Practical 8 –Inheritance, Code Reviews with PRs

In this practical, you will work on extending classes with inheritance and also do a **code review** for another student using a **Pull Request** on GitHub.

Your reason for doing this subject/degree probably has something to do with getting a job in the IT industry.

Our reason for teaching you things like we are today is the same – to prepare you for work in the industry.

A common and important part of development jobs in the IT industry is doing **code reviews**, where a peer evaluates and comments on another's work with the goal of improving the final result (and mutual learning).

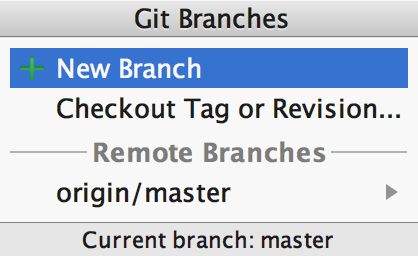
So to help you towards being a better programmer and being more familiar with industry practices, today you will work on some programs, then review another student's work.

This means you need a partner so you can review each other's code. **Organise someone right now and swap GitHub usernames**. It's also OK to review multiple people's code or to have a cycle of people reviewing the next person's... it doesn't have to be one-to-one.

Code reviews are often done via **pull requests** on GitHub **branches**.

A branch is another copy or version of an entire repository. Branches are useful for things like working on new features or bug fixes, or in our case getting feedback.

1. Open your PyCharm Practicals project.
2. Create a branch called "feedback" by: **VCS > Git > Branches** then click **+ New Branch**



It may look like nothing happened, but your local repository now has a new branch, which is currently exactly the same as the 'master' branch, and it is now the one that is "checked out". See in the footer:



So now the work you do and commit will be in the feedback branch, not the master. We will then do a pull request from feedback to master, which is basically a request for someone to **merge** the changes in feedback into the master branch... which gives us the opportunity to provide comments via GitHub.

So you're on the feedback branch? Great! Let's write some code using inheritance...

# Inheritance

Last time, we 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 we’ve included in the taxi file for convenience…

Download taxi.py: <https://github.com/CP1404/Practicals2016/blob/master/Prac07/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.

## Modifications

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 $1.20/km
2. Drive the taxi 40km and print the details and the current fare
3. Restart the meter (start a new fare) and then drive the car 100km
4. 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.

1. Extract the **instance variable** price\_per\_km and make it a **class variable**.  
   This will involve changing the **\_\_init\_\_** function and anywhere it is referred to as **self.price\_per\_km**, which you will change to **Taxi.price\_per\_km** (although self.price\_per\_km will still work since Python looks for the variable in the object, then looks up to the class if it doesn't find it there)
2. Set the default **price\_per\_km** to $1.20/km when you declare it
3. Test your code and see if you get the same output

# Intermediate Exercises

## UnreliableCar

Let’s make our own derived class for an **UnreliableCar** that inherits from **Car**.

**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, 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

Write this up, and write some testing code to verify each method.

## 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**.

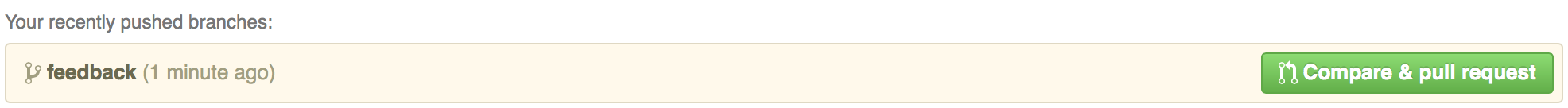
1. Add a new *attribute*, **fanciness**, which is a **float** that scales the **price\_per\_km**  
   Set this **fanciness** value in the constructor
2. **SilverServiceTaxis** also have an extra charge for each new fare, so add a **flagfall** *class variable* set to $4.50
3. Add or override whatever method you need to (think about it…) in order to calculate the fare.
4. Write some test code to see that your **SilverServiceTaxi** calculates fares correctly.  
   For a 10km trip in a **SilverServiceTaxi** with fanciness of 2, the fare should be $28.50 (yikes!)

