CSC148 Recipe for Designing Classes

Part 1: Define the API for the class

Download the sample code here: `http://www.cs.toronto.edu/~david/csc148/content/oop/course.py`.
A summary can be found on the last page.

1. **Class name and description.** Pick a noun to be the name of the class, write a one-line summary of what that class represents, and (optionally) write a longer, more detailed description. These should help others quickly figure out what your class represents.

```
class Course:
    """A university course.
    """
```

In this case, there isn't much to say, since most everyone knows what a university course is. You may wonder why we don't explain what aspects of a course the class is going to model, but that will come at a later step.

2. **Example.** Write some simple examples of client code that uses your class. This will help you figure out what the API should be. By taking the point of view of the client, your design is likely to make the class convenient to use. Focus for now on *standard* cases (as opposed to a tricky or corner case.) You can put your code in a "main block." Example:

```
if __name__ == '__main__':
    csc148 = Course('csc148', 120, dict(a1=10, a2=10, t1=15, t2=15, final=50))
    csc148.enrol('123456789')
    csc148.enrol('111111111')
    csc148.enrol('888888888')
    csc148.record_grade('123456789', 'a1', 86)
    csc148.record_grade('123456789', 'final', 86)
    csc148.record_grade('888888888', 'a1', 74)
    print(csc148.grade('888888888', 'a1'))
    print(csc148.course_grade('123456789'))
    print(csc148.class_average('a1'))
    print(csc148)
```

In order to come up with this code, many decisions had to be made. For example, we chose to specify an enrolment cap (in this example, it is 120) and had to come up with a way to specify the course marking scheme (we chose to use a dictionary). As we progress through the recipe, we may realize some decisions weren't great. That's fine; we can come back and revise.

3. **Public methods.** Using your example as a starting point, decide what services your class should provide for client code, i.e., what actions could be performed on instances of this class. For each of these actions, use the first four steps of the Function Design Recipe to define the API for a method that will provide the action:

(1) Example

(2) Type Contract

(3) Header

(4) Description

Since you are writing methods, not functions, don't forget to include `self` as the first parameter.

You *must* define a constructor, `__init__`, and often will want to define `__str__` to generate a string representation of instances of the class, and `__eq__` to check whether two instances are equal.

For brevity, only the constructor and one other method is shown below. You can see the rest in the full example code.

```
    def __init__(self, name, cap, scheme):
        """Initialize this course.

        @type self: Course
        @type name: str
            The name of this course.
        @type cap: int
            The enrolment cap for this course.
        @type scheme: dict[str, int]
            The marking scheme for this course.  Each key is an element of the
            course, and its value is the weight of that element towards the
            course grade.
            Precondition: The sum of all weights must be 100.
        @rtype: None
```

```
        >>> c = Course('cscFun', 50, {'exam': 100})
        >>> c.name
        'cscFun'
        >>> c.cap
        50
        """
        pass

    def enrol(self, student_id):
        """Enrol a student in this course.

        Enrol the student with id <student_id> in this course, if there is
        room.

        @type self: Course
        @type student_id: str
        @rtype: bool
            True iff enrolment was successful, i.e., this student was not
            already enrolled, and there was room for to enrol him or her.

        >>> c = Course('cscFun', 50, {'exam': 100})
        >>> c.enrol('12345')
        True
        >>> c.grade('12345', 'exam') == None
        True
        """
        pass
```

4. **Public attributes.** Decide what data you would like client code to be able to access without calling a method. This is not a clear-cut decision, since one could require *all* data to be accessed by calling a method (and in some languages, that is the convention). Python takes the oppositive point of view: treat attributes as public unless you have a good reason not to.

   Here are two situations when it makes sense to treat the attribute as private. In these cases, we expect the user to access the data by calling methods.

   - An attribute with complex restrictions on its value. If client code were to assign a value to the attribute directly, it might inadvertently violate the restriction. If instead it is required to call a method to change the value, the method implementation can enforce any restriction.
   - An attribute that represents data from the domain in a complex way. (We'll learn some fairly complex data structures this term.) By expecting client code to access the information through a method call, we spare the client code from having to be aware of the complex details, and we also avoid the problem of client code accidentally messing up important properties of the data structure.

   Once you have chosen the public attributes, add a section to your class docstring after the description, specifying the **type**, **name**, and **desecription** of each of these attributes. Use the format below.

```
class Course:
    """A university course.

    === Attributes ===
    @type name: str
        The name of this course.
    @type cap: int
        The enrolment cap for this course, i.e., the maximum number of
        students who may enrol.
    """
```

At this point you have defined everything that client code needs in order to use the class successfully.

Part 2: Implement the class

Now turn your attention to implementing the class. Any comments you write at this point concern implementation details, and are for the developers of the class itself. As a result, they will not go in the class docstring or in method docstrings; these are for developers of client code.

