Lab Exercise 4

Learning Outcomes Covered

- Using inheritance in OOP
- Basic file I/O
- Exception handling
- System commands

Instructions

- Unless specified otherwise,
    - you are allowed to use anything in the Python Standard Library;
    - you are NOT allowed to use any third-party library in your code;
    - you can assume all user inputs are always valid and your program for each question need not include code to validate them.
- The underlined blue texts in the Sample Runs section of some questions refer to the user inputs. It does NOT mean that your program needs to underline them.
- **Please name each of your Python script files exactly as the name in parenthesis as shown in each question heading. Marks will be deducted if violating this instruction.**

Part I: In-Class Exercise (Deadline: Feb 24, 11:59pm)

Question 1: Inheritance (q1.py) (20%)

Write two classes named Person and Programmer so that the following client code runs. Note that the Programmer class must inherit from the Person class, which maintains two attributes, name and age. Programmer has an extra attribute, skillset (a Python list), to describe the skills (e.g. C++, Java, Linux) the programmer acquired. The methods add_skill(s), remove_skill(s) and has_skill(s) add s to, remove s from, and check existence of s in the skillset list, respectively.

Sample Usage:

```
mike = Person("Mike", 21)
sheila = Programmer("Sheila", 23)
sheila.add_skill("C++")
sheila.add_skill("Python")
sheila.add_skill("Linux")
sheila.remove_skill("Python")

print(mike.name, mike.age)          # Mike 21
print(sheila.name, sheila.age)      # Sheila 23

print(isinstance(sheila, Person))   # True
print(isinstance(mike, Programmer)) # False

print(sheila.has_skill("C++"))      # True
print(sheila.has_skill("Python"))   # False
print(sheila.has_skill("Java"))     # False
print(sheila.has_skill("Linux"))    # True
```

Part II : Take-home Exercise (Deadline: Mar 7, 11:59pm)

Question 2: Inheritance (q2.py) (20%)

Write a small game which consists of two characters, heroes and beasts. Every character has three attributes, namely `name`, `AP` (attack points) and `HP` (health points). A hero attacks a beast and vice versa. The character being attacked will drop in HP by an amount equal to the attacker's AP.

To implement this game, your program defines the following three classes:

```
class Character(ABC):
    # Your code goes here

class Hero(Character):
    # Your code goes here

class Beast(Character):
    # Your code goes here
```

Note that `Character` is an *abstract base class (ABC)* with an *abstract method* `attack()`, which must be overridden by every subclass. For `Character` objects `a` and `b`, the call `a.attack(b)` means that `a` attacks `b`. The game ends when a character's HP drops to or below zero.

	Hero	**Beast**
Default HP:	100	70
Default AP:	20	25
Attack behavior:	print "Punching <enemy name> ..."	print "Biting <enemy name> ..."

Your code must pass the sample test as follows:

Sample Client:

```
try:
    c = Character("Dummy", 10, 200)  # with HP = 200, AP = 10
except TypeError:
    print("Can't instantiate abstract class Character")

hero = Hero('Spider-Man')     # with HP = 100, AP = 20
beast = Beast('Two-Headed')   # with HP = 70, AP = 25

while True:
    if hero.HP <= 0:
        break
    hero.attack(beast)
    if beast.HP <= 0:
        break
    beast.attack(hero)

hero.print_info()
beast.print_info()
print((beast.name if beast.HP <= 0 else hero.name) + " is dead!")
```

Sample Output:

```
Can't instantiate abstract class Character
Punching Two-Headed ...
Biting Spider-Man ...
Punching Two-Headed ...
Biting Spider-Man ...
Punching Two-Headed ...
Biting Spider-Man ...
Punching Two-Headed ...
Spider-Man: AP = 20, HP = 25
Two-Headed: AP = 25, HP = -10
Two-Headed is dead!
```

Question 3: File I/O (q3.py) (20%)

Given the file names of two text files, which contains one integer per line. Write a function called `merge_and_sort()` which merges the contents of the two files, sort all the integers in ascending order, and output the result as a new file. The function's call signature is given as follows:

```python
def merge_and_sort(input_file_A, input_file_B, output_file):
    # Your code goes here

merge_and_sort("a.txt", "b.txt", "c.txt") # Creates c.txt
```

For example, suppose that file a.txt contains

```
6
1
3
```

and file b.txt contains

```
2
4
5
```

The above function call will create a new file c.txt which contains:

```
1
2
3
4
5
6
```

Normally, all the files are accessible in the current directory (containing the Python script) and they should store one integer per line. However, there is chance that these assumptions are violated. Your code must include the following exception handling when reading an input file:

- If the input file does not exist or the user has no permission to access it, print the error message "File access error!".
- If some line(s) in the file is not an integer, print error message “Invalid input!”

In either case, no output file will be generated.

When creating the output file, you may assume that no exception handling is needed (e.g., it is always given a different name from the input files; there is always sufficient disk space for its creation).

