Persistence with Files in Python

Roberto A. Bittencourt

Files in Text Mode

Example: Writing into a file

```
# open file for writing
file1 = open('test.txt', 'w')

# write text into a file
file1.write('Testing writing.')

# close file
file1.close()
```

Example: copying a file into another file

```
reading = open('arq1.txt', 'r')
writing = open('arq2.txt', 'w')

text = reading.read()
writing.write(text)

reading.close()
writing.close()
```

Example: copying a list of numbers into a file

```
lst = []
n = int(input('Enter the list size: '))
print ('Type the list elements: ')
for i in range(n):
    element = int(input())
    lst.append(element)
file1 = open('list1.txt', 'w')
file1.write('%d\n' % n)
for i in range(n):
    file1.write('%d\n' % lst[i])
file1.close()
```

Text file commands in Python

- Text Files in Python
  - Object from **`file`** type represents a file
- Which file will be opened?
  - **`open`** function opens a file object, given path + filename and opening mode

```
file1 = open('mytext.txt','w')

file2 = open('c:/doc/file.c','r')
```

File modes

Opening files:

```
fileobj = open(filename, mode)
```

Opening modes:

- **`r`** : read-only
- **`w`** : write-only
- **`a`** : read/write, stream at end (append)
- **`r+`** : read/write, stream at start
- **`w+`** : read/write, always creates a new file

File commands in Python

- `fileobj = open(filename, mode)`
  - Opens file
- `fileobj.write(str)`
  - Similar to *print()*, one parameter only (must be string), does not add newline at the end
- `fileobj.read()`
  - similar to *input()*, reads and returns a string with all the text in the file
- `fileobj.readline()`
  - similar to *read()*, but reads just one line from the file and returns a string with that line
- `fileobj.close()`
  - Closes file

The `with` command

- The **`with`** command simplifies file handling
  - The **`with`** blocks starts with file opening
  - File is closed when the **`with`** block ends

Example:

```
with open('test.txt', 'w') as file:
    file.write('Testing writing.')
```

Example: copying a list of numbers into a file using the **with** command

```
lst = []
n = input('Type list size: ')
print ('Type list elements: ')
for i in range(n):
    element = int(input())
    lst.append(element)
with open('list1.txt', 'w') as file:
    file.write('%d\n' % n)
    for i in range(n):
      file.write('%d\n' % lst[i])
```

Example: reading a list of numbers from a file using the `with` command

```
lst = []
with open('list1.txt', 'r') as file:
    n = int(file.readline())
    for i in range(n):
        element = int(file.readline())
        lst.append(element)
print('Read %d elements:' % n)
for i in range(n):
    print(lst[i])
```

Example: reading a file whose number of lines is unknown

```
# not the best solution
str = input('Type file name: ')
with open(str, 'r') as file:
    line  = file.readline()
    while(line != '')
        print(line)
        line = file.readline()

# this is a better solution
str = input('Type file name: ')
with open(str, 'r') as file:
    for line in file:
        print(line)
```

Files in Binary Mode

Files in binary mode

- Oriented to bytes instead of characters
- File opening

```
fileobj = open(filename, mode)
```

- Opening modes (**b** : indicates binary mode):
  - **rb** : read-only in binary mode
  - **wb** : write-only in binary mode
  - **ab** : read/write in binary mode, stream at end (append)
  - **rb+** : read/write in binary mode, stream at start
  - **wb+** : read/write in binary mode, always creates a new file
- Example:

```
file = open('data.dat', 'wb')
```

pickle Library

- To deal with complex data in Python, there are various alternatives (e.g., pickle, JSON, marshal)
  - One of the simplest solutions is the **pickle** library
- File write command in pickle:

```
dump(object, fileobj)
```

  - Writes the data stored in `object` into the `fileobj` file.
  - `object` may be either a primitive type or a complex type (e.g., list, dictionary, tuple, object)
- File read command in pickle:

```
load(fileobj)
```

  - Reads from `fileobj` the object that is stored in the associated file and returns it

Example: a list

```
from pickle import dump, load

list1 = [7, 40, 12, 0, -5]
print('List:', list1)

with open('file01.dat', 'wb') as file:
  dump(list1, file)

with open('file01.dat', 'rb') as file2:
  list2 = load(file2)

print('Retrieved list:', list2)
```

Example: two lists in sequence

```
from pickle import dump, load
list1 = [7, 40, 12, 0, -5]
list2 = ['house', 'car', 'bike']
print('List 1:', list1)
print('List 2:', list2)

with open('file02.dat', 'wb') as file:
  dump(list1, file)
  dump(list2, file)

with open('file02.dat', 'rb') as file2:
  list3 = load(file2)
  list4 = load(file2)

print('Retrieved list 1:', list3)
print('Retrieved list 2:', list4)
```

Example: two objects in sequence (1)

```
from pickle import dump, load

class Student:
    def __init__(self, name, number, birth_year):
        self.name = name
        self. number = number
        self. birth_year = birth_year

student1 = Student('Marcos Santos',19211175, 1997)
student2 = Student('Mary Shaw',18111130, 1996)
print('Student 1 - Name: %s, Number: %d, Birth Year:
  %d' % (student1.name, student1.number,
  student1.birth_year))
print('Student 2 - Name: %s, Number: %d, Birth Year:
  %d' % (student2.name, student2.number,
  student2.birth_year))

# continues in the next slide
```

Example: two objects in sequence (2)

```
# continuing the example

with open('file03.dat', 'wb') as file:
  dump(student1, file)
  dump(student2, file)

with open('file03.dat', 'rb') as file2:
  student3 = load(file2)
  student4 = load(file2)

print('Retrieve student 1 - Name: %s, Number: %d,
  Birth Year: %d' % (student3.name, student3.number,
  student3.birth_year))
print('Retrieved student 2 - Name: %s, Number: %d,
  Birth Year: %d' % (student4.name, student4.number,
  student4.birth_year))
```

Example: a 5x3 matrix

```
from pickle import dump, load
grades = []
for i in range(5):
    row = []
    for j in range(3):
        print('For student %d, type the grade %d: '
            % (i+1,j+1))
        grade = float(input())
        row.append(grade)
    grades.append(row)
with open('file04.dat', 'wb') as file:
  dump(grades, file)
print('File saved!')
with open('file04.dat', 'rb') as file2:
  grades2 = load(file2)
print('File loaded!')
print('\nValues read:\n')
for i in range(5):
    for j in range(3):
        print('%5.1f' % grades2[i][j], end = '')
    print()
```

Example: a list, one element at a time

```
from pickle import dump, load
list1 = []
n = int(input("Choose the number of elements: "))
for i in range(n):
    number=int(input("Type the element %d: " % (i + 1)))
    list1.append(number)
# one can save a list one element at a time
with open('file05.dat', 'wb') as file:
  for i in range(n):
      dump(list1[i], file)
print('File saved!')
list2 = []
# later, one can read one element at a time
with open('file05.dat', 'rb') as file2:
  for i in range(n):
    element = load(file2)
    list2.append(element)
print('File loaded!')
print('Retrieved list: ')
for i in range(n):
    print(list2[i])
```

