

```

1  /* *****
2      INTRODUCTION TO STRUCTURED QUERY LANGUAGE FOR DATA ANALYTICS
3          SF25SQL6172025, 2025/11/17 - 2025/12/17
4          https://folvera.commons.gc.cuny.edu/?cat=35
5  *****
6
7  SESSION #8 (2025/12/15): CREATING DATABASE OBJECTS
8
9  1. Altering databases, schemata, tables
10 2. Understanding `NULL` and `NOT NULL`
11 3. Parameters, user-defined functions and stored procedures
12 *****
13
14 1. LAB #7
15 Write a query without duplicate rows (`SELECT DISTINCT`)
16 1.01. to get all shared values from tables `AP1.InvoiceLineItems` and
17       `AP1.GLAccounts` (`INNER JOIN`),
18 1.02. adding today's date as `TodaysDate` formatted as short date
19 1.03. where `AP1.GLAccounts.AccountDescription` starts with `book`
20       (`AP1.GLAccounts.AccountDescription LIKE('book%')`) and
21       `AP1.InvoiceLineItems.InvoiceLineItemAmount` is at least 1000.00
22       (inclusive) -- first condition composed of two conditions
23 1.04. or where `AP1.GLAccounts.AccountDescription` contains `mail` and
24       `AP1.InvoiceLineItems.InvoiceLineItemAmount` is no more than
25       100.00 (inclusive) -- second condition composed of two conditions
26       (second condition in parenthesis (OR secondary_codition1 AND
27       secondary_condition2))
28 1.05. ordered first by `AP1.GLAccounts.AccountDescription` and then by
29       `AP1.InvoiceLineItems.InvoiceLineItemAmount`.
30 1.06. Then make a view in schema `lab07` in database `labs`.
31 ***** */
32
33 SELECT DISTINCT AP1.InvoiceLineItems.InvoiceID,
34 AP1.InvoiceLineItems.InvoiceSequence,
35 AP1.InvoiceLineItems.AccountNo,
36 AP1.InvoiceLineItems.InvoiceLineItemAmount,
37 AP1.InvoiceLineItems.InvoiceLineItemDescription,
38 -- AP1.GLAccounts.AccountNo AS Expr1,
39 AP1.GLAccounts.AccountDescription
40 /*,
41 FORMAT(GETDATE(), 'd', 'en-us') AS TodaysDate*/
42 FROM AP1.InvoiceLineItems
43 INNER JOIN AP1.GLAccounts
44 ON AP1.InvoiceLineItems.AccountNo = AP1.GLAccounts.AccountNo
45 WHERE
46 (
47     -- 1. first block of two
48     -- conditions that must be
49     -- true
50     AP1.GLAccounts.AccountDescription LIKE 'book%'
51     AND AP1.InvoiceLineItems.InvoiceLineItemAmount >= 1000
52 )
53 OR
54 -- 2. `OR` to indicate that
55 -- either the first block
56 -- (above) or the second
57 -- (below) must be true
58 (
59     -- 3. second block of two
60     -- conditions that must be
61     -- true
62     AP1.GLAccounts.AccountDescription LIKE '%mail%'
63     AND AP1.InvoiceLineItems.InvoiceLineItemAmount <= 100
64 )
65 ORDER BY AP1.GLAccounts.AccountDescription,
66 AP1.InvoiceLineItems.InvoiceLineItemAmount,
67 AP1.InvoiceLineItems.InvoiceID,
68 AP1.InvoiceLineItems.InvoiceSequence,
69 AP1.InvoiceLineItems.AccountNo,
70 AP1.InvoiceLineItems.InvoiceLineItemDescription;

