COMPSCI 105 Summer School 2017
Assignment One

Worth: 6% of your final grade
Deadline: 5:00pm, Friday 20th January

Please submit what you have completed before the deadline
No extensions, unless *exceptional* circumstances apply
(discuss with Paul as soon as possible)

What you need to do

For this assignment, your task is to define *three* Python classes. The classes you must define are:

- Polynomial
- Car
- Traffic

All three class definitions should appear in a single source file, which must be called "Asst1.py". This is the file you will submit for marking. The general structure of your submission will therefore be:

Asst1.py

```
class Polynomial:
    ....
    ....

class Car:
    ....
    ....

class Traffic:
    ....
    ....
```

Please note: it is very important that the "Asst1.py" source file you submit for marking does not contain any program statements other than the three class definitions (i.e. there should be no statements appearing outside the class definitions).

Your submission will be marked using a separate program that imports your class definitions (see the example "TestProgram.py" provided), and tests your implementations by creating instances of the classes and calling the methods you have defined on those objects.

You must submit this single source file to the Assignment Drop Box:

https://adb.auckland.ac.nz

prior to 5:00pm on Friday 20th January.

Paul Denny
January 2017

Question 1: The Polynomial class (50 marks)

For this question you must define a class called **Polynomial**. This class definition must include the following methods:

- __init__() - the initialiser for the class
- __str__() - returns a formatted string representation of a Polynomial object
- add_term() - adds a new term (coefficient and exponent) to the Polynomial
- add() - modifies the *existing* Polynomial by adding another one to it
- __add__() - returns a *new* Polynomial object that is the sum of two polynomials
- scale() - scales a Polynomial object by multiplying each coefficient by a factor
- evaluate() - evaluates a Polynomial object for a given *x*-value
- get_degree() - returns the degree of the Polynomial object
- collapse() - reduces the Polynomial object to a single term

Background

A *polynomial* is a mathematical expression that is the sum of a series of *terms* that each consist of a variable (we will use *x*) which is taken to a given power and multiplied by a given coefficient. For example, the following are two different polynomials:

$$2x^4 + x^2 - 7x + 13$$

$$x^3 - 5x^2 + 1$$

For this question you will be defining a class called Polynomial that can be used to represent polynomials in our programs. When a Polynomial object is first created it contains no terms, and thus has the corresponding value 0. You can add terms to a Polynomial object by calling the add_term() method. Each term consists of a coefficient (which should be non-zero) and an exponent.

The __init__() method

The initialiser method takes no inputs and just creates an empty Polynomial object. An example of this is shown below:

```
p1 = Polynomial()
```

Once you have completed this initialiser method, you should define the __str__() method so that you can print out Polynomial objects. The value of a newly created Polynomial object will be 0. For example:

```
p1 = Polynomial()
print('p1 =', p1)
```

will print:

```
p1 = 0
```

A more complete description of the __str__() method will be given shortly. First, to add terms to a Polynomial object, you should define the add_term() method.

The add_term() method
This method adds one term to a Polynomial. The method takes two inputs - the first is the coefficient value, and the second is the exponent value. For example, to add the term x^2 to a Polynomial object:

```
p1 = Polynomial()
p1.add_term(1, 2)
```

To create the Polynomial object that represents:

$$2x^4 + x^2 - 7x + 13$$

we could have:

```
p1 = Polynomial()
p1.add_term(2, 4)
p1.add_term(13, 0)
p1.add_term(1, 2)
p1.add_term(-7, 1)
```

Notice that the order in which the terms are added to the Polynomial object does not matter. When a Polynomial object is printed (i.e. when the __str__() method is called), it will always be printed in order of *decreasing* exponents. For example, if we print the Polynomial just created above:

```
print('p1 =', p1)
```

the output will be:

```
p1 = 2x^4 + x^2 - 7x + 13
```

Let's now look at the __str__() method in more detail.

The __str__() method
This method is called *automatically* by Python whenever a Polynomial object is provided as input to the print() function. The function should return a string that represents the polynomial.

