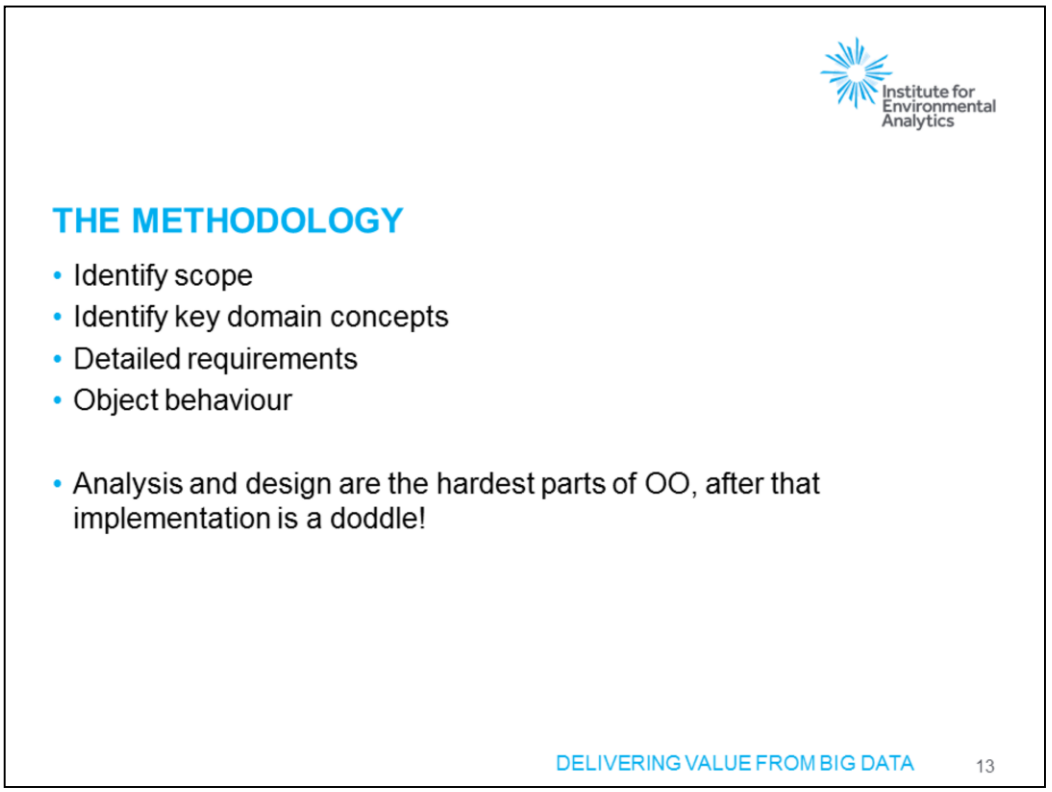
Institute for Environmental Analytics

COURSE OVERVIEW (1)

- Day 1
  - Version Control
    - Revision
    - Team working
    - GitHub
  - OO introduction
- Day 2
  - OO Design & CRC
  - Diagrams
  - More OO concepts
  - Standards & Naming



Scope

What's in and what's out… who are the users? (actors) and what do they do? (use cases) This is the equivalent of a requirement specification which you saw when doing procedural design.

Domain Concepts

Use cases are an external view of the system, so we need a framework to hang the internals on, and the concepts provide the vocabulary by identifying atomic entities, their definitions and their relationships.

Detailed Requirements

We flesh out the system using objects and classes and identify the relationships between them. If it's a large system, then we'd actually start with bigger 'lumps' of functional responsibility and develop a component diagram.