```

```

70 /* *****
71      At this point, we make a view in schema `lab07` in database `labs`
72      excluding the `ORDER BY` clause as it cannot be used when creating
73      views.
74      ***** */
75
76 CREATE VIEW lab07.InvoiceLineItemsGLAccountsVW
77 AS
78 SELECT DISTINCT SF25SQL6172025.AP1.InvoiceLineItems.InvoiceID,
79 SF25SQL6172025.AP1.InvoiceLineItems.InvoiceSequence,
80 SF25SQL6172025.AP1.InvoiceLineItems.AccountNo,
81 SF25SQL6172025.AP1.InvoiceLineItems.InvoiceLineItemAmount,
82 SF25SQL6172025.AP1.InvoiceLineItems.InvoiceLineItemDescription,
83 -- SF25SQL6172025.AP1.GLAccounts.AccountNo AS Expr1,
84 SF25SQL6172025.AP1.GLAccounts.AccountDescription
85 FROM SF25SQL6172025.AP1.InvoiceLineItems
86 INNER JOIN SF25SQL6172025.AP1.GLAccounts
87     ON SF25SQL6172025.AP1.InvoiceLineItems.AccountNo =
88     SF25SQL6172025.AP1.GLAccounts.AccountNo
89 WHERE (
90     SF25SQL6172025.AP1.GLAccounts.AccountDescription LIKE 'book%'
91     AND SF25SQL6172025.AP1.InvoiceLineItems.InvoiceLineItemAmount >= 1000
92 )
93 OR (
94     SF25SQL6172025.AP1.GLAccounts.AccountDescription LIKE '%mail%'
95     AND SF25SQL6172025.AP1.InvoiceLineItems.InvoiceLineItemAmount <= 100
96 );
97
98
99 /* *****
100      As an alternative, we can use an alias (`AS`) for each table to make
101      our code tidier and to avoid repeating the database and schema before
102      each table (`<database>.<schema>.<table>` or `<schema>.<table>`)
103      throughout the query.
104
105      `i` for `SF25SQL6172025.AP1.InvoiceLineItems`
106      `g` for `SF25SQL6172025.AP1.GLAccounts`
107      ***** */
108
109 CREATE VIEW lab07.InvoiceLineItemsGLAccountsVW
110 AS
111 SELECT DISTINCT i.InvoiceID,
112     i.InvoiceSequence,
113     i.AccountNo,
114     i.InvoiceLineItemAmount,
115     i.InvoiceLineItemDescription,
116     g.AccountDescription
117 FROM SF25SQL6172025.AP1.InvoiceLineItems AS i
118 INNER JOIN SF25SQL6172025.AP1.GLAccounts AS g
119     ON i.AccountNo = g.AccountNo
120 WHERE (
121     g.AccountDescription LIKE 'book%'
122     AND i.InvoiceLineItemAmount >= 1000
123 )
124 OR (
125     g.AccountDescription LIKE '%mail%'
126     AND i.InvoiceLineItemAmount <= 100
127 );
128
129
130 /* *****
131 2. LAB #8 (CREATING OBJECTS)
132 2.01. Create database `labs`.
133 2.02. Create schema `lab08` in database `labs`.
134 2.03. Create table `my_family` in schema `lab08` with the following
135 structure choosing the best data type for each column and assign
136 `NOT NULL` to each.
137
138     row_id

