California State University Fullerton

CPSC-223P

Python Programming

Stephen T. May

Python Standard Library

Data Persistence – File Formats

https://docs.python.org/release/3.9.6/tutorial/index.html

Slide Notes

- Command typed at the Linux command prompt ($)

```
$ python3.9
```

- Command typed at the Python interpreter command prompt (>>>)

```
>>> Ctrl-D
```

- Python source code

```
print(“Hello world!”)
```

- Mixed example

```
>>> the world is flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

Data Persistence – sqlite3

- SQLite is a C library that 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
- Some applications can use SQLite for internal data storage
- It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle
- To use the module, you must first create a *Connection* object that represents the database
- You can also supply the special name `:memory:` to create a database in RAM.
- Once you have a *Connection*, you can create a *Cursor* object and call its `execute()` method to perform SQL commands

```
import sqlite3
con = sqlite3.connect('example.db')
```

```
cur = con.cursor()

# Create table
cur.execute('''CREATE TABLE stocks
               (date text, trans text, symbol text, qty real, price
real)''')

# Insert a row of data
cur.execute("INSERT INTO stocks VALUES ('2006-01-
05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
con.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be
lost.
con.close()
```

Data Persistence – `sqlite3` (cont.)

- The data you've saved is persistent and is available in subsequent sessions
- To retrieve data after executing a SELECT statement, you can either treat the cursor as an `iterator`, call the cursor's `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows

```
>>> for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

Python and SQLite types

Python type	SQLite type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

Data Persistence – `sqlite3` (cont.)

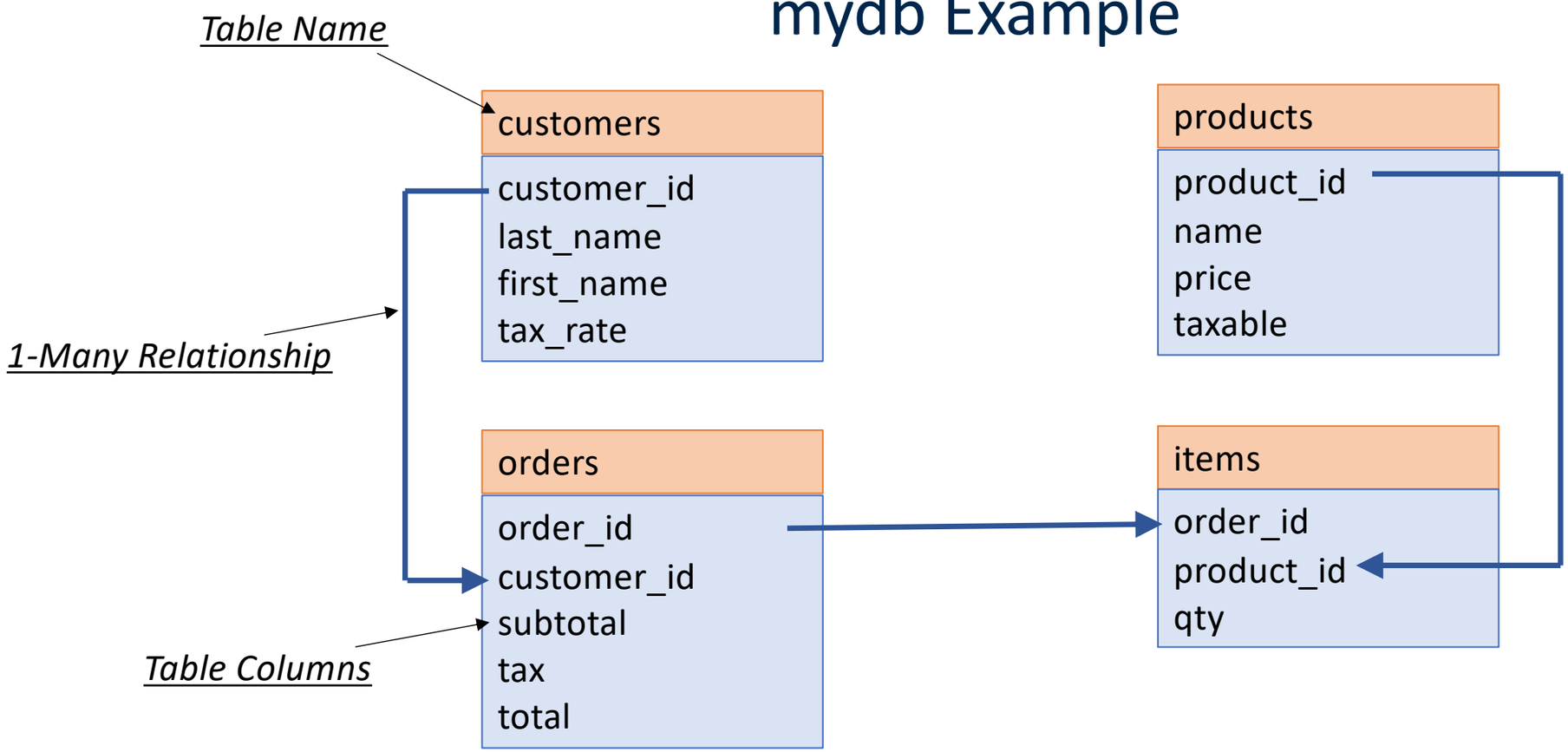
- Usually your SQL operations will need to use values from Python variables
- You shouldn't assemble your query using Python's string operations because doing so is insecure
- It makes your program vulnerable to an SQL injection attack
- Instead, use the DB-API's parameter substitution
- Put a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's `execute()` method
- An SQL statement may use one of two kinds of placeholders:
  - For the *qmark style*, parameters must be a sequence
  - For the *named style*, it can be either a sequence or dict instance
- The length of the sequence must match the number of placeholders, or a ProgrammingError is raised
- If a dict is given, it must contain keys for all named parameters
- Any extra items are ignored

```
# Never do this -- insecure!
symbol = 'RHAT'
cur.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)
```

