Chapter 12 - Database Programming

CS 172 - Computer Programming 2
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Database Programming

- There are different kinds of databases
- One of the most popular is the *Relational Database Management System* (RDBM)
  - These systems use tables (spreadsheet-like grids) with
    - rows equating to records
    - columns equating to fields
  - Tables and their data are created/manipulated using statements in SQL
  - Python provides an API for working with SQL databases
    - which is normally distributed with the SQLite 3 database as standard

Database Programming

- Another kind is *Database Manager* (DBM)
  - store any number of key-value items;
  - DBMs work just like Python dictionaries, except
    - they are normally held on disk rather than in memory
    - their keys and values are always `bytes` objects

    The `shelve` module provides a convenient DBM interface
    - that allows us to use string keys and any (pickable) objects as values

Database Programming

- We will implement 2 versions of a program to manages a list of DVDs
  - The programs keep track of each DVD's
    - title
    - year of release
    - length (in minutes), and
    - director
  1. The 1st version uses a DBM via the `shelve` module
  2. The 2nd uses the SQLite database
- Both programs can also load and save a simple XML format
  - which allows, for example, to export DVD data from one program
  - and import it into the other
- The SQL-based version offers slightly more functionality
  - and has a slightly cleaner data design

Outline

The shelve module

- The `shelve` module provides a wrapper around a DBM
  - That allows us to interact with the DBM as it were a dictionary
  - As long as we use only string keys and picklable values
- `shelve` converts the keys and values to and from `bytes` objects
- `shelve` uses the best DBM available in the computer
  - it is possible that a DBM file saved on one machine won't be readable on another;
    - if the other machine doesn't have the same DBM
  - A solution we use is to provide XML import/export transportable files

Example: a database for DVDs

In our program

- We will use the DVD's titles as keys
- And tuples holding the director, year and duration as values
- Thanks to `shelve`
  - we do not need to do any data conversion, and
  - can treat the DBM object as a dictionary
- The structure of the program is similar to menu-driven programs we have seen before
  - We will focus on those aspects that are specific do DBM programming
- The program offers options to add, edit, list, remove, import and export DVD data
  - We will not cover importing and exporting data from/to XML
  - And apart from adding, we will omit most of the user interface code
- The entire solution is provided in file `dvds-dbm.py`

main

- We start by analyzing a fragment of the program's `main` function
  - with the menu handling omitted

```
db = None
try:
    db = shelve.open(filename, protocol=pickle.HIGHEST_PROTOCOL)
    ...
finally:
    if db is not None:
        db.close()
```

- We have opened (or created if it doesn't exist) the specified DBM file
  - both for reading and writing
- Each item's value is saved as a pickle using the specified protocol
- At the end, the DBM is closed
  - which has the effect of clearing the DBM's internal cache, and
  - ensuring that the disk file reflects any changes that have been made
  - as well as closing the file

Add a DVD

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    ...
```

- As all other functions, `add_dvd()` is passed the DBM object
  - as its sole parameter
- Most of the function is concerned with getting from the input the DVD's details

Add a DVD

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    db[title] = (director, year, duration)
    db.sync()
```

- In the penultimate line we store the key-value item in the DBM file
  - with the title as key, and
  - the director, year and duration (pickled together by `shelve`) as value
- The last line ensures the data is saved to the disk

DBM Databases API

- DBMs provide the same API as dictionaries
- We don't need to learn any new syntax, except
  - `shelve.open()`
    - opens a persistent dictionary producing a `Shelf` object
  - `shelve.Shelf.sync()`
    - empties the cache, and
    - synchronizes the persistent dictionary on disk;
    - is called automatically when the shelf is closed with `close()`
  - `shelve.Shelf.close()`
    - synchronizes and closes the persistent `dict` object

Edit a DVD

- To be able to edit a DVD, the user must first choose a DVD
  - this is just a matter of getting the title since titles are used as keys
  - we have factored it out into a separate `find_dvd()` function

```
def edit_dvd(db):
    old_title = find_dvd(db, "edit")
    if old_title is None:
        return
    title = Console.get_string("Title", "title", old_title)
    if not title:
        return
    director, year, duration = db[old_title]
    ...
    db[title] = (director, year, duration)
    if title != old_title:
        del db[old_title]
    db.sync()
```

- If the DVD is found, we get the user's changes
  - using the existing values as defaults to speed up interaction
- At the end, we store the data just as we did when adding
  - If the title is unchanged, this will override the associated value
  - If not, a new key-value item is created, and the old one must be deleted

Find a DVD

- To make finding a DVD as easy as possible
  - we require the user to type only some of the first characters of its title

