

## OOP introduction (1/2)

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

[alberto.manfreda@pi.infn.it](mailto:alberto.manfreda@pi.infn.it)

INFN–Pisa



## Smart variables

- ▷ Working with containers like lists or dictionaries, you may have noticed that they can do many things besides holding the data
  - ▷ You can extend a list using `append()` or `insert()`
  - ▷ Trying to access a non-existent index in a list triggers a specific error (`IndexError`)
  - ▷ You can iterate on a list using the handy for-loop Python syntax
  - ▷ and so on...
- ▷ In other words, a list is a variable that, in addition to its data, shows some kind of specific *behaviour*.
- ▷ How is that implemented?



# Object Oriented Programming (OOP)

- ▷ Programming paradigm based on *objects*
- ▷ An object is a code entity that has:
  - ▷ **State** → data (usually called *member variables* or *attributes*)
  - ▷ **Behaviour** → functions (usually called *member functions* or *methods*)
- ▷ The idea is: keep together the variables and the code that manipulates them
- ▷ As the name suggests, OOP is very often adopted for modelling systems that are naturally described in terms of objects
  - ▷ Physical simulations
  - ▷ Graphical engines
  - ▷ Graphical User Interfaces (GUI)



## Why should I care?

- ▷ Object-oriented programming is one of the most widely used paradigm today
- ▷ That doesn't necessarily mean it is the best one – nor the right tool for any job
  - [https://en.wikipedia.org/wiki/Object-oriented\\_programming#Criticism](https://en.wikipedia.org/wiki/Object-oriented_programming#Criticism)
- ▷ There is a number of famous programmers which really dislike it (e.g. *Linus Torvalds*)
- ▷ Still, it is something you definitely want in your toolbox
- ▷ And there is a fairly good chance that you will have to interact with some OOP code in your life



# A good reason for learning OOP

- ▷ (Almost) every popular languages nowadays is OO

Sep 2021	Sep 2020	Change	Programming Language	Ratings	Change
1	1		C	11.83%	-4.12%
2	3	▲	Python	11.67%	+1.20%
3	2	▼	Java	11.12%	-2.37%
4	4		C++	7.13%	+0.01%
5	5		C#	5.78%	+1.20%
6	6		Visual Basic	4.62%	+0.50%
7	7		JavaScript	2.55%	+0.01%
8	14	▲	Assembly language	2.42%	+1.12%
9	8	▼	PHP	1.85%	-0.64%
10	10		SQL	1.80%	+0.04%
11	22	▲	Classic Visual Basic	1.52%	+0.77%
12	17	▲	Groovy	1.48%	+0.48%
13	15	▲	Ruby	1.27%	+0.03%
14	11	▼	Go	1.13%	-0.33%
15	12	▼	Swift	1.07%	-0.31%
16	16		MATLAB	1.02%	-0.07%
17	37	▲	Fortran	1.01%	+0.65%
18	9	▼	R	0.98%	-1.40%

<https://www.tiobe.com/tiobe-index/#null>

<https://www.tiobe.com/tiobe-index/>

- ▷ Besides C of course!



## OOP: Classes and Objects

- ▷ Basic definitions:
  - ▷ A **class** is a blueprint for creating objects
  - ▷ An **object** is a concrete realization of a class
- ▷ You can imagine a class like a project, which is used to describe how objects are built and how they work
- ▷ You can have multiple objects of the same class
- ▷ The relationship is similar to the one between types and variables:
  - ▷ A type is an abstract concept, describing how a variable is represented in memory
  - ▷ A variable is a concrete realization of it
  - ▷ You can have several variables of the same type (like several integers or several strings)
- ▷ Indeed, to some extent, a class is the generalization of the concept of type. It specifies not only how an object *is made* but also how *it behaves*.



## OOP: an example



- ▷ Let's consider a familiar object, like a television. It has:
  - ▷ A state
    - ▷ On/off (and possibly standby)
    - ▷ Currently displayed channel
    - ▷ Volume
    - ▷ Brightness, contrast, etc...
  - ▷ A behaviour
    - ▷ Pressing the 'power' button will turn ON/OFF
    - ▷ Rotating the volume knob will increase/decrease the volume
    - ▷ Using the buttons on the remote control will change displayed channel, brightness, contrast etc...
    - ▷ And don't forget you need to plug-in before use!



