Project 5: Payroll (Part 2)

CS 1410

Background

Look at the spec in Part 1 for background information.

Requirements (Part 2)

Implement your design From Part 1, including the five module-level functions used in the **main** function in *p5.py* by the final due date. Use *p5.py* verbatim as your main module.

After running your program, compare the three new entries in *paylog.txt* to *paylog_old.txt* that was copied earlier in **main**, and verify that the pay amount has been changed appropriately for the three employees updated in **main**. Submit your *payroll.py*, *paylog_old.txt*, and *paylog.txt* output files.

Implementation Notes

Every time **run_payroll** executes, it first deletes any previous *payroll.txt* file, if it exists (the main program in *p5.py* saves it for you). Here is **run_payroll** (you can just use it):

```
def run_payroll():
    if os.path.exists(PAY_LOGFILE): # pay_log_file is a global variable holding 'payroll.txt'
        os.remove(PAY_LOGFILE)
    for emp in employees:               # employees is the global list of Employee objects
        emp.issue_payment()             # issue_payment calls a method in the classification
                                        # object to compute the pay
```

Every time you issue the payment for an Hourly or Commissioned employee, clear their respective timecard or receipt list afterward, so these entries won't be used again for the next pay period.

After you have run the main module in the file *p5.py*, *payroll.txt* should contain:

```
Mailing 5599.44 to Issie Scholard at 11 Texas Court Columbia Missouri 65218
Mailing 2584.64 to Reynard Lorenzin at 3233 Spaight Point Houston Texas 77030
Mailing 1880.00 to Jed Netti at 85 Coolidge Terrace San Antonio Texas 78255
```

The entries above indicate the format you should use for printing payroll results to *payroll.txt*

Here is a suggested development sequence:

1. Write **load_employees**. It opens *employees.csv*, ignores the first line, and then reads a line at a time, splitting its arguments on a comma. Create a new Employee object initialized with the string attributes. Then create the appropriate instances for the employee's classification and add it as an attribute to the new Employee object. Finally, add the Employee object to your global list of employees.
2. Write **find_employee_by_id** by searching the list of employees and returning the Employee object.
3. Implement **Employee**
4. Implement the **Classification** Hierarchy
5. Implement **process_timecards**
6. Implement **process_receipts**

FAQs

Q. I'm confused on how read the employee data and create employee objects in **load_employees**?
A. Really? What exactly is confusing to you?

Q. Well, not all of the data in each line applies to the employee.
A. True. You know the data common to all employees. So, create the Employee object with just that common data. Have your Employee constructor only take those parameters. Then you can add the correct classification object afterwards.

Q. How do we create the employee's classification?
A. Depending on the type code of the employee in the file, call either **make_hourly**, **make_salaried**, or **make_commissioned**, passing it the corresponding data from the file. The **make**_-methods in Employee create the correct type of concrete classification object and attach it as an attribute of the Employee object (**self**).

Q. Why are you so smart?
A. That's nice of you to say, but I'm not, actually. I just have a lot of experience. I'm trying to impart some of that to you.

Q. Why do we need the Classification class, anyway? My program works without it.
A. In this case, it is true that the Classification class is not needed for the program to run correctly. But that is because there isn't much to put in classification: just a **compute_pay** abstract method. If there were something more in common to all the classification types, like shared data, it would need to go in Classification to avoid duplication. But even so, it is good to show *in code* that the classification classes are *related*, and that they must implement **compute_pay**. Using Classification as an abstract class enforces this and expresses the design nicely as well.

Q. How do I round pay to 2 decimals?
A. Use the **round** built-in function in **compute_pay** before you return the amount.

Q. How do I avoid paying employees twice for the same hours or sales?
A. You must have missed that above in the spec. Clear the list of hours or receipts in **compute_pay** before you return the payment amount.

Q. Any other hints?
A. No. You can do this.