Lecture+6+Lambdas+and+Classes

October 25, 2019

1 Introduction to programming

1.0.1 Lecture 6: lambda and classes

Lecturers: Xavier Parent (xavier.parent@uni.lu) and Giovanni Casini (giovanni.casini@uni.lu)
Revamped version of material from Clément Guérin

2 Lambda

2.1 A very short overview on Lambda-calculus

So-called lambda-calculus (denoted λ-calculus) is a logical system for manipulating and reasoning about functions and the way in which they combine. It was invented in 1930 by Church as part of a theory intended as a foundation for mathematics.

The λ-calculus has been and is still widely used in Computer Science.

One of the most original achievement of λ-calculus is the **mechanisation of a (mathematical) proof** in a computer. See **Isabelle/HOL** or the **COQ proof assistant**

A vast collection of Isabelle examples and applications is available on the web site "Archive of formal proofs" (link below).

Ourselves at CSC we have been collaborating with Benzmueller, whose theorem-prover Leo 3 won the last CASC-27 world championship in LTB division (and was runner up in THF division).

References

- Short English introduction to λ-calculus: http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf
- Isabelle/HOL prof assistant: https://isabelle.in.tum.de/
- Archive of formal proof: https://www.isa-afp.org/
- home page of C. Benzmueller: http://page.mi.fu-berlin.de/cbenzmueller/
- COQ: https://coq.inria.fr

Formally, a typical λ-term in λ-calculus: $\lambda x.t$ where x is a variable and t is a term that might contain the variable x. $\lambda x.t$** represents the function sending x to $t(x)$.

One advantage of the lambda notation is that it allows us to easily talk about higher-order functions, i.e., functions whose inputs and/or outputs are themselves functions.

t o t is written as $\lambda x.t(t(x))$.

The function sending t to t o t is written as $\lambda t \lambda x.t(t(x))$.

2.2 "Lambda-calculus" in Python

In Python, **there is nothing which could be called λ-calculus**, but there is a command **lambda** that you can use to define a function. The formalism goes like this :

lambda x : *expression containing* x

This is the function sending x to the *expression containing* x. Therefore writing the following

$somefunc =$ **lambda** x : *expression containing* x

is equivalent to

def $somefunc(x)$:

return *expression containing* x

No restrictions on the number of arguments * lambda can receive zero, one, two or more arguments * another function can be passed as an argument too. * Default value arguments are possible too. However, you can override the default values by passing new values as the arguments.

There is no formal difference between the functions defined using the **lambda** command and those using the **def** command. The difference is in the ergonomy: * A lambda expression is more compact, and so the code is clearer * A lambda expression does not need a name. For anonymous functions, lambda is preferred. * With a lambda expression, the code is quicker to run for the interpretor

```
In [1]: squaring=lambda x: x*x #Computing the square of a number
        squaring(9)

Out[1]: 81

In [2]: def squaring2(x): #Computing the square of a number
            return x*x
        print(squaring2(100)) # Same result as for squaring(100)
        print(type(squaring)) # Type : function
        print(type(squaring2)) # same type

10000
<class 'function'>
<class 'function'>


In [3]: # Explain the different lines of codes.

        #I want to know the value of 2+3
        print(2+3)
        #OK

        #Composition of two functions
        print((lambda x : lambda y: x+y)(2)(3))
        #
        #(lambda x : lambda y: x+y)(2) is a function sending y to 2+y

        #A single function with two variables
        print((lambda x,y : x+y)(2,3))
        #(lambda x,y : x+y) is one function with two arguments
```

```
5
5
5
```

In [4]:
```
#Examples of lambda terms

Id=(lambda x:x) # identity function
Id(2)

fst=(lambda x,y: x) # selection function 1
fst(3,4)

snd=(lambda x,y : y) # selection function 2
snd(3,4)

cst=(lambda x : True) # constant function returning boolean value true
cst(2)

cst=(lambda x : 5) # constant function returnung 5
cst(2)
```