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        matches = []
        start = Console.get_string(message, "title")
        if not start:
            return None
        for title in db:
            if title.lower().startswith(start.lower()):
                matches.append(title)
        ...
```

- Once we have the start of the title, we create a list of `matches`

Find a DVD

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        ...
        if len(matches) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(matches) == 1:
            return matches[0]
        elif len(matches) > DISPLAY_LIMIT:
            print("Too many dvds start with {0}; try entering "
                  "more of the title".format(start))
            continue
        ...
```

- If there no matches, the user needs to input more title characters
- If there is only one match, we return it
- If there are too many matches (> `DISPLAY_LIMIT`),
  - the user needs to input more title characters
  - to filter more the DVDs

Find a DVD

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        ...
        else:
            matches = sorted(matches, key=str.lower)
            for i, match in enumerate(matches):
                print("{0}: {1}".format(i + 1, match))
            which = Console.get_integer("Number (or 0 to cancel)",
                                        "number", minimum=1, maximum=len(matches))
            return matches[which - 1] if which != 0 else None
```

- If there are several matches, but fewer than DISPLAY_LIMIT
  - we display them in case-insensitive order with a number beside
  - so that the user can choose the tile just by entering its number

Listing DVDs

- Listing all DVDs, or those whose title starts with a particular substring
  - is simply a matter of iterating over the DBM's items

```
def list_dvds(db):
    start = ""
    if len(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
    print()
    for title in sorted(db, key=str.lower):
        if not start or title.lower().startswith(start.lower()):
            director, year, duration = db[title]
            print("{title} ({year}) {duration} minute{0}, by "
                  "{director}".format(Util.s(duration), **locals()))
```

- `Utils.s()` is simply `s = lambda x: "" if x == 1 else "s"`
  - so here it returns an `"s"` if the duration is not one minute (to make plural)

Removing a DVD

- Removing a DVD is a matter of finding the one the user wants to remove
  - then asking for confirmation, and if we get it
  - deleting the item from the DBM

```
def remove_dvd(db):
    title = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        del db[title]
        db.sync()
```

DBM Databases

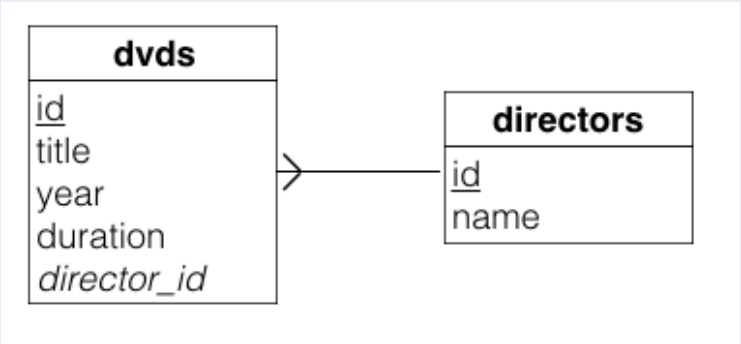
- We have now seen how to:
  - open (or create) a DBM file using the `shelve` module, and
  - to add items to it, edit its items,
  - iterate over its items, and remove items
- Still, we can improve our data design:
  - director names are duplicated in the DBM,
    - e.g., Danny DeVito may sometimes be entered as `"Danny De Vito"`, and others as `"Danny deVito"`
  - which can easily lead to inconsistencies
    - e.g., if you are searching for all the films directed by Danny DeVito
- We try to avoid this flaw in the next section's SQL database version of the program by using two tables
  - one for DVDs, and
  - another for directors

Outline

SQL Databases

- Python comes with the `sqlite3` module
- Although `sqlite3` lacks many features of, e.g., PostgreSQL
  - it is very convenient for prototyping, and
  - may prove sufficient in many cases

- The SQL version of the DVDs program is given in `dvds-sql.py`
- It stores directors separately from the DVD to avoid duplication, and
  - offers one more menu option that lets the user list the directors



- The program is similar to the one we have already seen
  - but we use SQL queries instead of dictionary-like operations
  - and we must create the database's tables the 1st time the program runs

Creating tables

- The `main()` function is similar to before
  - except this time we call a `connect()` function to connect the DB
- The `sqlite3.connect()` function returns a database object
  - having opened the database file it is given, and
  - created an empty database file if the file did not exist