## OOP: an example

- ▷ How would that be represented in the code?
  - ▷ The state can be represented by some **attributes** (variables):
    - ▷ A boolean can represent the ON/OFF state
    - ▷ For the currently displayed channel you can use an integer
    - ▷ Volume, contrast, luminosity etc... they all get their own variable(s)
  - ▷ The behaviour can be implemented through the **methods**:
    - ▷ For example the turn\_on() and turn\_off() functions may change the value of the variable and also produce all the related changes (i.e. start/stop video and audio)
    - ▷ You will probably have the next\_channel() and previous\_channel() functions for zapping and so on...
    - ▷ Of course it can be much more complex than that!
- ▷ Attributes and methods are collectively called **members** of the class
- ▷ Each object of a specific class is an **instance** of that class



## Python classes

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class\\_tv\\_basic.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_tv_basic.py)

```
1 # Here we define the class
2 class Television:
3     """ Television class. I will follow the convention of starting class names
4     with an uppercase. """
5     pass # oops we have no code yet!
6
7     """To create instances of a class in python we use the parenthesis operator '()' .
8     The syntax is similar to calling a function -- which is actually what is
9     happening behind the scenes, as we will see later"""
10 my_television = Television() # my_television is an instance of the class Television
11
12 print(type(my_television)) # Check its type
13
14 your_television = Television() # And this is another instance
15
16 # Let's check that they are really two different objects
17 print(my_television is not your_television)
18
19 [Output]
20 <class '__main__.Television'>
21 True
```



## Bonus

In Python everything is an object of some class!

```
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/everything_is_a_class.py
1 # Create an integer variable
2 this_is_an_int = 5
3 # Now check its type
4 print(type(this_is_an_int))
5
6 # Same with a string
7 this_is_a_string = 'Hello world!'
8 print(type(this_is_a_string))
9
10 # Same with a list
11 this_is_a_list = ['Frodo', 'Samvise', 'Meriadoc', 'Peregrino']
12 print(type(this_is_a_list))
13
14 # And even a function!
15 def this_is_a_function():
16     return 0
17
18 print(type(this_is_a_function))
19
20 [Output]
21 <class 'int'>
22 <class 'str'>
23 <class 'list'>
24 <class 'function'>
```



# Methods

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class\\_methods.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_methods.py)

```
1  class Television:
2      """ Class describing a television.
3      """
4      def turn_on(self, channel=1): # Class method
5          """All the class methods get the object instance as their first argument.
6          It is customary to call this argument 'self', though is not required
7          by the language rules (you can call it 'piippo' and it will work
8          just as well)
9          """
10     print('Turning on {}'.format(self))
11     print('Showing channel {}'.format(channel))
12
13 tv = Television()
14 # Class methods and members are accessed through the '.' (dot) operator
15 # You must not pass the 'self' argument, it is added automatically!
16 tv.turn_on(channel=3)
17
18 [Output]
19 Turning on <__main__.Television object at 0x7fc718217470>
20 Showing channel 3
```



# Attributes

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class\\_attributes\\_1.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_attributes_1.py)

```
1  class Television:
2      """ Class describing a television.
3      """
4      pass
5
6      tv = Television()
7      # Add an attribute manually, with a simple assignment
8      # Attributes are accessed through the '.' (dot) operator
9      tv.x = 1
10     print (tv.x)
11     # This attribute is not shared with other instances of the class
12     another_tv = Television()
13     print(another_tv.x)
14
15     [Output]
16     1
17     Traceback (most recent call last):
18         File "snippets/class_attributes_1.py", line 13, in <module>
19             print(another_tv.x)
20     AttributeError: 'Television' object has no attribute 'x'
```



# Attributes

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class\\_attributes\\_2.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_attributes_2.py)

```
1  class Television:
2      """ Class describing a television.
3      """
4
5      def add_an_attribute(self):
6          """ Add a class attribute (remember the meaning of 'self') """
7          self.current_channel = 1
8
9      tv = Television()
10     # Add an attribute from inside a class method
11     tv.add_an_attribute()
12     print(tv.current_channel)
13
14     # Again, attributes are not shared
15     another_tv = Television()
16     another_tv.add_an_attribute()
17     # Changing the attribute for one will not affect other instances of the class
18     tv.current_channel = 5
19     # The following line will print 1, not 5
20     print(another_tv.current_channel)
21
22 [Output]
23 1
24 1
```



