```
/* ************************************************************************
           INTRODUCTION TO STRUCTURED QUERY LANGUAGE FOR DATA ANALYTICS
                     WS24SQL10001, 2024/03/11 - 2024/04/10
                   https://folvera.commons.gc.cuny.edu/?cat=34
 ************************************************************************

  SESSION #8 (2024/04/03): CREATING DATABASE OBJECTS

  1. Altering databases, schemata, tables
  2. Understanding `NULL` and `NOT NULL`
  3. Parameters, user-defined functions and stored procedures
 ************************************************************************

 1. LAB #7
    Write a query without duplicate rows (`SELECT DISTINCT`)
    1.01. to get all shared values from tables `AP1.InvoiceLineItems` and
          `AP1.GLAccounts` (`INNER JOIN`),
    1.02. adding today's date as `TodaysDate` formatted as short date
    1.03. where `AP1.GLAccounts.AccountDescription` starts with `book`
          (`AP1.GLAccounts.AccountDescription LIKE('book%')`) and
          `AP1.InvoiceLineItems.InvoiceLineItemAmount` is at least 1000.00
          (inclusive) -- first condition composed of two conditions
    1.04. or where `AP1.GLAccounts.AccountDescription` contains `mail` and
          `AP1.InvoiceLineItems.InvoiceLineItemAmount` is no more than
          100.00 (inclusive) -- second condition composed of two conditions
          (second condition in parenthesis (OR secondary_codition1 AND
          secondary_condition2))
    1.05. ordered first by `AP1.GLAccounts.AccountDescription` and then by
          `AP1.InvoiceLineItems.InvoiceLineItemAmount`.
    1.06. Then make a view in schema `lab07` in database `labs`.
 ************************************************************************ */

SELECT DISTINCT AP1.InvoiceLineItems.InvoiceID,
  AP1.InvoiceLineItems.InvoiceSequence,
  AP1.InvoiceLineItems.AccountNo,
  AP1.InvoiceLineItems.InvoiceLineItemAmount,
  AP1.InvoiceLineItems.InvoiceLineItemDescription,
  -- AP1.GLAccounts.AccountNo AS Expr1,
  AP1.GLAccounts.AccountDescription
  /*,
  FORMAT(GETDATE(), 'd', 'en-us') AS TodaysDate*/
FROM AP1.InvoiceLineItems
INNER JOIN AP1.GLAccounts
  ON AP1.InvoiceLineItems.AccountNo = AP1.GLAccounts.AccountNo
WHERE
  (                                          -- 1. first block of two
                                             --    conditions that must be
                                             --    true
    AP1.GLAccounts.AccountDescription LIKE 'book%'
    AND AP1.InvoiceLineItems.InvoiceLineItemAmount >= 1000
    )
  OR                                         -- 2. `OR` to indicate that
                                             --    either the first block
                                             --    (above) or the second
                                             --    (below) must be true
  (                                          -- 3. second block of two
                                             --    conditions that must be
                                             --    true
    AP1.GLAccounts.AccountDescription LIKE '%mail%'
    AND AP1.InvoiceLineItems.InvoiceLineItemAmount <= 100
    )
ORDER BY AP1.GLAccounts.AccountDescription,
  AP1.InvoiceLineItems.InvoiceLineItemAmount,
  AP1.InvoiceLineItems.InvoiceID,
  AP1.InvoiceLineItems.InvoiceSequence,
  AP1.InvoiceLineItems.AccountNo,
  AP1.InvoiceLineItems.InvoiceLineItemDescription;
```

```
/* ************************************************************************
          At this point, we make a view in schema `lab07` in database `labs`
          excluding the `ORDER BY` clause as it cannot be used when creating
          views.
 ************************************************************************ */

CREATE VIEW lab07.InvoiceLineItemsGLAccountsVW
AS
SELECT DISTINCT WS24SQL10001.AP1.InvoiceLineItems.InvoiceID,
  WS24SQL10001.AP1.InvoiceLineItems.InvoiceSequence,
  WS24SQL10001.AP1.InvoiceLineItems.AccountNo,
  WS24SQL10001.AP1.InvoiceLineItems.InvoiceLineItemAmount,
  WS24SQL10001.AP1.InvoiceLineItems.InvoiceLineItemDescription,
  -- WS24SQL10001.AP1.GLAccounts.AccountNo AS Expr1,
  WS24SQL10001.AP1.GLAccounts.AccountDescription
FROM WS24SQL10001.AP1.InvoiceLineItems
INNER JOIN WS24SQL10001.AP1.GLAccounts
  ON WS24SQL10001.AP1.InvoiceLineItems.AccountNo =
    WS24SQL10001.AP1.GLAccounts.AccountNo
WHERE (
    WS24SQL10001.AP1.GLAccounts.AccountDescription LIKE 'book%'
    AND WS24SQL10001.AP1.InvoiceLineItems.InvoiceLineItemAmount >= 1000
    )
  OR (
    WS24SQL10001.AP1.GLAccounts.AccountDescription LIKE '%mail%'
    AND WS24SQL10001.AP1.InvoiceLineItems.InvoiceLineItemAmount <= 100
    );


/* ************************************************************************
          As an alternative, we can use an alias (`AS`) for each table to make
          our code tidier and to avoid repeating the database and schema before
          each table (`<database>.<schema>.<table>` or `<schema>.<table>`)
          throughout the query.