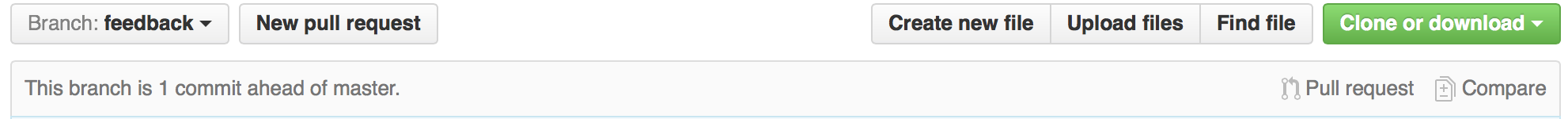


# Code Reviews with Pull Requests

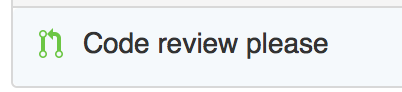
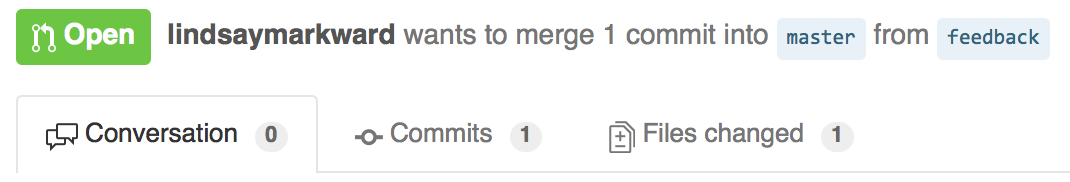
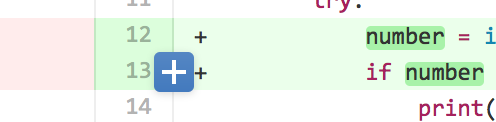
This is a process based on how code reviews and pull requests (PRs) happen in the IT industry but simplified to suit our teaching environment.

1. Commit your changes making sure to add any new files that you created today.   
   You have already created and switched to the 'feedback' branch so your commits will go only to this branch.
2. Open the repository in a web browser and you should see a notice like:



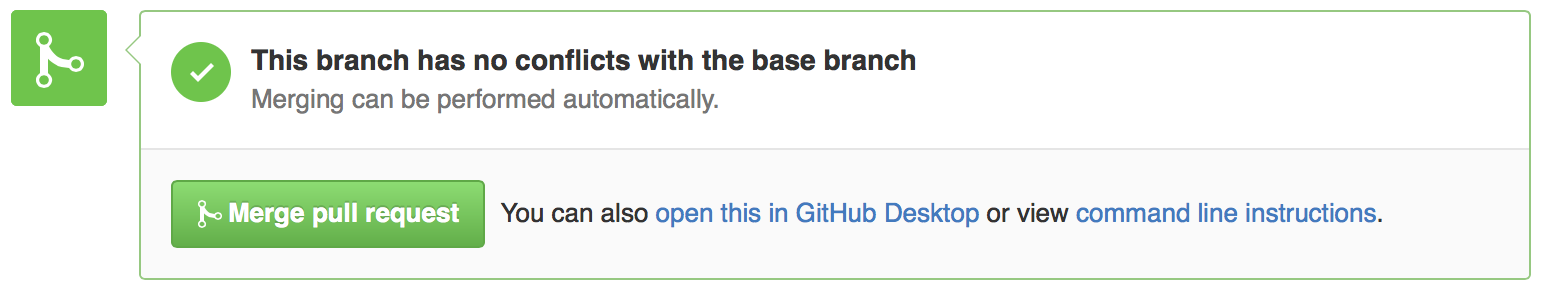
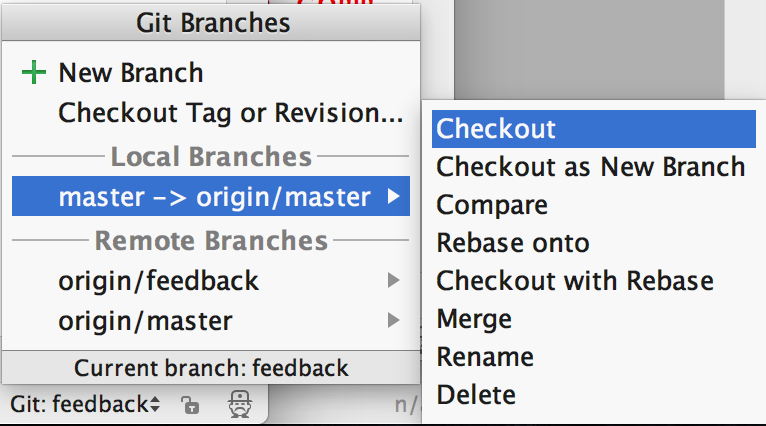
1. Click the green button to make a pull request from feedback to master.   
   If that notice doesn't appear, you can switch to the feedback branch and click Pull Request:  
     
