Lab 1: OOP Practice

Object-Oriented Programming is a *really* big part of implementing data structures, so it's important that we practice before our first Project.

Part One: A very simple class

This class will be simple in functionality, but will serve as an example for class structure. In a file called `car.py`, make a class called `Car`. This class should have:

- A constructor (sometimes called an initializer in Python), which takes two string arguments: the make and model of the car. It should store these in the class as fields (you might know fields as "self" variables). Additionally, it should initialize a field "odometer", representing how many miles the car has driven, to 0.
- A function called `drive`, which takes an integer representing how many miles the car has driven during a new trip and adds that to the odometer field.
- A function called `getMileage` that takes no arguments and returns the number of miles on the odometer.

Copy-paste the following code into a file called `lab1Test.py`, in the same directory as `car.py`. Running `python3 lab1Test.py` should run your class correctly, and print the number 70:

```
from car import Car
myCar = Car("Nissan", "Leaf")
myCar.drive(35) #Me driving to work
myCar.drive(35) #Me driving home
print(myCar.getMileage())
```

Part Two: The `__str__` Method

Suppose we create a class called `AClass`, and make an object of that type, like so: `anObject = AClass()`. We then call `print(anObject)`. What will print?

A right answer here is "I have no idea." The reason you have no idea is because Python doesn't know what data is stored within this object, so how can it possibly know what the right way to print it to the screen is?

Fortunately, there's a way of telling Python how to print things: the `__str__` function (https://docs.python.org/3/reference/datamodel.html#object.__str__) (special functions in Python usually start and end with double-underscores, to avoid accidental name collisions). Whenever an object needs to be printed, or otherwise converted to a string, Python goes looking for that object's `__str__` function; if it exists, it runs that function, and that function returns a string (**it does not print the string, it just returns it**); the returned string is then printed, or used as the string representation.

Add a `__str__` method to your `Car` class, so that when an object of that type is printed, you get the make and model of the car. For instance, if you add the line "print(myCar)" to the code above, it should output:

```
Nissan Leaf
```

Part Three: Keeping Track of Network Resources

We're going to make a little program to keep track of which processes are running on a machine and what their memory requirements are. Here's a description of the objects involved. A Process is an object which contains a name, and an int which indicates the number of KB of memory required to run that Process. A Machine is an object with a name, a list of Processes running on that Machine, and an int indicating the maximum amount of available memory in KB.

Description of Classes

Place both of these classes in a single file named `machine.py`. You can download a testing file here (labs/testIt.txt), which can help you test after finishing each class.

Note that there is code in that file to test the Machine and Network classes as well as the Proces class, but it is commented out so that you can incrementally test your lab as you are writing it. When you want to test Machine and Network, uncomment those portions of the test file.
Process has two fields, `name` (a String) and `memoryReq` (an int). As far as methods, it should have:

- a constructor (sometimes called an initializer), which takes both a String and an int, in that order, and
- `__str__`, which returns a string of the form `processName: memoryReqKB` (for example, "aName: 5KB")

Machine has three fields, `name` (a String), `processList` (a list of Processes running on that machine), and `totalMem` (an int). For methods, it should have:

- a constructor, which takes a name and totalMem as arguments, and sets `processList` to be an empty list,
- `addProcess`, which takes a Process as an argument, and appends it to `processList`,
- `availableMemory`, which takes no arguments (aside from `self`), and returns how much memory the machine has free. To do this, subtract the amount of memory used by each process from the machine's total memory.
- `__str__`, which returns a string describing the machine of the form

```
machineName, totalMemKB
  aProcessName: someKB
  anotherProcessName: someOtherKB
```

Network has as a field a list of Machines named `machines`. For methods, it should have:

- a constructor, which takes no arguments (again, aside from `self`), and sets `machines` to be an empty list,
- `addMachine`, which takes a Machine as an argument, and appends it to the list `machines`,
- `addProcess`, which takes a Process as an argument, figures out which Machine in `machines` has the most available memory (by calling their `availableMemory` method), and assigned the Process to that Machine (using its `addProcess` method), and
- `__str__`, which returns a string describing the entire network, of the form:

```
Network:
machineName, totalMemKB
  aProcessName: someKB
  anotherProcessName: someOtherKB
anotherMachineName, totalMemKB
  aProcessName: someKB
  anotherProcessName: someOtherKB
```

Grading

- Car: 40%
  - Constructor: 10%
  - drive: 10%
  - getMileage: 10%
  - __str__: 10:
- Process: 10%
  - Constructor: 5%
  - __str__: 5%
- Machine: 25%
  - Constructor: 5%
  - addProcess: 5%
  - availableMemory: 10%
  - __str__: 5%
- Network: 25%
  - Constructor: 5%
  - addProcess: 10%
  - addMachine: 5%
  - __str__: 5%