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table lang (name, first_appeared)")

# This is the qmark style:
cur.execute("insert into lang values (?, ?)", ("C", 1972))

# The qmark style used with executemany():
lang_list = [
    ("Fortran", 1957),
    ("Python", 1991),
    ("Go", 2009),
]
cur.executemany("insert into lang values (?, ?)", lang_list)

# And this is the named style:
cur.execute("select * from lang where first_appeared=:year",
{"year": 1972})
print(cur.fetchall())

con.close()
```

Data Persistence – `sqlite3` (cont.)

Entity-Relationship Diagram
mydb Example



`cvs` - CSV File Reading and Writing

- Comma Separated Values format is the most common import and export format for spreadsheets and databases.
- The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications.
- While the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.
- The `csv` module implements classes to read and write tabular data in CSV format.
- The `csv` module's `reader` and `writer` objects read and write sequences. Programmers can also read and write data in dictionary form using the `DictReader` and `DictWriter` classes.

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|',
quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

cvs - CSV File Reading and Writing (cont.)

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name':    'Baked',    'last_name':
'Beans'})
    writer.writerow({'first_name':    'Lovely',    'last_name':
'Spam'})
    writer.writerow({'first_name':  'Wonderful',  'last_name':
'Spam'})
```

`configparser` - Configuration file parser

- This module provides the `ConfigParser` class which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files.
- You can use this to write Python programs which can be customized by end users easily.
- The structure of INI files is described in the following section.
- Essentially, the file consists of sections, each of which contains keys with values.
- `configparser` classes can read and write such files.
- As you can see, we can treat a config parser much like a dictionary.
- There are differences, outlined later, but the behavior is very close to what you would expect from a dictionary.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

example.ini:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

`configparser` - Configuration file parser (cont.)

- Now that we have created and saved a configuration file, let's read it back and explore the data it holds.
- As we can see above, the API is pretty straightforward.
- The only bit of magic involves the DEFAULT section which provides default values for all other sections.
- Note also that keys in sections are case-insensitive and stored in lowercase.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```