Example: retrieving a list whose size is unknown

```
from pickle import load

# when one does not know how many elements the list has,
# EOFError can be used – exception raised when
# the end of file is reached
lista = []
with open('file05.dat', 'rb') as file:
  while True:
    try:
        element = load(file)
        lista.append(element)
    except EOFError:
        break

print('File loaded!')

print('Retrieved list: ')
for element in lista:
    print(element)
```

Example: a list of objects

```
from pickle import dump, load
class Student:
    def __init__(self, name, number, birth_year):
        self.name = name
        self.number = number
        self.birth_year = birth_year
students = []
n = int(input("Type the number of students: "))
for i in range(n):
    name = input('Name: ')
    number = int(input('Number: '))
    birth_year = int(input('Birth year: '))
    student = Student(name, number, birth_year)
    students.append(student)
with open('file07.dat', 'wb') as file:
  dump(students, file)
print('File saved!')
with open('file07.dat', 'rb') as file2:
  students2 = load(file2)
print('File loaded!')
print('Retrieved list of students: ')
for student in students2:
    print('Name: %s, Number: %d, Birth Year: %d' %
      (student.name, student.number, student.birth_year))
```

Example: saving a list of objects, writing one object at a time

```
from pickle import dump
class Student:
    def __init__(self, name, number, birth_year):
        self.name = name
        self.number = number
        self.birth_year = birth_year
# If the data input process is slow,
# one can save the elements one at a time.
students = []
n = int(input("Type the number of students to save: "))
for i in range(n):
    name = input('Name: ')
    number = int(input('Number: '))
    birth_year = int(input('Birth Year: '))
    student = Student(name, number, birth_year)
    students.append(student)
    # saving one student at a time
    with open('file08.dat', 'ab') as file:
      dump(student, file)
print('Students saved in the file!')
```

Example: loading a list of objects, reading one object at a time

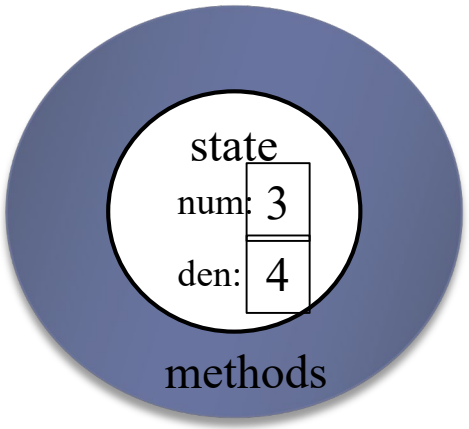
```
from pickle import load
class Student:
    def __init__(self, name, number, birth_year):
        self.name = name
        self.number = number
        self.birth_year = birth_year
# Loading the students that were saved from another list
students2 = []
with open('file08.dat', 'rb') as file2:
  while True:
    try:
        student = load(file2)
        students2.append(student)
    except EOFError:
        break
print('File loaded!')

print('Retrieved list of students: ')
for student in students2:
    print('Name: %s, Number: %d, Birth Year: %d' %
      (student.name, student.number, student.birth_year))
```

JSON

Question?

- Given a particular set of data, how do you store it permanently?
  - What do you store on disk?
  - What format?
  - Can you easily transmit over the web?
  - Will it be readable by other languages?
  - Can humans read the data?

- Examples:
  - A square
  - A dictionary



Storage using plain text

- Advantages
  - Human readable (good for debugging / manual editing)
  - Portable to different platforms
  - Easy to transmit using web
- Disadvantages
  - Takes more memory than needed
- Use a standardized format – JSON
  - Makes the information more portable

JavaScript Object Notation

- Text-based notation for data interchange
  - Human readable

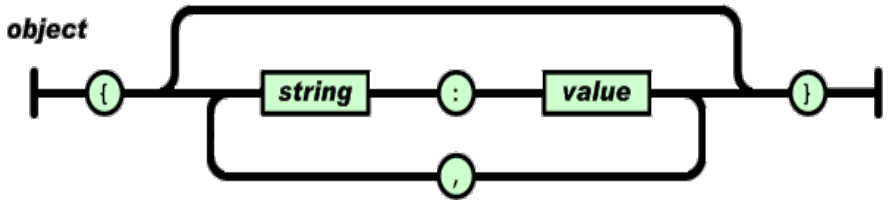
- Object
  - Unordered set of name-value pairs
  - names must be strings
  - `{ name1 : value1, name2 : value2, …, nameN : valueN }`

- Array
  - Ordered list of values
  - [value1, value2, … valueN]

JSON Data – A name and a value

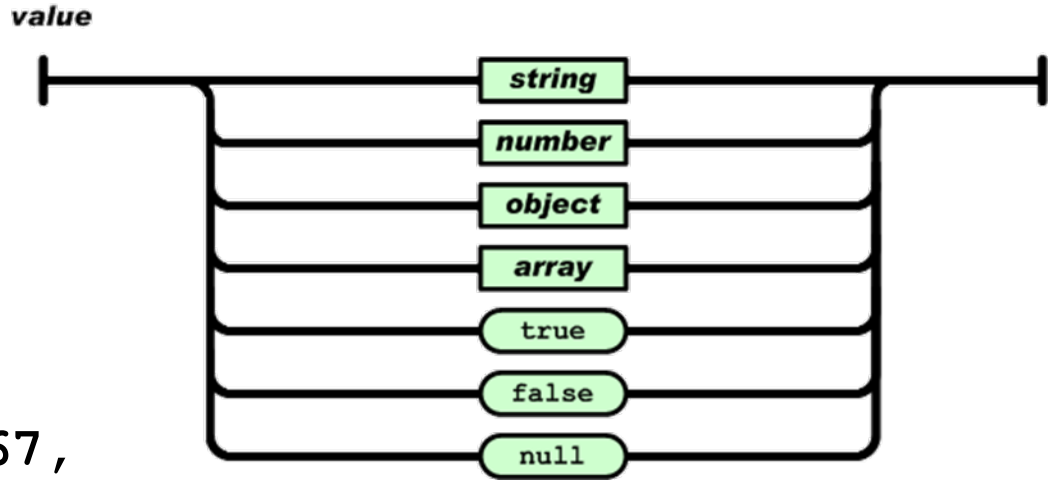
- A name/value pair consists of:
  - a field name (in **double quotes**), followed by a colon, followed by a value
- Unordered sets of name/value pairs
- Begins with **{** (left brace)
- Ends with **}** (right brace)
- Each name is followed by **:** (colon)
- Name/value pairs are separated by **,** (comma)

```
{
   "employee_id": 1234567,
   "name": "Jeff Fox",
   "hire_date": "1/1/2013",
   "location": "Norwalk, CT",
   "consultant": false
}
```



