Python - Accesso ai dati

Novembre 2019 http://bit.ly/accessoaidati

Rev 4.26

- PostgreSQL (**psycopg2**, txpostgres, ...)
- MySQL (mysql-python, PyMySQL, ...)
- MS SQL Server (adodbapi, pymssql, mxODBC, **pyodbc**, ...)
- Oracle (**cx_Oracle**, mxODBC, pyodbc, ...)
- DB2 (ibm_db)
- sqlite3

- A Constructor
  - DB API provides a constructor connect, which returns a Connection object:
    - connect(parameters)
  - This can be considered the entry point for the module. Once you've got a connection, everything else flows from there.
- The parameters required and accepted by the connect constructor will vary from implementation to implementation, since they are highly specific to the underlying database.

A Connection

- Some of the methods of the connection may not be supported by all implementations:
- .close()
  - Closes the connection to the database permanently. Attempts to use the connection after calling this will raise a DB-API Error.
- .commit()
  - explicitly commit any pending transactions to the databse. The method should be a no-op if the underlying db does not support transactions.
- .rollback()
  - This optional method causes a transaction to be rolled back to the starting point. It may not be implemented everywhere.
- .cursor()
  - returns a Cursor object which uses this Connection.

A Cursor - settings

- You can use a few values to control the rows returned by the cursor:
- .arraysize
  - An integer which controls how many rows are returned at a time by .fetchmany (and optionally how many to send at a time with .executemany) Defaults to 1
- .setinputsizes(sizes)
  - Used to set aside memory regions for paramters passed to an operation
- .setoutputsize(size[, column])
  - Used to control buffer size for large columns returned by an operation (BLOB or LONG types, for example)
- The final two methods may be implemented as no-ops

A Cursor - operations

- The cursor should be used to run operations on the database:
- .execute(operation[, parameters])
  - Prepares and then runs a database operation. Parameter style (seq or mapping) and markers are implementation specific (see paramstyle)
- .executemany(operation[, seq_of_params])
  - Prepares and the runs an operations once for each set of parameters provided (this replaces the old v1 behavior of passing a seq to .execute).
- .callproc(procname[, parameters])
  - Calls a stored DB procedure with the provided parameters. Returns a modified version of the provided parameters with output and input/output parameters replaced

A Cursor - attributes

- These attributes of Cursor can help you learn about the results of operations:
- .rowcount
  - Tells how many rows have been returned or affected by the last operation. The number will be -1 if no operation has been performed.
- .description
  - Returns a sequence of 7-item sequences describing each of the columns in the result row(s) returned (None if no operation has been performed):
    - name
    - type_code
    - display_size (optional)
    - internal_size (optional)
    - precision (optional)
    - scale (optional)
    - null_ok (optional)

A Cursor - results

- The return value .execute or .executemany is undefined and should not be used. These methods are the way to get results after an operation:
- .fetchone()
  - Returns the next row from a result set, and None when none remain.
- .fetchmany([size=cursor.arraysize])
  - Returns a sequence of size rows (or fewer) from a result set. An empty sequence is returned when no rows remain. Defaults to arraysize.
- .fetchall()
  - Returns all (remaining) rows from a result set. This behavior may be affected by arraysize
  - Note that each of these methods will raise a DB API Error if no operation has been performed (or if no result set was produced)

Data Types & Constructors

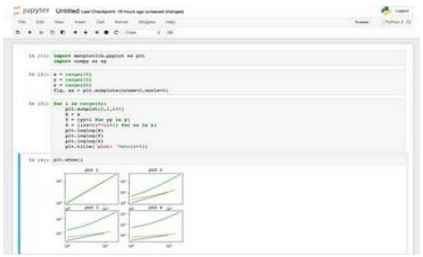
- The DB-API provides types and constructors for data:
  - Date(year, month, day) - constructs an object holding a date value
  - Time(hour, min, sec) - constructs an object holding a time value
  - Timestamp(y, m, d, h, min, s) - constructs an object holding a timestamp
    - Each of the above has a corresponding <name>FromTicks(ticks) which returns the same type given a single integer argument (seconds since the epoch)
  - Binary(string) - constructs an object to hold long binary string data
  - STRING - a type to describe columns that hold string values (CHAR)
  - BINARY - a type to describe long binary columns (BLOB, RAW)
  - NUMBER - a type to describe numeric columns
  - DATETIME - a type to describe date/time/datetime columns
  - ROWID - a type to describe the Row ID column in a database
  - SQL NULL values are represented by Python's None

