Models in Memory in Python

Roberto A. Bittencourt

Basics

CRUD

- In a collection of students, as in any data collection, the program user will want to manipulate the data.
- Such data manipulation is what we call **CRUD**.

CRUD

- CRUD are the typical data manipulation operations:
  - **(C)REATE**
    - Create, register, add, insert
  - **(R)ETRIEVE**
    - Retrieve, consult, view, read
  - **(U)PDATE**
    - Update, modify, change, edit
  - **(D)ELETE**
    - Delete, remove, erase, exclude, eliminate

Search

- Very often, one performs a search before executing any CRUD operation
  - **CREATE**
    - Creating an object with an existing key must be avoided. So a search is needed to check whether the key is in use.
  - **RETRIEVE**
    - One may need to locate the data before retrieving it. That happens with a singleton search, other retrieval operations may be different.
  - **UPDATE**
    - One needs to locate the data before updating it.
  - **DELETE**
    - One needs to locate the data before deleting it.

Manipulating object collections

- For each CRUD operation, and additionally for search, we will design algorithms and programs to perform them
  - **Naïve algorithm**
    - Presents a typical sethatnce of suboperations, ignoring particular cases.
  - **Improved algorithm**
    - Solves the operation problem, fully handling all particular cases.
  - **Program in Python**
    - Implements the improved algorithm, generally as a function, illustrated by another program that uses that function.

Typical Object Collections

- It is usual to find information systems with three typical object collections:
  - **Unordered Object Lists**
    - Collections accessed by numerical indexs
    - They follow no particular order
  - **Ordered Object Lists**
    - Collections accessed by numerical indexs
    - They are ordered by one of their attributes (typically the key)
  - **Object Dictionaries**
    - Collections of key-value pairs accessed by their keys
    - Since Python 3.7, dictionaries are ordered by insertion order
    - We typically want a different sorting order (e.g., by key or by name), so we will consider them unordered

The **Student** model class below will be used in our examples...

```
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)
```

The **School** model class will be the used in our examples as a container of students...

```
# we will either use the option with lists
from student import Student
class School:
    def __init__(self, name):
        self.name = name
        self.students = []

# or another option with dictionaries
from student import Student
class School:
    def __init__(self):
        self.name = name
        self.students = {}
```

- All the following code is written either inside the School class, except for the `main()` function, which is at the main program level.

Unit Testing

Testing the **Student** class (student_test.py)

```
from unittest import TestCase
from unittest import main
from student import Student

class StudentTest(TestCase):
   # unit testing methods for Student added here

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

Testing Student **equality** (student_test.py)

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

     self.assertIsNotNone(student1, "initializing student 1")
     self.assertEqual(student1, student1a, "same students")
     self.assertNotEqual(student1, student2, "different students")
     self.assertNotEqual(student2, student3, "different students")
```

Testing Student **string form** (student_test.py)

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

     self.assertIsNotNone(str(student1))
     self.assertEqual("name: Peter, number: 123, birth year: 2010",
            str(student1), "Peter string")
     self.assertEqual("name: Kala, number: 345, birth year: 2009",
            str(student3), "Kala string")
     self.assertEqual(str(student1a), str(student1),
            "same students, same strings")
     self.assertNotEqual(str(student2), str(student1),
            "different students, different strings")
     self.assertNotEqual(str(student3), str(student1),
            "different students, different strings")
```

Testing the **School** class (school_test.py)

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

class SchoolTest(TestCase):
   # unit testing methods for School added here

  # The tests for School methods will be the CRUD operations
  # They will be shown after each operation is described

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

Search

Search: Naïve Algorithm

1. Enter the object key.
2. Locate the key inside the collection.
3. Return either the object index or the object itself.

Search: Improved Algorithm

1. Enter the object key.
2. Locate the key inside the collection.
3. If the key is found:
    1. Return either the object index or the object itself.
4. Else:
    1. Return an invalid index or a null object

Search: Program in Python with an ***unordered list***

```
# return the object indexed by key at the list
# if not found, return None
def search(self, key):
    # unordered list requests a linear search
    for element in self.students:
        if (element.number == key):
            return element
    return None

def main():
  school = School('Central High')
  # Suppose the students list has already been filled out
  key = int(input("Enter the student number: "))
  student = school.search(key)
  if student:
      print(student)
  else:
      print("No student with this number!")
if __name__ == '__main__': # showing main call only here
   main()
```

Search: Program in Python with an ***ordered list***

