Practical 8 -Inheritance

In this practical, you will work on extending classes with inheritance.

Inheritance

We recently started making our own classes and objects:

- A **class** is a blueprint (the code) for creating an **object**
- A class is a new **type**
- **Objects** store data in instance variables and provide access to the **methods** (functions) defined in the class

Inheritance: "is a" relationship

Inheritance is appropriate where you are building a more specialised version of a class.
When class B inherits from class A, it should always be the case that an is a relationship holds (B "is an" A). For example, a Tree is a Plant, but it's not true to say a Cat is a Dog. So, it is appropriate for a **Tree** class to inherit from a **Plant** class, but not appropriate for a **Cat** class to inherit from a **Dog** class.

Walkthrough Example - Inheritance

In the last practical, we looked at a **Car** class. This time we see that we can extend the **Car** class to make a **Taxi** class (a more specialised version of a **Car**).
You can use your car from last week, or the finished version in the solutions. Either way, copy your car.py file to your prac_08 folder.
Download taxi.py: https://github.com/CP1404/Practicals/blob/master/prac_08/taxi.py

Read the code and note that the **Taxi** class *extends* the **Car** class in two ways:

- it adds new attributes (**price_per_km**, **current_fare_distance**) and methods (**get_fare**, **start_fare**)
- it **overrides** methods (**drive**, **__init__** and **__str__**) to take account of the characteristics of a Taxi

Notice that the **drive** method still works the same way in terms of its *interface* - it takes in a distance parameter, and it returns the distance driven. This is important for polymorphism - so we can treat all subclasses of **Car** in the same way; i.e. we **drive()** a taxi the same way we **drive()** any car.

Test Taxi

Create a separate file, `taxi_test.py`, to try out your taxi (don't write client code in class files).
Write lines of code for each of the following (**hint**: use the methods available in the Taxi class):

1. Create a new taxi with name "Prius 1", 100 units of fuel and price of $1.23/km
2. Drive the taxi 40km
3. Print the taxi's details and the current fare
4. Restart the meter (start a new fare) and then drive the car 100km
5. Print the details and the current fare

Class Variables

Depending on what kind of system you're modelling with **Taxi**, it might make sense that all taxis have the same price per km. (We don't want to get into one Prius and pay $2.20, then find another one was only $1.20.) So, we can use a "**class variable**", which is a variable that is ***shared*** between all instances of that class. You define class variables directly after the class header line and before any method definitions.

1. Add the **class variable** `price_per_km` to the Taxi class (in taxi.py) and set it to 1.23
2. Change the **__init__** function so it doesn't take in the price_per_km, which means it doesn't need to set self.price_per_km because that's already set by the class variable.
3. Change any client code that passes in this value (where you create your Taxi object).
4. Test your code and see if you get the same output (you should).

Note: when using class variables, you can either:

- refer to the variable as **Taxi**.price_per_km, which explicitly refers to the class variable shared by any Taxi instances, or
- use **self**.price_per_km, which refers to the instance variable that may or may not exist... Python looks for an instance variable in the object and if it doesn't find it there it looks up to the class variable, then it looks to the parent class...
  This is usually preferred, because it allows you to update the value for each object if needed (as we will see later with the SilverServiceTaxi).

Intermediate Exercises

UnreliableCar

Let's make our own derived class for an **UnreliableCar** that inherits from **Car**. Write Python code for this class in a new file, unreliable_car.py, and write some testing code in unreliable_car_test.py to verify each method.

UnreliableCar has an additional attribute:

- **reliability**: a **float** between 0 and 100, that represents the percentage chance that the **drive** method will actually work

UnreliableCar should override the following methods:

- **__init__(self, name, fuel, reliability)**
  - call the **Car**'s version of **__init__**, and then set the reliability
- **drive(self, distance)**
  - generate a random number between 0 and 100, and only drive the car if that number is less than the car's reliability
  - **Super Important Note:** the drive method in the base class (Car) always returns the distance driven, so your derived class (UnreliableCar) should also always return a distance. You must match the "signature" of any method you override.

