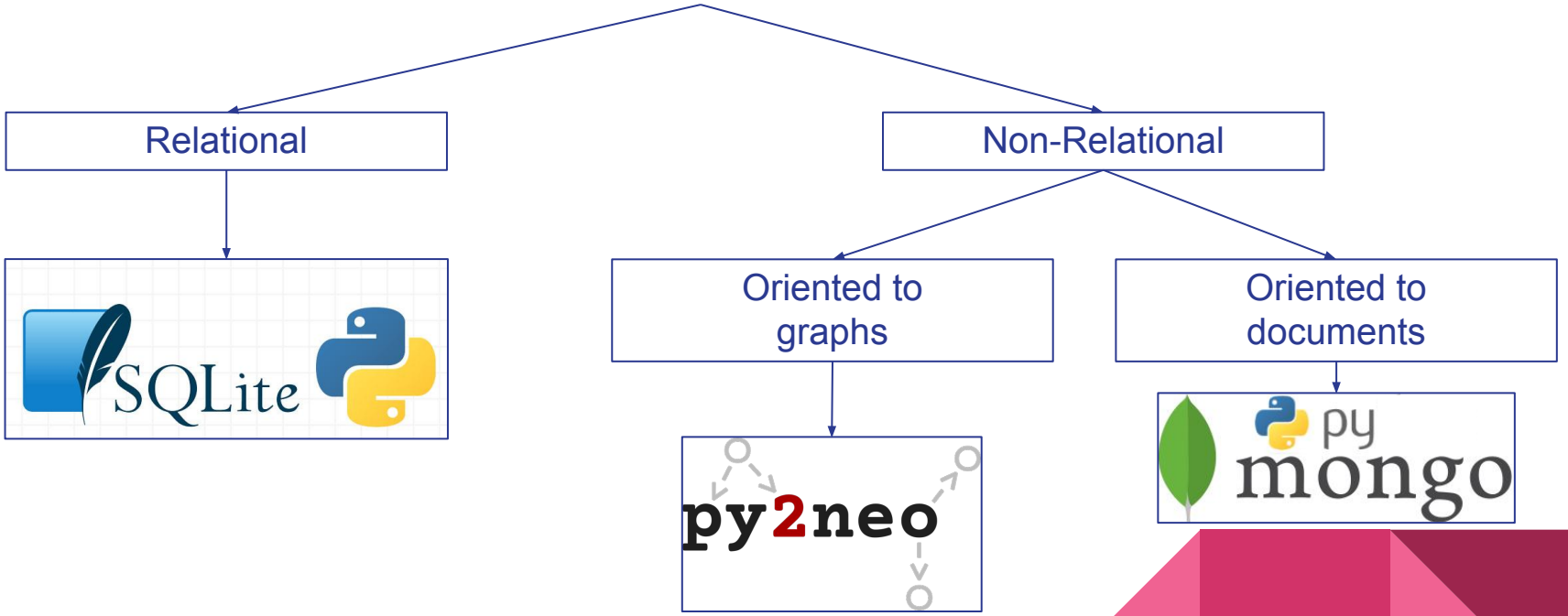


# Scripting no Processamento de Linguagem Natural

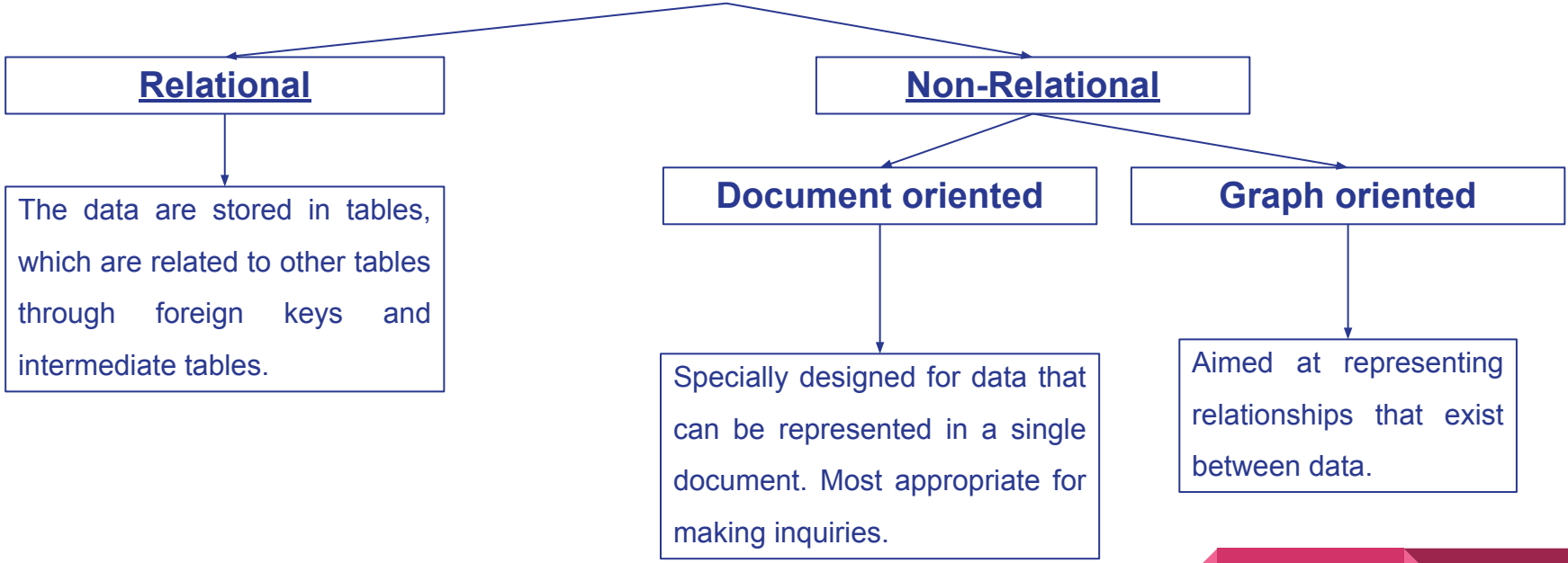
## Trabalho Prático 2

Etienne Costa - a76089
Gonçalo Pinto - a83732
Luís Ribeiro - a85954

**OBJECTIVE**: Analysis and study of Python modules that allow the use of different database models.



# Models Characterization

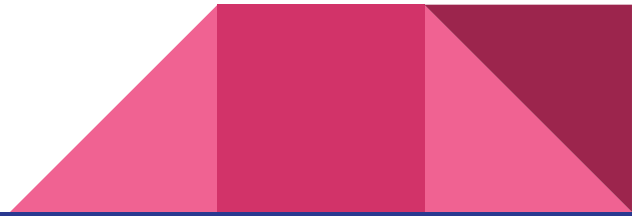


# PyMongo

# PyMongo

“PyMongo” is a python distribution containing tools for working with MongoDB, and is the recommended way to work with MongoDB from python.

```
$ pip3 install pymongo
```



# MongoDB

“MongoDB” is a document database, which means it stores data in JSON-like documents.
The database for modern applications.

Advantages :

- Schemaless
- Faster access to data
- Open Source
- Sharding

Disadvantages :

- More memory usage for data storage
- There is a limit for document size
- There is no transaction

# MongoDB Instance

The code bellow will connect on the default host and port:

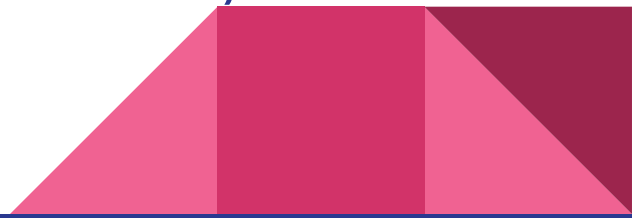
```
client = MongoClient()
```

We also can change that to specified the host and port :

```
client = MongoClient('localhost',27017)
```

Or use the MongoDB URI format :

```
client = MongoClient('mongodb://localhost:27017')
```



