

Python

Utilisation avancée

Arnaud Legout
INRIA

Email: arnaud.legout@inria.fr

Organisation de la formation

- Horaires
 - 9h00 – 12h30 et 14h00 – 17h30
 - Pause à 11h00 et à 15h30
 - Repas ensemble
 - Pas d'obligation si vous avez des contraintes

Discussion avant formation

- Quelles sont vos connaissances en programmation ?
 - Connaissances en Python ?
- Quels sont vos besoins ?
 - Pourquoi faire cette formation ?
- Quelles sont vos attentes ?
- Quels sont vos objectifs ?

Méthode pédagogique

- Apprendre par l'exemple et l'expérimentation
 - Prend plus de temps que de donner une liste de règles sans les essayer
 - Mais c'est le seul moyen de comprendre

Plan

- Méthodes statiques et de classe
- Les décorateurs
- Fonction génératrice et conception d'itérateurs
- La gestion avancée des attributs
- La méthode `__new__`
- Les métaclasses

Méthodes statiques et de classe

À quoi ça sert ?

- Les méthodes statiques et de classe peuvent travailler sur les arguments d'une classe sans avoir besoin d'une instance
 - Par exemple, pour compter le nombre d'instances d'une classe

Méthodes unbound et bound

- Une méthode unbound est une méthode appelée sur la classe
 - L'instance n'est pas automatiquement passée comme premier argument
 - C'est un objet fonction classique qui n'a pas besoin d'avoir une instance comme premier argument

```
>>> class C:
    def f(self):
        print(self)

>>> C.f
<function C.f at 0x03270ED0>
```


Méthodes unbound et bound

```
>>> C.f(1)      # on peut passer n'importe quel objet
1
>>> i = C()
>>> C.f(i)      # on peut évidemment passer une instance
<__main__.C object at 0x02BDD090>
```

Méthodes unbound et bound

- Une méthode bound est une méthode appelée sur l'instance
 - L'instance est automatiquement passée comme premier argument de la méthode
 - C'est un objet `bound method`

```
>>> class C:
    def f(self):
        print(self)

>>> i = C()

>>> i.f      # equivalent à C.f(i)

<bound method C.f of <__main__.C object at 0x0327CEF0>>
```

Méthode unbound en Python 2.x

- En Python 2.x
 - Une méthode unbound attend nécessairement comme premier argument une instance de la classe
 - Ça n'est pas une fonction classique comme en Python 3.x, mais un objet `unbound`

Méthode unbound en Python 2.x

```
Python 2.7.9 (default, Dec 10 2014, 12:24:55) [MSC  
v.1500 32 bit (Intel)] on win32  
Type "copyright", "credits" or "license()" for more  
information.
```

```
>>> class C:  
    def f(self):  
        print(self)  
  
>>> C.f  
<unbound method C.f>
```

Méthode unbound en Python 2.x

```
>>> C.f(1)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#9>", line 1, in <module>
```

```
    C.f(1)
```

```
TypeError: unbound method f() must be called with C  
instance as first argument (got int instance instead)
```

```
>>> i = C()
```

```
>>> C.f(i)
```

```
<__main__.C instance at 0x02B35D78>
```

```
>>> C().f
```

```
<bound method C.f of <__main__.C instance at  
0x02B34580>>
```

Comment appeler une méthode sans instance (d'une classe ou d'une instance ?)

- Une méthode appelée sur une instance est `bound`, elle prend automatiquement comme premier argument l'instance (`self`)
- Par contre, une méthode appelée sur une classe est une fonction classique
- Comment appeler une méthode qui travaille sur **les arguments de la classe** indifféremment d'une classe ou d'une instance
 - Par exemple pour compter le nombre d'instances

Cas 1 : méthode sans argument

```
class C:
    numInstances = 0
    def __init__(self):
        C.numInstances = C.numInstances + 1

    def printNumInstances(): #méthode qui ne prend
                           #pas self en argument
        print(f"Nombre d'instances : {C.numInstances}")
```

Cas 1 : méthode sans argument

```
>>> C.printNumInstances() # appelle sur la classe
Nombre d'instances : 0
>>> i = C()
>>> C.printNumInstances()
Nombre d'instances : 1
>>> i.printNumInstances() # mais pas sur l'instance
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#1513>", line 1, in <module>
```

```
    i.printNumInstances()
```

```
TypeError: printNumInstances() takes 0 positional
arguments but 1 was given
```


Cas 2 : méthode avec self comme argument

```
class C:
    numInstances = 0
    def __init__(self):
        C.numInstances += 1
    def printNumInstances(self): #méthode d'instance
        print(f"Nombre d'instances : {C.numInstances}")
```

```
>>> c = C()
>>> c.printNumInstances() # appelle de l'instance
Nombre d'instances : 1
>>> d = C()
>>> c.printNumInstances()
Nombre d'instances : 2
```