SilverServiceTaxi

Now create a new class for a **SilverServiceTaxi** that inherits from **Taxi**.
So **SilverServiceTaxi** *is a* **Taxi** and **Taxi** *is a* **Car** (which means **SilverServiceTaxi** is a **Car**)

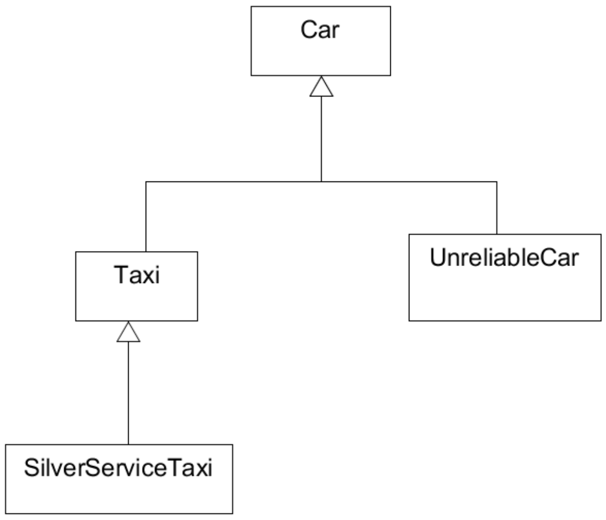
This allows you to have a different effective **price_per_km**, based on the fanciness of the **SilverServiceTaxi**.

1. Add a new *attribute*, **fanciness**, which is a **float** that scales the **price_per_km**
   Pass the **fanciness** value into the constructor and multiply self.price_per_km by it.
   Note that here we can get the initial base price using Taxi.price_per_km, then customise our object's instance variable, self.price_per_km. So, if the class variable (for all taxis) goes up, the price change is inherited by all SilverServiceTaxis.
2. **SilverServiceTaxis** also have an extra charge for each new fare, so add a **flagfall** *class variable* set to $4.50
3. Add or override whatever method you need to (think about it…) in order to calculate the fare.
4. Create an overridden __str__ method so you can add the flagfall to the end. It should display like (for a Hummer with a fanciness of 4):
   ```
   Hummer, fuel=200, odo=0, 0km on current fare, $4.92/km plus flagfall of $4.50
   ```
   Note that you can reuse the parent class method like: `super().__str__()`
5. Write some test code in a file called **silver_service_taxi_test.py** to see that your **SilverServiceTaxi** calculates fares correctly.
   For an 18km trip in a **SilverServiceTaxi** with fanciness of 2, the fare should be $48.78 (yikes!)

Let's stop and think about what we've learned and done so far:

- We can create new classes by *extending* existing ones - e.g. Taxi extends Car
- Derived (child) classes inherit all of the attributes and behaviours of their base (parent) classes - e.g. we do not need to write a new **add_fuel()** method for Taxi because it comes from Car already
- We can *override* (customise) derived classes by modifying existing methods so they do different (but similar) things, and we should always make sure that the signature of overridden methods matches the base class - e.g. the **drive()** method in Taxi also calculates a fare, and in UnreliableCar it may or may not drive… but in all cases, the method takes in a distance value and returns the distance driven.

Here is what the **class hierarchy** looks like now for **Car** and its related classes (remember you can "read" these arrows like "Taxi *is a* Car"):



Inheriting Enhancements

One more thing before we move on… It's important to see how inheritance benefits the systems we model with classes. Currently, all Taxis (including SilverServiceTaxis and any other derived classes we might make) calculate the fare as a regular calculation (price_per_km * current_fare_distance), and you can get results like $48.78 in the example above…
What if we decided that all taxis should have final fares that are rounded to the nearest 10c (so that $48.78 would change to $48.80)? Well, we should only need to make this change to the base Taxi class, and it will be inherited by SilverServiceTaxis… *but only if* we have called **super().get_fare()** and not rewritten the calculation in our new classes. That is, we should only need to change one place because we should have practised the "Don't Repeat Yourself" (DRY) principle. Make sense? Let's do it.

- First, run your silver_service_taxi_test program from above and make sure your output produces something that's not already a multiple of 10c (such as the example above that produces $48.78).
- Now update the **get_fare()** method in Taxi and use the **round()** function.
- Re-run your test and you should get a result rounded to 10c (e.g. $48.80). If it worked, you should see that we only changed Taxi and that enhancement was inherited by SilverServiceTaxi. Nice :)

Yet Another Important Note About Functions: **get_fare()** must return a *number*, not a *string*! Even though we may want to format the result like currency, that's not this function's single responsibility. What if we wanted to add fares together? They must be numbers. Do your string formatting *outside* the function.

Keep going!

Do-from-scratch Exercise - Inheritance

Write a taxi simulator program that uses your **Taxi** and **SilverServiceTaxi** classes.
Each time, until they quit:
The user should be presented with a *list* of available taxis and get to choose one,
Then they can choose how far they want to drive,
At the end of each trip, show them the trip cost and add it to their bill.

