Classes et Métaclasses en Python

Classes et Métaclasses en Python

« La Programmation Orientée Objet, c'est bien joli, mais ça fait passer cinq fois plus de temps à réfléchir qu'à coder ! »

« En Python, tout est objets : nombres, chaînes, listes, tuples, dictionnaires, fonctions, modules. Et tout est aussi instance d'une classe»

note liminaire : le texte qui suit est relatif à **Python 3**.

* pour passer à **Python 2**, la différence essentielle est le format du **`print` (parenthèses)** et l'utilisation de l'unicode (mettre un **u** devant les chaînes avec caractères non ASCII)

* lorsque des différences plus substantielles ont été introduites en **Python 3**, on le signale explicitement.

* si on prend bien soin de **faire toujours hériter les classes**, il n'y a pas de différence entre les deux versions ; sauf que **Python 3** admet une classe parent par défaut (**`object`**) si on ne précise pas et que **Python 2** demande la spécification de la superclasse pour fonctionner correctement.

1. quelques préalables

1.1. un script

Voici un script qui servira de base pour les scripts ultérieurs. On y trouvera les formats de recherche classiques d'un attribut (valeur ou méthode) dans une classe ou une instance.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test1-appel.py

class Test_Classe (object) :
    varClasse = "varClasse : je suis une variable de classe"
    def __init__ (self, p1, p2) :
        object.__init__(self)
        def fnLocale () :
            print ( "\n"+"#"*30)
            print ( "Je suis une fonction locale")
            print ( "#"*30, "\n")

        self.nom = p1.upper()
        self.code = p2
        self.methodeLocale = fnLocale

    def display (self) :
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)

# création de l'instance
instanceTest = Test_Classe("Joséphine", 17.37)

# méthodes d'affichage
instanceTest.display()
instanceTest.methodeLocale()
print ()
# tests d'appel
print ("valeur de instanceTest.nom : ", instanceTest.nom)
print ("valeur de instanceTest.varClasse :", instanceTest.varClasse)
print ()
# appel depuis la classe
print ("valeur de Test_Classe.varClasse :", Test_Classe.varClasse)
# deux appels non classiques
Test_Classe.display (instanceTest)
instanceTest.__class__.display (instanceTest)
```

```
 nom : JOSÉPHINE ●  code : 17.37

##############################
Je suis une fonction locale
##############################
```

```
valeur de instanceTest.nom :  JOSÉPHINE
valeur de instanceTest.varClasse : varClasse : je suis une variable de classe

valeur de Test_Classe.varClasse : varClasse : je suis une variable de classe
▌ nom : JOSÉPHINE ●  code : 17.37
▌ nom : JOSÉPHINE ●  code : 17.37
```

1.2. dictionnaire

Tout objet a un attribut dictionnaire (**__dict__**) qui comporte les attributs propres de l'objet (pas ceux dont il hérite, mais en incluant les objets redéfinis) sous la forma **clef/valeur** ; la recherche d'un attribut se fait dans le ou les dictionnaires relatifs à l'objet qui utilise cet attribut. (pour savoir si un attribut appartient à un objet ou ses ascendants, utiliser la fonction pré-définie **hasattr()**. On trouvera ci-dessous une fonction globale qui affiche le dictionnaire d'un objet. On notera que **items()** retourne un énumérable (une vue), pas une liste comme dans **Python 2**.

exemple portant sur le script qui précède

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test2-dict.py

def fnDico (obj, msg) :
    print ( "@"*30)
    print ( ">>> ", msg)
    listeKeys = []
    maxlg = 0
    for clef in obj.__dict__.keys() :
        listeKeys.append (clef)
        if len(str(clef)) > maxlg : maxlg = len(str(clef))
    listeKeys.sort()
    for clef in listeKeys :
        print (("   "+str(clef)+" "*20)[:maxlg+4],"\u279e",obj.__dict__[clef])
    print ( "@"*30, "\n")

class Test_Classe (object) :
    varClasse = "varClasse : je suis une variable de classe"
    def __init__ (self, p1, p2, pa1="négoce", pa2="62") :
        object.__init__(self)
        def fnLocale () :
            print ( "\n"+"#"*30)
            print ( "Je suis une fonction locale")
            print ( "#"*30, "\n")

        self.nom = p1.upper()
        self.code = p2
        self.methodeLocale = fnLocale

    def display (self) :
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)

# création de l'instance
instanceTest = Test_Classe("Joséphine", 17.37)
# test de hasattr()
print ('hasattr (Test_Classe, "__name__ >>>"',hasattr (Test_Classe, "__name__"))

# dictionaires :
fnDico (instanceTest, "dict de l'instance")
fnDico (Test_Classe , "dict de la classe Test_Classe")
fnDico (object, "dict de l'objet racine object")
```

```
hasattr (Test_Classe, "__class__ >>>" True
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
>>>  dict de l'instance
   code           → 17.37
   methodeLocale  → <function Test_Classe.__init__.<locals>.fnLocale at 0x7f46cb7fd200>
   nom            → JOSÉPHINE
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
>>>  dict de la classe Test_Classe
```

```
   __dict__      → <attribute '__dict__' of 'Test_Classe' objects>
   __doc__       → None
   __init__      → <function Test_Classe.__init__ at 0x7f46cb7fd560>
   __module__    → __main__
   __weakref__   → <attribute '__weakref__' of 'Test_Classe' objects>
   display       → <function Test_Classe.display at 0x7f46cb809950>
   varClasse     → varClasse : je suis une variable de classe
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
>>>  dict de l'objet racine object
   __class__          → <attribute '__class__' of 'object' objects>
   __delattr__        → <slot wrapper '__delattr__' of 'object' objects>
   __dir__            → <method '__dir__' of 'object' objects>
   __doc__            → The most base type
   __eq__             → <slot wrapper '__eq__' of 'object' objects>
   __format__         → <method '__format__' of 'object' objects>
   __ge__             → <slot wrapper '__ge__' of 'object' objects>
   __getattribute__   → <slot wrapper '__getattribute__' of 'object' objects>
   __gt__             → <slot wrapper '__gt__' of 'object' objects>
   __hash__           → <slot wrapper '__hash__' of 'object' objects>
   __init__           → <slot wrapper '__init__' of 'object' objects>
   __le__             → <slot wrapper '__le__' of 'object' objects>
   __lt__             → <slot wrapper '__lt__' of 'object' objects>
   __ne__             → <slot wrapper '__ne__' of 'object' objects>
   __new__            → <built-in method __new__ of type object at 0x8e51a0>
   __reduce__         → <method '__reduce__' of 'object' objects>
   __reduce_ex__      → <method '__reduce_ex__' of 'object' objects>
   __repr__           → <slot wrapper '__repr__' of 'object' objects>
   __setattr__        → <slot wrapper '__setattr__' of 'object' objects>
   __sizeof__         → <method '__sizeof__' of 'object' objects>
   __str__            → <slot wrapper '__str__' of 'object' objects>
   __subclasshook__   → <method '__subclasshook__' of 'object' objects>
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

1.3. classe d'un objet

On a affirmé en exergue que «**tout objet est instance d'une classe**». Du point de vue de l'héritage, comme toute classe hérite de près ou de loin de la classe **`object`**, on doit retrouver dans cette classe le moyen commun aux objets d'accéder à leur classe : c'est l'attribut **`__class__`**

On peut compléter le script précédent :

```
# ********************** classes ****************************
print ("********* classes *********")
print ("instanceTest.__class__", instanceTest.__class__)
print ("instanceTest.__class__.__class__", instanceTest.__class__.__class__)
print ("Test_Classe.__class__", Test_Classe.__class__)
print ("object.__class__", object.__class__)
print ("type.__class__" ,type.__class__)
fnDico (type, "la classe type")
print ("**************************")
```

```
********* classes *********
instanceTest.__class__ <class '__main__.Test_Classe'>
instanceTest.__class__.__class__ <class 'type'>
Test_Classe.__class__ <class 'type'>
object.__class__ <class 'type'>
type.__class__ <class 'type'>
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
>>>  la classe type
   __abstractmethods__  → <attribute '__abstractmethods__' of 'type' objects>
   __base__             → <member '__base__' of 'type' objects>
   __bases__            → <attribute '__bases__' of 'type' objects>
   __basicsize__        → <member '__basicsize__' of 'type' objects>
   __call__             → <slot wrapper '__call__' of 'type' objects>
   __delattr__          → <slot wrapper '__delattr__' of 'type' objects>
   __dict__             → <attribute '__dict__' of 'type' objects>
   __dictoffset__       → <member '__dictoffset__' of 'type' objects>
   __dir__              → <method '__dir__' of 'type' objects>
   __doc__              → <attribute '__doc__' of 'type' objects>
   __flags__            → <member '__flags__' of 'type' objects>
   __getattribute__     → <slot wrapper '__getattribute__' of 'type' objects>
   __init__             → <slot wrapper '__init__' of 'type' objects>
   __instancecheck__    → <method '__instancecheck__' of 'type' objects>
```

```
   __itemsize__          → <member '__itemsize__' of 'type' objects>
   __module__            → <attribute '__module__' of 'type' objects>
   __mro__               → <member '__mro__' of 'type' objects>
   __name__              → <attribute '__name__' of 'type' objects>
   __new__               → <built-in method __new__ of type object at 0x8e54e0>
   __prepare__           → <method '__prepare__' of 'type' objects>
   __qualname__          → <attribute '__qualname__' of 'type' objects>
   __repr__              → <slot wrapper '__repr__' of 'type' objects>
   __setattr__           → <slot wrapper '__setattr__' of 'type' objects>
   __sizeof__            → <method '__sizeof__' of 'type' objects>
   __subclasscheck__     → <method '__subclasscheck__' of 'type' objects>
   __subclasses__        → <method '__subclasses__' of 'type' objects>
   __weakrefoffset__     → <member '__weakrefoffset__' of 'type' objects>
   mro                   → <method 'mro' of 'type' objects>
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

On attire l'attention sur les attributs de type redéfinis dans type, comme **__doc__**, **__dict__** et quelques autres attributs qui serviront dans la suite : **__base__**, **__bases__**, **__name__**, **__subclasses__**

Attention à ne pas confondre classe et héritage ! La classe d'une classe s'appelle sa **métaclasse** (**metaclass**)

Ici, il n'apparaît qu'une seule **métaclasse** qui s'appelle **type**. Elle est sa propre **métaclasse** ! Cette **métaclasse** **type** n'en hérite pas moins de **object**.

