1 Overview

In Project 3, we’ll be working more extensively with classes. I’m a railfan, so I’ve constructed a project about trains. In this project, you will have three classes: Locomotive, Wagon[1] and Train; each will have some required methods.

As before, we’ll use test files - with a series of commands - as our input, and we’ll print out text showing the results of various commands. But this time, you won’t have to write the parser: I have provided a module which will run the testcase for you. You will write a second file, which my code will import.

NOTE: Please see the end of the spec for an updated procedure about how to submit files to D2L.

1.1 Imports

Writing a module in Python is easy: simply create a .py file, put it in the same directory as a second file, and then use import to import the first into the second.

```
------------
useful.py:
------------
def foo():
    ...

class Bar:
    ...

------------
needy.py:
------------
import useful

    x = useful.foo()
    obj = useful.Bar()
```

[1]In the US, we normally call the cargo parts of a train “cars.” In Europe, they call them “wagons.” I normally prefer the US terminology - but it gets a little confusing in a program. So I’m using the European terminology for this project.
In this project, you will provide the import file (which you will name `trains.py`), and I will provide main program, in a file named `proj03_main.py`.

### 1.2 Style

Remember to read the class [http://www2.cs.arizona.edu/classes/cs120/summer17/style/pgm-style.html](http://www2.cs.arizona.edu/classes/cs120/summer17/style/pgm-style.html).

### 1.3 Short and Long Problems

This week, you will have “short problems” (due on Monday), which you will complete on CloudCoder ([https://practice.cs.arizona.edu](https://practice.cs.arizona.edu)). These are auto-graded for correctness, and the TAs will give you, by Wednesday, feedback on your style and design. The short problems will typically either be small parts of - or practice for - the long programs, which you will turn in through D2L, which are due on Friday.

### 1.4 Checking Your Output

Write the program exactly as described below. Test it using IDLE or something equivalent (make sure that you are using Python 3).

Your output should match the required output exactly, because we will be using a script to check it for correctness. We recommend that you use [http://www.diffchecker.com](http://www.diffchecker.com) to confirm that the output from your program matches the required output exactly!

**NOTE:** The “required output” includes everything that you print, including the prompts (if any) for user input. Pay attention to the requirements: if a prompt is required, print it - but if none is required, don’t add one!

### 1.5 main()

In this project, I have provided a `main()` function - it’s part of `proj03_main.py`, which I’ve provided for you. While you don’t need to write it or edit it this time, you should take a look how I wrote it. I’ll probably require you to write your own `main()` function in the next Project!

### 2 The Input/Output Format

In a perfect world, I’d love to have the testcases simply be Python files - then I wouldn’t have to write an interpreter at all. However, I can’t have you writing unknown Python code and running it on my Google App Engine site.

So instead, I’ve come up with a simplified testcase file that **looks** a lot like Python, but is very, very limited. In this project, you can create new objects,
and assign them to variables. (However, my code will not allow you to ever change a variable once it’s assigned.)

```python
foo = Locomotive(100, "Brutus")
bar = Wagon(50, "gondola", "coal", 10)
```

In this project, you can also call methods on objects you have created. You can (of course) only call the methods that I’ve defined in this spec. Most of these methods take parameters, which I will pass on to your code. For most of these methods, if they return something, my code will print that out. However, the `split_at(x)` method is special: since it breaks a train into two pieces, you must assign it to a new variable as well.

```python
baz = Train()
baz.add_locomotive(foo)
baz.add_wagon(bar)
tail = baz.split_at(1)
```

For functions that require strings (such as the `__init__` function for `Wagon`), my code will only support double-quoted strings, like the example above. (My code doesn’t support zero-length strings, or strings with backslashes. But otherwise, it just picks up whatever you type, from one double-quote to another.)

For functions that require numeric parameters, my code will only support non-negative integers - and not mathematical operators will be allowed.

My input format will ignore whitespace (including indentation). It will also recognize comments, which start with `#`, just like in Python.