Cas 2 : méthode avec self comme argument

```
>>> C.printNumInstances() # mais pas de la classe
Traceback (most recent call last):
  File "<pyshell#1527>", line 1, in <module>
    C.printNumInstances()
TypeError: printNumInstances() missing 1 required positional
argument: 'self'
```

En résumé

- On ne peut pas appeler uniformément la même méthode depuis la classe et l'instance
 - Soit on peut l'appeler de l'instance, mais pas de la classe
 - Soit on peut l'appeler de la classe, mais pas de l'instance

Méthode sans argument en Python 2.x

- En Python 2.x
 - Une méthode appelée sur une classe est `unbound`, elle attend donc, contrairement à Python 3.x, une instance comme premier argument

Méthode sans argument en Python 2.x

Python 2.7.9 (default, Dec 10 2014, 12:24:55) [MSC
v.1500 32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for more
information.

```
>>> class C:
    numInstances = 0
    def __init__(self):
        C.numInstances = C.numInstances + 1

    def printNumInstances(): #méthode qui ne prend
                            #pas self en argument
        print(f"Nombre d'instances : {C.numInstances}")
```

Méthode sans argument en Python 2.x

```
>>> C.printNumInstances()
```

```
Traceback (most recent call last):
```

```
File "<pyshell#15>", line 1, in <module>
```

```
    C.printNumInstances()
```

```
TypeError: unbound method printNumInstances() must be  
called with C instance as first argument (got nothing  
instead)
```

```
>>> i = C()
```

```
>>> i.printNumInstances()
```

```
Traceback (most recent call last):
```

```
File "<pyshell#17>", line 1, in <module>
```

```
    i.printNumInstances()
```

```
TypeError: printNumInstances() takes no arguments (1  
given)
```

En résumé en Python 2.x

- En Python 2.x
 - On ne peut appeler une méthode sans argument ni d'une classe ni d'une instance
 - Une méthode avec argument ne pourra être appelée que d'une instance, mais pas d'une classe

Méthode de module

- Une solution consiste à sortir la méthode travaillant sur les arguments de la classe hors de la classe et d'en faire une méthode de module

Méthode de module

```
def printNumInstances(): #méthode de module
    print(f"Nombre d'instances : {C.numInstances}")
```

```
class C:
    numInstances = 0
    def __init__(self):
        C.numInstances = C.numInstances + 1
```

```
>>> printNumInstances()
Nombre d'instances : 0
>>> a = C()
>>> printNumInstances()
Nombre d'instances : 1
```

Problème avec les méthodes de module

- Le code travaillant sur la classe n'est pas lié à la classe
 - Maintenance difficile
 - Lecture du code difficile
- Pas de possibilité de customisation par héritage

Méthodes statiques et de classe

- Pour appeler une méthode sans instance (d'une classe ou d'une instance), il y a deux possibilités
 - Les **méthodes statiques** ne prennent pas l'instance en premier argument
 - Indépendante de l'instance
 - Créées avec `staticmethod`
 - Les **méthodes de classe** prennent comme premier argument une classe (et non une instance)
 - Indépendante de l'instance
 - Créées avec `classmethod`

Méthodes statiques

```
class C:
    numInstances = 0
    def __init__(self):
        C.numInstances += 1
    def printNumInstances():
        print("Nombre d'instances : " + str(C.numInstances))
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> C.printNumInstances()
```

```
Nombre d'instances : 0
```

```
>>> c = C()
```

```
>>> C.printNumInstances()
```

```
Nombre d'instances : 1
```

```
>>> c.printNumInstances()
```

```
Nombre d'instances : 1
```

Méthodes statiques

- Une méthode statique surchargée dans une sous classe doit être redéfinie comme statique dans la sous classe

```
class SousC(C):  
    def printNumInstances():  
        print("depuis sousC")  
        C.printNumInstances()  
    #printNumInstances = staticmethod(printNumInstances)  
  
>>> i = SousC()
```

Méthodes statiques

```
>>> i.printNumInstances()
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#1537>", line 1, in <module>
```

```
    i.printNumInstances()
```

```
TypeError: printNumInstances() takes 0 positional arguments  
but 1 was given
```

```
>>> SousC.printNumInstances()
```

```
depuis sousC
```

```
Nombre d'instances : 2
```

```
>>> C.printNumInstances()
```

```
Nombre d'instances : 2
```

Méthodes statiques

```
class SousC(C):  
    def printNumInstances():  
        print("depuis SousC")  
        C.printNumInstances()  
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> i = SousC()                                # on peut appeler une  
>>> i.printNumInstances()                       # méthode statique d'une  
depuis SousC                                   # instance  
Nombre d'instance : 3  
>>> SousC.printNumInstances()  
depuis SousC  
Nombre d'instance : 3
```

Méthodes statiques

```
class AutreSousC(C):  
    pass
```

```
>>> AutreSousC.printNumInstances()
```

```
Nombre d'instances : 3
```

```
>>> d = AutreSousC()
```

```
>>> d.printNumInstances()
```

```
Nombre d'instances : 4
```

