



Formation Python

Partie 2 : Programmation Orienté Objet

POO

Définition

La programmation orientée objet (POO) est un concept de programmation très puissant qui permet de structurer ses programmes **d'une manière nouvelle**. En POO, on définit un « objet » qui peut contenir des « attributs » ainsi que des « méthodes » qui agissent sur lui-même. En Python, on utilise une « classe » pour construire un objet.



Jon



Objet

Nom : Jon
Sexe: Homme
Note M1:14
Note M2:15
Note M3:12
Note M4:15



Attributs

Calcul_note : Méthode

Classe : Etudiant



POO

Définition

Une classe définit des objets qui sont **des instances (des représentants)** de cette classe. Les objets peuvent posséder des attributs (**variables associées aux objets**) et des méthodes (qui sont **des fonctions associées aux objets** et qui peuvent agir sur ces derniers ou encore les utiliser).



POO

Construction d'une classe

```
>>> class Etudiant:
...     pass
...
>>> etudiant1=Etudiant()
>>> etudiant1.nom='Jon'
>>> etudiant1.sexe='homme'
>>> etudiant1.noteM1=14
>>> etudiant1.noteM2=15
>>> etudiant1.noteM3=12
>>> etudiant1.noteM4=15
>>> Etudiant
<class '__main__.Etudiant'>
>>> type(Etudiant)
<class 'type'>
>>> etudiant1
<__main__.Etudiant object at 0x000001EC41536680>
>>> isinstance (etudiant1,Etudiant)
True
```

POO

Construction d'une classe

```
>>> dir(etudiant1)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'nom', 'noteM1', 'noteM2', 'noteM3', 'noteM4', 'sexe']
>>> etudiant1.noteM5=18
>>> dir(etudiant1)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__new__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'nom', 'noteM1', 'noteM2', 'noteM3', 'noteM4', 'noteM5', 'sexe']
>>> etudiant1.nom
'Jon'
>>> etudiant1.__dict__
{'nom': 'Jon', 'sexe': 'homme', 'noteM1': 14, 'noteM2': 15, 'noteM3': 12, 'noteM4': 15, 'noteM5': 18}

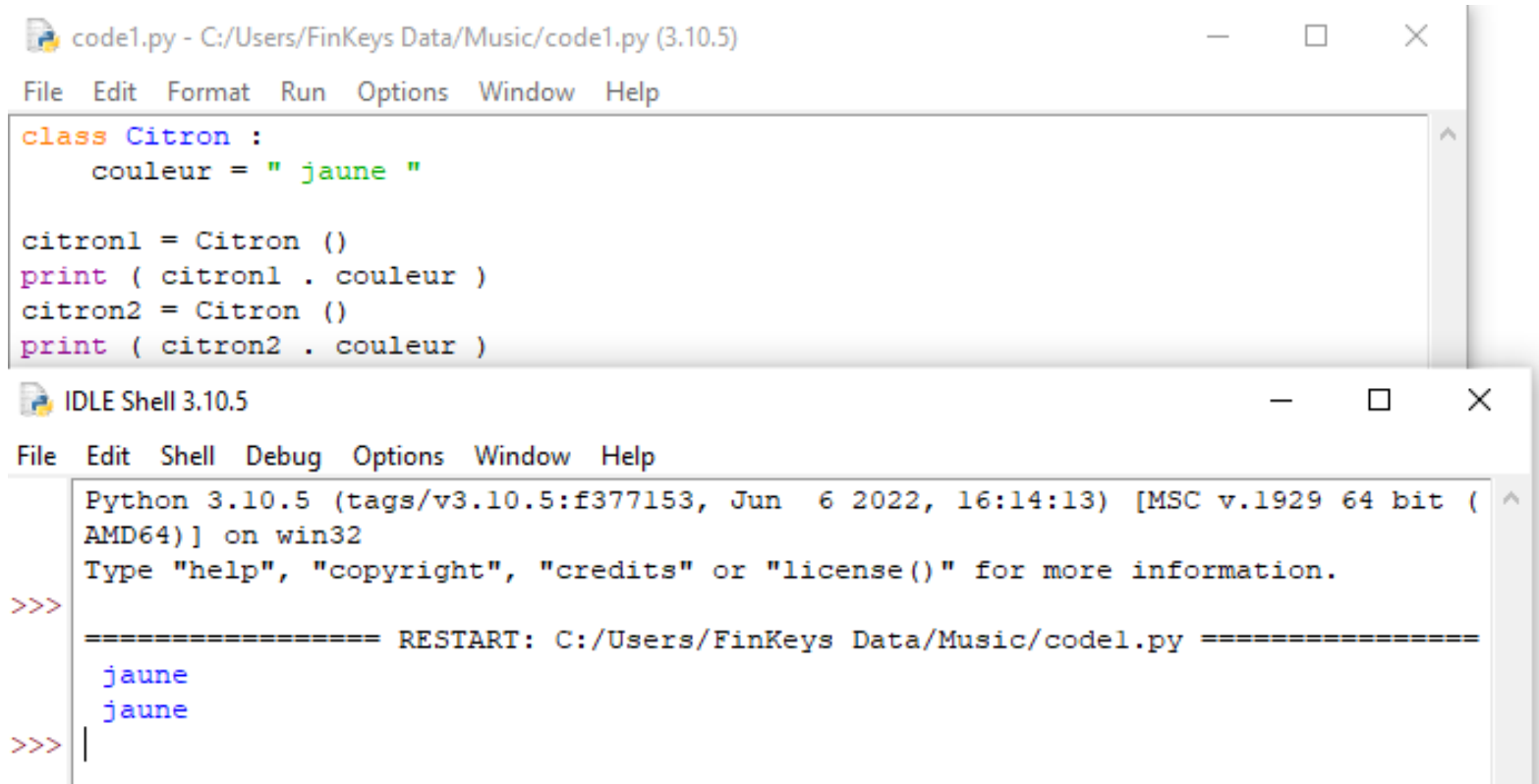
>>> del etudiant1.noteM5
>>> etudiant1.__dict__
{'nom': 'Jon', 'sexe': 'homme', 'noteM1': 14, 'noteM2': 15, 'noteM3': 12, 'noteM4': 15}
>>> |
```

Une variable ou attribut d'instance est **une variable accrochée à une instance** et qui est spécifique à cette instance. Cet attribut n'existe donc pas forcément pour toutes les **instances d'une classe donnée**, et d'une instance à l'autre il ne prendra pas forcément la même valeur. On peut retrouver tous les attributs d'instance d'une instance donnée avec une syntaxe `instance.__dict__`.

POO

Attributs de la classe

Une variable de classe ou attribut de classe est un attribut qui sera identique pour chaque instance.



The image shows a screenshot of a Python IDE with two windows. The top window, titled 'code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)', contains the following Python code:

```
class Citron :  
    couleur = " jaune "  
  
citron1 = Citron ()  
print ( citron1 . couleur )  
citron2 = Citron ()  
print ( citron2 . couleur )
```

The bottom window, titled 'IDLE Shell 3.10.5', shows the output of the code execution. It displays the Python version and system information, followed by a restart message and the output of the print statements:

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun  6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> ===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====  
jaune  
jaune  
>>> |
```

POO

Attributs de la classe