```
def connect(filename):
    create = not os.path.exists(filename)
    db = sqlite3.connect(filename)
    if create:
        cursor = db.cursor()
        cursor.execute("CREATE TABLE directors ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
            "name TEXT UNIQUE NOT NULL)")
        cursor.execute("CREATE TABLE dvds ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
            "title TEXT NOT NULL, "
            "year INTEGER NOT NULL, "
            "duration INTEGER NOT NULL, "
            "director_id INTEGER NOT NULL, "
            "FOREIGN KEY (director_id) REFERENCES directors)")
        db.commit()
    return db
```

Creating tables

- We check whether the database is going to be created from scratch
  - If it is, we must create the tables the program relies on
- All queries are executed through a database `cursor()`
- Both tables are created with `id`s that have an `AUTOINCREMENT` constraint
  - SQLite will automatically populate the IDs with unique numbers

```
def connect(filename):
    create = not os.path.exists(filename)
    db = sqlite3.connect(filename)
    if create:
        cursor = db.cursor()
        cursor.execute("CREATE TABLE directors ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
            "name TEXT UNIQUE NOT NULL)")
        cursor.execute("CREATE TABLE dvds ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
            "title TEXT NOT NULL, "
            "year INTEGER NOT NULL, "
            "duration INTEGER NOT NULL, "
            "director_id INTEGER NOT NULL, "
            "FOREIGN KEY (director_id) REFERENCES directors)")
        db.commit()
    return db
```

Some SQL facts

- Most fields are of `INTEGER` or `TEXT` type
  - e.g. name, title, year, duration
- The ids are marked with `PRIMARY KEY` which allows the DB to index the row of each (unique) id
- `director_id` is a *foreign key* from table `dvds` to table `directors`
  - it means only existing director ids (from table `directors`) can be used in the DVDs' table

Add a DVD

- The `add_dvd()` function starts with the same code as `dvds-dbm.py`

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director_id = get_and_set_director(db, None)
    if director_id is None:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    cursor = db.cursor()
    cursor.execute("INSERT INTO dvds "
                   "(title, year, duration, director_id) "
                   "VALUES (?, ?, ?, ?)",
                   (title, year, duration, director_id))
    db.commit()
```

- But asking for the director is done as to avoid duplication
  - as we will see, this done by inspecting the directors table,
  - using the `get_and_set_director()` function
    - which returns the director id, which is inserted in the dvds table entry

Add a DVD

- In the query we have used question marks for placeholders
  - Each ? is replaced by the corresponding value
  - in the sequence that follows the string containing the SQL statement (similar to `format` for strings)

```
cursor.execute("INSERT INTO dvds "
               "(title, year, duration, director_id) "
               "VALUES (?, ?, ?, ?)",
               (title, year, duration, director_id))
```

Getting the director

- Function `get_and_set_director()` returns the ID of the director

```
def get_and_set_director(db, director):
    director_id = None
    cursor = db.cursor()
    cursor.execute("SELECT COUNT(*) FROM directors")
    count = cursor.fetchone()[0]
    if (count):
        list_directors_and_ids(db)
        director_id = Console.get_integer("Number (or 0 to cancel)", "director
    ...
```

- The function starts by executing an SQL query to get all directors
- If there are directors, it then list all of them
  - and asks the user if one of them is to be used
    - this is achieved relying on function `list_directors_and_ids()`

Getting the director

```
def get_and_set_director(db, director):
    ...
    if (count):
        list_directors_and_ids(db)
        director_id = Console.get_integer("Number (or 0 to cancel)", "director")
    if director_id is not None and director_id is not 0:
        return director_id
    else:
        director = Console.get_string("Director", "director", director)
        if not director:
            return
        cursor = db.cursor()
        cursor.execute("INSERT INTO directors (name) VALUES (?)", (director,))
        db.commit()
        return get_director_id(db, director)
```

- If the user does not want to use an existing direction,
  - a new director is registered and inserted in the directors table

Listing directors

- Function `list_directors_and_ids()` is quite simple:

```
def list_directors_and_ids(db):
    cursor = db.cursor()
    cursor.execute("SELECT * FROM directors")
    dirs = cursor.fetchall()
    print("Registered directors:")
    for fields in dirs:
        print("{0}: {1}".format(fields[0], fields[1]))
```

- But it is by the use of this function that we seek to avoid duplication
- The function lists all registered directors
  - Together with their associated IDs;
- This makes it easy for the user, in function `get_and_set_director()`, to
  - associate a registered director to a new dvd;
  - This association prevents the user to insert the name of a director multiple times
    - which, previously, could lead to inconsistencies!

