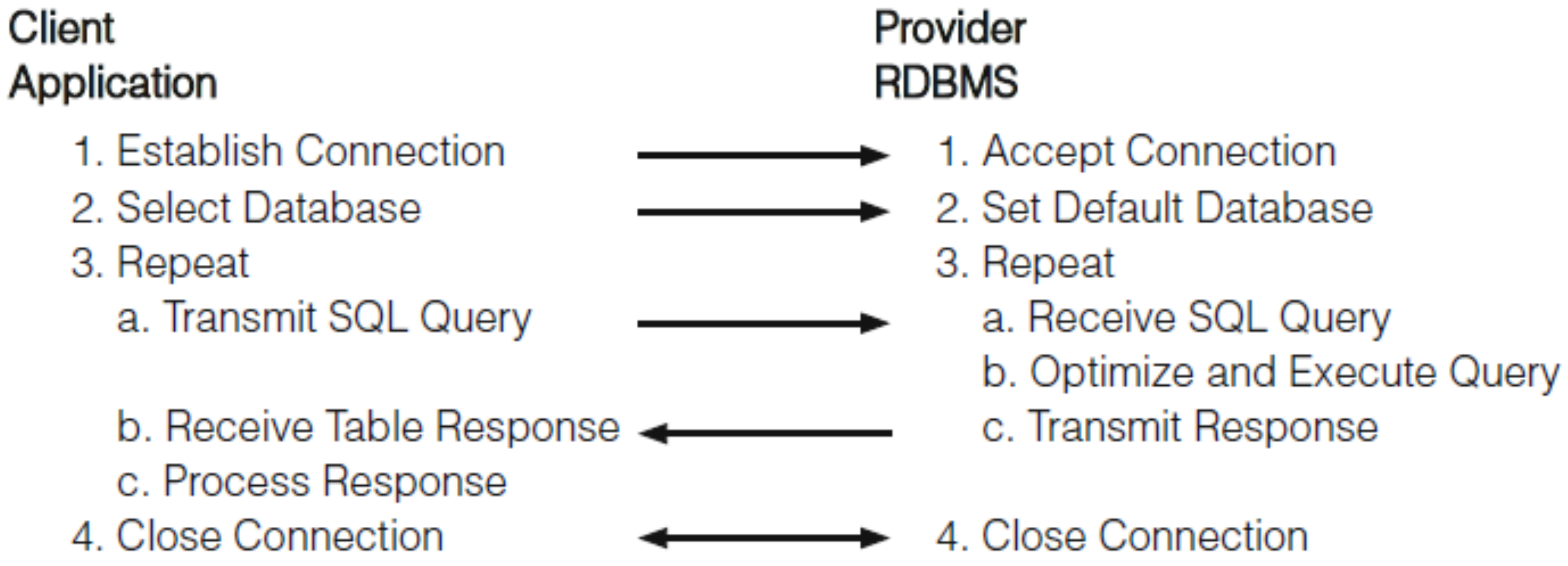
Chapter 13

Relational Model: Database Programming

Client/Provider Interaction



Database Connection

- For many database systems, the connection may occur over a network, but for some, like SQLite, this connection is *virtual* and is realized through interaction with the local operating system.
- Over this connection, individual SQL queries are transmitted. Note that different database systems use different protocols.
- The interaction between a relational database client and a database management system proceeds in the same way, whether the client is written in Python, R, Java, or C++.
- In this chapter we will proceed using Python and will be using the `sqlalchemy` package.

The Connection String

- For `sqlalchemy` to make a connection to a database, it needs multiple pieces of information. It must determine:
  - What protocol scheme and/or kind of database system is on the other end of the connection.
  - Which lower-level database library (driver) will be required.
  - If the database system is over a network, the network specifics for the machine and software process executing the provider-side software.

The Connection String Continued

– If the database is local, like SQLite, the file system path of the database.

– If the database system supports multiple users in the same system, the credentials for the user associated with the currently executing client application connecting to this system.

– If the database system supports multiple databases, the specific database schema to select as the default database.

Connection Strings

- For illustration, we will show the construction of connection strings for both a network example, using a MySQL provider on a non-local machine, and for a local example, using SQLite.
- For information on connection strings for other specific database systems, the reader should refer to the database/engine connection documentation.
- In `sqlalchemy`, the connection string bears an intentional resemblance to a URL.

Dictionary

- Information about a database location is changeable and may contain password information, so it is advisable to put such information into an external file.

- A JSON file with a dictionary mapping data sources to the information needed in an appropriate connection string is shown as follows:

```
{
    "mysql": {"protocol": "mysql+mysqlconnector",
              "host": "server.college.edu",
              "user": "user",
              "password": "pass",
              "database": "book"},
    "sqlite": {"protocol": "sqlite+pysqlite",
               "path": "datadir/book.db"}
}
```

Python String Building

- Building a connection string is a matter of using Python string building and incorporating the information from the dictionary.
- In the upcoming examples, we use the Python string `format()` method with a pattern string to place the values into their corresponding places.

Python’s `format()` Method

- The `format()` method was introduced with Python3 for handling complex string formatting more efficiently.

- Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces `{}` into a string and calling `str.format()`.

Python's `format()` Method Examples

```
print('We all are {}.'.format('equal’))
```

Output:

```
We all are equal.
```

```
print('{2} {1} {0}'.format('directions',
                           'the', 'Read’))
```

This uses *index-based* positioning.

Output:

```
Read the directions.
```

`format()` with Assigned Keywords

```
print('a: {a}, b: {b}, c: {c}'.format(a = 1,
                                      b = 'Two',
                                      c = 12.3))
```

Output:

```
a: 1, b: Two, c: 12.3
```

```
print('The first {p} was alright, but the {p}
{p} was tough.'.format(p = 'second’))
```

Output:

```
The first second was alright, but the second
second was tough.
```

MySQL String Connection Example

```
credspath = os.path.join(datadir, "creds.json")
with open(credspath, ‘r’) as file:
    creds = json.load(file)

mysqlD = creds["mysql"]
pattern = "{}://{}:{}@{}/{}"
cstring = pattern.format(mysqlD["protocol"], mysqlD["user"],
                         mysqlD["password"], mysqlD["host"],
                         mysqlD["database"])
print(cstring)
```

```
| "mysql+mysqlconnector://user:pass@server.college.edu/book"
```

SQLite String Connection Example

```
credspath = os.path.join(datadir, "creds.json")
with open(credspath, ’r’) as file:
    creds = json.load(file)

sqliteD = creds["sqlite"]
pattern = "{}:///{}"
cstring = pattern.format(sqliteD["protocol"],
                         sqliteD["path"])
cstring
```

```
  | "sqlite+pysqlite:///datadir/book.db"
```

Connecting and Closing the Connection

- We import the `sqlalchemy` package and use an `as` clause to define a mnemonic (`sa`) to reference the functions and objects of the package.

- Online tutorials and references for the `sqlalchemy` package often use a

  ```
  from sqlalchemy import xxx, yyy
  ```

  form of import to gain access to `sqlalchemy` functions and objects without having to use the mnemonic, but while we are learning, being explicit about what package a function belongs to is preferred.

