5 - Classes and objects

1. Features of object-oriented programming
2. Syntax of the class and instance creation and use
3. Parameterized creation of instances
4. Recursive definitions
5. Class inheritance

5.1 - Features of object-oriented programming

- New Python objects can be created as instances of classes which are defined by the **attributes** and **methods** applying to these objects.
- The **attributes** of a class are properties used to describe its instances.
- The **methods** of a class are functions that can be applied to its instances.

5.1 - Features of object-oriented programming

- Classes allow typing to be customized, a class being viewed as a customized type and an instance of it as a typed object.

- The **inheritance** mechanism allows new classes to be defined from old ones by specifying them.

5.2 - Syntax of the creation and use of classes and instances

- The syntax of a class creation is the following :

  class < *name of the class* > ***:***

  < *description comment*>

  < *instruction block* >

  The < *description comment* > part is a string describing the class. The < *instruction block* > in particular includes attribute initializations and method definitions.

5.2 - Syntax of the creation and use of classes and instances

- The syntax of a method definition is that of a function definition with a first obligatory parameter *self* representing the instance of the class concerned with the method call.
- The use of an attribute *attr* for an object *obj* is performed with the expression *obj.attr*.
- The use of a method *m,* which has *m(self, arg_1, arg_2, …, arg_n)* as a definition head, associated with an object *obj* is performed with the function call *obj.m(arg_1, arg_2, …, arg_n).*

5.2 - Syntax of the creation and use of classes and instances

```
>>> class Rectangle :
    u"Geometric rectangles"
   #attributes
    length = None
    width = None
    #methods
    def perimeter(self) :
        return (self.length +self.width)*2
    def area(self):
        return self.length *self.width
```

```
>>> r1 = Rectangle()
>>> r1.length= 5
>>> r1.width = 3
>>> print  u"area : ", r1.area(),   u"   perimeter : " , r1.perimeter()
area :15    perimeter : 16
```

5.3 - Parameterized creation of instances

- The parameterization of instance creation is used to initialize the values of the attributes at the time of the instance creation.

- In the class definition, a special method *__init__* must be defined. This method is executed at the moment that an instance is created.

- The syntax of the *__init__* definition is the following :

  def __init__(*self,* arg_1*,* arg_2*, …,* arg_n***) :***

  < statement block >

  In this definition, the first parameter *self* represents the instance of the class being created.

- The instruction of creation for the instance *obj* of a class *C* has the form : *obj = C(*arg_1*,* arg_2*, …,* arg_n*)*

5.3 -Parameterized creation of instances

```
>>> class Rectangle :
    u"Geométric rectangles"
    def __init__(self, lo, la)
      self.length = lo
      self.width = la
    def perimeter(self) :
       return (self.length +self.width)*2
    def area(self):
       return self.length *self.width
```

```
>>> r1 = Rectangle(5,3)
>>> print  u"area: ", r1.area(),   u"   perimeter: " , r1.perimeter()
area: 15    perimeter: 16
>>> r1.length = 10
>>> print  u"area: ", r1.area(),   u"   perimeter: " , r1.perimeter()
area: 30    perimeter: 26
```

5.4 - Recursive definitions

Recursivity can be introduced in the value of attributes or in the definition of methods:

- Values of attributes can refer to instances of the same class.
- The definition of a method can include calls to the same method.

5.4 - Recursive definitions

```
>>> class Liste :
    def __init__(self, h = "", t = None):
        self. head = h
        self.tail = t
    def length(self):
        if self.tail == None :
            return 1
        else:
            return self.tail.length() +1
    def display(self):
        if self.tail == None :
            return [self.head]
        else :
            return [self.head] +self.tail.display()
```

```
>>> l1 = Liste("Marie")
>>> l2 = Liste("aime", l1)
>>>l3 = Liste("Jean",l2)
>>> print l3.length()
3
>>> print l3.display()
['Jean', 'aime', 'Marie']
```

5.5 - Class inheritance

- A class can be defined by **inheritance** from another class. It takes its attributes and methods but it can add new ones to particularize the mother class.

- The syntax of the head for the definition of a class C_1 that inherits a class C_2 is the following: ***class*** $C_1(C_2)$ **:**

- In the definition of a class C_1 that inherits a class C_2 , methods of C_2 can be redefined. This mechanism is called **method overriding**.