- `AutreSousC()` appelle automatiquement le constructeur de la classe `C` ce qui incrémente le compteur d'instances

Méthodes de classe

```
class C:
    numInstances = 0
    def __init__(self):
        C.numInstances += 1
    def printNumInstances(cls):
        print("Nombre d'instances : ", cls, cls.numInstances)
    printNumInstances = classmethod(printNumInstances)
```

```
>>> c = C()
>>> c.printNumInstances()
Nombre d'instances : <class '__main__.C'> 1
>>> C.printNumInstances()
Nombre d'instances : <class '__main__.C'> 1
```

Méthode de classe

- La classe passée à la méthode de classe est
 - Si elle est appelée par une instance, la classe qui a créé l'instance
 - Si elle est appelée par une classe, la classe de l'appel

```
class SousC(C):  
    pass  
  
>>> c, sousC= C(), SousC()  
>>> C.printNumInstances()  
Nombre d'instances : <class '__main__.C'> 3  
>>> SousC.printNumInstances()  
Nombre d'instances : <class '__main__.SousC'> 3  
>>> sousC.printNumInstances()  
Nombre d'instances : <class '__main__.SousC'> 3
```

Nombre d'instances par sous classe

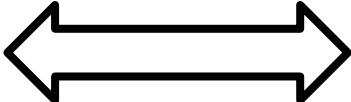
```
class C:
    numInstances = 0
    def __init__(self):
        self.count()
    def count(cls):
        cls.numInstances += 1
    def printNumInstances(cls):
        print("Nombre d'instances : ", cls, cls.numInstances)
    printNumInstances = classmethod(printNumInstances)
    count = classmethod(count)

class SousC(C):
    numInstances = 0
```

Nombre d'instances par sous classe

```
>>> c = C()
>>> sous1, sous2 = SousC(), SousC()
>>> c.printNumInstances()
Nombre d'instances : <class '__main__.C'> 1
>>> sous1.printNumInstances()
Nombre d'instances : <class '__main__.SousC'> 2
```

Introduction aux décorateurs

<pre>def f(): pass f = decorateur(f) f()</pre>		<pre>@decorateur def f(): pass f()</pre>
---	--	---

```
class C:
    def f(c):
        pass
    def g():
        pass
    def h(self):
        pass
    f = classmethod(f)
    g = staticmethod(g)
```

```
class C:
    @classmethod
    def f(c) :
        pass
    @staticmethod
    def g() :
        pass
    def h(self) :
        pass
```

Quand utiliser `staticmethod` ou `classmethod` ?

- La méthode statique est adaptée lorsque l'on veut le même comportement sur une classe et ses sous classes
 - Typiquement on ne manipule pas des attributs de la classe, ou en code en dur le nom de la super classe

Quand utiliser `staticmethod` ou `classmethod` ?

- La méthode de classe (puisque'elle reçoit la classe lors de l'appel) est adapté si
 - On a un comportement spécifique en fonction de la sous classe
 - On ne veut travailler sur des attributs de la classe, mais on ne veut pas coder en dur son nom
- Voir cet exemple intéressant d'utilisation des `staticmethod` et `classmethod`
 - <http://stackoverflow.com/questions/12179271/python-classmethod-and-staticmethod-for-beginner>

Les décorateurs

À quoi ça sert ?

- À faire un traitement avant et après l'appel d'une fonction
 - Calculer un temps d'exécution
 - Compter le nombre d'appels d'une fonction
 - Ajouter un log à chaque appel d'une fonction
 - Etc.

```
class C:
    def f(c):
        pass
    def g():
        pass
    def h(self):
        pass
    f = classmethod(f)
    g = staticmethod(g)
```

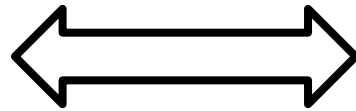
```
class C:
    @classmethod
    def f(c):
        pass
    @staticmethod
    def g():
        pass
    def h(self):
        pass
```

```
@decorateur
```

```
def f() :
```

```
    pass
```

```
f()
```



```
def f() :
```

```
    pass
```

```
f = decorateur(f)
```

```
f()
```

`f` n'est plus la fonction,
mais l'objet retourné par
`decorateur(f)`

Qu'est-ce qu'un décorateur ?

C'est un *callable* qui prend comme argument la fonction à décorer et retourne un *callable*

Qu'est-ce qu'un *callable* ?

C'est un objet O que l'on peut appeler avec $O()$

- Instance d'une classe qui implémente `__call__`
- Fonction

```
@decorateur  
def f(a, b):  
    pass
```

- `decorateur(f)` **retourne un callable** `O`
- `f(a, b)` **appelle**
en réalité `O(a, b)`

À quoi sert un décorateur ?