   Add a title like "Code review request" and some detail like "check formatting, naming and logic" (or anything else you want checked).   
   In this description, mention the reviewer with their GitHub username and the @ symbol – e.g. @jondo – so they will be notified (depending on their GitHub preferences)

OK, now if you are the first to do this, you're finished for now... move on to the next section ("Do From Scratch") and come back here when you receive a mention to do a review for someone else...   
If you've already got a PR to review, then carry on with the next steps:

1. On the GitHub website, click on the notifications icon at the top:  which will have a dot on it if you've received your code review request.  
   Open the PR you have been mentioned in by clicking the notification link there, e.g.:   
   Read the request (see if there's anything specific to review), then click on Commits to see the commits:   
     
   Then click on the commit to see the code in "diff" view.
2. Read through the code on GitHub and add line comments. Hover your mouse over the lines and look for the plus icon to add a comment.   
     
   Your job is to look for anything that could be improved including incorrect, inconsistent or non-ideal naming, formatting, logic... anything relevant. Add clear explanations, suggestions or questions.

(When you've finished, there's nothing more to do. In a collaborative environment where you and the reviewer have push access to the repository, there would be more, including the option to merge and close the pull request.)

OK, so at some point you will receive the comments from the reviewer and you can respond to them by making changes in your own code in PyCharm (as well as replying to the comments on GitHub), then commit your work back to GitHub, still in the feedback branch.

1. Ideally, the reviewer would re-check this new work after the updates and make more comments... then the author does more work if need be... reviewer adds more comments... until all good (the reviewer decides when it's finished)... then the reviewer would close the pull request.  
   In our simplified version of this process, you can just **Merge the pull request** now (add a comment if you want):  
      
   GitHub will tell you that the feedback branch can be deleted. You're welcome to delete it.
2. Now the master branch has been updated on GitHub, but not locally.  
   You should switch back to the master branch locally by clicking in the footer where it shows the branch:  
   
3. Pull your changes from master (remote) to update your local repo: **VCS > Git > Pull** then click **Pull**.

All done!

*What did you learn from this?*

To read more about Pull Requests: <https://help.github.com/articles/using-pull-requests/>

# 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

### Sample Output:

Let's drive!

q)uit, c)hoose taxi, d)rive

>>> f

Invalid option

Bill to date: $0.00

q)uit, c)hoose taxi, d)rive

>>> c

Taxis available:

0 - Limo, fuel=100, odo=0, $1.20/km, 0km on current fare

1 - Prius, fuel=100, odo=0, $1.20/km, 0km on current fare

2 - Hummer, fuel=200, odo=0, $1.20/km, 0km on current fare

Choose taxi: 1

Bill to date: $0.00

q)uit, c)hoose taxi, d)rive

>>> d

Drive how far? 40

That trip cost you $48.00

Bill to date: $48.00

q)uit, c)hoose taxi, d)rive

>>> c

Taxis available:

0 - Limo, fuel=100, odo=0, $1.20/km, 0km on current fare

1 - Prius, fuel=60.0, odo=40.0, $1.20/km, 40.0km on current fare

2 - Hummer, fuel=200, odo=0, $1.20/km, 0km on current fare

Choose taxi: 2

Bill to date: $48.00

q)uit, c)hoose taxi, d)rive

>>> d

Drive how far? 40

That trip cost you $172.50

Bill to date: $220.50

q)uit, c)hoose taxi, d)rive

>>> d

Drive how far? 35

That trip cost you $151.50

Bill to date: $372.00

q)uit, c)hoose taxi, d)rive

>>> c

Taxis available:

0 - Limo, fuel=100, odo=0, $1.20/km, 0km on current fare

1 - Prius, fuel=60.0, odo=40.0, $1.20/km, 40.0km on current fare

2 - Hummer, fuel=125.0, odo=75.0, $1.20/km, 35.0km on current fare

Choose taxi: 0

Bill to date: $372.00

q)uit, c)hoose taxi, d)rive

>>> d

Drive how far? 200

That trip cost you $244.50

Bill to date: $616.50

q)uit, c)hoose taxi, d)rive

>>> q

Total trip cost: $616.50

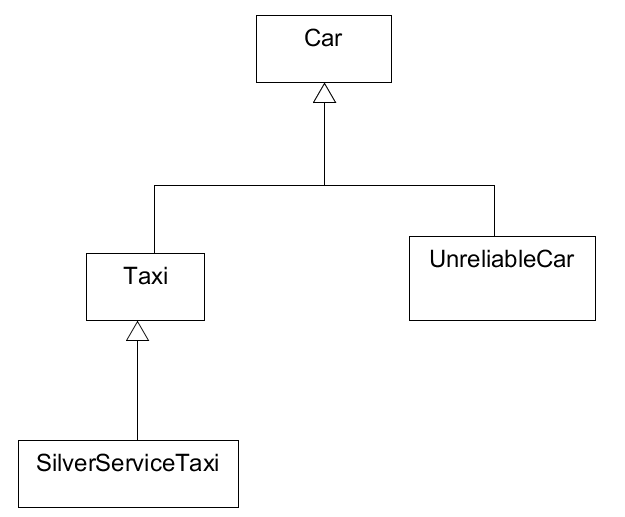
Taxis are now:

0 - Limo, fuel=0, odo=100, $1.20/km, 100km on current fare

1 - Prius, fuel=60.0, odo=40.0, $1.20/km, 40.0km on current fare

2 - Hummer, fuel=125.0, odo=75.0, $1.20/km, 35.0km on current fare

Here is what the **class hierarchy** looks like now for **Car** and its related classes:



# 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 the files, read through them, but focus on the **main()** function.

* language\_file\_reader.py - <https://github.com/CP1404/Practicals2016/blob/master/Prac07/language_file_reader.py>
* programming\_language.py - <https://github.com/CP1404/Practicals2016/blob/master/Prac07/programming_language.py>
* languages.csv - <https://github.com/CP1404/Practicals2016/blob/master/Prac07/languages.csv>

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 basically just ignores the header in the CSV file
* a for loop is used to read the rest of the file

(There are a few other versions that use Python’s **csv** module and a **namedtuple**. Don’t look at these now, but you can 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](http://www.jvoegele.com/software/langcomp.html) page) and make sure it still works properly.

# Practice & Extension Work

## More Guitars!

Download the file: <https://github.com/CP1404/Practicals2016/blob/master/Prac07/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:

* 1. **GasGussler** - uses more fuel than it should
  2. **Bomb** - doesn’t actually move when you drive it, but still uses the fuel
  3. **EcoTaxi** - uses half the fuel and gives a 10% on the price per fare
  4. [**CrazyTaxi**](https://en.wikipedia.org/wiki/Crazy_Taxi)

### 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.

**Download these two files:**<https://github.com/CP1404/Practicals2016/blob/master/Prac07/trees.py>

<https://github.com/CP1404/Practicals2016/blob/master/Prac07/treestester.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

treestester.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 **getASCIITrunk** and **getASCIILeaves** 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. 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)