JSON Data – A name and a value

- In JSON, *values* must be one of the following data types:
  - a string
  - a number
  - an object (JSON object)
  - an array
  - a boolean
  - null




```
{
   "employee_id": 1234567,
   "name": "Jeff Fox",
   "hire_date": "1/1/2013",
   "location": "Norwalk, CT",
   "consultant": false
}
```

JSON Data – A name and a value

- Strings in JSON must be written in double quotes.

```
{ "name":"John" }
```

- Numbers in JSON must be an integer or a floating point.

```
{ "age":30 }
```

- Values in JSON can be objects.

```
{
  "employee":{ "name":"John", "age":30, "city":"New York" }
}
```

- Values in JSON can be arrays.

```
{
  "employees":[ "John", "Anna", "Peter" ]
}
```

JSON basics in Python

Using JSON with Python

- To work with JSON (string or file containing JSON objects), you can use Python's JSON module.

```
import json
```

Loading JSON data from a file

- Example:

```
def load_json(filename):
    with open(filename, 'r') as file:
        jsn = json.load(file)
        #file.close()
    return jsn
person = load_json('person.json')
```

- This function above parses the **`person.json`** using **`json.load()`** method from the **`json`** module.
  - The result is a Python dictionary

Writing a JSON object into a file

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}

with open('person_to_json.json', 'w') as fp:
    json.dump(person, fp, indent=4)
```

- Using **`json.dump()`**, we can convert Python Objects into a JSON file.

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your **person.json** as follows.

```
person = load_json('person.json')
```

To access the "name" property:

```
print(person["name"])
John Smith
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your **person.json** as follows.

```
person = load_json('person.json')
```

To access the "age" property:

```
person["age"]
35
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your **person.json** as follows.

```
person = load_json('person.json')
```

To access the "street" property inside "address":

```
print(person["address"]["street"])
5 Main St.
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your **person.json** as follows.

```
person = load_json('person.json')
```

To access the "city" property inside "address":

```
print(person["address"]["city"])
Austin
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your **person.json** as follows.

```
person = load_json('person.json')
```

To access the element at index 0 from the "children" property:

```
print(person["children"][0])
Mary
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your **person.json** as follows.

```
person = load_json('person.json')
```

To access the element at index 1 from the "children" property:

```
print(person["children"][1])
Abel
```

Python – JSON Objects

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

More JSON File Handling in Python

Example01.py

Writing JSON using Python

- **`json.dumps( data )`**
  - Accepts Python object as an argument
  - Returns a string containing the information in JSON format
  - One typically write this string into a file
  - This operation is usually called serialization

```
def write(data, filename):
    file = open(filename, 'w')
    str_out = json.dumps(data)
    file.write(str_out)
    file.close()
```

Example01.py

Reading JSON using Python

- **`json.loads( data )`**
  - Accepts string as an argument
  - The string should be in JSON format
  - Returns a Python object corresponding to the data
  - This operation is usually called deserialization

Double quotes

```
"Hello World"
```

'hello.txt'

```
def read(filename):
    file = open(filename)
    str_in = file.read()
    file.close()
    data = json.loads(str_in)
    return data
```

```
write('Hello World', 'hello.txt')
print(read('hello.txt'))
```

Example02.py

Example 2: Writing a dictionary

- Create a dictionary

```
my_dict = {'Angela':'86620','adriana':'87113, 'ann':'84947'}
file_name = 'test_dict.txt'
write(my_dict, file_name)
```

```
{"ann": "84947", "adriana": "87113", "Angela": "86620"}
```

```
print(read(file_name))
```

Example03.py

Writing JSON using pretty printing

- **`json.dumps( data )`**

A dictionary

```
{'b': ['HELLO', 'WORLD'], 'a': ['hello', 'world']}
```

- **`json.dumps`**`( data, indent=4, sort_keys=True )`
  - Formats the output over multiple lines

```
{
    "a": [
        "hello",
        "world"
    ],
    "b": [
        "HELLO",
        "WORLD"
    ]
}
```

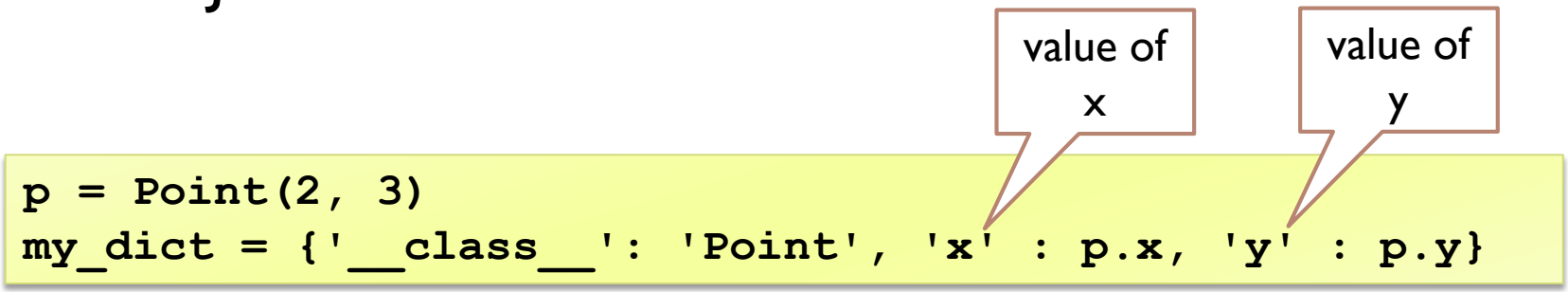
Double quotes

What about user-defined classes?

- Point class

```
class Point:
    def __init__(self, loc_x, loc_y):
        self.x = loc_x
        self.y = loc_y

    def __str__(self):
        return str(self.x) + ',' + str(self.y)
```

- If you can create a dictionary to store state information, then use JSON



What about user-defined classes?