À ajouter une couche de logique à
une fonction avec une syntaxe
explicite `@decorateur`

Comment implémenter un décorateur ?

```

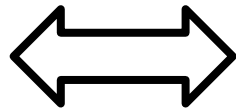
class NbAppel:
    def __init__(self, f):
        self.appel = 0
        self.f = f
    def __call__(self, *args):
        self.appel = self.appel + 1
        s = f'{self.f.__name__} {self.appel} appels'
        print(s)
        return self.f(*args)

```

```

@NbAppel
def f(a, b):
    print(a, b)

```



```

def f(a, b):
    print(a, b)
    f = NbAppel(f)

```

- `f` n'est plus une fonction, mais une instance de `NbAppel` qui retourne la valeur de retour de l'appel de la fonction originale
- `f(a, b)` appelle `__call__` sur l'instance
- On utilise `*args` pour accepter n'importe quelle signature de fonction à décorer
- On peut également utiliser `*args, **kwargs`

```
>>> f(1, 2)
f : 1 appels
1 2
>>> f(3, 'a')
f : 2 appels
3 a
```

```
>>> @NbAppel
def g(a, b, c):
    print(a, b, c)
```

```
>>> g(1, 2, 3)
g : 1 appels
1 2 3
```

- Il y a d'autres manières d'implémenter un décorateur
 - Clôture de fonction
 - Attribut `nonlocal`
 - Attribut de fonction

Qu'est-ce que la clôture de fonction ?

- Une technique héritée du lambda calcul pour garder un état dans une fonction entre deux appels

Variable libre

- Une variable est libre lorsqu'elle n'est ni locale, ni globale
 - C'est une variable définie dans les fonctions englobante

Qu'est-ce qu'un terme clôt ?

- Un terme est clôt lorsque toutes les variables sont soit locales, soit libres (donc définies dans une fonction englobante)

Qu'est-ce qu'un terme clôt ?

```
y = 3
```

```
def incremente(x):    # incremente n'est pas un terme clôt  
    return x + y      # parce que y n'est ni locale ni libre
```

```
print(incremente(5))
```

```
def incremente_par_n(y):
```

```
    def incremente(x):    # incremente est maintenant clôt  
        return x + y  
    return incremente
```

```
plus3 = incremente_par_n(3)
```

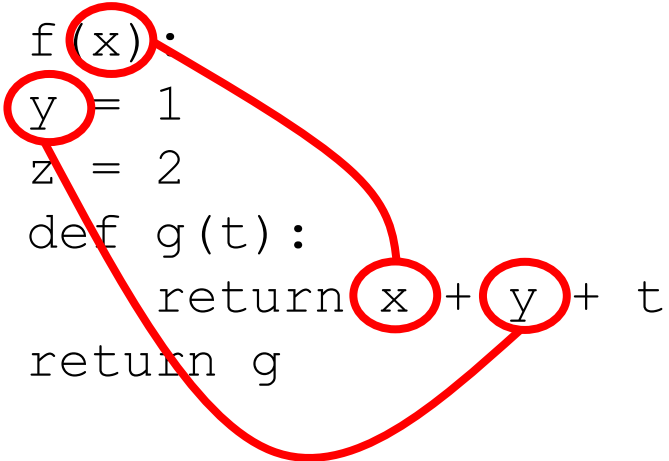
```
print(plus3(5))
```

```
>>>
```

```
8
```

Fonctionnement de la clôture

```
def f(x):  
    y = 1  
    z = 2  
    def g(t):  
        return x + y + t  
    return g
```



```
g1 = f(5)  
g2 = f(10)
```

```
>>> print(g1(20), g2(20))  
26 31
```

La fonction `f` retourne la fonction `g` qui garde dans un attribut `__closure__` un lien vers les objets définis dans `f` et référencés dans `g`

- Chaque appel à `f` crée un nouvel objet fonction `g`
- La clôture peut-être utilisée pour garder dans `g` une trace d'un objet défini dans `f` ou passé en argument à `f`

Fonctionnement de la clôture

```
>>> g1.__closure__  
(<cell at 0x024ED990: int object at 0x006A7660>, <cell  
at 0x025F16F0: int object at 0x006A7690>)
```

```
>>> g2.__closure__  
(<cell at 0x025F16D0: int object at 0x006A7624>, <cell  
at 0x025F15F0: int object at 0x006A7690>)
```

```
>>> g1.__closure__[0].cell_contents  
5
```

```
>>> g1.__closure__[1].cell_contents  
1
```

```
>>> g2.__closure__[0].cell_contents  
10
```

```
>>> g2.__closure__[1].cell_contents  
1
```

Utilisation de la clôture de fonction

```
import time
def timer(f):
    def wrapper(*args, **dargs):
        start = time.time()
        res = f(*args, **dargs)
        print(f'{time.time()-start} s')
        return res
    return wrapper

@timer
def f():
    return sum(x**3 for x in range(3_000_000))

f()

>>>
1.0937023162841797 s
```

