Solutions to CS 61A Challenge Problems: Object Oriented Programming

Other worksheets and solutions at https://alextseng.net/teaching/cs61a/

Alex Tseng

1 Defining a Class

We define a new class to create objects that represent high school students. A basic high school student has a name, a grade, and a favorite subject.

(a) Fill in the functions below to complete the class definition so that the doctests pass.

```
class Student:
    students_enrolled = 0
    student_years = {}
    def __init__(  self, name, grade  ):
        self.name = name
        self.grade = grade
        self.fave_subject = "No favorite"
        Student.students_enrolled += 1
        if grade in Student.student_years:
            Student.student_years[grade] += 1
        else:
            Student.student_years[grade] = 1

    def set_favorite_subject(  self, subject  ):
        self.fave_subject = subject

    def students_in_grade(  grade  ):
        return Student.student_years[grade]

>>> tiffany = Student("Tiffany", 9)  # a student by default has no favorite subject
>>> tiffany.name
"Tiffany"
>>> tiffany.grade
9

>>> mike = Student("Michael", 11)
>>> mike.set_favorite_subject("Biology")
>>> mike.fave_subject
"Biology"
>>> tiffany.fave_subject
"No favorite"

>>> Student.students_in_grade(9)
1
>>> Student.students_in_grade(10)
0

>>> zack = Student("Zackary", 11)
>>> Student.students_in_grade(11)
2
```

```
>>> Student.students_enrolled
3
```

Within the `__init__` function, we set the name and grade of each student to be instance variable, so we must use `self`. When referencing a class variable such as `student_years`, we must use the dot notation starting with the class name, `Student`. This is because class variables are shared across all instances, and you must tell Python that you mean to reference a class variable and not a local variable (this works a little bit like the `nonlocal` statement). Whenever we create a new student, we also increment the number of students that are enrolled, and add 1 to the correct grade in the student years dictionary. Notice the if/else: we need to make sure that the grade exists in the dictionary before incrementing it! What would happen if we tried to increment the grade and it were not in the dictionary? We would get a KeyError. If we'd like, we can add a test to check if the grade is in the dictionary first, or use the `get` function.
Notice that in `students_in_grade`, there is no `self` in the function declaration. This is because it is a class function, not a bound instance method. This is meant to be called as `Student.students_in_grade(x)`.

(b) What would happen if we called `mike.students_enrolled`?
We would get the correct answer, although this is technically incorrectly used. Recall that `mike` is an instance variable, so it *inherits* all of its class variables. However, this could also be a violation of a data abstraction barrier, because it is not meant to be used this way. This variable is meant to be a shared class variable, not an individual instance variable. It is possible to make this call return the wrong answer by explicitly setting another instance variable: `self.students_enrolled = "foo"`. If this were the case, we would get `"foo"` instead of the total number of enrolled students, since an instance variable overwrites a class variable if you use the notation `instance.attribute`.

(c) What would happen if we called `Student.student_years[9]`?
We would get the correct answer. However, this is a major violation of a data abstraction barrier. We cannot assume that the student years are stored as a dictionary. Because a class function exists to retrieve the student years, it is clear that the user is meant to use this function instead of directly referencing the dictionary. Because this violates a DAB, even though this call works now, it may easily fail in the future if we decide to change some of the inner implementations.
For example, since there are only 4 grades in high school, we can decide to store `student_years` as a list of 4 items instead, so that `student_years[0]` is the number of freshman students. Then we could easily change the corresponding function `students_in_grade`, so that it would still work correctly. Then all users that are respecting the DAB will still have correct code. But a user that directly references `student_years` will get an IndexError for trying to get the 9th element from a 4-element list. Always respect data abstraction barriers.

(d) High school students also get crushes on each other. A student can have a crush on any other student (or none at all). Many students may have a crush on the same person, but a student can only have a crush on one person. Write a bound method `develop_crush` that stores a student's crush as an instance attribute `crush`. By default, a student has no crush (`crush` is `None`). What change would you need to make to `__init__`?

```
def __init__(self, name, grade):
    ...
    self.crush = None  # Add this line

def develop_crush(self, crush):
    self.crush = crush
```

Notice that we must edit `__init__` so that the default crush is `None`.