```
def binary_search(self, key, start, end):
    while start <= end:
        middle = (start + end)//2
        if self.students[middle].number == key:
            return middle
        elif self.students[middle].number < key:
            start = middle + 1
        else:
            end = middle - 1
    return -1
def search(self, key):
    # ordered list is faster with a binary search
    index = self.binary_search(key, 0, len(self.students)-1)
    return self.students[index] if index != -1 else None
def main():
  school = School('Central High')
  # Suppose the students list has already been filled out
  key = int(input("Enter the student number: "))
  student = school.search(key)
  if student:
      print(student)
  else:
      print("No student with this number!")
```

Search: Program in Python with a ***dictionary***

```
# return the object indexed by key at the dictionary
# if not found, return None
def search(self, key):
    return self.students.get(key)

def main():
 school = School('Central High')
  # Suppose the students dictionary has been filled out
  key = int(input("Enter the student number: "))
  student = school.search(key)
  if student:
      print(student)
  else:
      print("No student with this number!")
```

Testing School: **search student** (school_test.py)

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

    school = School('Central High')
    self.assertIsNone(school.search(123), "empty student collection")

    # adding students directly to the collection for the lack of a create() method
    # Python encapsulation allows it. But is that good testing practice?
    # Could have tested create and search together instead
    school.students.append(student1)
    self.assertIsNotNone(school.search(123), "student1 is registed at school")

    # This assertEqual() relies on the fact that Student equality has been tested
    self.assertEqual(student1a, school.search(123),
          "registered student1 data should be the same as student1a")

    # add more students
    school.students.append(student2)
    school.students.append(student3)
    self.assertEqual(student1, school.search(123), "student1 remains registered")
    self.assertEqual(student2, school.search(234), "student2 is registered")
    self.assertEqual(student3, school.search(345), "student3 is registered")
```

Create

Create: Naïve Algorithm

1. Prepare the object data.
2. Create the object with its data.
3. Find the correct position to add the object in the collection
4. Add the object in the collection.

Create: Improved Algorithm

1. Search for the key to check whether the object has already been created in the collection
2. If the key is not found:
    1. Prepare the object data.
    2. Create the object with its data.
    3. Find the correct position to add the object in the collection
    4. Add the object in the collection.
    5. Return success
3. If key is found:
    1. Return failure

Create: Program in Python with an ***unordered list***

```
def create(self, name, number, birth_year):
    if not search(self.students, number):
        student = Student(name,number,birth_year)
        self.students.append(student)
        return True
    else:
        return False

def main():
  school = School('Central High')
  if school.create('Peter', 123, 1995):
      print('Peter added.')
  if school.create('Ali', 234, 1998):
      print('Ali added.')
  if school.create('Latifa', 123, 1999):
      print('Latifa added.')
  else:
      print('Error adding Latifa. Inexistent Number.')
```

Create: Program in Python with an ***ordered list***

```
def create(self, name, number, birth_year):
    if not self.search(number):
        student = Student(name, number, birth_year)
        i = 0
        while (i < len(self.students)):
            if number < self.students[i].number:
                break
            i += 1
        self.students.insert(i, student)
        return True
    else:
        return False

def main():
  school = School('Central High')
  if school.create('Peter', 123, 1995):
      print('Peter added.')
  if school.create('Ali', 234, 1998):
      print('Ali added.')
  if school.create('Latifa', 123, 1999):
      print('Latifa added.')
  else:
      print('Error adding Latifa. Inexistent Number.')
```

Create: Program in Python with a ***dictionary***

```
def create(self, name, number, birth_year):
    if self.students.get(number):
        return False
    student = Student(name, number, birth_year)
    self.students[number] = student
    return True

def main():
  school = School('Central High')
  if school.create('Peter', 123, 1995):
      print('Peter added.')
  if school.create('Ali', 234, 1998):
      print('Ali added.')
  if school.create('Latifa', 123, 1999):
      print('Latifa added.')
  else:
      print('Error adding Latifa. Inexistent Number.')
```

Testing School: **create student** (school_test.py)

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

    # search() is already tested, it will be used to aid in testing create()
    school = School('Central High')
    self.assertIsNone(school.search(123), "empty student collection")

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

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