```
>>> citron1.couleur='rouge'
>>> citron1.couleur
'rouge'
>>> citron2.couleur
'jaune '
>>> citron1.name='citron'
>>> citron1.name
'citron'
>>> citron1.name='limon'
>>> citron1.name
'limon'
>>> |
```

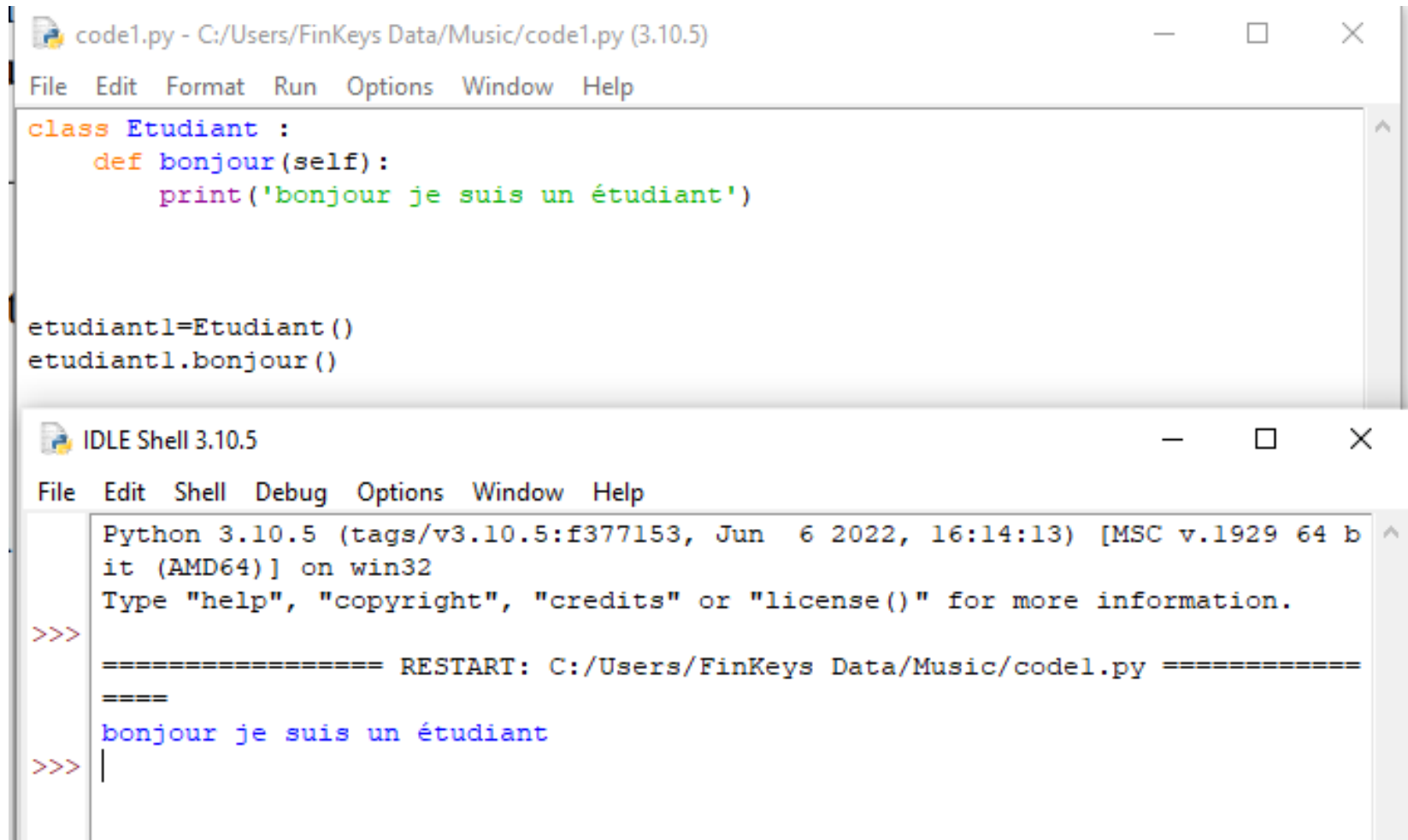
Attributs de la classe

VS

Attributs d'instance

POO

Les méthodes



The image shows a screenshot of the Python IDLE environment. The top window, titled 'code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)', contains the following Python code:

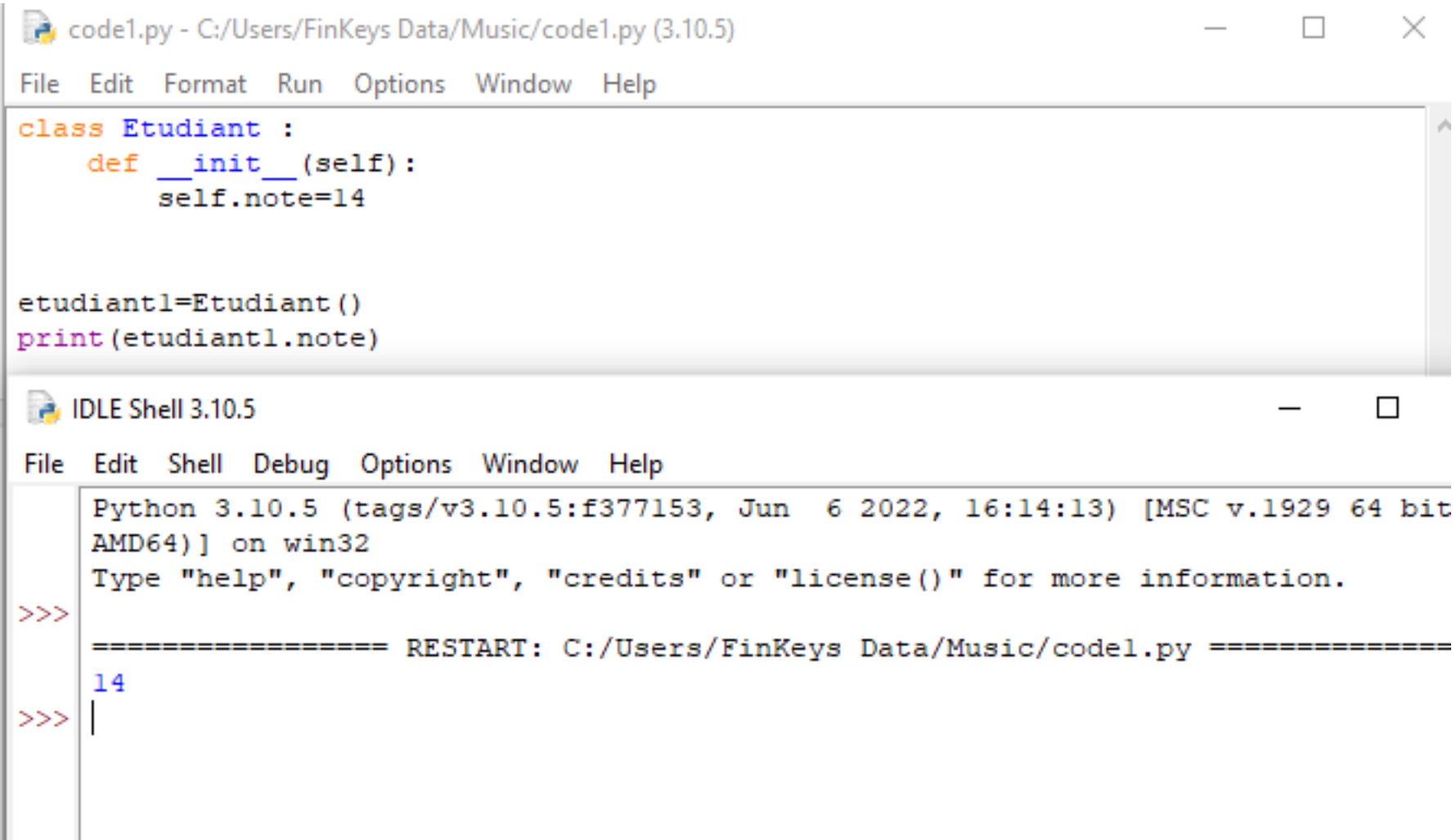
```
class Etudiant :  
    def bonjour(self):  
        print('bonjour je suis un étudiant')  
  
etudiant1=Etudiant()  
etudiant1.bonjour()
```

The bottom window, titled 'IDLE Shell 3.10.5', shows the output of running the code. It displays the Python version and architecture, followed by a restart message and the output of the 'bonjour' method call:

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 b  
it (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> ===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====  
====  
bonjour je suis un étudiant  
>>> |
```


POO

Le constructeur



The image shows a Python IDE window titled 'code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following Python code:

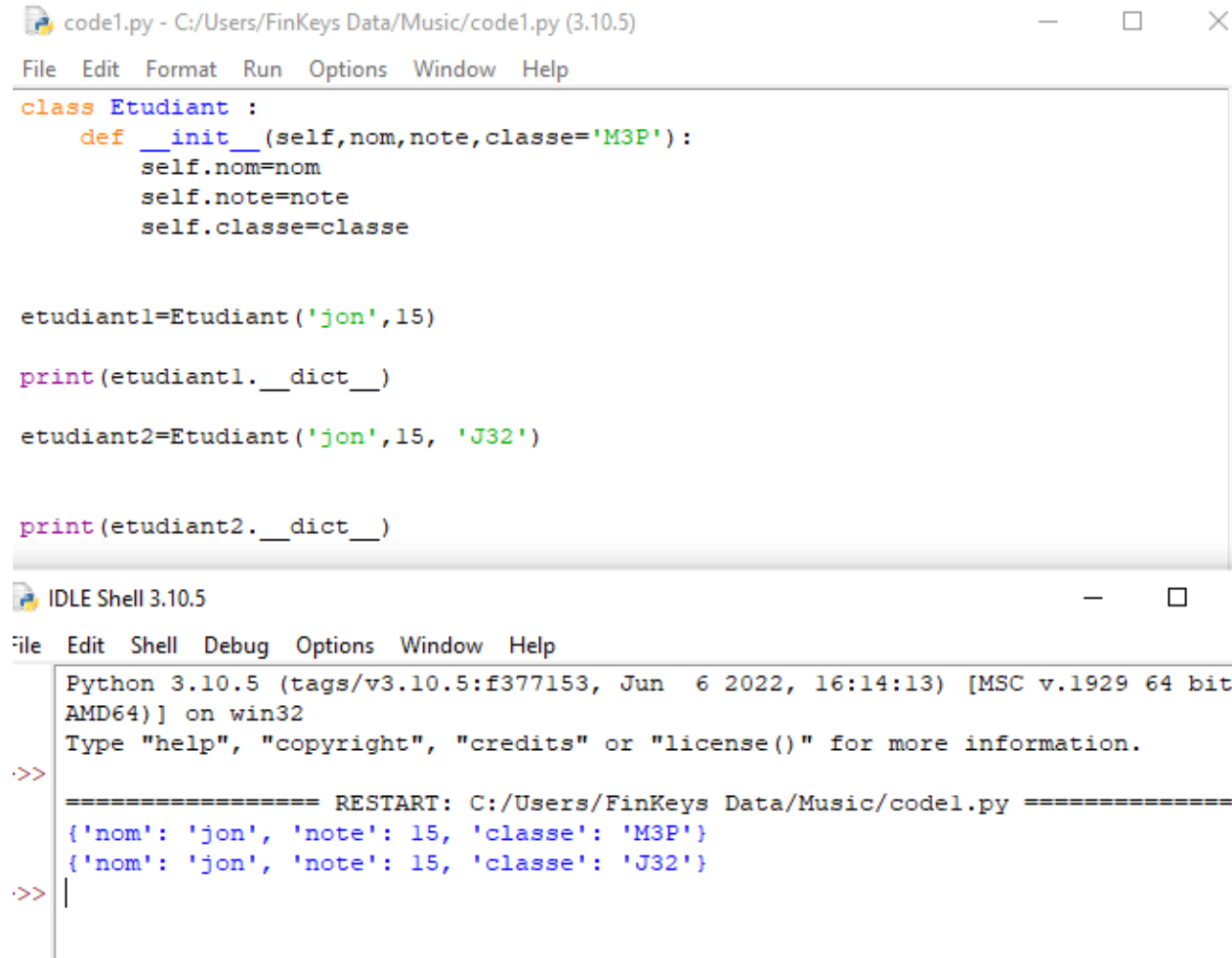
```
class Etudiant :  
    def __init__(self):  
        self.note=14  
  
etudiant1=Etudiant()  
print(etudiant1.note)
```

Below the code editor is the 'IDLE Shell 3.10.5' window. Its menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell displays the following output:

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit  
AMD64] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====  
14  
>>> |
```

POO

Le constructeur



The image shows a screenshot of a Python IDE with two windows. The top window, titled 'code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)', contains the following Python code:

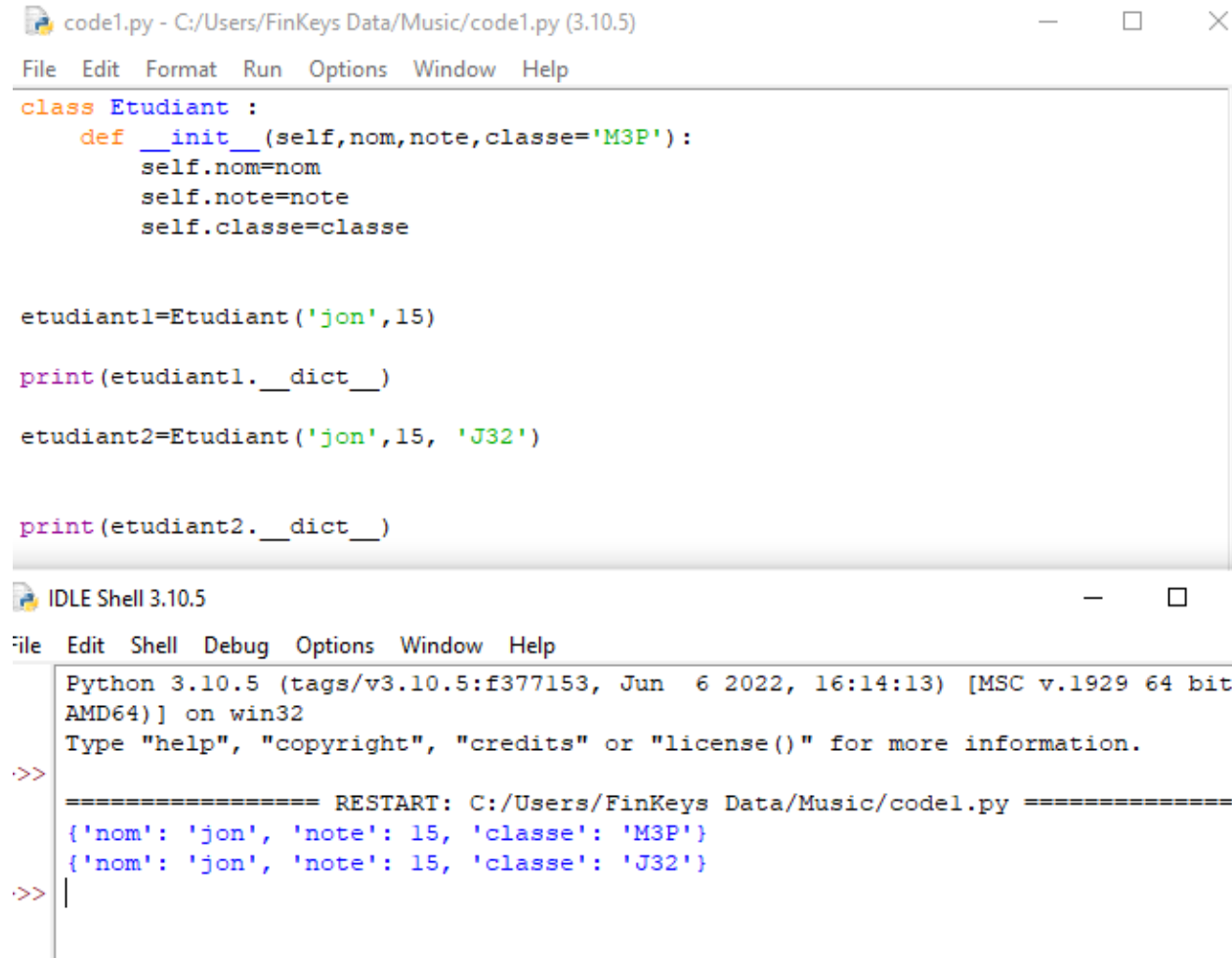
```
class Etudiant :  
    def __init__(self,nom,note,classe='M3P'):  
        self.nom=nom  
        self.note=note  
        self.classe=classe  
  
etudiant1=Etudiant('jon',15)  
  
print(etudiant1.__dict__)  
  
etudiant2=Etudiant('jon',15, 'J32')  
  
print(etudiant2.__dict__)
```

The bottom window, titled 'IDLE Shell 3.10.5', shows the output of the code execution:

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit  
AMD64] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====  
{'nom': 'jon', 'note': 15, 'classe': 'M3P'}  
{'nom': 'jon', 'note': 15, 'classe': 'J32'}  
>>> |
```

POO

Le constructeur



The image shows a screenshot of a Python IDE with two windows. The top window, titled 'code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)', contains the following Python code:

```
class Etudiant :  
    def __init__(self,nom,note,classe='M3P'):  
        self.nom=nom  
        self.note=note  
        self.classe=classe  
  
etudiant1=Etudiant('jon',15)  
  
print(etudiant1.__dict__)  
  
etudiant2=Etudiant('jon',15, 'J32')  
  
print(etudiant2.__dict__)
```

The bottom window, titled 'IDLE Shell 3.10.5', shows the output of the code execution:

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit  
AMD64] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>>  
===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====  
{'nom': 'jon', 'note': 15, 'classe': 'M3P'}  
{'nom': 'jon', 'note': 15, 'classe': 'J32'}  
>>> |
```

POO

Polymorphisme

En programmation, le polymorphisme est la capacité d'une fonction (ou méthode) à se comporter différemment en fonction de l'objet qui lui est passé. Une fonction donnée peut donc avoir plusieurs définitions.

```
>>> sorted('Big Data')
[' ', 'B', 'D', 'a', 'a', 'g', 'i', 't']
>>> sorted([1,5,1,4,7,9,8])
[1, 1, 4, 5, 7, 8, 9]
>>> |
```

Le polymorphisme est intimement lié au concept de redéfinition des opérateurs.

La redéfinition des opérateurs est la capacité à redéfinir le comportement d'un opérateur en fonction des opérandes utilisées.

```
>>> 10+2
12
>>> 'cours'+' ' +'python'
'cours python'
>>> |
```

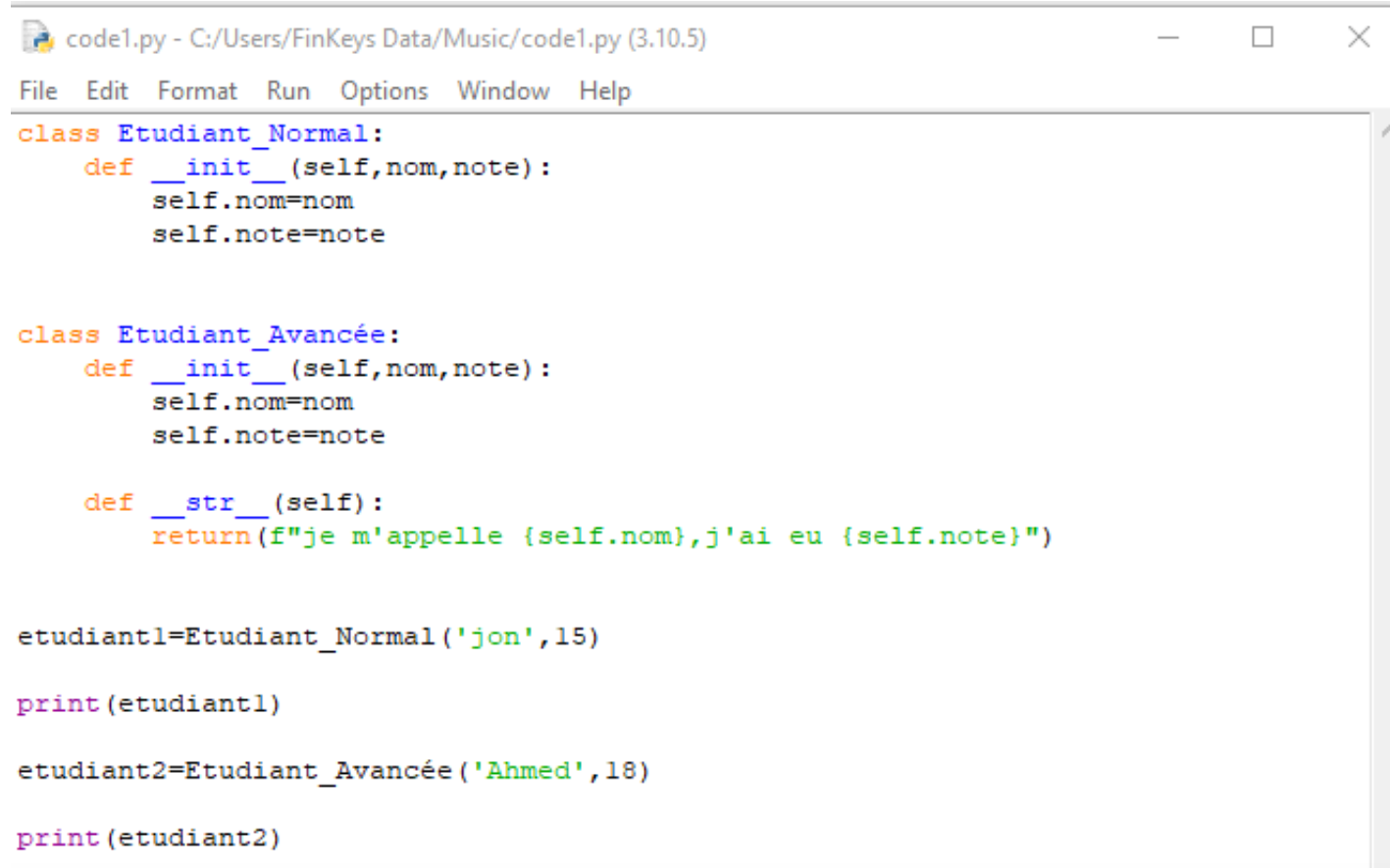
POO

Polymorphisme

Une méthode magique est une méthode spéciale dont le nom est entouré de double underscores. Par exemple, la méthode `__init__()` est une méthode magique. Ces méthodes sont destinées au fonctionnement interne de la classe. Nombre d'entre elles sont destinées à changer le comportement de fonctions ou opérateurs internes à Python avec les instances d'une classe que l'on a créée.

POO

Polymorphisme



```
code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)
File Edit Format Run Options Window Help

class Etudiant_Normal:
    def __init__(self,nom,note):
        self.nom=nom
        self.note=note

class Etudiant_Avancée:
    def __init__(self,nom,note):
        self.nom=nom
        self.note=note

    def __str__(self):
        return(f"je m'appelle {self.nom},j'ai eu {self.note}")

etudiant1=Etudiant_Normal('jon',15)