SQL Databases

- Function `get_director_id()` is also quite simple:

```
def get_director_id(db, director):
    cursor = db.cursor()
    cursor.execute("SELECT id FROM directors WHERE name=?",
                   (director,))
    fields = cursor.fetchone()
    return fields[0] if fields is not None else None
```

- It is an auxiliary function that searches for the ID associated with a director
  - which can be used when adding (or editing) a new DVD entry
    - recall that each dvds table entry has a director ID foreign key
- We use `fetchone()` since there is either zero or one matching record
  - but the fetch methods always return a sequence of fields
    - or `None` if there are no records
  - even if we ask to retrieve only a single field

SQL Databases

- To edit a DVD record, we must first find the record the user wants:

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    year = Console.get_integer("Year", "year", year, 1896, datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:identity", locals())
    db.commit()
```

- If a record is found,
  - we begin by giving the user the opportunity to change the title

SQL Databases

- Then we retrieve the other fields so that we can provide the existing values as defaults
  - to minimize what the user must type
    - they can just press Enter to accept a default

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    year = Console.get_integer("Year", "year", year, 1896, datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:identity", locals())
    db.commit()
```

SQL Databases

- We have used named placeholders of the form `:name`
  - and must therefore provide the corresponding values using a mapping
    - for the `SELECT` statement we have used a freshly created dictionary
    - for the `UPDATE` statement we have used the `locals()` dictionary

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    year = Console.get_integer("Year", "year", year, 1896, datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:identity", locals())
    db.commit()
```

SQL Databases

- Once we have all the fields and the user has entered any changes
  - we retrieve the corresponding ID
  - and then update the database with the new data

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    year = Console.get_integer("Year", "year", year, 1896, datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:identity", locals())
    db.commit()
```

SQL Databases

- The `find_dvd()` function returns a 2-tuple
  - (title, DVD ID) or (`None`, `None`)
  - depending on whether a record was found

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "title")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title", (start + "%",))
        records = cursor.fetchall()
        if len(records) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Too many dvds ({0}) start with {1}; try entering "
                  "more of the title".format(len(records), start))
            continue
        else:
            for i, record in enumerate(records):
                print("{0}: {1}".format(i + 1, record[0]))
            which = Console.get_integer("Number (or 0 to cancel)",
                        "number", minimum=1, maximum=len(records))
            return records[which - 1] if which != 0 else (None, None)
```

SQL Databases

- Instead of iterating over all data, we use the SQL wildcard operator %
  - so only the relevant records are retrieved

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "title")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title", (start + "%",))
        records = cursor.fetchall()
        if len(records) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Too many dvds ({0}) start with {1}; try entering "
                  "more of the title".format(len(records), start))
            continue
        else:
            for i, record in enumerate(records):
                print("{0}: {1}".format(i + 1, record[0]))
            which = Console.get_integer("Number (or 0 to cancel)",
                        "number", minimum=1, maximum=len(records))
            return records[which - 1] if which != 0 else (None, None)
```

SQL Databases

- Since we expect the number of matching records to be small
  - we fetch them all at once into a sequence (of sequences)
- If there is more than one match and few enough to display
  - we print the records with a number beside each one
    - so that the user can choose the one they want

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "title")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title", (start + "%",))
        records = cursor.fetchall()
        if len(records) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Too many dvds ({0}) start with {1}; try entering "
                  "more of the title".format(len(records), start))
            continue
        else:
            for i, record in enumerate(records):
                print("{0}: {1}".format(i + 1, record[0]))
            which = Console.get_integer("Number (or 0 to cancel)",
```

SQL Databases

- To list the details of each DVD we do a `SELECT` that joins the tables
  - adding a second element to the `WHERE` clause
    - if there are more records than the display limit
- We then execute the query and iterate over the results
- Each record is a sequence whose fields are those matching the `SELECT` query

```
def list_dvds(db):
    cursor = db.cursor()
    sql = ("SELECT dvds.title, dvds.year, dvds.duration, "
           "directors.name FROM dvds, directors "
           "WHERE dvds.director_id = directors.id")
    start = None
    if dvd_count(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
        sql += " AND dvds.title LIKE ?"
    sql += " ORDER BY dvds.title"
    print()
    if start is None:
        cursor.execute(sql)
    else:
        cursor.execute(sql, (start + "%",))
    for record in cursor:
        print("{0[0]} ({0[1]}) {0[2]} minutes, by {0[3]}".format(record))
```

SQL Databases

- Function `dvd_count()` is used in several different functions
  - and simply counts how many DVDs records exist in the dvds table

```
def dvd_count(db):
    cursor = db.cursor()
    cursor.execute("SELECT COUNT(*) FROM dvds")
    return cursor.fetchone()[0]
```

- Finally, function `remove_dvd()` is called when the user wants to delete a record

```
def remove_dvd(db):
    title, identity = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        cursor = db.cursor()
        cursor.execute("DELETE FROM dvds WHERE id=?", (identity,))
        db.commit()
```