Object Behaviour

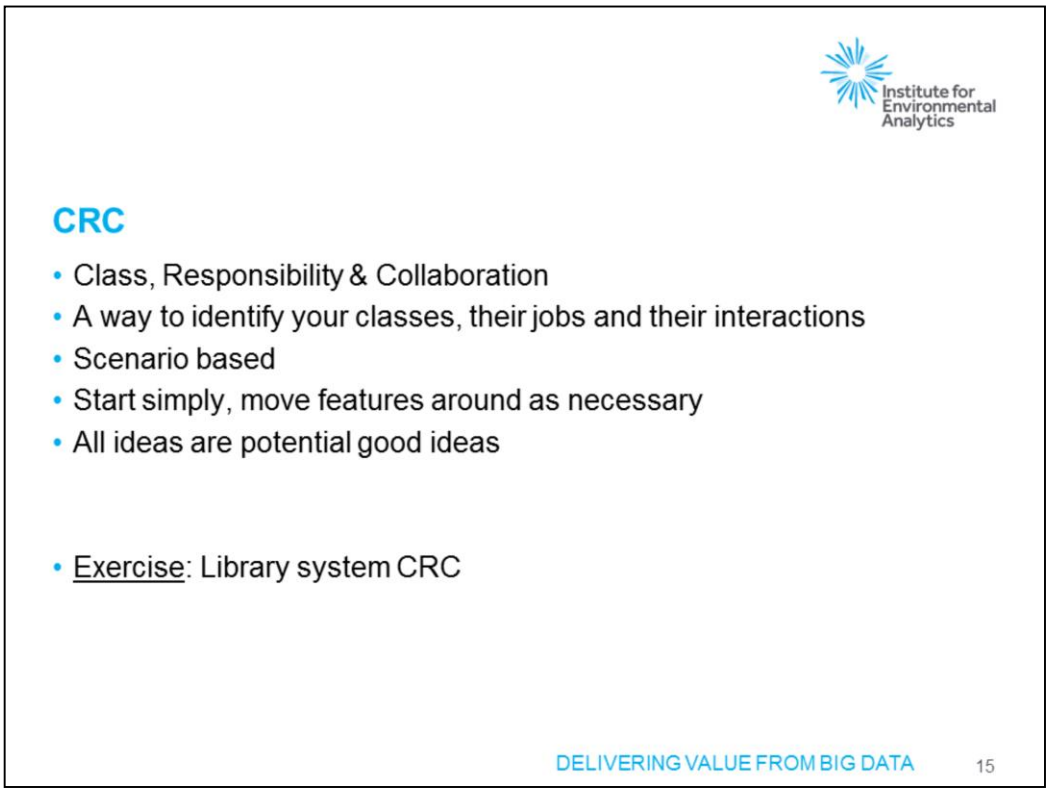
We think about scenarios and construct sequence diagrams to model behaviour. We might look at state diagrams to represent the behaviour of a single object – an object may behave differently to the same message depending upon its internal state.

Institute for Environmental Analytics

CONCEPTS OF OBJECT TECHNOLOGY

- What is an object?
- What is an association?
- What is aggregation?
  - And what is composition?
- What is a class?
- What is inheritance?

- Homework review : identify classes, hierarchies and relationships
  - random objects
  - library system



This is moving on from identifying nouns and noun phrases and into developing the messages, data and objects in a system.

CRC stands for Class, Responsibility & Collaboration.

It's a brainstorming opportunity. Each person assumes the role of a class or an actor, and they go through a scenario in order to identify what information they hold, how they are grouped, whom they collaborate with, and what messages pass between them. They each take responsibility for an aspect of the system functionality.

• Class

A set of objects that share common structure and common behaviour

Super-class : a class from which another class inherits

Subclass : a class that inherits from one or more classes/interfaces

• Responsibility

Some behaviour for which an object is held accountable.

• Collaboration

Process whereby several objects cooperate to provide some higher-level behaviour.