Exceptions

- The DB API specification requires implementations to create the following hierarchy of custom Exception classes:
  - StandardError
    - Warning
    - Error
      - InterfaceError (a problem with the db api)
      - DatabaseError (a problem with the database)
        - DataError (bad data, values out of range, etc.)
        - OperationalError (the db has an issue out of our control)
        - IntegrityError
        - InternalError
        - ProgrammingError (something wrong with the operation)
        - NotSupportedError (the operation is not supported)

Esempi: DB2

vedi jupyter notebook
ibmdb2



Esempio PyODBC con sql server

- Database connectivity interface
  - A database connectivity interface allows an application to access data from a variety of DBMSs, using a specific driver for a specific DBMS and operating system.
  - The application can be written without depending on a specific DBMS or the operating system
- Open DataBase Connectivity (ODBC) is a standard **Microsoft Windows** interface that enables applications (typically written in C or C++) to connect to DBMSs.
  - also on linux
  - on windows replaced (really never replaced) by oledb and ado
- Java DataBase Connectivity (JDBC)
  - a standard Oracle interface that enables applications written in Java to connect to DBMSs

Pyodbc

- Step 1: install pyodbc through pip
  - pip install pyodbc
- Step 2: install the required driver for the DBMS-database you want to connect to. For example, if you want to connect to a Microsoft SQL Server-Database, you need to download and install the driver from Microsoft, after choosing your operating system
- Connecting to a database
- To make a connection to a database, we need to pass a connection string to the connect() function of pyodbc.
- pyodbc passes the connection string directly to the DBMS-database driver unmodified. **Therefore, connection strings are driver-specific**
- For example, to connect to a Microsoft SQL Server-Database, we provide the following connection string:
- connections strings examples

```
import pyodbc
cnxn = pyodbc.connect('driver={SQL Server};'
                      'server=serverName;'
                      'database=databaseName;'
                      'trusted_consnection=yes')
```

pyodbc

- L'esempio precedente faceva riferimento ad una singola stringa di connessione
  - in python stringhe giustapposte sono concatenate
- In alternativa è possibile passare i singoli argomenti che poi saranno concatenati in una singola stringa:
- e' possibile anche utilizzare un DSN che deve essere però preimpostato sul sistema

```
cnxn = pyodbc.connect('DSN=DsnName; user=qualcuno; pwd=password')
```

```
cnxn = pyodbc.connect(driver='{SQL Server}',
server='serverName',
database='databaseName',
uid='UserID',
pwd='password')
```

Pyodbc

- Installazione su Linux
- Installazione su Windows
  - Selezionare → Origini dati ODBC dallo start
- **Conversioni di formato**
- Riferimenti

Con Pyodbc

- import pyodbc
- import pandas as pd
- ...
- cnxn = pyodbc.connect('DRIVER={{Microsoft Access Driver (*.mdb, *.accdb)}};DBQ='
  +
- '{};Uid={};Pwd={};'.format(db_file, user, password)
-
- query = "SELECT * FROM mytable WHERE INST = '796116'"
- dataf = pd.read_sql(query, cnxn)
- cnxn.close()

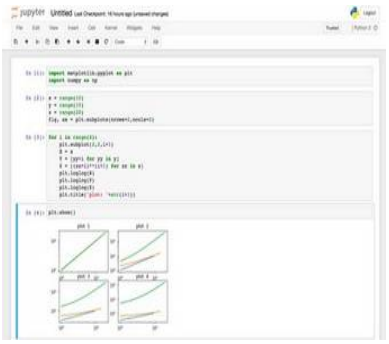
Da dizionario

```
         0   1
0  2012-07-02  392
1  2012-07-06  392
2  2012-06-29  391
3  2012-06-28  391
...
```

```
In [12]: pd.DataFrame(d.items(), columns=['Date',
'DateValue'])
Out[12]:
        Date  DateValue
0  2012-07-02        392
1  2012-07-06        392
2  2012-06-29        391
```

vedi jupyter notebook pyodbc

```
def get_response(q):
    d=[]
    try:
        cnx = mdb.connect(**DSN)
    except Exception as err:
        hexcept(err,exit=True)
    try:
        with cnx.cursor() as cursor:
            cursor.execute(q)
            desc = cursor.description
            column_names = [col[0] for col in desc]
            d = [dict(zip(column_names, row)) for row in cursor.fetchall()]
    except Exception as err:
        hexcept(err, exit=True)
    else:
        return d
```