                    `i` for `WS24SQL10001.AP1.InvoiceLineItems`
                    `g` for `WS24SQL10001.AP1.GLAccounts`
 ************************************************************************ */

CREATE VIEW lab07.InvoiceLineItemsGLAccountsVW
AS
SELECT DISTINCT i.InvoiceID,
  i.InvoiceSequence,
  i.AccountNo,
  i.InvoiceLineItemAmount,
  i.InvoiceLineItemDescription,
  g.AccountDescription
FROM WS24SQL10001.AP1.InvoiceLineItems AS i
INNER JOIN WS24SQL10001.AP1.GLAccounts AS g
  ON i.AccountNo = g.AccountNo
WHERE (
    g.AccountDescription LIKE 'book%'
    AND i.InvoiceLineItemAmount >= 1000
    )
  OR (
    g.AccountDescription LIKE '%mail%'
    AND i.InvoiceLineItemAmount <= 100
    );


/* ************************************************************************
 2. LAB #8 (CREATING OBJECTS)
    2.01. Create database `labs`.
    2.02. Create schema `lab08` in database `labs`.
    2.03. Create table `my_family` in schema `lab08` with the following
          structure  choosing the best file type for each column and assign
          `NOT NULL` to each.

                    row_id
```

```
                    person_fname
                    person_lname
                    relation

    2.04. Insert values accordingly.
    2.05. Modify table `my_family` to add a column `dob`.
    2.06. Update the table with data in `dob` (new values in an existing record
          in table `labs.lab08.my_family`).
    2.07. Change column `dob` to `NOT NULL`.
 ********************************************************************************* */

CREATE DATABASE labs;                           -- 1. creating database `labs`
                                                --    1.1. run #1 (all `CREATE
                                                --         DATABASE` statements
                                                --         run together, but
                                                --         separately from
                                                --         other statements)

CREATE SCHEMA lab08;                            -- 2. creating schema `lab08`

CREATE TABLE lab08.my_family (                  -- 3. creating table
  row_id INT NOT NULL,                          --    `lab08.my_family`
  person_fname VARCHAR(25) NOT NULL,            --    3.1. run #3 (all `CREATE
  person_lname VARCHAR(25) NOT NULL,            --         TABLE` statement run
  relation VARCHAR(15) NOT NULL                 --         together, but
  );                                            --         separately from
                                                --         other statements)

INSERT INTO lab08.my_family                     -- 4. inserting new values into
VALUES (                                        --    table `lab08.my_family`
  1,                                            --    4.1. each row/record
  'John',                                       --         within a set of
  'Doe',                                        --         parenthesis followed
  'crazy uncle'                                 --         by a comma between
  ),                                            --         rows/records
  (                                             --    4.2. run #4 (all `INSERT`
  2,                                            --         statements run
  'Michael',                                    --         together, separately
  'Jones',                                      --         from other
  'cousin'                                      --         statements)
  ),
  (
  3,
  'Lucy',
  'Smith',
  'aunt'
  );

ALTER TABLE lab08.my_family                     -- 5. altering table
ADD dob DATE;                                   --    `lab08.my_family` to add
                                                --    column `dob` with data
                                                --    type `DATE`
                                                --    5.1. run #5 (all `ALTER`
                                                --         statements run
                                                --         together, separately
                                                --         from other
                                                --         statements)

UPDATE lab08.my_family                          -- 6. updating table
SET dob = '1970-01-01'                          --    `lab08.my_family` to pass
WHERE row_id = 1;                               --    a new values to column
                                                --    `dob` in the existing
UPDATE lab08.my_family                          --    table `lab08.my_family`
SET dob = '1980/05/09'                          --    6.1. run #6 (all `UPDATE`
WHERE row_id = 2;                               --         statements run
                                                --         together, separately
UPDATE lab08.my_family                          --         from other
SET dob = '1988/08/19'                          --         statements)
WHERE row_id = 3;
```

```
208
209 ALTER TABLE lab08.my_family                        -- 7. changing new column `dob`
210 ALTER COLUMN dob DATE NOT NULL;                    --    to `NOT NULL` as column
211                                                    --    now has values
212                                                    --    7.1. run #7 (this `ALTER`
213                                                    --         statement run after
214                                                    --         populating new
215                                                    --         column `dob`
216
217
218 /* ***************************************************************************
219  3. LAB #9 (ALTERING OBJECTS)
220     Make some changes to `AP1.ContactUpdates` and `AP1.Vendors`.
221
222     3.01. Add column `Email` to `AP1.ContactUpdates`, which should be
223           `VARCHAR(100)` and `NOT NULL`.
224
225           HINT: `UPDATE` first, then `NOT NULL` since a new column has no
226           values).
227
228           First we need to add the column to the table.
229  *************************************************************************** */
230
231 ALTER TABLE AP1.ContactUpdates
232 ADD Email VARCHAR(100);
233
234
235 /* ***************************************************************************
236     3.02. Populate the column (every field).
237
238           If we use `LastName` as part of the email, we should remove the
239           apostrophe in `O'Sullivan`.  Make sure to push the new values to an
240           existing row in lower case.
241
242           HINT: `UPDATE`
243
244           Note below the changes that happen when using `REPLACE()` to change
245           certain characters to empty strings in order to make the email
246           accounts using `VendorName`.
247
248           Note that, if `REPLACE()` does not find the string that we ask the
249           function to replace, the value in unchanged.  In the email shown
250           below, there is no ampersand (`&`) so the value is unchanged in the
251           third (3rd) instance of `REPLACE()` before it continues to the next
252           instance.  The same happens in the fifth (5th) instance when
253           searching for apostrophes (`''`, called with two single quotes as an
254           escape character).