Note: Use a list of taxi objects, which you create in main and pass to functions that need it. When you choose the taxi object, your code will call the drive() method on that object and it will use the right method for that class... so from the client code point of view, driving a taxi will work the same whether the object is a Taxi or SilverServiceTaxi, but the results (including the price) do depend on the class. This is ***polymorphism***.

The taxis used in this example would be like:

```python
taxis = [Taxi("Prius", 100), SilverServiceTaxi("Limo", 100, 2),
          SilverServiceTaxi("Hummer", 200, 4)]
```

Sample Output (to show you how to write your program):

```
Let's drive!
q)uit, c)hoose taxi, d)rive
>>> c
Taxis available:
0 - Prius, fuel=100, odo=0, 0km on current fare, $1.20/km
1 - Limo, fuel=100, odo=0, 0km on current fare, $2.40/km plus flagfall of $4.50
2 - Hummer, fuel=200, odo=0, 0km on current fare, $4.80/km plus flagfall of $4.50
Choose taxi: 0
Bill to date: $0.00
q)uit, c)hoose taxi, d)rive
>>> d
Drive how far? 333
Your Prius trip cost you $120.00
Bill to date: $120.00
q)uit, c)hoose taxi, d)rive
>>> c
Taxis available:
0 - Prius, fuel=0, odo=100, 100km on current fare, $1.20/km
1 - Limo, fuel=100, odo=0, 0km on current fare, $2.40/km plus flagfall of $4.50
2 - Hummer, fuel=200, odo=0, 0km on current fare, $4.80/km plus flagfall of $4.50
Choose taxi: 1
Bill to date: $120.00
q)uit, c)hoose taxi, d)rive
>>> d
Drive how far? 60
Your Limo trip cost you $148.50
Bill to date: $268.50
q)uit, c)hoose taxi, d)rive
>>> c
Taxis available:
0 - Prius, fuel=0, odo=100, 100km on current fare, $1.20/km
1 - Limo, fuel=40.0, odo=60.0, 60.0km on current fare, $2.40/km plus flagfall of $4.50
2 - Hummer, fuel=200, odo=0, 0km on current fare, $4.80/km plus flagfall of $4.50
Choose taxi: 2
Bill to date: $268.50
q)uit, c)hoose taxi, d)rive
>>> d
```

```
Drive how far? 60
Your Hummer trip cost you $292.50
Bill to date: $561.00
q)uit, c)hoose taxi, d)rive
>>> c
Taxis available:
0 - Prius, fuel=0, odo=100, 100km on current fare, $1.20/km
1 - Limo, fuel=40.0, odo=60.0, 60.0km on current fare, $2.40/km plus flagfall of $4.50
2 - Hummer, fuel=140.0, odo=60.0, 60.0km on current fare, $4.80/km plus flagfall of $4.50
Choose taxi: 1
Bill to date: $561.00
q)uit, c)hoose taxi, d)rive
>>> d
Drive how far? 50
Your Limo trip cost you $100.50
Bill to date: $661.50
q)uit, c)hoose taxi, d)rive
>>> q
Total trip cost: $661.50
Taxis are now:
0 - Prius, fuel=0, odo=100, 100km on current fare, $1.20/km
1 - Limo, fuel=0, odo=100.0, 40.0km on current fare, $2.40/km plus flagfall of $4.50
2 - Hummer, fuel=140.0, odo=60.0, 60.0km on current fare, $4.80/km plus flagfall of $4.50
```

Practice & Extension Work

First, a Walkthrough Example - Files & Classes

This example program loads a number of "Programming Languages" from a file and saves them in objects using the class we wrote recently.
Download 3 files from https://github.com/CP1404/Practicals/blob/master/prac_08
For now let's start with:

- language_file_reader.py (the client program)
- programming_language.py (the class)
- languages.csv (the data file)

Read the comments and the code in language_file_reader.py to see how it works.
Notice how:

- the file is opened and closed
- readline() is used to read (only) the first line, which just ignores the header in the CSV file
- a for loop is used to read the rest of the file
- reflection is stored in the file as a string, but this client code converts it to a Boolean ready for the class. It's not the class's job to do this conversion but the client's.

(There are also a few other versions included that use Python's **csv** module and a **namedtuple**. Read through them later as extension work if you're interested!)

Modifications

1. Add another language to the file (use data at this Programming Language Comparison page) and make sure it still works properly.
2. Add another attribute to your ProgrammingLanguage class: **Pointer Arithmetic**.
   This will take a bit of effort, as you need to update the class and any code that uses it. You also need to add the correct values to your data file (it's similar to reflection).

More Guitars!

