OOP introduction (1/2)

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN–Pisa

Compiled on September 29, 2021

Smart variables

- Working with containers like lists or dictionaries, you may have noticed that they can do many thing 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 errror (*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`
- 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	1.46%	+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 defnitions:
  - A class is a blueprint for creating objects
  - An object is a concrete relization of a class
- You can imagine a class like a project, which is used to describe how objects are built and how they works
- 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 varibale 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 volumne
    - 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 netx_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

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

```
1  class Television:
2      """ Class describing a televsion.
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 'pippo' 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

```
1  class Television:
2      """ Class describing a televsion.
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

```
1  class Television:
2      """ Class describing a televsion.
3      """
4      def add_an_attribute(self):
5          """ Add a class attribute (remember the meaning of 'self') """
6          self.current_channel = 1
7
8  tv = Television()
9  # Add an attribute from inside a class method
10 tv.add_an_attribute()
11 print (tv.current_channel)
12
13 # Again, attributes are not shared
14 another_tv = Television()
15 another_tv.add_an_attribute()
16 # Changing the attribute for one will not affect other instances of the class
17 tv.current_channel = 5
18 # The following line will print 1, not 5
19 print(another_tv.current_channel)
20
21 [Output]
22 1
23 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[1]
- Class methods like *__init__*, with the name surronded by two underscores, are called special methods or dunder methods.
- Is is good practice to define all your class attributes inside the constructor!

[1] 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

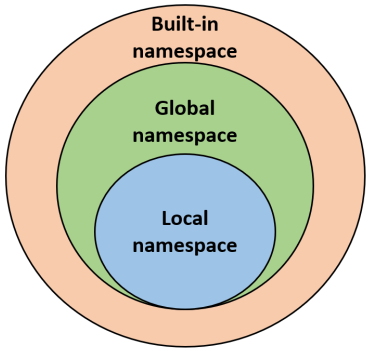
Constructor

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

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



Type of Namespaces

- 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 it 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

(And their strange behaviour)

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_attributes_3.py

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

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 instanciated 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

- 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

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

Advantages of encapsulation

- This is a good idea for several reasons:
  - The development of a part (e.g. `the television') is clearly separated from the rest of the project (e.g. `the dvd reader')
  - An object can be reused in different contexts: you can easily connect the same TV to the dvd reader, the game consol or the VHS, provided that the required interface (port + cable) is the same
  - You can even change your object, for example buy a new model, and, as long as the interfaces are the same, you will still be able to use it in the same way!
- These advantages may be extremely important for large-scale project!

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!

How do you get *real* encapsulation?

Digression - the boring languages

- In languages like C++ you can explicitly declare that some class attributes (and methods) are *private*
- Private attributes cannot be accessed outside the class - the compiler enforce that!
- There is this sort of common wisdom that dictates that *all* class data should be private
- If you really want to provide access to some of them, you can use the infamous 'getters' and 'setters' methods
- That's a lot of code to write - plus, it's not the pythonic way!

Pythonic encapsulation

- 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

```
1  class Television:
2      """ Class describing a televsion.
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

- 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 getters/setters
- But the awsome 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
- That is a *huge* advantage: it means you can start without them, and add the code only if really needed!
- Let's compare the two methods in the following example

Example of a class without encapsulation

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

```
1  class Television:
2      """ Class describing a televsion.
3      """
4      def __init__(self, owner):
5          """ Class constructor"""
6          # Here there is no encapsulation. The attribute can be read and modified
7          # freely
8          self.owner = owner
9
10 tv = Television('Batman')
11 print('This tv belongs to {}'.format(tv.owner)) # Read the attribute
12 tv.owner = 'Joker' # Change the attribute - this works
13 print('This tv belongs to {}'.format(tv.owner))
14
15 [Output]
16 This tv belongs to Batman
17 This tv belongs to Joker
```

Old-style encapsulation: never do that!

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

```
1  """ Now we want prevent the owner from being changed using old-style
2  encapsulation, with setters and getters"""
3
4  class Television:
5      """ Class describing a televsion.
6      """
7      def __init__(self, owner):
8          """ Class constructor"""
9          self._owner = owner
10
11     def get_owner(self):
12         """ Old-style getter."""
13         return self._owner
14
15     def set_owner(self, new_owner):
16         """ Old-style setter to control access - in this case preventing the
17         attribute from being changed."""
18         print('Nope {}. Do you want to steal my tv?'.format(new_owner))
19
20 tv = Television('Batman')
21 # In the following lines we had to change the code!
22 print('This tv belongs to {}'.format(tv.get_owner()))
23 tv.set_owner('Joker')
24 print('This tv belongs to {}'.format(tv.get_owner()))
25
26 [Output]
27 This tv belongs to Batman
28 Nope Joker. Do you want to steal my tv?
29 This tv belongs to Batman
```

