

---

# Assignment 1: Spotting Stolen Cars

---

6% of overall grade  
Due Friday 24 August 2018, 11:55pm

## 1 Overview

### 1.1 Introduction

Thanks to advances in computer and network technologies we live in an age of increasingly pervasive surveillance. As with most things, there are pros and cons to such widespread surveillance but we will leave that philosophical discussion to your Philosophy courses and/or COSC101. This assignment isn't going to take sides in the debate—our aim is simply to give you some insight into how data from such surveillance systems might be used efficiently to meet whatever good/evil ends that one might have for the data.

### 1.2 Due date

Your assignment code should be submitted by Friday 24 August 2018, 11:55pm. The drop dead date is 7 days later, and any late assignments will incur a 15% absolute penalty. That is your mark will be the raw mark less 15 marks (given the assignment is marked out of 100).

### 1.3 Submission

Submit via the quiz server. The submission page will be open closer to the due date.

### 1.4 Implementation

All the files you need for this assignment are on the quiz server. You must not import any additional standard libraries unless explicitly given permission within the task outline. For each section there is a file with a function for you to fill in. You need to complete these functions, but you can also write other functions that these functions call if you wish. All submitted code needs to pass `PYLINT` program checking (it will make you proud of your stylish code).

## 1.5 Getting help

The work in this assignment is to be carried out individually, and what you submit needs to be your own work. You must not discuss code-level details with anyone other than the course tutors and lecturers. You are, however, permitted to discuss high-level details of the program with other students. If you get stuck on a programming issue, you are encouraged to ask your tutor or the lecturer for help. You may not copy material from books, the internet, or other students. We will be checking carefully for copied work. If you have a general assignment question or need clarification on how something should work, please use the class forums on Learn, as this enables everyone to see responses to common issues, but **never** post your own program code to Learn. Remember that the main point of this assignment is for you to exercise what you have learnt from lectures and labs, so make sure you do the relevant labs first, so don't cheat yourself of the learning by having someone else write material for you.

## 2 Spotting stolen cars

Modern camera and computer vision technologies make it possible to use roadside cameras to collect the number plates of all the vehicles that pass the camera (other information such as speed and direction can also be paired with the plate numbers). This has many applications such as:

- monitoring entry and exit from toll road systems (to allow for automatic charging of tolls)
- simple vehicle counting, to help with traffic flow models
- spotting the cars of wanted criminals, etc. Check out [this image](#) from [Parking Today](#).
- automatic sending of speeding tickets
- providing messages on motorway signs asking particular cars to slow down. See See section 4.2 in [this document](#)<sup>1</sup>
- and finally, spotting stolen cars, which is the focus of this assignment.

In this assignment you will be working with the data from a roadside camera based Automatic Number Plate Recogniser (ANPR). You will be carrying out a simplified version of what might be done in reality—so you can focus on the fun stuff. Basically you will be processing a stream of number plates from a single ANPR and making a list of the seen number plates that appear in a list of stolen car plates. In a more realistic scenario you would make a list of stolen cars that had been spotted, including the time and location so that the whereabouts of stolen cars could be narrowed down—or in a real-time system, police patrol cars could be sent out to intercept stolen cars when they exited a motorway (at a [Paége/toll-both](#) for example).

For each task of the assignment the functions you write will need to take two lists containing sequences of NumberPlate objects. The first list contains number plates of stolen vehicles and the second contains number plates of vehicles that have been sighted by a number plate camera. You may assume that there are no duplicate number plates in a single list (ie, each number plate occurs at most once in each list).

---

<sup>1</sup>The plate shown on the sign is an example (ie, ABC-123) and *trop vite* translates to *too quick!*)

In each task you will need to use specific data structures (e.g. `NumberPlates`), which you will import into your program. You do not have to implement these data structures, you will just have to interact with them. They are in the file `classes.py`. Details of these data structures are in the relevant section below.

