Digital Skills Lab
Lesson 11 (practice) - 16 and 18/03/2021 - r2

The goal of this practice is developing the information system of a university, in a bottom-up fashion.

Informally, our information system shall provide the following features:

- Store students and their exams:
  - Create a new student
  - Add an exam to an existing student
- Retrieve students by id, and by last name
- Save and load data from files
- Generate reports:
  - list of the exams of a specific student
  - list of all the students, with the number of exams and the average grade of each student

1. Class `Person`

We wish to represent students, but we notice that, in a future version of our software, we may want to add other kinds of people, such as teachers/professors. For this reason we generalize, starting from a class `Person`.

Create a file named `university.py`, and define a simple class `Person` with the following members:

- A constructor taking two parameters: `last_name` and `first_name`
- Two instance variables, named `last_name` and `first_name`

Test your class:

```
john = Person('Smith', 'John')
print(john.last_name)   # Smith
print(john.first_name)  # John
```

2. Class Student

Now, we define a class `Student` which extends class `Person`. Additional data that we wish to represent include:

- A student id
- The list of exams passed by the student, which is initially empty. We will represent each exam as a 2-tuple `(course_name, exam_grade)`, such as `('Programming', 30)`

We want to be able to perform the following operations:

- Add a new exam
- Read the list of exams
- Get the number of exams
- Get the average grade of exams (or `None`, if no exam has been passed)

Thus, our class `Student` shall provide the following methods:

- Constructor: takes parameters `student_id`, `last_name`, `first_name`
- `add_exam(course_name, grade)`: adds a new exam
- `get_all_exams()`: returns the list of exams
- `get_exam_count()`: returns the number of exams
- `get_exam_average_grade()`: returns the average grade, or `None` if there are no exams

This is how your class will look like; complete it where needed:

```
class Student(Person):
  def __init__(self, stud_id, last_name, first_name):
    """
    Constructor
    """
    super().__init__( ...pass parameters to constructor of Person... )
    self.stud_id = stud_id
    self._exams = []
    # ... define other instance variables if useful ...

  def add_exam(self, course_name, grade):
    """
    Add a new exam
    """
    exam = (course_name, grade)
    # ... now append exam to self._exams

  def get_all_exams(self):
    """
    Return the list of exams
    """
    # ...

  def get_exam_count(self):
    """
    Return the number of exams
    """
    # ...


  def get_exam_average_grade(self):
    """
    Returns the average grade

    If there are no exams, return None
    """
    # ...
```

Always test your code! For example:

```
john = Student('0123', 'Smith', 'John')
print(john.last_name)                # Smith
print(john.stud_id)                  # 0123
print(john.get_all_exams())          # []
print(john.get_exam_average_grade()) # None


john.add_exam('Programming', 30)
john.add_exam('Algorithms', 28)

print(john.get_all_exams())          # [('Programming', 30), ('Algorithms', 28)]
print(john.get_exam_average_grade()) # 29.0
```

Computational complexity: Consider method `get_exam_average_grade()`. If you compute the average grade by reading list `self._exams`, the cost will be O(*n*), where *n* is the number of exams. Can you develop a more efficient solution? You need to add more instance variables to class `Student`, and keep them updated when you add a new exam, so that method `get_exam_average_grade()` will be able to compute the average in O(1) (i.e., without a loop).

3. University (core methods)

Now we are ready to group students together in a university.

We define a first version of a class `University` which shall provide the following methods:

- Constructor: builds up an empty university
- `add_student(s)`: adds an object `s` of class `Student` to the university

- `get_student_by_id(stud_id)`: returns the student whose student id is the string `stud_id`.

Our class shall hold the collection of university students in some data structure. The data structure shall support efficient lookup by student id: which data structure can we choose?

We can choose a data structure that maps student id's (that are strings) to objects of class `Student`: a dictionary will be a perfect choice, where:

- keys are student id's (strings)
- values are `Student` objects

Here is the skeleton of our class. Complete it where needed, and test your class.

```
class University:
  def __init__(self):
    """
    Constructor: build up an empty university
    """
    # We initialize an empty dictionary, that will map student id's to
    # student objects
    self._students = {}

  def add_student(self, s):
    """
    Add an object s of class Student to the university
    """
    # Extract student id from object s
    stud_id = ... (read stud_id from s) ...
    # Add s to the dictionary with key stud_id
    self._students[stud_id] = s

  def get_student_by_id(self, stud_id):
    """
    Return the student whose student id is the string stud_id
    """
    # ... lookup key stud_id in the dictionary
    # ... return the associated value (that is a Student object)
```

Some test code for you (please write more tests on your own):

```
john = Student('0123', 'Smith', 'John')
mary = Student('0222', 'Smith', 'Mary')

u = University()
u.add_student(john)
u.add_student(mary)

s = u.get_student_by_id('0222')
print(s.first_name)               # Mary
```

4. Load students from a CSV file

Add a method `load_students(filename)` to class `University` that load students from a CSV file. The format of each line of the file is `student_id;last_name;first_name`.