Out[4]:
```
5
```

In [ ]:
```
#composition of functions

f=lambda x : x * 2
g=lambda x : x+1

fog=lambda x : f(g(x))
fog(10)

# 3 arguments

multi = lambda x, y, z : x * y * z   ## lambda
print(multi(5, 2, 6))
```

In [ ]:
```
# Defaults argument values

add = lambda x = 10, y = 20, z = 30 : x + y + z
print(add()) # 10 + 20 + 30

multi = lambda x = 10, y = 20, z = 30 : x * y * z
print(multi()) # 10 * 20 * 30

sub = lambda x = 10, y = 45: y - x
print(sub()) # 45 - 10
```

In [ ]:
```
#However, you can override the default values by passing new values as the arguments.
```

```
add = lambda x = 10, y = 20, z = 30 : x + y + z
print(add(12, 14, 16)) # 12 + 14 + 16
print(add(75, 126)) # 75 + 126 + 30
print(add(222)) # 222 + 20 + 30
print(add()) # 10 + 20 + 30

print("Multiplication Values")
multi = lambda x = 10, y = 20, z = 30 : x * y * z
print(multi(2, 4, 5)) # x = 2, y = 4, z = 5
print(multi(100, 22)) # x = 100, y = 22, z = 30
print(multi(9)) # x = 9, y = 20, z = 30
print(multi()) # 10 * 20 * 30
```

In [ ]:
```
# We define a map "timing" sending n to the map "x goes to n*x"

# With an auxiliary function
def timing(n):
    def auxiliaryfunction(x): # Its name will never be heard of again
        return n*x
    return auxiliaryfunction

# With a lambda function
def timing2(n):
    return lambda x : n*x # Simpler and more explicit

#nested lambda functions
timing3=lambda n :lambda x: n*x


# Each of  timing(n), timing2(n) and timing3(n)  is the function sending x goes to n*x
```

In [ ]:
```
print(timing(5)('Example '))
print(timing2(5)('Example '))
print(timing3(5)('Example ')) # They all do the same thing!
```

2.3 List/Iterator manipulation with lambda command

In Python, you have a few built-in functions that can be used to construct objects out of a function and a container (or an iterator). We are going to see each of them, **filter**, **reduce**, **map**. The function you put as an argument of these functions is typically a **lambda**-type function.

Here a summary of what the three commands do:

- filter: it filters out elements in list
- reduce: it applies a rolling computation to sequential values in a list
- map: it applies the same operation to all the elements in a list

2.3.1 filter command

We have already seen an example of this command. This is almost self explanatory. The call **filter**$(func, L)$ will return the **iterator** generating the list of elements $L[i]$ of L such that $func(L[i])$

is **True**.

Whatever might be the type of the container in the argument, you always end up with a generator (not a list).

2.3.2 reduce command

The **reduce** command is contained in the **functools** package. In Python 2 it was a built-in function but to avoid conflict with other functions (apparently we "reduce" a lot of things in Python), the choice has been made to make it a built-in function in a specific module.

```
In [ ]: from functools import reduce
```

The **reduce** command takes as an argument a two arguments function $func$ and a list/iterator and returns $func(func(\dots, func(func(L[0], L[1]), L[2]), \dots, L[len(L)-2], L[len(L)-1])$.

It is a really useful function for performing some computation on a list and returning the result. It applies a rolling computation to sequential pairs of values in a list. For example, if you need to compute the product of a list of integers.