Utilisation de la clôture de fonction

```
@timer
def f2(size):
    [x for x in range(size) if 'a' in range(size)]
```

```
f2(5000)
```

```
@timer
def f3(size, test):
    [x for x in range(size) if test in range(size)]
```

```
f3(test = 'a', size = 5000)
```

```
>>>
```

```
1.0313596725463867 s
```

```
1.0000133514404297 s
```

Limitation de la clôture de fonction

- La fonction englobée peut uniquement accéder à l'attribut de la fonction englobante, mais pas le modifier
- Sinon, l'attribut devient local à la fonction englobée

Attribut `nonlocal`

- Un attribut `nonlocal` permet de modifier une variable libre dans une fonction englobante

```
def caller(f):
    called = 0
    def wrapper(*args, **dargs):
        nonlocal called
        called = called + 1
        print(f'calling function {f.__name__}, '
              f' called {called} times')
        return f(*args, **dargs)
    return wrapper
```

```
@caller
def f():
    pass
```

```
@caller
def g(a, b):
    print('in g()', a, b)
```

```
g(1, 2)
g(b = 2, a = 1)
g('a', b = [])
f()
```

```
>>>
```

```
calling function g, called 1 times
in g() 1 2
calling function g, called 2 times
in g() 1 2
calling function g, called 3 times
in g() a []
calling function f, called 1 times
```

Attribut de fonction

- Permet de garder un état dans l'objet fonction `wrapper` et de le modifier
- À chaque appel de caller il y a un nouvel objet fonction, donc un nouvel attribut `called`

Attribut de fonction

```
def caller(f):  
    def wrapper(*args, **dargs):  
        wrapper.called = wrapper.called + 1  
        print(f'calling function {f.__name__}, '  
              f' called {wrapper.called} times')  
        return f(*args, **dargs)  
    wrapper.called = 0  
    return wrapper
```

```
@caller  
def f():  
    pass
```

```
@caller  
def g(a, b):  
    print('in g()', a, b)
```

Attribut de fonction

```
g(1, 2)
g(b = 2, a = 1)
g('a', b = [])
f()
```

```
>>>
```

```
calling function g, called 1 times
in g() 1 2
calling function g, called 2 times
in g() 1 2
calling function g, called 3 times
in g() a []
calling function f, called 1 times
```

Comment décorer une méthode built-in sans modifier le code source built-in ?

- On ne peut pas ajouter `@decorateur` puisqu'on ne veut pas modifier le code built-in

```
import builtins
import time
def change_set():
    original = builtins.set
    def wrapper(*args, **dargs):
        wrapper.called = wrapper.called + 1
        start = time.time()
        res = original(*args, **dargs)
        wrapper.duration += time.time() - start
        s = (f'calling set, called {wrapper.called} '
            f'times for {wrapper.duration} s')
        print(s)
        return res
    wrapper.called = 0
    wrapper.duration = 0
    builtins.set = wrapper

change_set()
set(range(100000000))
```



```
>>>  
calling set, called 1 times for 0.5313780307769775 s
```

Décoration de méthodes

- Une méthode prend comme premier argument l'instance
- Si on décore la méthode avec une classe, c'est l'instance de la classe du décorateur qui sera passée comme premier argument

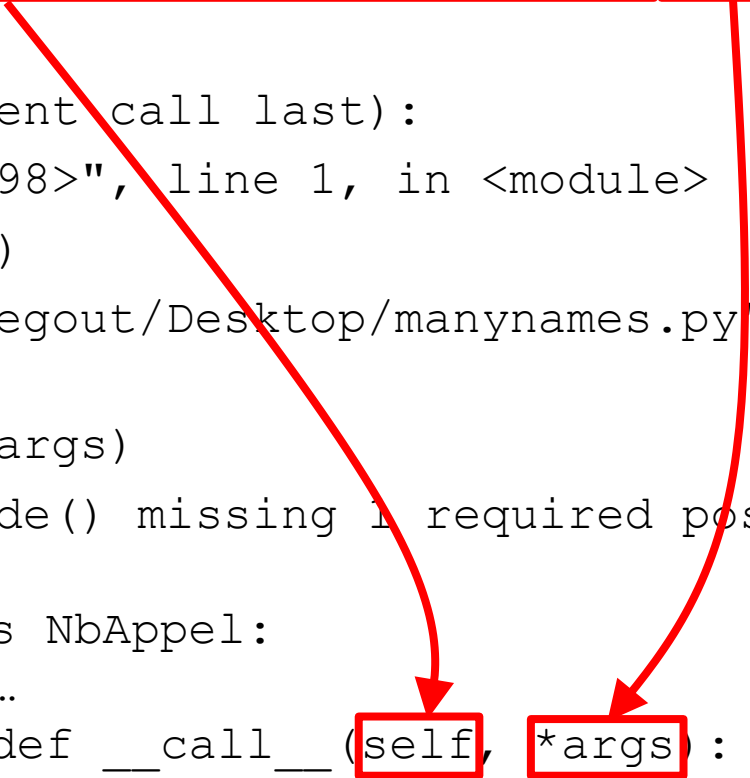
```
class NbAppel:
    def __init__(self, f):
        self.appel = 0
        self.f = f
    def __call__(self, *args):
        self.appel = self.appel + 1
        s = f'{self.f.__name__} : {self.appel} appels'
        print(s)
        print(self, args)
        return self.f(*args)
```

```
class C:
    @NbAppel
    def ma_methode(self, x):
        self.x = x
```

```
>>> c = C()
>>> c.ma_methode(10)
ma_methode : 1 appels
< main .NbAppel instance at 0x0273D9E0> (10,)
```

```
Traceback (most recent call last):
  File "<pyshell#1598>", line 1, in <module>
    c.ma_methode(10)
  File "C:/Users/alegout/Desktop/manynames.py", line 10, in
__call__
    return self.f(*args)
TypeError: ma_methode() missing 1 required positional argument:
'x'
```

```
class NbAppel:
    ...
    def __call__(self, *args):
        ...
        print self, args
        return self.f(*args)
```