```

```

139         person_fname
140         person_lname
141         relation
142
143     2.04. Insert values accordingly.
144     2.05. Modify table `my_family` to add a column `dob`.
145     2.06. Update the table with data in `dob` (new values in an existing record
146           in table `labs.lab08.my_family`).
147     2.07. Change column `dob` to `NOT NULL`.
148     ***** */
149
150 CREATE DATABASE labs; -- 1. creating database `labs`
151 -- 1.1. run #1 (all `CREATE
152 -- DATABASE` statements
153 -- run together, but
154 -- separately from
155 -- other statements)
156
157 CREATE SCHEMA lab08; -- 2. creating schema `lab08`
158
159 CREATE TABLE lab08.my_family ( -- 3. creating table
160     row_id INT NOT NULL, -- `lab08.my_family`
161     person_fname VARCHAR(25) NOT NULL, -- 3.1. run #3 (all `CREATE
162     person_lname VARCHAR(25) NOT NULL, -- TABLE` statement run
163     relation VARCHAR(15) NOT NULL -- together, but
164 ); -- separately from
165 -- other statements)
166
167 INSERT INTO lab08.my_family -- 4. inserting new values into
168 VALUES ( -- table `lab08.my_family`
169     1, -- 4.1. each row/record
170     'John', -- within a set of
171     'Doe', -- parenthesis followed
172     'crazy uncle' -- by a comma between
173 ), -- rows/records
174 ( -- 4.2. run #4 (all `INSERT`
175     2, -- statements run
176     'Michael', -- together, separately
177     'Jones', -- from other
178     'cousin' -- statements)
179 ),
180 (
181     3,
182     'Lucy',
183     'Smith',
184     'aunt'
185 );
186
187 ALTER TABLE lab08.my_family -- 5. altering table
188 ADD dob DATE; -- `lab08.my_family` to add
189 -- column `dob` with data
190 -- type `DATE`
191 -- 5.1. run #5 (all `ALTER`
192 -- statements run
193 -- together, separately
194 -- from other
195 -- statements)
196
197 UPDATE lab08.my_family -- 6. updating table
198 SET dob = '1970-01-01' -- `lab08.my_family` to pass
199 WHERE row_id = 1; -- a new values to column
200 -- `dob` in the existing
201 UPDATE lab08.my_family -- table `lab08.my_family`
202 SET dob = '1980/05/09' -- 6.1. run #6 (all `UPDATE`
203 WHERE row_id = 2; -- statements run
204 -- together, separately
205 UPDATE lab08.my_family -- from other
206 SET dob = '1988/08/19' -- statements)
207 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 is 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

```

```

346 /* *****
347     3.04. Call all the values from `AP1.ContactUpdates` with any corresponding
348         values in `AP1.Vendors`.
349
350         HINT: `LEFT JOIN` to get 8 records
351     ***** */
352
353 SELECT DISTINCT *
354 FROM AP1.ContactUpdates
355 LEFT JOIN AP1.Vendors
356     ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID;
357
358
359 /* *****
360     3.05. Make a view named `AP1.ContactUpdatesVendorsVW` from the prior query.
361     ***** */
362
363 CREATE VIEW AP1.ContactUpdatesVendorsVW
364 AS
365 (
366     SELECT DISTINCT AP1.ContactUpdates.VendorID,
367         AP1.ContactUpdates.LastName,
368         AP1.ContactUpdates.FirstName,
369         AP1.ContactUpdates.Email,
370         -- AP1.Vendors.VendorID AS Expr1,
371         AP1.Vendors.VendorName,
372         AP1.Vendors.VendorCity,
373         AP1.Vendors.VendorState,
374         AP1.Vendors.VendorZipCode,
375         AP1.Vendors.VendorPhone,
376         AP1.Vendors.VendorContactLName,
377         AP1.Vendors.VendorContactFName,
378         AP1.Vendors.DefaultTermsID,
379         AP1.Vendors.DefaultAccountNo,
380         AP1.Vendors.VendorAddress
381     FROM AP1.ContactUpdates
382     LEFT JOIN AP1.Vendors
383         ON AP1.ContactUpdates.VendorID = AP1.Vendors.VendorID
384     );
385
386
387 /* *****
388     As an alternative, we can use an alias (`AS`) for each table to make
389     our code tidier and to avoid repeating the database and schema before
390     each table (`<database>.<schema>.<table>` or `<schema>.<table>`)
391     throughout the query.
392
393         `c` for `AP1.ContactUpdates`
394         `v` for `AP1.Vendors`
395     ***** */
396
397 ALTER VIEW AP1.ContactUpdatesVendorsVW
398 AS
399 (
400     SELECT DISTINCT c.VendorID,
401         c.LastName,
402         c.FirstName,
403         c.Email,
404         v.VendorName,
405         v.VendorCity,
406         v.VendorState,
407         v.VendorZipCode,
408         v.VendorPhone,
409         v.VendorContactLName,
410         v.VendorContactFName,
411         v.DefaultTermsID,
412         v.DefaultAccountNo,
413         v.VendorAddress
414     FROM AP1.ContactUpdates AS c