Question 4: File I/O (q4.py) (20%)

There is a database log file named `attendance.txt`, which records students’ course attendance information formatted as follows:

```
('2020-01-13 11:50:09', 271, 131),
('2020-01-14 10:52:19', 273, 131),
('2020-01-13 11:50:19', 271, 126)
```

Every line is one record of attendance formatted as (time, course id, student id).

Write a function called `reformat_as_dict()`, which takes filename as parameter and returns a dictionary with student id as key and a list of reformatted attendance records as value. A reformatted attendance record is a dictionary with two entries, course id and attendance time.

```python
def reformat_as_dict(filename):
    # Your code goes here
```

For example, for the three records above, the function returns the following dictionary:

```
{131: [{'courseid': 271, 'attendtime': '2020-01-13 11:50:09'}, {'courseid':
273,'attendtime':'2020-01-14 10:52:19'}], 126: [{'courseid': 271, 'attendtime':
'2020-01-13 11:50:19'}]}
```

The above output is hard to read. So, for the final output of this program, it dumps the dictionary as a JSON file using the following code. Suppose that `data_dict` is the dictionary returned from the call of `reformat_as_dict("attendance.txt")`.

```python
import json

jstr = json.dumps(data_dict, indent=4, sort_keys=True)
with open("attendance.json", "w") as f:
    f.write(jstr)
```

This code snippet will produce an output file called `attendance.json` which contains the following neatly indented data (also sorted by key) in JSON format:

```
{
    "126": [
        {
            "attendtime": "2020-01-13 11:50:19",
            "courseid": 271
        }
    ],
    "131": [
        {
            "attendtime": "2020-01-13 11:50:09",
            "courseid": 271,
        },
        {
            "attendtime": "2020-01-14 10:52:19",
            "courseid": 273
        }
    ]
}
```

Note: the provided input file `attendance.txt` has many more records than the simplified example above. We have provided the expected output `attendance.json` to ease your program correctness check. You may make use of some text comparison tools like https://www.diffchecker.com or https://text-compare.com to compare your output file content against the provided one.

Question 5: OS Interface (q5.py) (20%)

Write a program which lets the user enter a directory path via console input and finds all the Python script files (i.e., files with file extension ".py") under the specified directory (including all its subfolders, recursively) and creates a text file named `python_files.txt` to store the relative paths of all the files found.

Hint: import the os module and use `os.walk()` to walk the directory tree at the specified path. Check the documentation of this method to see how to use it.

We have provided a sample directory whose content is depicted as follows:

```
q5_dir
├── Sheila
│   ├── Python
│   │   └── hello_world.py
│   ├── hello_world.cpp
│   └── q_learning.py
├── dqn.py
├── ppo.py
└── sum.cpp
```

Put this directory next to your q5.py script file before running your program.

Sample Usage (on Linux / macOS):

Example 1:

```
Input path to search: q5_dir
```

python_files.txt contains:

```
q5_dir/dqn.py
q5_dir/ppo.py
q5_dir/Sheila/q_learning.py
q5_dir/Sheila/Python/hello_world.py
```

Example 2:
Suppose that the sample directory is put under /Users/sheilazheng.

```
Input path to search: /Users/sheilazheng/q5_dir
```

python_files.txt contains:

```
/Users/sheilazheng/q5_dir/dqn.py
/Users/sheilazheng/q5_dir/ppo.py
/Users/sheilazheng/q5_dir/Sheila/q_learning.py
/Users/sheilazheng/q5_dir/Sheila/Python/hello_world.py
```

Example 3:

```
Input path to search: q5_dir/Sheila
```

python_files.txt contains:

```
q5_dir/Sheila/q_learning.py
q5_dir/Sheila/Python/hello_world.py
```

Example 4:

```
Input path to search: a non-existent dir
```

An empty file python_files.txt is generated.

Sample Usage (on Windows):

On a Windows platform, the file separator becomes "\" instead of "/". Below we repeat Example 1 and Example 2 when running the Python program on a Windows platform:

Example 1:

```
Input path to search: q5_dir
```

`python_files.txt` contains:

```
q5_dir\dqn.py
q5_dir\ppo.py
q5_dir\Sheila\q_learning.py
q5_dir\Sheila\Python\hello_world.py
```

Example 2:
Suppose that the sample directory is put under C:\Users\sheilazheng.

```
Input path to search: C:\Users\sheilazheng\q5_dir
```

`python_files.txt` contains:

```
C:\Users\sheilazheng\q5_dir\dqn.py
C:\Users\sheilazheng\q5_dir\ppo.py
C:\Users\sheilazheng\q5_dir\Sheila\q_learning.py
C:\Users\sheilazheng\q5_dir\Sheila\Python\hello_world.py
```

More Notes:

Hint: os.path.sep may be useful for writing platform-independent code. It returns a right pathname separator on a specific OS platform.

The appearance order of the file paths in your output file need not follow exactly the same as that in our sample output as shown.