255  *************************************************************************** */
256
257 UPDATE AP1.ContactUpdates
258 SET Email=LOWER(
259             REPLACE(
260               REPLACE(
261                 REPLACE(
262                   REPLACE(
263                     CONCAT (
264                       FirstName,
265                       '.',
266                       LastName,
267                       '@',
268                       VendorName,
269                       '.foo'
270                     ),      -- 1. `Anthony.Antavius@Courier Companies, Inc.foo`
271                   ' ', ''), -- 2. `Anthony.Antavius@CourierCompanies,Inc.foo`
272                 '&', ''),   -- 3. `Anthony.Antavius@CourierCompanies,Inc.foo`
273               ',', ''),     -- 4. `Anthony.Antavius@CourierCompaniesInc.foo`
274             '''', '')       -- 5. `anthony.antavius@couriercompaniesinc.foo`
275           )                 -- 6. `anthony.antavius@couriercompaniesinc.foo`
276                             --    (no change in the `Anthony Antavius` example)
```

```
277   FROM AP1.ContactUpdates
278   INNER JOIN AP1.Vendors
279     ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;
280   FROM AP1.ContactUpdates
281   INNER JOIN AP1.Vendors
282     ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;
283
284
285   /* ************************************************************************
286             Since record with `VendorID` 76 does not have a related `VendorName`
287             (hence no means to make an email), we can assign a value.  In this
288             case, We can assign `NO EMAIL` to that record.
289    ************************************************************************ */
290
291   UPDATE AP1.ContactUpdates
292   SET Email = 'NO EMAIL'
293   WHERE VendorID = 76
294
295
296   /* ************************************************************************
297             At this point, we change the column to `NOT NULL`.
298    ************************************************************************ */
299
300   ALTER TABLE AP1.ContactUpdates
301   ALTER COLUMN Email VARCHAR(100) NOT NULL;
302
303
304   /* ************************************************************************
305       3.03. Add column `VendorAddress` to `AP1.Vendors`, which should be
306             `VARCHAR(150)` and `NOT NULL`.
307    ************************************************************************ */
308
309   ALTER TABLE AP1.Vendors
310   ADD VendorAddress VARCHAR(150);
311
312
313   /* ************************************************************************
314             Then we move the values of `VendorAddress1` and `VendorAddress2` to
315             `VendorAddress`.
316    ************************************************************************ */
317
318   UPDATE AP1.Vendors
319   SET VendorAddress = CONCAT (
320       VendorAddress1,
321       ' ',
322       VendorAddress2
323       );
324
325
326   /* ************************************************************************
327             Then we make sure the new column has the data and delete the original
328             two columns.
329    ************************************************************************ */
330
331   ALTER TABLE AP1.Vendors
332   DROP COLUMN VendorAddress1;
333
334   ALTER TABLE AP1.Vendors
335   DROP COLUMN VendorAddress2;
336
337
338   /* ************************************************************************
339             Then we change the new column to `NOT NULL`.
340    ************************************************************************ */
341
342   ALTER TABLE AP1.Vendors
343   ALTER COLUMN VendorAddress VARCHAR(150) NOT NULL;
344
345
```

```sql
/* ********************************************************************************
    3.04. Call all the values from `AP1.ContactUpdates` with any corresponding
          values in `AP1.Vendors.

          HINT: `LEFT JOIN` to get 8 records
 ********************************************************************************* */

SELECT DISTINCT *
FROM AP1.ContactUpdates
LEFT JOIN AP1.Vendors
  ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;


/* ********************************************************************************
    3.05. Make a view named `AP1.ContactUpdatesVendorsVW` from the prior query.
 ********************************************************************************* */

CREATE VIEW AP1.ContactUpdatesVendorsVW
AS
(
    SELECT DISTINCT AP1.ContactUpdates.VendorID,
      AP1.ContactUpdates.LastName,
      AP1.ContactUpdates.FirstName,
      AP1.ContactUpdates.Email,
      -- AP1.Vendors.VendorID AS Expr1,
      AP1.Vendors.VendorName,
      AP1.Vendors.VendorCity,
      AP1.Vendors.VendorState,
      AP1.Vendors.VendorZipCode,
      AP1.Vendors.VendorPhone,
      AP1.Vendors.VendorContactLName,
      AP1.Vendors.VendorContactFName,
      AP1.Vendors.DefaultTermsID,
      AP1.Vendors.DefaultAccountNo,
      AP1.Vendors.VendorAddress
    FROM AP1.ContactUpdates
    LEFT JOIN AP1.Vendors
      ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID
    );


/* ********************************************************************************
          As an alternative, we can use an alias (`AS`) for each table to make
          our code tidier and to avoid repeating the database and schema before
          each table (`<database>.<schema>.<table>` or `<schema>.<table>`)
          throughout the query.

                    `c` for `AP1.ContactUpdates`
                    `v` for `AP1.Vendors`
 ********************************************************************************* */

ALTER VIEW AP1.ContactUpdatesVendorsVW
AS
(
    SELECT DISTINCT c.VendorID,
      c.LastName,
      c.FirstName,
      c.Email,
      v.VendorName,
      v.VendorCity,
      v.VendorState,
      v.VendorZipCode,
      v.VendorPhone,
      v.VendorContactLName,
      v.VendorContactFName,
      v.DefaultTermsID,
      v.DefaultAccountNo,
      v.VendorAddress
    FROM AP1.ContactUpdates AS c
```

```
    LEFT JOIN AP1.Vendors AS v
      ON c.VendorID = v.VendorID
    );


/* ***************************************************************************
          We can also make the view in schema `lab09` database `labs`.
