Db2 and Jupyter Notebooks

George Baklarz
Db2 Digital Technical Engagement

Legal Disclaimer

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED. IN ADDITION, THIS INFORMATION IS BASED ON CURRENT THINKING REGARDING TRENDS AND DIRECTIONS, WHICH ARE SUBJECT TO CHANGE BY IBM WITHOUT NOTICE. FUNCTION DESCRIBED HEREIN MY NEVER BE DELIVERED BY IBM. IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION. NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, NOR SHALL HAVE THE EFFECT OF, CREATING ANY WARRANTIES OR REPRESENTATIONS FROM IBM (OR ITS SUPPLIERS OR LICENSORS), OR ALTERING THE TERMS AND CONDITIONS OF ANY AGREEMENT OR LICENSE GOVERNING THE USE OF IBM PRODUCTS AND/OR SOFTWARE.

Agenda

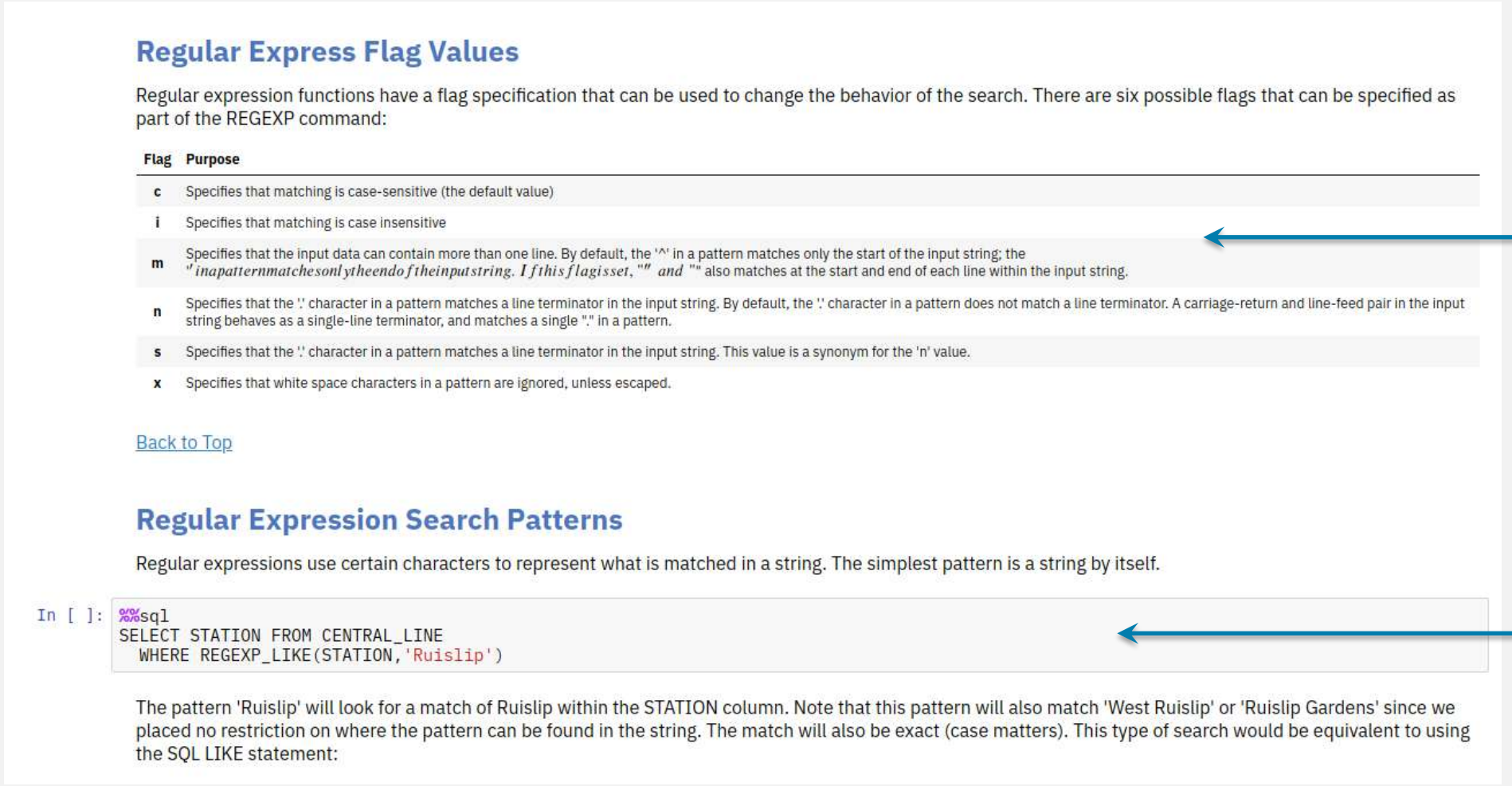
- What are Jupyter Notebooks?
- So Why do I care?
- Integrating Db2 into Notebooks
  - Up and Running
  - SQL Support
  - Advanced Features
- Resources to help get you started

Jupyter Notebooks

- Jupyter notebooks are based on IPython which started in development in the 2006/7 timeframe
- The existing Python interpreter was limited in functionality and work was started to create a richer development environment
- By 2011 the development efforts resulted in IPython being released (http://blog.fperez.org/2012/01/ipython-notebook-historical.html)
- Markdown is used to create the "text" of the notebook
- Code lines can contain:
  - Python
  - Magic commands (more on this later)
  - Java, C, GoLang, and a variety of other languages
- Allows interactive development and prototyping

What do Jupyter Notebooks Look Like?