- One can use JSON to read and extract the state information

```
file_name = 'test_point.txt'
write(my_dict, file_name)
```

```
{
    "__class__": "Point",
    "x": 2,
    "y": 3
}
```

- Example:

```
data = read(file_name)
result = Point( data['x'], data['y'] )
print (result)
```

JSON Encoding and Decoding

- Of course, manually creating dictionaries from objects and vice-versa is time-consuming and error-prone

- We may fix that by asking the **`json`** library to encode and decode objects through extending the classes:
  - **`json.JSONEncoder`**
  - **`json.JSONDecoder`**
- Then we call the **`encode()`** and **`decode()`** methods from the extended classes
- Let us look at an example in the following

Let us work with the Car class…

- Example:

```
class Car:
  def __init__(self, make, model, year, price):
    self.make = make
    self.model = model
    self.year = year
    self.price = price
```

- To create a new Car object, we can simply call the Car constructor with the appropriate arguments.

```
car = Car("Toyota", "Camry", 2022, 25000)
```

- If we try to serialize the Car object as-is, we will get a TypeError:

```
car_str = json.dumps(car)
TypeError: Object of type 'Car' is not JSON
serializable
```

Encoding the Car class…

- CarEncoder extends the JSONEncoder class:

```
class CarEncoder(json.JSONEncoder):
  def default(self, obj):
    if isinstance(obj, Car):
      return {"__type__": "Car", "make": obj.make, "model": \
       obj.model, "year": obj.year, "price": obj.price}
    return super().default(obj)
```

- Now we can get any Car object, encode it and save it.

```
car = Car("Toyota", "Camry", 2022, 25000)
car_json = json.dumps(car, cls=CarEncoder)
file.write(car_json)
print(car_json)
{"__type__": "Car", "make": "Toyota", "model": "Camry",
"year": 2022, "price": 25000}
```

Decoding the Car class…

- CarDecoder extends the JSONDecoder class:

```
class CarDecoder(json.JSONDecoder):
  def __init__(self, *args, **kwargs):
    super().__init__(object_hook=self.object_hook, *args, **kwargs)

  def object_hook(self, dct):
    if '__type__' in dct and dct['__type__'] == 'Car':
      return Car(dct['make'], dct['model'], dct['year'], dct['price'])
    return dct
```

- Now we can load any JSON dictionary representing a Car, decode it and create the Car object.

```
car_json = '{"__type__": "Car", "make": "Toyota", "model": "Camry",
"year": 2022, "price": 25000}'  # or use car_json = file.read()
car = json.loads(car_json, cls=CarDecoder)
print(car.make)   # Output: "Toyota"
print(car.model)  # Output: "Camry"
print(car.year)   # Output: 2022
print(car.price)  # Output: 25000
```

DAO Pattern

Data Access Object (DAO)

- The Data Access Object (DAO) pattern is a structural design pattern that provides an abstract interface to some type of database or other persistence mechanism.
- By mapping application calls to the persistence layer, DAOs facilitate the separation of business logic from database operations.
- DAO is also known as:
  - Data Mapper
  - Repository Pattern (closely related)

DAO Goals

- The goals of the DAO pattern are to:
  - Separate database access logic from business logic.
  - Encapsulate the access to data sources.
  - Provide a uniform interface to access data from different sources.

DAO Pros and Cons

- **Key Features:**
  - **Encapsulation**: Hides the details of data access logic.
  - **Abstraction**: Provides an abstract interface to different types of data sources.
  - **Decoupling**: Decouples business logic from data access logic.
  - **Maintainability**: Makes the code easier to maintain and test
- **Trade-offs**
  - **Complexity**: Adds an extra layer of abstraction which could increase the complexity of the project.
  - **Performance**: Overhead of additional method calls and object creation.

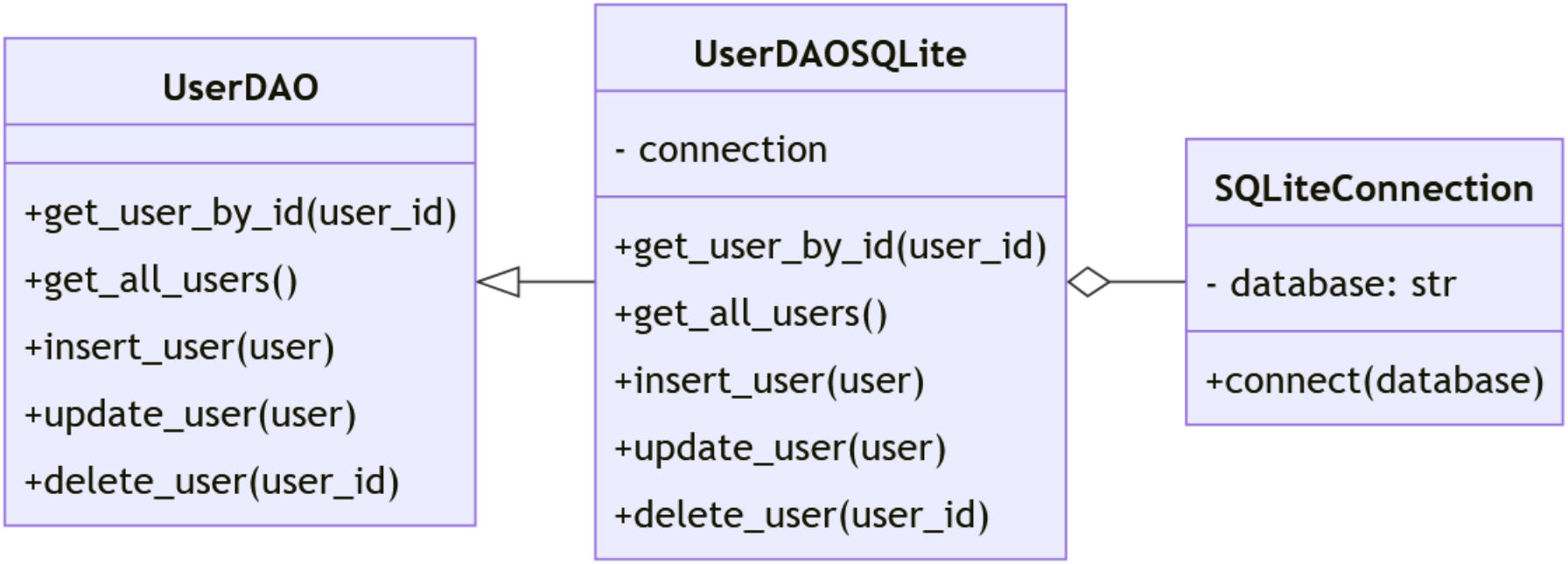
How it works

- The DAO pattern abstracts and encapsulates all access to the data source.
- The underlying data source could be a database, file system, or any other persistence mechanism.
- This abstraction allows for flexibility and makes the application easier to maintain and test.