1.4. cas d'un héritage

On va définir la classe **Test_Classe** comme dérivée d'une classe antérieurement définie, **Super_Classe**

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test3-super.py

def fnDico (obj, msg) :
    print ( "@"*30)
    print ( ">>> ", msg)
    listeKeys = []
    maxlg = 0
    for clef in obj.__dict__.keys() :
        listeKeys.append (clef)
        if len(str(clef)) > maxlg : maxlg = len(str(clef))
    listeKeys.sort()
    for clef in listeKeys :
        print (("   "+str(clef)+" "*20)[:maxlg+4],"\u279e",obj.__dict__[clef])
    print ( "@"*30, "\n")

class Super_Classe (object) :
    varSuper = "varSuper : je suis une variable de classe"
    def __init__ (self) :
        self.titre = "test de super classe"

class Test_Classe (Super_Classe) :
    varClasse = "varClasse : je suis une variable de classe"
    def __init__ (self, p1, p2) :
        Super_Classe.__init__(self)
        def fnLocale () :
            print ( "\n"+"#"*30)
            print ( "Je suis une fonction locale")
            print ( "#"*30, "\n")

        self.nom = p1.upper()
        self.code = p2
        self.methodeLocale = fnLocale

    def display (self) :
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)

# création de l'instance
instanceTest = Test_Classe("Joséphine", 17.37)
instanceTest.display()
print ("instanceTest.varSuper >>> ", instanceTest.varSuper)
print ("instanceTest.titre >>> ", instanceTest.titre)
print ()
# dictionnaire
fnDico (instanceTest, "instanceTest")
```

```
# classes
print ("Super_Classe.__class__ >>> ", Super_Classe.__class__)
print ("Test_Classe.__name__>>> ", Test_Classe.__name__)
print ("Test_Classe.__base__ >>> ", Test_Classe.__base__)
print()
print (dir (instanceTest))
print (dir(Test_Classe))
```

```
 nom : JOSÉPHINE ●  code : 17.37
instanceTest.varSuper >>>  varSuper : je suis une variable de classe
instanceTest.titre >>>  test de super classe

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
>>>  instanceTest
   code           → 17.37
   methodeLocale  → <function Test_Classe.__init__.<locals>.fnLocale at 0x7f2018c1b950>
   nom            → JOSÉPHINE
   titre          → test de super classe
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

Super_Classe.__class__ >>>  <class 'type'>
Test_Classe.__name__>>>  Test_Classe
Test_Classe.__base__ >>>  <class '__main__.Super_Classe'>

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'code', 'display',
'methodeLocale', 'nom', 'titre', 'varClasse', 'varSuper']

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

1.5. résolution d'un attribut

Résoudre un attribut, c'est retrouver l'objet où il a été défini (redéfini) et ainsi pourvoir l'utiliser.

Étant donné un attribut, on dispose pour le résoudre en premier lieu du **dictionnaire associé à l'objet** dont il est l'attribut : le premier niveau de recherche est ce dictionnaire. Si l'objet est une classe, on enchaîne les dictionnaires de ses ascendants (super classe, super super classe etc).

Si le premier niveau est insuffisant, **on passe au dictionnaire de sa classe**. Comme en Python tout est objet, le dictionnaire de **`object`** est disponible, et donc l'accès à la classe (par **`__class__`**). On procède de même en enchaînant les dictionnaires des classes hiérarchiquement supérieures. Si on est amené à la racine, **`object`**, il y a échec et génération d'une erreur.

La fonction prédéfinie **`hasattr()`** procède ainsi. La fonction **`dir()`** retourne tous les attributs accessibles d'un objets, les attributs propres et ceux hérités. Si on veut disposer des attributs accessibbles par une classe, il faut aussi interroger sa métaclasse.

2. les divers genres de méthodes

2.1. définitions

Une méthode définie dans une classe peut être de l'un des trois genres suivants, qui, chacun ont un mode d'appel particulier :

*** méthode d'instance**

Leur mode d'appel est le mode par défaut ; elle est toujours déclarée avec au moins un paramètre usuellement appelée **self** (mais qui peut être nommé comme on veut. Ce mode est toujours appliqué, en l'absence de déclaration contraire. On a vu au paragraphe 1.1 un exemple de ce genre, avec les modes d'appel, soit à partir d'une instance, soit à partir de la classe.

*** méthode statique**

On peut déclarer qu'une méthode est statique : une telle méthode est appelée depuis une instance ou la classe comme si c'était une fonction «ordinaire» : les paramètres formels et les paramètres réels se correspondent exactement.

*** méthode de classe**

C'est la moins utilisée de trois genres de méthodes ; lors de la déclaration, le premier paramètre appelé usuellement **cls** (mais là aussi, c'est indifférent) et ce paramètre prend la valeur de la classe, que la méthode soit appelée depuis la classe ou depuis une instance.

Il existe plusieurs façons de déclarer le genre de la méthode ; la plus sure est d'utiliser les fonctions prédéfinies **`staticmethod()`** et **`classmethod()`**. Il n'y a pas d'indication spéciale pour les méthodes d'instance. On évitera les commodités apportées en **Python 3**, comme de considérer comme statique une fonction sans paramètre. L'autre procédé est d'utiliser les décorateurs (voir un exemple au paragraphe 2.3).

2.2. le script d'illustration

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test4-methodes.py

def fnGlobale (par1 = "", par2 = "") :
    print ("je suis une fonction redéfinie en méthode")
    print (par1)
    print (par2)
    print ("**********************\n")


class Test_Classe (object) :
    varClasse = "varClasse : je suis une variable de classe"
    def __init__ (self, p1, p2) :
        object.__init__(self)
        def fnLocale () :
            print ( "\n"+"#"*30)
            print ( "Je suis une fonction locale")
            print ( "#"*30, "\n")

        self.nom = p1.upper()
        self.code = p2
        self.methodeLocale = fnLocale

    def display (self, p1 = "") :
        print ("paramètres de display : ", p1)
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)

    # ceci est une méthode d'instance
    methodeInstanceFnGlobale = fnGlobale
    # ceci est une méthode statique
    methodeStatiqueFnGlobale = staticmethod(fnGlobale)
    # ceci est une méthode de classe
    methodeClasseFnGlobale = classmethod(fnGlobale)

# création de l'instance
instanceTest = Test_Classe("Joséphine", 17.37)

# méthodes d'affichage
instanceTest.display("appelée par instanceTest")
instanceTest.methodeLocale()
fnGlobale ("appelée globalement", "avec deux paramètres")

# autres méthodes definies dans la classe
# -------- méthode d'instance !
instanceTest.methodeInstanceFnGlobale(" appel depuis une instance, avec un paramètres")
Test_Classe.methodeInstanceFnGlobale(instanceTest,
           "appel depuis la classe, avec (deux) paramètres")
# -------- spécifique Python 3
Test_Classe.methodeInstanceFnGlobale ("appel depuis la classe", "deux vrais paramètres")
# -------- méthode statique !
instanceTest.methodeStatiqueFnGlobale("static : appel depuis une instance,",
                                      "avec deux paramètres")
Test_Classe.methodeStatiqueFnGlobale("static : appel depuis la classe,",
                                     "avec deux paramètres")
# ------- méthode de classe
instanceTest.methodeClasseFnGlobale("class : appel depuis une instance (un paramètre)")
Test_Classe.methodeClasseFnGlobale("class : appel depuis la classe (un paramètre) ")
```

```
paramètres de display :  appelée par instanceTest
█ nom : JOSÉPHINE ●  code : 17.37
```

```
##############################
Je suis une fonction locale
##############################

je suis une fonction redéfinie en méthode
appelée globalement
avec deux paramètres
**********

je suis une fonction redéfinie en méthode
<__main__.Test_Classe object at 0x7f73c6543f90>
 appel depuis une instance, avec un paramètres
**********

je suis une fonction redéfinie en méthode
<__main__.Test_Classe object at 0x7f73c6543f90>
appel depuis la classe, avec (deux) paramètres
**********

je suis une fonction redéfinie en méthode
appel depuis la classe
deux vrais paramètres
**********

je suis une fonction redéfinie en méthode
static : appel depuis une instance,
avec deux paramètres
**********

je suis une fonction redéfinie en méthode
static : appel depuis la classe,
avec deux paramètres
**********

je suis une fonction redéfinie en méthode
<class '__main__.Test_Classe'>
class : appel depuis une instance (un paramètre)
**********

je suis une fonction redéfinie en méthode
<class '__main__.Test_Classe'>
class : appel depuis la classe (un paramètre)
**********
```

attention Python2 / Python3