The string returned by this method should use the '^' character to indicate exponentiation. The following example shows a Polynomial with both a negative and positive exponent:

```
p1 = Polynomial()
p1.add_term(123, 987654321)
p1.add_term(-9, -987654321)
print('p1 =', p1)
```

the output here is:

```
123x^987654321 - 9x^-987654321
```

Please pay **very careful** attention to the following points:

- If the coefficient of a term is 1, then the coefficient is *not* shown
  - i.e. x + 2 rather than 1x + 2
- If the coefficient of a term is negative, then a *single minus sign* should be shown rather than a plus and minus sign
  - i.e. x^2 - 2x + 2 rather than x^2 + -2x + 2
- There should be a single space character on either side of a '+' or '-' sign that appears *internal* to the polynomial. The only exception to this rule is at the very start of the polynomial. For the very first term, if the coefficient is positive then no sign should be shown, but if the coefficient is negative then a single minus sign should be shown *without any* surrounding spaces
- The terms should appear in *descending order* of exponent value - note, this may differ from the order in which the terms were added to the Polynomial. Only terms with non-zero coefficients should be displayed!

Here is another example that incorporates these rules:

```
p1 = Polynomial()
p1.add_term(1, 3)
p1.add_term(-4, 2)
p1.add_term(-100, 25)
p1.add_term(-66, 0)
p1.add_term(6, -7)
print('p1 =', p1)
```

which should produce the output:

```
p1 = -100x^25 + x^3 - 4x^2 - 66 + 6x^-7
```

The evaluate() method

The evaluate method takes an input value of *x* (which will be an *integer*) and it should return the *value of the polynomial* if evaluated using that value for *x*. Because exponents may be negative, the output may be a floating point number. For example, the following code:

```
p1 = Polynomial()
p2 = Polynomial()

p1.add_term(3, 5)
p1.add_term(-2, 2)
p1.add_term(1, -1)
result1 = p1.evaluate(2)

p2.add_term(-7, 10)
result2 = p2.evaluate(2)

print('result1 =', result1, 'and result2 =', result2)
```

would produce the output:

```
result1 = 88.5 and result2 = -7168
```

The get_degree() method
The *degree* of a polynomial is simply the value of the greatest exponent. For example, the following code:

```
p1 = Polynomial()
p1.add_term(5, 2)
p1.add_term(1, 11)
p1.add_term(8, -7)
print('p1 =', p1)
degree = p1.get_degree()
print('degree =', degree)
```

should produce the output:

```
p1 = x^11 + 5x^2 + 8x^-7
degree = 11
```

The scale() method
The scale method takes a single integer as input, and it multiplies the polynomial by that value as a factor. This means that each *coefficient* of the polynomial is multiplied by the factor. Note, the scale() method modifies the existing Polynomial object. For example, the following code:

```
p1 = Polynomial()
p1.add_term(-4, 3)
p1.add_term(2, 2)
p1.add_term(-1, 1)
p1.add_term(7, 0)
p1.add_term(6, -2)
p1.scale(-2)
print('p1 =', p1)
```

produces the output:

```
p1 = 8x^3 - 4x^2 + 2x - 14 - 12x^-2
```

The __add__() method
This method is called *automatically* by Python whenever the + operator is used on a Polynomial object. Strictly speaking, when the expression “a + b” appears, the actual method call that is made is: a.__add__(b). This method should add the two Polynomial objects together and return a *new* Polynomial object representing their sum. Note that in this case, neither of the original polynomials should be modified. Adding two polynomials is just a matter of combining all of the terms from each polynomial, and adding the coefficients of any terms that have the same exponent.

The add() method
The add() method is similar to the __add__() method in that it performs addition on two Polynomial objects, however this method should actually *modify* the Polynomial object on which it is called, rather than returning a new Polynomial object. In other words, when a.add(b) is called, the terms of b should be added to the Polynomial represented by a.

The examples below illustrate the difference between the __add__() and add() methods. Firstly, note that when using the + operator to add two Polynomial objects, neither of the two operands are modified:

```
p1 = Polynomial()
p1.add_term(10, 2)
p1.add_term(-5, 0)

p2 = Polynomial()
p2.add_term(5, -1)

p3 = p1 + p2
print('p1 =', p1)
print('p2 =', p2)
print('p3 =', p3)
```

produces the output:

```
p1 = 10x^2 - 5
p2 = 5x^-1
p3 = 10x^2 - 5 + 5x^-1
```

However, following on from this example - when calling the add() method, the object on which the method is called actually changes:

```
p2.add(p1)
print('p1 =', p1)
print('p2 =', p2)
```

produces the output:

```
p1 = 10x^2 - 5
p2 = 10x^2 - 5 + 5x^-1
```

CAREFUL... BE CAREFUL...