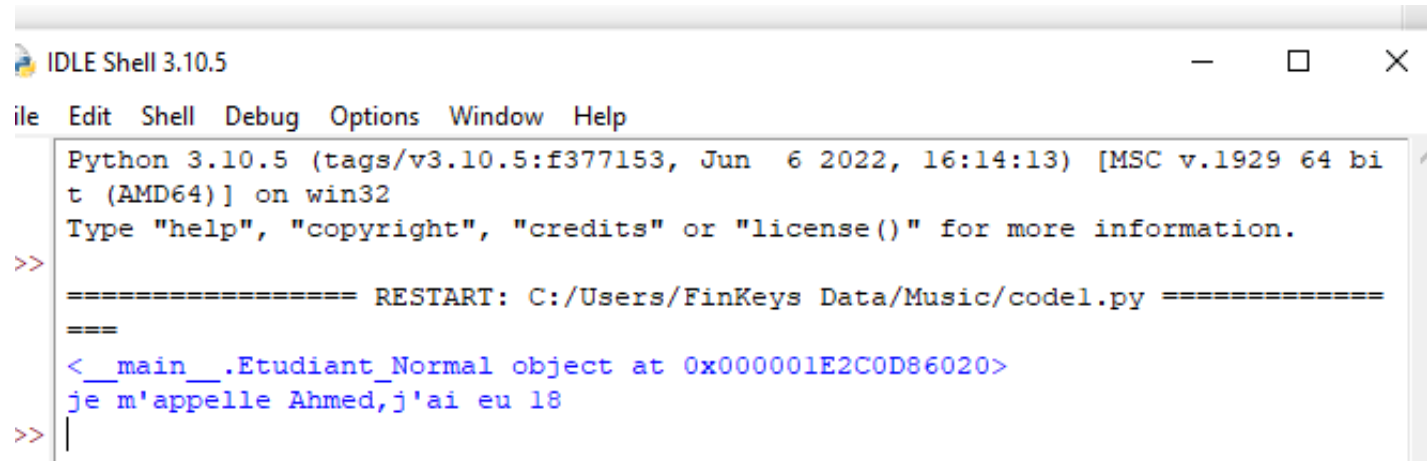
print(etudiant1)

etudiant2=Etudiant_Avancée('Ahmed',18)

print(etudiant2)
```

POO

Polymorphisme



```
IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>
===== RESTART: C:/Users/FinKeys Data/Music/codel.py =====
>>>
<__main__.Etudiant_Normal object at 0x000001E2C0D86020>
je m'appelle Ahmed,j'ai eu 18
>>>
```

POO

Polymorphisme

Il existe une multitude de méthodes magiques, en voici quelques unes :

- `.__repr__()` : redéfinit le message obtenu lorsqu'on tape le nom de l'instance dans l'interpréteur;
- `.__add__()` : redéfinit le comportement de l'opérateur +; — `.__mul__()` : redéfinit le comportement de l'opérateur *.
- `.__del__()` : redéfinit le comportement de la fonction del.

POO

Polymorphisme

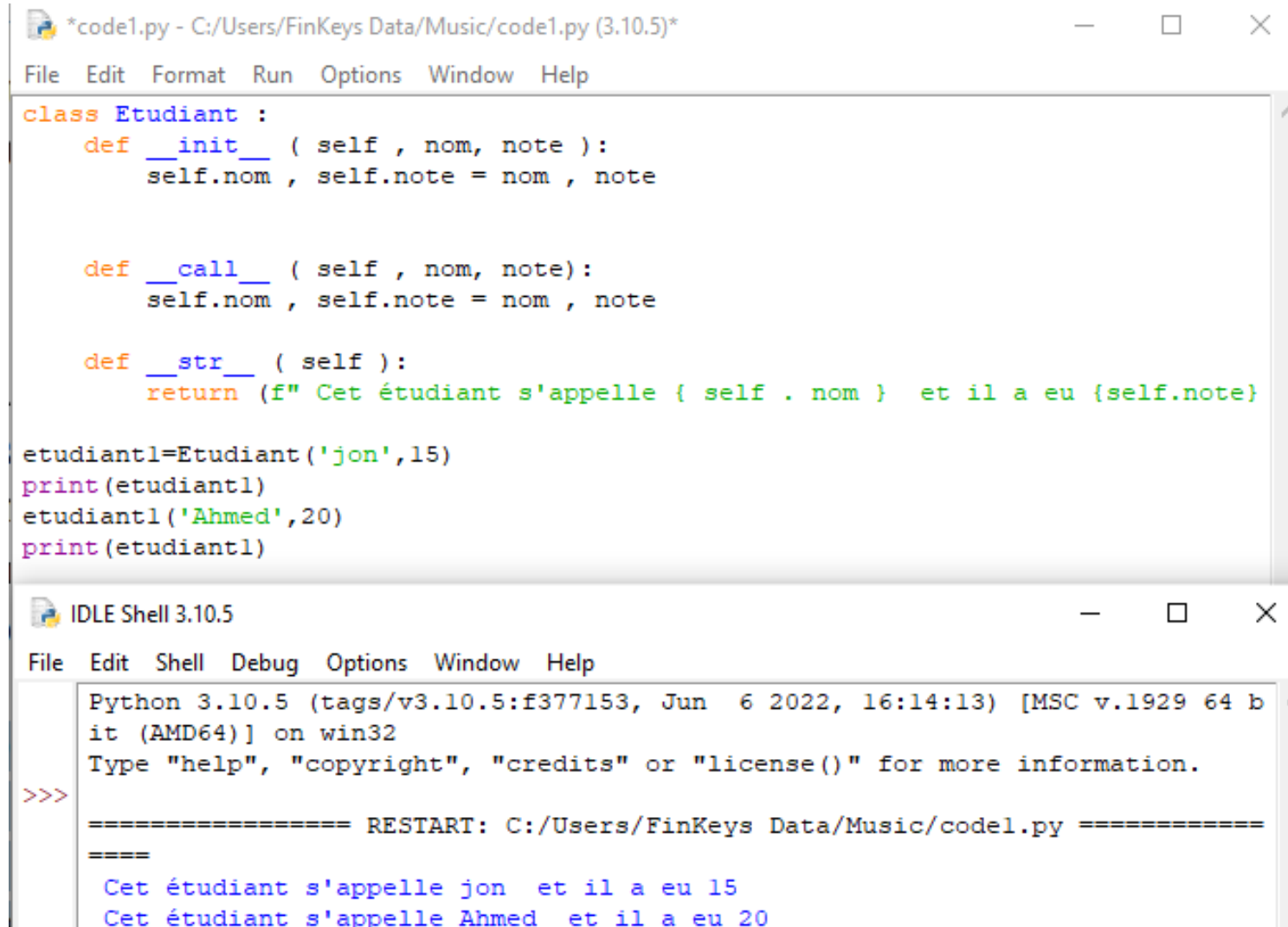
Si on conçoit une classe produisant des objets séquentiels (comme des listes ou des tuples), il existe des méthodes magiques telles que :

- `.__len__()` : redéfinit le comportement de la fonction `len()`;
- `.__getitem__()` : redéfinit le comportement pour récupérer un élément;
- `.__getslice__()` : redéfinit le comportement avec les tranches.

Certaines méthodes magiques font des choses assez impressionnantes. Par exemple, la méthode `.__call__()` crée des instances que l'on peut appeler comme des fonctions ! Dans cet exemple, nous allons vous montrer que l'on peut ainsi créer un moyen inattendu pour mettre à jour des attributs d'instance :

POO

Polymorphisme



The image shows a screenshot of a Python IDE with two windows. The top window, titled '*code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)*', contains a Python class definition for 'Etudiant' and some test code. The class has three methods: '__init__', '__call__', and '__str__'. The test code creates an instance of 'Etudiant' with the name 'jon' and the value 15, prints it, then updates the name to 'Ahmed' and the value to 20, and prints it again. The bottom window, titled 'IDLE Shell 3.10.5', shows the output of the code execution. It displays the Python version and system information, followed by a restart message for the file 'code1.py'. The output shows the string representation of the 'Etudiant' object twice: 'Cet étudiant s'appelle jon et il a eu 15' and 'Cet étudiant s'appelle Ahmed et il a eu 20'.

```
*code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)*
File Edit Format Run Options Window Help

class Etudiant :
    def __init__ ( self , nom, note ) :
        self.nom , self.note = nom , note

    def __call__ ( self , nom, note):
        self.nom , self.note = nom , note

    def __str__ ( self ) :
        return (f" Cet étudiant s'appelle { self . nom } et il a eu {self.note}")

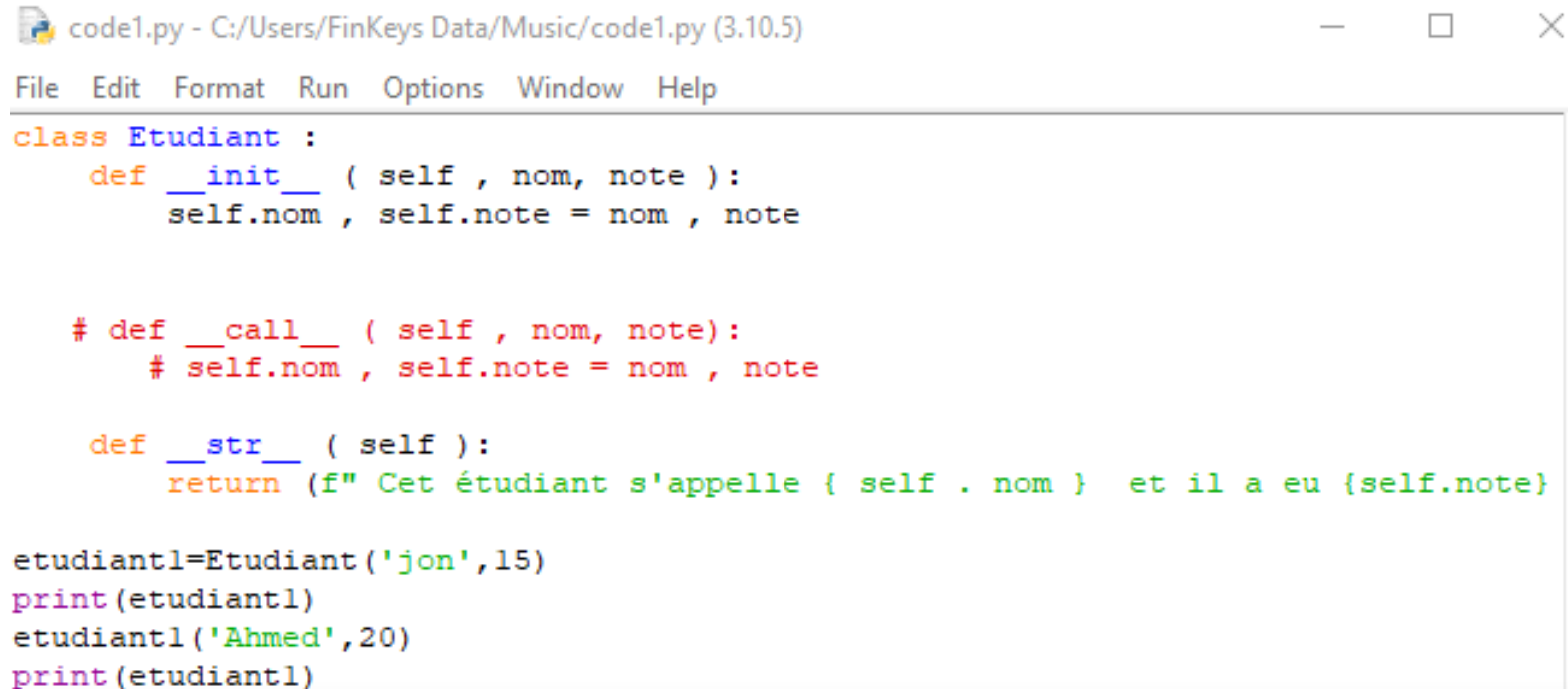
etudiant1=Etudiant('jon',15)
print(etudiant1)
etudiant1('Ahmed',20)
print(etudiant1)
```

