SCRIPTING AND PROGRAMMING LABORATORY FOR DATA ANALYSIS

Lecture 10

Object-oriented programming (OOP)

"Objects are Python's abstraction for data.

All data in a Python program is represented by objects

or by relations between objects."

From the official python documentation

“Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of attributes, and code, in the form of functions known as methods. A distinguishing feature of objects is that an object's method can access and often modify the data attributes of the object with which they are associated (objects have a notion of "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.”

Kindler, Krivy

Introduction to object oriented programming (OOP)

Nearly everything in python is an ***object***: it has an identifier, a type and a value.

```
>>>  n = 5
```

n: identifier (unique); int: type; 5: value

How to create (customizable) objects? With **CLASSES.**

Objects are *instances of classes* [classes are objects themselves, btw (*metaclasses, metaprogramming*)].

AN EXAMPLE: BIKE CLASS

statement for defining a class

Variables

Methods (class “functions”)

Class instances

```python
# let's define the class Bike
class Bike:
    def __init__(self, colour, frame_material):
        self.colour = colour
        self.frame_material = frame_material

    def brake(self):
        print("Braking!")
```

```python
# let's create a couple of instances
red_bike = Bike('Red', 'Carbon fiber')
blue_bike = Bike('Blue', 'Steel')
# let's inspect the objects we have
print(red_bike.colour) # prints: Red
print(red_bike.frame_material) # prints: Carbon fiber
print(blue_bike.colour) # prints: Blue
print(blue_bike.frame_material) # prints: Steel
# let's brake!
red_bike.brake() # prints: Braking!
```

Note: **__init__** is a magic method (initializer)

Defining the simplest class

```
[3]: class Simplest(): # empty parentheses are optional
         pass
```

```
[4]: print(type(Simplest))
```

```
<class 'type'>
```

```
[5]: simp = Simplest()
     print(type(simp))
```

```
<class '__main__.Simplest'>
```

Classes/attributes

```
[6]: class Person:
         species = 'Human'
```

Class **attribute**

```
[7]: print(Person.species)
Human
```

dot is for accessing an attribute (or method!)

```
[8]: Person.alive = True
     print(Person.alive)
True
```

dynamical-ly added

```
[10]: man = Person()
      print(man.species)
Human
```

inherited

```
[11]: print(man.alive)
True
```

```
[12]: Person.alive = False
      print(man.alive)
False
```

Note:

```
[13]: man.name = 'Darth'
      man.surname = 'Vader'
      print(man.name, man.surname)
Darth Vader
```

```
[14]: print(Person.name, Person.surname)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-82e2dd53ba1c> in <module>
----> 1 print(Person.name, Person.surname)

AttributeError: type object 'Person' has no attribute 'name'
```

Only defined for the man instance, not for all the class Person!

ATTRIBUTES SHADOWING

```
[1]: class Point:
         x = 10
         y = 7
```

```
[2]: p = Point()
     print(p.x)
     print(p.y)
```

Inherited from class attributes

```
10
7
```

```
[3]: p.x = 12
     print(p.x)
```

Here p gets its own attribute [***shadowing***]

```
12
```

```
[4]: print(Point.x)
```

```
10
```

Class attribute remains unvaried

We delete x from namespace p: now python will search it in the class, since it could not be found in the instance

```
[5]: del p.x
     print(p.x)
```

```
10
```

Add attribute z to instance p of class Point

```
[6]: p.z = 3
     print(p.z)
```

```
3
```

```
[7]: print(Point.z)
```

```
-------------------
AttributeError
<ipython-input-7-5
```

Instances get whatever is in the class, but the opposite is not true.

The 'self' variable

The **self** special variable within a class is a way of referring to an instance within the class definition. It is the first variable to be defined (automatically!) within a class

self is a reference to an instance

```
[1]: class Square:
         side = 8
         def area(self):
             return self.side ** 2
```

equivalent

side found in the class

```
[3]: sq = Square()
     print(sq.area())
     print(Square.area(sq))

64
64
```

side found in the instance

```
[4]: sq.side = 10
     print(sq.area())
     print(Square.area(sq))

100
100
```

The 'self' variable

```
[1]: class Price:
         def final_price(self, vat, discount=0):
             """Returns price after applying vat and fixed discount."""
             return (self.net_price * (100 + vat) / 100) - discount
```

```
[2]: p1 = Price()
     p1.net_price = 100
     print(Price.final_price(p1, 20, 10))
```

```
110.0
```

```
[3]: print(p1.final_price(20, 10)) # equivalent
```

```
110.0
```

Nothing prevents us from using arguments when declaring methods. We can use the exact same syntax as we used with the function, **but the first argument will always be the instance.** *Always call it* **self** *!!!* (important convention)

Properly initialize an instance of a class

The **`__init__`** function is the *class initializer* in python (analogous to the constructor in other languages, but here it works on an already created instance). It is a *magic method* run right after the object is created.

```
[1]: class Rectangle:
         def __init__(self, side_a, side_b):
             self.side_a = side_a
             self.side_b = side_b
         def area(self):
             return self.side_a * self.side_b
```

```
[2]: r1 = Rectangle(10, 4)
     print(r1.side_a, r1.side_b)
     print(r1.area())
```

```
10 4
40
```

```
[3]: r2 = Rectangle(7, 3)
     print(r2.area())
```

```
21
```

__init__ is run automatically when the object is created

ALWAYS use __init__ when writing your own classes!

Important:

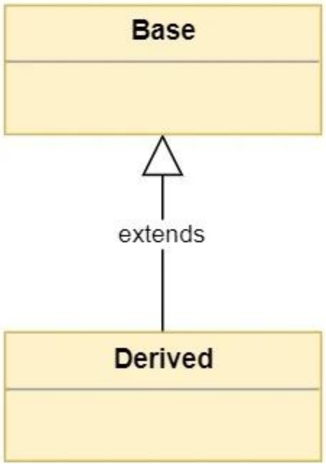
Object oriented programming is about code reuse!

Inheritance VS Composition

Relationships between classes

'is-a' relationship

When you have a **Derived** class that inherits from a **Base** class, you created a relationship where Derived is a specialized version of Base.




```
[1]: class Engine:
         def start(self):
             pass
         def stop(self):
             pass
```

```
[2]: class ElectricEngine(Engine):
         pass

     class V8Engine(Engine):
         pass
```

is-a

is-a

```
[3]: ee = ElectricEngine()
     ee.start()
```

```
[4]: v8e = V8Engine()
     v8e.start()
```

'has-a' relationship

When you have a **Composite** class, it contains one or more components from another class (named **Component**)

```
[1]: class Engine:
         def start(self):
             pass
         def stop(self):
             pass
```

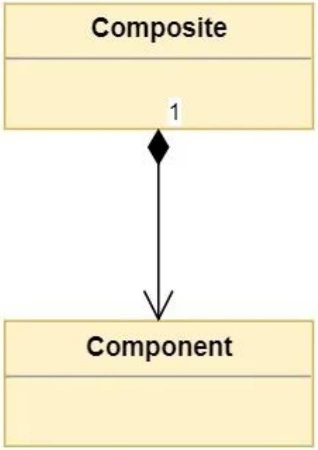
```
[2]: class Car:
         engine_cls = Engine
         def __init__(self):
             self.engine = self.engine_cls()
         def start(self):
             print('Starting engine... Wroom, wroom
             self.engine.start()

         def stop(self):
             self.engine.stop()
```

has-a

```
[3]: mycar = Car()
```

```
[4]: mycar.start()
     Starting engine... Wroom, wroom!
```

```
[5]: mycar.stop()
```



INHERITANCE

When we define the derived class with the () after the class name, all methods and attributes of the Base class are *inherited* by the derived class

```
class A(B):

    pass
```

→ A is the child (or derived) of B, and B is the parent (or base) of A.

→ B inherits or extends class A

[1]:
```
class Engine:
    def start(self):
        pass
    def stop(self):
        pass
```

[2]:
```
class ElectricEngine(Engine):
    pass

class V8Engine(Engine):
    pass
```

Parent (or Base) class

[3]:
```
ee = ElectricEngine()
ee.start()
```

[4]:
```
v8e = V8Engine()
v8e.start()
```

ee and v8e have start() and stop() methods inherited from Engine

Note: ee is an instances of BOTH ElectricEngine AND Engine

Composition

A class can contain instances from other classes/a composite class has component(s) of other class(es)

Note: polymorphism!

```
[1]: class Engine:
         def start(self):
             pass
         def stop(self):
             pass
```

```
[2]: class Car:
         engine_cls = Engine
         def __init__(self):
             self.engine = self.engine_cls()
         def start(self):
             print('Starting engine... Wroom, wroom
             self.engine.start()

         def stop(self):
             self.engine.stop()
```

```
[3]: mycar = Car()
```

```
[4]: mycar.start()
```

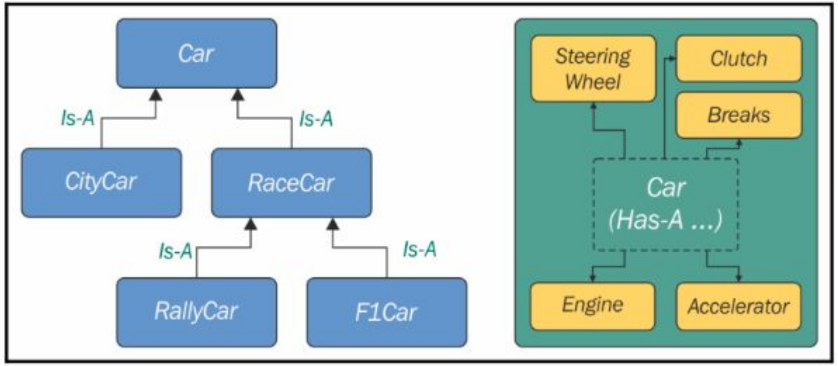
```
Starting engine... Wroom, wroom!
```

```
[5]: mycar.stop()
```

```python
class Engine:
    def start(self):
        pass
    def stop(self):
        pass

class ElectricEngine(Engine): # Is-A Engine
    pass

class V8Engine(Engine): # Is-A Engine
    pass

class Car:
    engine_cls = Engine
    def __init__(self):
        self.engine = self.engine_cls() # Has-A Engine
    def start(self):
        print('Starting engine... Wroom, wroom!')
        self.engine.start()
    def stop(self):
        self.engine.stop()

class RaceCar(Car): # Is-A Car
    engine_cls = V8Engine
class CityCar(Car): # Is-A Car
    engine_cls = ElectricEngine
class F1Car(RaceCar): # Is-A RaceCar & Is-A Car
    pass # engine_cls same as parent
```

Example

```python
car = Car()
racecar = RaceCar()
citycar = CityCar()
f1car = F1Car()
```



Accessing a base class

Note: if a class has not a base class, python will set the special object class as the base class for the one we're defining. Ultimately, all classes derive from an object.

```
class A:

    pass
```

or

```
class A():

    pass
```

or

```
class A(object):

    pass
```

Are exactly the same thing. The ***object*** class is a special class in that it has the methods that are common to all Python classes

ACCESSING A BASE CLASS: 3 EXAMPLES

```
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        self.title = title
        self.publisher = publisher
        self.pages = pages
        self.format_ = format_
```

Not ideal, redundant, prone to errors, a lot of maintenance

```
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        Book.__init__(self, title, publisher, pages)
        self.format_ = format_
```

Better, can be improved

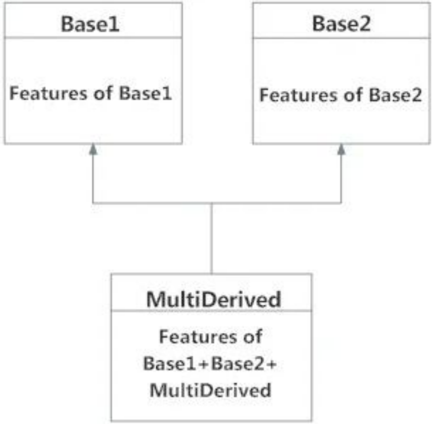
```
class Book:
    def __init__(self, title, publisher, pages):
        self.title = title
        self.publisher = publisher
        self.pages = pages

class Ebook(Book):
    def __init__(self, title, publisher, pages, format_):
        super().__init__(title, publisher, pages)
        # Another way to do the same thing is:
        # super(Ebook, self).__init__(title, publisher, pages)
        self.format_ = format_
```

Best practice, even if one modifies the name of the parent class everything still works! super() keyword is useful!

***super** is a function that returns a proxy object that delegates method calls to a parent (or sibling) class.*

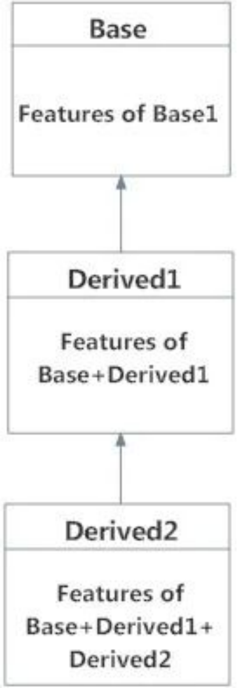
MULTIPLE AND MULTILEVEL INHERITANCE

In python you can define a class that has a base more than one base class (*multiple inheritance*).




```
[1]: class Base1:
         pass

     class Base2:
         pass

     class MultiDerived(Base1, Base2):
         pass
```

You can in principle have a chain of class parent child (*multilevel inheritance*) but it is probably not a very good idea to have a long set of parent-childs.

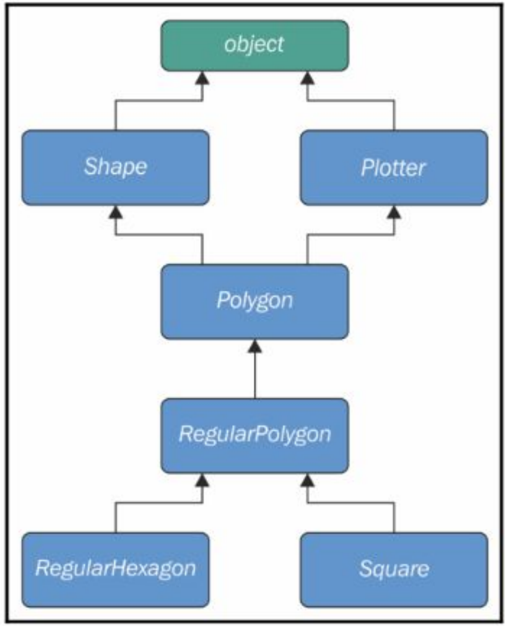



```
[2]: class Base:
         pass

     class Derived1(Base):
         pass

     class Derived2(Derived1):
         pass
```

Example: MULTIPLE INHERITANCE

```
[1]: class Shape:
         geometric_type = 'Generic Shape'
         def area(self):
             raise NotImplementedError
         def get_geometric_type(self):
             return self.geometric_type

     class Plotter:
         def plot(self, ratio, topleft):
             print('Plotting at {}, ratio {}.'.format(topleft, ratio))

     class Polygon(Shape, Plotter):
         geometric_type = 'Polygon'

     class RegularPolygon(Polygon):
         geometric_type = 'Regular Polygon'
         def __init__(self, side):
             self.side = side

     class RegularHexagon(RegularPolygon):
         geometric_type = 'RegularHexagon'
         def area(self):
             return 1.5 * (3 ** .5 * self.side ** 2)

     class Square(RegularPolygon):
         geometric_type = 'Square'
         def area(self):
             return self.side * self.side
```

```
[2]: hexagon = RegularHexagon(10)
     print(hexagon.area())

     259.8076211353316

[3]: print(hexagon.get_geometric_type())

     RegularHexagon

[4]: hexagon.plot(0.8, (75, 77))

     Plotting at (75, 77), ratio 0.8.

[5]: square = Square(12)
     print(square.area())

     144

[6]: print(square.get_geometric_type())

     Square

[7]: square.plot(0.93, (74, 75))

     Plotting at (74, 75), ratio 0.93.
```



Method resolution order (MRO) in python

When you ask for *someobject.attribute* and *attribute* is not found on that object, Python starts searching in the class that *someobject* was created from. If it's not there either, Python searches up the inheritance chain until either attribute is found or the object [top] class is reached.

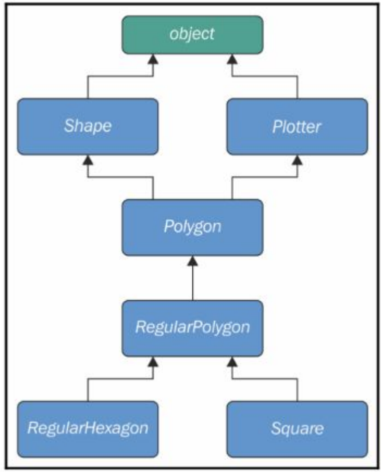
But how is the inheritance working if multiple inheritance is involved?

The left-most class wins!

```
# in the preceding example:
# you can look for the MRO with:
print(square.__class__.__mro__)
```

```
(<class '__main__.Square'>, <class '__main__.RegularPolygon'>, <class '__main__.Polygon'>, <class '__main__.Shape'>, <class '__main__.Plotter'>, <class 'object'>)
```

```
# ordered as for MRO
```



Another example:

```
class A:
    label = 'a'
class B(A):
    label = 'b'
class C(A):
    label = 'c'
class D(B, C):
    pass

d = D()
print(d.label)
```

```
b
```

```
class A:
    label = 'a'
class B(A):
    pass # was: label = 'b'
class C(A):
    label = 'c'
class D(B, C):
    pass

d = D()
print(d.label)
```

```
c
```

```
print(d.__class__.mro())
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

A very brief mention to @decorators

In python, decorators are special features that add functionalities to an existing code. In general, decorators precede a function declaration. What they practically do is to take a function and add some particular functionality to it, and then return it.

Decorators can be recognized as they have a leading @

STATIC METHODS

Sometimes it makes sense to group functionalities under a class. Static methods are perfect for this as they are not passed any special argument (e.g. no `self`) as leading argument, and they are bound to the class, not to an instance of the class.

So they are useful when we need some functionality that is not referred to any instance. This is advantageous when we need to create Utility methods as they aren't tied to an object lifecycle.

Static methods can be created by simply putting the decorator @staticmethod before the method definition

Without static method:

```
[1]: class StringUtil:
         def get_unique_words(sentence):
             return set(sentence.split())

[2]: StringUtil.get_unique_words("Hello world")

[2]: {'Hello', 'world'}

[3]: s = StringUtil()
     s.get_unique_words("Hello world")

---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-3-36862a3c98e8> in <module>
      1 s = StringUtil()
----> 2 s.get_unique_words("Hello world")

TypeError: get_unique_words() takes 1 positional argument but 2 were given
```

`self` is implicitly the first argument of **get_unique_words**

With static method:

```
[1]: class StringUtil:
         @staticmethod
         def get_unique_words(sentence):
             return set(sentence.split())

[2]: s = StringUtil()
     s.get_unique_words("Hello world")

[2]: {'Hello', 'world'}
```

Here now the class can act as a container for functions!

CLASS METHODS

Analogous to static methods in many ways, with the important difference that they take as an argument the class object itself. So static methods don't know anything about the class, it only knows its parameters; class methods know everything about the class. Class methods are useful for creating factory methods [methods to create classes].

Class methods can be created by simply putting the decorator @classmethod before the method definition

Without class method:

```
[1]: class Person:
         age = 25
         def printAge(cls):
             print('The age is:', cls.age)
```

```
[2]: Person.printAge()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-b1f55ee52dbd> in <module>
----> 1 Person.printAge()

TypeError: printAge() missing 1 required positional argument: 'cls'
```

Error as it **was expecting the class itself** as argument

With class method:

```
[3]: class Person:
         age = 25
         @classmethod
         def printAge(cls):
             print('The age is:', cls.age)
```

```
[4]: Person.printAge()
The age is: 25
```

The parameter is always the class itself, so that all attributes are accessible!

Private or not private?

In python everything in a class is public and can be accessed from the outside! [In other programming languages, some variables/methods in a class can be made private, and can only be accessed within the class itself]

To deal with this, in python one has to rely on conventions and on *name mangling*.

CONVENTION: if an attribute's name has **no leading underscores**, it is considered **public.** This means you can access it and modify it freely. When the name has [at least] **one leading underscore**, the attribute is considered **private**, i.e. meant to be used internally. When the name has two leading underscores, magic happens (see next).

Examples of ideally private attributes: *helper methods that are supposed to be used by public ones (possibly in call chains in conjunction with other methods), and internal data, such as scaling factors, or any other data that ideally we would put in a constant (a variable that cannot change – there is no constant in python though!)*

Example

```
[1]: class A:
         def __init__(self, factor):
             self._factor = factor
         def op1(self):
             print('Op1 with factor {}...'.format(self._factor))

     class B(A):
         def op2(self, factor):
             self._factor = factor
             print('Op2 with factor {}...'.format(self._factor))

[2]: obj = B(100)

[3]: obj.op1()

     Op1 with factor 100...

[4]: obj.op2(42)

     Op2 with factor 42...

[5]: obj.op1()

     Op1 with factor 42...
```

_factor should not be modified!

_factor is modified as it is called by the cold class B in op2: **BAD!!**

Solution for this in the next →

NAME MANGLING

Any attribute name that has **at least two leading underscores and at most one trailing underscore**, such as __my_attr, **is replaced with a name that includes an underscore and the class name before the actual name**, such as _ClassName__my_attr.

Bottom line: if you want to have a "private" variable, start its identifier with two leading underscores (__)

```
[1]: class A:
         def __init__(self, factor):
             self.__factor = factor
         def op1(self):
             print('Op1 with factor {}...'.format(self.__factor))

     class B(A):
         def op2(self, factor):
             self.__factor = factor
             print('Op2 with factor {}...'.format(self.__factor))

[2]: obj = B(100)

[3]: obj.op1()

     Op1 with factor 100...

[4]: obj.op2(42)

     Op2 with factor 42...

[5]: obj.op1()

     Op1 with factor 100...
```

SOLVED! __factor now has two leading underscores, no confusion!

NAME MANGLING EXAMPLE #2

Any attribute name that has **at least two leading underscores and at most one trailing underscore**, such as __my_attr, **is replaced with a name that includes an underscore and the class name before the actual name**, such as _ClassName__my_attr.

Bottom line: if you want to have a "private" variable, start its identifier with two leading underscores (__)

```
[1]: class Sample:
         def __init__(self, nv, pv):
             # normal variable
             self.nv = nv
             # private variable(not really)
             self.__pv = pv
```

```
[2]: sample = Sample('Normal variable', 'Private variable')
     # accessing *nv*
     print(sample.nv)
```

```
Normal variable
```

```
[3]: # accessing *__pv**
     print(sample.__pv)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most re
<ipython-input-3-dc8c42f5bc72> in <module>
      1 # accessing *__pv**
----> 2 print(sample.__pv)

AttributeError: 'Sample' object has no attribute '__pv'
```

```
[4]: # how to access it
     #(not intuitive, and just AVOID IT)
     print(sample._Sample__pv)
```

