1 Administrative Topics

- We look at project 7.

2 More about Students

On Monday, we talked about classes. Classes are used to describe new object types. The new object type we designed on Monday was called Student, and it stored and reported a year of graduation and a name. It has two accessor methods, getName and getYear.

Today, we will add new methods to that class.

We begin with the \texttt{str} method, which is automatically called by Python when we print the object. This method must return a string.

\begin{verbatim}
def __str__(self):
    return self.name + ' ' + str(self.year)
\end{verbatim}

It is automatically called by the second line of this code:

\begin{verbatim}
s = Student( 'John', 2016 )
print s
\end{verbatim}

Next, we add a mutator method \texttt{setYear} so that students can update their graduation gate if necessary.
3 Giving Students Grades

Let’s add the possibility for Student objects to include grades. Let’s be super simple about this. Grades are simply stored in a list and they must be numbers between 0 and 4.

A student record is initialized without any grades (a new student doesn’t have any).

Grades are added one by one. For this, we need a mutator method named \texttt{addGrade}.

We should be able to get the list of grades, so we need an accessor method \texttt{getGrades}.

And we should be able to compute a GPA.

First, we add the line \texttt{self.grades = []} to the init function:

```python
# init with name and year
def __init__(self, name, year):
    self.name = name
    self.year = year
    self.grades = []
```

This ensures the data field is there, and is consistent with our statement that no student enters Colby with grades.

Next, we write \texttt{addGrade}:

```python
# add a grade. grade must be a tuple:
#   0: course name <str>
#   1: grade <int> or <flt>
def addGrade(self, grade):
    self.grades.append(grade)
```

Next, we write \texttt{getGrades}. Note that it would be straightforward to write code that simply returns \texttt{self.grades}. But we don’t want to do this. We want to
“protect” our data. First, remember that we want code within class methods to manipulate data in the object’s symbol table and NO ONE ELSE. Second, remember what type self.grades is. It is a list. Lists are mutable. If we simply return a reference the grades field, then we are giving the caller (which could be code outside the class) the ability to edit the list. Ahhh. We don’t want that. Instead, we want to return a clone of the list. To do that, we simply make a new list, and copy each element in. Since the elements are tuples (and tuples are immutable), we don’t need to do that in any special way.
So, the code for `getGrades` is:

```python
# return a clone of the grades
# this is a list of (course name, grade) duples
def getGrades(self):
    #return self.grades # straight-forward, but not safe!
    grades = []
    for grade in self.grades:
        grades.append(grade)
    return grades
```

Our next goal is to compute the GPA, and to do that, it would be nice to have a list of numbers instead of a list of duples. So, let’s make a method that extracts the number grades from the duples list:

```python
# return a list of numeric grades
def getNumericGrades(self):
    nums = []
    for duple in self.grades:
        nums.append(duple[1])
    return nums
```

Finally, we are ready to write `getGPA`. The most interesting aspect of this function is the syntax for calling one method from another. We use the dot notation with the `self` variable as the object.

```python
# return the average of the number grades
def getGPA(self):
    ng = self.getNumericGrades()
    if len(ng) == 0:
        return 0.0
    avg = 0.0
    for g in ng:
        avg += g  # equiv to avg = avg + g
    avg /= len(ng)  # equiv to avg = avg / len(ng)
    return avg
```

The complete code has been uploaded to the course web page.
4 Miscellany

Here is a summary of the Student class.

It has data fields

- name
- year
- grades

and methods

- getName (an accessor method)
- getYear (an accessor method)
- setYear (a mutator method)
- __init__
- addGrade (a mutator method)
- getGrades (an accessor method)
- getNumericGrades (an accessor method)
- getGPA (an accessor method)

And here are a couple of notes:

- Note that there is an entry for each data field and each method in a Student object’s symbol table. One implication is that it means you cannot make methods and data fields with the same name.

- Also, note that the methods are added to the object’s symbol table when Python first creates the object. The data fields are added when __init__ is executed. As a general rule, if any data field that is ever going to be in an object, it should be added in the __init__ method. This makes for easy reading later.
4.1 Should every field have an associated parameter in the init method?

When designing a class, we need to decide which fields we want to be initialized by parameters to the init method. Here are guidelines for making that decision:

Is there a natural default value for that field name?

- If not, then make a parameter. (e.g. a Student’s name)
- If not, and we don’t want the caller to be able to give it a value, then we do not include a parameter. The grades field in the Student class is a good example of this situation. The design of our Student class is that all students start out with no grades. We don’t want a parameter to allow our caller to interfere with that design.
- If there is, but we want the calling code to have the ability to change it, then we can include a parameter, but give that parameter a default value. A good candidate for this approach would be the year of graduation in the Student class. Suppose you are setting up the code to run it for students entering Colby a particular year. Most of the will be graduating in four years, but some of them are transfer students who will be graduating in less time. Then we could do this:

```
# init with name and year
def __init__(self, name, year=2021):
    self.name = name
    self.year = year
    self.grades = []
```

4.2 How to make sure data are the right type

Here is an issue that appears whether we are using object-oriented design or top-down design. We write our code in a way that requires values for parameters to be certain types. For example, we expect grades to be numbers. There are 3 strategies one can take to make those expectations clear (or not):
• Passive-aggressive. Just assume the data is the right type and operate on it accordingly. If it is the wrong type, there will just be an exception (dramatic blow-up)

• Compliant, fix-it-under-the-hood. do your best to massage the data before putting it into your fields (e.g. call the str function on it). This never lets the user know they did anything wrong.

• Pro-active. Explicitly test the type of the data. Throw an error if it isn’t correct.