- Looks a lot like Microsoft Word, but with live code



Regular Express Flag Values

Regular expression functions have a flag specification that can be used to change the behavior of the search. There are six possible flags that can be specified as part of the REGEXP command:

Flag	Purpose
c	Specifies that matching is case-sensitive (the default value)
i	Specifies that matching is case insensitive
m	Specifies that the input data can contain more than one line. By default, the '^' in a pattern matches only the start of the input string; the "*inapatternmatchesonlytheendoftheinputstring. Ifthisflagisset,*"" *and* "" also matches at the start and end of each line within the input string.
n	Specifies that the '.' character in a pattern matches a line terminator in the input string. By default, the '.' character in a pattern does not match a line terminator. A carriage-return and line-feed pair in the input string behaves as a single-line terminator, and matches a single "." in a pattern.
s	Specifies that the '.' character in a pattern matches a line terminator in the input string. This value is a synonym for the 'n' value.
x	Specifies that white space characters in a pattern are ignored, unless escaped.

Back to Top

Regular Expression Search Patterns

Regular expressions use certain characters to represent what is matched in a string. The simplest pattern is a string by itself.

```
In [ ]: %%sql
SELECT STATION FROM CENTRAL_LINE
  WHERE REGEXP_LIKE(STATION,'Ruislip')
```

The pattern 'Ruislip' will look for a match of Ruislip within the STATION column. Note that this pattern will also match 'West Ruislip' or 'Ruislip Gardens' since we placed no restriction on where the pattern can be found in the string. The match will also be exact (case matters). This type of search would be equivalent to using the SQL LIKE statement:

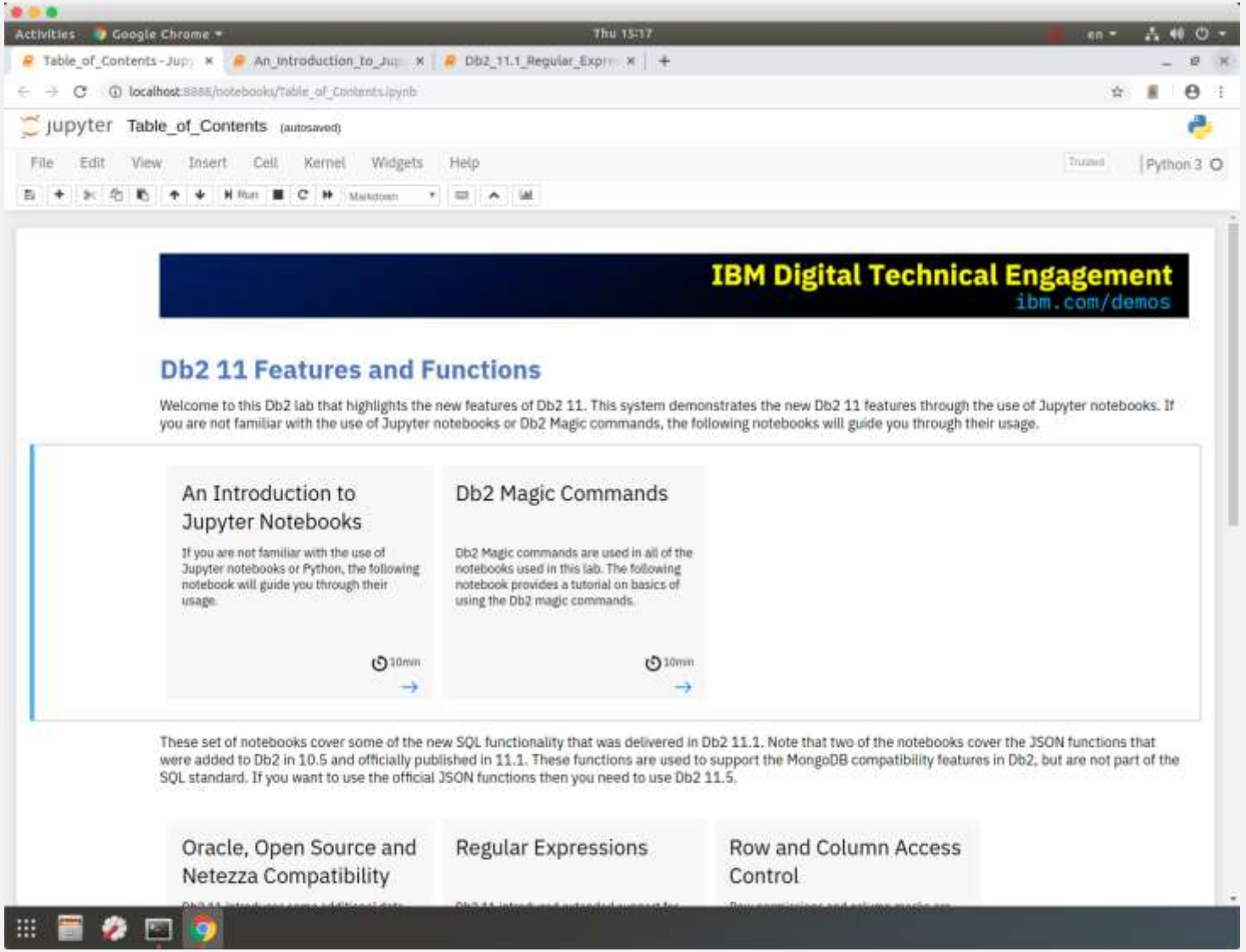
Text

Code

Jupyter Notebook Requirement

- Runs on all popular desktop platforms
  - Linux
  - Mac OSX
  - Windows
- Content is viewed via a web browser
  - Chrome, IE, Safari, Firefox, etc…
- Jupyter can run locally on a workstation, or as part of a service
  - Access Notebooks at **`localhost:8888`**
- Installation images can be found at:
  - https://jupyter.org/
  - https://www.anaconda.com/distribution/
  - Recommend that you get the full Anaconda stack which includes Python and many of the libraries that you will need

Example of Complex Jupyter Notebook



Why Do I Care About Jupyter Notebooks?