Connecting

- With most of the work being accomplished by building the connection string, establishing the connection can be realized with:

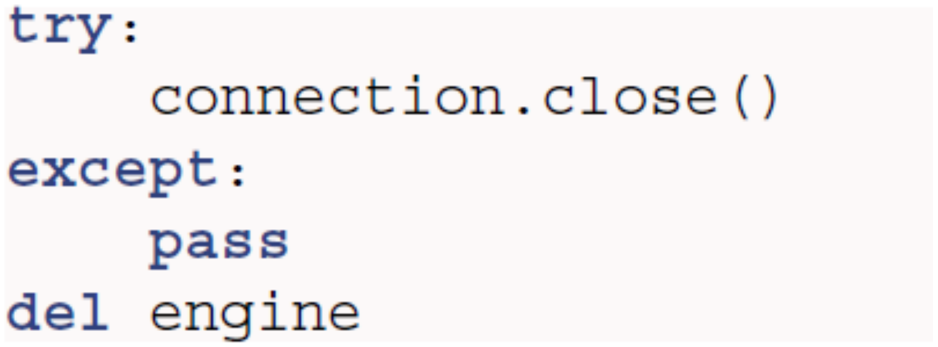
```
import sqlalchemy as sa

engine = sa.create_engine(cstring)
connection = engine.connect()
```

Closing

- A connection must be closed when its use is complete, and the engine should be deleted.

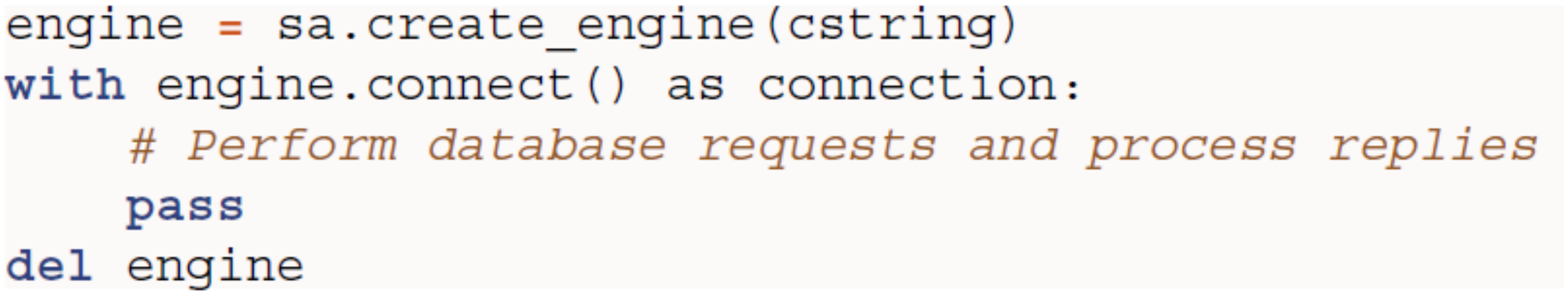
- Because a connection to a provider is consuming local resources and is consuming remote resources, with the provider using memory and execution threads for each client interaction, we need to be especially careful to do our proper cleanup.



```
try:
    connection.close()
except:
    pass
del engine
```

Using `with`

- Similar to opening and using files, we can use a `with` construct that, at the end of the `with`, will automatically perform a `close()`:



```
engine = sa.create_engine(cstring)
with engine.connect() as connection:
    # Perform database requests and process replies
    pass
del engine
```

Basic Query

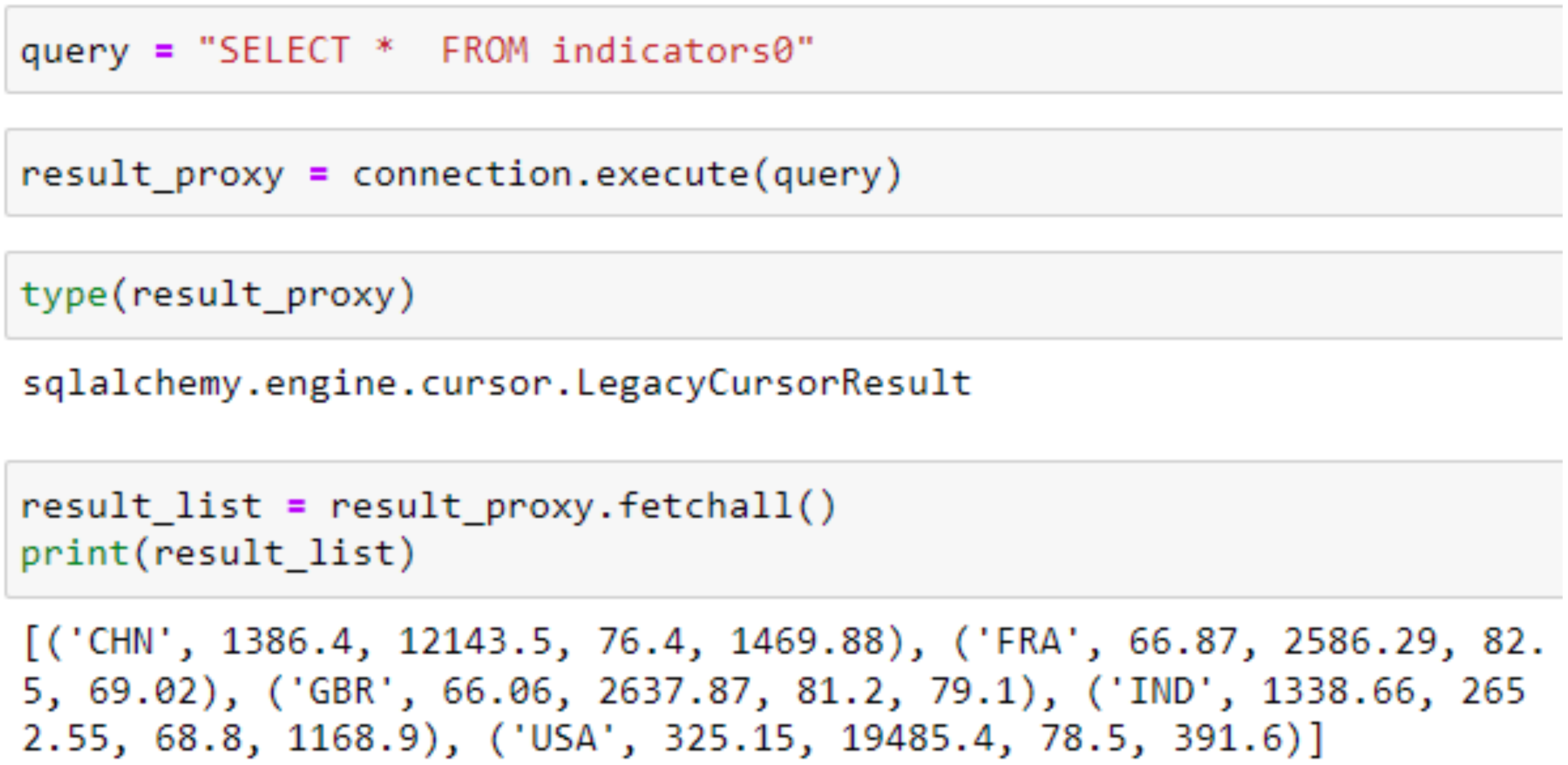
- Making a request always requires the two-step process of building the query into a Python string, and then calling the `execute()` method on the connection with the built string.

- Any valid SQL query, as covered in the previous two chapters, could make up the value of the Python query string.

Four Step Process

1. Compose the SQL query as a Python string.
2. Execute the query.
3. Fetch all records of the result.
4. Construct a `pandas` `DataFrame`.

