CSC148, Lab #3
week of September 30th, 2013

This document contains the instructions for lab number 3 in CSC148H1. To earn your lab mark, you must actively participate in the lab. We mark you in order to ensure a serious attempt at learning, NOT to make careful critical judgments on the results of your work.

General rules

We will use the same general rules as for the first lab (including pair programming). See the instructions at the beginning of Lab #1 to refresh your memory.

Overview

In this lab, you will write classes representing various kinds of vehicles. The goal is to make you explore and understand inheritance. You will also write test cases using the unittest framework, to test your vehicle classes. The goal is to ensure you understand how to use unittest.

Using inheritance

Getting started

(Student `s1` drives and student `s2` navigates.)

Create a new folder called `lab3`. Start Wing and create a new file called `motorized.py` within folder `lab3`. In file `motorized.py`, write code for a class `Motorized` to represent vehicles that are powered by a gas engine. `Motorized` objects have the following attributes:

- `num_wheels (int)`: the number of wheels of this vehicle.
- `current_pos (float, float)`: the current position of this vehicle on a 2D plane — new vehicles start out at the original position (0.0, 0.0), or 0.0 km east of the origin, 0.0 km north of the origin.
- `fuel_capacity (float)`: the capacity of the fuel tank of this vehicle (in litres).
- `fuel_level (float)`: the current level of fuel (in litres) in this vehicle's fuel tank — new vehicles start out with their tank full.
- `fuel_economy (float)`: the number of litres of fuel required for this vehicle to travel 100 km.

Motorized objects have the following methods:[1]

- `fill_up()`: fill up the fuel tank.
- `travel_to(new_pos)`: change the vehicle's position to `new_pos` if there is enough fuel in the tank. If successful, the vehicle's position and fuel levels are adjusted accordingly; otherwise, a `NotEnoughFuelError` exception is raised and the vehicle's position and fuel are unchanged. (This will require you to create an additional class for the exception.)

  Since road vehicles cannot count on travelling directly between two points "as the crow files", you have to calculate the distance between the vehicle's start position (x_1, y_1) and end position (x_2, y_2) using "Manhattan distance" — assume the vehicle travels in straight east-west segments and north-south segments. The Manhattan distance formula is: $|x_2 - x_1| + |y_2 - y_1|$

Show your work to your TA and switch roles.

Testing `Motorized`

(Student `s2` drives and student `s1` navigates.)

In a new file `test_motorized.py`, write a class `TestMotorized` that is a subclass of `unittest.TestCase`. Write test methods in this class to test the functionality of your class `Motorized`:

- Make sure that you can create objects appropriately, and that the objects you create have attributes set as you would expect.
- Make sure that each one of your methods works the way you expect. You may need to write multiple test cases for some of the methods, to make sure to test their different behaviours.

Try to make your tests thorough — to test every possible use of your methods — but don't spend too long on this task: start with some basic testing, then move on and come back to this later to add more test cases, if you have time.

Have a look at this week's lecture materials to see how to write test cases using the `unittest` module. You may also find it useful to have a look at the Python documentation for unittest!

> http://docs.python.org/3.1/library/unittest.html

Now write code for the following classes, and place the code inside `motorized.py`. Be sure to make appropriate use of inheritance, in order not to duplicate code. Test your new classes when you're done.

- A `Car` is a `Motorized` vehicle with 4 wheels, a 50 liter fuel tank, and a fuel economy of 8 litres per 100 km.
- A `Motorcycle` is a `Motorized` vehicle with 2 wheels, a 15 litre fuel tank, and a fuel economy of 5 litres per 100 km.

Show your work to your TA, and switch roles.

[1] Those of you with a background in Java or C++ may be shocked by the absence of accessor (getter/setter) methods. Don't fret, this is handled, where necessary, in Python with `property(...)`. If you've never worried about this sort of thing, continue not to worry, it will come up later in the course.

Adding one more

(Student `s1` drives and student `s2` navigates.)

In a new file `human_powered.py`, write code for a class `HumanPowered` to represent vehicles that are powered by a human being.

`HumanPowered` objects have the following attributes:

- `num_wheels (int)`: the number of wheels of this vehicle.
- `current_pos (float, float)`: the current position of this vehicle on a 2D plane — new vehicles start out at the origin: position (0.0, 0.0), or 0.0 km east of the origin, 0.0 km north of the origin.
- `speeds (int)`: the number of different "speeds" for this vehicle.

`HumanPowered` objects have the following methods:

- `travel_to(new_pos)`: change the vehicle's position to `new_pos`.

Now, write code for each of the following classes. in the file `human_powered.py`. Be careful to make appropriate use of inheritance throughout in order to avoid duplicated code as much as possible.

- A `Bicycle` is a `HumanPowered` vehicle with 2 wheels and 10 "speeds".
- A `Unicycle` is a `HumanPowered` vehicle with 1 wheel and 1 "speed".

Show your work to your TA and switch roles.

Testing your other classes

(Student `s2` drives and student `s1` navigates.)

Create new files to test each one of your other classes, similarly. You may be able to reuse some of the same test cases — in this case, think carefully about how you may be able to use inheritance to your advantage... You may be able to write subclasses of your own testing cases where you change only the `setUp` and `tearDown` methods without having to change the test cases themselves.

As you write your test cases, try them out!

Show your work to your TA and switch roles.

Taking it further

(Student `s1` drives and student `s2` navigates.)

If you have taken the goal of reducing code duplication to heart, there is something about the set of classes we have above that should be bugging you: `Motorized` and `HumanPowered` vehicles have certain attributes and methods in common...

- Write code for a class `Vehicle` in a new file `vehicle.py`. This is an abstract class representing any kind of vehicle. What this means is that the purpose of this class is to serve as a parent class to other classes that represent vehicles, but that we never intend to create objects from class `Vehicle` directly. Think carefully about the attributes and methods that are duplicated in your classes above and move them to class `Vehicle` instead — this is an example of re-factoring code: writing an improved version that carries out the same task as before, but in a better way.
- Now, complete the re-factoring by making `Motorized` and `HumanPowered` subclasses of `Vehicle`, and removing the common attributes and methods from each subclass.

- Note that some methods (well, just one method really) belong to both subclasses but have different implementations in each subclass. There is no way to combine these methods into a single implementation that makes sense for class Vehicle, so what are we to do? Luckily, Python has a way to handle this situation: include the method in class Vehicle (as an indication that every vehicle can carry out this action), but have it raise an exception when it is called (as an indication that the method is "abstract" and does not make sense for type `Vehicle`; it is merely a place-holder to indicate that the method must be overridden in every subclass — something that is already done in this case). There is even a built-in exception specifically for this purpose: `NotImplementedError`. Make sure that this exception is raised from the method in class `Vehicle`.
- Now, check that you've done your re-factoring correctly by running your test cases again, and fix any bugs that come up!