 *************************************************************************** */

CREATE SCHEMA lab09;

CREATE VIEW lab09.ContactUpdatesVendorsVW
AS
(
    SELECT DISTINCT
      WS24SQL10001.AP1.ContactUpdates.VendorID, -- calling database
      WS24SQL10001.AP1.ContactUpdates.LastName, -- `WS24SQL10001` since the
      WS24SQL10001.AP1.ContactUpdates.FirstName,-- view is in database `labs`
      WS24SQL10001.AP1.ContactUpdates.Email,
      -- WS24SQL10001.AP1.Vendors.VendorID AS Expr1,
      WS24SQL10001.AP1.Vendors.VendorName,
      WS24SQL10001.AP1.Vendors.VendorCity,
      WS24SQL10001.AP1.Vendors.VendorState,
      WS24SQL10001.AP1.Vendors.VendorZipCode,
      WS24SQL10001.AP1.Vendors.VendorPhone,
      WS24SQL10001.AP1.Vendors.VendorContactLName,
      WS24SQL10001.AP1.Vendors.VendorContactFName,
      WS24SQL10001.AP1.Vendors.DefaultTermsID,
      WS24SQL10001.AP1.Vendors.DefaultAccountNo,
      WS24SQL10001.AP1.Vendors.VendorAddress
    FROM WS24SQL10001.AP1.ContactUpdates
    LEFT JOIN WS24SQL10001.AP1.Vendors
      ON WS24SQL10001.AP1.ContactUpdates.VendorID =
        WS24SQL10001.AP1.Vendors.VendorID
    );

/* ***************************************************************************
          As an alternative, we can use an alias (`AS`) for each table to make
          our code tidier and to avoid repeating the database and schema before
          each table (`<database>.<schema>.<table>` or `<schema>.<table>`)
          throughout the query.

                    `c` for `WS24SQL10001.AP1.ContactUpdates`
                    `v` for `WS24SQL10001.AP1.Vendors`
 *************************************************************************** */

ALTER VIEW lab09.ContactUpdatesVendorsVW
AS
(
    SELECT DISTINCT c.VendorID,
      c.LastName,
      c.FirstName,
      c.Email,
      v.VendorName,
      v.VendorCity,
      v.VendorState,
      v.VendorZipCode,
      v.VendorPhone,
      v.VendorContactLName,
      v.VendorContactFName,
      v.DefaultTermsID,
      v.DefaultAccountNo,
      v.VendorAddress
    FROM WS24SQL10001.AP1.ContactUpdates AS c    -- quick change, adding the
    LEFT JOIN WS24SQL10001.AP1.Vendors AS v      -- database name in the `FROM`
      ON c.VendorID = v.VendorID                 -- clause rather than placing
    );                                           -- the name of the database
                                                 -- before each column

```

```
/* ***************************************************************************
 4. The following set of concepts that is good for we to know involve how
    humans communicate with the computer and vice versa.

        ``A command line interface (CLI) is a text-based user interface (UI)
        used to view and manage computer files.  Command line interfaces are
        also called command-line user interfaces, console user interfaces and
        character user interfaces...
        Before the mouse, users interacted with an operating system (OS) or
        application with a keyboard. Users typed commands in the command line
        interface to run tasks on a computer.
        Typically, the command line interface features a black box with white
        text.  The user responds to a prompt in the command line interface by
        typing a command.  The output or response from the system can include
        a message, table, list, or some other confirmation of a system or
        application action.
        Today, most users prefer the graphical user interface (GUI) offered by
        operating systems such as Windows, Linux and macOS.  Most current
        Unix-based systems offer both a command line interface and a graphical
        user interface.
        The MS-DOS operating system and the command shell in the Windows
        operating system are examples of command line interfaces.  In
        addition, programming languages can support command line interfaces,
        such as Python.``
        https://searchwindowsserver.techtarget.com/definition/command-line-interface-CLI

        ``A GUI (usually pronounced GOO-ee) is a graphical (rather than purely
        textual) user interface to a computer.  As we read this, we are
        looking at the GUI or graphical user interface of your particular Web
        browser.  The term came into existence because the first interactive
        user interfaces to computers were not graphical; they were
        text-and-keyboard oriented and usually consisted of commands we had
        to remember and computer responses that were infamously brief.  The
        command interface of the DOS operating system (which we can still get
        to from your Windows operating system) is an example of the typical
        user-computer interface before GUIs arrived.  An intermediate step in
        user interfaces between the command line interface and the GUI was the
        non-graphical menu-based interface, which let we interact by using a
        mouse rather than by having to type in keyboard commands.
        https://searchwindevelopment.techtarget.com/definition/GUI

 5. Now that we are going to start programability, we use parameters to pass
    values either to a SQL script and/or receiving parameters from external
    programs, built-in and/or user-defined procedures and/or functions.

        ``In information technology, a parameter (pronounced puh-RAA-meh-tuhr,
        from Greek for, roughly, through measure) is an item of information
        -- such as a name, a number, or a SELECT DISTINCT option -- that is
        passed to a program by a user or another program.  Parameters affect
        the operation of the program receiving them.``
        http://whatis.techtarget.com/definition/parameter

        ``Parameters can be passed to the stored procedures.  This makes the
        procedure dynamic.