## Constructor

- ▷ Adding attributes like that would be crazy... what would happen if I forgot to call the 'add\_a\_class\_attribute()' method in the previous example?
- ▷ Luckily there is a solution for that: the class **constructor**
- ▷ The constructor is a special method that is called automatically each time a class instance is created
- ▷ A specificity of the constructor is that it cannot return anything
- ▷ In Python the constructor is the `__init__` method<sup>1</sup>
- ▷ Class methods like `__init__`, with the name surrounded by two underscores, are called **special** methods or **dunder** methods.
- ▷ Is good practice to define all your class attributes inside the constructor!

---

<sup>1</sup>Actually the real constructor – that is the function responsible for creating the class instances – is the `__new__` operator, but 99% of the time you don't need to define that, as all classes have a default one which does the job for you



# Constructor

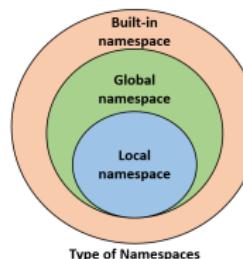
```
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_constructor.py

1  class Television:
2      """ Class describing a television.
3      """
4
5      def __init__(self, owner):
6          """ The special method __init__ is called each time a class instance is
7              created. We can pass arguments to the constructor, just like any
8              function."""
9
10     print('Creating a television instance...')
11     self.model = 'Sv32X-553T' # This class attribute is hard-coded
12     self.owner = owner # This is set to the value of the argument
13
14     def print_info(self): # Let's see
15         """ Print the model and owner"""
16         message = 'This is television model {}, owned by {}'
17         print(message.format(self.model, self.owner))
18
19     my_television = Television('Alberto')
20     my_television.print_info()
21     batman_television = Television('Batman')
22     batman_television.print_info()
23
24 [Output]
25 Creating a television instance...
26 This is television model Sv32X-553T, owned by Alberto
27 Creating a television instance...
28 This is television model Sv32X-553T, owned by Batman
```



## Digression - namespaces

- ▷ ‘Namespaces are one honking great idea – let’s do more of those!’ (*Tim Peters - The Zen of Python*)
- ▷ A **namespace** in Python is essentially a dictionary of *unique names*, each one associated to an object (which can be anything: a variable, a function, a class etc...)



- ▷ Python creates separate namespaces for many things: for example, each time a function is called a namespace for local variables is created
- ▷ You can access objects in the local namespace (and those above – see picture) just with their name(s); for others you need the '.' (dot) operator.
- ▷ The space of visibility of a variable is called its **scope**.



## Instance attributes vs class attributes

- ▷ Each class has a namespace. Plus, each instance of the class get its own additional namespace
- ▷ The class namespace is automatically visible from each instance namespace, but not the opposite
- ▷ An attribute in an instance namesespace is an **instance attribute**, and cannot be seen or modified by other instances of the class
- ▷ An attribute in the class namespace is a **class attribute** and is shared among all the instances
- ▷ Since class attributes are not related to a specific instance, they can be accessed without creating one!
- ▷ Class attributes are useful to set class constants, or default values, or share data among instances



# Class attributes

(And their strange behaviour)

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class\\_attributes\\_3.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_attributes_3.py)

```
1  class Television:
2      """ Class describing a television.
3      """
4
5      NUMBER_OF_CHANNELS = 999 # This is a class attribute
6
7      # We don't need an instance to access class attributes
8      print(Television.NUMBER_OF_CHANNELS)
9      # But we can also access it through instances
10     tv = Television()
11     print(tv.NUMBER_OF_CHANNELS)
12
13     # Changing the attribute in the class namespace will change it for every instance
14     another_tv = Television()
15     Television.NUMBER_OF_CHANNELS = 998
16     print(another_tv.NUMBER_OF_CHANNELS)
17     # But assigning to that attribute in an instance namespace will create a copy!
18     # Result: the other instances won't be affected!
19     tv.NUMBER_OF_CHANNELS = 997
20     print(another_tv.NUMBER_OF_CHANNELS)
21
22 [Output]
23 999
24 999
25 998
26 998
```