```
# -------- spécifique Python 3
Test_Classe.methodeInstanceFnGlobale ("appel depuis la classe", "deux vrais paramètres")
```

La ligne précédente provoque une erreur en **Python2** ; en **Python3**, lorsque la méthode d'instance **`methodeInstanceFnGlobale()`** est appelée par une classe, à la différence de Python 2, aucun contrôle n'est fait sur le premier paramètre et elle se comporte comme une méthode **statique**.

2.3 les décorateurs

décorateurs en Python

Les décorateur en Python constituent un framework qui permet d'envelopper les classes et les fonctions pour modifier leur comportement par défaut. On notifie un décorateur en commençants la ligne qui précède celle qui comporte le **`class`** ou le **`def`** par **`@`** suivi du nom du décorateur et en terminant la ligne sur cette appellation. Les deux décorateurs de méthode sont **`@classmethod`** et **`@staticmethod`**. C'est une alternative à l'utilisation des fonctions prédéfinies **`classmethod`** et **`staticmethod`**.

le script d'illustration

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test5-decorateurs.py
```

```
class Test_Classe (object) :
    varClasse = "varClasse : je suis une variable de classe"
    def __init__ (self, p1, p2) :
        object.__init__(self)
        def fnLocale (par="") :
            print ( "\n"+"#"*30)
            print (par, "Je suis une fonction locale")
            print ( "#"*30, "\n")

        self.nom = p1.upper()
        self.code = p2
        self.methodeLocale = fnLocale

    def display (self, p1 = "") :
        print ("paramètres de display : ", p1)
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)

    # ceci est une méthode statique
    @staticmethod
    def methodeStatique (p1, p2) :
        print ("\n+++++++ méthode statique")
        print (p1, "\n", p2)
    """ @cstaticmethod au lieu de methodeStatique = staticmethod (methodeStatique)
    """
    # ceci est une méthode de classe
    @classmethod
    def methodeDeClasse (cls, p1) :
        print ("\n+++++++ méthode de classe")
        print ("la classe en premier paramètre : ",cls.__name__)
        print (p1)
    """ @classmethod au lieu de  methodeDeClasse = classmethod (methodeDeClasse)
    """

# création de l'instance
instanceTest = Test_Classe("Joséphine", 17.37)

# méthodes avec affichage
instanceTest.display("méthode appelée par instanceTest")
instanceTest.methodeLocale("fonction définie dans __init__")

# autres méthodes definies dans la classe

# -------- méthode statique
instanceTest.methodeStatique("static : appel depuis une instance,",
                                    "avec deux paramètres")
Test_Classe.methodeStatique("static : appel depuis la classe,",
                                   "avec deux paramètres")
# ------- méthode de classe
instanceTest.methodeDeClasse("class : appel depuis une instance (un paramètre)")
Test_Classe.methodeDeClasse("class : appel depuis la classe (un paramètre) ")
```

```
paramètres de display :  méthode appelée par instanceTest
 nom : JOSÉPHINE ●  code : 17.37

##############################
fonction définie dans __init__ Je suis une fonction locale
##############################

+++++++ méthode statique
static : appel depuis une instance,
 avec deux paramètres

+++++++ méthode statique
static : appel depuis la classe,
 avec deux paramètres

+++++++ méthode de classe
la classe en premier paramètre :  Test_Classe
class : appel depuis une instance (un paramètre)

+++++++ méthode de classe
la classe en premier paramètre :  Test_Classe
class : appel depuis la classe (un paramètre)
```

3. les classes callables

3.1. définition

Une classe est **`callable`** (appelable) lorsqu'elle contient une méthode d'instance appelée **`__call__()`**.

propriété des classes callables

Soit la classe **`Cl`** callable, et **`instCl`** une instance de cette classe. On suppose que la méthode **`__call__`** est définie dans **`Cl`** sur le modèle suivant (ou un équivalent, cf. section 2) :

```
def __call__ (self, p1, p2,p3 ...)
```

Alors les expressions suivants sont équivalentes :

```
instCl.__call__(p1, p2, p3 ...)
Cl.__call__(instCl, p1, p2, p3 ...)
inst (p1, p2, p3 ...)
```

Dit d'une autre façon, **`instCl`** devient une fonction de paramètres **`p1, p2, p3 ...`**

classes callables déjà rencontrée.

On utilise dans l'instanciation le nom de la classe comme nom d'une fonction. Par exemple :

```
instanceTest = Test_Classe("Joséphine", 17.37)
```

Comment cela peut-il s'opérer, en restant dans la logique élémentaire de Python ? Dans cet appel, **`Test_Classe`** désigne alors une fonction, qui prend des paramètres, retourne une instance initialisée. L'initialisation se fait par un appel à **`__init__()`** en lui passant exactement le bon nombre de paramètres, c'est à dire l'instance, et les paramètres réels définis dans l'appel. Cela implique au préalable que cette fonction **`__call__`** ait créé cette instance.

Mais il ne faut pas oublier que l'on doit faire l'appel sur une instance, **`Test_Classe`**.

On peut en conclure :

1. la métaclasse de **`Test_Classe`** (soir : **`Test_Classe__.__class__`**) comporte une méthode d'instance **`__call__`** (définie ou héritée)
2. cette méthode d'instance, lors de l'appel, doit prendre comme premier paramètre le nom de «l'instance» à créer de **`Test_Classe`**
3. elle doit créer de toutes pièces une telle instance.
4. elle doit faire un appel à **`Test__Classe.__init__()`** en lui passant les bons paramètres (voir les sections 1 et 2), ce qui implique que la signature de l'appel de **`Test_Class()`** doit être conforme à l'attente du **`__init__()`**.
5. elle doit retourner l'instance créée et initialisée de **`Test_Classe`**.

un script d'illustration

Pour vérifier au moins partiellement les 5 propositions qui précèdent, voici un petit script d'illustration.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test6-call.py

class Test_Classe (object) :
    varClasse = "varClasse : je suis une variable de classe"
    def __init__ (self, p1, p2) :
        object.__init__(self)
        self.nom = p1.upper()
        self.code = p2

    def display (self, p1 = "") :
        print ("display : ", p1)
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)
        print ("~"*40, "\n")

# création d'une instance par la méthode traditionnelle
instanceTest = Test_Classe("Joséphine", 13.46)
instanceTest.display("méthode appelée par instanceTest")
```

```
# création d'une instance avec __call__
print ("métaclasse de Test_Classe : ", Test_Classe.__class__)
callTest = Test_Classe.__class__.__call__(Test_Classe, "Marie-Christine", 13.57)
callTest.display("méthode appelée par callTest")

# la mauvaise façon de faire
mauvaisCall = Test_Classe.__call__("Anne-Françoise", 14.16)
mauvaisCall.display("méthode appelée par mauvaisCall")
```

```
display :  méthode appelée par instanceTest
▌ nom : JOSÉPHINE ●  code : 13.46
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

métaclasse de Test_Classe :  <class 'type'>
display :  méthode appelée par callTest
▌ nom : MARIE-CHRISTINE ●  code : 13.57
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

display :  méthode appelée par mauvaisCall
▌ nom : ANNE-FRANÇOISE ●  code : 14.16
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

On peut vérifier que **`type`**, métaclasse de **`Test_Classe`** possède bien une méthode **`__call__`** dans la section sur les dictionnaires (§ 1.3.). Un **slot** est une fonction ; **wrapper** signifie que la fonction enveloppe une autre fonction, non visible, c'est à dire est capable d'exécuter la fonction enveloppée, mais an plus, d'assurer d'autres services.

On peut s'interroger sur la **troisième partie** de l'exemple. Pourquoi le procédé n'est-il pas adapté, même s'il paraît fonctionner ? Si on vérifie **la résolution de l'attribut** **`__call__`** dans **`Test_Classe.__call__()`**, on voit qu'il est appelé à partir de **`Test_Classe`** : c'est une classe et l'attribut et recherché dans cette classe. Ce n'est que parce que la classe **`Test_Classe`** n' a pas cet attribut que Python considère que **`Test_Classe`** est une **instance**, et qu'il faut chercher dans sa **métaclasse**. Cette recherche est alors un succès. Mais si **`Test_Classe`** avait été **`callable`**, le déroulement aurait été tout différent (et fautif)

3.2. création de fonctions

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test7-callable.py

class Test_Classe (object) :
    varClasse = "varClasse : je suis une variable de classe"
    def __init__ (self, p1, p2) :
        self.nom = p1.upper()
        self.code = p2

    def display (self, p1 = "") :
        print ("display : ", p1)
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)
        print ("~"*40, "\n")

    def __call__ (self, p1 ="") :
        print ("display : ", p1)
        print ( "\u2588 nom :", self.nom, "\u2688  code :", self.code)
        print ("~"*40, "\n")

# création d'une instance
instanceTest = Test_Classe("Joséphine", 13.46)
instanceTest.display("méthode display")
instanceTest ("méthode __call__")
```

```
display :  méthode display
▌ nom : JOSÉPHINE ●  code : 13.46
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

display :  méthode __call__
▌ nom : JOSÉPHINE ●  code : 13.46
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

3.3. en complément

On peut légitimement se demander si les fonctions ne sont pas elles aussi les instances d'une classe callable. On

trouvera ci-dessous un test qui permet de réponde par l'affirmative. Python n'offre pas de démarche simple pour aller plus loin (comme c'est le cas en **JavaScript**).