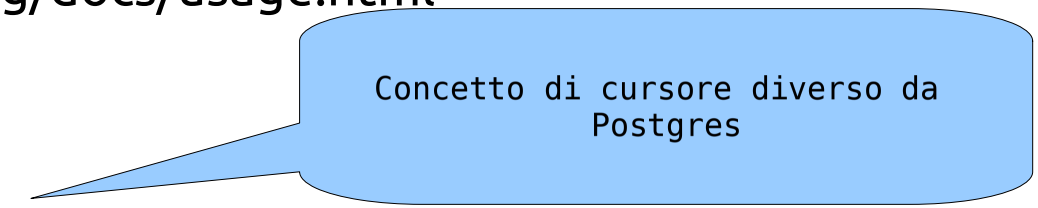


Esempi Postgres

- Via psycopg2

```
#!/usr/bin/python2.4
#
# Small script to show PostgreSQL and Pyscopg together
#
import psycopg2
try:
    conn = psycopg2.connect("dbname='template1' / user='dbuser' host='localhost'/
password='dbpass'")
except:
    print "I am unable to connect to the database"
```

- Documentation on http://initd.org/psycopg/docs/usage.html




```
cur = conn.cursor()

cur.execute("""SELECT datname from pg_database""")
rows = cur.fetchall()
```

Never, never, NEVER use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string. Not even at gunpoint.

Context manager

```
with psycopg2.connect(DSN) as conn:
    with conn.cursor() as curs:
        curs.execute(SQL)
```

- Il blocco interno non chiude la connessione ma la sessione (fa il commit)
- una connection può essere usata in più che un “with statement” ed ognun “with block” è di fatto wrappato in una singola transazione
- nei db bisogna stare attenti alla implementazione del context manager potrebbe comportarsi in maniera non attesa – leggere sempre la documentazione
- server side cursors

Lavorare con i file

`copy_from()`

Reads data from a file-like object appending them to a database table (COPY table FROM file syntax). The source file must have both read() and readline() method.

`copy_to()`

Writes the content of a table to a file-like object (COPY table TO file syntax). The target file must have a write() method.

`copy_expert()`

Allows to handle more specific cases and to use all the COPY features available in PostgreSQL.

Corrispondenza di tipi

Python	PostgreSQL	See also
None	NULL	Constants adaptation
bool	bool	
float	real double	Numbers adaptation
int long	smallint integer bigint	
Decimal	numeric	
str unicode	varchar text	Strings adaptation
buffer memoryview bytearray bytes Buffer protocol	bytea	Binary adaptation
date	date	Date/Time objects adaptation
time	time timetz	
datetime	timestamp timestamptz	
timedelta	interval	
list	ARRAY	Lists adaptation
tuple namedtuple	Composite types IN syntax	Tuples adaptation Composite types casting
dict	hstore	Hstore data type
Psycopg's Range	range	Range data types
Anything™	json	JSON adaptation
uuid	uuid	UUID data type

executemany

```
namedict = ({"first_name":"Joshua", "last_name":"Drake"},
            {"first_name":"Steven", "last_name":"Foo"},
            {"first_name":"David", "last_name":"Bar"})


cur = conn.cursor()
cur.executemany("""INSERT INTO bar(first_name,last_name)  VALUES (%(first_name)s, %(last_name)s)""", namedict)

cur.executemany('''insert into bar(first_name, last_name) values ({first_name}, {last_name})''', namedict)
```

- executemany cicla sul cursore per gli elementi namedict

Row via column name

```
#/usr/bin/python2.4
#
#
# load the adapter
import psycopg2
# load the psycopg extras module
import psycopg2.extras
# Try to connect
try:
    conn=psycopg2.connect("dbname='foo' user='dbuser' password='mypass'")
except:
    print "I am unable to connect to the database."
# If we are accessing the rows via column name instead of position we
# need to add the arguments to conn.cursor.
cur = conn.cursor(cursor_factory=psycopg2.extras.DictCursor)
try:
    cur.execute("""SELECT * from bar""")
except:
    print "I can't SELECT from bar"
#
# Note that below we are accessing the row via the column name.
rows = cur.fetchall()
for row in rows:
    print "   ", row['notes'][1]
```

Esempi

```
DSN= {'user':'root', 'db':'limey','passwd':'wicomemaster', 'host':'172.18.0.2', 'port':3306}
def get_response(q):
    d=[]
    try:
        cnx = mdb.connect(**DSN)
    except Exception as err:
        if DBG :
           print("GetResponse",d)
        hexcept(err,f"-- Executing query in get_response --{q}",exit=True)
    try:
        with cnx.cursor() as cursor:
            cursor.execute(q)
            desc = cursor.description
            column_names = [col[0] for col in desc]
            d = [dict(zip(column_names, row)) for row in cursor.fetchall()]
    except Exception as err:
        hexcept(err, f"-- Executing query in get_response --{q}",level=logging.CRITICAL,exit=True)
    else:
        return d
```