Basic Query Example



```
query = "SELECT *  FROM indicators0"
```

```
result_proxy = connection.execute(query)
```

```
type(result_proxy)
```

```
sqlalchemy.engine.cursor.LegacyCursorResult
```

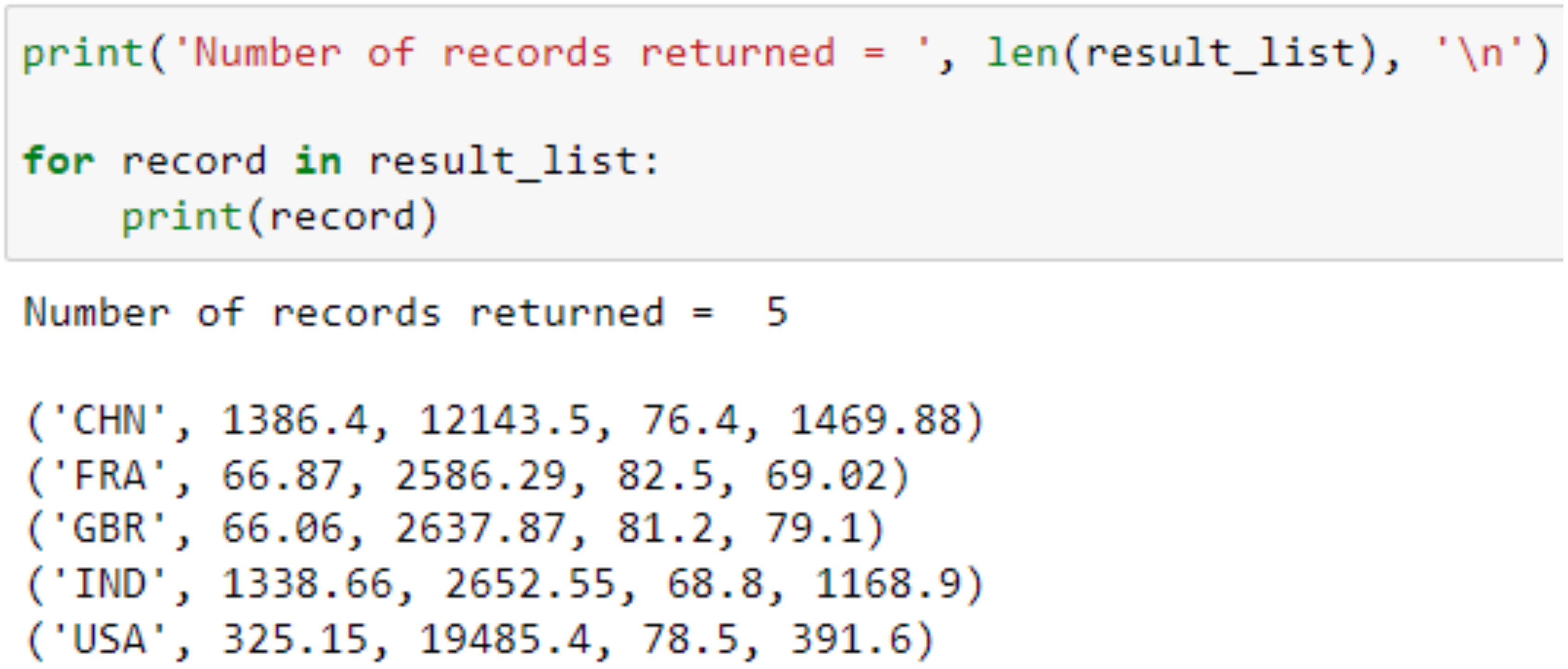
```
result_list = result_proxy.fetchall()
print(result_list)
```

```
[('CHN', 1386.4, 12143.5, 76.4, 1469.88), ('FRA', 66.87, 2586.29, 82.
5, 69.02), ('GBR', 66.06, 2637.87, 81.2, 79.1), ('IND', 1338.66, 265
2.55, 68.8, 1168.9), ('USA', 325.15, 19485.4, 78.5, 391.6)]
```

- The result is a list of the records of the resultant table.

Example Continued

- We can use `len()` to find the number of records resulting from the query.
- We can iterate over the list and print tuple records.



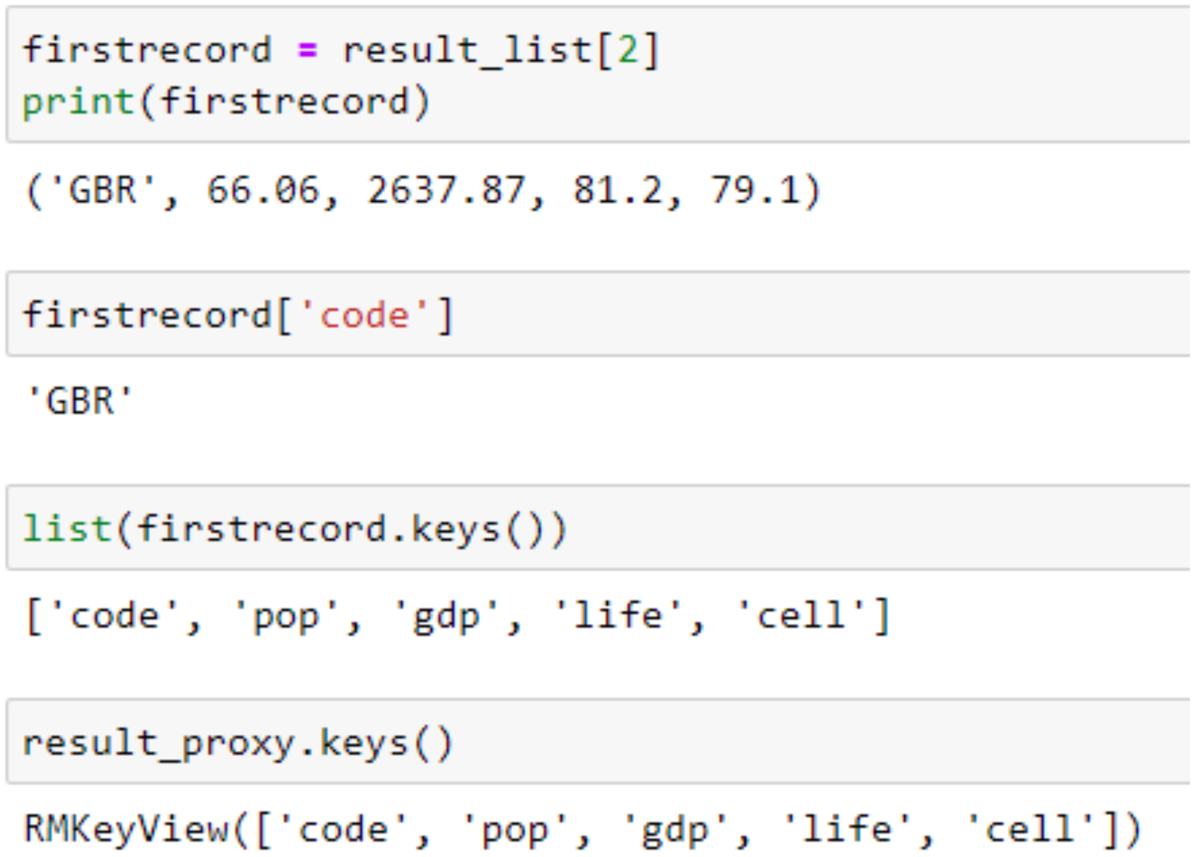
```
print('Number of records returned = ', len(result_list), '\n')

for record in result_list:
    print(record)
```

```
Number of records returned =  5

('CHN', 1386.4, 12143.5, 76.4, 1469.88)
('FRA', 66.87, 2586.29, 82.5, 69.02)
('GBR', 66.06, 2637.87, 81.2, 79.1)
('IND', 1338.66, 2652.55, 68.8, 1168.9)
('USA', 325.15, 19485.4, 78.5, 391.6)
```

Additional Functionality of Tuples



```
firstrecord = result_list[2]
print(firstrecord)
```

```
('GBR', 66.06, 2637.87, 81.2, 79.1)
```

```
firstrecord['code']
```

```
'GBR'
```

```
list(firstrecord.keys())
```

```
['code', 'pop', 'gdp', 'life', 'cell']
```

```
result_proxy.keys()
```

```
RMKeyView(['code', 'pop', 'gdp', 'life', 'cell'])
```

Native Data Structures to `pandas`

- We would like to built a `pandas` data frame from the results of an SQL query.