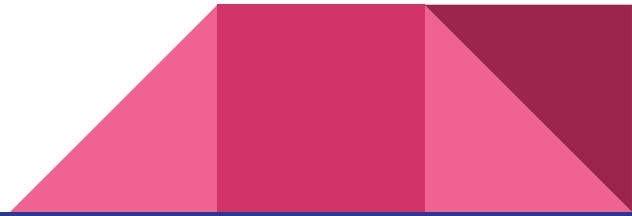
# MongoDB - Accessing Databases and Collections

Create or Access a database named spln

spln = client.spln or spln = client[‘spln’]

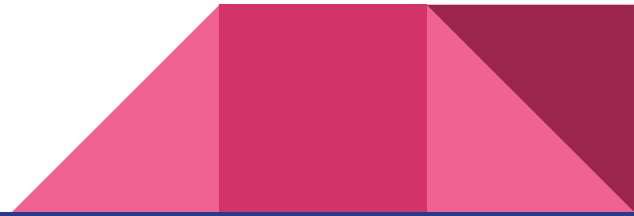
Create or Access a collection named students

students = spln.students or students = spln[‘students’]



# MongoDB - Data in MongoDB

Data in MongoDB is represented and stored using JSON-style documents.
In PyMongo we use dictionaries to represent documents.



# MongoDB - Data in MongoDB

```
student = {
    “number”: ”A76089”,
    “name”: ”Etienne Costa”,
    “course”: “MIEI”
    “date”: datetime.datetime.utcnow()
    }
```

# MongoDB - Data in MongoDB

```
spln_students= [ {
                    ”number” : ”A85954”,
                    ”name” : ”Luís Ribeiro”,
                    ”course” : ”MIEI”
                    ”date” : datetime.datetime.utcnow()
                    },
                    {
                     ”number” : ”A83732”,
                     ”name” : ”Gonçalo Pinto”,
                     ”course” : ”MIEI”,
                     ”date” : datetime.datetime.now()
                    } ]
```

# MongoDB - Inserting Document

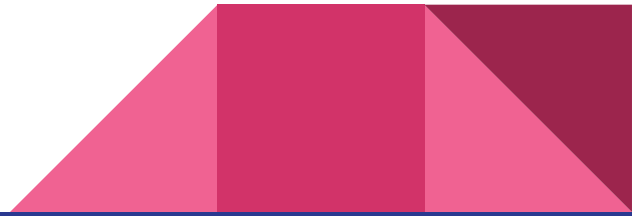
We have different manners to insert data into a collection :

To insert a document into a collection we can use the insert_one() method :

```
students.insert_one(student)
```

To insert multiples documents into a collection we can use the insert_many() method :

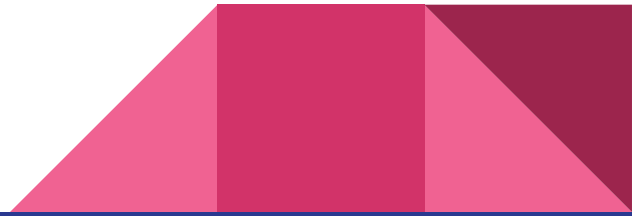
```
students.insert_many(spln_students)
```



# MongoDB - Inserting Document

When a document is inserted a special key, ”_id”, is automatically added if the document doesn’t already contain an ”_id” key.

The value of ”_id ” must be unique across the collection.



