When Objects are Alike

1. Terminology

Superclass -> Class that is being inherited from

Subclass -> Class that inherits from a superclass

Class variable -> Variables that are part of the class definition and are shared by all instance of the class

1. Inheritance

Mostly used to extend the functionality of a class. I.E. You have 2 objects A and B. You want them to share functionality X but B also has to have functionality Y. Therefore, B can inherit from A

Inheritance can also be used to change behaviour. This is done via overriding methods from the superclass in the subclass.

The super function returns ther object as an instance fo the parent class. We can use super to call parent methods. Super can be made inside any method. All methods fcan be modified via overriding and calls to super.

1. Multiple Inheritance and Mixins

Multiple inheritance refers to when a subclass inherits from more than 1 parent class and can access functionality form both of them. In general you should not do this.

A form of multiple inheritance is a mixin. A mixin is a superclass that is not meant to be instantiated but meant to be inherited by some other class to provide extra functionality.

For example, let’s say we want to add functionality to multiple classes to send emails. We can create a MailSender mixin and have multiple classes inherit from it.

However, another method instead of a mixin is simply creating a standalone Python function for sending an email and just calling that function with the correct email address supplied as a parameter. [This is the preferred option]

Another method again is to just use composition. I.E. Instead of inheriting from MailSender, we can simply create MailSender object and have our classes that want to send emails have a property that is an object of MailSender.

1. Composition

Composition refers to the act of collecting various objects together to create a new one. I.E. A car is made up of many other objects such as tires, hood, bumper, engines. Other objects such as a van or a plane also require the same thing.

Objects in OOP can refer to more than physical objects They can be abstract ideas as well.

Example modelling a game of chess

**Objects**

* Player
* Chess Set
  + Chess Board
  + Pieces
  + Position

We can see that 3 other objects make up chess set. Hence, chess set is composed of various other objects.

A full OOP diagram is like this

**Player**

Methods

* Make move

**Chess Set**

Attribute (i) pieces -> List (ii) Board -> Board

**Board**

Attribute (i) chess\_set -> ChessSet (ii) positions -> Positions

**Position**

Attribute (i) chess\_board -> Board

**Piece**

Attribute (i) chess\_set -> ChessSet

1. Polymorphism

Concept basically means different behaviours happening depending on which subclass is being used without having to explicitly know what the subclass actually is. As an example, imagine we have a program that play audion files. A media player might need to load an **AudioFile** object and *play* it.

The **AudioFile** thus might have a *play* method which is responsible for decompressing or extracting the audio and routing it to sound card and speakers.

However, different types of AudioFiles would need to be decompressed and extracted differently. I.E> .wav file, .mpc and .ogg files all different.

We can use inheritance with polymorphism to simplify the design. Each type of file can be a subclass of **AudioFile** and each would have a play method that is implemented different for each class. The media player object would never need to know which subclass of AudioFile it is referring to. Just calls play and polymorphically lets the object take care of the implementation detail.

Note the parent class is able to access class variables that are defined in subclasses.

We can also say that the details of playing is left to each subclass and hence has been **encapsulated.**

Python has duck typing which means that any object that provides the required behaviour can be used without forcing that object to be a subclass.

Polymorphism is one of the big reasons to use inheritance instead of composition outside of Python. In Python, any object that supply the correct interface can be used interchangeably in Python. It thus reduces the need for polymorphic common superclasses. In Python, if all that is being shared is the public interface, duck typing is all that is required.

1. Abstract base class (“ABCs”)

ABCs define a set of methods and properties that a class must implement to be considered a duck-type instance of that class. The class can extend the ABC itself to be used as an instance of that class but must supply all the methods defined in ABC.

Python contains ABCs in the native collections module. One of the ABCs is Container which contains 1 method named contain that takes 1 argument. This means that for any class to be a valid subclass of Container, it would need to implement the contains method which takes 1 argument.

Note: In Python cause of duck typing, classes don’t need to explicitly inherit from Container ABC to be considered an instance of Container. If we just created a class OddContainer and have the methods contains as so,

class OddContainer:

def \_\_contains\_\_(self, x):

if not isinstance(x, int) or not x % 2:

return False

return True

This is one of the biggest advantage of duck typing. That we can create relationships without writing code that sets up inheritance.

**Case Study Notes**

1. Requirements

Trying to create automated grading system for programming assignments. Needs to provide class-based interface for course writers to create their assignment and give useful error messages if interface is not being fulfilled.

Writer needs to be able to supply their lesson content and to write custom answer checking code. They should be able to access student name to display messages to make content more familiar and friendly.

1. Design Notes

Abstract Base Class of Assignment created. 2 classes of IntroToPython and Statistics Created. AssignmentGrader manages how many attempts the student has made at any given assignment. Grader manages which assignments are available and which one each student is currently working on.

The design flow for grader is as follow

1. Register assignments
2. Start\_assignment method. -> This stores student and instance of assignment as key value pair in student\_graders dictionary
3. Get\_lesson method -> This accesses the instance of assignment in student\_grader and runs the lesson method. The lesson method displays the lesson they are expected to use
4. Check\_assignment -> This runs the check method on student and takes in student code
5. Assignment\_summary -> Prints assignment summary

The idea here is that there is lots of composition and no inheritance (though there is duck typing with an ABC).

Grader is composed of many objects such as (i) Assignment (ii) Assignment Grader

Assignment Grader is also composed of Assignment

The division of labour is also important. Assignment Grader -> Just count attempts

Assignment -> Display the assignment + contain code to check whether the assignment is correct

In our case, important that all AssignmentClass follows the interface defined in Assignment as in other places where AssignmentClass instance is being passed in, we can reliably just call methods that are in the interface such as check and lesson.