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test8-fonction.py

def fnTest (p1, p2) :
    print ("appel de la fontion fnTest")
    print (p1)
    print (p2)

fnTest ("Jean-François", "04/10/1937")
print ()
fun = fnTest.__class__
print (fun)
print ()
fun.__call__(fnTest, "Marc_Antoine", "12/08/1958")
```

```
appel de la fontion fnTest
Jean-François
04/10/1937

<class 'function'>

appel de la fontion fnTest
Marc_Antoine
12/08/1958
```

4. la création d'instance

4.1. problématique

*** On rappelle la problématique :**

la classe :

```
class Test_Classe (object) :
```

appel de la classe callable

```
instanceTest = Test_Classe ("Marie-Ange", 17.41)
```

équivalence avec __call__

```
instanceTest = Test_Classe.__class__.call__ ("Marie-Ange", 17.41)
```

`Test_Classe.__class__` est la métaclasse de `Test_Classe`

*** création d'une métaclasse**

Par défaut, la métaclasse est `type`. Mais rien n'empêche de la redéfinir. Le procédé est simple : on crée une classe qui hérite de `type` et on déclare que `Test_Classe` a pour métaclasse la classe créée.

La syntaxe n'est pas la même en **Python 2** et en **Python 3**. En **Python 2** fait dans la classe la déclaration :

```
__metaclass__ = la métaclasse
```

En **Python 3** on ajoute aux classes parentes, le paramètre à mot-clef `metaclass`.

* on est alors libre de **rédéfinir** tout ce que l"on veut dans la **métaclasse.**

Ce qui nous intéresse ici, c'est de redéfinir la fonction `__call_` qui retourne les instances de `Test_Classe` et peut donc les moduler à loisir.

Rappelons les hypothèses qui restent à confirmer : cette fonction peut fabriquer et initialiser une instance, en appelant une fonction *ad hoc* de création et la fonction `__init__`.

4.2. la méthode __new__()

Python n'échappe pas aux modalités habituelles des langages objets ; seulement, elles sont bien enveloppées pour créer une programmation fluide et lisible. Pour instancier une classe, on crée tout d'abord une structure en mémoire conforme à la classe : c'est l'opérateur `new` traditionnel. Et on la personnalise ensuite.

Cette fonction `__new__()` est héritée des classes parentes, et *in fine* de la classe `object`. On peut vérifier en regardant le dictionnaire de `object` (§ 1.2 et § 1.3) que celle-ci a bien la méthode comme attribut.

La méthode `__new__` n'est pas une méthode d'instance, et son premier paramètre est une classe, celle où elle est définie. Dans l'exemple, on l'a définie comme méthode de classe ; tout dépend comment on l'utilise dans la

redéfinition de **__call__()** et on peut aussi en faire une méthode statique, ou même une fonction globale ! On verra **ultérieurement** ce qu'il faut déclarer pour le **__call__()** par défaut, celui hérité de **type**.

4.3. le script de référence

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test9-new.py

class Meta_Test_Classe (type) :
    # méthode d'instance de NouvelleMeta ; appel sous la forme :
    # MaClasseTemoin.__class__.__call__(MaClasseTemoin,"Jean", 16.58)
    def __call__ (selfcls, pNom, pCode) :
        print ("\u27a8 trace \u27a8  __call__ (selfcls, pNom, pCode")
        objet = selfcls.__new__()
        selfcls.__init__ (objet, pNom, pCode)
        print ("\u27a8 trace \u27a8  return objet" )
        return objet

class Test_Classe (object, metaclass = Meta_Test_Classe ) :
    # __metaclass =  Meta_Test_Classe en Python 2
    def __new__ (cls) :
        print ("\u27a8 trace \u27a8  __new__ (cls)" )
        return object.__new__(cls)
    __new__ = classmethod (__new__)

    def __init__ (self, parNom, parCode) :
        print ("\u27a8 trace \u27a8  __init__ (self, parNom, parCode")
        self.nom = parNom
        self.code = parCode

    def display (self) :
        print ("\u27a8 trace \u27a8  display (self))")
        print (self.nom, self.code)

instanceTest = Test_Classe ("Marie-Ange", 17.41)
instanceTest.display()
```

```
➸ trace ➸  __call__ (selfcls, pNom, pCode
➸ trace ➸  __new__ (cls)
➸ trace ➸  __init__ (self, parNom, parCode
➸ trace ➸  return objet
➸ trace ➸  display (self))
Marie-Ange 17.41
```

Bien **prendre garde** que **__call__** est une méthode d'instance ! et que comme elle est appelée depuis une instance, (ici, la classe **Test_Classe**) son paramètre usuellement appelé **self**, et ici **selfcls** pour bien montrer la différence et que l'on désigne par **selfcls** la classe instance callable **Test_Classe**.

4.4. la méthode __call__ par défaut

C'est pratiquement toujours la méthode **__call__** de **type** ou celle d'une sous-classe.

la méthode __call__ utilise un __new__ générique

Si on essaie de créer une méthode **__new__()** pour le **__call__()** par défaut, on constate que la signature de **__new__()** doit être compatible avec celle de **__init__()**. Autrement dit, mis à par le **cls/self,** les paramètres formel de **__new__()** sont exactement ceux de **__init__()**, ou compatibles (en énumérés comme en clef/valeur). En pratique, on déclare le **__new__** de la façon suivante :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test10-essai.py

class Test_Classe (object) :

    def __new__ (cls, *l, **kw) :
        print ("\u27a8 trace \u27a8  __new__ (cls, *l, **kw)")
        return object.__new__(cls)

    def __init__ (self, parNom, parCode) :
        print ("\u27a8 trace \u27a8  __init__ (self, parNom, parCode")
```

```
        self.nom = parNom
        self.code = parCode

    def display (self) :
        print ("\u27a8 trace \u27a8  display (self))")
        print (self.nom, self.code)

instanceTest = Test_Classe ("François-Joseph", 11.11)
instanceTest.display()
```

```
➸ trace ➸  __new__ (cls, *l, **kw)
➸ trace ➸  __init__ (self, parNom, parCode
➸ trace ➸  display (self))
François-Joseph 11.11
```

***l** désigne les paramètres énumérés (0, 1 ou plus) et ****kw** les paramètres formels à mots clef qui suivent. Pour le rappeler nous les nommerons plutôt ***ven**, et ****vkw** (*variables énumérées et variables à mots clefs*).

un équivalent à la méthode `__call__` générique

Cet équivalent est fait pour illustrer, et ne saurait recouvrir tout le fonctionnement du **`__call__()`** par défaut, en particulier dans le passage de paramètres ; le mieux est de ne prendre les paramètres du **`__new__()`** et du **`__init__()`** qu'en lecture seule (penser que ce peut être des listes ou des dictionnaires), et sans les modifier, à moins de vérifier précisément ce que l'on fait !

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test11-call.py

class Meta_Test_Classe (type) :
    # méthode d'instance de NouvelleMeta ; appel sous la forme :
    # MaClasseTemoin.__class__.__call__(MaClasseTemoin,"Jean", 16.58)
    def __call__ (selfcls, *ven, **vkw) :
        print ("\u27a8 trace \u27a8  __call__ (selfcls, *ven, **vkw")
        print ("\u27a8 trace \u27a8       \u27a8 ven : ", ven)
        print ("\u27a8 trace \u27a8       \u27a8 vkw : ", vkw)
        selfcls.__new__ = classmethod (selfcls.__new__)
        objet = selfcls.__new__(*ven, **vkw)
        selfcls.__init__ (objet, *ven, **vkw)
        print ("\u27a8 trace \u27a8  return objet" )
        return objet

class Test_Classe (object, metaclass = Meta_Test_Classe ) :
    #__metaclass =  Meta_Test_Classe en Python 2
    def __new__ (cls, *ven, **vkw) :
        print ("\u27a8 trace \u27a8  __new__ (cls, *ven, **vkw)")
        return object.__new__(cls)

    def __init__ (self, parNom, parCode) :
        print ("\u27a8 trace \u27a8  __init__ (self, parNom, parCode)")
        self.nom = parNom
        self.code = parCode

    def display (self) :
        print ("\u27a8 trace \u27a8  display (self))")
        print (self.nom, self.code)

instanceTest = Test_Classe ("Marie-Renée", 11.23)
instanceTest.display()
```

```
➸ trace ➸  __call__ (selfcls, *ven, **vkw
➸ trace ➸       ➸ ven :  ('Marie-Renée', 11.23)
➸ trace ➸       ➸ vkw :  {}
➸ trace ➸  __new__ (cls, *ven, **vkw)
➸ trace ➸  __init__ (self, parNom, parCode)
➸ trace ➸  return objet
➸ trace ➸  display (self))
Marie-Renée 11.23
```

4.5. le singleton avec __new__

Une classe singleton et une classe qui ne peut avoir qu'une seule instance. Lorsqu'une première instance est créée, elle ne peut que retourner la même instance si on l'invoque de nouveau.