# MongoDB - Basic Queries

Verify all of the collections in our database :

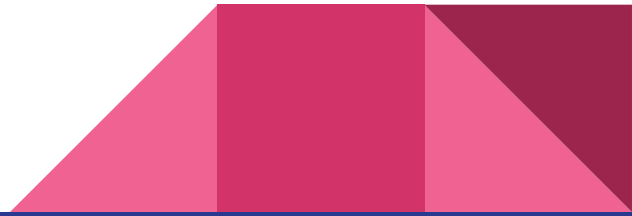
```
collections = spln.list_collection_names()
```

Number of students from ”MIEI” :

```
students.count_documents({”course” : ”MIEI”})
```

Return a single document matching a query or None if there are no matches :

```
students.find_one()
```



# MongoDB - Basic Queries

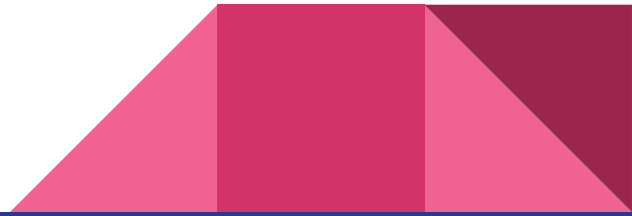
Returns all documents from the students collections :

```
students.find()
```

For a better visualization we can use the pretty print module:

```
import pprint

for s in students.find():
    pprint.pprint(s)
```



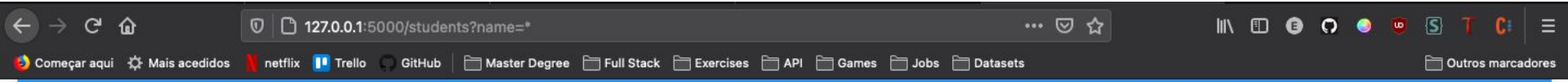
# Flask - Requiring something from url

## Flask is a micro web framework written in Python

```
app = Flask(__name__)
@app.route('/students',methods=['GET'])
def list_students():
        name = request.args['name'])
        result=""
        if name=="*":
                students = list(mongo.get_all_students())
        else :
                students = list(mongo.get_students_by_name(name))

        for s in students :
                result+="""
                        <tr>
                                <td>{id}</td>
                                <td>{name}</td>
                                ...
                        </tr>
                                """.format(id=s["number"], name=s["name"],...,...)
        return some_html_text
```

# Flask - Requiring something from url




## SPLN Students

| ID | Name | Course | Date |
|---|---|---|---|
| A76089 | Etienne Costa | MIEI | 2021-04-30 16:24:51.637000 |
| A85954 | Luís Ribeiro | MIEI | 2021-04-30 16:24:51.637000 |
| A83732 | Gonçalo Pinto | MIEI | 2021-04-30 16:24:51.637000 |
| A20200 | Albert Einstein | MEI | 2021-04-30 16:24:51.637000 |

PyMongo Tutorial

# Explicit Encryption with Automatic Decryption

```
import os
import pprint
from pymongo import MongoClient
from pymongo.encryption import Algorithm,ClientEncryption
from pymongo.encryption_options import AutoEncryptionOpts
```

os.urandom(n) -> str : Return n random bytes suitable for cryptographic use :

```
local_master_key = os.urandom(96)
```

Key Management Service : Specifies that the key is from a local provider :

```
kms_providers = {”local” : { ”key” : local_master_key } }
```

# Explicit Encryption with Automatic Decryption

The MongoDB namespace used to store the encryption data keys :

```
key_vault_namespace = ”encryption.pymongoTestKeyVault”
```

Split the namespace to get the database and collection name :

```
key_vault_db_name, key_vault_coll_name = key_vault_namespace.split(”.”,1)
```

bypass_auto_encryption = True disable automatic encryption but keeps the automatic decryption behavior :

```
auto_encryption_opts = AutoEncryptionOpts (
kms_providers,key_vault_namespace,bypass_auto_encryption = True )
```

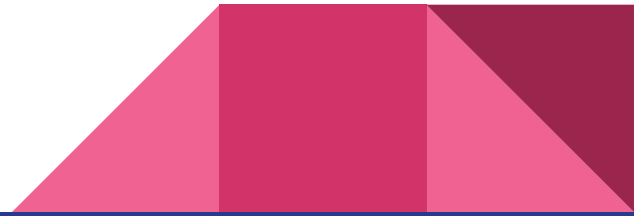
# Explicit Encryption with Automatic Decryption

A single instance of MongoDB with auto_encryption_opts configuration :

```
client = MongoClient(auto_encryption_opts=auto_encryption_opts)
```

