CSE231 - Lab 10

Intro to Classes

A reminder to… read

I wish I didn’t have to bring this up, but yes -- I’m going to remind you all to read. The book, of course (that’s a given), but also the syllabus and our exam instructions.

We have found **many** students that checked their multiple choice answers in the coding portion of the exam. This was *explicitly stated* as being against the rules when opening the assignment on Mimir. How did we find out? Surprise! We can see everything you’ve written inside the code editors, from the moment you began typing, up until your final answer.

If we found you checking a multiple choice answer in Mimir’s code editor, you received a 0% for the coding question that you were checking.

A reminder to… read (cont.)

With S/NS grading coming back for this semester, the professors also wanted to remind everyone that we have a course grading policy that is often overlooked when students believe they can coast for the rest of the semester…

“To be eligible to earn a non-zero grade in the course, a student normally must earn at least 50% of the total points for the examinations and earn at least 50% of the total points for the computer projects. If you do not take the final exam, your course grade will be a zero.”

You can verify this quote for yourself by visiting the syllabus, under “Course Grades”.

The Final Weeks

Classes bring together everything that we’ve done so far. If you don’t have a thorough understanding of the content leading up to this point, **you will** have to catch up a little. The remaining time we have in this course will *only* be on classes. They’re hugely relevant to programming in today’s world, and it can be a difficult thing to wrap your head around at first.

For today, we’re introducing the *concept* of classes, but we’re not going to be making one of our own until next week.

We’ll be working with another, pre-made class that you’ll be using for an upcoming project. Before I explain what it is, we first need to discuss classes and the motive behind them.

What is a class?

Classes are programmatic representations of some entity. What this entity represents, and what it can do, is decided by the programmer.

We might want to think of something like a car, where that car contains features/attributes that we can manipulate and enact “functions” on.

- Car
  - Manufacturer: Tesla
  - Model: Model 3
  - Range: 423 km
  - Functionality: `accelerate()`, `decelerate()`, `steer()`, `recharge()`

(I know absolutely nothing about cars, please don’t ask me questions regarding cars)

What is a class? (cont.)

We’ve secretly been using classes this entire time. Dictionaries, tuples, strings, ints, floats -- all of these are classes.

```
print(type(“Hello World!”))    # <class ‘str’>
```

It’s important to note here that “types”, as we’ve been referring to them traditionally, are synonymous with “classes”. The string we made above is an **instance** of a class, called ‘`str`’, that holds a value: `“Hello World!”`

Classes act as templates for the kind of data you want to store. When you create an object from the class, it’s referred to as an **instance**, which is why you may have heard me refer to things as “instances” in past lectures. Classes are helpful to us because we can have some manipulations handled by the instantiation, and some manipulations handled by code that *uses* the instantiation.

Class Behavior

What do I mean by “have some manipulations handled by the instantiation”? Well, remember back to the method functions you’ve invoked on other classes:

Lists have `.sort()`, `.pop()`, `.append()`...

Dictionaries have `.keys()`, `.values()`, `.items()`...

These method functions are *dedicated* to manipulating data being held by the instantiation. We don’t want to come up with code that will sort a list for us, we want to call onto the list with its already-implemented `.sort()` method.

This often raises the question: “how do I know when I should make a class?”. The answer is: *usually* when you have many different things that could be grouped under the same name.

The `Card` and `Deck` Classes

Dr. Enbody really likes card games. And so, you’ll be working with two classes that simulate playing cards and a typical deck.

Again, I want to emphasize that we’re **not** making them, we’re **using** them.

For those unaware (I always get at least 2 students that don’t know how cards work): playing cards are thin pieces of card stock with each labeled with one of four “suits” (a club, diamond, heart or spade), and one out of thirteen ranks (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King). There is a card for each combination of features in a typical deck, which means there are 4 * 13 = 52 cards in total.

`Card` Methods

`.rank()` - Returns the rank of the card (type: `int`).

`.suit()` - Returns the suit of the card (type: `int`).

`.value()` - Returns the value of the card (type: `int`) (in many games, it’s 1 for Ace, 2-9 for 2-9, and 10 for Jack, Queen or King (the “face” cards)).

`.is_face_up()` - Returns `True` if the card is facing up, else `False` (in many games, you might flip cards as a part of the objective).

`.flip_card()` - Flips the card, returns None.

Deck Methods

`.shuffle()` - Randomly shuffles the deck, returns None.

`.deal()` - Returns the top `Card` from the deck, returns None if deck is empty.

`.is_empty()` - Returns `True` if the deck is empty, else `False`.

`.display(cols=13)` - Prints the deck in a row-column table format. `cols` argument specifies the number of columns to produce, returns None.

L10-1, L10-2