Le procédé consiste à conserver dans la classe une référence sur l'instance et de demander à **__new__** de retourner cette instance.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test12-sing-new.py

class Test_Classe_Singleton (object) :
    __refInstance = None
    def __new__ (cls, *ven, **vkw) :
        if cls.__refInstance == None :
            return object.__new__(cls)
        return cls.__refInstance

    def __init__ (self, parNom, parCode) :
        if self.__class__.__refInstance == None :
            self.nom = parNom
            self.code = parCode
            self.__class__.__refInstance = self

    def display (self, p) :
        print (p, self.nom, self.code)

instanceTest = Test_Classe_Singleton ("Michel", 12.03)
instanceTest.display("instanceTest")

autreInstance = Test_Classe_Singleton ("N'importe", 12.14)
autreInstance.display("autreInstance")
print ("instanceTest == autreInstance : ", instanceTest == autreInstance)

del (instanceTest)
autreInstance.display("autreInstance")
```

```
instanceTest Michel 12.03
autreInstance Michel 12.03
instanceTest == autreInstance :  True
autreInstance Michel 12.03
```

Ne pas oublier que **del()** supprime la référence, pas l'objet ! Pour supprimer un objet en Python, il faut supprimer toutes ses références ; dans le cas présent, si une instance est créée, il n'existe pas de moyen simple de la supprimer.

4.6. le singleton avec __call__

Il existe un moyen de réaliser une classe singleton en retouchant la classe *a minima* : en indiquant seulement qu'elle est un singleton. Il suffit de créer une métaclasse avec un **__call__()** générique redéfini. On se trouve donc dans le cas du paragraphe 4.4.

La variable de classe qui enregistre l'instance est créée dynamiquement. Le retour de **__call__()** est modulé selon que l'instance existe déjà ou non.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test11-call.py

class Meta_Classe_Singleton (type) :

    def __call__ (selfcls, *ven, **vkw) :
        if hasattr (selfcls, "instanceBrKhZZhxwVW321") :
            return  selfcls.instanceBrKhZZhxwVW321
        objet = selfcls.__class__.__class__.__call__(selfcls, *ven, **vkw )
        selfcls.instanceBrKhZZhxwVW321 = objet
        return objet

class Test_Classe_Singleton (object, metaclass = Meta_Classe_Singleton ) :

    def __init__ (self, parNom, parCode) :
        self.nom = parNom
        self.code = parCode

    def display (self, p) :
        print (p)
        print (self.nom, self.code)
```

```
instanceTest = Test_Classe_Singleton ("Rose-Marie", 14.08)
instanceTest.display("instanceTst : ")

autreInstance = Test_Classe_Singleton ("N'importe", 12.14)
autreInstance.display("autreInstance : ")
print ("instanceTest == autreInstance : ", instanceTest == autreInstance)
```

```
instanceTst :
Rose-Marie 14.08
autreInstance :
Rose-Marie 14.08
instanceTest == autreInstance :  True
```

* **"instanceBrKhZZhxwVW321"** quelque chose d'improbable !

* **selfcls.__class__** : c'est **Meta_Classe_Singleton**

* **selfcls.__class__.__class__** : ici, c'est **type**

* **selfcls.__class__.__class__.__call__** : le **__call__** de la classe mère

5. créer une classe à partir d'une métaclasse

5.1. objectif

Puisqu'une classe est l'instance d'une métaclasse, on peut créer directement une classe directement à partir d'une métaclasse, et pas seulement en créant la classe et la métaclasse et en faisant le lien après coup. Nous avons utilisé cette méthode indirecte parce que nous voulions éviter d'avoir à définir toute la métaclasse, mais seulement à en contrôler des parties en redéfinissant le **__call__** par exemple.

5.2. un exercice d'école

Il convient d'être clair et au préalable, nous proposons un petit script qui permet de bien différencier ce qui est métaclasse et ce qui est héritage :

* on rappelle que la métaclasse se retrouve avec l'attribut **__class__** ; l'attribut **__class__** hérité de **object** s'applique à tous les objets, même si ce ne sont pas de classes.

* l'attribut **__base__** dans l'héritage simple permet de récupérer la classe parent.

* l'attribut **__bases__** dans l'héritage multiple permet de récupérer les classes parents sous forme d'un tuple. L'attribut **__base__** est le premier élément de ce tuple.

* l'attribut **__name__** permet de retrouver sous forme d'une chaîne le nom de la classe.

le script

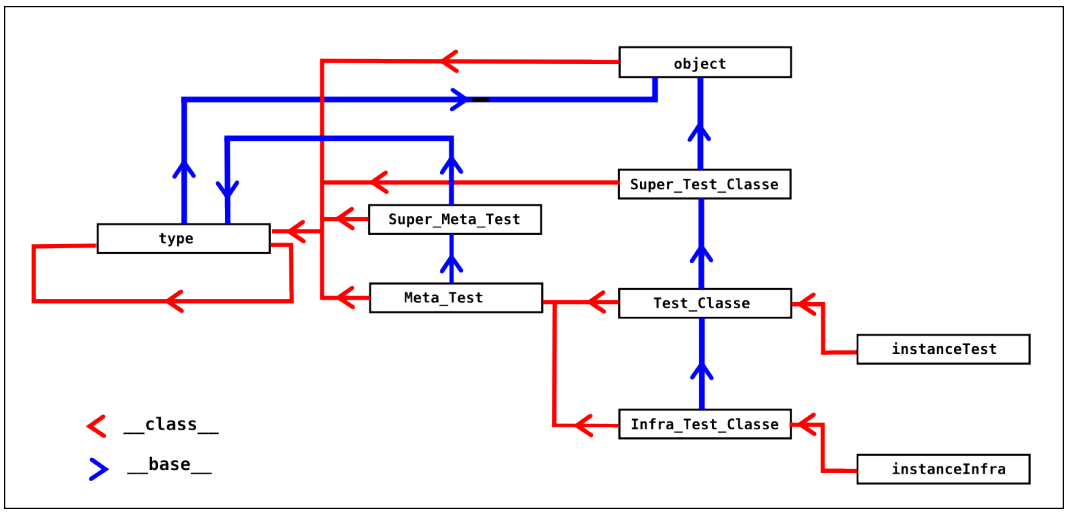
```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test14-ecole.py

def fnDico (obj, msg) :
    print ( "@"*30)
    print ( ">>> ", msg)
    listeKeys = []
    maxlg = 0
    for clef in obj.__dict__.keys() :
        listeKeys.append (clef)
        if len(str(clef)) > maxlg : maxlg = len(str(clef))
    listeKeys.sort()
    for clef in listeKeys :
        print (("   "+str(clef)+" "*20)[:maxlg+4],"\u279e",obj.__dict__[clef])
    print ( "@"*30, "\n")

class Super_Meta_Test (type) :
    titre = "je suis la métaclasse Super_Meta_Test"

class Meta_Test (Super_Meta_Test) :
    titre = "je suis la métaclasse Meta_Test"

class Super_Test_Classe (object) :
    titre = "je suis la classe parent Super_Test_Class"
    def __init__ (self, paramNation) :
        self.nation = paramNation
```

```
    def displaySuper (self) :
        print ("> nationalité :", self.nation)

class Test_Classe (Super_Test_Classe, metaclass = Meta_Test) :
    titre = "je suis la classe centrale TestClasse"
    def __init__ (self, parNom, parCode, parNation) :
        Super_Test_Classe.__init__(self, parNation)
        self.nom = parNom
        self.code = parCode

    def display (self) :
        self.displaySuper()
        print (">> nom : ",self.nom, " //  code : ", self.code)

class Infra_Test_Classe (Test_Classe) :
    titre = "je suis la classe enfant de Test_Classe"
    def __init__ (self, pNom, pCode, pNation, pSigne) :
        Test_Classe.__init__ (self, pNom, pCode, pNation)
        self.signe = pSigne
    def displayInfra (self) :
        self.display()
        print (">>> signalement " ,self.signe)

# test sur la hiérarchie des classes
instanceTest = Test_Classe ("Louis_Philippe", 17.34, "français")
instanceTest.display()
print ("~"*10)
instanceInfra = Infra_Test_Classe ("Richard", 18.38, "irlandais", "cheveux roux")
instanceInfra.displayInfra ()
print ("*"*30)
# dictionnaire
fnDico (instanceInfra, "dictionnaire de instanceInfra")

# test avec __base__
print ("object_.__base__ >>>",object.__base__)
print ("Super_Test_Classe.__base__ >>>",Super_Test_Classe.__base__)
print ("Test_Classe.__base__ >>>",Test_Classe.__base__)
print ("Infra_Test_Classe.__base__ >>>",Infra_Test_Classe.__base__)
print ("~"*10)
print ("Meta_Test.__base__ >>>",Meta_Test.__base__)
print ("Super_Meta_Test.__base__ >>>",Super_Meta_Test.__base__)
print ("~"*10)
print ("type.__base__ >>>",type.__base__)
print ("*"*30)

# test avec __classe__
print ("object_.__class__ >>>",object.__class__)
print ("Super_Test_Classe.__class__ >>>",Super_Test_Classe.__class__)
print ("Test_Classe.__class__ >>>",Test_Classe.__class__)
print ("Infra_Test_Classe.__class__ >>>",Infra_Test_Classe.__class__)
print("~"*10)
print ("Meta_Test.__class__ >>>",Meta_Test.__class__)
print ("Super_Meta_Test.__class__ >>>",Super_Meta_Test.__class__)
print ("~"*10)
print ("type.__class__ >>>",type.__class__)
```

```
> nationalité : français
>> nom :  Louis_Philippe  //  code :  17.34
~~~~~~~~~~
> nationalité : irlandais
>> nom :  Richard  //  code :  18.38
>>> signalement  cheveux roux
******************************
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
>>>  dictionnaire de instanceInfra
   code    → 18.38
   nation  → irlandais
   nom     → Richard
   signe   → cheveux roux
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

object_.__base__ >>> None
Super_Test_Classe.__base__ >>> <class 'object'>
Test_Classe.__base__ >>> <class '__main__.Super_Test_Classe'>
Infra_Test_Classe.__base__ >>> <class '__main__.Test_Classe'>
```