```

```

415     LEFT JOIN AP1.Vendors AS v
416         ON c.VendorID = v.VendorID
417     );
418
419
420 /* *****
421     We can also make the view in schema `lab09` database `labs`.
422     ***** */
423
424 CREATE SCHEMA lab09;
425
426 CREATE VIEW lab09.ContactUpdatesVendorsVW
427 AS
428 (
429     SELECT DISTINCT
430         SF25SQL6172025.AP1.ContactUpdates.VendorID, -- calling database
431         SF25SQL6172025.AP1.ContactUpdates.LastName, -- `SF25SQL6172025` since the
432         SF25SQL6172025.AP1.ContactUpdates.FirstName, -- view is in database `labs`
433         SF25SQL6172025.AP1.ContactUpdates.Email,
434         -- SF25SQL6172025.AP1.Vendors.VendorID AS Expr1,
435         SF25SQL6172025.AP1.Vendors.VendorName,
436         SF25SQL6172025.AP1.Vendors.VendorCity,
437         SF25SQL6172025.AP1.Vendors.VendorState,
438         SF25SQL6172025.AP1.Vendors.VendorZipCode,
439         SF25SQL6172025.AP1.Vendors.VendorPhone,
440         SF25SQL6172025.AP1.Vendors.VendorContactLName,
441         SF25SQL6172025.AP1.Vendors.VendorContactFName,
442         SF25SQL6172025.AP1.Vendors.DefaultTermsID,
443         SF25SQL6172025.AP1.Vendors.DefaultAccountNo,
444         SF25SQL6172025.AP1.Vendors.VendorAddress
445     FROM SF25SQL6172025.AP1.ContactUpdates
446     LEFT JOIN SF25SQL6172025.AP1.Vendors
447         ON SF25SQL6172025.AP1.ContactUpdates.VendorID =
448            SF25SQL6172025.AP1.Vendors.VendorID
449     );
450
451 /* *****
452     As an alternative, we can use an alias (`AS`) for each table to make
453     our code tidier and to avoid repeating the database and schema before
454     each table (`<database>.<schema>.<table>` or `<schema>.<table>`)
455     throughout the query.
456
457         `c` for `SF25SQL6172025.AP1.ContactUpdates`
458         `v` for `SF25SQL6172025.AP1.Vendors`
459     ***** */
460
461 ALTER VIEW lab09.ContactUpdatesVendorsVW
462 AS
463 (
464     SELECT DISTINCT c.VendorID,
465         c.LastName,
466         c.FirstName,
467         c.Email,
468         v.VendorName,
469         v.VendorCity,
470         v.VendorState,
471         v.VendorZipCode,
472         v.VendorPhone,
473         v.VendorContactLName,
474         v.VendorContactFName,
475         v.DefaultTermsID,
476         v.DefaultAccountNo,
477         v.VendorAddress
478     FROM SF25SQL6172025.AP1.ContactUpdates AS c -- quick change, adding the
479     LEFT JOIN SF25SQL6172025.AP1.Vendors AS v -- database name in the `FROM`
480         ON c.VendorID = v.VendorID -- clause rather than placing
481     ); -- the name of the database
482 -- before each column
483

```

484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552

```
/* *****  
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://searchwindowserver.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.``
```

Every time users pass values to a query commonly using a web form, we are at risk of SQL injections where the user could pass a SQL statement, which the server may execute. For this reason, every database should have a read-only account that queries data returning values to the front-end application limiting the possibility of SQL injections and similar exploits (<http://searchsecurity.techtarget.com/definition/exploit>).