```
IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help

Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====
====
Cet étudiant s'appelle jon et il a eu 15
Cet étudiant s'appelle Ahmed et il a eu 20
```

POO

Polymorphisme

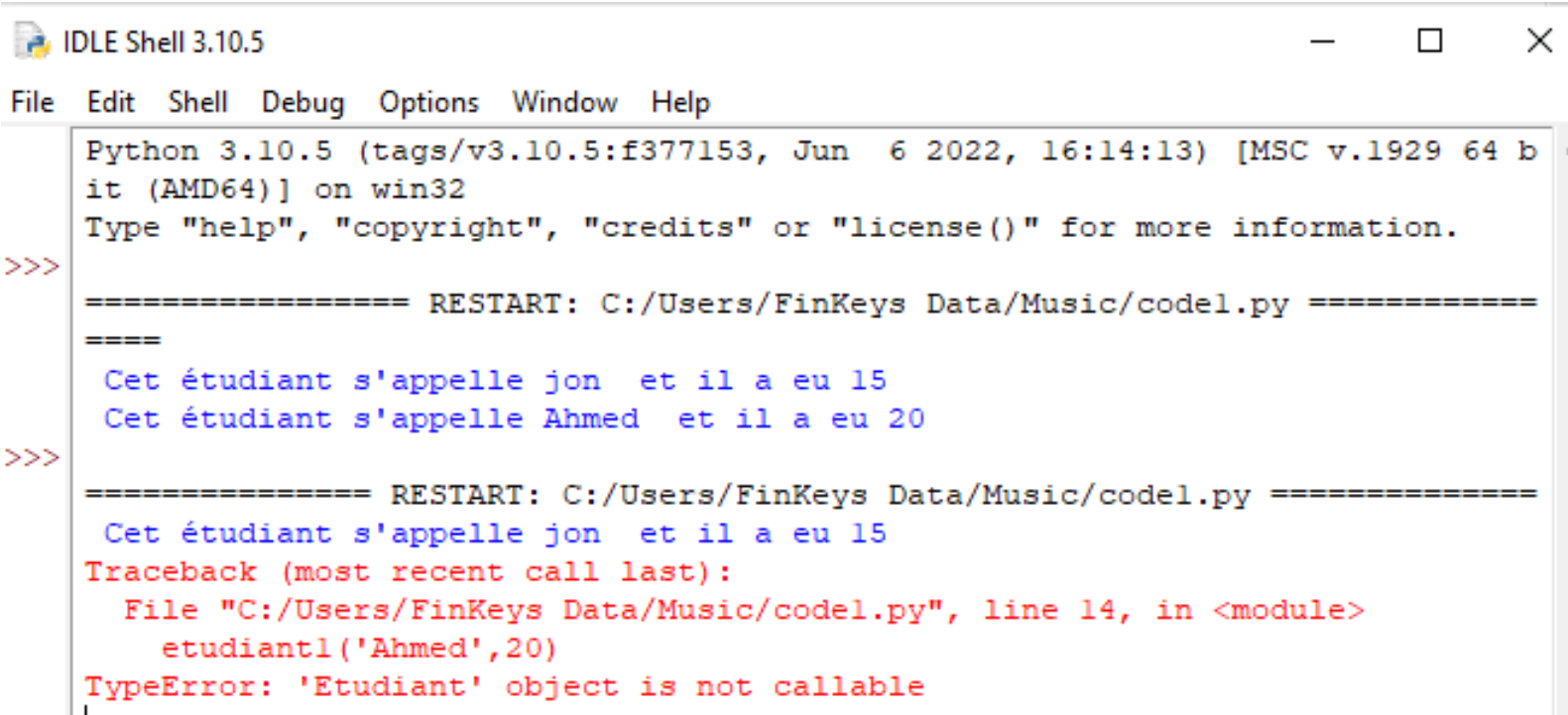


The image shows a screenshot of a Python IDE window titled "code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)". The window has a menu bar with "File", "Edit", "Format", "Run", "Options", "Window", and "Help". The code editor contains the following Python code:

```
class Etudiant :  
    def __init__ ( self , nom, note ):  
        self.nom , self.note = nom , note  
  
    # def __call__ ( self , nom, note):  
    #     self.nom , self.note = nom , note  
  
    def __str__ ( self ):  
        return (f" Cet étudiant s'appelle { self . nom }  et il a eu {self.note}  
  
etudiant1=Etudiant('jon',15)  
print(etudiant1)  
etudiant1('Ahmed',20)  
print(etudiant1)
```

POO

Polymorphisme



```
IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/FinKeys Data/Music/codel.py =====
====
Cet étudiant s'appelle jon et il a eu 15
Cet étudiant s'appelle Ahmed et il a eu 20
>>>
===== RESTART: C:/Users/FinKeys Data/Music/codel.py =====
Cet étudiant s'appelle jon et il a eu 15
Traceback (most recent call last):
  File "C:/Users/FinKeys Data/Music/codel.py", line 14, in <module>
    etudiant1('Ahmed',20)
TypeError: 'Etudiant' object is not callable
```

POO

Héritage

En programmation, l'héritage est la capacité d'une classe d'hériter des propriétés d'une classe préexistante. On parle **de classe mère** et de **classe fille**. En Python, l'héritage peut être multiple lorsqu'une classe fille hérite de **plusieurs classes mères**.

POO

Héritage

code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)

File Edit Format Run Options Window Help

```
class Mere:
    def hello(self):
        return "hello"

class Classe_Fille(Mere):
    def hi(self):
        return "hi"

fille1=Classe_Fille()
print(fille1.hi())
print(fille1.hello())
```

IDLE Shell 3.10.5

File Edit Shell Debug Options Window Help

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====
==
hi
hello
>>> |
```

POO

Héritage

```
>>> fille=Classe_Fille()  
>>> isinstance(fille,Classe_Fille)  
True  
>>> isinstance(fille,Mere)  
True  
>>> |
```

POO

Héritage

```
code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)
File Edit Format Run Options Window Help

class Mere:
    def hello(self):
        return "hello"

class Classe_Fille(Mere):
    def hello(self):
        return "hi"

fille1=Classe_Fille()
print(fille1.hello())
```

La redéfinition de méthode

```
IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help

Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bi
t (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/FinKeys Data/Music/codel.py =====
>>>
hi
```


POO

Héritage

code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)

File Edit Format Run Options Window Help

```
class Merel:
    def hello(self):
        return "hello"

class Mere2:
    def hello(self):
        return "hola"

class Classe_Fille(Merel,Mere2):
    def hello(self):
        return "hi"

fille1=Classe_Fille()
print(fille1.hello())
```

IDLE Shell 3.10.5

le Edit Shell Debug Options Window Help

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>
===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====
hi
>> help(Classe_Fille)
Help on class Classe_Fille in module __main__:

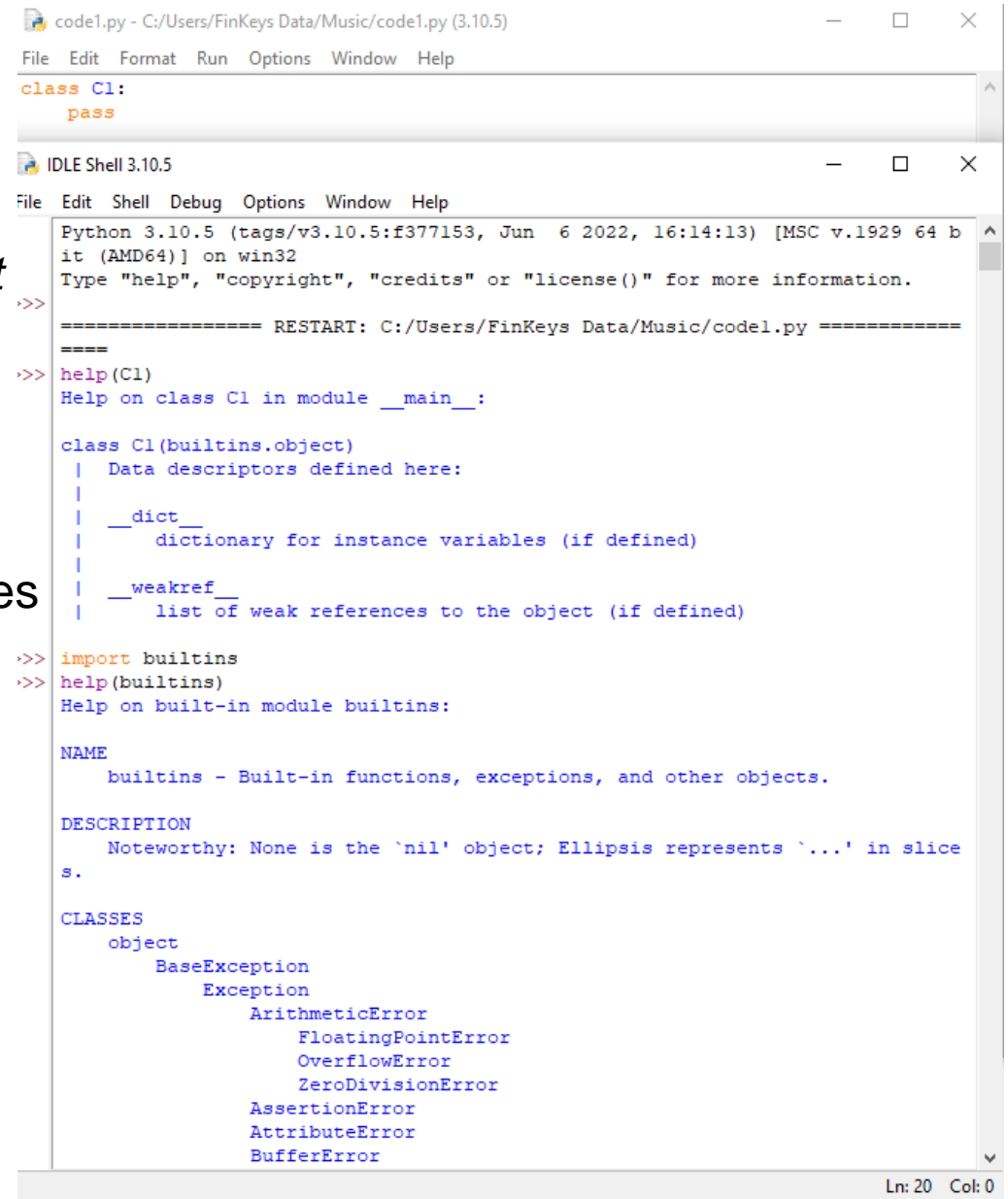
class Classe_Fille(Merel, Mere2)
|   Method resolution order:
|       Classe_Fille
|       Merel
|       Mere2
|       builtins.object
|
|   Methods defined here:
|
|       hello(self)
|
|   -----
|   Data descriptors inherited from Merel:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
>> |
```

POO

Héritage

En Python, il existe une classe interne nommée *object* qui est en quelque sorte la classe ancêtre de tous les objets. Toutes les classes héritent de *object*.

Le module *builtins* possède toutes les fonctions internes à Python.



