Day 1

“Classes” allow us to define our own types. For example, consider the following class for an employee, which uses the `datetime` package:

```python
from datetime import *

class Employee:
    def __init__(self, name):
        self.name = name
        print 'New hire! Employee name:', self.name

    def clock_in(self, start_time):
        self.clocked_in_at = start_time
        print self.name, 'just clocked in at', self.clocked_in_at

    def clock_out(self, end_time):
        duration = end_time - self.clocked_in_at
        print self.name, 'just clocked out after working', duration, 'hours.'

oliver = Employee('Oliver Serang')
# Oliver works on Halloween:
# start work at 9am
oliver.clock_in( datetime(month=10, day=31, year=2018, hour=9) )
# end work at 6pm
oliver.clock_out( datetime(month=10, day=31, year=2018, hour=18) )
print

thomas = Employee('Thomas Winkler')
# Thomas works on Sunday before Halloween:
thomas.clock_in( datetime(month=10, day=28, year=2018, hour=12) )
thomas.clock_out( datetime(month=10, day=28, year=2018, hour=17) )
```

Note that everything we declare as `self.something = ...` will attach a variable to our class (which we can lookup later via `self.something`). These variables are called “member variables”. Classes are useful because we can
use them to store the state of an object (instead of sending many variables into a function). They behave as a gestalt.

Functions that we define in the class can be called via `my_obj.do_something()` instead of `do_something(my_object)`. These functions are called member functions. They can have additional external arguments, but they do not require them. Regardless, we always include `self` argument, the object itself, to these functions.

There are also special member functions that use underscores in the names. For example, the `__init__` member function is called a “constructor” and is used to create a new object. Likewise, the `__str__` member function converts the object to a string and returns that string. Like `__init__`, `__str__` must accept `self` as an argument so that it has access to the object.

1. Update the Employee class so that the employee records the total number of shifts that they have worked in their lifetime.

2. Print an Employee object. What does it look like?

3. Update the Employee class further so that there is a `__str__` member function, which returns a string that contains the employee’s name and their work history (i.e., the number of shifts they have worked since they were hired). Call `str(...)` on the employee and print it.

4. Print an Employee object again. What does the output look like now?

5. Add a new member variable, `is_clocked_in`, which keeps track of whether the employee is currently working. Modify the `clock_in` and `clock_out` functions so that the employee cannot clock in if they are currently working and cannot clock out if they are not currently working.

6. Add a new member variable, `total_number_of_hours_worked`, which is initialized as 0 in the constructor and accumulates hours every time the player clocks out. Add a new member function, `get_hours_worked`, which returns the number of cumulative hours that this employee has worked in their career.
Day 2

1. Consider the `count_objects` function that you created in your poker problem. Create a class called `Multiset`. This object will contain a single member variable, called `item_to_count`, which will be a dictionary. Define two member function for `Multiset`: The first member function, `add_item`, will take one argument (an item) and put it into `item_to_count`, increasing the total count for that item. The second member function, `get_count` will accept one argument (an item) and return the number of times it has occurred.