Pythonic encapsulation with properties

/github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/class_tv_encapsulation_properties_ver

```
1  class Television:
2      """ Class describing a televsion.
3      """
4      def __init__(self, owner):
5          """ Class constructor"""
6          self._owner = owner
7
8      def get_owner(self):
9          """ Same as the getter"""
10         return self._owner
11
12     def set_owner(self, new_owner):
13         """ Same as the setter"""
14         print('Nope {}. Do you want to steal my tv?'.format(new_owner))
15
16     # This is where we do the property trick!
17     owner = property(fget=get_owner, fset=set_owner)
18
19 tv = Television('Batman')
20 # Notice the difference: this is the same code as before
21 print('This tv belongs to {}'.format(tv.owner))
22 tv.owner = 'Joker' # Change the attribute - this time it does not work
23 print('This tv belongs to {}'.format(tv.owner))
24
25 [Output]
26 This tv belongs to Batman
27 Nope Joker. Do you want to steal my tv?
28 This tv belongs to Batman
```

Even more pythonic encapsulation

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

```
""" This time with the special @ syntax, which is much more concise and elegant. """

class Television:
    """ Class describing a televsion.
    """
    def __init__(self, owner):
        """ Class constructor"""
        # Old-style encapsulation, using setters and getters. Never do that.
        self._owner = owner

    @property
    def owner(self):
        return self._owner

    @owner.setter
    def owner(self, new_owner):
        """ This function must be called 'owner'"""
        print('Nope {}. Do you want to steal my tv?'.format(new_owner))

tv = Television('Batman')
# Again: this is the same code as before
print('This tv belongs to {}'.format(tv.owner))
tv.owner = 'Joker'
print('This tv belongs to {}'.format(tv.owner))

[Output]
This tv belongs to Batman
Nope Joker. Do you want to steal my tv?
This tv belongs to Batman
```

- A physicist thinks:
  - "I have this super-cool algorithm to solve the problem I am working on: I will code it carefully, than 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?"

Interfaces

- 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 oustide 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 specialyzing 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 transtive: 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

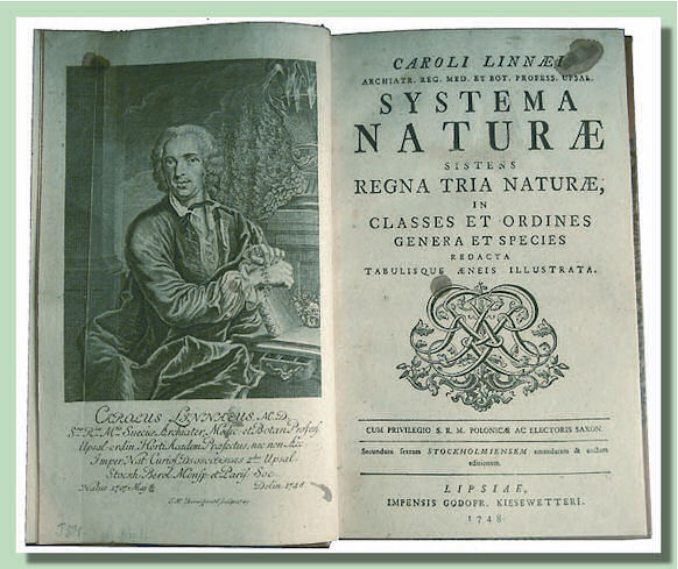
https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/overload.py

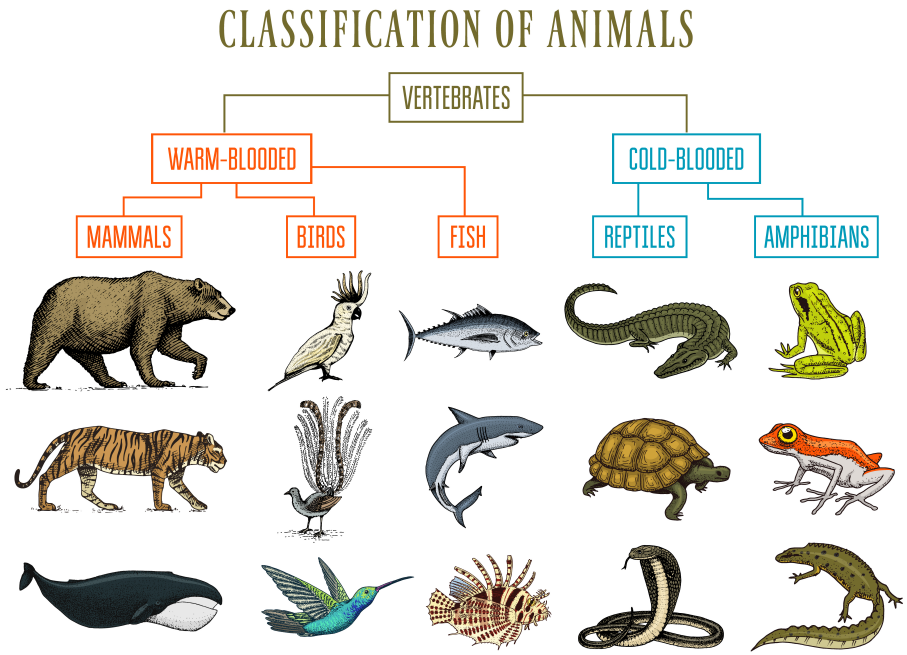
```
1  class Animal:
2      def sound(self):
3          return None
4
5  class Dog(Animal):
6      def sound(self):
7          """ This will shadow the method in the base class"""
8          return 'Woof!'
9
10 class Cat(Animal):
11     def sound(self):
12         """ This will shadow the method in the 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