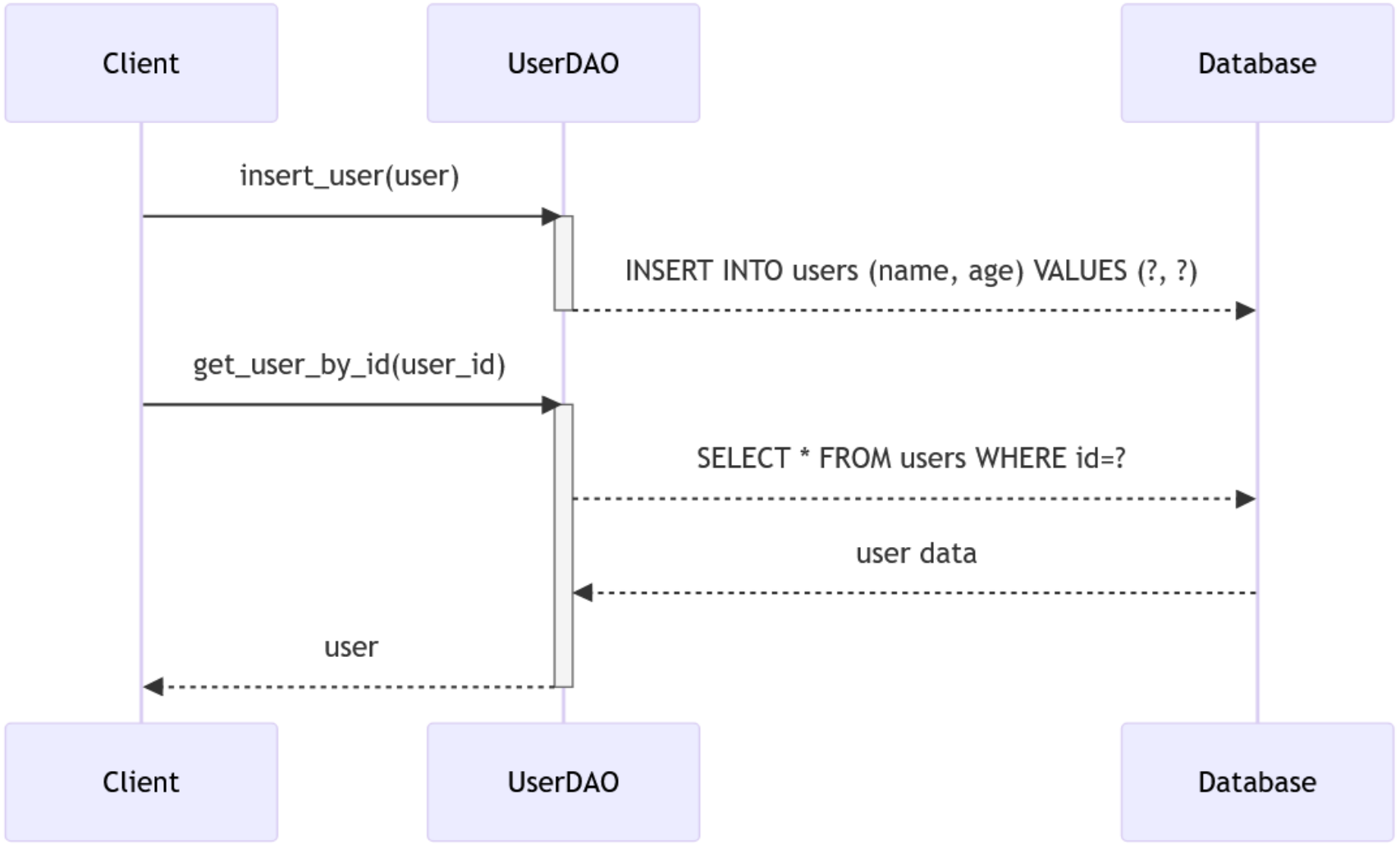
Components of a DAO PAttern

- The pattern typically involves the following components:
  - **DAO Interface**: Defines the standard operations to be performed on a model object(s).
  - **DAO Implementation**: Implements the concrete operations defined in the DAO interface.
  - **Model Object**: Represents the data being accessed.
  - **Client**: Utilizes the DAO interface for performing CRUD operations.

DAO Structure



DAO Behavior



Code: Model Object

```
class User:
    def __init__(self, user_id, name, age):
        self.id = user_id
        self.name = name
        self.age = age
```

Code: DAO Interface

```
from abc import ABC, abstractmethod
class UserDAO(ABC):
    @abstractmethod
    def get_user_by_id(self, key):
        pass

    @abstractmethod
    def get_all_users(self):
        pass

    @abstractmethod
    def insert_user(self, user):
        pass

    @abstractmethod
    def update_user(self, user):
        pass

    @abstractmethod
    def delete_user(self, user_id):
        pass
```

Code: DAO Implementation for databases

```
class UserDAOSQLite(UserDAO):
    def __init__(self, database):
        self.connection = sqlite3.connect(database)
    def get_user_by_id(self, user_id):
        cursor = self.connection.cursor()
        cursor.execute('SELECT * FROM users WHERE id=?', (user_id,))
        return cursor.fetchone()
    def get_all_users(self):
        cursor = self.connection.cursor()
        cursor.execute('SELECT * FROM users')
        return cursor.fetchall()
    def insert_user(self, user):
        cursor = self.connection.cursor()
        cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)',
         (user.name, user.age))
        self.connection.commit()
    def update_user(self, user):
        cursor = self.connection.cursor()
        cursor.execute('UPDATE users SET name=?, age=? WHERE id=?',
         (user.name, user.age, user.id))
        self.connection.commit()
    def delete_user(self, user_id):
        cursor = self.connection.cursor()
        cursor.execute('DELETE FROM users WHERE id=?', (user_id,))
        self.connection.commit()
```

Code: Client Code

```
def main():
    user_dao = UserDAOSQLite('users.db')

    new_user = User(None, 'John Doe', 30)
    user_dao.insert_user(new_user)

    user = user_dao.get_user_by_id(1)
    print(user)

    all_users = user_dao.get_all_users()
    print(all_users)


if __name__ == "__main__":
    main()
```

DAO Pattern Implementation with Python File Libraries

Some design decisions with our example model scenario…

- Reusing the example classes from our exemple model:
  - School and Student
- Recalling that School has a collections of students
  - We will need access to this collection in the DAO class
- Student will be our model class
  - We will then create StudentDAO
- Using a single file to store the collection of students
- Testing our system for persistence by changing the tests for our School class

Recalling the **Student** model class...

```
from datetime import date
class Student:
  def __init__(self,name,number,birth_year):
    self.name = name
    # number will be the key
    self.number = number
    self.birth_year = birth_year

  def age(self):
    currentYear = date.today().year
    return currentYear - self. birth_year

  def __eq__(self, other):
    return self.name == other.name and self.number  == \
     other.number and self.birth_year == other.birth_year

  def __str__(self):
    return "name: %s, number: %d, birth year: %d" % \
     (self.name, self.number, self.birth_year)
```

Recalling the **School** model class that works as a container of students...

```
class School:
    def __init__(self, name):
        self.name = name
        self.students = []

    def search(self, key):
        # code returns a Student
    def create(self, name, number, birth_year):
        # code returns a boolean
    def retrieve(self, name):
        # code returns a list of students
    def update(self, key, name, number, birth_year):
        # code returns a boolean
    def delete(self, key):
        # code returns a boolean
    def list_all(self):
        # code returns a list of students
```

StudentDAO Interface

```
from abc import ABC, abstractmethod
class StudentDAO(ABC):
    @abstractmethod
    def search_student(self, key):
        pass
    @abstractmethod
    def create_student(self, student):
        pass
    @abstractmethod
    def retrieve_students(self, name):
        pass
    @abstractmethod
    def update_student(self, key, student):
        pass
    @abstractmethod
    def delete_student(self, key):
        pass
    @abstractmethod
    def list_all_students(self):
        pass
```

How to handle collections in this new scenarios with persistence?