        The following points are to be noted:
        * One or more number of parameters can be passed in a procedure.
        * The parameter name should proceed with an @ symbol.
        * The parameter names will be local to the procedure in which they are
          defined.
        The parameters are used to pass information into a procedure from the
        line that executes the parameter.  The parameters are given just after
        the name of the procedure on a command line.  Commas should separate
        the list of parameters.
        * The values can be passed to stored procedures by:
        * By supplying the parameter values exactly in the same order as given
          in the CREATE PROCEDURE statement.
        * By explicitly naming the parameters and assigning the appropriate
          value.``
```

```
553           http://devguru.com/technologies/t-sql/7132
554
555       Every time users pass values to a query commonly using a web form, we are
556       at risk of SQL injections where the user could pass a SQL statement, which
557       the server may execute.  For this reason, every database should have a
558       read-only account that queries data returning values to the front-end
559       application limiting the possibility of SQL injections and similar exploits
560       (http://searchsecurity.techtarget.com/definition/exploit).
561
562            ``SQL injection is a type of security exploit in which the attacker
563            adds Structured Query Language (SQL) code to a Web form input box to
564            gain access to resources or make changes to data.  An SQL query is a
565            request for some action to be performed on a database.  Typically, on
566            a Web form for user authentication, when a user enters their name and
567            password into the text boxes provided for them, those values are
568            inserted into a SELECT DISTINCT query.  If the values entered are
569            found as expected, the user is allowed access; if they aren't found,
570            access is denied.  However, most Web forms have no mechanisms in place
571            to block input other than names and passwords.  Unless such
572            precautions are taken, an attacker can use the input boxes to send
573            their own request to the database, which could allow them to download
574            the entire database or interact with it in other illicit ways.``
575            http://searchsoftwarequality.techtarget.com/definition/SQL-injection
576
577       5.01. In the example below, we first declare parameter `@foo` and `@pi`
578             (always starting with the `@` sign) to tell SQL Server that it should
579             expect a value within SQL script and/or receiving a value from an
580             external programs, built-in and/or user-defined procedure and/or
581             function.
582
583             Note that there are no limit to the number of parameters we can use
584             in a SQL script.
585
586             Go to https://en.wikipedia.org/wiki/Foobar if we are interested about
587             terms `foobar`, `foo` and `bar`.
588
589             We first declare parameters `@foo` with data type INT and `@pi` with
590             data type FLOAT.
591
592             We then set (assign) values to parameters `@foo` and `@pi`.
593
594             We can then call the value of `@foo` and `@pi` within a SQL
595             statement.
596    ************************************************************************ */
597
598   DECLARE @foo INT = 3,                              -- 1. declaring parameter
599                                                      --    `@foo` as INT initialized
600                                                      --    with value of 3
601     @pi FLOAT = 3.14159265358979;                    -- 2. declaring parameter `@pi`
602                                                      --    as FLOAT initialized with
603                                                      --    value of 3.14159265358979
604
605
606   /* ************************************************************************
607       5.02. At this point, we can call the values of parameters `@foo` and `@pi`
608             in a SQL statement.
609
610                      +-------+------------------+--------------------+
611                      | foo   | pi(e)            | foo pi(e)          |
612                      +-------+------------------+--------------------+
613                      | 3     | 3.14159265358979 | 9.42477796076937   |
614                      +-------+------------------+--------------------+
615
616             Note that we use a `SELECT DISTINCT` statement in the same way we use
617             `PRINT` in other languages to show (not return, as we would have no
618             access to the value) the output in the console.
619    ************************************************************************ */
620
621   SELECT DISTINCT @foo AS 'foo',                     -- 1. displaying value of
```

```
                                                --    `@foo` (INT) with alias
                                                --    `foo`
  @pi AS 'pi(e)',                               -- 2. displaying value of
                                                --    `@pi` (FLOAT) with alias
                                                --    `pi(e)`
  @foo * @pi AS 'foo pi(e)';                    -- 3. displaying value of
                                                --    `@foo` multiplied by
                                                --    `@pi`, which returns a
                                                --    FLOAT due to data type
                                                --    conversion with alias
                                                --    `foo pi(e)`


/* ************************************************************************
 6. ``In SQL Server, a procedure is a stored program that you can pass
    parameters into.  It does not return a value like a function does.
    However, it can return a success/failure status to the procedure that
    called it.``
    https://techonthenet.com/sql_server/procedures.php

                    CREATE PROCEDURE procedure_name [@input_param data_type]
                    AS
                    BEGIN
                      [DECLARE @output_param data_type
                      SET @output_param = some_value]
                      executable_code
                    END;

    This means that we can take the code that we used to capitalize the first
    letter in a string and make it into a procedure that we can call indicating
    the input parameter instead of writing the same code several times and
    avoid the possibility of errors.

                    SELECT DISTINCT CONCAT (
                      UPPER(LEFT(`hello`, 1)),
                      LOWER(SUBSTRING(`hello`, 2, LEN(`hello`) - 1))
                      );

    6.01. In the example below, we declare function `AP5.properUDP`.  We end
          the name of the function with `UDP` to identify it as an user-defined
          procedure.

          The procedure has input parameter `@in_string` declared as
          VARCHAR(50) -- in this case, the string `hello`.

          We enclose the executable section between `BEGIN` and `END`.

          We create output using parameter `@out_string`, which must have the
          same file type as the input, in order to print (not return) the value
          of `hello` as `Hello`.