- We begin by simply using the `DataFrame` constructor passing the result:

```
ind0 = pd.DataFrame(result_list)
ind0
```

```
|      0        1         2     3        4
| 0  CHN  1386.40  12143.50  76.4  1469.88
| 1  FRA    66.87   2586.29  82.5    69.02
| 2  GBR    66.06   2637.87  81.2    79.10
| 3  IND  1338.66   2652.55  68.8  1168.90
| 4  USA   325.15  19485.40  78.5   391.60
```

Column Names

- The data in the table is as desired, but we want to provide `pandas` with the column/field names.
- We can use the `keys()` method of the result proxy to retrieve these as a list of strings and then pass both the data and the column names:

```
fields = result_proxy.keys()
ind0 = pd.DataFrame(result_list, columns=fields)
ind0
```

```
|    code       pop       gdp  life     cell
| 0   CHN   1386.40  12143.50  76.4  1469.88
| 1   FRA     66.87   2586.29  82.5    69.02
| 2   GBR     66.06   2637.87  81.2    79.10
| 3   IND   1338.66   2652.55  68.8  1168.90
| 4   USA    325.15  19485.40  78.5   391.60
```

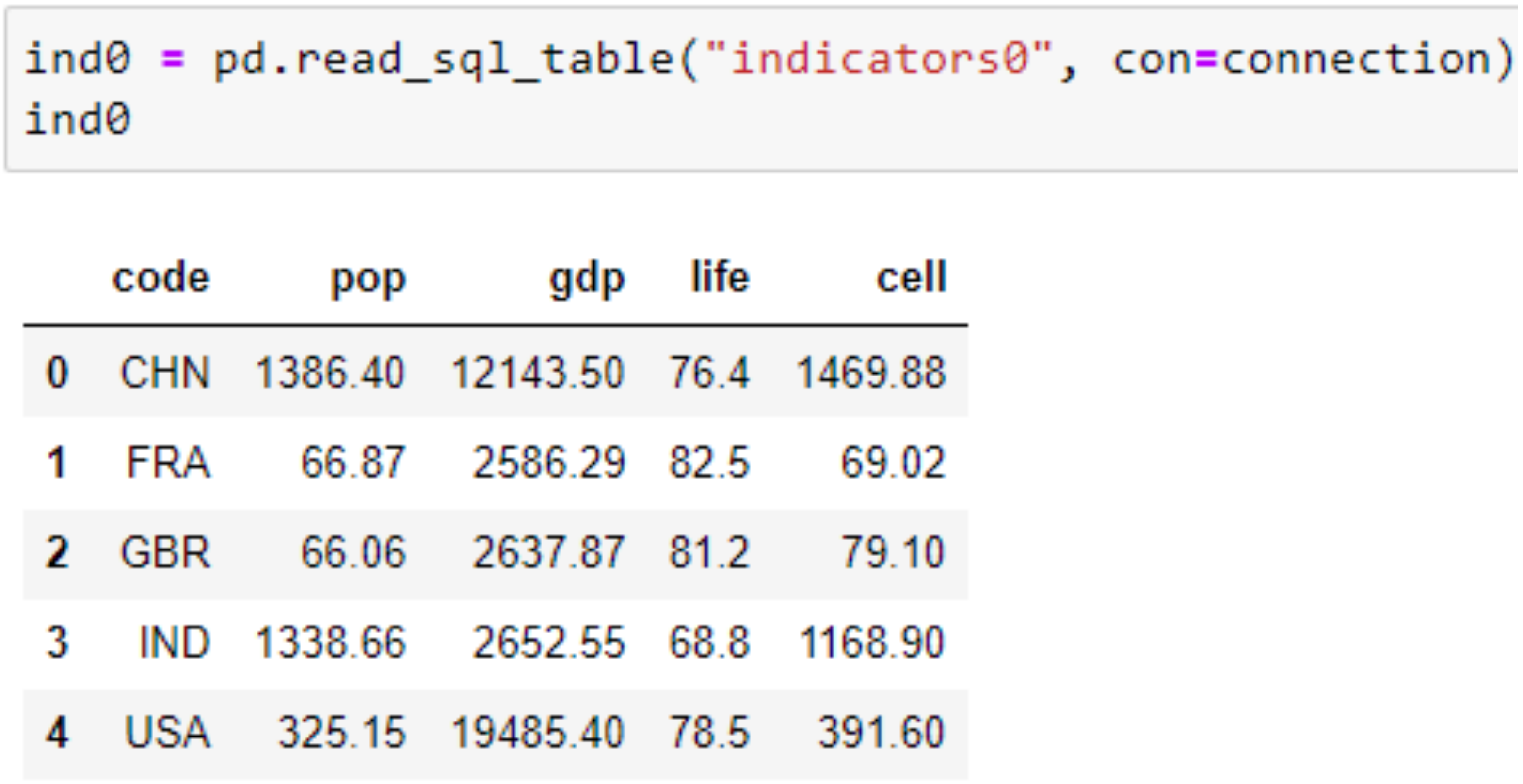
Database requests Directly Through Pandas

- The `pandas` module provides two useful functions for streamlining the four-step process.

1. If we desire an entire table, we can use the `read_sql_table()` function and the result is a Pandas data frame, compressing all four steps into one.

2. If we are not requesting an entire table, we can submit an SQL query using the `read_sql_query()` function. Here we compose the query and then invoke the function and the result is a pandas data frame.

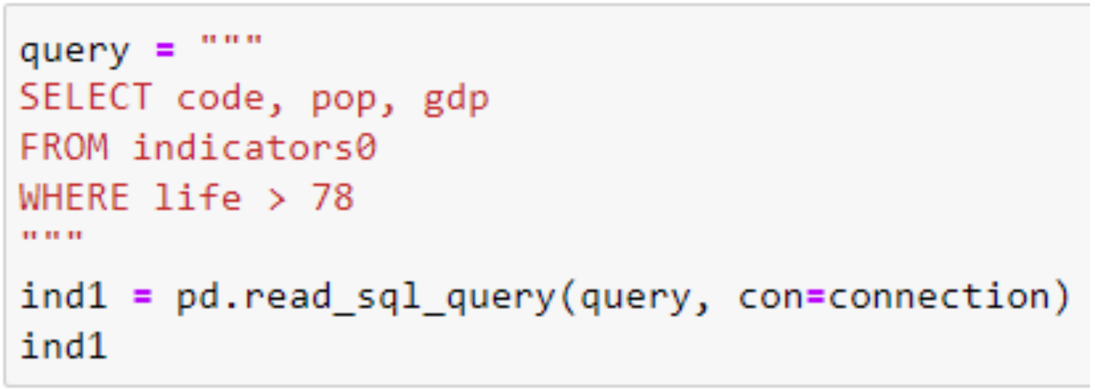
Read Entire Table



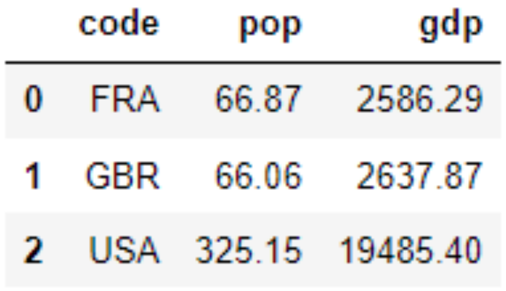
```
ind0 = pd.read_sql_table("indicators0", con=connection)
ind0
```

	code	pop	gdp	life	cell
0	CHN	1386.40	12143.50	76.4	1469.88
1	FRA	66.87	2586.29	82.5	69.02
2	GBR	66.06	2637.87	81.2	79.10
3	IND	1338.66	2652.55	68.8	1168.90
4	USA	325.15	19485.40	78.5	391.60