Students is an instance of a collection inside the SPLNEncrypted database :

```
students = client.SPLNEncrypted.students
```



# Explicit Encryption with Automatic Decryption

The ClientEncryption provides an API for explicity encryption and decrypting values, and creating data keys :

```
client_encryption = ClientEncryption(
                        kms_providers,
                        key_vault_namespace,
                        # The MongoClient to use for reading/writing to the key vault.
                        client,
                        # The CodecOptions class used for encrypting and decrypting.
                        students.codec_options )
```

Create a new data key for the encryptedField :

```
data_key_id = client_encryption.create_data_key(
  'local', key_alt_names=[ 'pymongo_encryption_example' ])
```

# Explicit Encryption with Automatic Decryption

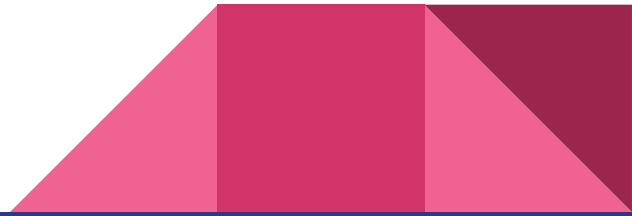
Explicitly encrypt a field :

```
password = client_encryption.encrypt(
                ”123456789”,
                Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic,
                key_alt_name=’pymongo_encryption_example’
                )
```

User with encrypted_password :

```
user = { ”username” : ”Etienne Costa”,
          ”password” : password

        }
```



# Explicit Encryption with Automatic Decryption

To insert a document into a collection :

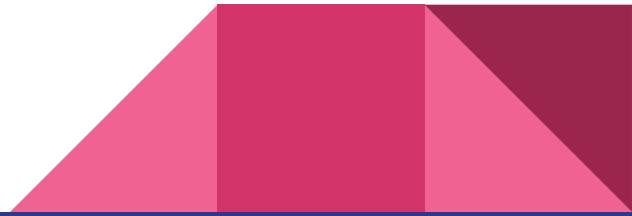
```
students.insert_one(user)
```

Automatically decrypts any encrypted fields :

```
print('=========================================')
print('Decrypted document : ')
pprint.pprint(students.find_one())
```

To check that document field is encrypted :

```
print('=========================================')
unencrypted_coll = MongoClient().SPLNEncrypted.students
print('Encrypted document  :')
pprint.pprint(unencrypted_coll.find_one())
```



# Explicit Encryption with Automatic Decryption

```
============================================================
Decrypted document:
{'_id': ObjectId('608c1e44f2c3cca8560ab96f'),
 'password': '123456789',
 'username': 'Etienne Costa'}
============================================================
Encrypted document:
{'_id': ObjectId('608c1e44f2c3cca8560ab96f'),
 'password': Binary(b'\x01\x1e\x94\xa4\xacH\xafF3\x84\x8c\xb3\xb1\x8a\x90\xa2>\x02\xf9\xc0\xf5v\xea\xfe\xd2\xd3Q?\x8d\xe1\xecJ\x01#*r\xea:\x02\xf4N\x95\x88\xefM\xbe\xd6\xe8\xea\xe4\x91\x97\xe0\x0fY9q\xde\xd8#\x89\x1f5\x86\x85\x9a\x7f]\xeb\x1a3\xa3\xfet(\xb0\x08\xa8\x10\xb4d\xed', 6),
 'username': 'Etienne Costa'}
```

# Py2Neo

# Graph-oriented non-relational model

In order to use a non-relational graph-oriented model, there are several options of management systems for this type of model, such as:

- Amazon Neptune;
- ArangoDB;
- Neo4j;
- Ontotext GraphDB;
- etc.

Since we already have some experience with the Neo4j software we present the module that is a driver for this type of databases, which is Py2neo, however it is not the official driver!

```
$ pip3 install py2neo
```

# Py2Neo vs. Official Driver

Py2neo offers more features, such as a higher level API, however the official driver (neo4j) provides mechanisms for working with clusters.