5.5 - Class inheritance

```
>>> class LabelledWord :
    def __init__(self, w = "", l = "") :
        self.word = w
        self.label = l
    def display(self) :
        return (self.word, self.label)

>>> class LabelledWordList(Liste) :
    def __init__(self, lw=LabelledWord(), t=None) :
        Liste.__init__(self, lw,t)
```

```
>>> lw1 = LabelledWord("Jean", "NP")
>>> lw2 = LabelledWord("aime", "V")
>>> lw3 = LabelledWord("Marie", "NP")
>>> l1 = LabelledWordList(lw3)
>>> l2 = LabelledWordList(lw2, l1)
>>> l3 = LabelledWordList(lw1, l2)
>>> print l3.display()
[<__main__.LabelledWord instance at 0xcd7300>, <__main__.LabelledWord instance at
    0xcd7328>, <__main__.LabelledWord instance at 0xcd7350>]
```

5.5 - Class inheritance

```
>>> class LabelledWordList(Liste):
    def __init__(self, lw=LabelledWord(), t=None):
        Liste.__init__(self, lw,t)
    def display(self):
        if self.tail == None :
            return [self.head.display()]
        else :
            x = [self.head.display()]
            x.extend(self.tail.display())
            return x
```

```
>>> lw1 = LabelledWord("Jean", "NP")
>>> lw2 = LabelledWord("aime", "V")
>>> lw3 = LabelledWord("Marie", "NP")
>>> l1 = LabelledWordList(lw3)
>>> l2 = LabelledWordList(lw2, l1)
>>> l3 = LabelledWordList(lw1, l2)
>>> print l3.display()
[('Jean', 'NP'), ('aime', 'V'), ('Marie', 'NP')]
```

5.6 - Exercises

1. *InflectedWord* is a class with 3 attributes : *word, lemma, category*. The first attribute represents an inflected word, the second one its lemma and the last one its grammatical category. It has a unique method, *display,* which returns the values of the attributes in the form of a triple.
   Define the *InflectedWord* class and the subclasses *InflectedVerb* and *InflectedNoun* that take the inflection parameters into account for verbs and nouns in English.

2. A syntactic tree is a tree labelled in which every leaf is labelled with a word and every other node with a grammatical category. We consider syntactic trees with two daughters for each node at most.

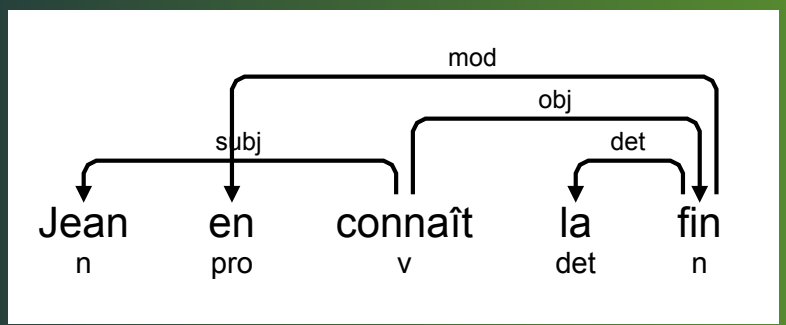
   a) Write a class *BinaryTree* to represent binary trees (trees with two daughters for each node at most). The class has 3 attributes: *label, left_daughter, right_daughter*. It has a method *depth* which returns the depth of a tree and a method *display* which returns a presentation of the tree in the form of a parenthesized word.

   b) Write a class *SyntacticTree* which inherits the *BinaryTree* class with an additional method *treeyield* which returns the list of words that are the leaves of the tree, in the same order as in the tree.

   c) With the previous classes, write a program that displays the syntactic tree of the sentence *"the teacher congratulates the good students"*.

5.6 - Exercises

3. We propose to create a class *DepTree* to deal with syntactic dependency trees. A syntactic dependency tree is a tree, the nodes of which are labelled with the words of a sentence and the edges are labelled with grammatical functions. Here is an example for the sentence *"Jean en connaît la fin" ("Jean knows the end of it")* .



a) Write a definition of the *DepTree* class with 3 attributes: *word, position, dependencies*:

- *word* is a string representing the word labeling the root of the tree,
- *position* is a natural number indicating the position in the sentence of the word labelling the root ,
- *dependencies* is a list of dependencies in the form of pairs *(label, daughter)*, where *label* is a string representing a dependency and *daughter* is the dependency tree, the root of which is the target of the dependency.

5.6 - Exercises

Moreover, define the following methods:

- tree*yield* that gives the list of words labelling the nodes of the dependency tree ranked in the linear order of the sentence,
- *depth* giving the depth of the tree,
- *display* returning the tree in the form a parenthesized string

b) With the *DepTree* class, write a program displaying the dependency tree of the sentence *"Jean en connaît la fin"*