          Then we pass the value of `UPPER(LEFT(@in_string, 1)) +
          LOWER(SUBSTRING(@in_string, 2, LEN(@in_string)-1))` to parameter
          `@out_string`.
 ************************************************************************ */

CREATE SCHEMA AP5;                              -- 1. creating schema `AP5` if
                                                --    not created already

CREATE PROCEDURE AP5.properUDP                  -- 2. creating stored procedure
                                                --    `AP5.properUDP`
  @in_string VARCHAR(50)                        -- 3. declaring input parameter
                                                --    '@in_string` with data
                                                --    type `VARCHAR(50)`
AS
BEGIN                                           -- 4. beginning of executable
                                                --    code
  DECLARE @out_string VARCHAR(50)               -- 5. declaring output
                                                --    parameter `@out_string`
```

```
                                                --    with same data type as
                                                --    input parameter
                                                --    `@in_string` with data
                                                --    type `VARCHAR(50)`
  SET @out_string = CONCAT (                    -- 6. setting value of output
      UPPER(LEFT(@in_string, 1)),               --    parameter `@out_string`
      LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
      )
  PRINT @out_string;                            -- 7. printing (not returning)
                                                --    value of `@out_string`
END;                                            -- 8. end of executable code
                                                --    and stored procedure


/* ************************************************************************
    6.02. In order to execute (`EXEC`) our user-defined procedure (`UDP`), we
          must indicate the schema where it resides -- in this case, `AP5`.

          Since the output is printed (displayed only) and not returned, we
          cannot use the value by the procedure (`AP5.properUDP`).
 ************************************************************************ */

EXEC AP5.properUDP @in_string = 'hELLO';


/* ************************************************************************
    6.03. Procedures do not need input and/or output parameters if the
          executable code does not need parameters in order to work.

          In the example below, we have procedure `AP5.create_tempUDP` to
          create database `TEMP` and prints a message when it has been
          completed without the need of parameters.
 ************************************************************************ */

CREATE PROCEDURE AP5.create_tempUDP             -- 1. creating stored procedure
AS                                              --    `AP1.create_tempUDP`
BEGIN                                           -- 2. beginning of procedure
                                                --    executable code
  CREATE DATABASE TEMP                          -- 3. creating database `TEMP`
  PRINT 'Database Complete'                     -- 4. message displayed (not
                                                --    returned)
END;                                            -- 5. end of executable code
                                                --    and stored procedure

EXEC AP5.create_tempUDP;                        -- 6. executing procedure;
                                                --    no parameters needed in
                                                --    this example

/* ************************************************************************
    6.04. In the example below, we have procedure `AP1.drop_tempUDP` to create
          database `TEMP` and prints a message when it has been completed
          without the need of parameters.
 ************************************************************************ */

CREATE PROCEDURE AP5.drop_tempUDP               -- 1. creating stored procedure
AS                                              --    `AP1.drop_tempUDP`
BEGIN                                           -- 2. beginning of executable
                                                --    code
  DROP DATABASE TEMP                            -- 3. dropping database `TEMP`
  PRINT 'Database Dropped'                      -- 4. message displayed (not
                                                --    returned)
END;                                            -- 5. end of executable code
                                                --    and stored procedure

EXEC AP5.drop_tempUDP;                          -- 6. executing procedure;
                                                --    no parameters needed in
                                                --    this example
```

```
/* ************************************************************************
    6.05. As with all types of data objects, we can DROP procedures too.
 ************************************************************************ */

DROP PROCEDURE AP5.create_tempUDP;              -- dropping procedure


/* ************************************************************************
 7. In the two examples below, we have two procedures to change Celsius to
    Fahrenheit and vice versa.

    7.01. We first create schema `temps` in the `labs` database.
 ************************************************************************ */

CREATE SCHEMA temps;


/* ************************************************************************
    7.02. We create procedure `temps.c2f` taking in one parameter declared as a
          FLOAT to convert temperatures in Celsius to Fahrenheit.
 ************************************************************************ */

CREATE PROCEDURE temps.c2f @in_temp FLOAT       -- 1. input parameter
                                                --    initialized as a FLOAT
AS
BEGIN
  -- formula needed (9/5 C) + 32
  DECLARE @out_temp FLOAT                       -- 2. declaring output
                                                --    parameter `@out_temp`
                                                --    with the same datatype as
                                                --    `@in_temp`, in this case
                                                --    a FLOAT
  SET @out_temp = (9 / 5 * @in_temp) + 32       -- 3. formula to convert
                                                --    Celsius to Fahrenheit
                                                --    including `@in_temp`
  DECLARE @out_result VARCHAR(150)              -- 4. new output to take the
                                                --    the value of
  SET @out_result = CONCAT (                    -- 5. passing values including
      CONVERT(VARCHAR(25), @in_temp),           --    `@in_temp` (temperature
      'C = ',                                   --    in Celsius), `@out_temp`
      CONVERT(VARCHAR(25), @out_temp),          --    `@out_temp` (temperature
      'F'                                       --    in Fahrenheit)
      )
  PRINT @out_result                             -- 6. printing value to screen
END;

EXEC temps.c2f 75;                              -- 7. executing procedure
                                                --    `temps.c2f` passing 75
                                                --    as temperature in Celsius
                                                --    returning `75C = 107F`


/* ************************************************************************
    7.03. We create procedure `temps.f2c` taking in one parameter declared as a
          FLOAT to convert temperatures in Fahrenheit to Celsius.