For each task you must return a list containing the the stolen number plates that were sighted by the camera and the number of `NumberPlate` comparisons that were made in producing the list with the specified method. Note, that you should only count `NumberPlate` comparisons. That is, you should only count comparisons between `NumberPlate` objects (because this can be an expensive operation if the plates are large) — you shouldn't count comparisons of indexes, counters, and so on. The provided `NumberPlate` class will also automatically count comparisons as they are made. This will allow you to check how many comparisons have actually occurred, but this is intended for debugging only. If you use this comparison checking in any code you submit (rather than doing the comparison counting yourself) then your code will fail the Quiz Server (Code Runner) tests used for submitted assignments.

## 2.1 Testing your code

Off-line tests are available in the `tests.py` module and you should use them to check your implemented code before submission (the submission quiz won't fully mark your code until submissions close). The provided tests use the Python `unittest` framework. You are not expected to know how these tests work; you just need to know that the tests will check that your code finds the list of stolen number plates that were sighted, and that the number of comparisons that your code makes is appropriate.

For the first algorithm you should be able to match the exact number of comparisons, but for binary search the comparisons made are checked to be in the right ballpark for the expected number of comparisons. If you think that you have an implementation that is very close to the test range, and works reliably, you can ask to have the expected number of comparisons range reconsidered. The tests used on the quiz server will be mainly based on the tests in `tests.py`, but a small number secret test cases will be added, so passing the unit tests is not a guarantee that full marks will be given for that question (it is however a strong indication your code is correct).

For example you should be testing various edge cases such as when an input list is empty or when no number plates are sighted etc...


## 2.2 Provided classes

The `classes.py` module contains the definition for the `NumberPlate` class. The `NumberPlate` class is basically a wrapper around the existing Python `str` class, but with the added benefit of being able to keep track of every `NumberPlate` comparison that is made.

**NumberPlate** : The most fundamental class is the `NumberPlate`. For now, you may consider a `NumberPlate` to be a `str`, in that all the common comparisons (`==`, `<`, `>`, `<=`, `>=`, `!=`) are available (eg, if `a` and `b` are `NumberPlates` then expressions like `a < b` and `a == b`, etc, will work as expected). The difference is that we keep track of these comparisons in the background, via the `StatCounter` class. You should be counting how many `NumberPlate` comparisons your functions do—in a similar fashion to what you did in Lab 1.

The `stats.py` module contains another class that you may find useful: `StatCounter`. This class holds all the comparison information that you might need to check your code against. We also use it to verify that you do perform the correct number of comparisons.

`StatCounter` : In order to count how many comparisons your code makes, we provide a `StatCounter` class. Note that because of the way it is implemented, it will not behave like a regular class. This class is only for testing purposes, and not in your final code. You should only use the `StatCounter.get_comparisons()` method for testing and should remove all calls to it from the code you submit to the Quiz Server. You will, of course, need to remove any imports related to `stats` from your code—otherwise *pylint* might get angry with an unused import.

 **Important:** You cannot rely on `StatCounter` (or more specifically, the methods and instance variables in `StatCounter`) being available to you on the quiz server at the time of submission, so do not use it in your final code.

### 2.3 Provided tools and test data

The `utilities.py` module contains functions for reading test data from test files. Test files are in the folder `test_data` to make it easier to test your own code. The test data files are named `stolen[s]-sighted-matches-seed.txt`, where *stolen* is the number of stolen plates in the file [followed by an *s* to indicate that they are in sorted order]; *sighted* is the number of plates in the list that were sighted by the camera; *matches* is the list of expected plates (in the order your functions should return them) which is used for testing; and finally the *seed* is the seed value that was used when generating the test file.

For example `10-20-5-a.txt` was generated with `a` as the random key and contains 10 stolen plates and 20 sighted plates, where 5 of the sighted plates were in the stolen list. If the file name was `10s-20-5-a.txt` then the stolen plates would be provided in sorted order—obviously useful for testing your binary search function. You should open some of the test data files and make sure you understand the format—remember to open them in Wing or a suitably smart text editor. If you are using windows then try Notepad++ because the normal Notepad is pretty horrible. If you are using Linux (or a Mac) then almost any text editor will do.

The test files are generated in a quasi-random fashion and the seed suffix indicates the random seed that was used when generating the data—you don't need to worry about this. But, you do need to worry about the fact that we can easily generate different random files to test with.