SQL Query



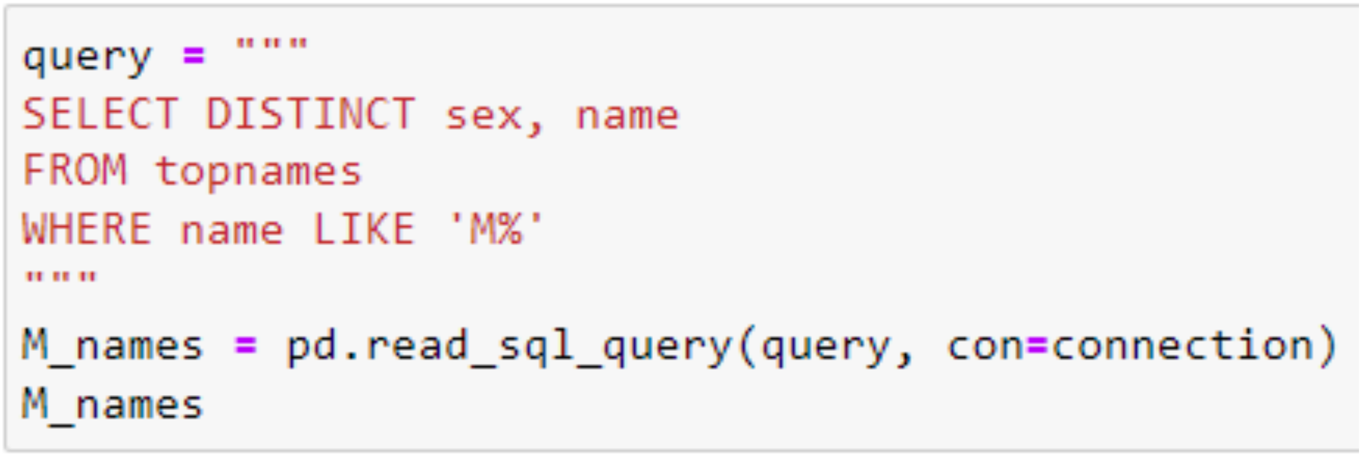
```
query = """
SELECT code, pop, gdp
FROM indicators0
WHERE life > 78
"""
ind1 = pd.read_sql_query(query, con=connection)
ind1
```



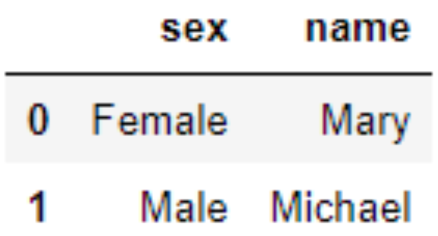
	code	pop	gdp
0	FRA	66.87	2586.29
1	GBR	66.06	2637.87
2	USA	325.15	19485.40

- It is advisable to used triple-double quoted strings when composing queries because you can compose the SQL query over multiple lines for readability (and since SQL itself uses single quote for string literals).

SQL Query with String Literal



```
query = """
SELECT DISTINCT sex, name
FROM topnames
WHERE name LIKE 'M%'
"""
M_names = pd.read_sql_query(query, con=connection)
M_names
```



	sex	name
0	Female	Mary
1	Male	Michael

Incorporating Variables

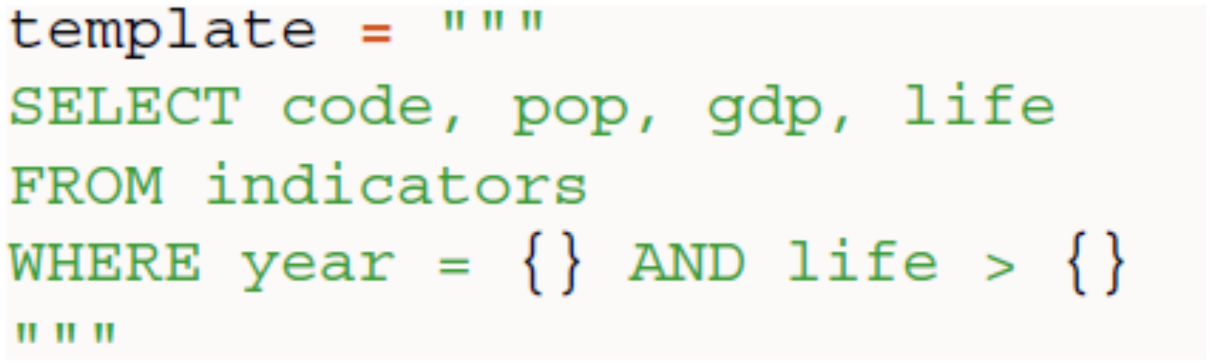
- We now want to learn how to incorporate variables into our SQL queries.

- That is, we wish to generalize a specific query to abstract some part of it so that we can dynamically create and execute the query based on parameters.

- Specifically, we can combine Python string manipulations, such as `format()`, string `+`, and `str()` with variables to create the queries.

Python String Composition

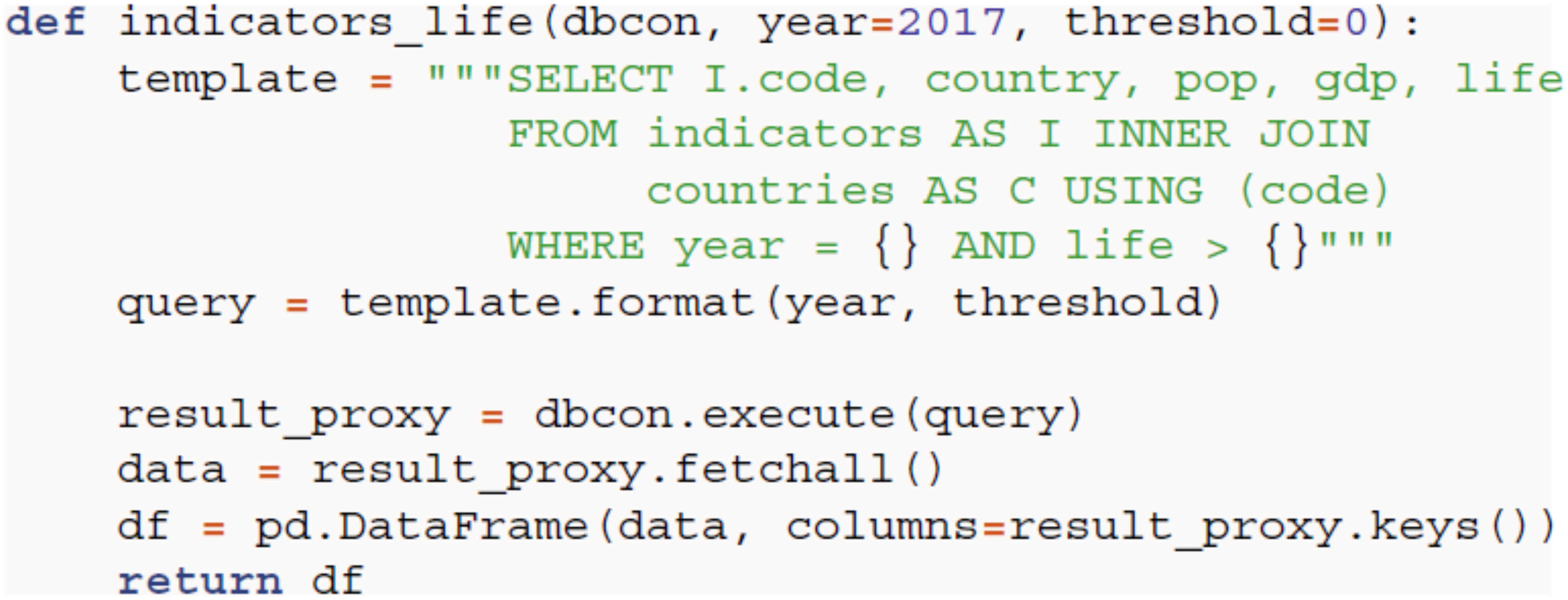
- Suppose we often request data from the `indicators` table for a specific year and based on a life expectancy threshold.
- We wish the value of the year and the value of the threshold to come from variables.
- The generic form of this query is:



```
template = """
SELECT code, pop, gdp, life
FROM indicators
WHERE year = {} AND life > {}
"""
```

