude the time

```
In [55]: %%sql

         SELECT numeric_column,
         CAST(numeric_column AS integer),
         CAST(numeric_column AS varchar(6))
         FROM number_data_types;

 * postgresql://postgres:***@localhost:5432/analysis
3 rows affected.


Out[55]: [(Decimal('0.70000'), 1, '0.7000'),
          (Decimal('2.13579'), 2, '2.1357'),
          (Decimal('2.13580'), 2, '2.1358')]
```

- this statement returns the numeric_column three times
- first is the original one

- then as an integer, which will be rounded to a whole number
- then as a char with the lenght of 6

```
In [57]: %%sql

         SELECT CAST(char_column AS integer) FROM char_data_types;

         # This will not work!!!

 * postgresql://postgres:***@localhost:5432/analysis
(psycopg2.errors.UndefinedTable) relation "char_data_types" does not exist
LINE 1: SELECT CAST(char_column AS integer) FROM char_data_types;
                                                 ^

[SQL: SELECT CAST(char_column AS integer) FROM char_data_types;]
(Background on this error at: http://sqlalche.me/e/f405)
```

- letters can not become integers, so this will produce an error (not this erris, it will be an invalid input syntax for integer)

```
In [50]: %%sql

         SELECT timestamp_column::varchar(10)
         FROM date_time_types;

 * postgresql://postgres:***@localhost:5432/analysis
4 rows affected.


Out[50]: [('2018-12-31',), ('2018-12-31',), ('2018-12-30',), ('2019-10-25',)]
```

- this will work

1.5 Try if Yourself

– 1. Your company delivers fruit and vegetables to local grocery stores, and – you need to track the mileage driven by each driver each day to a tenth – of a mile. Assuming no driver would ever travel more than 999 miles in – a day, what would be an appropriate data type for the mileage column in your – table. Why?

- a numeric(4,1)
- four digits precision
- a value more than 999.9 would be possible
- in practice it would be a numeric(5,1)

– 2. In the table listing each driver in your company, what are appropriate – data types for the drivers' first and last names? Why is it a good idea to – separate first and last names into two columns rather than having one – larger name column?

- varchar(50)
- 50 characters for a name is reasonable
- seperating name and last name will make them sortable

– 3. Assume you have a text column that includes strings formatted as dates. – One of the strings is written as ‘4//2017’. What will happen when you try – to convert that string to the timestamp data type?

- converting a string that does not have the date format will cause to an error