Décoration de méthodes

- La solution est de décorer avec une fonction

```
def caller(f):  
    def wrapper(*args, **dargs):  
        wrapper.called = wrapper.called + 1  
        print(f'calling function {f.__name__}, '  
              f' called {wrapper.called} times')  
        return f(*args, **dargs)  
    wrapper.called = 0  
    return wrapper
```

```
class C:  
    @caller  
    def ma_methode(self, x):  
        self.x = x
```

```
>>> c = C()  
>>> c.ma_methode(10)  
calling function ma_methode, called 1 times
```

Comment garder les métadonnées de la fonction décorée

```
def mon_decorateur(func):  
    def wrapper(*args, **kwargs):  
        print('avant func')  
        func(*args, **kwargs)  
        print('apres func')  
    return wrapper  
  
@mon_decorateur  
def ma_fonction(a, b):  
    'une fonction qui ne fait presque rien'  
    print('dans ma Fonction')  
    print(a, b)
```

Comment garder les métadonnées de la fonction décorée

```
>>> ma_fonction(1, 2)
```

```
avant func
```

```
dans ma Fonction
```

```
1 2
```

```
apres func
```

```
>>> print(ma_fonction.__doc__)
```

```
None
```

```
>>> print(ma_fonction.__name__)
```

```
wrapper
```


Comment garder les métadonnées de la fonction décorée

Pour garder les métadonnées (principalement les attributes `__doc__` et `__name__`) on décore le wrapper avec `functools.wraps`

Comment garder les métadonnées de la fonction décorée

```
from functools import wraps
```

```
def mon_decorateur(func):  
    @wraps(func)  
    def wrapper(*args, **kwargs):  
        print('avant func')  
        func(*args, **kwargs)  
        print('apres func')  
    return wrapper
```

```
@mon_decorateur  
def ma_fonction(a, b):  
    'une fonction qui ne fait presque rien'  
    print('dans ma Fonction')  
    print(a, b)
```

Comment garder les métadonnées de la fonction décorée

```
>>> ma_fonction(1, 2)
```

```
avant func
```

```
dans ma Fonction
```

```
1 2
```

```
apres func
```

```
>>> print(ma_fonction.__doc__)
```

```
une fonction qui ne fait presque rien
```

```
>>> print(ma_fonction.__name__)
```

```
ma_fonction
```

Comment garder les métadonnées de la fonction décorée

```
>>> help(ma_fonction)
```

```
Help on function ma_fonction in module __main__:
```

```
ma_fonction(a, b)
```

```
    une fonction qui ne fait presque rien
```

Cascader les décorateurs

```
@timer
```

```
@nb_calls
```

```
def f() :
```

```
    ...
```

Est équivalent à

```
f = timer(nb_calls(f))
```

Passer des arguments au décorateur

- Il faut pour passer des arguments au décorateur ajouter une couche de logique au dessus
 - En général, on utilise une fonction au dessus du décorateur dont le seul rôle est de permettre au décorateur (fonction ou classe) de garder un accès aux arguments par une clôture

Un décorateur
couche de logique
pour capturer les
arguments

```
def nb_appel(label=''):
    class NbAppel:
        def __init__(self, f):
            self.appel = 0
            self.f = f
        def __call__(self, *args):
            self.appel = self.appel + 1
            s = (f'{label} {self.f.__name__} '
                 f': {self.appel} appels')
            print(s)
            return self.f(*args)
```

```
return NbAppel
```

La fonction retourne
le décorateur

```
@nb_appel("-->")
def f(a, b):
    print(a, b)
```

```
f = nb_appels("-->")(f)
f = NbAppel(f)
```

```
f(1, 2)
```

```
NbAppel(f)(1, 2) # dans __call__ 751
```

```
# attention à décorer avec les parenthèses si on ne passe
# pas d'argument et donc que l'argument optionnel doit
# être utilisé
@nb_appel()
def f(a, b):
    print(a, b)

f(1, 2)
```



```
def caller_builder(label=''):
```