Remark: if $func$ is something like $x, y \mapsto x + y$ and the elements of your list L are integers, you get something like :

$$\textbf{reduce}(func, L) = \sum_{i=0}^{\textbf{len}(L)-1} L[i].$$

There is no reason for the function $func$ to be associative (i.e. $func(func(x,y),z) = func(x, func(y,z))$) nor commutative (i.e. $func(x,y) = func(y,x)$). In this case, the first formula I gave with the dots explain exactly in which order is evaluated **reduce**. Equivalently :

$$\textbf{reduce}(func, L) = func(\textbf{reduce}(func, L[0 : \textbf{len}(L)-1], L[\textbf{len}(L)-1])$$

2.3.3 map command

The **map** command may is the most useful among these three commands. You use it when you want to apply the same function to all the elements of a list. **map** takes as argument a regular function $func$ or a lambda expression and a list/iterator L and returns the generator of the list $[func(L[i]) \mid i \in \{0, \dots, \textbf{len}(L)-1\}]$.

```
In [ ]: #An example dealt with functions or lambdas.

        #Make the list of integers from 1 to 100 which are divisible by 13.

        #1 Using definition of a function
        def Isdivisibleby13(x):
            return x%13==0
        aux=filter(Isdivisibleby13,range(1,100))

        #print(aux)      # the result of a filter is an iterator and not a list
        print(list(aux))# if you want to see the elements, you can just turn it into a list

        #2 You could also use lambda operator
```

```
        aux2=filter(lambda x: x%13==0, range(1,100)) # What this line means is clearer than the

        #print(aux==aux2)  # The iterator are not the same
        #print(list(aux)==list(aux2)) # The lists are the same Xav: why false outputed?
        print(list(aux2))
```

```
In [ ]: # Could you explain the result of this call?
        for x in filter(lambda x: x%13, range(1,50)):
               print(x,end=" ")
        lambda x: x%13
        # if x%13==0 then it is evaluated as False
        # if x%13!=0 then it is evaluated as True
```

```
In [ ]: from functools import reduce
        print(reduce((lambda x ,y: x-y), [1, 2, 3, 4]))
        print(reduce((lambda x ,y: y-x), [1, 2, 3, 4]))
```

```
In [ ]: from functools import reduce
        print(reduce((lambda x ,y,z: x+y-z), [1, 2, 3, 4]))
```

```
In [ ]: from functools import reduce
        print(reduce((lambda x, y: x * y),[1, 2, 3, 4]) )
```

```
In [ ]: This is the same as using a for loop over the list.
```

```
In [ ]: product = 1
        list = [1, 2, 3, 4]
        for num in list:
            product = product * num
        print(product)
```

```
In [ ]: # Why do we always get 0 here?
        from functools import reduce
        N=int(input('Enter a positive integer here : '))
        #print(reduce((lambda x,y:x+y),range(0,1)))
        print(reduce(lambda x,y:x+y,range(0,N+1)-N*(N+1)/2)
```

```
In [ ]: ## Map with a usual function
        numbers=[1,3,5,9]
        def square(x):
            return x**2

        print(list(map(square,numbers)))
```

```
In [ ]: ## Map with lambda expression

        numbers=[1,3,5,9]

        print(list(map(lambda x: x**2,numbers)))
```

```
In [ ]: # Example, we compute the different values of (x*x)-10 for x=1,...,100
        for x in map(lambda x : x*x-10,range(1,101)):
               print(x, end=" ")

In [ ]:
```

3 Classes

This second part is meant to complement the material on classes presented last time. These introduce a bit of a new (complicated) syntax.

3.1 Basic ingredients

- Variables (attributes)
- Methods (including magic ones)
- Overloaded operators

3.2 Inheritance and diamond problem

3.3 Decorators

3.3.1 Variable (attribute)

In the context of classes, a variable is more or less the same as an attribute. A class can include both **class** variables and **instance** variables. The class variables are defined as part of the class itself, while instance variables are defined as part of methods.

Class variables are initialized immediately after the class name. class variables are shared by all instances of the class.