## Short summary

- ▷ Object Oriented Programming (OOP) is a widespread programming paradigm, supported by many programming languages (old and modern), including Python
- ▷ An object has a state and a behaviour, represented by member variables (attributes) and member functions (methods) respectively
- ▷ A class is a blueprint for creating objects, each object is an instance of a class
- ▷ In Python classes are defined with the 'class' keyword and instantiated with the '' operator
- ▷ Class attributes and methods (globally called members) are accessed through the '.' operator
- ▷ All the class methods get the object instance as their first argument (usually named 'self')
- ▷ You should declare class attributes in the constructor a.k.a. the `__init__` function
- ▷ Instance attributes are not shared: each instance has its own copy of the data
- ▷ Class attributes are declared outside methods and are shared among all the instances of a class



## Encapsulation - hidden state and interfaces

Back to the TV example



- ▷ Note that part of the state is hidden from the user:
  - ▷ E.g. internal switches, transistors, etc...
- ▷ You do not need to know what's going on inside the case to operate a TV!
- ▷ All you need to know is how to use the **interface** (the remote control, the knobs, the power button, the plug...)
- ▷ The **implementation** details are hidden: only the TV producer cares about them, not the user.



## Encapsulation

- ▷ This leads us to the concept of **encapsulation**
  - ▷ *The state of an object should only be accessed and altered through its publicly exposed interface*
- ▷ That way it is easier to find bugs: you know that, if something is wrong with an object, the problem lays inside the class code
- ▷ That way you can also *enforce behaviour*: for example you can prevent from changing the channel if the TV is off



# Enforcing behaviour

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class\\_tv\\_zapping.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_tv_zapping.py)

```
1  class Television:
2      """ Class describing a television.
3      """
4
5      def __init__(self):
6          """ Class constructor"""
7          self.is_on = False
8          self.current_channel = 1
9
10     def turn_on(self):
11         """ Turn on the tv (I omit the turn_off() method for brevity) """
12         print('Turning on!')
13         self.is_on = True
14
15     def next_channel(self):
16         """ Go to next channel. Works only if the tv is on! """
17         if (self.is_on):
18             self.current_channel += 1
19
20     tv = Television()
21     tv.next_channel() # This will do nothing
22     print(tv.current_channel)
23     tv.turn_on()
24     tv.next_channel() # This will work
25     print(tv.current_channel)
26
27 [Output]
28 1
29 Turning on!
30 2
```



## Challenge: how do you get *real* encapsulation?

- ▷ At this point you may be wondering: I can read and modify any class attribute from outside the class using the '.' (dot) operator! Doesn't that break encapsulation?
- ▷ Answer: Yes it does - but there are ways to fix it!



## Pythonic encapsulation

- ▷ In languages like C++ you can explicitly declare that some class attributes (and methods) are *private*
- ▷ In Python there is no concept of *enforced* private attributes
- ▷ However, there exists a convention that any attribute/method name prepended by one or two underscore(s) should be considered "private"
- ▷ It's like a warning for the class user: you should never access that directly!
- ▷ In the case of two underscores Python will actually do a subtle thing to help keeping the data private – it will prepend `_classname` to the actual attribute name (see next example)
- ▷ However, not everyone in the Python community loves that
- ▷ "Never, ever use two leading underscore. This is annoyngly private"  
(Alex Martelli, member of the Python Software Foundation, author of 'Python in a Nutshell' and co-author of 'The Python cookbook')



## "Private" attributes in Python

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class\\_tv\\_private.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_tv_private.py)

```
1  class Television:
2      """ Class describing a television.
3      """
4      def __init__(self, owner):
5          """ Class constructor"""
6          # Single underscore - tells the user he shouldn't access the variable
7          # directly outside the class
8          self.__model = 'Sv32X-553T'
9          # Double underscore - python will prepend _Television to the name
10         self.__owner = owner
11
12
13 tv = Television('Alberto')
14 # The following line is bad practice, but it's technically possible
15 print(tv.__model)
16 # Even with two underscores I can still access it if I know the "trick"
17 print(tv._Television__owner)
18
19 [Output]
20 Sv32X-553T
21 Alberto
```