- Pros:
  - Used extensively by researchers and data scientists
  - Allows for sharing of text and code
  - Encourages trial and error, and the ability to document what is happening with the code

- Cons:
  - Need to understand Python (sometimes) to get things done
  - No native Db2 connections in notebooks (there are packages that make things easier, but they don't support Db2 extensions)

Getting Started

- **Install Anaconda or Jupyter Notebooks**
  - Consider using Docker if you don't want to install natively
- **Install the Db2 Python driver**
  - **easy_install ibm_db**
- **Download the Db2 Magic Command**
  - https://github.com/DB2-Samples/db2jupyter
  - Only one file is needed: **db2.ipynb**
  - "**ipynb**" is a Jupyter notebook
  - Import the notebook into your Jupyter environment

That Sounds Like a Pain…

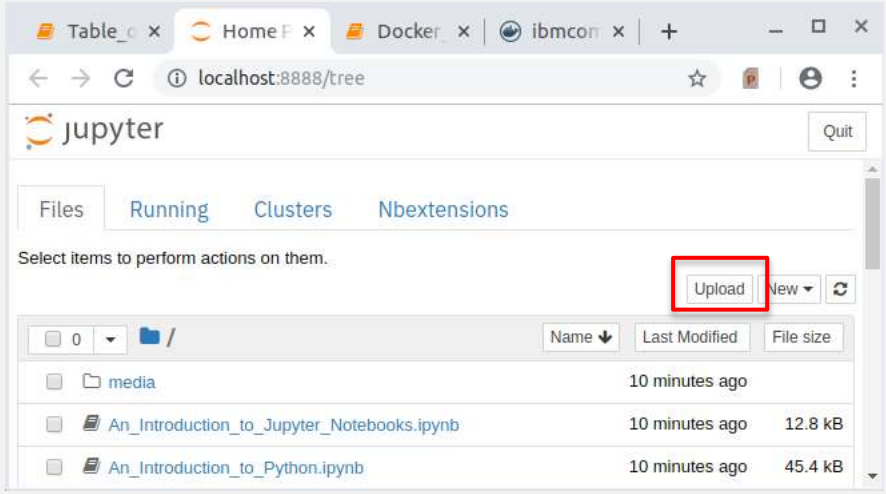
- **Install Docker**
  - See https://docs.docker.com/ for instructions
- **Create a Docker container**
  - https://github.com/DB2-Samples/db2jupyter
  - Download **db2jupyter.docker** into an **empty directory** of your choice
  - Use Kitematic CLI command line, or a terminal window on Mac or Linux
    - Navigate to the directory that the db2jupyter.docker file is located
    - Issue the following command to create a Docker image:

      docker build -t db2jupyter -f db2jupyter.docker . (←Keep the period!)

      docker run --name db2jupyter -d -p 8888:8888 db2jupyter
  - Use your favorite browser to navigate to **localhost:8888**

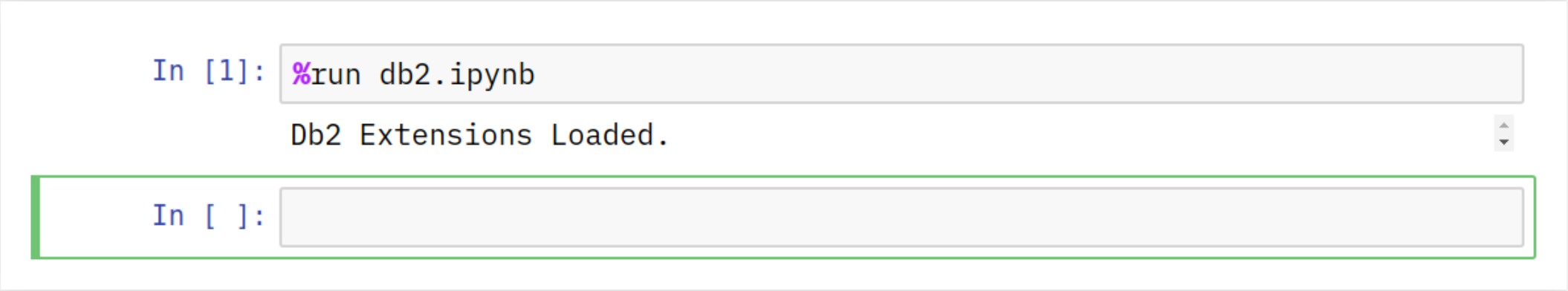
Now What?

- Start Jupyter Notebooks
- Make sure you have Db2 somewhere (or use a Docker install!)
- Navigate to your notebooks (localhost:8888)
- Import the db2.ipynb notebook, or place it in the path where your notebooks are going to be kept



Loading Db2 Magic Commands

- Open up a blank notebook and enter the following command



- Use the Jupyter "Run" button or Shift-Return to execute the code
- Some Magic Happens!

Magic Commands

- **Jupyter provides a series of Magic commands that allow for Python code or special actions to be executed on your behalf**
  - For example, **%system** will issue a system command
- **The %run db2.ipynb command will load the contents of the db2.ipynb file and create a new %sql and %%sql command**
- **The %sql command is used for single line commands while %%sql is meant to run a block of SQL**
  - Only the results of a **%sql** command can be assigned to a variable
  - Python variable subtitution using **{varname}** only works with **%sql** commands
  - Single **%sql** commands can be extended over multiple lines using the backslash \ character at the end of line
  - Multiple statements in a **%%sql** block use a semi-colon as a delimiter

Connecting to Db2

- Once you have loaded the Db2 magic commands, you are read to get connected
- The connection syntax is similar to Db2 CLP with a few twists

```
%sql CONNECT TO database
        USER DB2INST1 USING db2inst1
        HOST 172.5.2.132 PORT 50000 SSL
%sql CONNECT CREDENTIALS name
%sql CONNECT PROMPT
%sql CONNECT RESET
%sql CONNECT CLOSE
```