3.3.2 Methods

Methods are simply another kind of function that reside in classes. You create and work with methods in precisely the same way that you do functions, except that methods are always associated with a class (you don't see freestanding methods as you do with functions). You can create two kinds of methods:

- class methods: associated with the class itself (no self argument)
- instance methods: associated with the instances you created (self required as a first argument)

(There is a third kind of method, called static method)

3.3.3 Methods with variable number of arguments

Sometimes you create methods that can take a variable number of arguments. "Variable" in the sense that the exact number becomes known only as they are passed. Handling this sort of situation is something Python does well. Here are the two kinds of variable arguments that you can create: + *args: the method takes a variable number of positional arguments. + **kwargs: the method takes a variable number of keyword arguments.

3.3.4 Init method and self

- **init** is basically a function which will "initialize"/"activate" the properties of the class for a specific object, once created and matched to the corresponding class
- self represents that object which will inherit those properties

The **Init** method may take a number of parameters other than the self. This is what makes the different instances of the class unique, and how you distinguish them.

3.3.5 Magic methods

Each class also comes with a set of "magic" methods. A magic method is one that begins and ends with a double underscore.

They are not meant to be invoked directly by the user, but the invocation happens internally from the class on a certain action. For example, when you add two numbers using the + operator, internally, the **add**() method is called. This is how the + can be applied to different types.

To list all the magic methods a class has you type "dir(name-of-the-class)".

3.3.6 Operator overloading

This refers to the action of modifying the default meaning of an operator to make it. There are four main types of operators: arithmetical operators ; comparison operators ; assignment operator ; boolean operators.

All these operators are implemented using a magic method. So in order to make the overloaded behaviour available in your own custom class, the corresponding magic method should be overridden.

In the example below we redefine the add magic method so it applies to points in a 2-D coordinate system.

```
In [2]: # class variables

        class Shark:
              animal_type = "fish"

        # instance variables

        class MyClass:
            def DoAdd(self, Value1=0, Value2=0): # Value1 and Value2 are instance variables
                Sum = Value1 + Value2   # Sum is an instance variable too
                print("The sum of {0} plus {1} is {2}.".format(Value1, Value2, Sum))

        x=MyClass()
        x.DoAdd(3,2)
```

```
The sum of 3 plus 2 is 5.
```

```
In [10]: # class variables can be changed
```

```
        class Shark:
            animal_type = "fish" # Class variable initialized


        Shark.animal_type  # accessing the class variable
        Shark.animal_type = "bird" # Class variable re-initialized
        Shark.animal_type
```

```
Out[10]: 'bird'
```

```
In [5]: # Two kinds of methods

        class MyClass:

            def SayHello():  # class method
                print("Hello from the class")

            def SayHello2(self):   # instance method
                print("Hello from the instance!")

        MyClass.SayHello()          # Calling the class method
        MyInstance=MyClass()        # Creating an instance
        MyInstance.SayHello2()      # Calling the instance method
        #MyClass.SayHello2()
```

```
Hello from the class
Hello from the instance!
```

```
In [7]: # variable argument

        class MyClass:
            def PrintList1(*args):
                 for Count, Item in enumerate(args):
                     print("{0}. {1}".format(Count, Item))
            def PrintList2(**kwargs):
                 for Key, Value in kwargs.items(): # items() returns a list of dict's (key, va
                     print("{0} likes {1}".format(Key, Value))
        MyClass.PrintList1("Red", "Blue", "Green")
        MyClass.PrintList2(George="Red", Sue="Blue",Zarah="Green")
```

```
0. Red
1. Blue
2. Green
George likes Red
Sue likes Blue
Zarah likes Green
```

In [64]:
```
# init and self

class Box:

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height


# Create an instance of Box.
x = Box(10, 2)

# Print area.
print(x.area())
```

```
20
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-64-b02b3b559224> in <module>
 17 print(x.area())
 18
---> 19 Box.__getattribute__(x)


TypeError: expected 1 arguments, got 0
```

In [ ]:
```
### magic method

x,y=2,3

#x+y
x.__add__(y)
```

In [12]:
```
### getting to know your class

class test:
    print("This is just me, I am a package--take it or leave it")

dir(test)  # to print out all the magic functions a class has
```

```
This is just me, I am a package--take it or leave it
```

```
Out[12]: ['__class__',
          '__delattr__',
          '__dict__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__gt__',
          '__hash__',
          '__init__',
          '__init_subclass__',
          '__le__',
          '__lt__',
          '__module__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__setattr__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          '__weakref__']
```

```
In [ ]: # operator overlaod

        class Point:        # tries to simulate a point in 2-D coordinate system.
            def __init__(self, x, y):
                self.x = x
                self.y = y
        p1 = Point(2,3)
        p2 = Point(-1,2)
        #print(p1 + p2)            # it won't recognize it
        print(p1)
```

```
In [ ]: # operator overload

        class Point:
            def __init__(self, x, y):
                self.x = x
                self.y = y
```

```
        def __str__(self):  ## secondary: we change the meaning of str
            return "({0},{1})".format(self.x,self.y) # template string, followed by values

        def __add__(self,other):  ## we change the meaning of +
            x = self.x + other.x
            y = self.y + other.y
            return Point(x,y)

p1 = Point(2,3)
p2 = Point(-1,2)
print(p1+p2)
print(p1)

# 2+3 # outside the class + keeps its usual behavior
```

```
In [ ]: class Complex:
            def __init__(self, realpart, imagpart):
               self.r = realpart
               self.i = imagpart

x = Complex(3.0, -4.5)
#x = Complex() # positional arguments required
x.r, x.i
```

```
In [ ]: class MyClass:
            Greeting = ""
            def __init__(self, Name="there"):
                self.Greeting = Name + "!"

            def SayHello(self):
                print("Hello {0}".format(self.Greeting))

Myinstance=MyClass("John")
#Myinstance=MyClass() # default argument used
Myinstance.SayHello()
```

3.4 Inheritance

3.4.1 Parent vs child class

Say you have already defined a nice class with a lot of methods, overloaded many operators... But now you need a subclass with almost the same methods but with new and more specific things. Python allows you to do it quite smoothly. Indeed it suffices to call :

class *Subclass*(*Bigclass*) :
    indented lines of codes
    indented lines of codes
    indented lines of codes

3.4.2 Simple inheritance

By defaults the subclass or child class inherits features of the super class or parent class, adding new features to it. Of course inheritance goes in one direction only. This results into re-usability of code. For instance you do not need to initialize an instance of the child class again. (If you do, this will overwrite the init command in the parent class)

3.4.3 Multiple inheritance

Multiple inheritance is supported. In this case, the child class inherits the features of two (unrelated) parent classes. There are issues with the calling of the **init** of the parent classes (one way to make it work is shown in the example).

3.4.4 Overriding

A method or attribute of a parent class can be overridden by simply defining in the child class a method or attibute with the same name and the same sumber of parameters.

If a method is overridden in a class, the original method can still be accessed, but you have to do it by calling the method directly with the class name, like in the last example below.

```
In [31]: # Example of simple inheritance

         class User:   ## parent class

             def __init__(self, name):
                 self.name = name

             def printName(self):
                 print("Name  = ",self.name)

         class Programmer(User): # child class

             def doPython(self):      # new to the parent class
                 print("Programming Python")

         brian = User("brian")    # INSTANCE OF THE PARENT CLASS
         #brian.printName()
         #diana = Programmer("Diana") # INSTANCE OF THE CHILD CLASS
         #diana.printName()                    # inherited from parent class
         #diana.doPython()
         ## brian.doPython() inheritance does not go upwards

         ### Some useful command

         print(issubclass(Programmer,User)) # Allows you to check if a class is a subclass of
         print(issubclass(User,Programmer))

         print(isinstance(diana,Programmer))# Check if some object is an instance of some clas
         print(isinstance(diana,User))        # it also works with upper classes
```

In [36]:
```
# Multiple inheritance

class User:   ## parent class 1

    def __init__(self, name):
        self.name = name

    def printName(self):
        print("Name  = ",self.name)

class Origin:   ## parent class 2

    def __init__(self, country):
        self.country = country

    def printCountry(self):
        print("Country=",self.country)

class Programmer(User,Origin): # child class

    def __init__(self,name,country):    #
        User.__init__(self,name) # Calling constructors of parent class 1
        Origin.__init__(self,country) # Calling constructors of parent class 2

    def doPython(self):
        print("Programming Python")



brian = User("brian")
brian.printName()
diana = Programmer("Diana","Luxembourg")
diana.printName()  # inherited from parent class 1
diana.printCountry() # inherited from parent class 2
diana.doPython()
```

```
Name  =  brian
Name  =  Diana
Country= Luxembourg
Programming Python
```

In [37]:
```
# Simple inheritance overriding

class User:   ## parent class

    def __init__(self, name):
```

```
            self.name = name

        def printName(self):
            print("Name  = ",self.name)

    class Programmer(User): # child class

        #def __init__(self, name):
        #    self.name = name

        def doPython(self):
            print("Programming Python")

        def printName(self):  ## Overriding the method of the parent class
            pass

    brian = User("brian")
    brian.printName()

    diana = Programmer("Diana")
    diana.printName()       # inherited from parent class, but overriden
    diana.doPython()
```

```
Name  =  brian
Programming Python
```

```
In [40]: ## overriding a method and changing the number of arguments

    class Employee:

        def add(self, a, b):
            print('The Sum of Two = ', a + b)

    class Department(Employee):

        #def add(self, a, b, c):
        #    print('The Sum of Three = ', a + b + c)

        def add(self, a):
            print('The double  = ', a + a)

    emp = Employee()
    emp.add(10, 20)

    print('------------')
    dept = Department()
    #dept.add(50, 130, 90)
    dept.add(50)
```

```
The Sum of Two =  30
------------
The double  =  100
```

```
In [42]: ## Funny: an overriden method can still be called from within the child class

         class Employee:

             def message(self):
                 print('This message is from Employee Class')

         class Department(Employee):

             def message(self): #overrdiing the parent method
                 #Employee.message(self)  # the overriden method can still be called
                 super().message()    # another way to call the overriden method
                 print('This Department class is inherited from Employee')

         emp = Employee()
         emp.message()

         print('------------')
         dept = Department()
         dept.message()
```

```
This message is from Employee Class
------------
This message is from Employee Class
This Department class is inherited from Employee
```

3.5 Diamond problem

Diamond problem (also called the "deadly diamond of the death") is an ambiguity that arises when two classes B and C inherit from a class A, and class D inherits from B and C. If there is a method in A that is overriden by both B and C, but not by D, then which version of the method does D inherit: that of B or that of C?

It is called Diamond because of the shape of the class diagram.

The Diamond problem is not specific to Python. Any Object Orientated Programming language allowing for multiple inheritance faces it.

In AI, the Nixon diamond is a scenario in which default assumptions lead to mutually inconsistent conclusions. The scenario is: * usually, Quakers are pacifist * usually, Republicans are not pacifist * Richard Nixon is both a Quaker and a Republican Since Nixon is a Quaker, one could assume that he is a pacifist; since he is Republican, however, one could also assume he is not a pacifist. The problem is how a formal logic of nonmonotonic reasoning should deal with such cases.

3.5.1 Method resolution order

Method Resolution Order or 'MRO' in short denotes the way a programming language handles diamond-like inheritance.

MRO old style (Python 2.2) follow the "naive", depth-first left-to-right approach : DBAC

MRO new style (Python 2.3 and above) uses so-called C3 linearization algorithm. It often yields the same outcome as the naive approach, but not always.

You can check the Method Resolution Order of a class diragram. Python provides a **mro** attribute and the mro() method. With these, you can get the resolution order.

In the Diamond example, the MRO old style is DBAC, while the MRO new style is DBCA. The outcome is, or is not, the same depending on what each class does (see examples below).