## Pythonic encapsulation with properties

- ▷ The possibility of making variables "private" (enforced or not) is not enough of course, because sometimes we still want to let the user read or even modify the value of the attribute
- ▷ The "old" solution for that is providing access functions (the infamous "getters/setters")
- ▷ But the awesome solution is using **properties** (since Python 2.2)
- ▷ Properties look similar to getters and setters, but with a twist: you keep accessing the attribute with the dot operator
- ▷ In order to understand why this makes a *huge* difference let's start with an example: suppose you have a class for 2-D vectors



## Example of a class without encapsulation

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d\\_xy.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_xy.py)

```
1 import math
2
3 class Vector2d:
4     """ Class representing a Vector2d. We use float() to make sure of storing
5        the coordinates in the correct format """
6     def __init__(self, x, y):
7         self.x = float(x)
8         self.y = float(y)
9
10    def module(self):
11        return math.sqrt(self.x**2 + self.y**2)
12
13    def angle(self):
14        return math.atan2(self.y, self.x)
15
16 v = Vector2d(3., -1.)
17 print(v.x, v.y)
18 print(v.module(), v.angle())
19
20 [Output]
21 3.0 -1.0
22 3.1622776601683795 -0.3217505543966422
```



## Changing your class

- ▷ Suppose you later realize that you use your Vector2d a lot for performing rotations
- ▷ It would be much faster to store the angle and the module, instead of x and y, as the rotation would reduce to a simple addition of the angles
- ▷ You may think of rewriting your class in that way...



## Changing your class

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d\\_rtheta.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_rtheta.py)

```
1 import math
2
3 class Vector2d:
4     """ Class representing a Vector2d. We use float() to make sure of storing
5        the coordinates in the correct format """
6     def __init__(self, module, angle):
7         self.module = float(module)
8         self.angle = float(angle)
9
10    def x(self):
11        return self.module * math.cos(self.angle)
12
13    def y(self):
14        return self.module * math.sin(self.angle)
15
16 v = Vector2d(3.1622776601683795, -0.3217505543966422)
17 print(v.module, v.angle)
18 print(v.x(), v.y())
19
20 [Output]
21 3.1622776601683795 -0.3217505543966422
22 3.0 -1.0
```



## The problem

- ▷ ...now, however, you have a big problem: your old code is broken!
- ▷ In every place where you were calling `v.x` or `v.module()` now you get an error that needs to be fixed
- ▷ If other people use your code that is even worse, because you are breaking *their* code
- ▷ Without properties the solution would have been to design the class from the start with private attributes and access functions



## Old-style encapsulation: never do that!

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d\\_old\\_enc.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_old_enc.py)

```
1 import math
2
3 class Vector2d:
4     """ Class representing a Vector2d. Old-style encapsulation."""
5     def __init__(self, x, y):
6         self._x = float(x)
7         self._y = float(y)
8
9     def x(self):
10        return self._x
11
12    def y(self):
13        return self._y
14
15    def module(self):
16        return math.sqrt(self._x**2 + self._y**2)
17
18    def angle(self):
19        return math.atan2(self._y, self._x)
20
21 v = Vector2d(3., -1.)
22 print(v.x(), v.y())
23 print(v.module(), v.angle())
24
25 [Output]
26 3.0 -1.0
27 3.1622776601683795 -0.3217505543966422
```



## Enter properties

- ▷ The class data are now "encapsulated", but that is still not ideal:
  1. You have written a lot of code just to provide access to a bunch of variables
  2. You will have to write even more methods if you want to let the user modify those variables as well (e.g. `set_x()`, `set_y()` and so on)
  3. You have to write all this code right from the start, even if you never need it, otherwise you run the risk of getting screwed later
- ▷ **properties** solve the problem by *emulating attributes*



## Properties to emulate attributes

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d\\_property.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_property.py)

```
1 import math
2
3 class Vector2d:
4     """ Class representing a Vector2d - storing module and angle."""
5     def __init__(self, module, angle):
6         self.module = float(module)
7         self.angle = float(angle)
8
9     @property
10    def x(self):
11        return self.module * math.cos(self.angle)
12
13    @property
14    def y(self):
15        return self.module * math.sin(self.angle)
16
17 v = Vector2d(3.1622776601683795, -0.3217505543966422)
18 print(v.module, v.angle)
19 # We don't need to call v.x() anymore - we can simply use v.x!
20 print(v.x, v.y)
21
22 [Output]
23 3.1622776601683795 -0.3217505543966422
24 3.0 -1.0
```