Couche de logique
pour capturer les
arguments

```
    def caller(f):
```

```
        def wrapper(*args, **dargs):
```

```
            wrapper.called = wrapper.called + 1
```

```
            print(f'{label} {f.__name__}, '
```

```
                  f'called {wrapper.called} times')
```

```
            return f(*args, **dargs)
```

```
        wrapper.called = 0
```

```
        return wrapper
```

```
    return caller
```

La fonction retourne
le décorateur

```
class C:
```

```
    @caller_builder('method')
```

```
    def ma_methode(self, x):
```

```
        self.x = x
```

Un décorateur

```
@caller_builder('function')
```

```
def ma_fonction():
```

```
    pass
```

```
>>> C().ma_methode(1)
method ma_methode, called 1 times
>>> ma_fonction()
function ma_fonction, called 1 times
>>> ma_fonction()
function ma_fonction, called 2 times
```

Exemples de décorateurs

```
from functools import wraps

def runtime(func):
    """
    Décorateur qui affiche le temps d'exécution d'une
    fonction
    """
    import time
    @wraps(func)
    def wrapper(*args, **kwargs):
        t = time.perf_counter()
        res = func(*args, **kwargs)
        print(func.__name__, time.perf_counter()-t)
        return res
    return wrapper
```

```

from functools import wraps

def timer(nb_run=10, fmt='.2f'):
    def runtime(func):
        """
        temps d'exécution apres nb_run exécutions
        """
        import time
        @wraps(func)
        def wrapper(*args, **kwargs):
            t = time.perf_counter()
            for i in range(nb_run):
                res = func(*args, **kwargs)
            print(f"{func.__name__}:"
                  f" {(time.perf_counter()-t)/nb_run:{fmt}}"
                  f" on {nb_run} runs")
            return res
        return wrapper
    return runtime

```

```

def counter(func):
    """
    Décorateur qui affiche le nombre d'appels à une
    fonction
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        wrapper.count = wrapper.count + 1
        res = func(*args, **kwargs)
        print(f"{func.__name__} was called
{wrapper.count} times")
        return res
    wrapper.count = 0
    return wrapper

```

```

def logfunc(func):
    """
    Décorateur qui log l'activité d'une fonction.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        s = f"""
The function *{func.__name__}* was called:
- positional arguments: {args}
- named arguments: {kwargs}
- returned value: {res}
        """
        print(s)
        return res
    return wrapper

```

```
@logfunc
@counter
@runtime
def test(num, L):
    for i in range(num):
        'x' in L
    return 'Done'

test(100000, range(10))
```



```
import time
# partial re-implementation of the wraps decorator
def my_wraps(f):
    def wraps_deco(func):
        def wraps_wrapper(*args, **dargs):
            return func(*args, **dargs)
        wraps_wrapper.__name__ = f.__name__
        wraps_wrapper.__doc__ = f.__doc__
        return wraps_wrapper
    return wraps_deco

def timer(f):
    @my_wraps(f)
    def wrapper(*args, **dargs):
        start = time.time()
        res = f(*args, **dargs)
        print(f"{time.time()-start:.2f}s")
    return wrapper
```

```
@timer
def calcul():
    "my calcul docstring"
    return [x**3 for x in range(100_000)]

calcul(), calcul.__doc__, calcul.__name__
>>>
0.04s
(None, 'my calcul docstring', 'calcul')
```

Décorateurs de classes

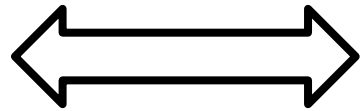
Même principe que pour les fonctions

```
@decorateur
```

```
class C:
```

```
    pass
```

```
c = C()
```



```
class C:
```

```
    pass
```

```
C = decorateur(C)
```

```
c = C()
```

C n'est plus la classe, mais
l'objet retourné par
decorateur(C)

```
@decorateur
```

```
class C:
```

```
...
```

- `decorateur(C)` **retourne un callable** `O`
- `C(a, b)` **appelle**
en réalité `O(a, b)`

Comment décore-t-on ?

- Comme pour les fonctions, on peut décorer avec une fonction ou une classe
- On peut utiliser les mêmes techniques : clôture, argument de fonctions, etc.
- Rappel: une classe est un objet mutable que l'on peut donc modifier après sa création
 - Le décorateur de classe est une manière de factoriser cette opération à de multiples classes

Exemples de décorateurs

- On veut décorer chaque méthode d'une classe avec `logfunc`
 - On peut évidemment décorer chaque méthode en ajoutant `@logfunc`
 - On peut faire un décorateur de classe qui le fait pour nous

```
class C:  
    def f(self):  
        pass  
    def g(self):  
        pass  
    def h(self):  
        pass
```

```
c = C()
```