```
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
```

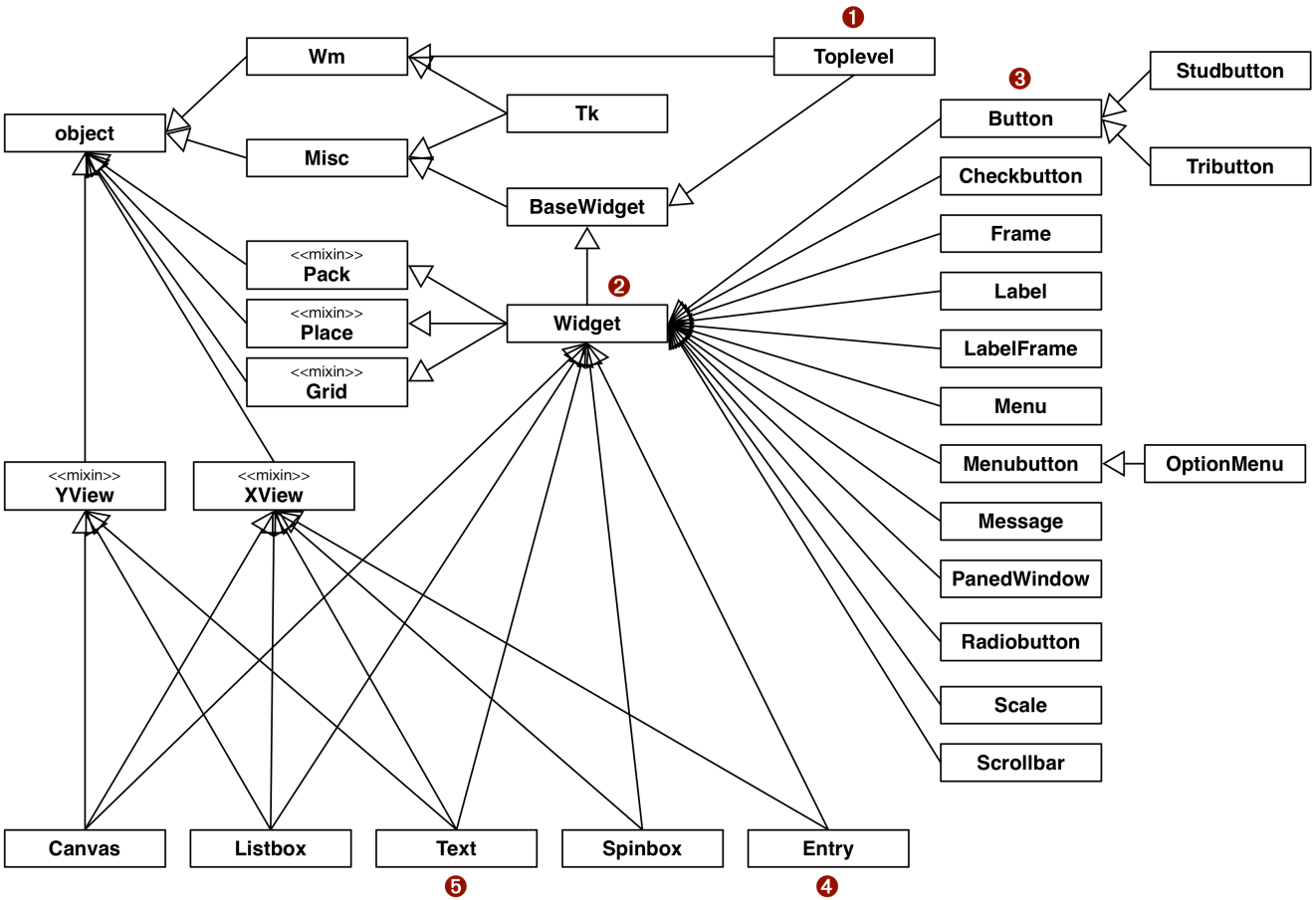




- Don't abuse inheritance!
- Though bringing order to the World may look appealing. . .

Inerithance

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

- Inheritance is a wild beast. There are entire libraries written about how and when (not) to use it

- Question for you: should a Square inherits 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 funcionalities from one or more other classes (Inheritance)
- The class that inherits is call 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!