- For a beginner in this new model who doesn't need to explore the search language a lot, which is Cypher, or who just needs to integrate data into a graph-oriented database, Py2neo is the best choice.

- On the other hand, if we are creating a robust application with high availability or that uses clusters, it is more recommended to use the official driver.

# Py2Neo - Linking & Data Objects

```
from py2neo import Graph, Node, Relationship
graph = Graph("http://localhost:7474", auth=("neo4j","password"))
```

- **Nodes** are the fundamental units and can contain properties, such as key-value pairs;
- **Relationships** are important components in this type of database, each relationship contains an initial node and an end node;
- **Properties** are pairs of key-values to describe both nodes and relationships;
- **Tags** are tags that associate a common name to a set of nodes or relationships.

```
# Node(*type*,**properties**)
a = Node("Person", name="Alice")
b = Node("Person", name="Bob")
# Relationship(*start_node*, *type*, *end_node*, **properties**)
ab = Relationship(a, "KNOWS", b)
graph.create(...)
```

# Py2Neo - Queries

Most database management systems that use graphs have their own query language, the system used (Neo4j) uses Cypher as the query language, so this module provides 2 methods for making queries about the information entered:

```
from py2neo.matching import *

# Node and relationship matching
nodes = NodeMatcher(graph)
bob = nodes.match("Person", name="Bob")

# Writing Cypher query
query = "MATCH (bob:Person{name:"Bob"}) RETURN bob"
bob = graph.run(query)
```

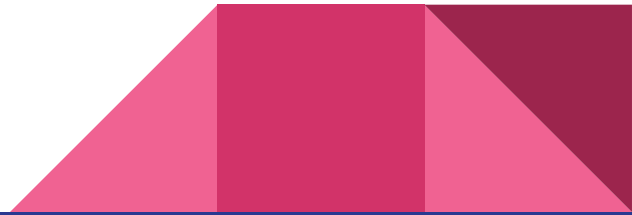
# Py2Neo - Demo

# SQLite

# SQLite

SQLite is a self-contained, file-based SQL database. SQLite comes bundled with Python and can be used in any of your Python applications without having to install any additional software.

Provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language.



# SQLite - Why should you use it

SQLite is often the technology of choice for small applications, particularly those of embedded systems and devices like phones and tablets, smart appliances, and instruments.

Advantages :

- Easy of use and requires minimal configuration.
- Good for IOT projects and local/offline projects.
- Reliable.
- Better Performance compared to the File System.
- Accessible.
- Lightweight.

Disadvantages :

- Multi-user database with multiple connections.
- Database size is restricted to 2GB in most cases - Big Data Problem.
- Lacks complex data manipulations.

# SQLite - Configuration

- You don't need to install sqlite3 module. It is included in the standard library (since Python 2.5).
- You can also check the documentation by executing $ pydoc sqlite3

To import this module into your python file, just add: `import` sqlite3

# SQLite - Useful Commands

```
import sqlite3

# Criar uma conexão ao ficheiro que representa a nossa BD.
con = sqlite3.connect('test.db')
# Depois de criar uma conexão, devemos criar um cursor para manipular a nossa BD.
cur = con.cursor()
# Garante a integridade dos dados relacionados por foreign keys.
con.execute("PRAGMA foreign_keys = 1")
# Usar o cursor para executar uma query SQL -> CREATE,INSERT,DELETE,SELECT...
cur.execute(query)
cur.executemany(query,list) # Dar uma lista de parâmetros que serão lidos para a query.
# Tratar da informação recebida de um SELECT à BD.
cur.fetchall()
# Guardar o estado alterado.
con.commit()
# Retornar ao último commit.
con.rollback()
# Usar extensões SQLite externas.
con.enable_load_extension(True)
# Fechar a conexão.
con.close()
```

# SQLite - Demo



# Scripting no Processamento de Linguagem Natural

## Trabalho Prático 2

Etienne Costa - a76089
Gonçalo Pinto - a83732
Luís Ribeiro - a85954