```
code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)
File Edit Format Run Options Window Help

class C1:
    pass

IDLE Shell 3.10.5
File Edit Shell Debug Options Window Help

Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/FinKeys Data/Music/code1.py =====
>>> help(C1)
Help on class C1 in module __main__:

class C1(builtins.object)
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
>>> import builtins
>>> help(builtins)
Help on built-in module builtins:

NAME
    builtins - Built-in functions, exceptions, and other objects.

DESCRIPTION
    Noteworthy: None is the 'nil' object; Ellipsis represents '...' in slice
    s.

CLASSES
    object
        BaseException
            Exception
                ArithmeticError
                    FloatingPointError
                    OverflowError
                    ZeroDivisionError
                AssertionError
                AttributeError
                BufferError
```

POO

Héritage : Getters et Setters

code1.py - C:/Users/FinKeys Data/Music/code1.py (3.10.5)

File Edit Format Run Options Window Help

```
class Etudiant:
    def __init__(self, nom, age):
        self.nom=nom
        self.age=age

    def get_nom(self):
        return self.nom

    def set_nom(self, value):
        self.nom=value

    def get_age(self):
        return self.age

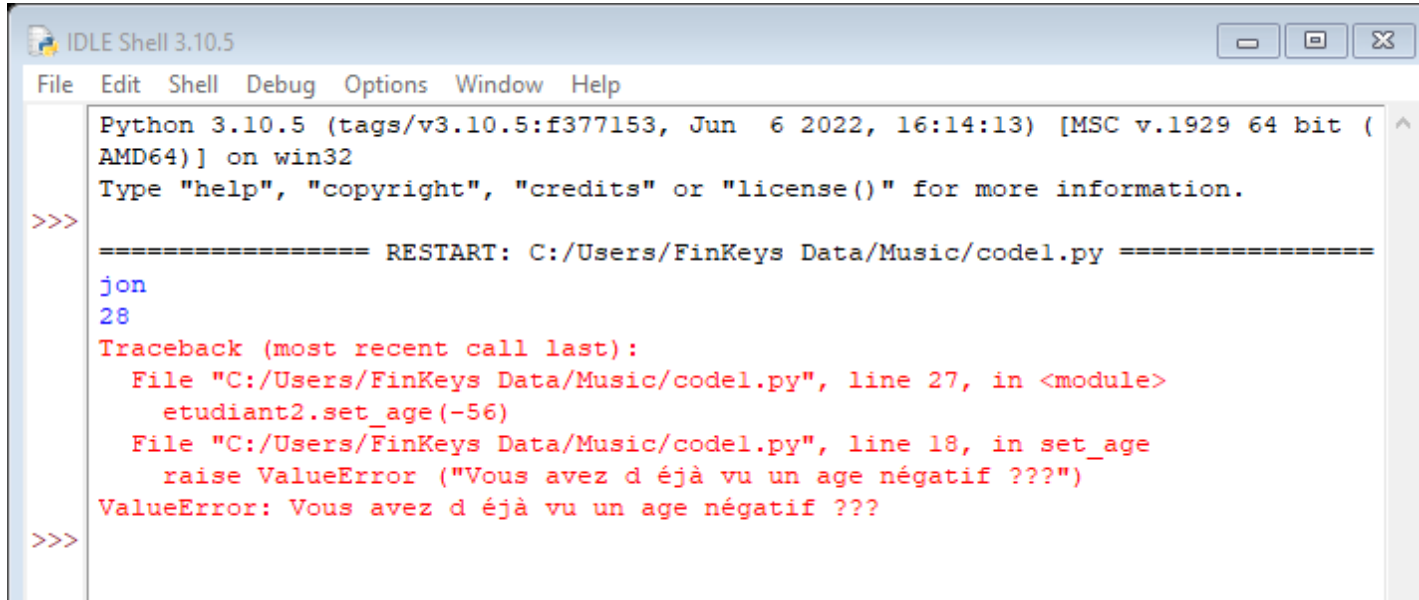
    def set_age(self, value):
        if value<0:
            raise ValueError ("Vous avez d éjà vu un age négatif ???")

        self.age=value

etudiant2=Etudiant('jon',28)
print(etudiant2.get_nom())
print(etudiant2.get_age())
etudiant2.set_age(-56)
```

POO

Héritage : Getters et Setters

A screenshot of the IDLE Shell 3.10.5 window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following content:

```
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
===== RESTART: C:/Users/FinKeys Data/Music/codel.py =====
jon
28
Traceback (most recent call last):
  File "C:/Users/FinKeys Data/Music/codel.py", line 27, in <module>
    etudiant2.set_age(-56)
  File "C:/Users/FinKeys Data/Music/codel.py", line 18, in set_age
    raise ValueError ("Vous avez d éjà vu un age négatif ???")
ValueError: Vous avez d éjà vu un age négatif ???

>>>
```

POO: Récapitulatif

Interfaces

Une interface est une forme particulière de classe où **toutes les méthodes sont abstraites**.

Lorsqu'une classe **implémente** une interface, elle indique ainsi qu'elle **s'engage** à fournir une implémentation (c'est-à-dire un corps) pour chacune des méthodes abstraites de cette interface.

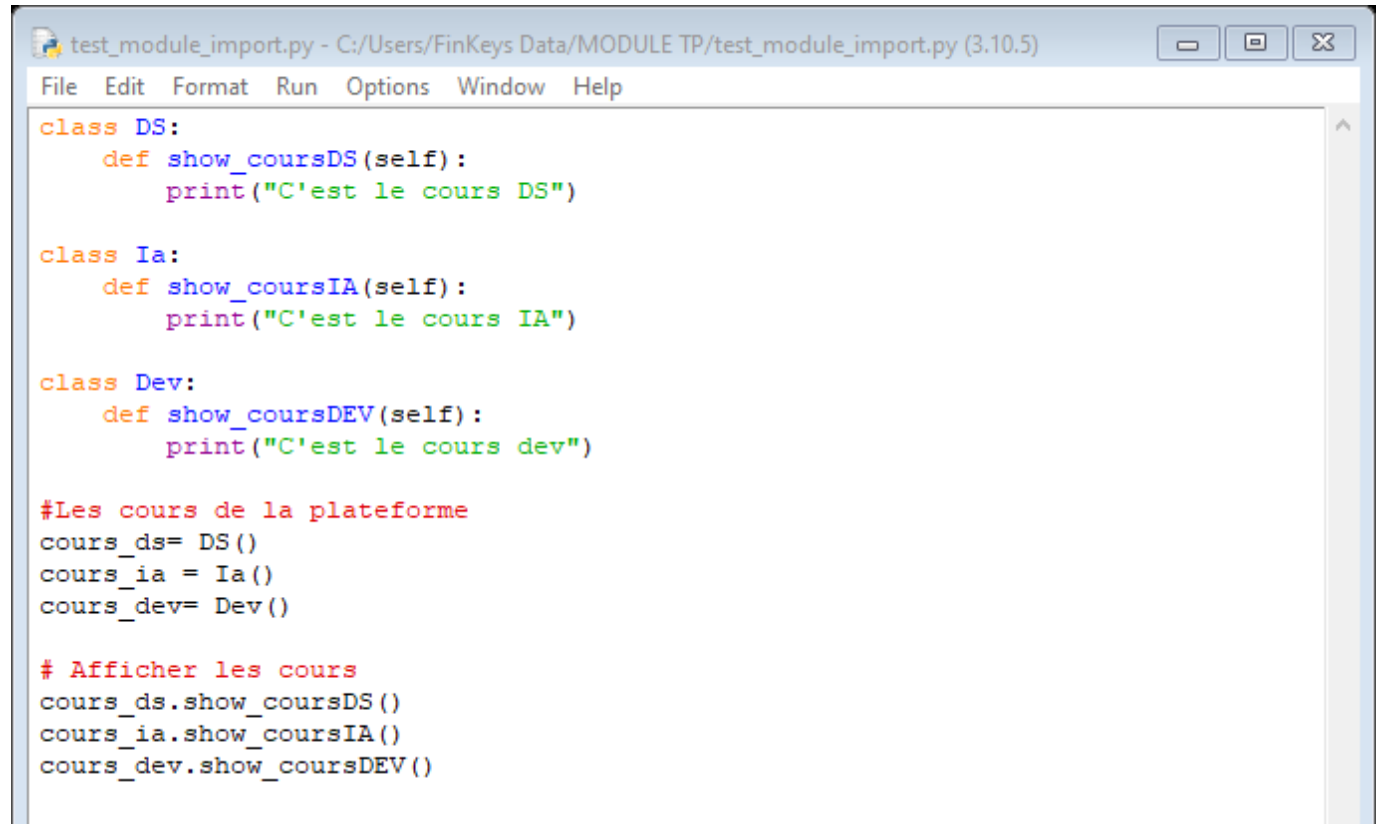
Si une classe **implémente** plus d'une interface, elle doit implémenter **toutes les méthodes abstraites de chacune des interfaces**.

POO: Récapitulatif

Interfaces

```
>>> C'est le cours DS  
>>> C'est le cours IA  
>>> C'est le cours dev
```

les méthodes qui font la même chose doivent avoir le même nom.