```
Private variable
```

Operators overloading

Overload an operator: give it a meaning according to the context in which it is used

Example: the + operator means addition when we deal with numbers, but concatenation when we deal with sequences and strings.

In Python, when you use operators, you're most likely calling the **special methods** of some objects behind the scenes. For example, the a[k] call roughly translates to type(a).__getitem__(a, k).

It is possible to override this behaviour in a class

special methods examples

+	→ object.__add__(self, other)
-	→ object.__sub__(self, other)
*	→ object.__mul__(self, other)
/	→ object.__div__(self, other)
//	→ object.__floordiv__(self, other)
%	→ object.__mod__(self, other)
**	→ object.__pow__(self, other)
+=	→ object.__iadd__(self, other)
-=	→ object.__isub__(self, other)
*=	→ object.__imul__(self, other)
/=	→ object.__idiv__(self, other)
//=	→ object.__ifloordiv__(self, other)
%=	→ object.__imod__(self, other)
**=	→ object.__ipow__(self, other)
<	→ object.__lt__(self, other)
<=	→ object.__le__(self, other)
>	→ object.__gt__(self, other)
>=	→ object.__ge__(self, other)
==	→ object.__eq__(self, other)
!=	→ object.__ne__(self, other)

special methods list here

Operators overloading: an example

```
[1]: class Weird:
         def __init__(self, s):
             self._s = s
         def __len__(self):
             return len(self._s)
         def __bool__(self):
             return '42' in self._s

[2]: weird = Weird('Hello! I am 9 years old!')
     print(len(weird))

     24

[3]: print(bool(weird))

     False

[4]: weird2 = Weird('Hello! I am 42 years old!')
     print(len(weird2))

     25

[5]: print(bool(weird2))

     True
```

@PROPERTY DECORATOR

Python programming features a built-in @property decorator which makes usage of getter and setters much easier in Object-Oriented Programming.

*!! A **getter** is a method that is called when we access an attribute for reading. A **setter** is a method that is called when we access an attribute to write it.*

Example:

Basic implementation

```
[1]: class Celsius:
         def __init__(self, temperature = 0):
             self.temperature = temperature

         def to_fahrenheit(self):
             return (self.temperature * 1.8) + 32
```

```
[2]: human = Celsius()
     human.temperature = 37
     print(human.temperature)
```

```
37
```

```
[3]: print(human.to_fahrenheit())
```

```
98.60000000000001
```

Getter/setter

```
[1]: class Celsius:
         def __init__(self, temperature=0):
             self.set_temperature(temperature)

         def to_fahrenheit(self):
             return (self.get_temperature() * 1.8) + 32
         # getter method
         def get_temperature(self):
             return self._temperature
         # setter method
         def set_temperature(self, value):
             if value < -273.15:
                 raise ValueError(
                     "T<-273.15! Not possible.")
             self._temperature = value
```

```
[2]: human = Celsius()
     human.set_temperature(37)
```

```
[3]: human.set_temperature(-500)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback
<ipython-input-3-20ef91b6a1ea> in <module>
----> 1 human.set_temperature(-500)
```

@Property :)

```
[1]: class Celsius:
         def __init__(self, temperature=0):
             self.temperature = temperature

         def to_fahrenheit(self):
             return (self.temperature * 1.8) + 32

         @property
         def temperature(self):
             print("Getting value...")
             return self._temperature

         @temperature.setter
         def temperature(self, value):
             print("Setting value...")
             if value < -273.15:
                 raise ValueError("T<-273.15! Not possible.")
             self._temperature = value
```

```
[2]: human = Celsius(37)
```

```
Setting value...
```

```
[3]: print(human.temperature)
```

```
Getting value...
37
```

```
[4]: print(human.to_fahrenheit())
```

```
Getting value...
98.60000000000001
```

```
[5]: coldest_thing = Celsius(-300)
```

```
Setting value...
---------------------------------------------------------------------------
ValueError                                Traceback (mos
```

@property decorator: another example

```
class Person:
    def __init__(self, age):
        self.age = age # anyone can modify this freely

class PersonWithAccessors:
    def __init__(self, age):
        self._age = age
    def get_age(self):
        return self._age
    def set_age(self, age):
        if 18 <= age <= 99:
            self._age = age
        else:
            raise ValueError('Age must be within [18, 99]')

class PersonPythonic:
    def __init__(self, age):
        self._age = age
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, age):
        if 18 <= age <= 99:
            self._age = age
        else:
            raise ValueError('Age must be within [18, 99]')
```