The most useful function in `utilities.py` is obviously `read_dataset`, it reads the contents of a test file and returns three lists. The lists all contain `NumberPlate` objects. The first list contains the number plates of stolen vehicles, the second contains all the number plates that were seen by the camera and the third contains the stolen number plates that were seen by the camera (this third list is basically the plates that your functions should return).

The following example shows how to use the utilities module:

```
>>> from utilities import read_dataset
>>> filename = './test_data/5-5-2-a.txt'
>>> stolen, sighted, matches = read_dataset(filename)
>>> print(stolen)
['IQ1998', 'FS1860', 'JI9400', 'NN3705', 'AS0734']
>>> print(sighted)
['AR3546', 'QT0780', 'NN3705', 'PB2873', 'AS0734']
>>> print(matches)
['NN3705', 'AS0734']
>>> type(stolen[0])
<class 'classes.NumberPlate'>
>>> filename = './test_data/5s-5-2-a.txt'
>>> stolen, sighted, matches = read_dataset(filename)
>>> print(stolen)
['AS0734', 'FS1860', 'IQ1998', 'JI9400', 'NN3705']
```

## 2.4 Provided tests

The `tests.py` provides a number of tests to perform on your code. Running the file will cause all tests to be carried out. Each test has a name indicating what test data it is using, what it is testing for, and a contained class indicating which algorithm is being tested. In the case of a test case failing, the test case will print which assertion failed or what exception was thrown. The `all_tests_suite()` function has a number of lines commented out indicating that the commented out tests will be skipped; uncomment these lines out as you progress through the assignment tasks to run subsequent tests.

 **Important:** In addition to the provided tests you are expected to do your own testing of your code. This should include testing the trivial cases such as empty lists and lists of differing sizes.

## 3 Tasks [100 Marks Total]

### 3.1 Sequential/Linear search [40 Marks]

This task requires you to complete the `simple_linear_plate_finder` function in the `linear_finder.py` module. This first method isn't going to be very efficient, but it's a starting point! You should start with an empty list representing the stolen number plates that have been sighted by the camera. Go through each `NumberPlate` in the `sighted_list` and sequentially search the `stolen_list` to try find the `NumberPlate`. If a match is found add that number plate to the stolen plates sighted so far and stop searching the rest of the `stolen_list` for that `NumberPlate` (as it can be assumed that no number plate occurs twice in the stolen list). You cannot assume that either the `stolen_list` or the `sighted_list` will be in any particular order.

Your function should return a tuple containing the populated list containing the stolen number plates that were sighted `NumberPlates` and the number of `NumberPlate` comparisons that the function made (see the example return line in the supplied code). The returned list of `NumberPlates` should be in the same order that the plates appear in `sighted_list`.

You can assume that both lists provided to your function will only contain unique elements. That is, no number plate will appear in the stolen list more than once and no number plate will appear in the sighted list more than once. You can use this knowledge to finish scanning the sighted list earlier in some cases. Think about when

it's Ok for your function to stop scanning through the sighted list before reaching the end of that list.

### 3.2 Finding sighted plates using binary search [60 Marks]

This task requires you to complete the `simple_binary_plate_finder` function in the `binary_finder.py` module.

In the first task, we made no guarantees about the order of the number plates. Can we make a more efficient algorithm for finding common number plates if we know one of the provided lists (in particular, the stolen list) is given in lexicographic order? Hopefully you answered, "Yes, of course! We can use a binary search!"

For this task you are required to design and implement a function that generates the list of stolen number plates that were sighted by the camera, using binary search to find each `NumberPlate` from the `sighted_list` in the `stolen_list`. You can assume the stolen list will be sorted.

Your function should return a tuple containing the populated list containing the stolen number plates that were sighted `NumberPlates` and the number of `NumberPlate` comparisons that the function made (see the example return line in the supplied code). The returned list of `NumberPlates` should be in the same order that the plates appear in `sighted_list`.

NOTE: Your binary search must only do one comparison per loop when it is searching the stolen list and should only compare number plates for equality once. This approach will basically narrow the search down to one number plate and then check if that is the one being searched for.

Again you can also use the fact that both lists will contain only unique number plates to allow your function to sometimes stop checking sighted number plates before the end of the sighted plates list is reached.



**Important:** Binary search can be difficult to implement correctly. Be sure to leave sufficient time to solve this task.

— Have fun —