Consider the following:

```
p1 = Polynomial()
p1.add_term(10, 2)
p1.add_term(-5, 1)
p2 = Polynomial()
p2.add_term(-10, 2)
p1.add(p2)
print('p1 =', p1)
```

which should produce the output:

```
p1 = -5x
```

Where did the high-order term (i.e. the term with exponent 2) go? In this case, when the two Polynomials were added, the high exponent terms *cancelled out*. The degree of p1 is now 1.

The collapse() method

The collapse() method reduces a Polynomial object to a single term. It does this by adding together all of the coefficients to produce a coefficient sum, and adding together all of the exponents to produce an exponent sum. The single term then just consists of the coefficient sum and the exponent sum. For example:

```
p1 = Polynomial()
p1.add_term(1, 10)
p1.add_term(2, 0)
p1.add_term(3, -1)
p1.add_term(4, -2)
p1.add_term(5, -3)
p1.collapse()
print('p1 =', p1)
```

should produce the output:

```
p1 = 15x^4
```

Of course, if the sum of the coefficients is zero, the resulting Polynomial will have the value 0:

```
p1 = Polynomial()
p1.add_term(1, 1000)
p1.add_term(-1, -10)
p1.collapse()
print('p1 =', p1)
```

should produce the output:

```
p1 = 0
```

One more thing to note:

If more than one term with the same exponent is added to a Polynomial, the corresponding coefficients of those terms are just summed. For example:

```
p1 = Polynomial()
p1.add_term(1, 10)
p1.add_term(1, 10)
p1.add_term(1, 10)
print('p1 =', p1)
```

would produce the output:

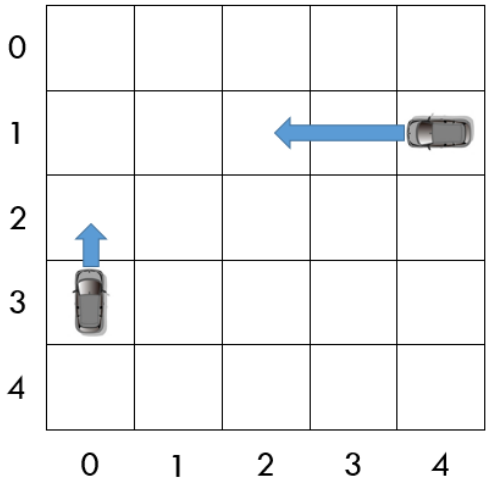
```
p1 = 3x^10
```

Question 2: The Car and Traffic classes (50 marks)

Imagine a very simplistic simulation of cars driving on a grid of squares. A number of cars are initially placed at different starting locations on the grid, and then the simulation begins. Each car moves in a fixed direction with a fixed speed, and the movement of each car is *discrete* - in other words, when a car moves, it simply *jumps* to a new square. Cars will continue to move unless they collide, at which point they stop moving. The goal of the simulation is to determine which cars will collide after moving a certain number of times.

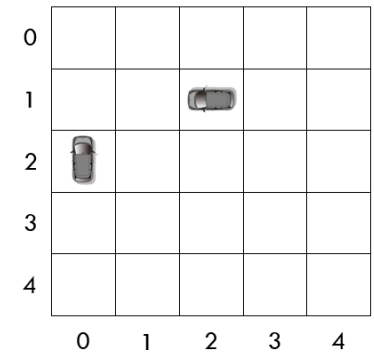
As an example, consider the initial configuration shown on the right. There are two cars on a 5x5 grid. The first car is at position (3, 0) and is moving up. The second car is at position (1, 4) and is moving to the left. The second car is also moving twice as quickly as the first car - while the first car moves up only one square at a time, the second car moves left two squares at a time.

Where and when will these cars collide?