Function Approach

- We can design a function to perform this query as follows:



```
def indicators_life(dbcon, year=2017, threshold=0):
    template = """SELECT I.code, country, pop, gdp, life
                  FROM indicators AS I INNER JOIN
                       countries AS C USING (code)
                  WHERE year = {} AND life > {}"""
    query = template.format(year, threshold)

    result_proxy = dbcon.execute(query)
    data = result_proxy.fetchall()
    df = pd.DataFrame(data, columns=result_proxy.keys())
    return df
```

Function Approach

```
indicators_life(connection, year=2000, threshold=80)
```

```
|    code                country     pop      gdp  life
| 0   HKG  Hong Kong SAR, China    6.66   171.67  80.9
| 1   JPN                  Japan  126.84  4887.52  81.1
| 2   MAC      Macao SAR, China    0.43     6.72  80.4
```

Function Approach Example 2

- When creating queries in this manner, we must be aware of the data types of both the Python variables and the SQL data types of the fields used in the integrated SQL query.

- Consider the example of writing a function to obtain a subset of the `students` table based on a string range that uses the `studentlast` field.

- We want to select those records where the student last name is between an inclusive low value and an exclusive high value.

Function Approach Example 2

```
template = """
SELECT studentid,
       studentlast || ', ' || studentfirst AS studentname,
       studentmajor
FROM students
WHERE studentlast >= {} AND studentlast < {}"""
```

- Consider the result of invoking `format()`, using an example low and high as shown:

```
low = "A"
high = "Am"
query = template.format(low, high)
print(query)
```

This query would not be interpreted correctly—why?

```
|
| SELECT studentid,
|        studentlast || ', ' || studentfirst AS studentname,
|        studentmajor
| FROM students
| WHERE studentlast >= A AND studentlast < Am
```

Function Approach Example 2

- SQL would interpret the previous query as asking if field `studentlast` is greater than or equal to *field* `A`, and if field studentlast is less than *field* `Am`.
- **Without the single quotes to inform SQL that these are string literals, an unadorned sequence of characters is interpreted as a field!**

Correct Solution

```
template = """
SELECT studentid,
       studentlast || ', ' || studentfirst AS studentname,
       studentmajor
FROM students
WHERE studentlast >= '{}' AND studentlast < '{}'
"""
```

Now we include single quotes with the template string

```
low = "A"
high = "Am"
query = template.format(low, high)
print(query)
```

```
|
| SELECT studentid,
|        studentlast || ', ' || studentfirst AS studentname,
|        studentmajor
| FROM students
| WHERE studentlast >= 'A' AND studentlast < 'Am'
```

Binding Variables

- `sqlalchemy` provide mechanisms to make incorporating variable values easier and more efficient that the previous string building examples. The general steps include:

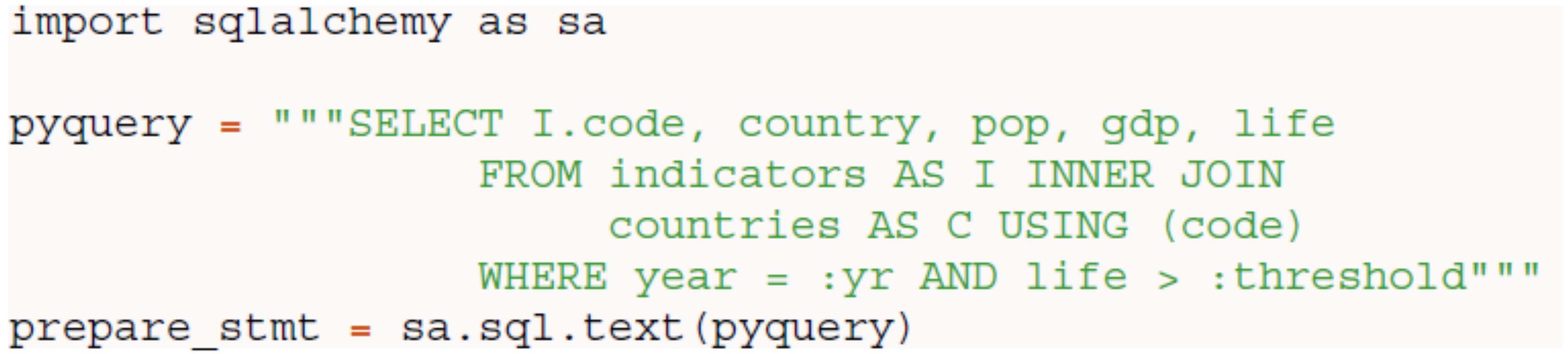
1. Construct an object that has a combination of SQL syntax and elements designating the places where the value of a variable should be substituted. This is called the ***prepare step***.

2. When the variable values are available, a ***bind operation*** is performed, that associates a set of values with their corresponding locations in the prepared SQL text. This step constructs a new object to obtain a ***bound statement***.

3. The bound statement is then conveyed as a request to the database.

`sql` Class of `sqlalchemy`

- Within the `sql` class of `sqlalchemy` is a `text()` constructor that builds a prepared object, consistent with step 1 above.



```
import sqlalchemy as sa

pyquery = """SELECT I.code, country, pop, gdp, life
                  FROM indicators AS I INNER JOIN
                       countries AS C USING (code)
                  WHERE year = :yr AND life > :threshold"""
prepare_stmt = sa.sql.text(pyquery)
```

- The `pyquery` variable references a Python string that uses the syntax for binding, which are designated with a colon (`:`) immediately followed by a programmer selected name (`yr` and `threshold`) that should obey Python variable naming syntax.

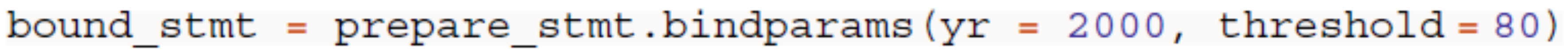
- This SQL-specific template is then prepared by calling `text()` and assigning to the `prepare_stmt` variable for later reference.

Bind Step

- The bind step invokes the `bindparams()` method of a prepared statement yielding a *bound statement*.

- The `bindparams()` method associates values by simply using named parameters as its arguments.

- In this case, the named parameters are `yr` and `threshold`, and creation of the bound object is the single step:



```
bound_stmt = prepare_stmt.bindparams(yr = 2000, threshold = 80)
```

Execute Step

- The execute step is performed, as before, but using the bound statement instead of a string for a query.