 ************************************************************************ */

CREATE PROCEDURE temps.f2c @in_temp FLOAT       -- 1. input parameter
                                                --    initialized as a FLOAT
AS
BEGIN
  -- formula needed 5/9(F - 32)
  DECLARE @out_temp FLOAT                       -- 2. declaring output
                                                --    parameter `@out_temp`
                                                --    with the same datatype as
                                                --    `@in_temp`, in this case
                                                --    a FLOAT
  SET @out_temp = (@in_temp - 32) * 5 / 9       -- 3. formula to convert
                                                --    Fahrenheit to Celsius
```

```
                                                    --    including `@in_temp`
  DECLARE @out_result VARCHAR(150)                  -- 4. new output to take the
                                                    --    the value of
  SET @out_result = CONCAT (                        -- 5. passing values including
      CONVERT(VARCHAR(25), @in_temp),               --    `@in_temp` (temperature
      'F = ',                                       --    in Fahrenheit),
      CONVERT(VARCHAR(25), @out_temp),              --    `@out_temp` (temperature
      'C'                                           --    in Celsius)
      )
  PRINT @out_result                                 -- 6. printing value to screen
END;

EXEC temps.f2c 73;                                  -- 7. executing procedure
                                                    --    `temps.f2c` passing 73
                                                    --    as temperature in
                                                    --    Fahrenheit returning
                                                    --    `73F = 22.7778C`


/* ***************************************************************************
 8. In SQL Server, a function is a stored program that we can pass parameters
    into and return a value.
    https://techonthenet.com/sql_server/functions.php

                    CREATE FUNCTION function_name (@input_param data_type)
                    RETURNS data_type
                    AS
                    BEGIN
                      DECLARE @output_param data_type
                      SET @output_param = some_value
                      executable_code
                      RETURN output_param
                    END;

    This also means that we can take the code that we used to capitalize the
    first letter in a string and make it into a function that we can call
    instead of writing the same code several times and avoid the possibility of
    errors.

                    SELECT DISTINCT CONCAT (
                      UPPER(LEFT(`hello`, 1)),
                      LOWER(SUBSTRING(`hello`, 2, LEN(`hello`) - 1))
                      );

    8.01. In the example below, we create function `AP5.properUDF`.  We end the
          name of the function with `UDF` to identify it as an user-defined
          function.  As explained before, no two objects of the same hierarchy
          can have the same name.  Therefore our user-defined procedure and
          function cannot share the name (`AP5.proper`) and a suffix tells
          the system which object to use.

          The function has input parameter `@in_string` declared as VARCHAR(50)
          -- in this case, the string `hello`.

          We enclose the executable section between `BEGIN` and `END`.

          We create output using parameter `@out_string`, which must have the
          same file type as the input parameter, in order to return the value
          of `hello` as `Hello`.

          Then we pass the value of concatenation

                    CONCAT(
                      UPPER(
                        LEFT(@in_string, 1)
                        ),
                      LOWER(
                        SUBSTRING(@in_string, 2,
                          LEN(@in_string) - 1)
```

```
                          )

          or concatenation using the `+` sign (considering that adding a value
          to NULL returns NULL, unless using the proper `CASE` clause)

                    UPPER(
                      LEFT(@in_string, 1)
                      ) +
                     LOWER(
                       SUBSTRING(@in_string, 2,
                        LEN(@in_string) - 1)
                        )

          to parameter `@out_string`.

          As the last step, we must indicate what value must be returned from
          the function -- in this case, parameter `@out_string`.
 ************************************************************************** */

CREATE FUNCTION AP5.properUDF                    -- 1. creating function
                                                 --    `AP5.properUDF`
  (@in_string VARCHAR(50))                       -- 2. declaring input parameter
                                                 --    `@in_string` as
                                                 --    VARCHAR(50)
RETURNS VARCHAR(50)                              -- 3. indicating the same data
                                                 --    type and size of output
                                                 --    parameter `@out_string`
AS
BEGIN                                            -- 4. beginning of executable
                                                 --    code
  DECLARE @out_string VARCHAR(50)                -- 5. declaring output
                                                 --    parameter `@out_string`
                                                 --    with same data type as
                                                 --    input parameter
                                                 --    `@in_string`; same data
                                                 --    type and size as
                                                 --    indicated after `RETURNS`
  SET @out_string = CONCAT (                     -- 6. setting value to output
      UPPER(LEFT(@in_string, 1)),                --    parameter `@out_string`
      -- LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
      LOWER(RIGHT(@in_string, LEN(@in_string) - 1))
      )
  RETURN @out_string;                            -- 7. returning value of output
                                                 --    parameter `@out_string`
END;                                             -- 8. end of executable code
                                                 --    and function


/* ************************************************************************
    8.02. In order to call our user-defined function (`UDF`), we must indicate
          the schema where it resides -- in this case, `AP5`.
 ************************************************************************** */

SELECT DISTINCT AP5.properUDF('hello');


/* ************************************************************************
    8.03. We can use `AP5.properUDF` on any string value in any table, schema
          or database as long as we have access to the data objects -- for
          example, columns `AP2.Customers.FirstName` and
          `AP2.Customers.LastName`.  Of course, first we insert values into
          `AP2.Customers`.