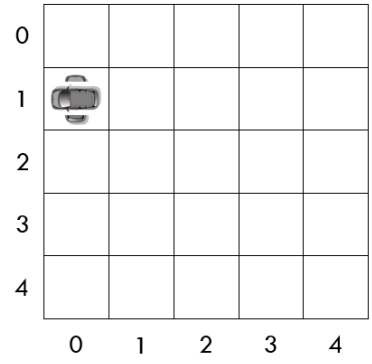


Initial configuration

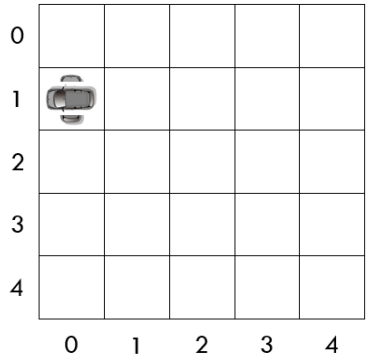
The table below shows the positions of the cars after one, two and three discrete moves. After one move (shown on the far left below), the first car is at position (2, 0) and the second car is at position (1, 2). After two moves, both cars are on square (1, 0) *and have collided*. Once the cars have collided, they can no longer move, and so the cars will remain at this position even if additional moves take place in the simulation. For example, notice that on the far right below, after three moves, both cars are still at position (1, 1).



After one "move"



After two "moves"



After three "moves"

For this question, you need to define a Car class and a Traffic class which can be used to carry out this simulation.

Overview

The Car class will be used to represent the car objects, and each car object will store its current location, speed and direction, as well as a unique license plate that can be used to identify the car. The Traffic class will represent the grid of squares and will store a collection of Car objects. The Traffic class will have a special method, called execute(), which can be used to run a specified number of moves of the simulation. The Traffic class will also contain methods that can be used to find out which cars have collided, and where.

To make this a little more concrete, a short example is shown next that illustrates the use of both of these classes. Following this example, a more detailed specification is provided for the Car and Traffic classes.

An example

Let's say we wanted to set up the simulation that was illustrated in the previous diagrams. We would create a Traffic object (representing the grid) consisting of 5 rows and 5 columns, and we would then add two Car objects to this grid:

```
t = Traffic(5, 5)

c1 = Car("UP123", 3, 0, -1, 0)
c2 = Car("LEFT123", 1, 4, 0, -2)

t.add_car(c1)
t.add_car(c2)
```

The first line of code creates an empty 5x5 grid. The next two statements create two Car objects. Notice that the two cars are given unique license plate numbers - "UP123" and "LEFT123". The license plate is the first input to the initialiser of the Car class. The next two digits that follow the license plate number in the initialiser are the row and column position of the initial location of the car. The final two digits represent the direction of the car - in other words, they indicate the amount by which the row and column position will change each time the car moves. For example, each time the car "UP123" moves, the row value decreases by 1, and the column value stays unchanged.

The final two statements in the code fragment above add the cars to the grid. We are now ready to begin running the simulation. Firstly, we can print the initial configuration of the simulation as follows:

```
print(t)
```

and this will display:

```
UP123_3_0 LEFT123_1_4
```

Notice the format here - each car is printed as a string representation. This string consists of the license plate of the car, then an underscore, then the row position, then another underscore, and finally the column position. The cars are listed *in the same order* as they were added to the Traffic object that represents the grid. There is also a single space character separating the two cars (but otherwise there is no leading or trailing whitespace in this output).

We can now execute one move of the simulation, and print the resulting configuration, as follows:

```
t.execute(1)
print(t)
```

After this one move, the output will be:

```
UP123_2_0 LEFT123_1_2
```

Note that this reflects the new positions of the two cars after the first move (the car “UP123” has moved up by one position, and the car “LEFT123” has moved left by two positions). If we now execute two more moves, and print the resulting configuration after each one:

```
t.execute(1)
print(t)

t.execute(1)
print(t)
```

the output produced would be:

```
UP123_1_0 LEFT123_1_0
UP123_1_0 LEFT123_1_0
```

Notice that in the first line of output above (representing the state of the grid just after the *second* move), both cars have arrived at location (1, 0) and they have therefore collided. Running the simulation further will not change their positions (collided cars cannot move), and so the state of the grid remains unchanged in the second line of output above (after the third move).

The Traffic class provides a method called “get_collisions()” which returns a sorted list of the positions (shown as tuples) at which collisions have occurred in the simulation. So in this example:

```
result = t.get_collisions()
print('collisions at:', result)
```

would print:

```
collisions at: [(1, 0)]
```

as there is just one location on the grid where collisions have occurred. Another method in the Traffic class is “get_plates()” which, given the location at which a collision has occurred, will return a sorted list of all of the license plates of the cars involved in the collision. For example, in this case:

```
plates = t.get_plates(1, 0)
print('plates:', plates)
```

will print:

```
plates: ['LEFT123', 'UP123']
```

The following sections list the *required* methods that you *must* include in the Car and Traffic classes, along with a description of the required behaviour. You may include additional methods in these classes if you wish.