- A simple solution would be not using DAO. Keep the CRUD methods in School just the way they are and:
  - Inside the School constructor, load the students' file into the student collection.
  - For each operation that changes data (create, update and delete), save the full student collection into the students' file
- The above solution is simple but is also bad
  - If we decide to change persistence technologies (e.g., using relational databases or using cloud storage), changes would affect much of the School code
- Our solution using DAO is simple:
  - Put both the student collection and file access inside the DAO

How would the DAO use affect School? Let us refactor the constructor first

```
class School:

  def __init__(self, name):
    self.name = name
    # the DAO below will depend on the technology
    # replace ConcreteStudentDAO with the concrete
    # DAO you decided to use according to chosen tech
    # (e.g, StudentDAOSQLite, StudentDAOPickle,
    #  StudentDAOJSON, StudentDAOCloudStorage)
    self.student_dao = ConcreteStudentDAO()
```

Let us also refactor the School methods to delegate the work to the DAO...

```
# ... Continuing the School class

def search(self, key):
    return self.student_dao.search_student(key)
def create(self, name, number, birth_year):
    student = Student(name, number, birth_year)
    return self.student_dao.create_student(student)
def retrieve(self, name):
    return self.student_dao.retrieve_students(name)
def update(self, key, name, number, birth_year):
    student = Student(name, number, birth_year)
    return self.student_dao.update_student(key, student)
def delete(self, key):
    return self.student_dao.search_student(key)
def list_all(self):
    return self.student_dao.list_all_students()
```

Now, let us create the concrete DAO class

- Your concrete DAO class will have:
  - The student collection
  - Access to the persistence mechanism
- Your DAO class will operate on:
  - Accessing the student collection (reuse and change the existing CRUD code from School)
  - Accessing the persistence mechanism (e.g, loading, saving)
- In the next slides, we will create concrete DAO classes for the Python **`pickle`** library

StudentDAO with the pickle library

Will the also DAO affect testing?

- Now your testing code will handle two different issues:
  - Testing whether the collection is working correctly
  - Testing whether the persistence is also working correctly
- Do you want your tests to handle those issues together or separately?
- Let us deal with them separately by using a system property called **autosave**
  - By default, autosave is off: your tests check collections only
  - If you turn **autosave** on: tests check both collections and persistence
  - You will need to retrieve the **autosave** value along your system when you build objects that may persist
    - A simple way to recover autosave is by using class variables in a Configuration class
- Obs.: if you are dealing with a database or with cloud storage, you will only need to handle persistence
  - The other issues are part of those outside system's internals

Let us first create a Configuration class to hold information about persistence...

```
class Configuration:
    # Note the lack of a constructor
    # Also note below the class variables ("static")
    # autosave is initially false, but integration tests
    # may change that in the setUp()
    autosave = False
    filename = 'students.dat'
```

Constructor for StudentDAOPickle

```
from pickle import load, dump
from student_dao import StudentDAO

class StudentDAOPickle(StudentDAO):

    def __init__(self):
        conf = Configuration()
        self.autosave = conf.__class__.autosave
        self.filename = conf.__class__.filename
        if self.autosave:
            try:
                with open(self.filename, 'rb') as file:
                    self.students = load(file)
            except FileNotFoundError:
                self.students = []
        else:
            self.students = []
```

CRUD methods for StudentDAOPickle (1)

```
# continuing StudentDAOPickle (1)

    def search_student(self, key):
        # unordered list requests a linear search
        for element in self.students:
            if (element.number == key):
                return element
        return None
    def create_student(self, student):
        if not self.search_student(student.number):
            self.students.append(student)
            # save file after creating student
            if self.autosave:
              with open(self.filename, 'wb') as file:
                  dump(self.students, file)
            return True
        else:
            return False
    def retrieve_students(self, name):
        retrieved = []
        for element in self.students:
            if name in element.name:
                retrieved.append(element)
        return retrieved
```

CRUD methods for StudentDAOPickle (2)

```
# continuing StudentDAOPickle (2)
    def update_student(self, key, student):
        existing_student = self.search_student(key)
        if existing_student:
            existing_student = student
            # save file after updating student
            if self.autosave:
              with open(self.filename, 'wb') as file:
                  dump(self.students, file)
            return True
        else:
            return False
    def delete_student(self, key):
        element = self.search_student(key)
        if element:
            self.students.remove(element)
            # save file after deleting student
            if self.autosave:
              with open(self.filename, 'wb') as file:
                  dump(self.students, file)
            return True
        else:
            return False
    def list_all_students(self):
        students_list = []
        for student in self.students:
            students_list.append(student)
        return students_list
```

Persistence Testing

Persistence Testing

- Adding persistence to a system complicates testing because files/databases end up saving state that can disturb the test cases that come after another test case
- A simple approach is the following:
  1. Start your testing with no files/database (or an empty file/database)
  2. Set up the test fixture to create the class that contains the DAO
  3. Exercise the test case that may save into file/database
  4. Tear down the test fixture by deleting/empyting the file/database
  5. The unit test framework will repeat steps 2, 3 and 4 for each test method, which will now be consistent

Recalling the order of your testing process

- When you run a SchoolTest test case:
  - First, in setUp(), it constructs a School object
  - Then, the School's constructor creates the StudentDAO object
  - The StudentDAO's constructor recovers information about the autosave property and file directories and names
    - If autosave is true, it reads the files to initialize the StudentDAO with the saved collection of students
  - Then, SchoolTest exercises school operations and assertions
  - Finally, in tearDown(), SchoolTest cleans up the used files so that the next test case starts from a clean environment

PersistenceSchoolTest (1 of 6) (setUp and tearDown)

```
import os
from unittest import TestCase
from unittest import main
from student import Student
from school import School

class SchoolTest(TestCase):

    def setUp(self):
        # set autosave to True to test persistence
        conf = Configuration()
        self.conf.__class__.autosave = True
        self.filename = conf.__class__.filename
        self.school = School('Central High')
        self.school.student_dao.filename = 'students.dat'

    def tearDown(self):
        if os.path.exists(self.school.student_dao.filename):
            os.remove(self.filename)
```


...