```
~~~~~~~~~~
Meta_Test.__base__ >>> <class '__main__.Super_Meta_Test'>
Super_Meta_Test.__base__ >>> <class 'type'>
~~~~~~~~~~
type.__base__ >>> <class 'object'>
**************************
object_.__class__ >>> <class 'type'>
Super_Test_Classe.__class__ >>> <class 'type'>
Test_Classe.__class__ >>> <class '__main__.Meta_Test'>
Infra_Test_Classe.__class__ >>> <class '__main__.Meta_Test'>
~~~~~~~~~~
Meta_Test.__class__ >>> <class 'type'>
Super_Meta_Test.__class__ >>> <class 'type'>
~~~~~~~~~~
type.__class__ >>> <class 'type'>
```

schéma résumé



5.3. l'objet `type`

Comme on l'a déjà vu, la création d'une instance met en jeu trois objets :

* l'objet créé comme instance,

* la classe de l'objet créé, où sont définis le **`__new__()`** et le **`__init__()`**. Ces méthodes sont héritées de **`object`**, et peuvent avoir été redéfinies. Lorsque la méthode **`__new__()`** ou **`__init__()`** n'est pas explicitée dans la classe, le nom de la méthode est **résolu en remontant dans la hiérarchie des classes**. La création d'un objet revient de toute façon à la méthode **`__new__()`**, lui donnant au minimum les attributs apportés par **`object`**. (voir les paragraphe 1.2. et 1.3. sur les dictionnaires)

* la métaclasse, par l'appel de son **`__call__()`** à partir de la classe, assure l'appel des méthodes **`__new__()`** et **`__init__()`** et leur paramétrage *ad hoc*. **La métaclasse est toujours un descendant de la méthode `type`. C'est ce qui la caractérise.** Par défaut, la métaclasse d'une classe est **`type`**, et c'est même le cas pour **`type`** lui-même !

  Ceci a pour conséquence que si on définit les fonctions **`__new__()`** et **`__init__()`** dans une métaclasse, ce sont les fonctions héritées de **`type`** et bien entendu, de **`object`**. Or **`type`** redéfinit bien ces méthodes comme on peut le constater au § 1.3. Noter que **`__call__()`** étant défini dans **`type`**, il n'y a pas à se poser la question de redéfinition dans le système. Et cette méthode transmet, par défaut sa liste de paramètres au méthodes **`__new__()`** et **`__init__()`** du niveau «inférieur» ; en clair, il faut appeler **`__call__()`** avec les paramètres attendus de **`__new__()`** et **`__init__()`**

Quels sont ces paramètres dans le cas où on utilise le **`__call__`** d'un ascendant d'une métaclasse (**`type`** par défaut) ? Une classe est caractérisée par son nom, les ascendants dont il hérite et ses attributs (usuellement nommés variables de classe et méthodes). Les paramètres sont donnés sous la forme suivante :

* le nom : **`name`** = chaîne de caractères,

* les ascendants : **`bases`** = tuple des ascendants,

* les attributs : **dict** = dictionnaire.

Pour **__new__()**, ne pas oublier que le premier paramètre désigne la classe actuelle et que la méthode est une méthode de classe. Pour **__init__()**, que la premier paramètre est la variable de l'instance qui est créée.

un exemple élémentaire

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test15-type.py

# création d'une classe
Test_Classe = type ("Test_Classe", (object,), {} )

# adjonction d'une variable de classe
Test_Classe.varClasse = "je suis une variable de classe"

# adjonction d'une méthode d'instance
Test_Classe.methodeInst = lambda self, x : "méthode lamba : " + x

# création d'une instance
instanceTest = Test_Classe ()
print ("instanceTest.varClasse : ", instanceTest.varClasse)
print (instanceTest.methodeInst ("appelée par instanceTest"))
```

```
instanceTest.varClasse :  je suis une variable de classe
méthode lamba : appelée par instanceTest
```

* on a l'instance **Test_Classe** et le nom **"Test_Classe"** ; le premier est un identificateur de variable, et qui sert de nom pour une variable de portée globale. Le second est la valeur de **__name__**. Il est de bonne guerre de prendre le même mot.

* La classe est vraiment minimale ! et elle est ensuite enrichie.

5.4. créer une classe à partir d'une métaclasse

objectif

L'exemple qui suit comporte une déclaration de classe **Super_Test_Classe** et une métaclasse **Meta_Test_Classe** dont le but est de donner comme instance **Test_Classe**, une classe similaire à celle de l'exemple précédent.

On y a ajouté une fonction **fnDict()** légèrement différente de **fnDico()** dans la mesure où le premier paramètre est un dictionnaire et non un objet quelconque.

Pour «corser» l'illustration, on a fourni les attributs de la classe créée dans les paramètres d'appel de la métaclasse, dans le __new__ de la métaclasse et dans le **__init__** de la métaclasse. Il faut simplement veiller à donner la bonne structure à l'introduction des données. À chaque niveau d'appel, on fournit l'état du dictionnaire de la classe/instance **Test_Classe**. On a laissé la métaclasse sans **métamétaclasse**, et l'appel **type.__init__()** dans __init__() est inutile mais laissé pour mémoire. On remarquera que ce sont bien les mêmes paramètres que **__call__()** passe au **__new__()** et au **__init__()**

le script

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test16-meta.py

def fnDict (dico, msg) :
    print ( "#"*30)
    print ( ">>> ", msg)
    listeKeys = []
    maxlg = 0
    for clef in dico.keys() :
        listeKeys.append (clef)
        if len(str(clef)) > maxlg : maxlg = len(str(clef))
    listeKeys.sort()
    for clef in listeKeys :
        print (("   "+str(clef)+" "*20)[:maxlg+4],"\u279e",dico[clef])
    print ( "#"*30, "\n")

class Super_Test_Classe (object) :
```

```
    titre = "je suis la classe parent Super_Test_Class"
    def __init__ (self, paramNation) :
        self.nation = paramNation
    def displaySuper (self) :
        print ("> nationalité :", self.nation)
"""
class Test_Classe (Super_Test_Classe) :
    titre = "je suis la classe créée dynamiquement "
    def __init__ (self, parNom, parCode, parNation) :
        Super_Test_Classe.__init__(self, parNation)
        self.nom = parNom
        self.code = parCode

    def display (self) :
        self.displaySuper()
        print (">> nom : ",self.nom, " //  code : ", self.code)

    def getTitre (self) :
        print (self.titre + self.__class__.__name__)
"""
class Meta_Test_Classe (type) :
    def __new__ (selfcls, nom, parents, dico) :
        # locale pour définir le __init__ de la classe
        def initialiser (p0, p1, p2, p3) :
            p0.__class__.__bases__[0].__init__ (p0, p3)
            p0.nom = p1
            p0.code = p2
        # enrichissement du dictionaire
        dico ["__init__"] = initialiser
        fnDict (dico, "exécution du __new__ de la métaclasse")
        # le __new__ de selfcls est laissé par défaut sur type
        return type.__new__ (selfcls, nom, parents, dico)

    def __init__ (selfcls, nom, parents, dico) :
        # locales pour définir les méthodes de la classe
        def afficher (p0) :
            p0.displaySuper()
            print (">> nom : ",p0.nom, " //  code : ", p0.code)

        def donnerTitre (p0) :
            print (p0.titre + p0.__class__.__name__)

        fnDict (dico, "paramètre pour __init__ de la métaclasse")
        # enrichissement du dictionnaire
        type.__init__(selfcls, nom, parents, selfcls.__dict__)
        selfcls.display = afficher
        selfcls.getTitre = donnerTitre
        fnDict (selfcls.__dict__, "exécution du __init__ de la métaclasse")



# faire la classe Test_Classe
Test_Classe = Meta_Test_Classe ("Test_Classe",
                                (Super_Test_Classe,),
                                {"titre" : "je suis la classe créée dynamiquement " }
                               )

# Test de la classe
instanceClasse = Test_Classe ("jean-philippe", 18.22, "espagnol")
fnDict (Test_Classe.__dict__, "instanceClasse")
instanceClasse.display()
instanceClasse.getTitre ()
```

```
##############################
>>>  exécution du __new__ de la métaclasse
   __init__  → <function Meta_Test_Classe.__new__.<locals>.initialiser at 0x7fa90d9d1290>
   titre     → je suis la classe créée dynamiquement
##############################

##############################
>>>  paramètre pour __init__ de la métaclasse
   __init__  → <function Meta_Test_Classe.__new__.<locals>.initialiser at 0x7fa90d9d1290>
   titre     → je suis la classe créée dynamiquement
##############################
```

```
##############################
>>>  exécution du __init__ de la métaclasse
   __doc__      → None
   __init__     → <function Meta_Test_Classe.__new__.<locals>.initialiser at 0x7fa90d9d1290>
   __module__   → __main__
   display      → <function Meta_Test_Classe.__init__.<locals>.afficher at 0x7fa90d99a440>
   getTitre     → <function Meta_Test_Classe.__init__.<locals>.donnerTitre at 0x7fa90d99a4d0>
   titre        → je suis la classe créée dynamiquement
##############################

##############################
>>>  instanceClasse
   __doc__      → None
   __init__     → <function Meta_Test_Classe.__new__.<locals>.initialiser at 0x7fa90d9d1290>
   __module__   → __main__
   display      → <function Meta_Test_Classe.__init__.<locals>.afficher at 0x7fa90d99a440>
   getTitre     → <function Meta_Test_Classe.__init__.<locals>.donnerTitre at 0x7fa90d99a4d0>
   titre        → je suis la classe créée dynamiquement
##############################

> nationalité : espagnol
>> nom :  jean-philippe  //  code :  18.22
je suis la classe créée dynamiquement Test_Classe
>>>
```

5.5. fonctionnement du mot clef `metaclass`

le mécanisme

On a vu précédemment que pour qu'une classe ait une métaclasse imposée, il suffisait de définir la classe (ex : **`Test_Classe`**), une métaclasse, (ex : **`Meta_Test_Classe`**) et de déclarer dans la définition de la classe le mot clef metaclass avec comme valeur la métaclasse :