The Car class

You must include at least the following methods in the Car class:

- __init__() - the initialiser for the class
- __str__() - returns a formatted string representation of a Car object
- get_plate() - returns the license plate of the car
- get_position() - returns the position of the car as a tuple: (row, column)
- move() - updates the position of the car (the grid size is provided as input)
- collides() - returns True if the two cars collide and False otherwise

The initialiser takes 5 inputs: the license plate, the row and column position of the location of the car, and the amounts by which the row and column positions should change on each update.

The move() method is passed two inputs, representing the size of the grid. When a car moves off the grid, it re-enters the grid on the *opposite* side (i.e. the grid *wraps-around*). In other words, a car at position (0, 0) moving one square to the left on a 10x10 grid would move to position (0, 9). The collides() method is passed one car object as input, and returns True if the input car is at the same location as the car on which the method is called.

```
c1 = Car('ABC123', 2, 2, 1, -1)
c2 = Car('DEF456', 5, 8, 0, 2)

print('c1 =', c1)
print('c2 =', c2)

c1_plate = c1.get_plate()
c1_pos = c1.get_position()

print('c1_plate =', c1_plate)
print('c1_pos =', c1_pos)

c1.move(10, 10)
c2.move(10, 10)
print('c1 =', c1)
print('c2 =', c2)

c3 = Car('GHI789', 3, 1, -1, 2)
print(c1.collides(c3))
print(c2.collides(c3))
```

should produce the output:

```
c1 = ABC123_2_2
c2 = DEF456_5_8
c1_plate = ABC123
c1_pos = (2, 2)
c1 = ABC123_3_1
c2 = DEF456_5_0
True
False
```

The Traffic class

You must include at least the following methods in the Traffic class:

- __init__() - the initialiser for the class, which is passed the grid size
- __str__() - returns a formatted string representation of a Traffic object
- add_car() - adds a Car object to the grid
- execute() - runs the specified number of "moves" of the simulation
- get_location() - gets the current location of the car with the specified plate
- get_plates() - returns a sorted list of the plates at the specified location
- get_collisions () - returns a sorted list of positions of all collisions on the grid

The initialiser method takes 2 inputs: the number of rows and the number of columns in the grid.

The __str__ method returns a string consisting of the string representations of each Car object in the grid (in the order they were added to the grid by the add_car() method), separated by single spaces, but with no leading or trailing white space.

The add_car() method adds a car to the grid. There are two special cases that you need to pay attention to:

1. Firstly, a car can only be added to the grid if the grid location is currently unoccupied. If the location of the car being added is the same as the location of a car that has already been added to the grid, then you should raise an Exception (of type ValueError) to indicate this error (and the grid should not be updated).
2. Secondly, each car on the grid must have a unique license plate. If the license plate of the car being added is the same as the license plate of a car that has already been added to the grid, then you should not add the new car - instead, you should just update the location of the existing Car object on the grid. In other words, the location of the existing car, already on the grid, should be changed to the new location of the Car object passed to this method as an input.

Both of these special cases for the add_car() function are described in further detail later.

The execute() method takes one integer input which represents the number of moves to perform for the simulation. This method should update the locations of each car on the grid. Once all of the car positions are updated, then this method should check for any collisions that have occurred (and it should handle these appropriately).

The get_collisions() method returns a sorted list of positions - these will be tuples of the form (row, column) - of all of the collisions that have occurred on the grid. If no collisions have occurred, this method should return an empty list.

The get_plates() method takes a grid position as input, and it returns a sorted list of the license plates of all of the cars at that grid location. If there are no cars at the specified location, this method should return an empty list.

The get_location() method takes a license plate as input and it returns the current location (as a tuple) of the corresponding car. If there is no car with the specified license plate, this method should return None.