http://userpages.umbc.edu/~cseaman/ifsm636/lect1108.pdf

*coweb.cc.gatech.edu/ice-gt/uploads/794/**CRC**Project.doc*

Completeness

Everything that is needed is present

Sufficiency

Everything that is present is needed

Primitiveness

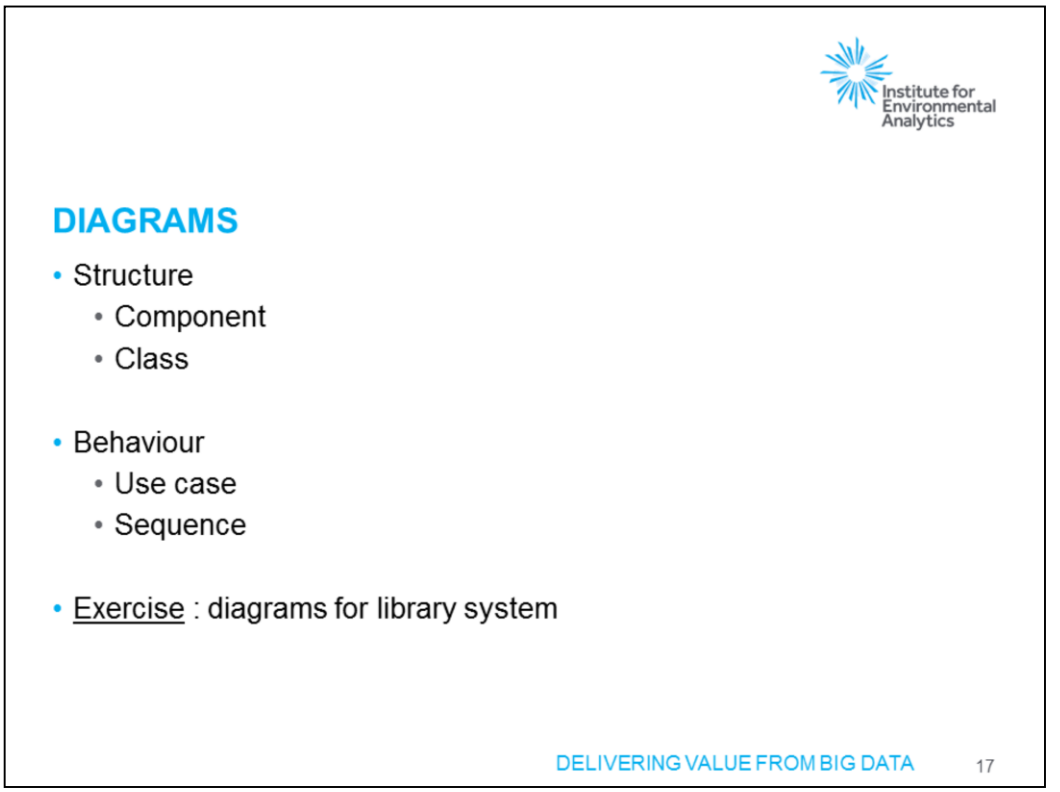
Everything that is present is primitive

High cohesion

Everything that is present is required to be together

Low coupling

Everything to which we are coupled is required



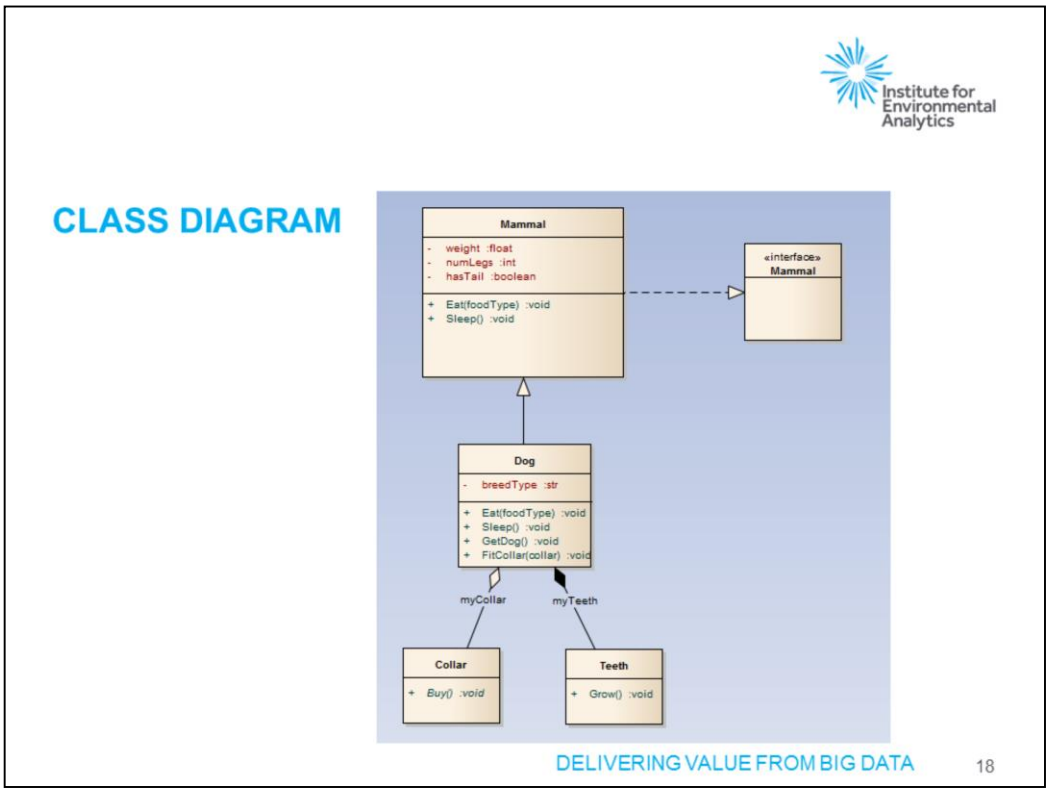
As with functional decomposition and procedural design, it's really helpful to get things straight on a piece of paper first (or appropriate tool if available). It allows you space to think through the problem, sketch out options for solutions, work through them and iterate until you have something which satisfies all the requirements.

Component Diagram

Represents big chunks of grouped functionality; you're not sure precisely what objects are inside, just there will be some and the result will be that a complex job will be done. For example, in a big banking system, there may be components such as customer database, transactions, web facilities, overnight processing etc. If you had to think about classes straight away, it'd get way too complicated.

Use Case

This is a static representation of behaviour: there are actors (a user, another system etc) which do 'stuff' to the system. A use case is a simple scenario of the actions an actor will expect to be able to do. For example, if you (a customer) were ringing up to order something, you'd need to 'check item status' and 'place order', possibly also 'establish credit'. A salesperson would also need to do the first two, and perhaps a supervisor also interacts with the last. There's also a 'fill orders' which is where the shipping clerk interacts. The use cases form the basis of scenarios for the system which you can then work through to tease out CRC.



The class diagram is probably the most useful to straighten out class relationships such as inheritance, aggregation, composition, interface, and association. There are various other things you can represent, but these are a good starting point.

Inheritance: this provides the polymorphism found in OO design. A superclass, sometimes also called a base class, holds all the attributes and methods which are common amongst all its descendents. For example, a class *Shape* may have an attribute of isRendered and a method RenderShape( ) which toggles its appearance on a display. This is common irrespective of whether the shape is a square or a circle. However, the *Circle* and *Square* subclasses would have their own data for dimensions: radius, and height & length respectively. If you called an abstract function GetArea( ) on the abstracted class *Shape*, it would pass on the responsibility of finding the area to specific GetArea( ) overridden methods in the individual concrete classes,*Circle* and *Square*.

An 'IS A' relationship.

Aggregation: classes which are often found together and reference each other but can exist separately

Composition: one class contains another which cannot exist otherwise

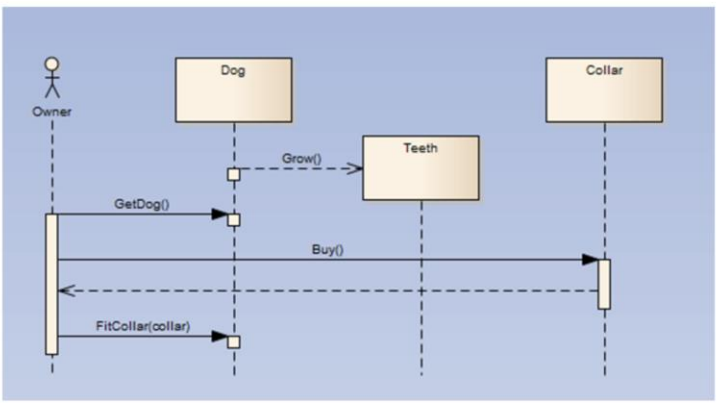
These are 'HAS A' relationships.

Interface: this is the way of representing what your public face is to the outside world. In reality, each class has a public face – its public methods. However, if you want a class to do its own version of another classes methods but you already inherit from a superclass, then you should only inherit additional interfaces. This is strictly enforced in Java, not in C++. The disadvantage of multiple inheritance is that you can end up with the same grandparent via different routes, at which point the runtime code does not know how to resolve which way to go! Python doesn't specifically have the notion of interfaces, it allows multiple inheritance, but you will see it in other languages.

Association: a looser relationship than aggregation or composition, but indicates that classes need to 'know' about each other in some way.

Institute for Environmental Analytics

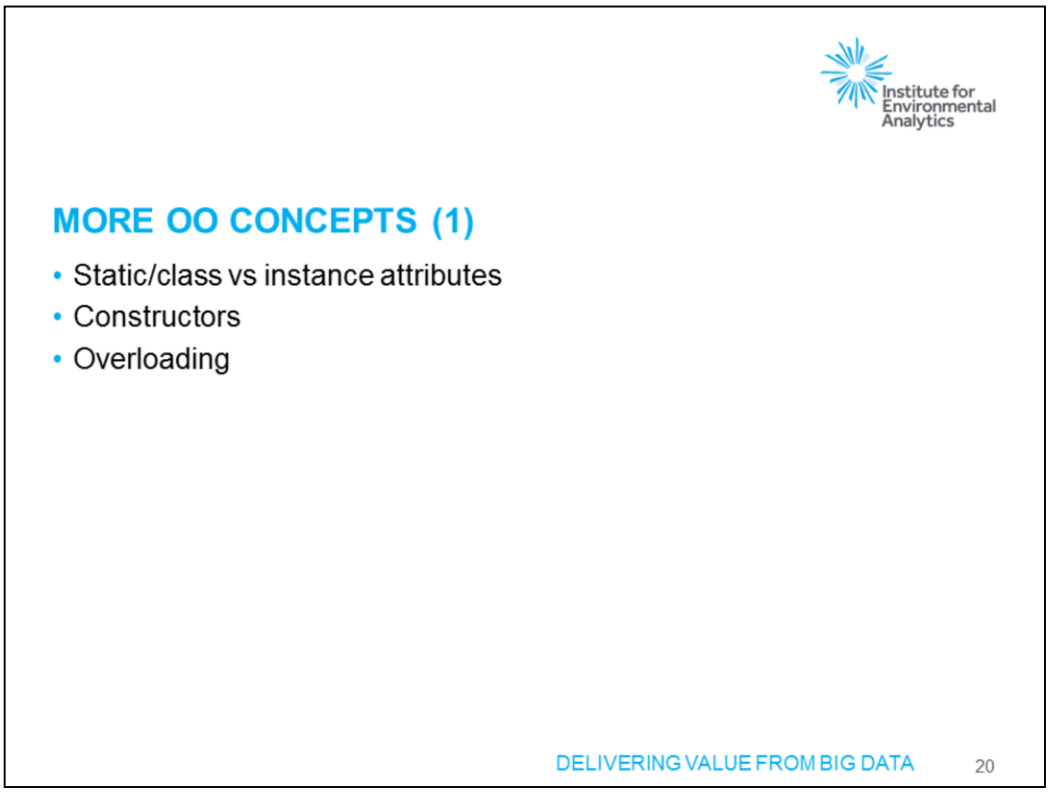
SEQUENCE DIAGRAM



DELIVERING VALUE FROM BIG DATA 19

A sequence diagram represents the path through a system for a given scenario. You cannot hope to capture all possible paths although some tools allow for alternate paths and iteration, but since you'll be sketching on paper, stick to one path!

In the diagram, time goes down the page, and it's normal to have the prods to the system, the actors, on the left, with objects appearing left to right as they're needed.



OO Terminology

Class: A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

Class variable: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.

Data member: A class variable or instance variable that holds data associated with a class and its objects.

Function overloading: The assignment of more than one behaviour to a particular function. The operation performed varies by the types of objects (arguments) involved.

Instance variable: A variable that is defined inside a method and belongs only to the current instance of a class.

Inheritance : The transfer of the characteristics of a class to other classes that are derived from it.

Instance: An individual object of a certain class. An object that belongs to a class Circle, for example, is an instance of the class Circle.

Instantiation : The creation of an instance of a class.

Method : A special kind of function that is defined in a class definition.

Object : A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

Operator overloading: The assignment of more than one function to a particular operator.

This is really good and well worth reading thoroughly to get to grips with the syntax:

http://www.tutorialspoint.com/python/python_classes_objects.htm

In Python, the runtime environment 'knows' which object (i.e. instance of a class) you are using when you call a method or access an attribute. It does this by silently passing a 'self' pointer into the class' code. Therefore all your methods have to have 'self' as their first argument, and you should refer to all attributes as 'self.attribute'. This maintains the scope of the name you've used to be precisely the one which belongs to the object.

class X(Y)

"Make a class named X that is-a Y."

class X(object): def __init__(self, J)

"class X has-a __init__ that takes self and J parameters."

class X(object): def M(self, J)

"class X has-a function named M that takes self and J parameters."

<in both of the above, the class is declared to explicitly inherit from the base 'object' class common to all Python types.>

foo = X()

"Set foo to an instance of class X."

foo.M(J)

"From foo get the M function, and call it with parameters self, J."

foo.K = Q

"From foo get the K attribute and set it to Q."

(http://learnpythonthehardway.org/book/ex41.html)

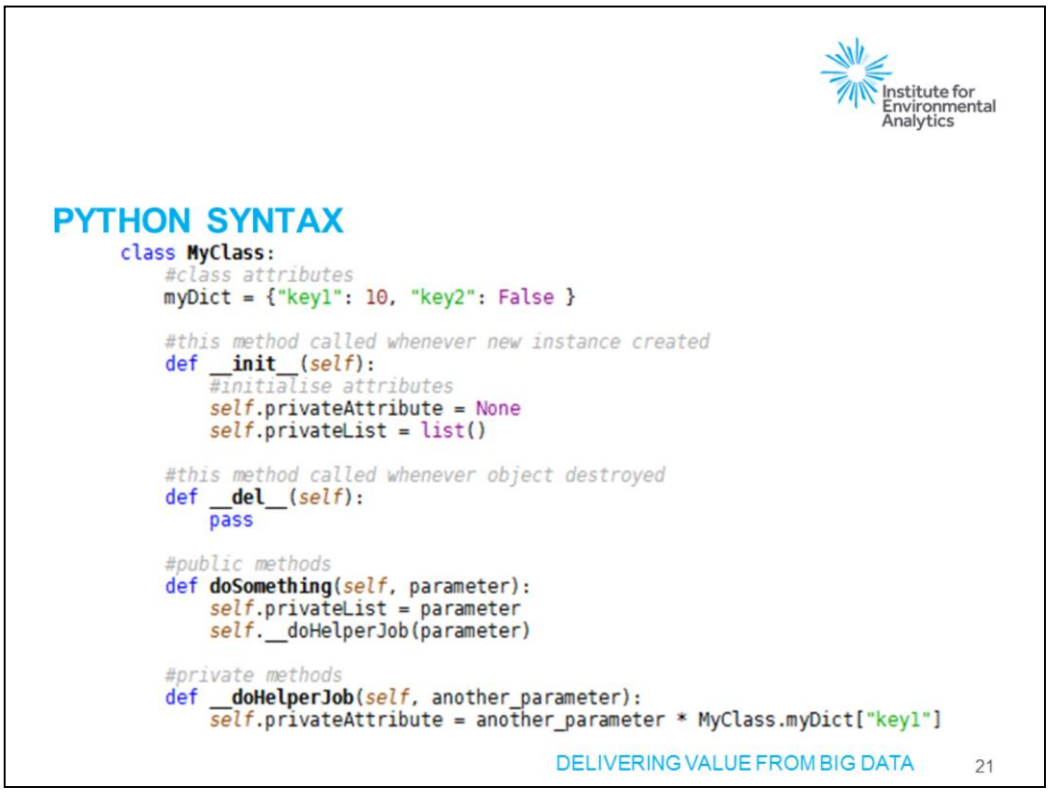
Other very good resources:

http://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide

http://www.diveintopython.net/object_oriented_framework/class_attributes.html

http://effbot.org/pyfaq/how-can-i-overload-constructors-or-methods-in-python.htm

http://stackoverflow.com/questions/682504/what-is-a-clean-pythonic-way-to-have-multiple-constructors-in-python (also factory methods)



Class attributes *can* be changed in an object instance – Python won't stop you!

But you can use MyClass.myDict without needing to create an instance of the class.

The special methods with '__' either side are inherited from the base 'object' class used by everything in Python. Here, we have over-ridden their behaviour. __init__ is the object constructor, __del__ is the destructor and it's good practice to put them both in even if they just do nothing (pass).

Methods cannot be overloaded in Python, which means you have to be a bit clever if you want overloaded constructors.

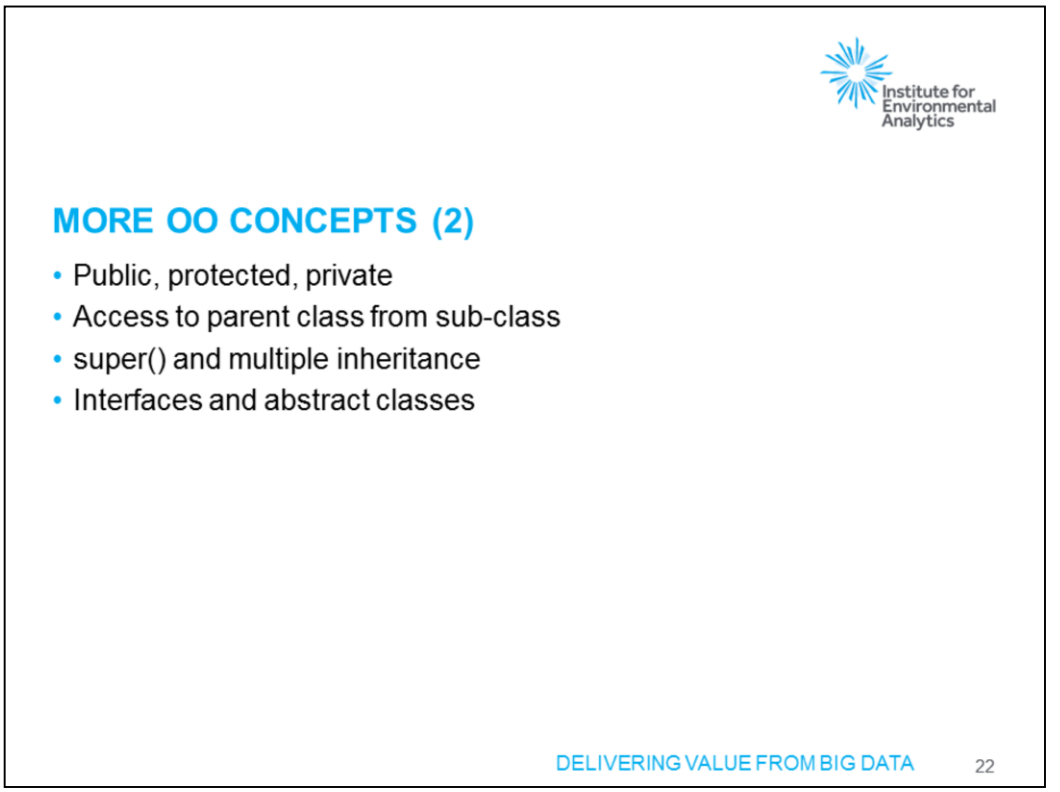
It's convention to use '_' in front of methods which are private to the class. They also won't have a docstring so users of your class should steer clear.

In Python, it's easy to break the rules of OO so you *must* adhere to conventions and not cheat! Other languages have keywords which enforce OO: public, protected, private,

On the subject of how you divide up classes between files, it should be 'sensible'! See these threads:

http://stackoverflow.com/questions/106896/how-many-python-classes-should-i-put-in-one-file

http://stackoverflow.com/questions/2864366/are-classes-in-python-in-different-files



"**Python's Super is nifty, but you can't use it"**
https://fuhm.net/super-harmful/

"**Python's super() considered super!"**
https://rhettinger.wordpress.com/2011/05/26/super-considered-super/

"Things to Know About Python Super"
http://www.artima.com/weblogs/viewpost.jsp?thread=237121

Python allows multiple inheritance and has a strict left-to-right search for matching methods/attributes. This can work to your advantage, but often leads to complex relationships and unintended consequences. It's very complex and not for the faint-hearted – avoid if at all possible.

It is possible to define an abstract base class / interface in Python.

An interface is a little like a blank book: it's a class with a set of method definitions that have no code.

An abstract class is the same thing, but not all functions need to be empty; some can have code.

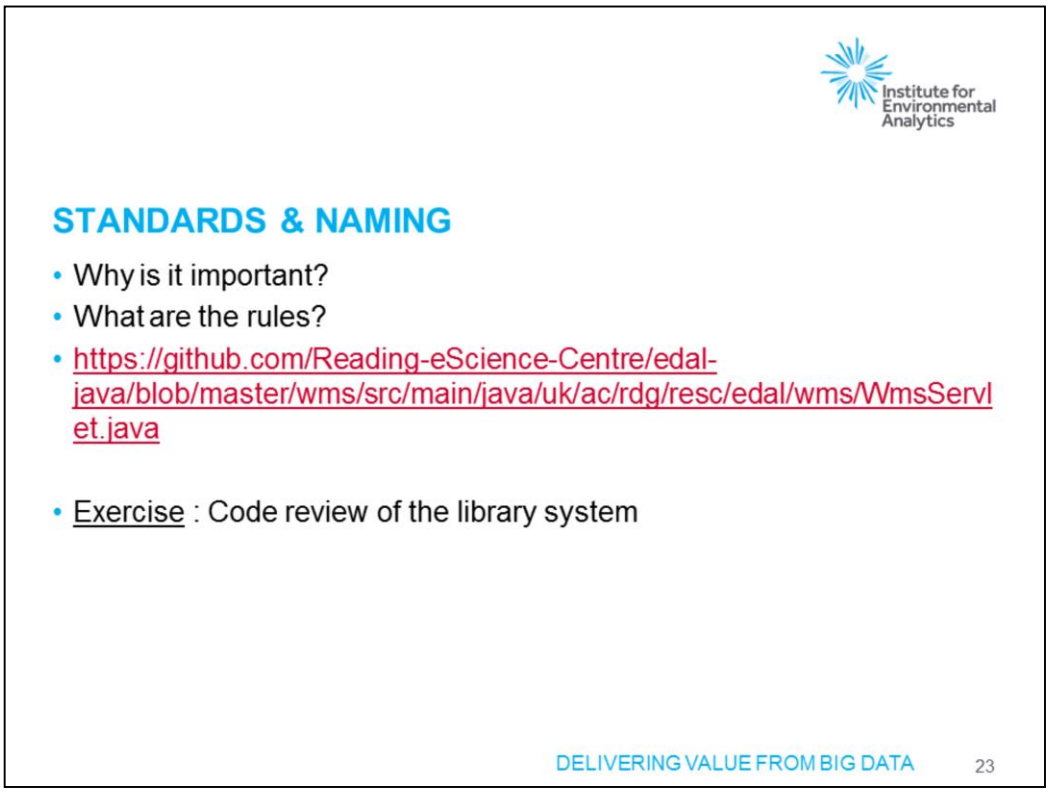
If you inherit from either, you need to define the empty methods to create a concrete class.

http://stackoverflow.com/questions/372042/difference-between-abstract-class-and-interface-in-python

Defining an Interface or Abstract Base Class

https://www.safaribooksonline.com/library/view/python-cookbook-3rd/9781449357337/ch08s12.html

https://docs.python.org/3/library/abc.html



Meaningful naming is arguably even more important in OO than in procedural programming. If done properly, most code will read like a piece of prose and will be self explanatory. When done badly, it's worse than confusing.

Typically, class names are nouns which encapsulate the job of the class: NetCDFFileReader

Attributes are names which explain their purpose: fileLocation

Methods are actions: readFile( ), processSSTData( )

or checks: isOpen( ), isValid( )

or data setters and getters: setFileLocation( ), getFileLocation( )

You already know that comments are important, so in the OO world they are used to describe a class and provide details of all the public methods available on it up front, in the case of Python, it's in the docstring. Private methods aren't exempt from comments, it's just that you'd not put them in the docstring, also you should keep attribute comments local too.

Look up the rules for the language you use.