Alchemy

SQLAlchemy può essere usato per caricare automaticamente le tabelle da un database usando qualcosa chiamato reflection. La riflessione è il processo di lettura del database e creazione dei metadati basati su tali informazioni.

vedi jupyter notebook Alchemy

CSV Methods

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

```
import csv
with open('employee_birthday.txt') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        else:
            print(f'\t{row[0]} works in the {row[1]} department, and was born in {row[2]}.')
            line_count += 1
    print(f'Processed {line_count} lines.')
```

Parameters

- **delimiter** specifies the character used to separate each field. The default is the comma (',').
- **quotechar** specifies the character used to surround fields that contain the delimiter character. The default is a double quote (' " ').
- **escapechar** specifies the character used to escape the delimiter character, in case quotes aren't used. The default is no escape character.

```
#Read as Dictionary
import csv
with open('employee_birthday.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    line_count = 0
    for row in csv_reader:
        print(f'\t{row["name"]} works in the \ {row["department"]}
department, and was born in \ {row["birthday month"]}.')
        line_count += 1
    print(f'Processed {line_count} lines.')
```

```
#Write CSV
import csv
with open('employee_file.csv', mode='w') as employee_file:
    employee_writer = csv.writer(employee_file,\            delimiter=',',
quotechar='"', quoting=csv.QUOTE_MINIMAL)
    employee_writer.writerow(['John ', 'Acct', 'Nov'])
    employee_writer.writerow(['Erica ', 'IT', 'Mar'])

#with dict
import csv
with open('employee_file2.csv', mode='w') as csv_file:
    fieldnames = ['emp_name', 'dept', 'birth_month']
    writer = csv.DictWriter(csv_file,fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'emp_name': 'John', 'dept':'Acct', 'birth_month':
'Nov'})
    writer.writerow({'emp_name': 'Erica', 'dept':'IT', 'birth_month':
'Mar'})
```

Riferimenti per CSV

- https://docs.python.org/3/library/csv.html

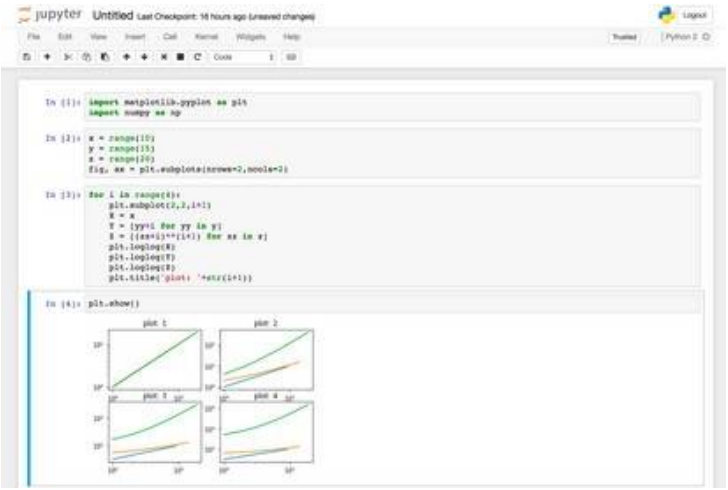
Pandas IO Manipulation

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

- A lot of reader
  - csv
    - text files (a.k.a. flat files) read_csv() and read_table().