```
class Test_Class (Super_Test_Classe, metaclass=Meta_Test_Classe) :
```

En **Python 2**, la syntaxe est différente, mais pas le mécanisme.

Lors de la création du script, le mécanisme est le suivant :

1. chaque classe est analysée sous sa forma brute et caractérisée par son nom, ses parents, son dictionnaire.
2. dans le cas de déclaration **`metaclass`**, la métaclasse crée la classe sous sa forme définitive et utilisée dans le script, en utilisant le nom, les parents et le dictionnaire de la classe brute.

Le script qui suit montre que c'est bien la métaclasse qui crée la classe.

le script

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test16-meta.py

def fnDict (dico, msg) :
    print ( "#"*30)
    print ( ">>> ", msg)
    listeKeys = []
    maxlg = 0
    for clef in dico.keys() :
        listeKeys.append (clef)
        if len(str(clef)) > maxlg : maxlg = len(str(clef))
    listeKeys.sort()
    for clef in listeKeys :
        print (("   "+str(clef)+" "*20)[:maxlg+4],"\u279e",dico[clef])
    print ( "#"*30, "\n")

class Super_Test_Classe (object) :
    titre = "je suis la classe parent Super_Test_Class"
    def __init__ (self, paramNation) :
        self.nation = paramNation
    def displaySuper (self) :
        print ("> nationalité :", self.nation)

class Meta_Test_Classe (type) :
    def __new__ (selfcls, nom, parents, dico) :
```

```
        print ("\u2588 Meta_Test_Class.__new__ \u2588")
        print (">>> parent de la classe ",parents[0].__name__)
        fnDict (dico, "dictionnaire passé à la métaclasse")
        return type.__new__ (selfcls, nom, parents, dico)

    def __init__ (selfcls, nom, parents, dico) :
         print ("\u2588 Meta_Test_Class.__init__ \u2588\n")

class Test_Classe (Super_Test_Classe, metaclass=Meta_Test_Classe) :
    titre = "je suis la classe créée dynamiquement "
    def __init__ (self, parNom, parCode, parNation) :
        Super_Test_Classe.__init__(self, parNation)
        self.nom = parNom
        self.code = parCode

    def display (self) :
        self.displaySuper()
        print (">> nom : ",self.nom, " //  code : ", self.code)

    def getTitre (self) :
        print (self.titre + self.__class__.__name__)

# Test de la classe
instanceClasse = Test_Classe ("marc-antoine", 12.17, "romain")
print()
instanceClasse.display()
instanceClasse.getTitre ()
```

```
█ Meta_Test_Class.__new__ █
>>> parent de la classe  Super_Test_Classe
##############################
>>>  dictionnaire passé à la métaclasse
   __init__      → <function Test_Classe.__init__ at 0x7fd21c3ad560>
   __module__    → __main__
   __qualname__  → Test_Classe
   display       → <function Test_Classe.display at 0x7fd21c3ad5f0>
   getTitre      → <function Test_Classe.getTitre at 0x7fd21c3ad680>
   titre         → je suis la classe créée dynamiquement
##############################

█ Meta_Test_Class.__init__ █

> nationalité : romain
>> nom :  marc-antoine  //  code :  12.17
je suis la classe créée dynamiquement Test_Classe
```

6. décorateur

Il existe un framework complet de **Python** sur les **décorateurs**. Son étude n'est pas l'objet de cette sections.

6.1. notion de décorateur

On a vu (section 2) qu'un décorateur est un dispositif qui change le comportement d'une classe ou d'une méthode. Que se passe-t-il si dans le **`__new__()`** de la métaclasse on modifie le dictionnaire ? Le comportement de la classe **`Test_Classe`** s'en trouve altéré : on a affaire à une méthode décorateur. On peut aussi conserver l'interface de la classe, et changer ses méthodes. Un procédé utile consiste à ajouter une fonctionnalité au code de la méthode originelle, tout en l'exécutant normalement ; on a ainsi créé une enveloppe (*wrapper*) à la méthode. On peut aussi changer le comportement de la méthode, et par exemple de transformer une méthode d'instance (par défaut) en méthode de classe...

Voici un **script d'école** pour illustrer l'enveloppement d'une méthode (par défaut «d'instance») en méthode de classe. Ce n'est évidemment pas une méthode réaliste, puisque l'on dispose d'autres moyens pour le faire.

```
#! /usr/bin/python3
# -*- coding: utf-8 -*
# meta-test18-enveloppe.py

class Meta_Test (type) :
    def __init__ (selfcls,nom, bases, dico) :
        remplace = selfcls.__dict__["methodeClasse"]
        print ("~~~~~~~ adresse de la méthode initiale : ", hex(id(remplace)))
        def methode (cls, *ven) :
            remplace(selfcls, *ven)
```

```
        selfcls.methodeClasse = methode
        print ("~~~~~~~ adresse de la méthode wrapper : ", hex(id(methode)))


class Test_Classe (object, metaclass = Meta_Test) :

    titre = "je suis la classe tracée nommée "
    def __init__ (self, parNom, parCode) :
        self.nom = parNom
        self.code = parCode

    def display (self) :
        """ cette méthode display est documentée """
        print (">>> nom : "+self.nom, " //  code : ", self.code)

    def getTitre (self) :
        return ">>>>>> "+ self.titre  + self.__class__.__name__

    def methodeClasse (cls, *ven) :
        print ()
        print ("nom de la classe : " + cls.__name__)
        print ()
        for clef, valeur in cls.__dict__.items() :
            print ((" "*4+clef+" "*20)[:20], valeur)
        print ()
        for p in range(len(ven)) :
            print ("*** paramètre énuméré "+str(p+1) +" : ", ven[p])


instance = Test_Classe ("Marie-Josèphe", 11.04)
instance.display ()
instance.methodeClasse("python", "est un langage réflexif")
```

```
~~~~~~~ adresse de la méthode initiale :  0x7f7f4d9005f0
~~~~~~~ adresse de la méthode wrapper :  0x7f7f4d8e1320
>>> nom : Marie-Josèphe  //  code :  11.04

nom de la classe : Test_Classe

    titre            je suis la classe tracée nommée
    __init__         <function Test_Classe.__init__ at 0x7f7f4d8f68c0>
    __module__       __main__
    display          <function Test_Classe.display at 0x7f7f4d900290>
    methodeClasse    <function Meta_Test.__init__.<locals>.methode at 0x7f7f4d8e1320>
    __doc__          None
    __weakref__      <attribute '__weakref__' of 'Test_Classe' objects>
    getTitre         <function Test_Classe.getTitre at 0x7f7f4d900440>
    __dict__         <attribute '__dict__' of 'Test_Classe' objects>

*** paramètre énuméré 1 :  python
*** paramètre énuméré 2 :  est un langage réflexif
```

6.2. traçage des instances et des méthodes d'une classe

Tracer une classe, c'est donner la possibilité de tenir un journal de l'activité de la classe : création ou suppression d'instance, appel d'une méthode. Le plus simple pour assurer une journalisation consiste à appeler la fonction **`print()`** avec un message ad hoc ; créer un fichier de journalisation ne poserait pas grande difficulté.

On se propose de faire un traçage sur une variante de l'exemple utilisé depuis le début. On crée une métaclasse à la classe **`Test_Classe`**, sans toucher au code explicite de la classe, à l'exception sauf qu'on lui déclare une **métaclasse** nommée **`Meta_Trace_Test_Classe`**-

* Lorsque l'on crée une instance, la méthode **`__new__()`** est appelée ; on a là un lieu d'intervention pour la création d'instance. Comme cette méthode n'est pas explicite, il suffit de demander à la métaclasse d'en ajouter une dynamiquement.

* Lorsque l'on supprime une référence à une instance (fonction **`del()`** par exemple), il ne se passe rien sauf si c'est la dernière référence, autrement dit lorsque l'instance est bonne pour **le ramasse miettes**. C'est ce qui nous intéresse ici. Dans ce cas, la méthode **`__del__`** de la classe de l'instance est appelée.

Les objets sont reconnus de façon **unique** par leur identificateur, un entier obtenu par la fonction **`hash()`**. Pour repérer les instances, c'est le seul procédé disponible.

* Pour tracer les appels de méthodes, on utilise la méthode de l'enveloppe.

En 6.3. on va se contenter de tracer les instances ; en 6.4. et 6.5.on créera un système d'enveloppe pour les méthodes.

6.3. traçage des instances

Les deux méthodes ont été ajoutées dans le **`__init__()`** de la métaclasse. Ce n'est évidemment pas une nécessité d'intervenir à ce niveau. Dans l'exemple suivant on illustrera l'intervention dans le **`__new__()`** de la métaclasse.