5. **Internal (private) attributes.** In order to write the bodies of the public methods, you will likely need additional attributes, but ones that the client code (a) need not know about and (b) should not access directly. For example, we need a way to record all the grades in the course. We chose a dictionary of dictionaries (organized first by student, then by course element), but client code shouldn't have to traverse this structure – that's the job of your class. A programmer who writes client code shouldn't even have to know which structure you chose. In fact, if client code always accesses data through your public methods, you have the freedom to change internal details without breaking any client code.

   An internal attribute is not part of the public interface of the class, and its name should begin with an underscore to indicate this.

   Create a class comment (using the hash symbol) below the class docstring. For each internal attribute, use the same format as above to define the **type**, **name**, and **description** of each of the internal attributes.

```
# === Private Attributes ===
# @type _scheme: dict[str, int]
#     The marking scheme for this course.  Each key is an element of the
#     course, and its value is the weight of that element towards the
#     course grade.
# @type _grades: dict[str, dict[str, int]]
#     The grades earned so far in this course.  Each key is a student
#     ID and its value is a dict mapping a course element to the student's
#     mark on that element.  If a student did not submit that element,
#     it does not appear as a key in the student's dict.  If, however,
#     they earned a grade of zero, this is recorded as any other grade
#     would be.
```

6. **Representation invariants.** Add a section to your internal class comment containing "invariants" that involve your attributes: things that must always be true (one could say they must never "vary" from truth, hence the name "invariant"). These may be restrictions that cannot be captured by types alone in Python. For example, a student's 'age' must be greater than 0, or every course code must consist of 3 letters and then 3 numbers. They may also express important relationships between the attributes.

```
# === Representation Invariants ===
# - The sum of all weights in self._scheme must be 100.
# - Each key in every student's dict of grades must be an element of the
#   course grading scheme, i.e., must occur as a key in self._scheme.
```

7. **Implement Public Methods.** Use the last two steps of the function design recipe to implement the public methods in your class.

   (5) Code the Body

   (6) Test your Method

   Use helper methods to simplify your code. A helper method is not part of the public interface of the class, and its name should begin with an *underscore* to indicate this.

   For each method, you should assume that the representation invariants are all satisfied when the method is called, but **you must ensure that the invariants are satisfied when the method exits**.

   Note that your constructor should initialize *all* of the attributes of the instance; it should not do anything else. You can find the complete implementation in the full code example.

```
def __init__(self, name, cap, scheme):
    """Initialize this course.

    @type self: Course
    @type name: str
        The name of this course.
    @type cap: int
        The enrolment cap for this course.
    @type scheme: dict[str, int]
        The marking scheme for this course.  Each key is an element of the
        course, and its value is the weight of that element towards the
        course grade.
        Precondition: The sum of all weights must be 100.
    @rtype: None
```

```
        >>> c = Course('cscFun', 50, {'exam': 100})
        >>> c.name
        'cscFun'
        >>> c.cap
        50
        """
        self.name = name
        self.cap = cap
        self._scheme = scheme
        self._grades = {}

    def enrol(self, student_id):
        """Enrol a student in this course.

        Enrol the student with id <student_id> in this course, if there is
        room.

        @type self: Course
        @type student_id: str
        @rtype: bool
            True iff enrolment was successful, i.e., this student was not
            already enrolled, and there was room for to enrol him or her.

        >>> c = Course('cscFun', 50, {'exam': 100})
        >>> c.enrol('12345')
        True
        >>> c.grade('12345', 'exam') is None
        True
        """
        if len(self._grades) < self.cap:
            if student_id in self._grades:
                return False
            else:
                self._grades[student_id] = {}
                return True
        else:
            return False
```

Notice that method `__init__` does not confirm that its precondition for parameter `scheme` is met. The same is true of other methods with preconditions. A method simply assumes that its preconditions are true, and makes no promises about what will happen if they are not.

Summary

Part 1: Define the API for the class

1. **Class name and description.** Pick a noun to be the name of the class, write a one-line summary of what that class represents, and (optionally) write a longer, more detailed description.
2. **Example.** Write some simple examples of client code that uses your class.
3. **Public methods.** Decide what services your class should provide. For each, define the API for a method that will provide the action. Use the first four steps of the Function Design Recipe to do so:
   (1) Example
   (2) Type Contract
   (3) Header
   (4) Description
4. **Public attributes.** Decide what data you would like client code to be able to access without calling a method. Add a section to your class docstring after the longer description, specifying the **type**, **name**, and **description** of each of these attributes.

Part 2: Implement the class

5. **Internal attributes.** Define the **type**, **name**, and **description** of each of the internal attributes. Put this in a class comment (using the hash symbol) below the class docstring.
6. **Representation invariants.** Add a section to your internal class comment containing any representation invariants.
7. **Public methods.** Use the last two steps of the function design recipe to implement all of the methods:
   (5) Code the Body
   (6) Test your Method