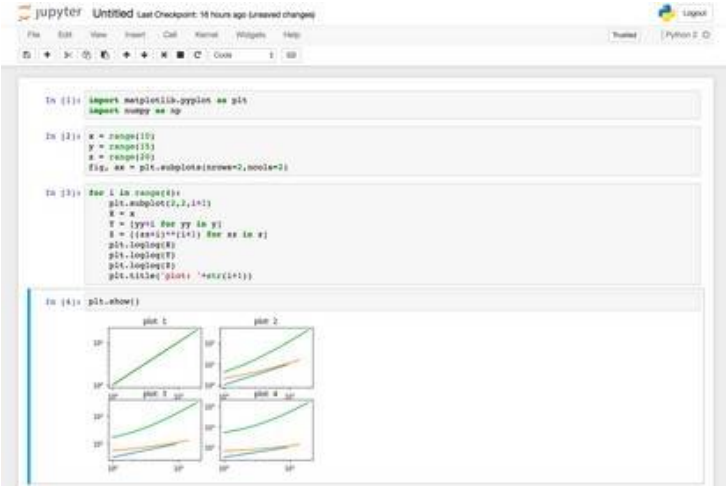
Esempio di lettura

vedi jupyter notebook
PandasLoadFile



Pandas IO CSV Basic Examples

vedi jupyter notebook
EsempiPandasIntro



Getting Data in/out

- HDF5 Store
  - HDF5 is a format designed to store large numerical arrays of homogenous type
    - a Python dictionary
    - import numpy as np
    - from pandas importHDFStore,DataFrame# create (or open) an hdf5 file and opens in append mode
    - hdf =HDFStore('storage.h5')
    - df =DataFrame(np.random.rand(5,3), columns=('A','B','C'))# put the dataset in the storage
    - hdf.put('d1', df, format='table', data_columns=True)
      - Accessing
        - print hdf['d1'].shape
        - from pandas import read_hdf
        - # this query selects the columns A and B# where the values of A is greather than 0.5
        - hdf = read_hdf('storage.h5','d1',where=['A>.5'], columns=['A','B'])
      - Appending
        - hdf.append('d1',DataFrame(np.random.rand(5,3),
        - columns=('A','B','C')),
        - format='table', data_columns=True)
        - hdf.close()# closes the file

CSV e PANDAS §

```
Name,Hire Date,Salary,Sick Days remaining
Graham Chapman,03/15/14,50000.00,10
John Cleese,06/01/15,65000.00,8
Eric Idle,05/12/14,45000.00,10
Terry Jones,11/01/13,70000.00,3
Terry Gilliam,08/12/14,48000.00,7
Michael Palin,05/23/13,66000.00,8
```

```
import pandas
df = pandas.read_csv('hrdata.csv')
print(df)
```

First, pandas recognized that the first line of the CSV contained column names, and used them automatically. I call this Goodness.

Pandas is also using zero-based integer indices in the DataFrame → we didn't tell it what our index should be.

Pandas has properly converted the Salary and Sick Days remaining columns to numbers, but the Hire Date column is still a String.

To use a different column as the DataFrame index, add the index_col optional parameter:

```
import pandas
df = pandas.read_csv('hrdata.csv',
            index_col='Employee',
            parse_dates=['Hired'],
            header=0,
            names=['Employee',
'Hired','Salary', 'Sick Days'])
print(df)
```

```
import pandas
df = pandas.read_csv('hrdata.csv', index_col='Name')
print(df)
```

Shell

Name	Hire Date	Salary	Sick Days remaining
Graham Chapman	03/15/14	50000.0	10
John Cleese	06/01/15	65000.0	8
Eric Idle	05/12/14	45000.0	10
Terry Jones	11/01/13	70000.0	3
Terry Gilliam	08/12/14	48000.0	7
Michael Palin	05/23/13	66000.0	8

```
# Rescan to interpret a column as a date
import pandas
df = pandas.read_csv('hrdata.csv', index_col='Name', parse_dates=['Hire Date'])
print(df)
```

CSV Writing with Pandas

```
import pandas
df = pandas.read_csv('hrdata.csv',
            index_col='Employee',
            parse_dates=['Hired'],
            header=0,
            names=['Employee', 'Hired', 'Salary',
'Sick Days'])
df.to_csv('hrdata_modified.csv')
```

PANDAS e SQL

- Vedi pandassql translate notebook

Librerie specifiche per xlsx, xls,csv

vedi jupyter notebook EsempiXlsX

- Openpyxl
  - Salvaguarda i formati
  - legge e scrive
  - non mantiene link tra i file
  - legge e modifica file esistenti (il che vuol dire che possiamo utilizzare template)
- XlsxWriter
  - Formatta
  - Mantiene le charts
  - Integrata con Pandas
  - Ottimizzazione per la memoria
  - Ma **It cannot read or modify existing Excel XLSX files.**

Python Json

- Supporto nativo
- codificare json → processo di serializzazione dei dati (marshaling)
- decodificare json → deserializzazione
- corrispondenza dei tipi

Python	JSON
`dict`	`object`
`list`, `tuple`	`array`
`str`	`string`
`int`, `long`, `float`	`number`
`True`	`true`
`False`	`false`
`None`	`null`

Librerie specifiche per json

- Javascript Object Notation
- De facto uno standard per il trasporto dei dati
- Molti database hanno un supporto diretto per i dati in formato json
- Fornitori ed erogatori di servizi dati tramite web service comunicano via json
- Alternative oggi sono XML e YAML
- Assomiglia molto ad un dizionario Python

```
{
    "firstName": "Jane",
    "lastName": "Doe",
    "hobbies": ["running", "sky diving", "singing"],
    "age": 35,
    "children": [
        {
            "firstName": "Alice",
            "age": 6
        },
        {
            "firstName": "Bob",
            "age": 8
        }
    ]
}
```