(e) *Challenge* Now we also wish for the *class* to know how popular certain students are. That is, we want to know how many people have crushes on each student. You will need a class variable to store how many crushes each person is the receiver of. You may also need to alter `__init__` and `develop_crush`.

```
crushes = {}  # Dictionary holding crushes, a class variable

def __init__(self, name, grade):
    ...
    self.crush = None
    Student.crushes[name] = 0

def develop_crush(self, crush):
    self.crush = crush
    Student.crushes[crush] += 1
```

This question is more challenging because you have the freedom to choose how you wish to implement this functionality. If you did not look at the question in (f), you would not even know why storing the crushes would be useful. This problem boils down to OOP *design*, or having the foresight to make conscientious and smart decisions on how you implement objects. There are multiple solutions as to what data structure you use to store the crushes, and how you retrieve them, but the easiest is probably with a dictionary. Note that we automatically set the crushes on any student to be 0 in this dictionary as soon as it is initialized. This means we don't have to do the if/else check like before whenever we develop a crush.
Notice that when we develop a crush, not only do we set the crush attribute to a student who is developing the crush, but we also increment the dictionary `crushes` appropriately.

(f) Now that you've implemented (e), write a class method `crushes_on` that is able to find the number of people who have a crush on some student, by name. For example, if `tiffany` and `zack` both have a crush on `mike`, then `Student.crushes_on("Michael")` should return 2. What happens if we call it on a student that doesn't exist?

```
def crushes_on(student):
    return Student.crushes[student]
```

Again, notice the lack of `self` because this is a class function, not a bound method. We reference the class variable `crushes`, which is a dictionary. Because whenever a student is initialized, we put their name into the dictionary, we do not need to check whether or not `student` is in the dictionary already. But this also means that if we call `crushes_on` on a student that doesn't exist, we will get a KeyError.

2 Special Methods

Python actually has many special methods, most of which are used quite rarely. We will explore the uses of several of the more useful ones. Consider the following basic definition of molecules:

```
class Molecule:
    def __init__(self, formula, name, weight):
        self.formula = formula
        self.name = name
        self.weight = weight
```

(a) Two molecules are equivalent if and only if they have the same name (we can't use the formula because constitutional isomers have the same formula but can have very different structures!). Add a bound method so that the following doctests pass. Recall that `__eq__(self, other)` returns a boolean and allows equality testing using ==.

```
>>> ethanol = Molecule("C2H6O", "Ethanol", 46.07)
>>> alcohol = Molecule("C2H6O", "Ethanol", 46.07)
>>> dimethyl_ether = Molecule("C2H6O", "Dimethyl Ether", 46.07)
>>> ethanol == alcohol
True
>>> ethanol == dimethyl_ether
False

def __eq__(self, other):
    return self.name == other.name
```

(b) We can also compare molecules using molecular weights with < and >. Hint: `__lt__` and `__gt__` work just the same as `__eq__`.

```
>>> ammonia = Molecule("NH3", "Ammonia", 17.03)
>>> ammonia < ethanol
True
>>> ammonia > dimethyl_ether
False

def __lt__(self, other):
    return self.weight < other.weight
def __gt__(self, other):
    return self.weight > other.weight
```

(c) Consider the following definitions of `__str__` and `__repr__`. What would Python print?

```
def __str__(self):
    return self.formula + ": " + self.name
def __repr__(self):
    return "Molecule('0', '1', 2)".format(self.formula, self.name, self.weight)

>>> print(ammonia)
NH3: Ammonia
>>> ethanol
Molecule('C2H6O', 'Ethanol', 46.07)
>>> str(alcohol)
'C2H6O: Ethanol'
>>> repr(ammonia)
"Molecule('NH3', 'Ammonia', 17.03)"
>>> eval(repr(ethanol))  # eval evaluates a string as native Python
Molecule('C2H6O', 'Ethanol', 46.07)
>>> print(repr(ethanol))
Molecule('C2H6O', 'Ethanol', 46.07)
```

You may be confused about the purpose of `str` vs. `repr`. `str` is used to display the object in a *human-readable* format. It should be easy to inspect and see the important attributes. Printing an object uses the `str` string. `repr` also returns a string, but this string is intended to be unambiguous and comprehensive. It is more useful in debugging. Oftentimes, the string that `repr` returns can be directly evaluated in producing the exact same object. When you directly query Python for the value of an object (like by just typing `ethanol`), you get the `repr` string value.
Notice that when you call `str` or `repr` explicitly on an object, Python displays the string with quotes. This is because those functions return strings, so they will be displayed in quotes, as normal. However, when you print or directly evaluate an object, the `str` string or `repr` string (respectively) are displayed *without* quotes. This is a special behavior of printing and direct evaluation. In general, when you print a string, the outside quotes are removed.