PersistenceSchoolTest (2 of 6)

```
# continuing SchoolTest (2)
    def test_create_search(self):
          student1 = Student('Peter', 123, 2010)
          student1a = Student('Peter', 123, 2010)
          student2 = Student('Ali', 234, 2011)
          student3 = Student('Kala', 345, 2009)

          # add new student
          self.assertTrue(self.school.create('Peter', 123, 2010), "creating student1
    should return success")
          self.assertIsNotNone(self.school.search(123), "student1 is registed at
    school")
          self.assertEqual(student1a, self.school.search(123),
                   "registered student1 data should be the same as student1a")

          # add more students
          self.assertTrue(self.school.create('Ali', 234, 2011), "creating student2
    should return success")
          self.assertTrue(self.school.create('Kala', 345, 2009), "creating student3
    should return success")
          self.assertEqual(student1, self.school.search(123), "student1 remains
    registered")
          self.assertEqual(student2, self.school.search(234), "student2 is
    registered")
          self.assertEqual(student3, self.school.search(345), "student3 is
    registered")
```

PersistenceSchoolTest (3 of 6)

```
# continuing SchoolTest (3)
    def test_retrieve(self):
        student1 = Student('Peter Jackson', 123, 2010)
        student2 = Student('Peter Parker', 234, 2011)
        student3 = Student('Kala', 345, 2009)

        self.assertEqual(0, len(self.school.retrieve('Peter')), "empty student
    collection")

        # add some students
        self.school.create('Peter Jackson', 123, 2010)
        self.school.create('Peter Parker', 234, 2011)
        self.school.create('Kala', 345, 2009)

        # Retrieve a singleton list
        retrieved = self.school.retrieve('Kala')
        self.assertEqual(1, len(retrieved),
                "there should be 1 student named Kala")
        self.assertEqual(student3, retrieved[0], "Retrieving Kala")

        # Retrieve a list with more than one element
        retrieved = self.school.retrieve('Peter')
        self.assertEqual(2, len(retrieved),
                "there should be 2 students named Peter")
        self.assertEqual(student1, retrieved[0], "Retrieving Peter Jackson")
        self.assertEqual(student1, retrieved[0], "Retrieving Peter Parker")
```

PersistenceSchoolTest (4 of 6)

```
# continuing SchoolTest (4)
    def test_update(self):
          student1 = Student('Peter', 123, 2010)
          student1a = Student('Peter Jackson', 123, 1995)
          student2 = Student('Ali', 234, 2011)
          student2a = Student('Ali Mesbah', 102, 2011)
          student3 = Student('Kala', 345, 2009)

          self.assertFalse(self.school.update(123, 'Peter Jackson', 123, 1995),
    "empty student collection")

          # add some students
          self.school.create('Peter', 123, 2010)
          self.school.create('Ali', 234, 2011)
          self.school.create('Kala', 345, 2009)

          # Do not update an unregistered student
          self.assertFalse(self.school.update(300, 'Latifa', 300, 2001),
    "Unregistered student")

          # Update a registered student, maintaning key
          self.assertTrue(self.school.update(123, 'Peter Jackson', 123, 1995),
    "Updating Peter, same key")

          # Update a registered student, changing key
          self.assertTrue(self.school.update(234, 'Ali Mesbah', 102, 1998),
    "Updating Ali, different key")
```

PersistenceSchoolTest (5 of 6)

```
# continuing SchoolTest (5)
   def test_delete(self):
         student1 = Student('Peter', 123, 2010)
         student2 = Student('Ali', 234, 2011)
         student3 = Student('Kala', 345, 2009)

         self.assertFalse(self.school.delete(123), "empty student collection")

         # add some students
         self.school.create('Peter', 123, 2010)
         self.school.create('Ali', 234, 2011)
         self.school.create('Kala', 345, 2009)

         # Do not delete an unregistered student
         self.assertFalse(self.school.delete(300), "Unregistered student")

         # Delete a registered student
         self.assertTrue(self.school.delete(123), "Deleting Peter")

         # Do not delete a student that has already been deleted
         self.assertFalse(self.school.delete(123), "Deleting Peter again should not
   be possible")

         # Delete another registered student
         self.assertTrue(self.school.delete(234), "Deleting Ali")
```

PersistenceSchoolTest (6 of 6)

```
# continuing SchoolTest (6)
    def test_list_all(self):
            student1 = Student('Peter', 123, 2010)
            student2 = Student('Ali', 234, 2011)
            student3 = Student('Kala', 345, 2009)
            student4 = Student('Jin', 567, 2007)
            student5 = Student('Marcos', 789, 2008)

            self.assertEqual(0, len(self.school.list_all()), "empty student collection")

            # add one student
            self.school.create('Peter', 123, 2010)

            # List a singleton list
            listed = self.school.list_all()
            self.assertEqual(1, len(listed), "there should be 1 student in the list")
            self.assertEqual(student1, listed[0], "Listing Peter")

            # add some students
            self.school.create('Ali', 234, 2011)
            self.school.create('Kala', 345, 2009)

            # List with three students
            listed = self.school.list_all()
            self.assertEqual(3, len(listed), "there should be 3 students in the list")
            self.assertEqual(student1, listed[0], "Listing Peter")
            self.assertEqual(student2, listed[1], "Listing Ali")
            self.assertEqual(student3, listed[2], "Listing Kala")

            # delete some students
            self.school.delete(345)
            self.school.delete(123)

            # List is a singleton list again
            listed = self.school.list_all()
            self.assertEqual(1, len(listed), "there should be 1 student in the list")
            self.assertEqual(student2, listed[0], "Listing Ali")

            # add some more students
            self.school.create('Jin', 567, 2007)
            self.school.create('Marcos', 789, 2008)

            # List is back with three students, different ones
            listed = self.school.list_all()
            self.assertEqual(3, len(listed), "there should be 3 students in the list")
            self.assertEqual(student2, listed[0], "Listing Ali")
            self.assertEqual(student4, listed[1], "Listing Jin")
            self.assertEqual(student5, listed[2], "Listing Marcos")

            # delete all students
            self.school.delete(567)
            self.school.delete(234)
            self.school.delete(789)

            # List is empty again
            self.assertEqual(0, len(self.school.list_all()), "list should be empty")
```

Stricter Persistence Testing

- To be stricter in your persistence tests, you will also want to test that each method that affects persistence:
  - Get access to persistence first and load the data
  - Execute each CRUD operation that persists information
  - After each CRUD operation that persists information, reset the persistence mechanism, loading the data again
- By following that stricter approach, you may catch persistence bugs inside each test method

Stricter PersistenceSchoolTest (1 of 6): (adding persistence reset)

```
import os
from unittest import TestCase
from unittest import main
from student import Student
from school import School

class SchoolTest(TestCase):

   def setUp(self):
        conf = Configuration()
        self.conf.__class__.autosave = True
        self.filename = conf.__class__.filename
        self.school = School('Central High', True)
        self.school.student_dao.filename = 'students.dat'

   def tearDown(self):
        if os.path.exists(self.school.student_dao.filename):
              os.remove(self.school.student_dao.filename)

   def reset_persistence(self):
        self.school = School('Central High')
```