JSON

- Dict → JSON
  - encoding json → serialization
  - json decoding → deserialization

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

```
data = {
    "president": {
        "name": "Zaphod Beeblebrox",
        "species": "Betelgeusian"
    }
}
```

{'president': {'name': 'Zaphod Beeblebrox', 'species': 'Betelgeusian'}}

```
with open("data_file.json", "w") as write_file:
    json.dump(data, write_file)

json_string = json.dumps(data)
```

Vari param:
indent=4
separators=(", ", ": ")

```
with open("data_file.json", "r") as read_file:
    data = json.load(read_file)

json_string = """
{
    "researcher": {
        "name": "Ford Prefect",
        "species": "Betelgeusian",
        "relatives": [
            {
                "name": "Zaphod Beeblebrox",
                "species": "Betelgeusian"
            }
        ]
    }
}
"""
data = json.loadS(json_string)
```

JSON Esempi

```
blackjack_hand = (8, "Q")
>>> encoded_hand = json.dumps(blackjack_hand)
>>> decoded_hand = json.loads(encoded_hand)
>>> blackjack_hand == decoded_hand
False
>>> type(blackjack_hand)
<class 'tuple'>
>>> type(decoded_hand)
<class 'list'>
>>> blackjack_hand == tuple(decoded_hand)
True
```

Altro esempio

```
user = {"UserName": "defelicem", "Password": "", "Domain": ""}
def dologin(user):
    try:
       logindata=json.dumps(user).encode('utf-8')
       headers = {'Content-type': 'application/json'}
       response = requests.post(urlogin, data=logindata,headers=headers)
    except Exception as  msg:
       hexcept(msg)
    #TODO Check errors
    ris = json.loads(response.json())
    if ris["Errore"]["@id_errore"] == "0":
        err = (0, "")
    else:
        err=ris["Errore"]["@id_errore"],ris["Errore"]["#text"]

    return (ris["Sessione"]["#text"],err)
```

Python e MongoDB

- SQL
  - The model is of a relational nature
  - Data is stored in tables
  - Suitable for solutions where every record is of the same kind and possesses the same properties
  - Adding a new property means you have to alter the whole schema
  - The schema is very strict
  - ACID transactions are supported
  - Scales well vertically
- NoSQL
  - The model is non-relational
  - May be stored as JSON, key-value, etc. (depending on type of NoSQL database)
  - Not every record has to be of the same nature, making it very flexible
  - Add new properties to data without disturbing anything
  - No schema requirements to adhere to
  - Support for ACID transactions can vary depending on which NoSQL DB is used
  - Consistency can vary
  - Scales well horizontally

Vari tipi ognuno in genere specificoper un caso d'uso

- Key-Value Store: DynamoDB
- Document Store: CouchDB, MongoDB, RethinkDB
- Column Store: Cassandra
- Data-Structures: Redis

Mongo DB

- Support for many of the standard query types, like matching (==), comparison (<, >), or even regex
- Store virtually any kind of data - be it structured, partially structured, or even polymorphic
- To scale up and handle more queries, just add more machines
- It is highly flexible and agile, allowing you to quickly develop your applications
- Being a document-based database means you can store all the information regarding your model in a single document
- Change the schema of your database on the fly
- Many relational database functionalities are also available in MongoDB (e.g. indexing)

Vantaggi

- Whether you need a standalone server or complete clusters of independent servers, MongoDB is as scalable as you need it to be
- Load balancing support by automatically moving data across the various shards
- Automatic failover support - in case your primary server goes down, a new primary will be up and running automatically
- Thanks to the memory mapped files, you'll save quite a bit of RAM, unlike relational databases
- If you take advantage of the indexing features, much of this data will be kept in memory for quick retrieval. And even without indexing on specific document keys, Mongo caches quite a bit of data using the least recently used method.

Drawbacks

- Lack of support for ACID transactions.
  - Mongo does support ACID transactions in a limited sense, but not in all cases.
  - At the single-document level, ACID transactions are supported (which is where most transactions take place anyway).
  - However, transactions dealing with multiple documents are not supported due to Mongo's distributed nature.
- Lacks support for native joins, which must be done manually (and therefore much more slowly).
  - Documents are meant to be all-encompassing, which means, in general, they shouldn't need to reference other documents.
  - In the real world this doesn't always work as much of the data we work with is relational by nature. Mongo seems to be used as a complementary db database to a SQL DB, but as you use MongoDB you'll find that is not necessarily true.