```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)  
File Edit Format Run Options Window Help  
  
class DS:  
    def show_coursDS(self):  
        print("C'est le cours DS")  
  
class Ia:  
    def show_coursIA(self):  
        print("C'est le cours IA")  
  
class Dev:  
    def show_coursDEV(self):  
        print("C'est le cours dev")  
  
#Les cours de la plateforme  
cours_ds= DS()  
cours_ia = Ia()  
cours_dev= Dev()  
  
# Afficher les cours  
cours_ds.show_coursDS()  
cours_ia.show_coursIA()  
cours_dev.show_coursDEV()
```

POO: Récapitulatif

Interfaces

```
>>>
```

```
C'est le cours DS  
C'est le cours IA  
C'est le cours dev
```

```
*test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)*  
File Edit Format Run Options Window Help  
from abc import ABC, abstractmethod  
# abc est un module python intégré, nous importons ABC et abstractmethod  
  
class Cours(ABC): # hériter de ABC (Abstract base class)  
    @abstractmethod # un décorateur pour définir une méthode abstraite  
    def show(self):  
        pass  
  
class DS(Cours):  
    def show(self):  
        print("C'est le cours DS")  
  
class Ia(Cours):  
    def show(self):  
        print("C'est le cours IA")  
  
class Dev(Cours):  
    def show(self):  
        print("C'est le cours dev")  
  
# Les cours de la plateforme  
cours_liste=[DS(), Ia(), Dev()]  
  
for cours in cours_liste:  
    cours.show()
```

POO: Récapitulatif

Interfaces

```
>>> C'est le cours DS,inscrivez vous à maintenant  
C'est le cours IA,inscrivez vous à maintenant  
C'est le cours dev,inscrivez vous à maintenant
```

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

from abc import ABC, abstractmethod
# abc est un module python intégré, nous importons ABC et abstractmethod

class Cours(ABC): # hériter de ABC(Abstract base class)
    @abstractmethod # un décorateur pour définir une méthode abstraite
    def show(self,action):
        pass

class DS(Cours):
    def show(self,action,time):
        print(f"C'est le cours DS,{action} vous à {time}")

class Ia(Cours):
    def show(self,action,time):
        print(f"C'est le cours IA,{action} vous à {time}")

class Dev(Cours):
    def show(self,action,time):
        print(f"C'est le cours dev,{action} vous à {time}")

#Les cours de la plateforme
cours_liste=[DS(),Ia(),Dev()]

for cours in cours_liste:
    cours.show(action="inscrivez", time="maintenant")
```


POO: Récapitulatif

Interfaces

```
>>> ont le meme age  
      meme longueur
```

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

from abc import abstractmethod

class Comparable:
    @abstractmethod
    def CompareTo():
        pass
# les classes qui implémentent l'interface :
class Personne(Comparable):
    def __init__(self,nom,age):
        self.__nom=nom
        self.__age=age
    def CompareTo(self,p):
        if self.__age==p.__age:
            return True
        else:
            return False
class Outils(Comparable):
    def __init__(self, long, prix):
        self.__longueur=long
        self.__prix=prix
    def CompareTo(self,o):
        if self.__longueur==o.__longueur:
            return True
        else:
            return False
#objets
p1=Personne("n1",20)
p2=Personne("n2",20)
o1=Outils(60,450)
o2=Outils(60,450)
if (p1.CompareTo(p2)):
    print("ont le meme age")
else:
    print("ages différents")
if (o1.CompareTo(o2)):
    print("meme longueur")
else:
    print("longueurs différents")
```

Ln: 27 Col: 20

Design Pattern: Factory Method

La méthode Factory est un modèle de **conception créative** qui permet à une interface ou à une **classe de créer un objet**, mais laisse les sous-classes décider de la classe ou de l'objet à instancier.

En utilisant la **méthode Factory**, nous disposons des **meilleurs moyens de créer un objet**.

POO: Récapitulatif

Design Pattern: Factory Method



Imaginez que vous ayez votre propre startup qui propose des **services de covoiturage** dans **différentes régions du pays**. La version initiale de l'application ne propose que **le covoiturage de deux roues**, mais au fil du temps, votre application devient populaire et vous voulez maintenant ajouter **le covoiturage de trois et quatre roues**.

Les développeurs doivent **modifier l'ensemble du code** car la majeure partie du code est désormais **couplée à la classe des deux roues** et les développeurs doivent modifier l'ensemble du code.

Après avoir effectué tous ces changements, les développeurs finissent soit avec un code **désordonné**.

POO

Design Pattern: Factory Method

```
>>> voiture  
car  
coche  
bicyclette  
bike  
bicicleta  
cyclette  
cycle  
ciclo
```

Un exemple d'une plateforme de traduction de trois langues : français, espagnol et anglais.

```
*test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)*  
File Edit Format Run Options Window Help  
  
def __init__(self):  
    self.translations = {"car": "voiture", "bike": "bicyclette",  
                        "cycle": "cyclette"}  
  
def localize(self, msg):  
    return self.translations.get(msg, msg)  
  
class SpanishLocalizer:  
  
    def __init__(self):  
        self.translations = {"car": "coche", "bike": "bicicleta",  
                            "cycle": "ciclo"}  
  
    def localize(self, msg):  
        return self.translations.get(msg, msg)  
  
class EnglishLocalizer:  
  
    def localize(self, msg):  
        return msg  
  
if __name__ == "__main__":  
  
    f = FrenchLocalizer()  
    e = EnglishLocalizer()  
    s = SpanishLocalizer()  
  
    message = ["car", "bike", "cycle"]  
  
    for msg in message:  
        print(f.localize(msg))  
        print(e.localize(msg))  
        print(s.localize(msg))  
  
Ln: 34 Col: 1
```

POO: Récapitulatif

Design Pattern: Factory Method

La solution consiste à remplacer les **appels directs à la construction d'objets** par des appels à la méthode spéciale **Factory Method**.

En fait, il n'y aura aucune différence dans la création des objets, mais **ils sont appelés au sein de la méthode factory**.

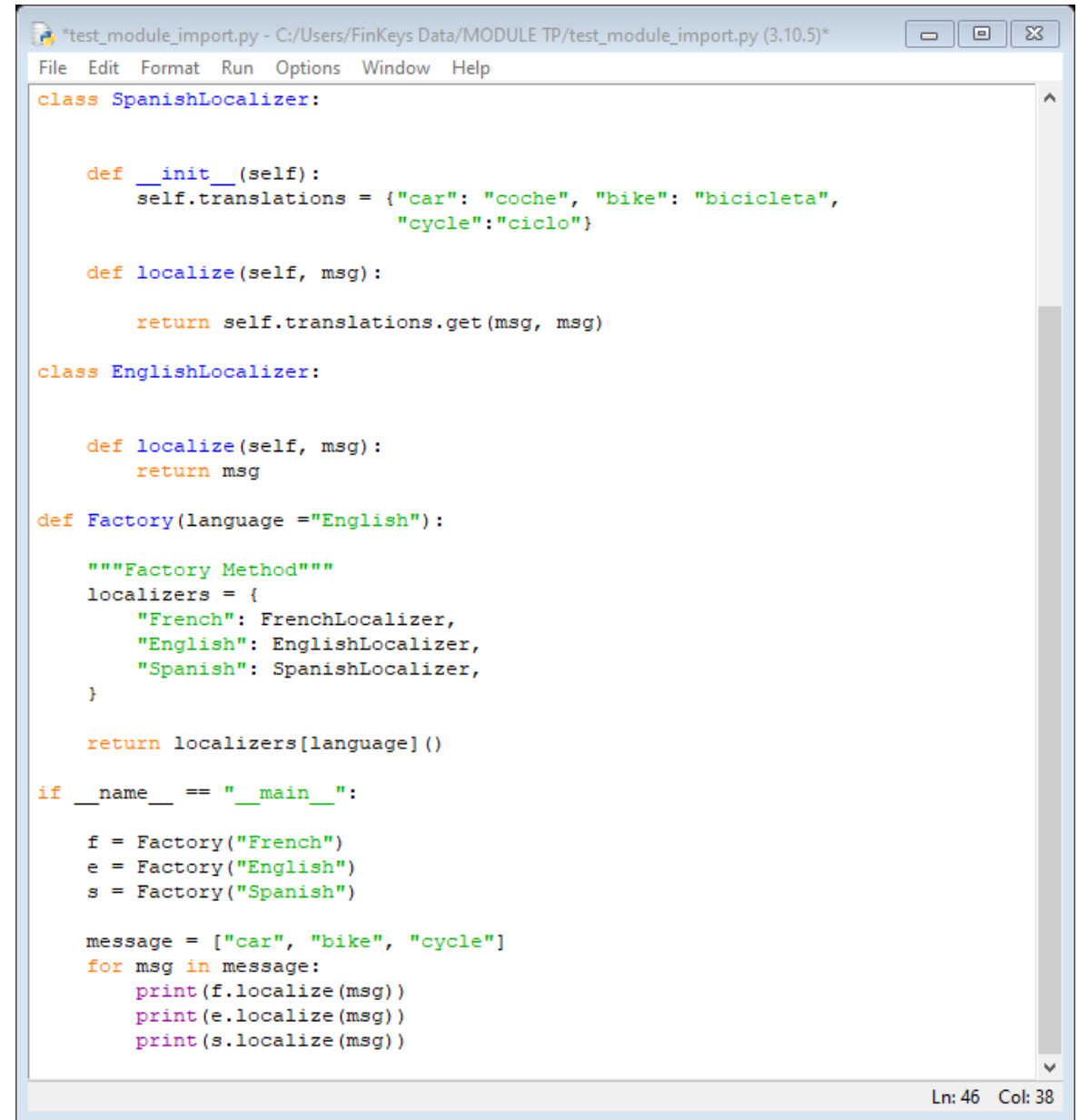
Par exemple, nos classes **Two_Wheeler**, **Three_Wheeler** et **Four_wheeler** devraient implémenter l'interface **ridesharing** qui déclarera une méthode appelée **ride**.

Chaque classe implémentera cette méthode de manière unique.

POO

Design Pattern: Factory Method