Stricter PersistenceSchoolTest (2 of 6)

```
def test_create_search(self):
  student1 = Student('Peter', 123, 2010)
  student1a = Student('Peter', 123, 2010)
  student2 = Student('Ali', 234, 2011)
  student3 = Student('Kala', 345, 2009)

  # add new student
  self.assertTrue(self.school.create('Peter', 123, 2010), "creating student1
should return success")
  self.reset_persistence()
  self.assertIsNotNone(self.school.search(123), "student1 is registed at school")
  self.reset_persistence() # Do we really need to reset persistence here?
  self.assertEqual(student1a, self.school.search(123),
     "registered student1 data should be the same as student1a")
  # add more students
  self.assertTrue(self.school.create('Ali', 234, 2011), "creating student2 should
return success")
  self.reset_persistence()
  self.assertTrue(self.school.create('Kala', 345, 2009), "creating student3
should return success")
  self.reset_persistence()
  self.assertEqual(student1, self.school.search(123), "student1 remains
registered")
  self.assertEqual(student2, self.school.search(234), "student2 is registered")
  self.assertEqual(student3, self.school.search(345), "student3 is registered")

        # code continues with the other stricter test methods...
```

Stricter PersistenceSchoolTest (3 of 6)

```
# continuing SchoolTest (3)
    def test_retrieve(self):
        student1 = Student('Peter Jackson', 123, 2010)
        student2 = Student('Peter Parker', 234, 2011)
        student3 = Student('Kala', 345, 2009)

        self.assertEqual(0, len(self.school.retrieve('Peter')), "empty student
    collection")

        # add some students
        self.school.create('Peter Jackson', 123, 2010)
        self.school.create('Peter Parker', 234, 2011)
        self.school.create('Kala', 345, 2009)
        self.reset_persistence()

        # Retrieve a singleton list
        retrieved = self.school.retrieve('Kala')
        self.assertEqual(1, len(retrieved),
                "there should be 1 student named Kala")
        self.assertEqual(student3, retrieved[0], "Retrieving Kala")

        # Retrieve a list with more than one element
        retrieved = self.school.retrieve('Peter')
        self.assertEqual(2, len(retrieved),
                "there should be 2 students named Peter")
        self.assertEqual(student1, retrieved[0], "Retrieving Peter Jackson")
        self.assertEqual(student1, retrieved[0], "Retrieving Peter Parker")
```

Stricter PersistenceSchoolTest (4 of 6)

```
# continuing SchoolTest (4)
    def test_update(self):
        student1 = Student('Peter', 123, 2010)
        student1a = Student('Peter Jackson', 123, 1995)
        student2 = Student('Ali', 234, 2011)
        student2a = Student('Ali Mesbah', 102, 2011)
        student3 = Student('Kala', 345, 2009)

        self.assertFalse(self.school.update(123, 'Peter Jackson', 123, 1995),
    "empty student collection")

        # add some students
        self.school.create('Peter', 123, 2010)
        self.school.create('Ali', 234, 2011)
        self.school.create('Kala', 345, 2009)
        self.reset_persistence()

        # Do not update an unregistered student
        self.assertFalse(self.school.update(300, 'Latifa', 300, 2001),
    "Unregistered student")

        # Update a registered student, maintaning key
        self.assertTrue(self.school.update(123, 'Peter Jackson', 123, 1995),
    "Updating Peter, same key")
        self.reset_persistence()

        # Update a registered student, changing key
        self.assertTrue(self.school.update(234, 'Ali Mesbah', 102, 1998),
    "Updating Ali, different key")
        self.reset_persistence()
```

Stricter PersistenceSchoolTest (5 of 6)

```
# continuing SchoolTest (5)
    def test_delete(self):
        student1 = Student('Peter', 123, 2010)
        student2 = Student('Ali', 234, 2011)
        student3 = Student('Kala', 345, 2009)

        self.assertFalse(self.school.delete(123), "empty student collection")

        # add some students
        self.school.create('Peter', 123, 2010)
        self.school.create('Ali', 234, 2011)
        self.school.create('Kala', 345, 2009)
        self.reset_persistence()

        # Do not delete an unregistered student
        self.assertFalse(self.school.delete(300), "Unregistered student")

        # Delete a registered student
        self.assertTrue(self.school.delete(123), "Deleting Peter")
        self.reset_persistence()

        # Do not delete a student that has already been deleted
        self.assertFalse(self.school.delete(123), "Deleting Peter again should not
    be possible")

        # Delete another registered student
        self.assertTrue(self.school.delete(234), "Deleting Ali")
        self.reset_persistence()
```

Stricter PersistenceSchoolTest (6 of 6)

```
# continuing SchoolTest (6)
    def test_list_all(self):
            student1 = Student('Peter', 123, 2010)
            student2 = Student('Ali', 234, 2011)
            student3 = Student('Kala', 345, 2009)
            student4 = Student('Jin', 567, 2007)
            student5 = Student('Marcos', 789, 2008)
            self.assertEqual(0, len(self.school.list_all()), "empty student collection")
            # add one student
            self.school.create('Peter', 123, 2010)
            self.reset_persistence()
            # List a singleton list
            listed = self.school.list_all()
            self.assertEqual(1, len(listed), "there should be 1 student in the list")
            self.assertEqual(student1, listed[0], "Listing Peter")
            # add some students
            self.school.create('Ali', 234, 2011)
            self.school.create('Kala', 345, 2009)
            self.reset_persistence()
            # List with three students
            listed = self.school.list_all()
            self.assertEqual(3, len(listed), "there should be 3 students in the list")
            self.assertEqual(student1, listed[0], "Listing Peter")
            self.assertEqual(student2, listed[1], "Listing Ali")
            self.assertEqual(student3, listed[2], "Listing Kala")
            # delete some students
            self.school.delete(345)
            self.school.delete(123)
            self.reset_persistence()
            # List is a singleton list again
            listed = self.school.list_all()
            self.assertEqual(1, len(listed), "there should be 1 student in the list")
            self.assertEqual(student2, listed[0], "Listing Ali")
            # add some more students
            self.school.create('Jin', 567, 2007)
            self.school.create('Marcos', 789, 2008)
            self.reset_persistence()
            # List is back with three students, different ones
            listed = self.school.list_all()
            self.assertEqual(3, len(listed), "there should be 3 students in the list")
            self.assertEqual(student2, listed[0], "Listing Ali")
            self.assertEqual(student4, listed[1], "Listing Jin")
            self.assertEqual(student5, listed[2], "Listing Marcos")
            # delete all students
            self.school.delete(567)
            self.school.delete(234)
            self.school.delete(789)
            self.reset_persistence()
            # List is empty again
            self.assertEqual(0, len(self.school.list_all()), "list should be empty")
```

More advanced integration tests

- The solution we found for testing persistence is somehow naïve and a bit expensive
- There are better ways to test persistence
  - Work with a simpler DAO object (**mock**) when not testing persistence
  - When testing persistence, use the real DAO object that persists information (these are the real **integration tests**)
- A course on Software Testing teaches **mocking** techniques
- Here we will use the simple naïve approach to persistence testing