Driver §

- pip install pymongo mongoengine
- persistenza su -v mongodb:/data/db/ (creare volume)
- docker run
- docker start
- use testdb
- db.people.save({firstname: 'fabrizio', lastname : 'iannucci'})
- db.people.find({ firstname: "Vlad" })

Espressioni regolari - https://pythex.org/

- Linguaggio per esprimere una stringa generica con determinate proprietà
- Si fa spesso confusione con il bash globbing (Wildcards)
- una espressione regolare rappresenta un pattern (modello) di caratteri ed utilizza per descriverlo un formalismo che fa uso di metacaratteri (ovvero caratteri che in un contesto assumono uno specifico significato)
- L'interprete del linguaggio è leggermente diverso a seconda dell'implementazione e di conseguenza il linguaggio non è proprio univoco
  - bash, ksh, awk, grep, vi possono usare motori leggermente diversi tra loro a seconda della implementazione, della versione anche all'interno dello stesso programma
- In genere ci si riferisce alla implementazione C,C++ ICU
  - vedi anche
  - bash usa la cosiddetta ERE extended regular expression
  - python è abbastanza simile ad ERE anche più a PCRE

Espressioni regolari

- ? 0 o una corrispondenza **del caratt precedente**; a?
  - globbing un qualsiasi singolo carattere
- * da zero a n **corrispondenze caratt. precedente**;
  - globbing 0 uno o più qualsiasi carattere
- + da una a n corrispondenze del car. precedente

- **Quoted braces**
  - {m} il caratt precedente appare esattamente m volte
    - cia**o**{N} – lo appare esattamente N volte;
  - {m,n} il car. precedente appare almeno m volte ma non più di n
  - {m,} il car. precedente appare almeno m volte

***?, +?, ??**

- The '*', '+', and '?' qualifiers are all **greedy**; they match as much text as possible.
- if the RE <.*> is matched against '<a> b <c>', it will match the entire string, and not just '<a>'.
- Adding ? after the qualifier makes it perform the match in non-greedy or minimal fashion; as few characters as possible will be matched. Using the RE <.*?> will match only '<a>'.

Espressioni regolari

- . ogni carattere;
- [a-z] - indica un singolo carattere all'interno dell'intervallo a-z;
- [abcd] - indica un singolo carattere all'interno dell'insieme indicato;
- [^AZ] ne A ne Z – globbing [!AZ]
- [ ] wildcard, almeno un **carattere** tra quelli tra parentesi. globbing uguale
  - { } indicano una wildcard, le **stringhe** tra parentesi separate da , ma ATTENZIONE solo globbing
- named class
  - [[:alpha:]], [[:blank:]], [[:cntrl:]], [[:digit:]], [[:lower:]], [[:upper:]],[[:punct:]], [[:space:]], and [[:alnum:]]
- **Wildcards**
  - ? 0 o una corrispondenza **del carattere precedente**;
    - globbing un qualsiasi singolo carattere
  - * da zero a n **corrispondenze caratt. precedente**;
    - globbing 0 uno o più qualsiasi carattere
    - Se vogliamo espicitare il numero delle ricorrenze o\{3\} match cooo ma non coro
- **Ancore**
  - **^ inizio della riga** – detta Ancora
    - ^ciao - la stringa "ciao" all'inizio della riga;
  - **$ fine della riga** – detta Ancora
    - ciao$ - la stringa "ciao" alla fine della riga;
    - ^a.*b$ - indica una riga che inizia con "a" e finisce con "b";
  - \< Inizio parola
    - es \<ca matcha cat ma non acaia
  - \>fine parola
    - es. \>ca matcha forca ma non cara
  - \b ai bordi della parola (inizio o fine)
  - \B non ai bordi della parola
  - p[ioa]zz[oia]
  - [A-z]

- \>giorno - la stringa "giorno" alla fine di una parola (es. buongiorno corrisponde a questo pattern);
- \b stringa vuota all'estremo di una parola
- \B stringa vuota non all'estremo di una parola
- \< stringa vuota all'inizio di una parola
- \> stringa vuota alla fine di una parola
- \ rappresenta normalmente l'escape dei metacaratteri

- + da una a n corrispondenze del car. precedente
- Vedi successiva quoted braces
- {m} il caratt precedente appare esattamente m volte
- cia**o**{N} – la o appare **esattamente** N volte;
- {m,n} il car. precedente appare almeno m volte ma non più di n
- {m,} il car. precedente appare **almeno** m volte