On prendra garde que le **`__new__()`** de **`object`** n'a qu'un paramètre.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test19-trace_inst.py

class Meta_Trace_Test_Classe (type) :
    def __new__ (selfcls, nom, parents, dico) :
        return type.__new__ (selfcls, nom, parents, dico)

    def __init__ (selfcls, nom, parents, dico) :
        parent = parents[0]
        def nouveau (p0, p1, p2) :
            inst = parent.__new__(p0)
            print ("\n+++\nl'instance créée a pour id :",hash(inst),"\n+++\n")
            return inst
        def effacer (p0) :
            print ("\n---\nl'instance suivante va disparaître :", hash(p0),"\n---\n")

        selfcls.__new__ = nouveau
        selfcls.__del__ = effacer


class Test_Classe (object, metaclass=Meta_Trace_Test_Classe) :
    titre = "je suis la classe tracée "
    def __init__ (self, parNom, parCode) :
        self.nom = parNom
        self.code = parCode

    def display (self) :
        """ cette méthode display est documentée """
        print (">>> nom : ",self.nom, " //  code : ", self.code)

    def getTitre (self) :
        return ">>>>>> self.titre " + self.__class__.__name__

# Test de la trace de création et de suppresions d'instance
instanceClasse = Test_Classe ("jean_marc", 17.52)
instanceClasse.display()
print (instanceClasse.getTitre ())

aliasInstanceClasse = instanceClasse
autreInstance = Test_Classe ("Françis", 15.53)
autreInstance.display ()
print ("\nsuppression de la référence instanceClasse " )
del (instanceClasse)
print ("suppression de la référence aliasInstanceClasse ")
del (aliasInstanceClasse)
```

```
+++
l'instance créée a pour id : 0x7f0330e29f50
+++

>>> nom :  jean_marc  //  code :  17.52
>>>>>> self.titre Test_Classe

+++
l'instance créée a pour id : 0x7f0330df0150
+++

>>> nom :  Françis  //  code :  15.53

suppression de la référence instanceClasse
suppression de la référence aliasInstanceClasse
```

```
---
l'instance suivante va disparaître : 0x7f0330e29f50
---
```

6.4. traçage d'une méthode par enveloppement

objectif

Ce script est destiné à montrer les difficultés de la modification d'une méthode. On reprend le script précédent, avec comme but de modifier, dans le **__new__()** de la métaclasse la méthode **display()**, de façon à la tracer.

Apparemment, il suffit de remplacer la fonction qui définit la méthode par une autre fonction

- qui d'une part fait le travail de traçage,
- et d'autre part appelle cette fonction qui ne retourne rien, comme si de rien n'était.

le script

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test20-trace_méthode.py

class Meta_Trace_Test_Classe (type) :
    def __new__ (selfcls, nom, parents, dico) :
        # traçage de Test_Classe.display ()
        methode = dico["display"] # la fonction à remplacer
        def fregoli (self) :
            print ("\n§§§\nla méthode : Test_Classe.display a été appelée\n§§§\n")
            methode (self)
        dico.update ( display=fregoli)
        return type.__new__ (selfcls, nom, parents, dico)

    def __init__ (selfcls, nom, parents, dico) :
        parent = parents[0]
        def nouveau (p0, p1, p2) :
            inst = parent.__new__(p0)
            print ("\n+++\nl'instance créée a pour id :",hex(id(inst)),"\n+++\n")
            return inst
        def effacer (p0) :
            print ("\n---\nl'instance suivante va disparaître :", hex(id(p0)),"\n---\n")

        selfcls.__new__ = nouveau
        selfcls.__del__ = effacer


class Test_Classe (object, metaclass=Meta_Trace_Test_Classe) :
    titre = "je suis la classe tracée "
    def __init__ (self, parNom, parCode) :
        self.nom = parNom
        self.code = parCode

    def display (self) :
        """ cette méthode display est documentée """
        print (">>> nom : ",self.nom, " //  code : ", self.code)

    def getTitre (self) :
        return ">>>>>> self.titre " + self.__class__.__name__


# création d'instance
instanceClasse = Test_Classe ("jean_marc", 17.52)

# appel de méthodes
instanceClasse.display()
print (instanceClasse.getTitre ())

# contrôles sur display
print ("\n************** contrôles *****************")
print ("le nom de la méthode display :",Test_Classe.display.__name__)
print ("la doc de la méthode display :",Test_Classe.display.__doc__)
print ("!!!!!!! ouh youh youh !!!!!")

# suppression d'instance
print ()
```

```
print ("suppression de la référence instanceClasse :")
del (instanceClasse)
```

```
+++
l'instance créée a pour id : 0x7fb1da09d210
+++


§§§
la méthode : Test_Classe.display a été appelée
§§§

>>> nom :  jean_marc  //  code :  17.52
>>>>>> self.titre Test_Classe

************** contrôles *****************
le nom de la méthode display : fregoli
la doc de la méthode display : None
!!!!!!! ouh youh youh !!!!!

suppression de la référence instanceClasse :

---
l'instance suivante va disparaître : 0x7fb1da09d210
---
```

* Quoique parfaitement correct et répondant *a minima* au cahier des charges, plusieurs points restent insatisfaisants : la méthode a été correctement enveloppée, mais le nom, et la documentation ne sont pas satisfaisant, et si la méthode originelle avait possédé des attributs propres, leur accès est compromis.

* Un autre problème se serait posé si la méthode avait retourné une valeur, comme la méthode **`getTitre()`**.

* Comme dans la question suivante , ce sont **toutes le méthodes déclarées** que l'on veut envelopper, il va falloir en plus détecter toute les méthodes déclarées parmi les attributs. Une méthode est un objet de type **fonction**. Il y a plusieurs manière de voir si un objet est une fonction :

    - soit en regardant si le type de l'objet est le type fonction ; ce type est un objet du module **`types`** (qu'il faut importer) et se nomme **`FunctionType`**. Il s'agit ici d'**«'être»**, pas **«être égal»** et l'opérateur est **`is`**, même si **`==`** fonctionne !.

    - soit en comparant le type de l'objet au type d'une fonction connue. Par exemple **`lambda : 0`**

On trouvera donc :

```
type (methode) is types.FunctionType
type (methode) is type(lambda : 0)
```

La seconde solution, plus artisanale, évite l'import de **`types`**

6.5. traçage de toutes les méthodes

On a tenu compte des remarques précédentes, et pour structurer correctement, on a créé une fonction d'enveloppement, **`enveloppeur()`** qui prend en paramètres le nom d'une méthode et la vielle valeur et retourne la nouvelle valeur (une fonction), avec les bons attributs (ceux empruntés à la fonction d'origine).

On a créé un nouveau dictionnaire pour éviter de modifier le dictionnaire que l'on balaie !

```
#!/usr/bin/python3
# -*- coding: utf-8 -*
# meta-test21-trace.py

class Meta_Trace_Test_Classe (type) :
    def __new__ (selfcls, nom, parents, dico) :
        def enveloppeur (pClef, pValeurMethode) :
            # définir un transformateur de méthode
            def fregoli (self, *pen, **pkw) :
                # trace de la méthode
                print ("\n§§§\nla méthode : "+pClef+" a été appelée\n§§§\n")
                # exécution de la méthode
                return pValeurMethode (self, *pen, **pkw )
            # parfaire le camouflage du substitut
            fregoli.__name__ = pValeurMethode.__name__
            fregoli.__doc__ = pValeurMethode.__doc__
            fregoli.__dict__.update (pValeurMethode.__dict__)
```

```
            return fregoli

        nouveauDico = {} # le dico de l'enveloppe du Test_Classe
        for clef in dico : # c'est le dico original
            valeur = dico[clef]
            if type(valeur) is type(lambda : 0) :
                nouvelleMethode = enveloppeur (clef, valeur)
                nouveauDico.update ({clef:nouvelleMethode})
            else :
                nouveauDico.update ({clef: valeur})
        return type.__new__ (selfcls, nom, parents, nouveauDico)

    def __init__ (selfcls, nom, parents, dico) :
        parent = parents[0]
        def nouveau (p0, p1, p2) :
            inst = parent.__new__(p0)
            print ("\n+++\nl'instance créée a pour id :",hex(id(inst)),"\n+++\n")
            return inst
        def effacer (p0) :
            print ("\n---\nl'instance suivante va disparaître :", hex(id(p0)),"\n---\n")

        selfcls.__new__ = nouveau
        selfcls.__del__ = effacer

class Test_Classe (object, metaclass=Meta_Trace_Test_Classe) :
    titre = "je suis la classe tracée nommée "
    def __init__ (self, parNom, parCode) :
        self.nom = parNom
        self.code = parCode

    def display (self) :
        """ cette méthode display est documentée """
        print (">>> nom : ",self.nom, " //  code : ", self.code)

    def getTitre (self) :
        return ">>>>>> "+ self.titre  + self.__class__.__name__

# création  d'instances
instanceClasse = Test_Classe ("jean-marc", 17.52)
instanceClasse.display()
print (instanceClasse.getTitre ())

autreInstance = Test_Classe ("antoinette", 11.48)
autreInstance.display()

# contrôles d'atributs
print ("\n************** contrôles *****************")
print ("le nom de la méthode display :",Test_Classe.display.__name__)
print ("la doc de la méthode display :",Test_Classe.display.__doc__)
print ("!!!!!!! hi! hi hi! !!!!!")

# suppression d'instances
aliasInstanceClasse = instanceClasse
print ("\nsuppression de la référence instanceClasse " )
del (instanceClasse)
print ("suppression de la référence aliasInstanceClasse ")
del (aliasInstanceClasse)
```

```
+++
l'instance créée a pour id : 0x7f29d32d00d0
+++


§§§
la méthode : __init__ a été appelée
§§§


§§§
la méthode : display a été appelée
§§§

>>> nom :  jean-marc  //  code :  17.52

§§§
```

```
la méthode : getTitre a été appelée
§§§

>>>>>> je suis la classe tracée nommée Test_Classe

+++
l'instance créée a pour id : 0x7f29d32d0390
+++


§§§
la méthode : __init__ a été appelée
§§§


§§§
la méthode : display a été appelée
§§§

>>> nom :  antoinette  //  code :  11.48

********** contrôles **********
le nom de la méthode display : display
la doc de la méthode display :  cette méthode display est documentée
!!!!!!! hi! hi hi! !!!!!

suppression de la référence instanceClasse :
suppression de la référence aliasInstanceClasse :

---
l'instance suivante va disparaître : 0x7f29d32d00d0
---
```

C'est tout pour aujourd'hui !