- We can use the `execute()` method and `fetchall()` to obtain the results, or we can use the `read_sql_query()` function of `pandas` to combine those steps and return a data frame:

```
df = pd.read_sql_query(bound_stmt, connection)
df
```

```
|   code              country     pop      gdp  life
| 0  HKG  Hong Kong SAR, China    6.66   171.67  80.9
| 1  JPN                 Japan  126.84  4887.52  81.1
| 2  MAC      Macao SAR, China    0.43     6.72  80.4
```

Named Parameters

- The `execute()` method, if given named parameters, will combine the bind and execute steps on our behalf, allowing us to skip an explicit bind:

```
result_proxy = connection.execute(prepare_stmt, yr=1999,
                                  threshold=70)
data = result_proxy.fetchall()
print('Number of records returned:', len(data))
```

```
Number of records returned: 96
```

Advantages?

- Two of the advantages of using SQL variable binding over string-based composition of queries are reusability and the carrying of type information.

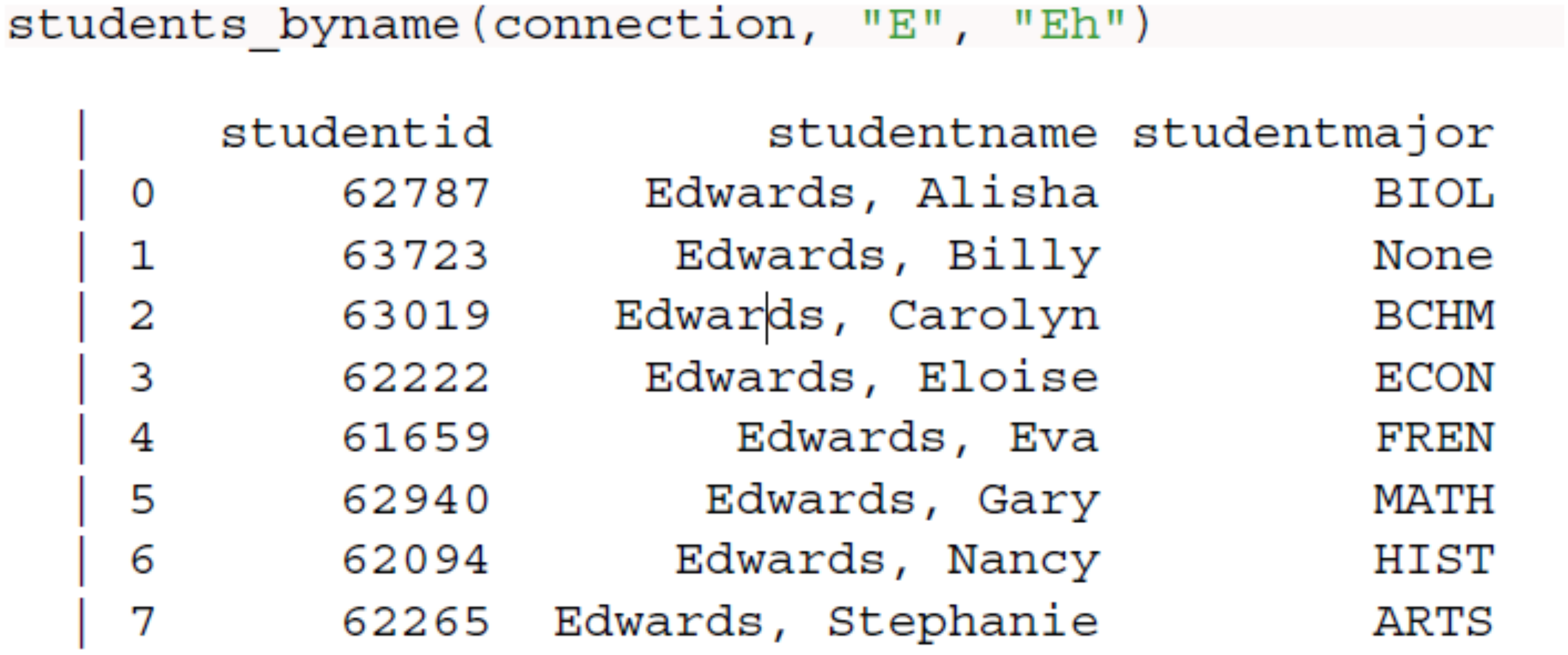
- Variable binding in our SQL queries can also give us a clean way to build functional abstractions that nicely generalize operations we wish to perform against a database.

Example

```
def students_byname ( dbcon, name_min, name_max ):
    pyquery = """SELECT studentid, studentlast || ', '
                          || studentfirst AS studentname,
                 studentmajor
                 FROM students
                 WHERE studentlast >= :low AND
                       studentlast <  :high
                 ORDER BY studentname"""
    prepare_stmt = sa.sql.text(pyquery)

    bound_stmt = prepare_stmt.bindparams(low=name_min,
                                         high=name_max)
    df = pd.read_sql_query(bound_stmt, con=dbcon)
    return df
```

Example Continued



```
students_byname(connection, "E", "Eh")
```

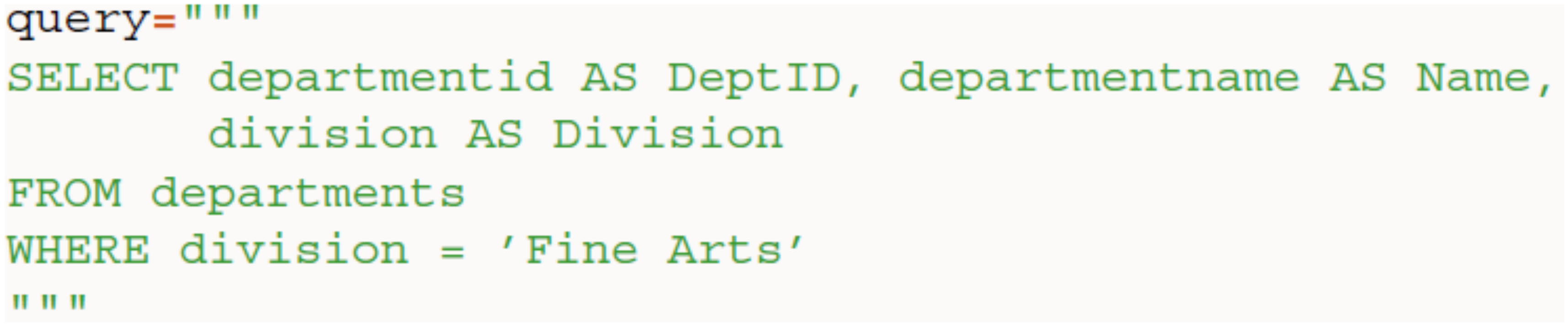
```
   studentid           studentname studentmajor
0      62787       Edwards, Alisha         BIOL
1      63723        Edwards, Billy         None
2      63019      Edwards, Carolyn         BCHM
3      62222       Edwards, Eloise         ECON
4      61659          Edwards, Eva         FREN
5      62940         Edwards, Gary         MATH
6      62094        Edwards, Nancy         HIST
7      62265   Edwards, Stephanie         ARTS
```

More Advanced Techniques

- As the number of rows retrieved in a result from a relational database system grows larger, we need greater control over how we retrieve the data itself.
- This control is one of the reasons that, on a query, the object returned is a result proxy.
- The result proxy is a relatively small object that can be transmitted from the provider to the client application, and then, using the proxy, we can fetch rows of the result.
- **In this way, the application can choose to retrieve subsets of the data, down to processing the result a record at a time**.