## Setter properties

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d\\_property\\_setter.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/vector2d_property_setter.py)

```
1 import math
2
3 class Vector2d:
4     """ Class representing a Vector2d - storing module and angle."""
5     def __init__(self, module, angle):
6         self.module = float(module)
7         self.angle = float(angle)
8
9     @property
10    def x(self):
11        return self.module * math.cos(self.angle)
12
13    @property
14    def y(self):
15        return self.module * math.sin(self.angle)
16
17    @x.setter
18    def x(self, x): # this function must be called as the property
19        """ Here we actually need to update module and angle"""
20        self.module, self.angle = math.sqrt(x**2 + self.y**2), \
21                                math.atan2(self.y, x)
22
23 v = Vector2d(3.1622776601683795, -0.3217505543966422)
24 print(v.x)
25 v.x = 1.
26 print(v.x)
27
28 [Output]
29 3.0
30 1.0000000000000002
```



## Make attributes read-only using properties

```
os://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_tv_encapsulation_properties

1  class Television:
2      """ Class describing a television.
3      """
4
5      def __init__(self, owner):
6          """ Class constructor"""
7          self._owner = owner # owner is private
8
9      @property
10     def owner(self):
11         return self._owner
12
13     @owner.setter
14     def owner(self, new_owner):
15         """ Make the attribute read-only by acting on the setter"""
16         print('Nope {}. Do you want to steal my tv?'.format(new_owner))
17
18     tv = Television('Batman')
19     print('This tv belongs to {}'.format(tv.owner))
20     tv.owner = 'Joker'
21     print('This tv belongs to {}'.format(tv.owner))
22
23 [Output]
24 This tv belongs to Batman
25 Nope Joker. Do you want to steal my tv?
26 This tv belongs to Batman
```



## Final note on properties

- ▷ Bottom line: when writing a class, you don't need to make attributes private right from the start
- ▷ You can start with public attributes, and use properties later to enforce access/modification rules
- ▷ That way you only write the code that you really need



# Interface vs implementation mindset

physicists vs programmers

- ▷ A physicist thinks:
  - ▷ "I have this super-cool algorithm to solve the problem I am working on: I will code it carefully, then put together quickly some basic interface to pass data to it and write results to screen / file. I need the results quickly for my paper; I can always improve the interface later, right?"
- ▷ A programmer thinks:
  - ▷ "I will create a nice interface for the user to handle input/output in different formats and I will try to keep it as stable as possible in the future. I will start with no algorithm at all – I will just use random numbers to test the interface. I can always implement the actual algorithm later, right?"



- ▷ You don't need to think like a programmer - doing physics is your goal - but remember that **interfaces are important**
- ▷ The concept of interface does not just apply to the program as a whole: every significant portion of code (function, class) has its interface
- ▷ The interface of a class in Python is made by all its "public" members (methods and attributes) – i.e. those without an underscore at the beginning of their name
- ▷ Changing the interface may break every other piece of code that uses it. You want to do that *as less as possible*
- ▷ You should not access "private" members directly - even if you can. Always pass through the interface



## Short summary (2)

- ▷ Encapsulation is the technique of hiding part or all the class state to the user; he can only access and modify that through the class methods
- ▷ Encapsulation helps debugging by limiting the number of places in the code that can mutate the state of an object
- ▷ It can also be useful to enforce behaviour
- ▷ Encapsulation in Python is not enforced by the language, but rather relies on conventions
- ▷ Class members with an underscore at the beginning of their name are considered 'private' and should not be accessed directly outside the class
- ▷ You can use properties to encapsulate your data at any moment in time - never use 'getters' and 'setters'
- ▷ Interfaces should not change frequently!



# Inerithance

## What and Why

- ▷ Suppose for a moment that you are coding the Monte Carlo for a physics experiment
- ▷ You want to simulate interactions of charged particles in some detector using OOP paradigm
- ▷ You may have a class Detetctor and a class for each particle that you need to simulate
- ▷ Let's say you have a class Electron, a class Positron, a class Proton and a class Alpha
- ▷ If you think about it, these classes will have a lot of code in common
- ▷ For example they all need to store their mass, charge, position, velocitiy (or momentum), possibly spin etc...
- ▷ They may also have similar behaviour, though that is less obvious
- ▷ We know that duplicate code is evil (DRY): how do we avoid that?