For example, create the following file `students.csv`:

```
0123;Smith;John
0222;Smith;Mary
0333;Gates;William
0444;Jobs;Steven
0555;Torvalds;Linus
```

Your method shall read the file, line by line. For each line, after reading the three elements (student id, last and first name), your method shall create a `Student` object:

```
def load_students(self, filename):
  # ... open file f ...
  for line in f:
    # ... strip trailing \n ...
    # ... split line in stud_id, last_name, first_name
    s = Student(stud_id, last_name, first_name)
    # Add student s to current university, i.e. to self:
    self.add_student( ... )

  # Don't forget to close the file!
```

Then test your method with the following code:

```
u = University()
u.load_students('students.csv')
bill = s.get_student_by_id('0333')
print(bill.last_name)               # Gates
```

5. Load exams from a CSV file

Add a method `load_exams(filename)` to class `University` that load exams from a CSV file. The format of each line of the file is `student_id;course_name;exam_grade`.

For example, create the following file `exams.csv`:

```
0333;Programming;24
0444;Programming;18
0333;Algorithms;22
0555;Programming;30
0555;Algorithms;29
```

Notice that your method, for every line of the file, shall:

1. Read student id
2. Lookup for the student (that shall already exist in the dictionary)
3. Add an exam to the student

Then, you should be able to run the following test code:

```
u = University()
u.load_students('students.csv')
u.load_exams('exams.csv')
bill = s.get_student_by_id('0333')
print(bill.get_exam_average_grade())  # 23.0
```

6. Generate a report

Now let's add a method `generate_student_report()` to class `University`, whose goal is to return a report of all the students, the number of exams they have passed, and their average grade.

More precisely, a call to `generate_student_report()` shall return a list of tuples, where each tuple contains data about a student. Each tuple has the following form:

```
(stud_id, last_name, first_name, exam_count, grade_avarage)
```

For example, after loading files `students.csv` and `exams.csv`, a call to `u.generate_student_report()` shall return the following list:

```
[('0123', 'Smith', 'John', 0, None),
 ('0456', 'Smith', 'Mary', 0, None),
 ('0333', 'Gates', 'William', 2, 23.0),
 ('0444', 'Jobs', 'Steven', 1, 18.0),
 ('0555', 'Torvalds', 'Linus', 2, 29.5)]
```

Suggestion 1: Your method will need to **iterate** over the dictionary of students, in order to read their data. When you iterate, you get the **keys** (i.e., student id's), that you will use to read the **values** (i.e., the student objects):

```
  def generate_student_report(self):
    out = []
    for student_id in self._students:
      # We use the key, student_id, to get the value from the dictionary
      student = ... read student object associated to student_id
      # ... get data from student, build the tuple and append it to list out

    return out
```

7. Get students by last name

We add another method to class `University` that allows to search students by last name. Notice that there may be several students that share the same last name; thus our method `get_students_by_last_name(last_name)` will return a **list** of students.

For example:

```
result = u.get_students_by_last_name('Smith')

for s in result:
  print(s.first_name)

# John
# Mary

result = u.get_students_by_last_name('Torvalds')

for s in result:
  print(s.first_name)

# Linus
```

We will implement the method using two different approaches.

7.1 First approach: an O(*n*) search

We first implement the method `get_students_by_last_name(last_name)` in a less efficient way:

- we build an output list `out`, initially empty
- we iterate over all the students in the dictionary
- for each student `s`, if `s.last_name == last_name` then we append `s` to `out`

Implement and test the method. Since the method scans through the whole dictionary, its asymptotic complexity is O(*n*).

7.2 Second approach: a dictionary lookup

We can improve the search cost using a dictionary lookup. The dictionary `self._students` cannot be of any help, because its keys are student id's.