Open the file: **guitars.csv**
This file contains lines like:

```
Fender Stratocaster,2014,765.4
```

So, the format/protocol is:

```
Name,Year,Cost
```

1. Write a program to read all of these guitars in and store them in a list of tuples.
   Display all of the tuples using a loop.

2. Write another version (save a new copy) that stores them in a list of Guitar objects, using the class that you wrote in practical 6 recently.
   Display these using a loop.

   Now **sort** the list by year (oldest to newest) and display them in sorted order…
   How do you do that? Sorting requires that Python knows how to compare objects...
   If we just use:

   ```
   guitars.sort()
   ```

   We get:

   ```
   TypeError: unorderable types: Guitar() < Guitar()
   ```

   So we need to define how the < operator should work. Do you remember how?
   Write code for the **`__lt__`** (less than) method. You should be able to figure this out…
   Then test and see if it sorts correctly now.

Write another version (save a new copy) that does the above, then asks the user to enter their new guitars (just like your practical 6 code).
Store these in your list of guitar objects, then
Write all of your guitars to the file **myguitars.csv**.
Test that this worked by opening the file, and also by running the program again to make sure it reads the new guitars.

Inheritance

1. Cars

Create two more kinds of cars that make sense to you and test them, e.g. select from:

a. **GasGussler** - uses more fuel than it should
b. **Bomb** - doesn't actually move when you drive it, but still uses the fuel
c. **EcoTaxi** - uses half the fuel and gives a 10% on the price per fare
d. **CrazyTaxi**

2. Trees

The focus of this exercise is on inheritance - looking for what methods need to be changed (overridden) in the derived classes. Don't get hung up on the details of the methods...

Trees: some grow wide, some grow thin; some grow fast, some grow slow.
Open these two files: trees.py and trees_tester.py

trees.py contains the **Tree** class. A Tree object has a **trunk_height**, and a number of **leaves**.
The **__str__** method of **Tree** returns a string representation of the **Tree**. For example, if **trunk_height** is 2, and leaves is 8, the Tree would look like

```
##
###
###
 |
 |
```

The size of a **Tree** can be changed by calling the **grow** method, which takes in **sunlight** and **water** and randomly increases the **trunk_height** and leaves.

Not all Trees look the same or grow the same, however, so we're going to build specialised classes to represent different types of trees. To achieve this, we're going to use inheritance.

There are already two completed subclasses of **Tree** in trees.py:

- **EvenTree**
  - even trees only grow leaves in multiples of three, that way the leaves always appear in clean rows
- **UpsideDownTree**
  - upside-down trees are drawn upside-down

trees_tester.py grows **seven** types of trees. Try running it now. The final four types of trees are for you to complete.

There are four more subclasses of **Tree** for you to complete:

1 **WideTree**
   a wide trees grow their leaves in rows of six, and have a trunk that is twice as wide as normal trees
   b you will need to redefine the **get_ascii_trunk** and **get_ascii_leaves** methods
   c example drawing

```
####
######
######
  ||
  ||
```

2 **QuickTree**
   a quick trees grow much quicker than normal trees - their leaves always increase by however much sunlight falls on them, and their trunks always grow by however much water they receive
   b you will need to redefine the **grow** method
   c quick trees look exactly the same as normal trees, they just grow differently

3 **FruitTree**
   a fruit trees have a number of **fruit**
   b add a **_fruit** variable to the **FruitTree** class; initialise it as 1
   c fruit trees sometimes gain an additional fruit when the **grow** method is called, the chance is 1 in 2
   d fruit trees sometimes lose a fruit when the grow method is called, the chance is 1 in 5
   e example drawing (fruit are represented by a dot **.**)
   the fruit should be displayed the same way as the leaves, wrapping within the maximum width

```
..
###
###
 |
|
|
```

4 **PineTree (challenge)**

a pine trees look like

```
   *
  ***
 *****
*******
   |
   |
```

b pine trees start off with four leaves (1 + 3)

c pine trees only ever add as many leaves as would make a full new row at the bottom of the tree

  i i.e. they must form a triangle shape

  ii row 1 always has 1 leaf, then 3 for row 2, 5 for row 3, 7 for row 4, 9 for row 5 and so on

  iii every time the grow method is called, the pine tree should add a new row of leaves if a random number between 0 and sunlight is bigger than 2

3. Taxi Simulator Enhancements

Enhance your taxi driving program so that it:

- doesn't let you drive until you've chosen a taxi
- has error-checking for choosing a valid taxi
- keeps track of the number of km you've done (actual distance driven not total requested)
- displays the taxis with their costs (flagfall and fanciness)