```
voiture  
car  
coche  
bicyclette  
bike  
bicicleta  
cyclette  
cycle  
ciclo  
>>>
```

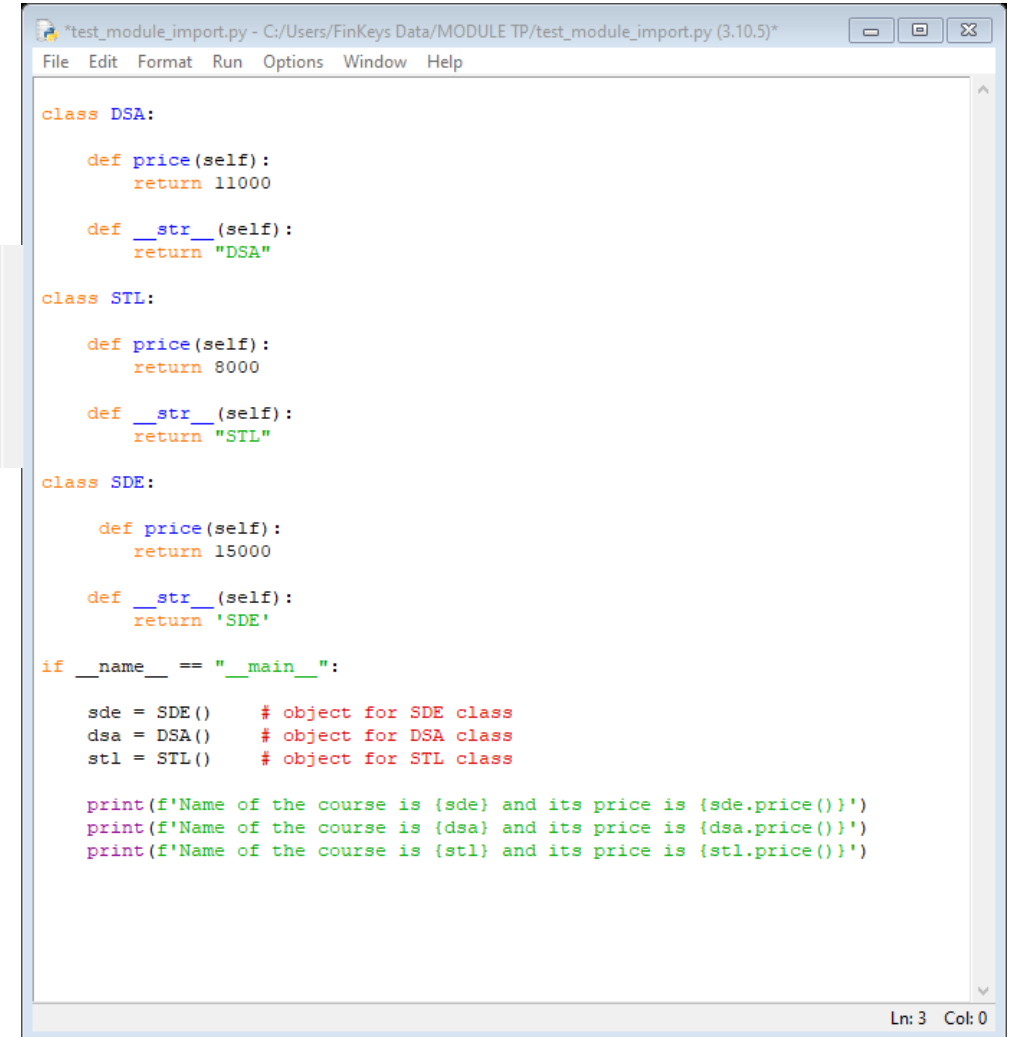


```
*test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)*  
File Edit Format Run Options Window Help  
class SpanishLocalizer:  
  
    def __init__(self):  
        self.translations = {"car": "coche", "bike": "bicicleta",  
                             "cycle": "ciclo"}  
  
    def localize(self, msg):  
  
        return self.translations.get(msg, msg)  
  
class EnglishLocalizer:  
  
    def localize(self, msg):  
        return msg  
  
def Factory(language = "English"):  
  
    """Factory Method"""  
    localizers = {  
        "French": FrenchLocalizer,  
        "English": EnglishLocalizer,  
        "Spanish": SpanishLocalizer,  
    }  
  
    return localizers[language]()  
  
if __name__ == "__main__":  
  
    f = Factory("French")  
    e = Factory("English")  
    s = Factory("Spanish")  
  
    message = ["car", "bike", "cycle"]  
    for msg in message:  
        print(f.localize(msg))  
        print(e.localize(msg))  
        print(s.localize(msg))  
  
Ln: 46 Col: 38
```

POO

Design Pattern: Abstract Factory Method

```
>>>
===== RESTART: C:/Users/FinKeys Data/MODULE TP/test_module_import.py =====
Name of the course is SDE and its price is 15000
Name of the course is DSA and its price is 11000
Name of the course is STL and its price is 8000
>>>
```



```
*test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

class DSA:

    def price(self):
        return 11000

    def __str__(self):
        return "DSA"

class STL:

    def price(self):
        return 8000

    def __str__(self):
        return "STL"

class SDE:

    def price(self):
        return 15000

    def __str__(self):
        return 'SDE'

if __name__ == "__main__":

    sde = SDE()      # object for SDE class
    dsa = DSA()      # object for DSA class
    stl = STL()      # object for STL class

    print(f'Name of the course is {sde} and its price is {sde.price()}')
    print(f'Name of the course is {dsa} and its price is {dsa.price()}')
    print(f'Name of the course is {stl} and its price is {stl.price()}')

Ln: 3 Col: 0
```

POO

Design Pattern: Abstract Factory Method

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

def Fee(self):
    return 11000

def __str__(self):
    return "DSA"

class STL:

    """Class for Standard Template Library"""

    def Fee(self):
        return 8000

    def __str__(self):
        return "STL"

class SDE:

    """Class for Software Development Engineer"""

    def Fee(self):
        return 15000

    def __str__(self):
        return 'SDE'

def random_course():

    """A random class for choosing the course"""

    return random.choice([SDE, STL, DSA])

if __name__ == "__main__":

    course = Course_At_GFG(random_course)

    for i in range(5):
        course.show_course()

Ln: 1 Col: 0
```

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

import random

class Course_At_GFG:

    """ GeeksforGeeks portal for courses """

    def __init__(self, courses_factory = None):
        """course factory is out abstract factory"""

        self.course_factory = courses_factory

    def show_course(self):

        """creates and shows courses using the abstract factory"""

        course = self.course_factory()

        print(f'We have a course named {course}')
        print(f'its price is {course.Fee()}')

class DSA:

    """Class for Data Structure and Algorithms"""

    def Fee(self):
        return 11000

    def __str__(self):
        return "DSA"

class STL:

    """Class for Standard Template Library"""

    def Fee(self):
        return 8000

    def __str__(self):
        return "STL"

Ln: 1 Col: 0
```


POO

Association entre les classes

L'association est l'un des concepts les plus importants de la programmation orientée objet. Il est difficile de développer un logiciel bien structuré sans avoir **une bonne maîtrise de ces concepts. Elle définit les règles de communication entre les classes.**

C'est **une relation entre deux classes** et **cette relation est établie à travers leurs objets.** Chaque **objet a son propre cycle de vie** et il n'y a pas **d'objet propriétaire.** Il s'agit d'un type de **relation faible.** Elle peut être de type **one-to-one, one-to-many, many-to-one ou many-to-many.**

Par exemple, **étudiants et enseignants**, les deux classes sont associées l'une à l'autre. Les objets de chaque classe ont **leur propre cycle de vie et il n'y a pas de propriétaire.**

POO

Agrégation entre les classes

C'est une association **unidirectionnelle**. Lorsqu'un objet peut accéder à un autre objet, cette relation est appelée agrégation. Les objets peuvent exister indépendamment dans cette relation.

Nous pouvons la définir de **manière plus concise** : l'agrégation est le fait qu'un objet d'une classe **puisse posséder ou accéder à un objet d'une autre classe**.

Prenons l'exemple d'un étudiant et d'un département. Il y a deux classes, l'une est un étudiant et l'autre est un département. Les étudiants se voient attribuer un numéro d'enregistrement en fonction de leur département spécifique.

POO

Agrégation entre les classes

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

class Etudiant:

    def __init__(self, id):
        self._id = id

    def enregistrement_num(self, department_id) -> str:
        return str(self._id) + '-' + department_id

class Department:

    def __init__(self, id, etudiant):
        self._id = id
        self._etudiant = etudiant

    def etudiant_enregistrement(self):
        return self._etudiant.enregistrement_num(self._id)

if __name__ == '__main__':
    etudiant = Etudiant(10)
    department = Department('ENG', etudiant)
    print(department.etudiant_enregistrement())
```

POO

Composition entre les classes

Elle définit un type de **relation forte**. Elle peut être définie comme le cas où une classe peut référencer un ou plusieurs objets d'une autre classe dans sa variable d'instance. Dans la composition, les objets **ne peuvent pas exister indépendamment**.

Nous pouvons mettre en œuvre l'exemple ci-dessus des étudiants et des départements d'une manière de composition. Dans cet exemple, l'objet étudiant est instancié dans l'objet département. Chaque fois qu'un objet département est détruit, un objet étudiant est automatiquement détruit.

POO

Composition entre les classes

```
test_module_import.py - C:/Users/FinKeys Data/MODULE TP/test_module_import.py (3.10.5)
File Edit Format Run Options Window Help

class Student:

    def __init__(self, id):
        self._id = id

    def registration_number(self, department_id) -> str:
        return str(self._id) + '-' + department_id

class Department:

    def __init__(self, department_id, student_id):
        self._id = department_id
        self._student = Student(student_id)

    def student_registration(self):
        return self._student.registration_number(self._id)

if __name__ == '__main__':
    department = Department('ENG', 10)
    print(department.student_registration())
```

POO: Récapitulatif

Association	Agrégation	Composition
La relation d'association est indiquée par une flèche.	La relation d'agrégation est indiquée par une ligne droite avec une pointe de flèche vide à une extrémité.	La relation de composition est indiquée par une ligne droite avec une flèche remplie à l'une de ses extrémités.
Une association peut exister entre deux ou plusieurs classes en UML.	L'agrégation fait partie d'une relation d'association.	La composition fait partie d'une relation d'association.
Il peut y avoir une association unique, une association multiple, une association multiple et une association multiple entre les classes d'association.	L'agrégation est considérée comme un type d'association faible.	La composition est considérée comme un type d'association forte.

POO: Récapitulatif

Association	Agrégation	Composition
Dans une relation d'association, un ou plusieurs objets peuvent être associés les uns aux autres.	Dans une relation d'agrégation, les objets qui sont associés les uns aux autres peuvent rester dans la portée d'un système sans être associés les uns aux autres.	Dans une relation de composition, les objets qui sont associés les uns aux autres ne peuvent pas rester dans la portée les uns sans les autres.
Les objets sont liés à l'un l'autre.	Les objets liés ne sont pas dépendants sur l'autre objet.	Les objets sont très dépendants de l'un l'autre.
Dans l'association UML, la suppression d'un élément peut ou non affecter un autre élément associé.	Dans l'agrégation UML, la suppression d'un élément n'affecte pas un autre élément associé.	Dans la composition UML, la suppression d'un élément affecte un autre élément associé.

POO: Récapitulatif

Exemples

Association	Agrégation	Composition
Un enseignant est associé à plusieurs élèves. Ou un enseignant donne des instructions aux élèves.	Une voiture a besoin d'une roue, mais elle ne nécessite pas toujours la même roue. Une voiture peut aussi fonctionner correctement avec une autre roue.	Un fichier est placé à l'intérieur du dossier. Si l'on supprime le dossier, le fichier associé à ce dossier est également supprimé.