# Inerithance

## What and Why

- ▷ Many languages - including Python offer a solution for that: **inheritance**
- ▷ A class can inherit from another one, automatically obtaining all its functionalities (attributes and methods) and then extending or specializing them
- ▷ The class which we inherit from is called *Base class*, *Parent class* or (in Python) *Superclass*
- ▷ The class inheriting is called *Derived class* or *Child class*
- ▷ In our problem we can imagine to have a base class 'Particle' and many specialized classes inheriting from it
- ▷ Inheritance is transitive: if class C inherits from class B, and class B inherits from class A, then class C is also a child of class A (and posses all its functionalities)



## Inheritance: a basic example

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/inheritance.py>

```
1 import math
2
3 class Particle:
4     """ Class describing a generic particle.
5     """
6     def __init__(self, mass, charge=0, name=None, momentum=0.):
7         """ Class constructor"""
8         self.mass = mass # in MeV
9         self.charge = charge # in e
10        self.name = name
11        self.momentum = momentum # in MeV/c
12
13    def energy(self):
14        """ Return the energy of the particle in MeV/c^2"""
15        return math.sqrt(self.momentum**2. + self.mass**2.)
16
17 class Electron(Particle):
18     """ Class describing an electron. We inherit from Particle
19     """
20     def __init__(self, momentum=0.):
21         """ Derived class constructor. We call the base class constructor"""
22         Particle.__init__(self, 0.511, -1., 'e-', momentum)
23
24 el = Electron(momentum=1.)
25 print('Energy of {} is {:.4f} MeV/c^2'.format(el.name, el.energy()))
26
27 [Output]
28 Energy of e- is 1.1230 MeV/c^2
```



# Overload

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/overload.py>

```
1  class Animal:
2      def sound(self):
3          return None
4
5  class Dog(Animal):
6      def sound(self):
7          """ This will shadow the method in the base class"""
8          return 'Woof!'
9
10 class Cat(Animal):
11     def sound(self):
12         """ This will shadow the method in the base class"""
13         return 'Meow!'
14
15 class SilentAnimal(Animal):
16     pass # I make no sound
17
18 animals = [Animal(), Cat(), Dog(), SilentAnimal()]
19 for animal in animals:
20     print(animal.sound())
21
22 [Output]
23 None
24 Meow!
25 Woof!
26 None
```



# Multiple inheritance

[https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/multiple\\_inheritance.py](https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/multiple_inheritance.py)

```
1  class AudioDevice:
2      def play(self, channel):
3          print('You are listening to channel n. {}'.format(channel))
4
5  class VideoDevice:
6      def play(self, channel):
7          print('You are looking to channel n. {}'.format(channel))
8
9  # Multiple inheritance!
10 class Television(AudioDevice, VideoDevice):
11     def show(self, channel):
12         AudioDevice.play(self, channel)
13         VideoDevice.play(self, channel)
14
15 tv = Television()
16 tv.show(5)
17 # Is this a good idea?
18 tv.play(5) # Which one do we get? Why?
19 # Hint
20 # print(Television.mro())
21
22 [Output]
23 You are listening to channel n. 5
24 You are looking to channel n. 5
25 You are listening to channel n. 5
```