Finally, my input format supports `print()` calls. You can pass `print()` zero arguments (to print a new line), or any number of arguments, separated by commas. However, the arguments can only be strings or numbers - no variables or method calls.

## 3 The Classes and Their Methods

As noted above, you must declare three classes:

- **Locomotive** models a locomotive. It has a length (an integer), and a name (a string).

- **Wagon** models a cargo wagon. Like a locomotive, it has a length. The second parameter to the `__init__` function is a “type” (a string), which tells what sort of car it is.³

³You are welcome to invent crazy car types for your testcases, but if you’re interested, some of the most common types include:

- **Well Car**
- ** Autorack**
- ** Tank Car**
- ** Gondola**
The third and fourth parameters are about the cargo: the third (a string) says what type of cargo it is, and the fourth (an integer) is how much. You can use any strings for these cargo types, except that “empty” is special: an empty car must have (‘’empty’’,0).

• **Train** models a complete train. It starts out empty (no locomotives or wagons). You can then add locomotives and wagons, one at a time. New locomotives are always added at the front, and new wagons are always added at the end. This class has a variety of methods which allow you to query the contents of the objects.

3.1 Encapsulation

In all three classes, **all data fields must be private**; their names should have a single leading underscore, and you should never touch another object’s private data. Instead, access all of these fields through methods. This will certainly mean that you will need to write some gettor methods; if you wish, you may also write some settor methods - although whether you do this or not is up to you.

Every class must define a **_str_** method, which will print out the fields of that object in a standard format (see each class below for the format).

Each class will have certain required methods, listed below. You may add additional methods, if you would like. Private helper methods should have names that start with a leading underscore; public methods should not. All methods need docstrings.

3.2 class Locomotive

Models a locomotive. Required methods:

• **Locomotive._init_(length, name)**

• **Locomotive._str_()** - returns a string, with the class name, length, and the locomotive name. The name should be surrounded by double-quotes. The overall format is as follows:

Locomotive "Brutus" 100 ft

• Gettors:

  Locomotive.get_length()
  Locomotive.get_name()

| Flatcar |
| Boxcar |

4Exception: If you have two objects of the same type - such as two trains - it is OK if you want to access the private fields of one object from another object. This is a common exception in many OOP languages.

5The descriptions below all skip the self parameter. You will need to add it, of course!
3.3 class Wagon
Models a cargo wagon. Required methods:

- Wagon.__init__(length, wagon_type, cargo_type, cargo_amt)
- Wagon.__str__() - returns a string, with the class name and the various parameters, formatted as follows:
  Wagon(gondola, 50 ft): 10 coal
  (Note that this format - unlike Locomotive - does not have any double-quotes. Also note that the parameters are not in the same order as in the init function.)

- Gettors:
  Wagon.get_length()
  Wagon.get_wagon_type()
  Wagon.get_cargo_type()
  Wagon.get_cargo_amt()

- Wagon.unload() - removes all cargo from the wagon. The cargo type should be set to ‘empty’ and the amount to 0.
  Returns the total quantity unloaded (could be 0).

- Wagon.load(cargo_type, amount) - adds new cargo to the wagon. There are three possible results. (1) If the wagon was empty, this sets the cargo type and amount. (2) If the wagon was not empty but the cargo types are the same, then add to the amount. (3) If the current cargo type doesn’t match, reject the load.
  This function will not print anything (even in case 3, where the load is rejected). Instead, it returns a boolean: True if the cargo was loaded, and False if not.

  NOTE: Since my little interpreter doesn’t support negative numbers, you don’t have to worry about amount being negative.

3.4 class Train
Models a complete train. When the object is first created, it is empty (no locomotives and no wagons). Locomotives and wagons are added later.

The split_at(x) method splits the train into two pieces, creating a new train. Normally, the new train will have some wagons (and occasionally some locomotives as well) - however, occasionally it may be empty.