SQL injection is a type of security exploit in which the attacker adds Structured Query Language (SQL) code to a Web form input box to gain access to resources or make changes to data. An SQL query is a request for some action to be performed on a database. Typically, on a Web form for user authentication, when a user enters their name and password into the text boxes provided for them, those values are inserted into a SELECT DISTINCT query. If the values entered are found as expected, the user is allowed access; if they aren't found, access is denied. However, most Web forms have no mechanisms in place to block input other than names and passwords. Unless such precautions are taken, an attacker can use the input boxes to send their own request to the database, which could allow them to download the entire database or interact with it in other illicit ways.``  
<http://searchsoftwarequality.techtarget.com/definition/SQL-injection>

5.01. In the example below, we first declare parameter `@foo` and `@pi` (always starting with the `@` sign) to tell SQL Server that it should expect a value within SQL script and/or receiving a value from an external programs, built-in and/or user-defined procedure and/or function.

Note that there are no limit to the number of parameters we can use in a SQL script.

Go to <https://en.wikipedia.org/wiki/Foobar> if we are interested about terms `foobar`, `foo` and `bar`.

We first declare parameters `@foo` with data type INT and `@pi` with data type FLOAT.

We then set (assign) values to parameters `@foo` and `@pi`.

We can then call the value of `@foo` and `@pi` within a SQL statement.

```
***** */
DECLARE @foo INT = 3, -- 1. declaring parameter
-- @foo as INT initialized
-- with value of 3
@pi FLOAT = 3.14159265358979; -- 2. declaring parameter @pi
-- as FLOAT initialized with
-- value of 3.14159265358979
```

5.02. At this point, we can call the values of parameters `@foo` and `@pi` in a SQL statement.

foo	pi(e)	foo pi(e)
3	3.14159265358979	9.42477796076937

Note that we use a `SELECT DISTINCT` statement in the same way we use `PRINT` in other languages to show (not return, as we would have no access to the value) the output in the console.

```
***** */
SELECT DISTINCT @foo AS 'foo', -- 1. displaying value of
```







```

829
830 DECLARE @out_result VARCHAR(150)
831
832 SET @out_result = CONCAT (
833     CONVERT(VARCHAR(25), @in_temp),
834     'F = ',
835     CONVERT(VARCHAR(25), @out_temp),
836     'C'
837 )
838 PRINT @out_result
839 END;
840
841 EXEC temps.f2c 73;

```

-- including `@in\_temp`  
-- 4. new output to take the  
-- the value of  
-- 5. passing values including  
-- `@in\_temp` (temperature  
-- in Fahrenheit),  
-- `@out\_temp` (temperature  
-- in Celsius)  
-- 6. printing value to screen  
-- 7. executing procedure  
-- `temps.f2c` passing 73  
-- as temperature in  
-- Fahrenheit returning  
-- `73F = 22.7778C`

```

848 /* *****
849 8. In SQL Server, a function is a stored program that we can pass parameters
850 into and return a value.
851 https://techonthenet.com/sql\_server/functions.php
852

```

```

853     CREATE FUNCTION function_name (@input_param data_type)
854     RETURNS data_type
855     AS
856     BEGIN
857         DECLARE @output_param data_type
858         SET @output_param = some_value
859         executable_code
860         RETURN output_param
861     END;
862

```

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.

```

863
864     SELECT DISTINCT CONCAT (
865         UPPER(LEFT(`hello`, 1)),
866         LOWER(SUBSTRING(`hello`, 2, LEN(`hello`) - 1))
867     );
868

```

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

```

891     CONCAT(
892         UPPER(
893             LEFT(@in_string, 1)
894         ),
895         LOWER(
896             SUBSTRING(@in_string, 2,
897                 LEN(@in_string) - 1)

```

```

898         )
899
900     or concatenation using the '+' sign (considering that adding a value
901     to NULL returns NULL, unless using the proper 'CASE' clause)
902
903         UPPER(
904             LEFT(@in_string, 1)
905         ) +
906         LOWER(
907             SUBSTRING(@in_string, 2,
908                 LEN(@in_string) - 1)
909         )
910
911     to parameter '@out_string'.
912
913     As the last step, we must indicate what value must be returned from
914     the function -- in this case, parameter '@out_string'.
915     *****/
916
917 CREATE FUNCTION AP5.properUDF                -- 1. creating function
918     (@in_string VARCHAR(50))                -- 2. declaring input parameter
919     RETURNS VARCHAR(50)                     -- 3. indicating the same data
920     AS                                       -- type and size of output
921     BEGIN                                    -- parameter '@out_string'
922     DECLARE @out_string VARCHAR(50)         -- 4. beginning of executable
923     -- code
924     DECLARE @out_string VARCHAR(50)         -- 5. declaring output
925     -- parameter '@out_string'
926     -- with same data type as
927     -- input parameter
928     -- '@in_string'; same data
929     -- type and size as
930     -- indicated after 'RETURNS'
931     SET @out_string = CONCAT (              -- 6. setting value to output
932         UPPER(LEFT(@in_string, 1)),         -- parameter '@out_string'
933         -- LOWER(SUBSTRING(@in_string, 2, LEN(@in_string) - 1))
934         LOWER(RIGHT(@in_string, LEN(@in_string) - 1))
935     )
936     RETURN @out_string;                    -- 7. returning value of output
937     -- parameter '@out_string'
938 END;                                        -- 8. end of executable code
939
940
941
942
943
944
945
946 /* *****/
947     8.02. In order to call our user-defined function ('UDF'), we must indicate
948     the schema where it resides -- in this case, 'AP5'.
949     *****/
950
951 SELECT DISTINCT AP5.properUDF('hello');
952
953
954 /* *****/
955     8.03. We can use 'AP5.properUDF' on any string value in any table, schema
956     or database as long as we have access to the data objects -- for
957     example, columns 'AP2.Customers.FirstName' and
958     'AP2.Customers.LastName'. Of course, first we insert values into
959     'AP2.Customers'.
960     *****/
961
962 INSERT INTO AP2.Customers                    -- all values in order
963 VALUES (
964     1,
965     'Smith',
966     'John',

```

```

967     '',
968     '',
969     '',
970     '',
971     ''
972 ),
973 (
974     2,
975     'Doe',
976     'Jane',
977     '123 Main St. Apt. 1',
978     'New York',
979     'NY',
980     '10001',
981     'jane.doe@foobar.foo'
982 );
983
984 INSERT INTO AP2.Customers (
985     CustomerID,
986     LastName,
987     FirstName
988 )
989 VALUES (
990     3,
991     'Smith',
992     'Tom'
993 );
994
995 INSERT INTO AP2.Customers
996 VALUES (
997     5,
998     'Doe',
999     'John',
1000     '',
1001     'New York',
1002     'NY',
1003     '10001',
1004     'john.doe@foobar.foo'
1005 ),
1006 (
1007     4,
1008     'Doe',
1009     'Jane',
1010     '',
1011     'New York',
1012     'NY',
1013     '',
1014     'jane.doe2@foobar.foo'
1015 );
1016
1017 UPDATE AP2.Customers
1018 SET FirstName = AP5.properUDF(FirstName),
1019     LastName = AP5.properUDF(LastName);
1020
1021
1022 /* *****
1023     8.04. We can also create functions to FORMAT dollar amounts
1024         (`AP5.dollarUDF`) and dates (`AP5.dateUDF`) considering that numeric
1025         values become strings when formatted.
1026     ***** */
1027
1028 CREATE FUNCTION AP5.dollarUDF (@in_dollar FLOAT)
1029 RETURNS VARCHAR(50)
1030 AS
1031 BEGIN
1032     DECLARE @out_dollar VARCHAR(50)
1033     SET @out_dollar = FORMAT(@in_dollar, 'c', 'en-us')
1034     RETURN @out_dollar
1035 END;

```



```
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=1415
1113  ***** */
```