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")
```

Retrieve

Retrieve: Naïve Algorithm

- Choose the search field.
- Enter the object search string.
- Search the string in the objects' search field and create either a collection of object indices or of objects themselves that satisfy the search criterion.
- Return the collection.

Retrieve: Improved Algorithm

- Enter the object search string.
- Search the string in the objects' search field and create either a collection of object indices or of objects themselves that satisfy the search criterion.
- If any object was found:
  - Return the collection
- If no object was found:
  - Return an empty collection

Retrieve: Program in Python with an ***unordered list*** or with an ***ordered list***

```
def retrieve(self, name):
    retrieved = []
    for element in self.students:
        if name in element.name:
            retrieved.append(element)
    return retrieved
def main():
  school = School('Central High')
  school.create('Peter Smith', 123, 1995)
  school.create('Rika Nakamura', 345, 1987)
  school.create('Peter Parker', 234, 1998)
  retrieved = school.retrieve('Peter')
  if retrieved:
    print("Retrieved students:")
    for student in retrieved:
      print(student)
  else:
    print("Students with name %d not found.\n\n" % name)
```

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

```
def retrieve(self, name):
    retrieved = []
    for value in self.students.values():
        if name in value.name:
            retrieved.append(element)
    return retrieved
def main():
  school = School('Central High')
  school.create('Peter Smith', 123, 1995)
  school.create('Rika Nakamura', 345, 1987)
  school.create('Peter Parker', 234, 1998)
  retrieved = school.retrieve('Peter')
  if retrieved:
    print("Retrieved students:")
    for student in retrieved:
      print(student)
  else:
    print("Students with name %d not found.\n\n" % name)
```

Testing School: **retrieve students** (school_test.py)

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

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

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

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

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

Update

Update: Naïve Algorithm

1. Enter the object key.
2. Search the key and return either the object index or the object itself.
3. Update the object.

Update: Improved Algorithm

1. Enter the object key.
2. Search the key and return either the object index or the object itself.
3. If the object was found:
    1. If the key will change with the update operation:
        1. Check whether the key needs to be updated in the collection
    2. Update the object
    3. Return success
4. If the object was not found:
    1. Return failure

Update: Program in Python with an ***unordered list*** or with an ***ordered list***

```
def update(self, key, name, number, birth_year):
    student = self.search(key)
    if student:
        student.name = name
        student.number = number # simpler in lists
        student.birth_year = birth_year
        return True
    else:
        return False

def main():
  school = School('Central High')
  school.create('Peter', 123, 1992)
  school.create('Rika', 345, 1987)
  school.create('Ali', 234, 1998)
  if school.update(123, 'Peter Jackson', 123, 1995):
      print('Peter updated')
  if school.update(234, 'Ali Mesbah', 102, 1998):
      print('Ali updated')
```

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

```
    def update(self, key, name, number, birth_year):
        student = self.search(key)
        if student:
            if key != number:
                student = self.students.pop(key)
                self.students[number] = student
            student.name = name
            student.number = number
            student.birth_year = birth_year
            return True
        else:
            return False
    def main():
      school = School('Central High')
      school.create('Peter', 123, 1992)
      school.create('Rika', 345, 1987)
      school.create('Ali', 234, 1998)
      if school.update(123, 'Peter Jackson', 123, 1995):
          print('Peter updated')
      if school.update(234, 'Ali Mesbah', 102, 1998):
          print('Ali updated')
```

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

```
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)

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

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

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

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

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

Delete

Delete: Naïve Algorithm

1. Enter the object key.
2. Search the key and return either the object index or the object itself.
3. Delete the object.

Delete: Improved Algorithm

1. Enter the object key.
2. Search the key and return either the object index or the object itself.
3. If the object was found:
    1. Delete the object
    2. Return success
4. If the object was not found:
    1. Return failure

Delete: Program in Python with an ***unordered list*** or with an ***ordered list***

```
def delete(self, key):
    element = self.search(key)
    if element:
       self.students.remove(element)
       return True
    else:
       return False

def main():
  school = School('Central High')
  school.create('Peter', 123, 1992)
  school.create('Rika', 345, 1987)
  school.create('Ali', 234, 1998)
  if school.delete(123):
      print('Peter deleted')
  if school.delete(200):
      print('Student deleted')
  else:
      print('Error deleting student')
```

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

```
def delete(self, key):
    student = self.search(key)
    if student:
        self.students.pop(key)
        return True
    else:
        return False

def main():
  school.create('Peter', 123, 1992)
  school.create('Rika', 345, 1987)
  school.create('Ali', 234, 1998)
  if school.delete(123):
      print('Peter deleted')
  if school.delete(200):
      print('Student deleted')
  else:
      print('Error deleting student')
```

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

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

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

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

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

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

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

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

List all

Listing all elements from the collection

- Beyond the typical CRUD operations, it is very common to create an operation for listing all elements from the object collection
- This is more typical when the collection is encapsulated in an object and one wants to create a getter method for the collection
- Usually a copy of the collection is created so that it may be operated later (e.g., sorting) without affecting the original collection
- Typically the collection will be returned as a simpler collection (e.g., a list) or as an Iterator