We need to define another dictionary in our class, whose keys are students' last names. We say that we are **indexing** students by last name. Thus, create a new dictionary in method `__init__` of class `University`:

```
  def __init__(self):
    # ...
    self._students_by_last_name = {}
```

Now we must modify method `add_student` because, every time we add a new student, it must be added to both the dictionaries:

- `self._students`
- `self._students_by_last_name`

Notice, however, that several students may share the same last name (like `Smith` in our example). On the other hand, dictionary keys must be unique. Thus our dictionary `self._students_by_last_name` will work differently than `self._students`. It will map each last name to a **list** of students, i.e. to all the students with that specific last name. For example our full dictionary will look like this:

```
{
  'Smith': [<Student: John Smith>, <Student: Mary Smith>],
  'Gates': [<Student: William Gates>],
  'Jobs': [<Student: Steven Jobs>],
  'Torvalds': [<Student: Linus Torvalds>]
}
```

In other words, in our dictionary:

- keys are strings (last names)
- values are list of `Student` objects

Method `add_student(s)` will need to update both the dictionaries; in particular it will:

- read `s.last_name`
- check whether `s.last_name` is already a key of dictionary `self._students_by_last_name`:
  - if not, then it is the first time we add a student with `s.last_name` as last name. We create a new empty list, and we add it to the dictionary: `self._students_by_last_name[s.last_name] = []`
- finally, we can append student `s` to the list `self._students_by_last_name[s.last_name]`

You also have to create (and test) method `get_students_by_last_name(last_name)`. This time implementation is trivial, because a lookup in our new dictionary is sufficient.

8. Save students and exams

Add to class `University` methods to save students and exams, in the same CSV format expected by load methods:

- `save_students(filename)`
- `save_exams(filename)`

9. User interface

Now that all our building blocks are ready, let's create a friendly user interface (UI), through which a user will be able to perform all the operations.

For the sake of this exercise we will create a **textual user interface**. Thanks to the modularity of our program, by reusing the same core classes (`University`, `Student`, etc.) we could also develop a **graphical user interface** (GUI) or a **web user interface**.

Our textual interface will show a main menu, like this:

```
Welcome to our University Information System!
==============================================

Please select an operation:

1. Load students from CSV
2. Load exams from CSV
3. Save students to CSV
4. Save exams to CSV
5. Add a student
6. Add an exam
7. Show student data (with exams) by student id
8. Search students by last name
9. Report students stats
0. Exit

Your choice -> 1
Students CSV file name -> students.csv
Students loaded successfully!

Please select an operation:

[...]

Your choice -> 7
Enter student id -> 0333

Student id: 0333
Last name: Gates
First name: William

List of exams:
- Programming (24)
- Algorithms (22)

Please select an operation:

[...]

Your choice -> 0
Goodbye!
```

We build our UI in a new Python module. Create a new file named `ui.py` and import the classes `Student` and `University`. Then, we define a "main" function called `main_menu`:

```python
from university import Student, University

def main_menu():
  u = University()

  print("Welcome to our University Information System!")

  while True:
    print("Please select an operation:\n")
    print("1. Load students from CSV")
    # ...

    choice = input("Your choice:> ")

    if choice == '1':
      load_from_csv(u)

    if choice == '2':
      # ...
```

For each operation, we can define a function that performs the operation using the methods of classes `University` and `Student`, and interacting with the user when needed; for example:

```
def load_from_csv(u):
  filename = input("Students CSV file name:> ")
  u.load_students(filename)
  print("Students loaded successfully!")
```

Complete the implementation to offer all the expected features.

10. Handle errors

In some cases our program will halt with an error, for example:

- If we try to open a nonexisting file
- If we search a nonexisting student id

We need to handle these errors, by catching the corresponding exceptions and showing a nice informational message.

For example, if we try to open a nonexisting CSV file, we get this exception:

```
FileNotFoundError: [Errno 2] No such file or directory: ...
```

Thus we need to catch the exception named `FileNotFoundError` in our `load_from_csv(u)` function (in `ui.py`):

```
def load_from_csv(u):
  filename = input("Students CSV file name: ")
  try:
    u.load_students(filename)
    print("Students loaded successfully!")
  except FileNotFoundError:
    print(f"Sorry, file {filename} does not exist, try again!")
```

Modify your functions so that errors are handled in the following situations:

- Loading students from a file
- Looking up a student by id (which exception is raised? Get a nonexisting student to discover the type of the raised exception)
- Loading exams. At least two things may go wrong while loading exams:
    1. The exams file does not exist
    2. A student id in exam list does not match any existing student