Required methods:
• Train.__init__()\footnote{In the split_at(x) method, you will create a new train. I expect most students to create an empty train, and then use for() loops to add wagons and/or locomotives one at a time. However, if you know how to use default arguments in Python, you may add default arguments to the init method to make things a little easier.}

• Train.__str__() - returns a string, with the class name and the various parameters, formatted as follows:

Train: 3 locomotives 10 wagons

(Note that this does not contain all of the information about the train. It’s just a quick summary.)

• Gettors:
  Train.get_locomotives()
  Train.get_wagons()

  Returns the current locomotives or wagons in the train, as a list. The order of the list must exactly match the order of the locomotives and wagons in the train.

• Train.add_locomotive(loco) - adds a new locomotive to the front of the train, before any other locomotives.

  NOTE: In a real program, we would want to ensure that no locomotive was ever added to multiple trains. We’ll ignore that for this Project. But it’s a good thing to think about - how could you add this feature?

• Train.add_wagon(w) - adds a new wagon to the back of the train, after any other wagons.

• Train.get_length() - returns the total length of the train. Note that this is the total of all of the lengths (of both locomotives and wagons). It is not the number of locomotives and wagons.

• Train.get_cargo() - returns a dictionary, listing the total cargo on the train. Each key in the dictionary is a cargo type, and the matching integer (never zero!) is the total amount of that cargo, on all wagons. The dictionary must not include the ‘empty’ key. If all wagons are empty, then the dictionary should be empty.

• Train.unload_by_type(cargo_type) - unloads all wagons that are carrying cargo of this type.

  Returns the total quantity unloaded.

• Train.unload_all() - unloads all cargo from all wagons.

  Returns nothing.
• **Train.fill(cargo_type, cargo_amount, max_per_wagon)** - Attempts to load cargo onto the train. Starting at the first wagon, it will attempt to load each wagon up to `max_per_wagon`. (If the wagon already has more than that - or if has a different type of cargo, then that wagon is not affected.)

Returns the total amount loaded (could be zero).

**NOTE:** Since my little interpreter doesn’t support negative numbers, you don’t have to worry about `amount` being negative.

• **split_at(pos)** - splits the **Train** at a certain position, returning a new **Train** which represents the back part of the train. (Make sure to remove those wagons (and perhaps locomotives) from the current train as well.)

The first locomotive is position 0 (or, if there are no locomotives, the first wagon is position 0). Thus, `split_at(0)` removes all of the locomotives and wagons from the current train, and moves them into a new train; `split_at(1)` would remove everything but the very head of the train.

If the position is beyond the end of the train, then simply return a new (empty) train, and leave the current train unchanged.

4 Testcases

We have provided a few testcases (and example input files) at [http://www.cs.arizona.edu/classes/cs120/summer17/projects/proj03/](http://www.cs.arizona.edu/classes/cs120/summer17/projects/proj03/)

Each testcase is made up of two files: an `.in` file - which contains the input; and an `.out` file - which is the correct output for that input.

4.1 Writing Your Own Testcases

In this project, you can test a lot of things with a single testcase. So we’re only going to require three different testcases - however, each testcase must have at least ten lines in the output file. (If you don’t have plenty of output to compare, how do you know if it worked?)

To find the correct output for each testcase you create, feed them into my solutions: [http://www2.cs.arizona.edu/classes/cs120/summer17/projects/proj03/demo.html](http://www2.cs.arizona.edu/classes/cs120/summer17/projects/proj03/demo.html)

(Note that

Save your input as a `.in` file (one input per line), and the output from my solution as a `.out` file. Please name them as follows:

```
test-trains-<NetID>-<number>.in
```

```
test-trains-<NetID>-<number>.out
```

(That is, use `trains` as the program name for all testcases.)
5 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project.

CHANGE: You must save all of your files to a .zip file or tarball. Only turn in a single file! (Of course, as normal, you can turn in multiple times - but each one must be a .zip file or tarball. We’ll use your most recent submission only - so include all of the files, even if you’re only changing a few.)

Turn in:

• Each required program

• All of the testcases that you have created