List all: Program in Python with an ***unordered list*** or with an ***ordered list***

```
def list_all(self):
    students_list = []
    for student in self.students:
        students_list.append(student)
    return students_list

def main():
  school = School('Central High')
  school.create('Peter Smith', 123, 1995)
  school.create('Rika Nakamura', 345, 1987)
  school.create('Peter Parker', 234, 1998)
  lista = school.list_all()
  print("Listing all students:")
  for student in lista:
    print(student)
```

List all: Program in Python with a ***dictionary***

```
def list_all(self):
    students_list = []
    for student in self.students.values():
        students_list.append(student)
    return students_list

def main():
  school = School('Central High')
  school.create('Peter Smith', 123, 1995)
  school.create('Rika Nakamura', 345, 1987)
  school.create('Peter Parker', 234, 1998)
  lista = school.list_all()
  print("Listing all students:")
  for student in lista:
    print(student)
```

Testing School: **list all students** (school_test.py) (1 of 2)

```
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)

    school = School('Central High')
    self.assertEqual(0, len(school.list_all()), "empty student collection")

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

    # List a singleton list
    listed = 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
    school.create('Ali', 234, 2011)
    school.create('Kala', 345, 2009)

    # List with three students
    listed = 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")
```

Testing School: **list all students** (school_test.py) (2 of 2)

```
# continuing
# delete some students
school.delete(345)
school.delete(123)

# List is a singleton list again
listed = 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
school.create('Jin', 567, 2007)
school.create('Marcos', 789, 2008)

# List is back with three students, different ones
listed = 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
school.delete(567)
school.delete(234)
school.delete(789)

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

Sorting Object Collections

Sorting an object list

- Sorting an object list is essentially similar to sorting an integer list
- The only differences will be:
  - Comparison will be done against the chosen object sorting field
  - When swapping elements, swap the objects themselves

Recalling **Selection Sort**

Given the list named `lista` and being `size` the list length...

```
size = len(lista)
for i in range(0, size-1):
   min = i
   for j in range(i+1, size):
      //Find the least value
      if (lista[j] < lista [min]):
         min = j
   lista[i], lista[min] = lista[min], lista[i]
```

Selection Sort: Program in Python that sorts an ***unordered list*** by **number**

```
def selection_sort_by_number(self):
    size = len(self.students)
    for i in range(0, size-1):
       min = i
       for j in range(i+1, size):
          # Find the least student number
          if (self.students[j].number < self.students[min].number):
             min = j
       self.students[i], self.students[min] = self.students[min], self.students[i]
def main():
  school = School('Central High')
  school.create('Marcos', 789, 1997)
  school.create('Rika', 345, 1987)
  school.create('Ali', 234, 1998)
  school.create('Latifa', 456, 1999)
  school.create('Peter', 123, 1995)
  school.create('Jin', 567, 2001)
  for student in school.students:
    print(student)
  print("\n\nSorting by student number: \n")
  school.selection_sort_by_number()
  for student in school.students:
    print(student)
```

Selection Sort: Program in Python that sorts an ***unordered list*** by **name**

```
def selection_sort_by_name(self):
    size = len(self.students)
    for i in range(0, size-1):
       min = i
       for j in range(i+1, size):
          # Find the name that comes first
          if (self.students[j].name < self.students[min].name):
             min = j
       self.students[i], self.students[min] = self.students[min], self.students[i]
def main():
  school = School('Central High')
  school.create('Marcos', 789, 1997)
  school.create('Rika', 345, 1987)
  school.create('Ali', 234, 1998)
  school.create('Latifa', 456, 1999)
  school.create('Peter', 123, 1995)
  school.create('Jin', 567, 2001)
  for student in school.students:
    print(student)
  print("\n\nSorting by student name: \n")
  school.selection_sort_by_name()
  for student in school.students:
    print(student)
```

Selection Sort: Program in Python that presents a ***dictionary*** sorted by number

```
def selection_sort_by_number(lista):
    size = len(lista)
    for i in range(0, size-1):
       min = i
       for j in range(i+1, size):
          # Encontra o menor
          if (lista[j] < lista[min]):
             min = j
       lista[i], lista[min] = lista[min], lista[i]
def keys_sorted_by_number(self):
    keys = list(self.students.keys())
    selection_sort_by_number(keys)
    return keys
def main():
  school = School('Central High')
  school.create('Marcos', 789, 1997)
  school.create('Rika', 345, 1987)
  school.create('Ali', 234, 1998)
  school.create('Latifa', 456, 1999)
  school.create('Peter', 123, 1995)
  school.create('Jin', 567, 2001)
  for key in students:
      print(students[key])
  print("\n\nPresenting students sorted by student number:\n")
  keys = school.keys_sorted_by_number()
  for key in keys:
      print(students[key])
```

Testing School: **selection sort by number** and **selection sort by name** (school_test.py)

- These testing methods are left to you as an exercise
- Which strategies would you use to test them?

Models in Memory in Python

They are not that difficult, isn't it? I hope you enjoyed them! Information systems strongly rely on models. So learning how to handle them is an important skill!