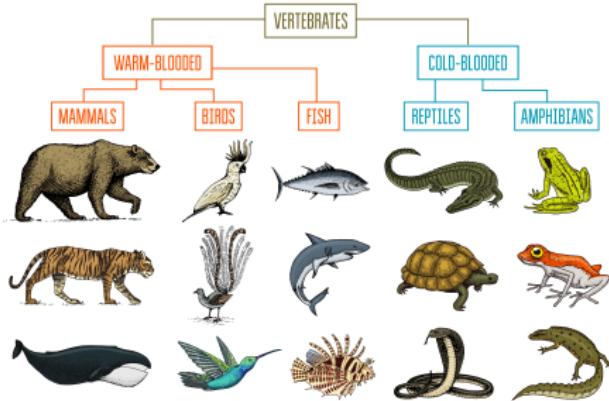


# Inerithance

## Infernus Linnaei



## CLASSIFICATION OF ANIMALS

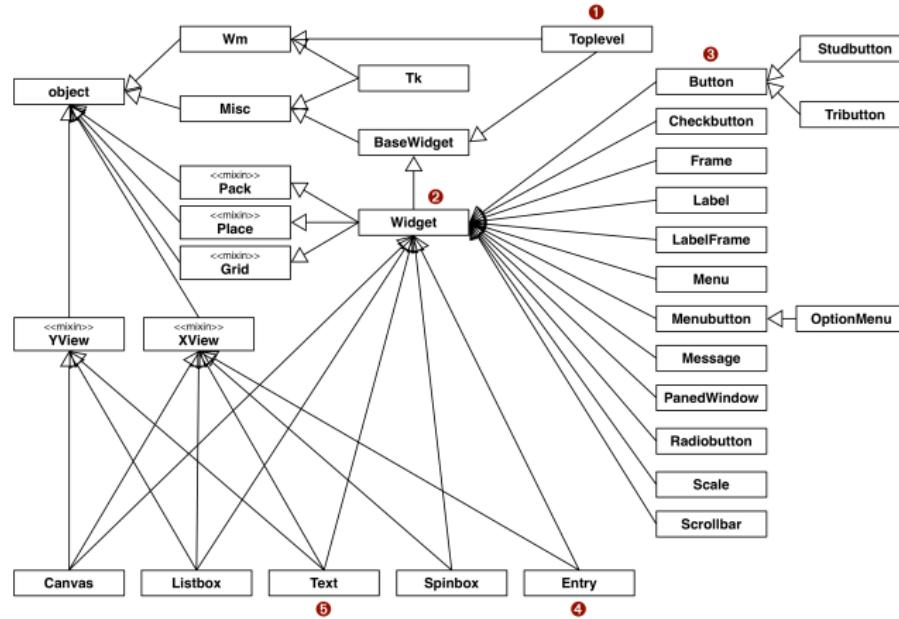


- ▷ Don't abuse inheritance!
- ▷ Though bringing order to the World may look appealing...



# Inheritance

A real case example



<https://wiki.python.org/moin/TkInter>

▷ ... the result may be more complicated than you expect!



## Composition

- ▷ **Composition** is a different technique for reusing functionalities
- ▷ The concept is simple: just use an object of some class as a member of a different one
- ▷ For example we can create the classes 'Enigne' and 'Wheel' and then the class 'Car' will have a member of type Engine and 4 members of type Wheel
- ▷ A class like 'Car' in the example is sometimes called an **aggregate** class



# Composition

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/composition.py>

```
1  class Engine:
2      """ Class describing a fuel engine
3      """
4
5      def start(self):
6          """ Start the engine """
7          print('Broom broom!')
8
9  class Car:
10     """Class describing a car.
11     """
12
13     def __init__(self):
14         self.engine = Engine()
15
16     def drive(self):
17         """ Start the car """
18         self.engine.start()
19
20 ferrari = Car()
21 ferrari.drive()
22
23 [Output]
Broom broom!
```



## Composition vs Inheritance

- ▷ Composition models a 'has-a' relation in the real world: a Car *has* a Engine
- ▷ Inheritance models a 'is-a' relation in the real world: an Electron *is* a Particle
- ▷ It may not always be obvious which one to use in your specific case: choose wisely!



## Pitfalls of Inheritance

- ▷ Inheritance is a wild beast. There are entire libraries written about how and when (not) to use it
- ▷ Question for you: should a Square inherit from a Rectangle?
- ▷ Seems legit: a Square *is* a specialized Rectangle
- ▷ But what happens if the Rectangle class has a changeHeight() method?
- ▷ **Liskov Substitution Principle:** you should always be able to use a derived class instead of a base class in your code
- ▷ In other words: a derived class should always extend or specialize the functionalities of the base class, never restrict them!



## Short summary (3)

- ▷ A class can inherit functionalities from one or more other classes (Inheritance)
- ▷ The class that inherits is called Derived (or Child) class the inherited one is the Base (or Parent) class
- ▷ Inheritance models an *is-a* relationship
- ▷ Classes can also incorporate other objects as class members (Composition)
- ▷ Composition models an *has-a* relationship
- ▷ Inheritance is tricky: use it with care!