Example Query

- To illustrate the different techniques for obtaining records we will use the following query:



```
query="""
SELECT departmentid AS DeptID, departmentname AS Name,
       division AS Division
FROM departments
WHERE division = 'Fine Arts'
"""
```

Result Proxy as an Iterator

- One alternative for getting the data a record at a time is through direct iteration with the result proxy. A result proxy can act as an iterator, and so we can use it in a for loop and obtain each of the records in the result, one at a time:

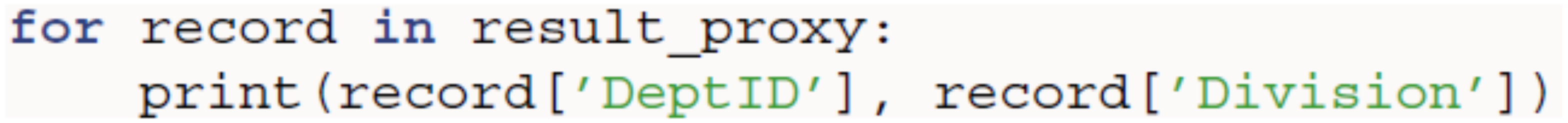
```
result_proxy = connection.execute(query)
```

```
for record in result_proxy:
    print(record)
```

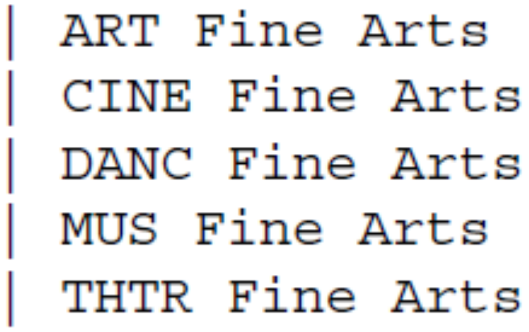
```
| ('ART', 'Art History and Visual Culture', 'Fine Arts')
| ('CINE', 'Cinema', 'Fine Arts')
| ('DANC', 'Dance', 'Fine Arts')
| ('MUS', 'Music', 'Fine Arts')
| ('THTR', 'Theatre', 'Fine Arts')
```

Result Proxy as an Iterator

- Alternatively:



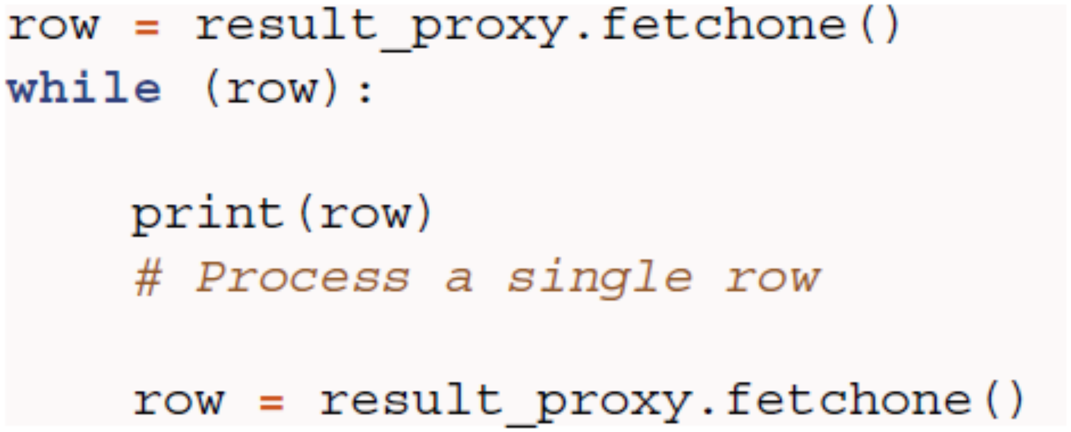
```
for record in result_proxy:
    print(record['DeptID'], record['Division'])
```



```
| ART Fine Arts
| CINE Fine Arts
| DANC Fine Arts
| MUS Fine Arts
| THTR Fine Arts
```

Fetch One

- A result proxy also has a method named `fetchone()` that obtains the next record, keeping track of a current position in the set of result records.



```
row = result_proxy.fetchone()
while (row):

    print(row)
    # Process a single row

    row = result_proxy.fetchone()
```

- Notice that the result, `row`, evaluates to `False` when there are no more results to be processed, and allows termination of the while loop.

Chunks

- Processing records one at a time involves significantly more network processing than fetching the entire result.

- We may want a compromise between fetching all the records, which for some tables may be too large as a single result, and fetching one at a time, which add network latency and delay for each record.

- Logically, we want the control to specify a ***chunk***, which is an application-specified number of records to retrieve at a time.

Fetch Many

- The result proxy has a `fetchmany()` method that provides the capability of specifying the maximum number of records to retrieve in one operation.

- The method takes a single argument of an integer number of records to attempt to fetch.

- To use a *chunk size*, we need our processing to operate at an outer level—where at each iteration we obtain a chunk-sized collection of result records, and an inner level—where at each iteration, we process the records within the chunk.

Chunk Example

- Consider the following example, where we process with a chunk size of (at most) two records per fetch.
- The outer loop is an indefinite loop that terminates when a `fetchmany()` returns a `None` result (no more chunks).
- The inner loop uses the chunk returned by `fetchmany()` as an iterator that, on each iteration, yields a single record of the chunk result.

Chunk Example

```
chunk_size = 2
rowset = result_proxy.fetchmany(chunk_size)
while (rowset):
    print("Next chunk @ length", len(rowset))
    for row in rowset:
        print(row)
        # Process a single row

    rowset = result_proxy.fetchmany(chunk_size)
```

```
| Next chunk @ length 2
| ('ART', 'Art History and Visual Culture', 'Fine Arts')
| ('CINE', 'Cinema', 'Fine Arts')
| Next chunk @ length 2
| ('DANC', 'Dance', 'Fine Arts')
| ('MUS', 'Music', 'Fine Arts')
| Next chunk @ length 1
| ('THTR', 'Theatre', 'Fine Arts')
```

Using Pandas with Chunk Size

- The Pandas `read_sql_query()` method has a named parameter, `chunksize=`, that we can use to control the number of results fetched at a time.

- The method returns an iterator.

- This iterator yields a complete data frame on each next item iteration of the for loop.

Pandas Chunk Example

```
chunk_size = 3
iterator = pd.read_sql_query(query, con=connection,
                             chunksize=chunk_size)
for df in iterator:
    print(type(df))
    print(str(df))
```

```
<class 'pandas.core.frame.DataFrame'>
       name     sex
0      John    Male
1     James    Male
2  Jennifer  Female
<class 'pandas.core.frame.DataFrame'>
      name     sex
0  Jessica  Female
1    Jacob    Male
```

Unified Data Frame Example

- In the example below, we use the chunk ability of `read_sql_query()`, but combine the data frames into a single table.

```
chunk_size = 3
iterator = pd.read_sql_query(query, con=connection,
                             chunksize=chunk_size)

try:
    result_df = pd.DataFrame()
    while (True):
        df = next(iterator)
        print(type(df))
        print(str(df))
        print()
        result_df = pd.concat([result_df, df], axis=0,
                              ignore_index=True)
except Exception as e:
    pass
```

Working with Two Databases

- Sometimes the need arises for a client to work with multiple databases within a single application. There are two options:

Option 1: Explicitly create two engine objects and two connection objects, using different Python variable names. Then each connection `execute()` or the connection passed to Pandas would need to decide which connection to use for which query.

Option 2: Some DBMSs support multiple database schema served by the same provider (MySQL, for instance).

- – You can check the documentation of your DBMS for more details.