In [44]:
```
# diamond problem and MRO--C3 algorithm vs depth first

class A:
    def x(self):
        print('I am in A')

class B(A):
    def x(self):
        print('I am in B')
class C(A):
    def x(self):
        print('I am in C')

class D(B,C):    # order of arguments is important
        pass
# class D(C,B)

d=D()
d.x()

print(D.__mro__)  # Display the lookup order
print(D.mro())
```

```
I am in B
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
```

In [45]:
```
# diamond problem and MRO-- C3 algorithm vs depth first

class A:
    def x(self):
        print('I am in A')

class B(A):
    pass
class C(A):
```

```
        def x(self):
            print('I am in C')

    class D(B,C):    # order of arguments is important
            pass
    # class D(C,B)

    d=D()
    d.x()

    print(D.__mro__)  # Display the lookup order
    print(D.mro())
```

```
I am in C
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
```

3.6 Class decorators

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of a method. Decorators are prefixed with @. We look below at two decorators: the property and setter decorators.

3.6.1 @property

It makes a method accessible like an attribute in read-only mode. Here is a fun fact about python classes: If you change the value of an attribute inside a class, the other attributes that are derived from the attribute you just changed don't automatically update. You need to make the derived attribute a method decorated with the @ property.

Note the method is made accessible in read-only mode. All attributes in a Python class are **public by default**. Any attribute can be accessed from outside the class environment and modified. The @Property decorator hence provides a way to encapsulate data.

```
In [47]: ## The problem property allows to handle

    class employee:
        def __init__(self, first,last):
            self.first=first
            self.last=last
            self.email= first + last + "@email.com"

    a=employee("Marc","Coyle")
```

```
In [48]: a.email
    a.first="John"
    a.first
    a.email
```

```
Out[48]: 'MarcCoyle@email.com'
```

```
In [50]: ### property (cont')

         class employee:
             def __init__(self, first,last):
                 self.first=first
                 self.last=last
             @property #Read only access as if it was an attribute
             def email(self):
                 return "{}{}@email.com".format(self.first,self.last)

         a=employee("Marc","Coyle")
         #a.email()

         a.first="John"
         a.email    # the method is now accessible as an attribute in read_only mode and the u
```

```
Out[50]: 'JohnCoyle@email.com'
```

3.7 @setter

You use the @setter decorator to enable the chain of updates in the backward direction, from the derived attribute to the initial ones. This presupposes that the method is made accessible as a write attribute.

```
In [53]: #why we need the setter decorator

         class employee:
             def __init__(self, first,last):
                 self.first=first
                 self.last=last

             @property # as before
             def email(self):
                 return "{}{}@email.com".format(self.first,self.last)

             @property # new
             def fullname(self):
                 return "{} {}".format(self.first,self.last)


         a=employee("Marc","Coyle")
         #a.email()

         a.first="John"
         a.fullname
         a.fullname="Martin Theobald" # I am changing the value of the derived attribute
         a.first
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-53-01515a2ccb17> in <module>
 20 a.first="John"
 21 a.fullname
---> 22 a.fullname="Martin Theobald" # I am changing the value of the derived attribute
 23 #a.first


AttributeError: can't set attribute
```

In [56]:
```
### why we need the setter

class employee:
    def __init__(self, first,last):
        self.first=first
        self.last=last

    @property # as before
    def email(self):
        return "{}{}@email.com".format(self.first,self.last)

    @property # new
    def fullname(self):
        return "{} {}".format(self.first,self.last)

    @fullname.setter  # new
    def fullname(self,name): #name is the value we are trying to set
        first,last = name.split(' ')  # we split the full name into two
        self.first=first # we set the first and last names equals to these values
        self.last=last

a=employee("Marc","Coyle")
#a.email()

#a.first="John"
#a.fullname
a.fullname="Martin Theobald" # I am changing the value of the derived attribute
a.first
```

Out[56]:
```
'Martin'
```