 ************************************************************************** */

INSERT INTO AP2.Customers                        -- all values in order
VALUES (
  1,
  'Smith',
  'John',
```

```
    '',
    '',
    '',
    '',
    ''
    ),
    (
    2,
    'Doe',
    'Jane',
    '123 Main St. Apt. 1',
    'New York',
    'NY',
    '10001',
    'jane.doe@foobar.foo'
    );

  INSERT INTO AP2.Customers (                       -- some values specifying the
    CustomerID,                                     -- order of the fields
    LastName,
    FirstName
    )
  VALUES (
    3,
    'Smith',
    'Tom'
    );

  INSERT INTO AP2.Customers
  VALUES (
    5,
    'Doe',
    'John',
    '',
    'New York',
    'NY',
    '10001',
    'john.doe@foobar.foo'
    ),
    (
    4,
    'Doe',
    'Jane',
    '',
    'New York',
    'NY',
    '',
    'jane.doe2@foobar.foo'
    );

  UPDATE AP2.Customers
  SET FirstName = AP5.properUDF(FirstName),
    LastName = AP5.properUDF(LastName);


  /* ***********************************************************************
      8.04. We can also create functions to FORMAT dollar amounts
            (`AP5.dollarUDF`) and dates (`AP5.dateUDF`) considering that numeric
            values become strings when formatted.
   *********************************************************************** */

  CREATE FUNCTION AP5.dollarUDF (@in_dollar FLOAT)
  RETURNS VARCHAR(50)
  AS
  BEGIN
    DECLARE @out_dollar VARCHAR(50)
    SET @out_dollar = FORMAT(@in_dollar, 'c', 'en-us')
    RETURN @out_dollar
  END;
```

```
1036
1037  SELECT DISTINCT AP5.dollarUDF(10000) AS FormattedDollarAmout;
1038
1039  CREATE FUNCTION AP5.dateUDF (@in_date DATE)
1040  RETURNS VARCHAR(10)
1041  AS
1042  BEGIN
1043    DECLARE @out_date VARCHAR(10)
1044    SET @out_date = FORMAT(@in_date, 'd', 'en-us')
1045    RETURN @out_date
1046  END;
1047
1048  SELECT DISTINCT AP5.dateUDF(GETDATE()) AS FormattedDate;
1049
1050
1051  /* ************************************************************************
1052      8.05. Going back to procedures, we can call user-defined functions inside
1053            user-defined procedures (commonly referred to as stored procedures).
1054            We can call `AP5.properUDF` on `AP2.Customers.FirstName` and
1055            `AP2.Customers.LastName`.
1056   ************************************************************************ */
1057
1058  CREATE PROCEDURE AP5.properUDP                  -- 1. creating stored procedure
1059                                                  --    without input parameters
1060  AS
1061  BEGIN                                           -- 2. beginning of executable
1062                                                  --    code without parameters
1063    UPDATE AP2.Customers                          -- 3. updating `AP2.Customers`
1064    SET FirstName = AP5.properUDF(FirstName);
1065    PRINT 'Proper case assigned to first names';  -- 4. printing message
1066    UPDATE AP2.Customers                          -- 5. updating `AP2.Customers`
1067    SET LastName = AP5.properUDF(LastName);
1068    PRINT 'Proper case assigned to last names';   -- 6. printing message
1069  END;                                            -- 7. end of executable code
1070                                                  --    and stored procedure
1071
1072  CREATE PROCEDURE AP5.CloneInvoicesUDP           -- 1. creating stored procedure
1073                                                  --    `AP1.CloneInvoicesUDP`
1074  AS
1075  BEGIN                                           -- 2. beginning of executable
1076                                                  --    code
1077    DROP TABLE AP1.CloneInvoices;                 -- 3. dropping old clone table
1078    PRINT 'Old table `AP1.Invoices` destroyed';   -- 4. displaying completion
1079                                                  --    message
1080    SELECT DISTINCT                               -- 5. SELECT DISTINCT all
1081      InvoiceID,                                  --    values from
1082      VendorID,                                   --    `AP1.Invoices`
1083      InvoiceNumber,
1084      AP5.dateUDF(InvoiceDate)                    -- 6. calling date values of
1085        AS InvoiceDate,                           --    columns using
1086      AP5.dollarUDF(InvoiceTotal)                 --    user-defined functions
1087        AS InvoiceTotal,                          --    `AP5.dateUDF` and
1088      AP5.dollarUDF(PaymentTotal)                 --    `AP5.dollarUDF`
1089        AS PaymentTotal,                          --
1090      AP5.dollarUDF(CreditTotal)
1091        AS CreditTotal,
1092      TermsID,
1093      AP5.dateUDF(InvoiceDueDate)
1094        AS InvoiceDueDate,
1095      AP5.dateUDF(PaymentDate)
1096        AS PaymentDate                            -- 7. pushing values from old
1097    INTO AP1.CloneInvoices                        --    table `AP1.Invoices` to
1098                                                  --    new table
1099                                                  --    `AP1.CloneInvoices`
1100                                                  --    (T-SQL extension, not
1101                                                  --    ANSI-complaint)
1102    FROM AP1.Invoices;                            -- 8. from `AP1.Invoices`
1103    PRINT 'New table `AP5.Invoices` created';     -- 9. displaying completion
1104                                                  --    message
```

```
1105  END;                                                -- 10. end of executable code
1106                                                      --     and stored procedure
1107
1108  EXEC AP5.CloneInvoicesUDP;
1109
1110
1111  /* ***************************************************************************
1112   https://folvera.commons.gc.cuny.edu/?p=1291
1113   ************************************************************************** */
```