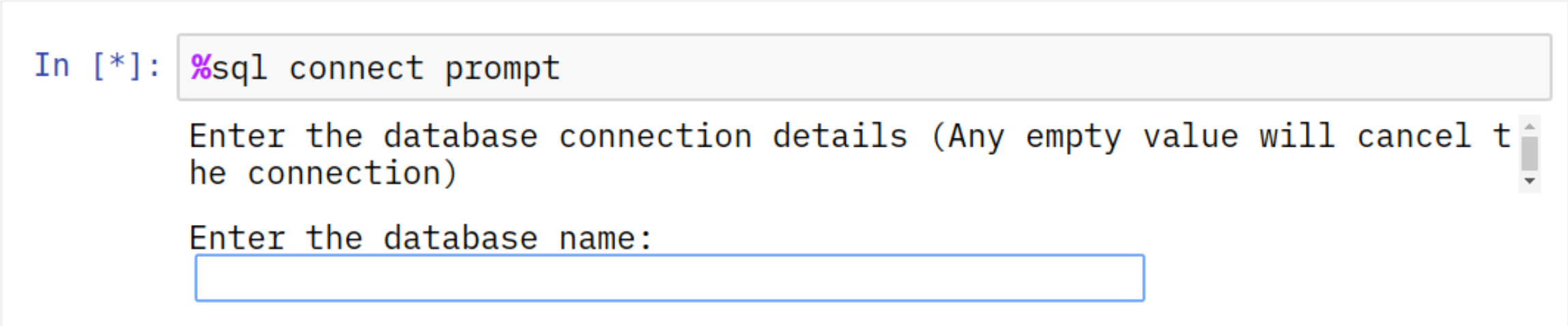
Connecting to Db2

- Once you have loaded the Db2 magic commands, you are read to get connected
- The connection syntax is similar to Db2 CLP with a few twists

```
%sql CONNECT TO database
        USER DB2INST1 USING db2inst1
        HOST 172.5.2.132 PORT 50000 SSL
%sql CONNECT CREDENTIALS name
%sql CONNECT PROMPT
%sql CONNECT
```

Connecting With a Prompt

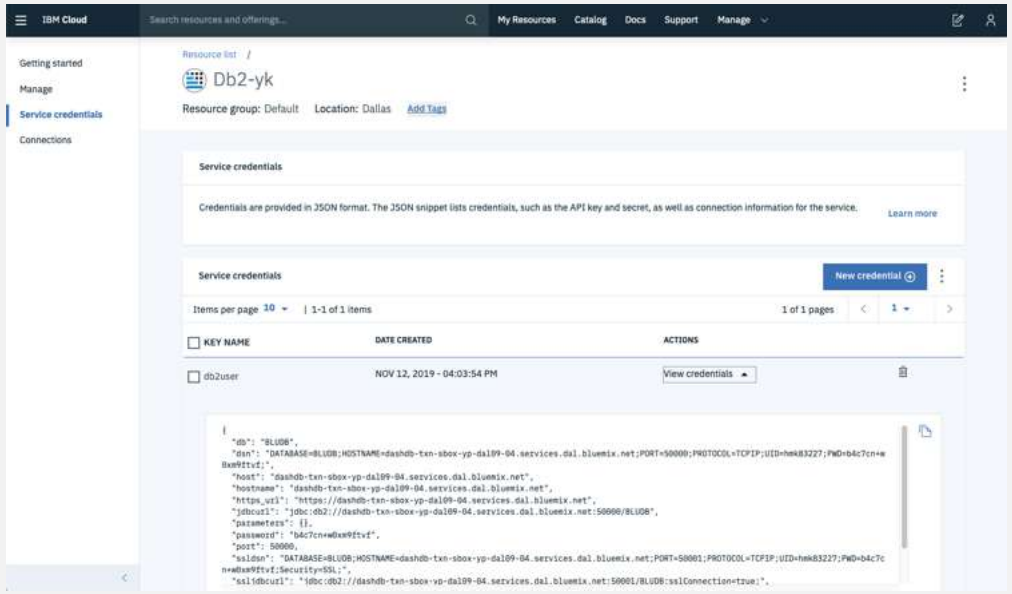
- **Connecting with PROMPT will ask you all the questions**



- **To stop at any time enter a null value**

Connecting With Credentials

- **Credentials come from a Cloud instance of Db2**



- **Assign the credentials to a Python variable**

```
myid = {"db": "BLUDB", "password":"iforgotit", …}
%sql CONNECT CREDENTIALS myid
```

Connecting With Values

- **Supply all of the values needed to connect to the database**

```
%sql CONNECT TO database USER DB2INST1 USING db2inst1
     HOST 172.5.2.132 PORT 50000 [SSL]
```

- **If you connected previously, the %sql CONNECT by itself will use the connection information that is saved on disk**
- **If any values are missing, they are supplied from a previous connection**
  - **%sql connect to sample**
  - Any previous user/using values will be supplied on your behalf
- **Implicit connects are done for you when you execute the first %sql statement**

Bad Connection

- **You will get full diagnostic information back from the connection**
  - You can decide if the information is useful!

```
In [3]: %sql connect to sample user db2inst using db2inst1
```

```
[IBM][CLI Driver] SQL30081N A communication error has been detected.
Communication protocol being used: "TCP/IP". Communication API being used:
"SOCKETS". Location where the error was detected: "1.1.1.1". Communication
function detecting the error: "connect". Protocol specific error code(s): "110",
"*", "*". SQLSTATE=08001 SQLCODE=-30081
```

Diagnostic Information

- **You can always get the diagnostic information from three built-in variables**
  - **sqlcode** – Last sqlcode generated
  - **sqlstate** – Last sqlstate generated
  - **sqlerror** – Last message generated
- **Useful if you are using programming logic**

```
In [111]: %sql SELECT NOTHING FROM UNKNOWN

          SQL0204N "DB2INST1.UNKNOWN" is an undefined name. SQLSTATE=42704
          SQLCODE=-204

          Command completed.

In [112]: sqlcode

Out[112]: -204

In [113]: sqlstate

Out[113]: '42704'

In [115]: sqlerror

Out[115]: 'SQL0204N  "DB2INST1.UNKNOWN" is an undefined name.  SQLSTATE=42704 SQ
          LCODE=-204'
```

Successful Connection

- **A successful connection will also return a message**

```
In [5]: %sql connect to sample host localhost port 50000 \
            user DB2INST1 using db2inst1

        Connection successful.
```

- **Note that %sql commands can only span one line**
  - To continue over one line, you must use the backslash \ character
  - Userids and Passwords are case sensitive
- **You can now connect in any notebook just using connect**

```
In [6]: %sql connect

        Connection successful.
```

Now What Can I Do?