Espressioni regolari

- Nota: sebbene possa essere ovvio per alcuni, vale la pena enfatizzare quali caratteri svolgono un ruolo nell'iterazione.
- In tutti gli esempi precedenti solo il carattere che precede immediatamente il carattere o l'espressione di iterazione prende parte all'iterazione, tutti gli altri caratteri nell'espressione di ricerca (espressione regolare) sono literal.
- Quindi, nell'esempio di espressione di ricerca **colou?r**, la stringa **colo** è un valore letterale e deve essere trovata prima che venga avviata la sequenza di iterazione (**u?**) Che, se soddisfatta, deve essere seguita anche dal letterale r perché si verifichi una corrispondenza .

Quiz

- Cosa matchano ?
  - grep '^[M-Z]' rubrica.txt
  - grep '^[^M-Z]' rubrica.txt
  - grep '^L.*3$' rubrica.txt
  - grep '^P.tt' rubrica.txt
  - grep 'n*' rubrica.txt

Graffe quotate – occorrenze consecutive

- regular_expression\{min, max\}
  - 2,3 o 4 occorrenze di una combinazione di caratteri di 3 e 4 e 5 consecutive
  - grep '[345]\{2,4\}' phone.list
    - **grep anche { senza \{ nelle versioni moderne o -E**
- regular_expression\{exact\}
  - ogni linea con esattamente due caratteri r
  - grep 'r\{2\}' phone.list
- regular_expression\{min,\}
  - almeno due r consecutive precedute da e
  - grep 'er\{2,\}' phone.list

Parentesi quotate

- \(regular expression\)
- memorizza in uno di 9 registri accessibili con \1 \2 .. \9
  - grep '\(.\)\1' rubrica.txt cerca due caratteri uguali consecutivi

QUIZ

- Nella nostra rubrica quale è la espressione regolare che ci permette di trovare:
  - Persone con cognome di 6 lettere
  - Persone con nome di piu di cinque lettere
  - Tutti gli elementi con prefisso 56
  - Cognomi che cominciano per A o B o C?

Estensioni specifiche Python - vedi

- (?…) → This is an extension notation (a '?' following a '(' is not meaningful otherwise). The first character after the '?' determines what the meaning and further syntax of the construct is. Extensions usually do not create a new group; (?P<name>...) is the only exception to this rule. Following are the currently supported extensions.
- (?aiLmsux) → (One or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags: re.A (ASCII-only matching), re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode matching), and re.X (verbose), for the entire regular expression. (The flags are described in Module Contents.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the re.compile() function. Flags should be used first in the expression string.
- (?:…) → A non-capturing version of regular parentheses. Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.
- (?aiLmsux-imsx:...)
  - (Zero or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x', optionally followed by '-' followed by one or more letters from the 'i', 'm', 's', 'x'.) The letters set or remove the corresponding flags: re.A (ASCII-only matching), re.I (ignore case), re.L (locale dependent), re.M (multi-line), re.S (dot matches all), re.U (Unicode matching), and re.X (verbose), for the part of the expression. (The flags are described in Module Contents.)
  - Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name name. Group names must be valid Python identifiers, and each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.
- Named groups can be referenced in three contexts. If the pattern is (?P<quote>['"]).*?(?P=quote) (i.e. matching a string quoted with either single or double quotes):
- Context of reference to group “quote”
  - Ways to reference it in the same pattern itself
    - (?P=quote) (as shown)
    - \1
  - when processing match object m
    - m.group('quote')
    - m.end('quote') (etc.)
  - in a string passed to the repl argument of re.sub()
    - \g<quote>
    - \g<1
    - \1
- (?P=name) → A backreference to a named group; it matches whatever text was matched by the earlier group named name.
- (?#…) → A comment; the contents of the parentheses are simply ignored.
- (?=…) → Matches if ... matches next, but doesn’t consume any of the string. This is called a lookahead assertion. For example, Isaac (?=Asimov) will match 'Isaac ' only if it’s followed by 'Asimov'.
- (?!…) → Matches if ... doesn’t match next. This is a negative lookahead assertion. For example, Isaac (?!Asimov) will match 'Isaac ' only if it’s not followed by 'Asimov'.
- (?<=…) → Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a positive lookbehind assertion. (?<=abc)def will find a match in 'abcdef', since the lookbehind will back up 3 characters and check if the contained pattern matches. The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* and a{3,4} are not. Note that patterns which start with positive lookbehind assertions will not match at the beginning of the string being searched; you will most likely want to use the search() function rather than the match() function: