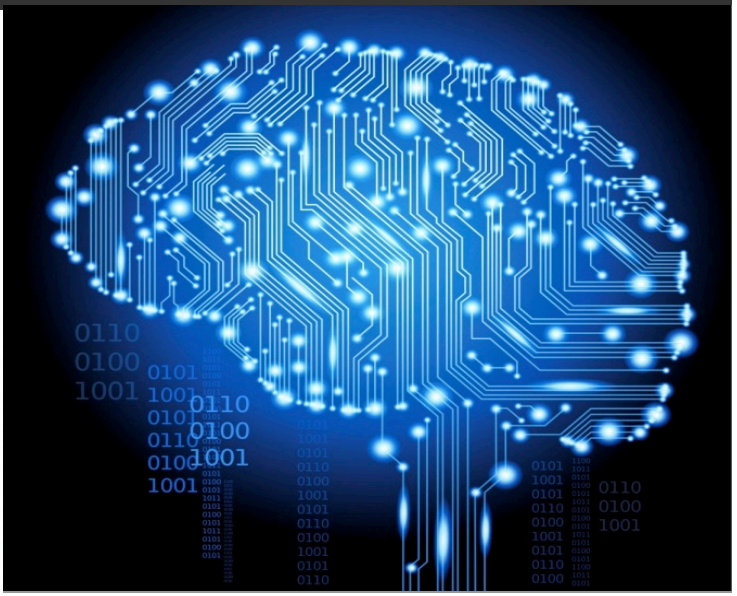
MONASH University

Information Technology

FIT1008/2085
Object Oriented Programming

Prepared by:
Maria Garcia de la Banda
Revised by D. Albrecht, J. Garcia



Objectives for this lesson

- **To learn the basics of Object Oriented (OO) Programming**
  - What are classes, objects and instances
- **To learn, in the context of Python:**
  - How to define basic classes containing both data and methods
  - How to instantiate them to create objects
  - How to access their attributes
- **To learn the absolute basics of the main OO component: inheritance**
  - To understand when and how to use it
- **To understand the main advantages of OO**

MONASH University

Defining Classes

Objects

- **Objects in computer science are similar to real life objects:**
  - They not only have associated data fields (or data attributes)
  - They also have methods that can be performed by/to the object
- **Example: a human "object" would have:**
  - Data describing the human (by its attributes): Name, age, gender, height, race, etc
  - Methods that can be performed by/on the human: Eat, sleep, run, study, die, etc
  - Both form the attributes of the object
  - Python's attributes are called "properties" in Js
- **In Python all values (e.g., integers like 3, and lists like [1,2,3]) are objects**
  - They always reside in dynamic memory (in MIPS they would be in the heap)
- **In contrast, JS has six "primitive types" (string, number, boolean, null, undefined, symbol), which do not follow OO principles**

Some of these data attributes change with time, others don't

Some methods change the state of the human's data attributes, some don't

Classes

In Python you cannot create objects without a class!

- **A class is a blueprint for building objects**
- **That is, it defines the object's attributes:**
  - The data that it can store
  - The methods it can use
  - How its variables are initialised

So, kind of a combination of constructor and prototype in JS

Class syntax now also in JS for prototypes

- **Classes are defined using the following syntax:**

```
class ClassName:
    <statement-1>
    .....
    <statement-N>
```

`class` is a keyword

The body is indented

Class header

Class names in Python start follow CapWords convention

Class body

where the most common statement is a method definition

- **Objects of type `ClassName` are created by instantiating class `ClassName`**

In Python, class and type are synonyms

Methods

- **Define operations that can be performed by any instance of the class**
  - That is, by all objects of that type
- **A method definition looks like a function definition**
- **Except that:**
  - It must appear inside the class
  - Its first argument must be a reference to the instance whose method was called
    - By (a very strong) convention, this argument is named **`self`**
    - You must explicitly add **`self`** to the method definition
    - But it is automatically added in a method call

The `__init__` method

Similar to the constructor in JS

- **By convention, (if defined) it is the first method in a class**
  - First code executed when creating an instance (code is executed automatically!)
  - Its aim is to initialize the instance variables (does NOT construct the object)
- **Its first argument is, as usual, `self`**
- **Example: the `Point` class**
  - It has two instance variables:
    - `self.xCoord`
    - `self.yCoord`
  - They start with `self.` and thus, they belong to the instances, not to the class:
    - Their value can (and usually will) be different between instances
  - But their name is global (i.e., accessible) to all methods in the class
  - If a variable doesn't start with `self.` it is either a parameter (`x` and `y`) or local to the bock that binds it (see next lecture for namespaces)

```
class Point:
    def __init__(self, x: float, y: float) -> None:
        self.xCoord = x
        self.yCoord = y
```

Don't forget `self`

No need to hint the type of `self`, as it must be `Point`

MONASH University

Instantiating and Using Classes

How do we instantiate a class?

- **Easy: call the class as if it were a function with the `__init__` arguments**

```
class Point:
 def __init__(self, x: float, y: float) -> None:
        self.xCoord = x
        self.yCoord = y
```

Let's put this code in file **`point.py`**

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p2 = point.Point(-4,7)
>>>
```

Integers are also considered floats

Why is **`point.`** needed? This is qualifying. It is needed to know where to access the class **`Point`**

No need to call **`__init__`** Done automatically when creating the instance

Accessing the object's (or instance) attributes

- **Notation:**
  - Some say an instance is the same as an object (i.e., they are synonyms)
  - Others say an object is the concept, while an instance is a particular object (in memory), or even the name (of the variable) given to an object
  - For simplicity, we will treat instances and objects as synonymous
- **So, how do we access the attributes (data and methods) of an instance?**
  - Using the "**dot**" notation which is common to many programming languages

```
>>> "abcd".upper()
'ABCD'
>>> x = [1,2,3,4]
>>> x.append(5)
>>> x
[1, 2, 3, 4, 5]
```

`"abcd"` and `[1,2,3,4]` are instances

`x.append` and `"abcd".upper` give us access to their `append` and `upper` methods

- **Dot notation is also referred to as "qualifying"**
  - Provides the context needed to access whatever we are looking for (e.g., `upper`)

Will see in detail when we discuss scoping

How do we use the instances of a class?

- **Let's continue the previous example**

```
class Point:
 def __init__(self, x: float, y: float) -> None:
        self.xCoord = x
        self.yCoord = y
```

Let's put this code in file **point.py**

No need to call **__init__** Done automatically when creating the instance

Every class instance has some built-in attributes like **__class__** We will see later why

```
>>> import point
>>> p1 = point.Point(1,3)
>>> p2 = point.Point(-4,7)
>>> p1.xCoord
1
>>> p1.yCoord
3
>>> p2.xCoord
-4
>>> p2.yCoord
7
>>> p1.__class__
<class 'point.Point'>
```

Why is **point.** needed? This is qualifying. It is needed to know where to access the class **Point**

Integers are also considered floats

p1. notation allows us to access the instance's attributes

Recap of OO

- **We create a class:**
  - By simply writing `class Name:` in some file
  - And adding indented statements (such as methods) to it
    - All methods have `self` as first argument
- **We create an object:**
  - By instantiating the class
  - That is, by simply calling `Name` with the appropriate arguments
    - Those used by method `__init__`
  - The object has access to all the attributes defined by the class using the "dot" notation (e.g., `the_list.append`)
- **A big advantage of OO: encapsulation**
  - All data and methods of any object are clearly defined within its class

And if the class does not have __init__?

- **The `__init__` method is not mandatory**
- **One could easily define:**

```
>>> class Silly:
...     i = 8
...
>>> Silly.i
8
>>> s1 = Silly()
>>> s1.i
8
>>> s2 = Silly()
>>> s2.i
8
```

A class variable. Defined in a class but outside any method. Its value is shared by all instances of the class

Belongs to the class. Thus, it exists without object and is accessed through the class

Can also be accessed through an instance (see later)

All instances of the class have the same value for `i`

```
>>> Silly.i = 11
>>> s1.i
11
>>> s2.i
11
>>> s1.i = 6
>>> s1.i
6
>>> s2.i
11
```

To modify `i` you must do it through the class not through an instance!

See next lesson to see why

What? if `i` is a class variable, shouldn't it be 6?

Class example

- **Let's write a Rectangle class:**
  - Instance variables for
    - Base (float)
    - Height (float)
  - Methods to compute values:
    - Area
    - Perimeter
  - Method to modify object:
    - Scale the object up/down

In MIPS: leave value in $v0

In MIPS: simply modify the input data in the heap

Again, no space for commenting the code, but you must!

```
class Rectangle:
    def __init__(self, base:float, height:float) -> None:
        self.base = base
        self.height = height

    def area(self) -> float:
        return self.base * self.height

    def perimeter(self) -> float:
        return 2 * self.base + 2 * self.height

    def scale_size(self, scale: float) -> None:
        self.base *= scale
        self.height *= scale
```

Return value

Returns nothing; modifies the instance

MONASH University

Inheritance

Inheritance

Multiple inheritance! Can inherit from more than one class

- **Mechanism to define a class (called child or derived class) that:**
  - Inherits all attributes of its parent (or base) classes (and all ancestors)
  - Can add more data and/or more methods
  - Can change some of the methods (called override them)
- **When to use:**
  - Often, when the child class is a special case of the parent one (is-a relationship)
  - Example:
    - First define a base class **`Polygon`** with common methods (**`perimeter`**, **`area`**, etc)
    - Then define child classes for **`Rectangle`**, **`Triangle`**, etc
- **Main advantages of using inheritance in OO:**
  - Common interface (**`perimeter`**, **`area`**, etc) - with perhaps extra data/methods
  - Reuse implementation as much as possible (instead of duplicating code)
  - Share the testing cases

Syntax for Inheritance

- **Very easy:**

```
class ParentClass:
    BODY OF THE ParentClass
class ChildClass(ParentClass):
    BODY OF THE ChildClass
```

Note that the parent class appears here

Can have several parents, separated by commas

- **Can the body of the child class define a method that appears in the parent?**
  - Yes! If so, the child method overrides the parent one
    - How does this work exactly? Via **scoping** (which will see in the next lesson)
  - Note: the child often extends methods. For example, the child method might:
    - First call the parent method to process the parent instance variables
    - Then execute extra code to process the child-specific instance variables

The `Object` class

- **All classes you define are derived from a built-in class `Object`**
- **The class `Object` has many generic methods, including:**

Operation	Class Method
`str(obj)`	`__str__(self)`
`len(obj)`	`__len__(self)`
`item in obj`	`__contains__(self,item)`
`y = obj[ndx]`	`__getitem__(self,ndx)`
`obj[ndx] = value`	`__setitem__(self,ndx,value)`
`obj == rhs`	`__eq__(self,rhs)`
`obj < rhs`	`__lt__(self,rhs)`
…	
`obj + rhs`	`__add__(self,rhs)`
`...`	

Intuitively: if the object defines the method being called, Python executes it. If not, Python iteratively looks to the parent, its parent, etc, until it finds it (possibly going all the way to class `Object`.) or does not find it and raises an exception.

For details, see next lesson.

- **To override them, simply re-define the method inside your class**

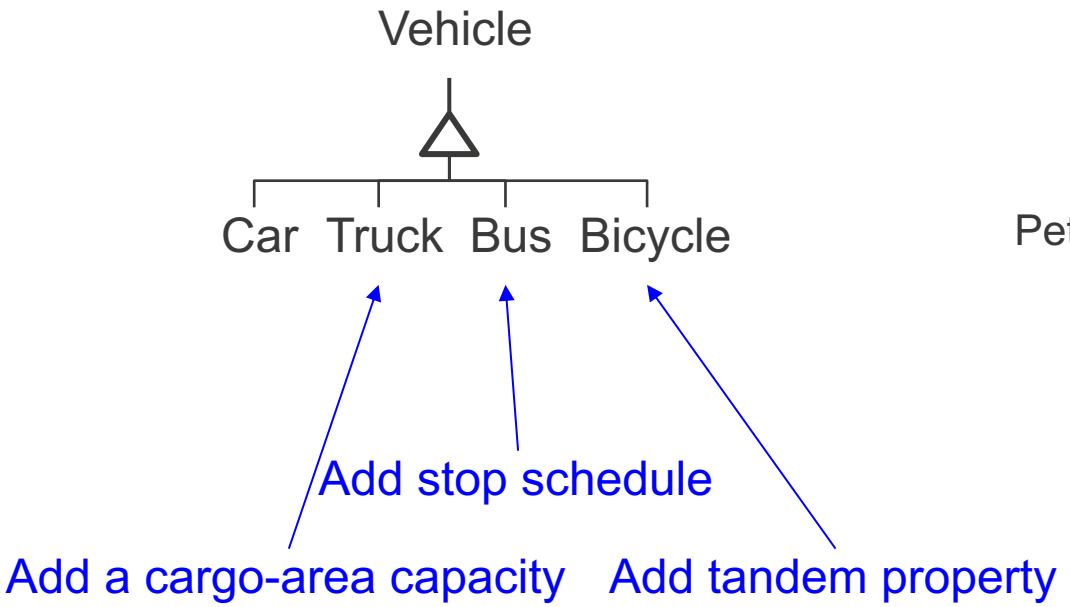
Example: Vehicles and their general properties

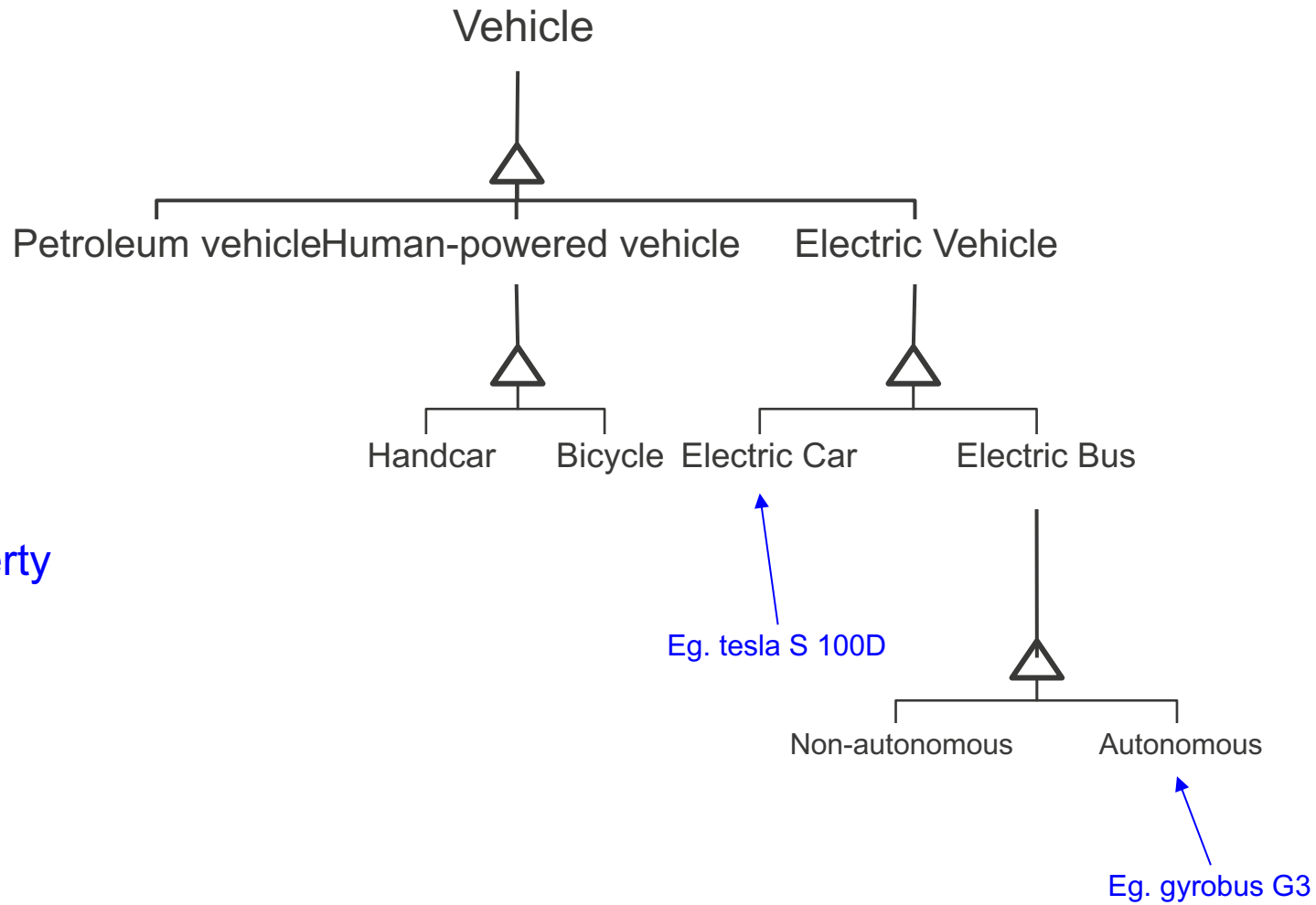
- **Let's consider some properties of common vehicles:**

	Car	Truck	Bus	Bicycle	Tram
Can travel on roads	y	y	y	y	n
Moves from A to B	y	y	y	y	y
Fuel capacity	y	y	y	n	n
Number of wheels	4	>=4	>=4	2	>= 4

- **Cars and trucks will be doing very similar tasks at a high level**
- **A lot of information is similar between them**
- **The same is true of buses, bicycles and a number of other vehicles**
  - each can be seen as a subclass (directly or not) of the vehicle class

Example: Possible Vehicle hierarchies





Vehicle class and Car class in Python

```
class Vehicle:
    def __init__(self, wheels:int, capacity:float, brightness:float, coordinates:float): -> None
        self.wheels = wheels
        self.fuel_capacity = capacity
        self.head_light_brightness = brightness
        self.coordinates = coordinates

    def can_drive(self, _: Route) -> bool:
        pass  # can't be implemented at this abstract stage

        # ... other methods
```

No body!

```
class Car(Vehicle):
    def __init__(self, capacity:float, brightness:float, coordinates:float, registration:int) -> None:
        Vehicle.__init__(self, 4, capacity, brightness, coordinates)
        self.registration = registration


    def __eq__(self, other: Car) -> bool:  # replace from Object
        return self.registration == other.registration

    def can_drive(self, _: Route) -> bool:  # replace from Vehicle
        # ...
        # Actual definition
        return True
```

Extra instance variable

This is used to access **Vehicle**'s method. Can also use **super(..)** which is more general but can do surprising things, so we will stick with simple qualifying

Object defines **==** as **True** if they have the same address. This is much more accurate

Vehicle class and Truck class in Python

```
class Vehicle:
    def __init__(self, wheels:int, capacity:float, brightness:float, coordinates:float): -> None
        self.wheels = wheels
        self.fuel_capacity = capacity
        self.head_light_brightness = brightness
        self.coordinates = coordinates

    def can_drive(self, _: Route) -> bool:
        pass  # can't be implemented at this abstract stage

        # ... other methods
```

```
class Truck(Vehicle):
    def __init__(self, wheels:int, capacity:float, brightness:float, coordinates:float,
                 registration:int, cargo_capacity:float, height:float) -> None:
        Vehicle.__init__(self, wheels, capacity, brightness, coordinates)
        self.registration = registration
        self.cargo_area_capacity = cargo_capacity
        self.height = height

    def __eq__(self, other: Truck) -> bool:  # replace from Object
        return self.registration == other.registration

    def can_drive(self, route: Route) -> bool:  # replace from Vehicle
        return self.height < route.clearance()
```

`Vehicle` used again

Some instance variables have the same name (will be different in other subclasses of `Vehicle`), some are not

Similar definition to `Car`

Different definition than `Car`

Abstract Methods and Abstract Classes

- **A method is abstract if it is declared but has no implementation (just `pass`)**
  - For example, **`can_drive`** in class **`Vehicle`**
- **A class is abstract if it has at least one abstract method (like `Vehicle`)**
- **An abstract class should not be instantiated**
  - But Python does not have native abstract classes
  - As a result, you can instantiate **`Vehicle`**, which is not a good idea
- **Python does have a module (`abc`) that provides the required infrastructure:**
  - You should import it as: **`from abc import ABC, abstractmethod`**
  - Then use class **`ABC`** as parent of the abstract class
  - And use **`@abstractmethod`** as a decorator of each abstract method
- **Classes derived from an abstract class cannot be instantiated unless all abstract methods are overridden**

For our example Vehicle class

Changes in red

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    def __init__(self, wheels:int, capacity:float, brightness:float, coordinates:float): -> None
        self.wheels = wheels
        self.fuel_capacity = capacity
        self.head_light_brightness = brightness
        self.coordinates = coordinates
    @abstractmethod
    def can_drive(self, _:Route) -> bool:
        pass  # can't be implemented at this abstract stage

        # ... other methods
```

Import

Parent class

Decorator

Summary

- **We now know the basis of Object Oriented (OO) Programming**
  - What are classes, objects, instances and the inheritance mechanism
- **We are able, in the context of Python, to:**
  - Define simple classes containing both data and methods (the attributes)
  - Instantiate them to create objects
  - Access their attributes
- **We know the basics of OO's inheritance and, in particular, how to define:**
  - Derived classes that may call the parent class and/or override its methods
  - Abstract classes and abstract methods
- **We know and understand the main advantages of OO**