- **Once you have connected, you can run SQL!**

```
In [7]: %sql select * from employee
```

Out[7]:

	EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
0	000010	CHRISTINE	I	HAAS	A00	3978	1995-01-01	PRES	18	F	1963-08-24	152750.0	1000.0	4220.0
1	000020	MICHAEL	L	THOMPSON	B01	3476	2003-10-10	MANAGER	18	M	1978-02-02	94250.0	800.0	3300.0
2	000030	SALLY	A	KWAN	C01	4738	2005-04-05	MANAGER	20	F	1971-05-11	98250.0	800.0	3060.0
3	000050	JOHN	B	GEYER	E01	6789	1979-08-17	MANAGER	16	M	1955-09-15	80175.0	800.0	3214.0
4	000060	IRVING	F	STERN	D11	6423	2003-09-14	MANAGER	16	M	1975-07-07	72250.0	500.0	2580.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
37	200240	ROBERT	M	MONTEVERDE	D21	3780	2004-12-05	CLERK	17	M	1984-03-31	37760.0	600.0	2301.0
38	200280	EILEEN	R	SCHWARTZ	E11	8997	1997-03-24	OPERATOR	17	F	1966-03-28	46250.0	500.0	2100.0
39	200310	MICHELLE	F	SPRINGER	E11	3332	1994-09-12	OPERATOR	12	F	1961-04-21	35900.0	300.0	1272.0
40	200330	HELENA		WONG	E21	2103	2006-02-23	FIELDREP	14	F	1971-07-18	35370.0	500.0	2030.0
41	200340	ROY	R	ALONZO	E21	5698	1997-07-05	FIELDREP	16	M	1956-05-17	31840.0	500.0	1907.0

42 rows × 14 columns

The Formatting is Weird

- **The default output is in Pandas Dataframes format**

```
In [7]: %sql select * from employee
```

Out[7]:

	EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO	HIREDATE	JOB	EDLEVEL	SEX	BIRTHDATE	SALARY	BONUS	COMM
0	000010	CHRISTINE	I	HAAS	A00	3978	1995-01-01	PRES	18	F	1963-08-24	152750.0	1000.0	4220.0
1	000020	MICHAEL	L	THOMPSON	B01	3476	2003-10-10	MANAGER	18	M	1978-02-02	94250.0	800.0	3300.0
2	000030	SALLY	A	KWAN	C01	4738	2005-04-05	MANAGER	20	F	1971-05-11	98250.0	800.0	3060.0
3	000050	JOHN	B	GEYER	E01	6789	1979-08-17	MANAGER	16	M	1955-09-15	80175.0	800.0	3214.0
4	000060	IRVING	F	STERN	D11	6423	2003-09-14	MANAGER	16	M	1975-07-07	72250.0	500.0	2580.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
37	200240	ROBERT	M	MONTEVERDE	D21	3780	2004-12-05	CLERK	17	M	1984-03-31	37760.0	600.0	2301.0
38	200280	EILEEN	R	SCHWARTZ	E11	8997	1997-03-24	OPERATOR	17	F	1966-03-28	46250.0	500.0	2100.0
39	200310	MICHELLE	F	SPRINGER	E11	3332	1994-09-12	OPERATOR	12	F	1961-04-21	35900.0	300.0	1272.0
40	200330	HELENA		WONG	E21	2103	2006-02-23	FIELDREP	14	F	1971-07-18	35370.0	500.0	2030.0
41	200340	ROY	R	ALONZO	E21	5698	1997-07-05	FIELDREP	16	M	1956-05-17	31840.0	500.0	1907.0

42 rows × 14 columns

- **It shows the first 5 and last 5 lines of output**
- **If you want to see more, you need to either use a special flag (-all), change some settings, or use –grid format**

Grid Format is Nicer

- Use the –grid option after the %sql command

```
In [8]: %sql -grid select * from employee
```

	EMPNO	FIRSTNME	MIDINIT	LASTNAME	WORKDEPT	PHONENO
0	000010	CHRISTINE	I	HAAS	A00	3978
1	000020	MICHAEL	L	THOMPSON	B01	3476
2	000030	SALLY	A	KWAN	C01	4738
3	000050	JOHN	B	GEYER	E01	6789
4	000060	IRVING	F	STERN	D11	6423

- Now you can scroll the answer set, change the column sizes, and sort by any column
- You can set this as the default viewing format by changing an OPTION

Options

- **There are some options available for changing the behavior of the commands:**

```
In [11]: %sql option list

(MAXROWS) Maximum number of rows displayed: 10
(MAXGRID) Maximum grid display size: 5
(RUNTIME) How many seconds to a run a statement for performance testing: 1
(DISPLAY) Use PANDAS or GRID display format for output: PANDAS
```

  - **MAXROWS** – The number of rows that Pandas display will show before hiding part of the answer set
  - **MAXGRID** – The maximum number of lines shown in a scrollable display
  - **RUNTIME** – When timing statement execution, how many seconds to run
  - **DISPLAY** – Use Pandas or Grid format for output
  - **LIST** – Display current settings
- **The command allows multiple options to be set at a time**

  %sql option maxgrid 10 display grid

SQL Statements versus SQL Blocks

- So far we have only shown single SQL statements
- What if you want to do a large block of SQL?
- Use the %%sql format to tell the program that everything in the Cell is Db2

```
In [12]: %%sql
DROP TABLE NIGHTSHOW;
CREATE TABLE NIGHTSHOW (
   ANNOUNCER VARCHAR(20) NOT NULL,
   SPEAKER   VARCHAR(20),
   TITLE     VARCHAR(50)
);
INSERT INTO NIGHTSHOW VALUES
  ('Martin','George','Jupyter Stuff');
```

```
SQL0204N "DB2INST1.NIGHTSHOW" is an undefined name. SQLSTATE=42704 SQLCODE=-204
```

```
Command completed.
```