An example on a 10x10 grid

The following example illustrates the behaviour of the Car and Traffic classes:

```
t = Traffic(10, 10)
c1 = Car('DEF456', 5, 5, -1, -1)
c2 = Car('ABC123', 1, 1, 1, 1)
c3 = Car('XYZ999', 9, 9, 0, 0)
c4 = Car('SLOW12', 8, 8, 0, 0)
c5 = Car('FAST99', 0, 4, 8, 4)
t.add_car(c1)
print(t)
t.add_car(c2)
print(t)
t.add_car(c3)
print(t)
t.add_car(c4)
print(t)
t.add_car(c5)
print(t)

print('location 1 =', t.get_location('DEF456'))
print('location 2 =', t.get_location('XYZ999'))

t.execute(1)
print(t)

t.execute(1)
print(t)

t.execute(1)
print(t)

print('collisions =', t.get_collisions())

print('plates 1 =', t.get_plates(3, 3))
print('plates 2 =', t.get_plates(9, 9))
```

This should produce the output below:

```
DEF456_5_5
DEF456_5_5 ABC123_1_1
DEF456_5_5 ABC123_1_1 XYZ999_9_9
DEF456_5_5 ABC123_1_1 XYZ999_9_9 SLOW12_8_8
DEF456_5_5 ABC123_1_1 XYZ999_9_9 SLOW12_8_8 FAST99_0_4
location 1 = (5, 5)
location 2 = (9, 9)
DEF456_4_4 ABC123_2_2 XYZ999_9_9 SLOW12_8_8 FAST99_8_8
DEF456_3_3 ABC123_3_3 XYZ999_9_9 SLOW12_8_8 FAST99_8_8
DEF456_3_3 ABC123_3_3 XYZ999_9_9 SLOW12_8_8 FAST99_8_8
collisions = [(3, 3), (8, 8)]
plates 1 = ['ABC123', 'DEF456']
plates 2 = ['XYZ999']
```

Please note:

When calling the add_car() method, there are two situations you must take care to handle correctly.

1. If the car being added to the grid is at the *same location* as a car that already exists on the grid, then a collision would be generated. This is not sensible, and so in this case, you must not modify the grid and in addition you must raise a “ValueError” exception to indicate this error. You can choose what to provide as a string input to the ValueError initialiser (or you can leave this empty).

2. If the car being added to the grid has the same license plate as a car that already exists on the grid, then you should simply update the existing position of the car to the new position - a new Car object should not be added to the grid.

The following example illustrates these two cases:

```
t = Traffic(10, 10)
c1 = Car('DEF456', 5, 5, -1, -1)
c2 = Car('ABC123', 1, 1, 1, 1)
c3 = Car('XYZ999', 5, 5, 0, 0)
c4 = Car('DEF456', 8, 2, 0, 0)

t.add_car(c1)
t.add_car(c2)

try:
    t.add_car(c3)
except ValueError:
    print('Caught the error!')

print(t)

t.add_car(c4)
print(t)
```

should produce the following output:

```
Caught the error!
DEF456_5_5 ABC123_1_1
DEF456_8_2 ABC123_1_1
```

Getting started with Assignment 1

To get started, you should download the resource files from Canvas:

- Asst1.py
- TestProgram.py

The Asst.py source file is a skeleton version of the file you must submit for marking. You should modify this file (don't remove any of the required methods that are provided to you, as this will prevent the marking program from running) by implementing the functionality described in this document.

If you run the TestProgram, it will attempt to create Polynomial, Car and Traffic classes (as defined in Asst1.py), and it will test these objects by calling various methods. You will not submit this program for marking, but you can use it to test your implementations of the classes. The actual marking program will use a similar format to this test program.

Initially, before you start work on implementing the Polynomial, Car and Traffic classes in "Asst1.py", you will see the following output from TestProgram.py:

```
Currently, your implementation of "Asst1.py" performs as follows:

  Tests passed: 0
  Tests failed: 50

Please note, this only gives an approximate indication of the
correctness of your "Asst1.py" source file.  You should design
additional tests to test your class definitions more thoroughly

Good luck!
```

As you implement "Asst1.py", you can keep track of how many of the provided tests you are passing. Please note though, this is just an approximate indication of correctness - the marking program will use a more detailed suite of tests.

IMPORTANT

Your own work

This is an individual assignment. You must write the code for your classes on your own. Under no circumstances should you copy source code from anyone else, or allow your work to be copied. Discussing ideas or algorithms in general is acceptable, but you must write all of the code you submit yourself.

This will be enforced, so please don't try to copy and make superficial changes to make your code "appear" different to the code you copied. It is simple to detect such copying and modifications, and all submissions for this assignment will be checked.

Submitting your assignment

When you have finished, submit your source file Asst1.py to the Assignment Drop box:

https://adb.auckland.ac.nz/

Your submitted classes will be marked automatically, so make sure that you adhere to the formats exactly as described in this document.

Have fun!