Delimiters

- **The default is a semi-colon (;) to delimit SQL statements**
- **Use the –d flag to change it to the at (@) sign**
- **Errors are ignored during the execution of a block**
  - If you want to suppress the errors, use the **–quiet** or **–q** flag
  - Output is still displayed

```
In [15]: %%sql -q -d
         DROP TABLE NIGHTSHOW@
         CREATE TABLE NIGHTSHOW (
            ANNOUNCER VARCHAR(20) NOT NULL,
            SPEAKER   VARCHAR(20),
            TITLE     VARCHAR(50)
         )@
         INSERT INTO NIGHTSHOW VALUES
           ('Martin','George','Jupyter Stuff')@
         SELECT * FROM NIGHTSHOW@
```

Out[15]:

	ANNOUNCER	SPEAKER	TITLE
0	Martin	George	Jupyter Stuff

SELECT Into a Variable

- **You can assign the result set to a Python variable**

```
In [16]: show = %sql SELECT * FROM NIGHTSHOW
```

- **Once it is in a variable you can display it, or slice it using Pandas syntax**
  - Useful if you know the name of the column but not the position

```
In [20]: show
Out[20]:
```

	ANNOUNCER	SPEAKER	TITLE
0	Martin	George	Jupyter Stuff

```
In [19]: show['ANNOUNCER'][0]
Out[19]: 'Martin'
```

SELECT Into a Variable

- You can also ask for the data in "raw" array format

```
In [21]: show = %sql -r SELECT * FROM NIGHTSHOW

In [23]: show

Out[23]: [['ANNOUNCER', 'SPEAKER', 'TITLE'], ['Martin', 'George', 'Jupyter Stuf
         f']]
```

- The first row of the array (row 0) is the column name and the remainder are the data rows

```
In [24]: show[1][0]

Out[24]: 'Martin'
```

Using Parameters

- **You can supply Python variables to the statements by using the format :var_name**

```
In [26]: speaker = "George"

In [27]: %sql SELECT * FROM NIGHTSHOW WHERE SPEAKER = :speaker
```

Out[27]:

	ANNOUNCER	SPEAKER	TITLE
0	Martin	George	Jupyter Stuff

- **It also works with IN lists!**

```
In [33]: empnos = ['000010','000020','000030']

In [38]: %sql SELECT EMPNO, LASTNAME FROM EMPLOYEE WHERE EMPNO IN (:empnos)
```

Out[38]:

	EMPNO	LASTNAME
0	000010	HAAS
1	000020	THOMPSON
2	000030	KWAN

String Substitution

- Use the {varname} format to supply values for SQL syntax items (columns, etc…)

```
In [55]: empnos = ['000010','000020','000030']
         columns = 'EMPNO,LASTNAME'

In [56]: %sql SELECT {columns} FROM EMPLOYEE WHERE EMPNO IN (:empnos)
```

Out[56]:

	EMPNO	LASTNAME
0	000010	HAAS
1	000020	THOMPSON
2	000030	KWAN

- Can only be used with single row %sql commands

INSERT

- INSERTs need some care when dealing with strings
- You can use typical INSERT with VALUEs clause

```
In [62]: %sql INSERT INTO SOMEDATA VALUES 'Hello'
```

- Quote inside strings is tricky – two quotes = one quote

```
In [63]: %sql INSERT INTO SOMEDATA VALUES 'Hello''s'
```

- Or just use variables! (No Quotes required!)

```
In [64]: hello = "Hellos's"
         %sql INSERT INTO SOMEDATA VALUES :hello
```

JSON INSERTs

- **Create a JSON Object (Dictionary) and INSERT directly**

In [73]:
```
%%sql
DROP TABLE SOMEDATA;
CREATE TABLE SOMEDATA (
  JSON VARCHAR(255)
);
```

```
Command completed.
```

In [74]:
```
x = {
    "first" : "Martin",
    "last"  : "Hubel"
}
```

In [75]:
```
%%sql
INSERT INTO SOMEDATA VALUES :x;
SELECT * FROM SOMEDATA;
```

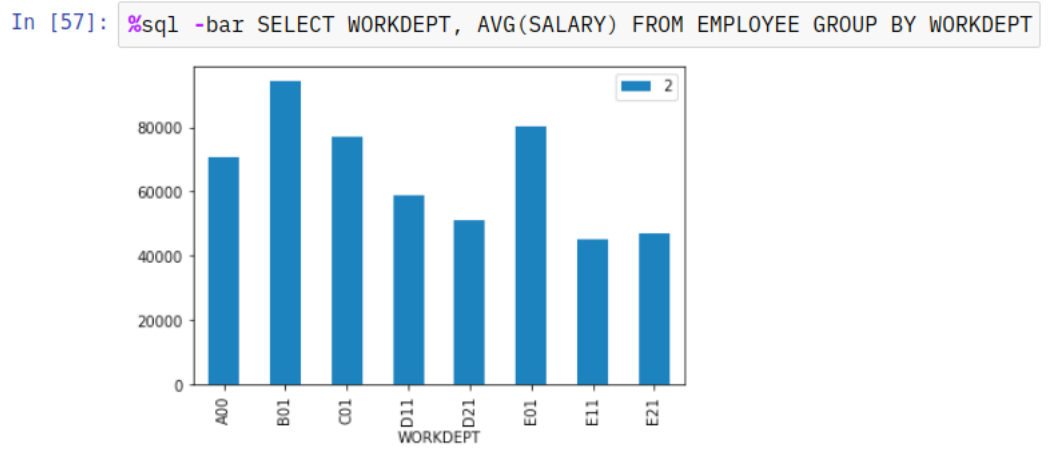
Out[75]:

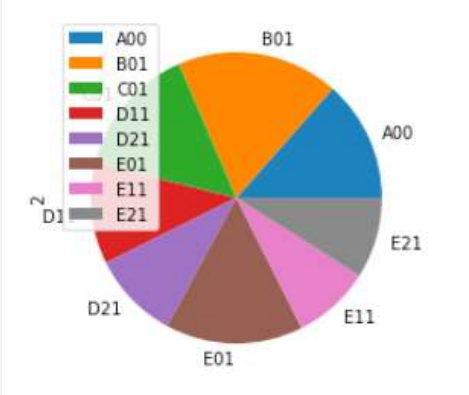
	JSON
0	{"first": "Martin", "last": "Hubel"}

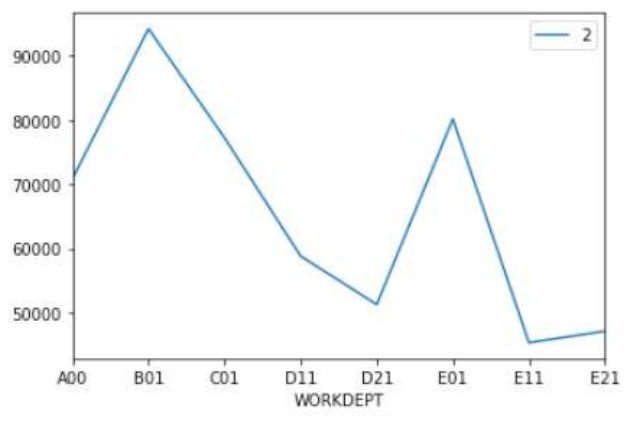
Simple Plotting

- **The %sql command allows for simple charting:**
  - **-pie** – Pie chart
  - **-bar** – Bar chart
  - **-line** – Line chart

```
%sql –bar SELECT WORKDEPT, AVG(SALARY) FROM EMPLOYEE \
      GROUP BY WORKDEPT
```







Commit Scope

- **The default mode of SQL execution is AUTOCOMMIT ON**
  - Every statement is committed
- **You can turn AUTOCOMMIT on and off**

  %sql AUTOCOMMIT OFF
  - The **AUTOCOMMIT** state remains in this state for the lifetime of the connection
- **Use the COMMIT statement at the end of the transaction**
  - **COMMIT WORK** – Closes the statement and commits all work
  - **COMMIT HOLD** – Keeps the statement open for further work
  - **ROLLBACK** – Rolls back all work done to this point

Stored Procedures

- **You can call stored procedures using the following syntax**

  result, p_a, p_b, p_c, … = %sql –r CALL proc(a,b,c,….)

- **This syntax (with the –r flag) will return the results, and the parameters**
  - p_a, p_b, p_c are the values supplied or returned by the procedure
- **If you only want the result set, use the statement by itself or assign it to a variable**

  x = %sql CALL ADMIN_CMD('DESCRIBE TABLE EMPLOYEE')

```
In [110]: %sql CALL ADMIN_CMD('DESCRIBE TABLE EMPLOYEE')
Out[110]:
```

	COLNAME	TYPESCHEMA	TYPENAME	LENGTH	SCALE	NULLABLE
0	EMPNO	SYSIBM	CHARACTER	6	0	N
1	FIRSTNME	SYSIBM	VARCHAR	12	0	N
2	MIDINIT	SYSIBM	CHARACTER	1	0	Y
3	LASTNAME	SYSIBM	VARCHAR	15	0	N
4	WORKDEPT	SYSIBM	CHARACTER	3	0	Y
…	…	…	…	…	…	…
9	SEX	SYSIBM	CHARACTER	1	0	Y
10	BIRTHDATE	SYSIBM	DATE	4	0	Y
11	SALARY	SYSIBM	DECIMAL	9	2	Y
12	BONUS	SYSIBM	DECIMAL	9	2	Y
13	COMM	SYSIBM	DECIMAL	9	2	Y

14 rows × 6 columns

Prepared Statements

- The PREPARE and EXECUTE commands are useful in situations where you want to repeat an SQL statement multiple times
- There isn't any benefit from using these statements for simple tasks that may only run occasionally
- The benefit of PREPARE/EXECUTE is more evident when dealing with a large number of transactions that are the same

Prepared Statements

- The PREPARE statement can only be used for the following SQL statements:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - MERGE
- To prepare a statement, you must use the following syntax:

```
stmt = %sql PREPARE sql ....
```

- The stmt variable is used when executing the statement

Prepared Statements

- **Once you have prepared a statement, you can execute it using the following syntax:**

```
%sql EXECUTE :stmt USING v1,v2,v3,....
```

- **You must provide the statement variable :stmt to the EXECUTE statement**
- **The values that following the USING clause are either constants or Python variable names separated by commas**
  - If you place a colon : in front of a variable name, it will be immediately substituted into the statement:

```
%sql EXECUTE :stmt USING 3,'asdsa',24.5,:x,y
```

  - The **:x** will be materialized directly into the parameter list, while the variable **y** will be bound to the statement and the contents dynamically transferred

Prepared Statements

- **With Python variables, you can specify four types of data:**
  - **char** - character data type (default)
  - **bin**, **binary** - binary data
  - **dec**, **decimal** - decimal data type
  - **int**, **integer** - numeric data type
- **These modifiers are added after the variable name by using the @ symbol:**

```
%sql EXECUTE :stmt USING v1@int, v2@binary, v3@decimal
```

- **The default is to treat variables as character strings**

Prepared Example

In [124]:
```
%%sql -q
DROP TABLE INTEGERS;
CREATE TABLE INTEGERS (
  ANUMBER INT
);
```

In [125]:
```
i = 1
stmt = %sql prepare INSERT INTO INTEGERS VALUES ?
while i <= 10:
    %sql execute :stmt using i@integer
    i = i + 1
%sql SELECT * FROM INTEGERS
```

Out[125]:

	ANUMBER
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10

More Statement Options

- **Additional Options are Available when Running SQL**
  - **-d** - Use alternative statement delimiter @
  - **-t**,**-time** - Time the statement execution
  - **-q**,**-quiet** - Suppress messages
  - **-j** - JSON formatting of the first column
  - **-json** - Retrieve the result set as a JSON record
  - **-a**,**-all** - Show all output
  - **-pb**,**-bar** - Bar chart of results
  - **-pp**,**-pie** - Pie chart of results
  - **-pl**,**-line** - Line chart of results
  - **-sampledata** Load the database with the sample EMPLOYEE and DEPARTMENT tables
  - **-r**,**-array** - Return the results into a variable (list of rows)
  - **-e**,**-echo** - Echo macro substitution
  - **-h**,**-help** - Display help information
  - **-grid** - Display results in a scrollable grid

Example: Retrieve Data as JSON Record

```
In [127]: %sql -json SELECT * FROM EMPLOYEE FETCH FIRST ROW ONLY

Out[127]: [{'empno': '000010',
            'firstnme': 'CHRISTINE',
            'midinit': 'I',
            'lastname': 'HAAS',
            'workdept': 'A00',
            'phoneno': '3978',
            'hiredate': '1995-01-01',
            'job': 'PRES    ',
            'edlevel': 18,
            'sex': 'F',
            'birthdate': '1963-08-24',
            'salary': 152750.0,
            'bonus': 1000.0,
            'comm': 4220.0}]
```

Macros

- **The %sql command also allows the use of macros**
  - Macros are used to substitute text into SQL commands that you execute
  - Macros substitution is done before any SQL is executed
  - This allows you to create macros that include commonly used SQL commands rather than having to type them in
  - Macros can access parameters and have limited logic capability

```
%%sql define describe
#
# The DESCRIBE command can either use the syntax DESCRIBE TABLE <name> or DESCRIBE TABLE SELECT
#
var syntax Syntax: DESCRIBE [TABLE name | SELECT statement]
#
# Check to see what count of variables is... Must be at least 2 items
#
if {argc} < 2
   exit {syntax}
endif
CALL ADMIN_CMD('{*0}');
```

Predefined Macros

- **DESCRIBE [SELECT or TABLE]**
  - Describe the contents of a table or a select statement
- **LIST TABLES [FOR ALL | FOR SCHEMA name]**
  - List tables in the current schema or database

```
In [130]: %sql LIST TABLES FOR SCHEMA DB2INST1
```

Out[130]:

	TABNAME	TABSCHEMA	DESCRIPTION
0	ACT	DB2INST1	Table
1	ADEFUSR	DB2INST1	Materialized query table
2	AS_EMP	DB2INST1	Table
3	BASE_EMP_TXS	DB2INST1	Table
4	CENTRAL_LINE	DB2INST1	Table
...	...	...	...
63	VPSTRDE1	DB2INST1	View
64	VPSTRDE2	DB2INST1	View
65	VSTAFAC1	DB2INST1	View
66	VSTAFAC2	DB2INST1	View
67	XYCOORDS	DB2INST1	Table

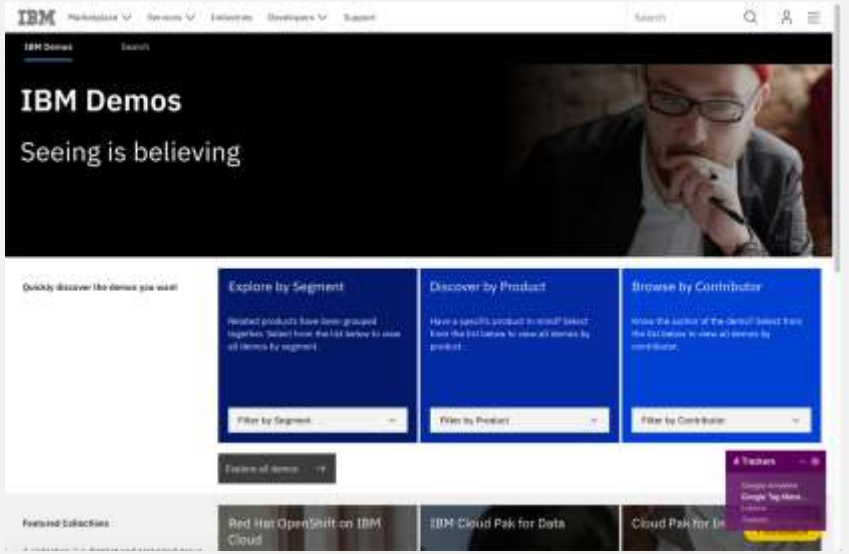
68 rows × 3 columns

Live Labs

- **Visit ibm.com/demos and search for Db2**
- **New Live Labs being made available shortly:**
  - Db2 Magic Commands and Programming in Python
  - Db2 11.1 SQL Features and Functions
  - Db2 11.5 SQL Features and Functions
  - An Introduction to External Tables
  - Using the New JSON ISO Functions in Db2
  - The Db2 Data Management Console and RESTful APIs
  - Db2 on Cloud Tutorial
  - Using Visual Studio Code and the GO Language
- **Steps:**
  - 1. Sign up for an IBM userid
  - 2. Select your lab
  - 3. Wait for the email that the lab is ready (2-3 minutes)
  - 4. Explore!

Additional Resources

- Visit the Digital Technical Engagement Site
  - The Digital Technical Engagement group (DTE) provides videos, product tours, and product labs for you to try out technology at your leisure
  - The product labs are fully functional servers that are provisioned for you
  - These servers contain the base products (Db2) along with self-paced examples
  - https://www.ibm.com/demos
- Read the new Db2 JSON Book
  - ibm.biz/db2json
- GitHub Db2-Samples
  - There are a number of Db2 sample programs available on GitHub
  - https://github.com/IBM/db2-samples



Db2 and Jupyter Notebooks

George Baklarz
Db2 Digital Technical Engagement