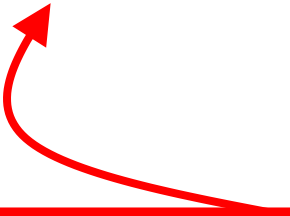
```
c.f(), c.g(), c.h()
```



```
def caller_allmethods(cls):  
    for name, obj in vars(cls).items():  
        if callable(obj):  
            setattr(cls, name, logfunc(obj))  
    return cls
```

```
@caller_allmethods  
class C:  
    def f(self):  
        pass  
    def g(self):  
        pass  
    def h(self):  
        pass
```

```
c = C()  
c.f(), c.g(), c.h()
```



On décore chaque
méthode avec logfunc

Pour aller plus loin

Exemples de décorateur

<https://zestedesavoir.com/tutoriels/954/notions-de-python-avancees/5-exercises/2-3-decorators/>

<https://realpython.com/primer-on-python-decorators/>

Pour aller plus loin

David Beazley: Python 3 Metaprogramming

<http://www.dabeaz.com/py3meta/index.html>

<http://sahandsaba.com/python-decorators.html>

Fonction génératrice et conception d'itérateurs

À quoi ça sert ?

- À créer facilement des itérateurs en abstrayant la complexité du traitement

Fonction générateur (generator)

- Un générateur est une fonction normale, mais qui au lieu de s'exécuter et de retourner une seule valeur, va retourner une valeur, se suspendre, reprendre de l'état suspendu au prochain appel, retourner une valeur, se suspendre, etc.
 - On utilise le mot clef `yield` pour retourner la valeur à la place de `return`
 - On utilise la méthode `__next__()` sur le générateur pour obtenir la prochaine valeur retournée par `yield`

Fonction générateur (generator)

- Lors de l'appel d'une fonction standard
 - Un espace de nommage est créé pour les variables locales à la fonction
 - L'espace de nommage est détruit à l'appel de `return` c'est-à-dire à la sortie de la fonction
- Lors de l'appel d'une fonction générateur
 - Un espace de nommage est créé pour les variables locales à la fonction
 - Cet espace de nommage est conservé jusqu'à la fin de l'itération

Fonction générateur (generator)

- Python choisi créer une fonction normale ou un générateur à la pré-compilation
 - Si le pré-compilateur trouve le mot clef `yield` dans le corps de la fonction, alors Python créera un générateur

Fonction générateur (generator)

```
>>> def f():  
    yield 2
```

```
>>> f()  
<generator object f at 0x027263F0>  
>>> it = f()  
>>> next(it)  
2  
>>> next(it)
```

```
Traceback (most recent call last):  
  File "<pyshell#1321>", line 1, in <module>  
    next(it)  
StopIteration
```

Fonction générateur (generator)

```
def func(n):  
    for i in range(n):  
        yield i**2
```

```
for i in func(10):  
    print(i, end=' ')
```

```
>>>
```

```
0 1 4 9 16 25 36 49 64 81
```

- La boucle `for` crée l'itérateur du générateur et appelle la fonction `__next__()` sur cet itérateur

Fonction générateur (generator)

- L'itérateur du générateur est le générateur lui même
 - On peut donc directement appeler `__next__()` sur le générateur
 - `__next__()` retourne simplement la prochaine valeur retournée par `yield`

Fonction générateur (generator)

```
>>> x=func(4)
>>> x.__next__()
0
>>> x.__next__()
1
>>> x.__next__()
4
>>> x.__next__()
9
>>> x.__next__()
Traceback (most recent call last):
  File "<pyshell#456>", line 1, in <module>
    x.__next__()
StopIteration
```

Fonction générateur (generator)

```
>>> x=func(2)
>>> x
<generator object func at 0x02BBB940>
>>> y = iter(x)
>>> y
<generator object func at 0x02BBB940>
>>> z = iter(y)
>>> z
<generator object func at 0x02BBB940>
```

Fonction générateur (generator)

```
>>> next(x)
```

```
0
```

```
>>> next(y)
```

```
1
```

```
>>> next(z)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#1341>", line 1, in <module>
```

```
    next(z)
```

```
StopIteration
```

Que se passe-t-il si on mélange `yield` et `return` ?

```
>>> def f():  
...     yield 1  
...     yield 2  
...     return 3  
...     yield 4  
...  
>>> for i in f():  
...     print(i)  
...  
1  
2  
>>>
```


Que se passe-t-il si on mélange `yield` et `return` ?

```
>>> g = f()
```

```
>>> next(g)
```

```
1
```

```
>>> next(g)
```

```
2
```

```
>>> next(g)
```

```
Traceback (most recent call last):
```

```
  File "C:\IPython\core\interactiveshell.py", line  
2881, in run_code
```

```
    exec(code_obj, self.user_global_ns, self.user_ns)
```

```
  File "<ipython-input-4-5f315c5de15b>", line 1, in  
<module>
```

```
    next(g)
```

```
StopIteration: 3
```

Que se passe-t-il si on mélange `yield` et `return` ?

- Les générateurs s'arrêtent en produisant `StopIteration`
- Si la sortie est faite par un `return`, alors, la valeur du `return` devient un argument de `StopIteration`

Que se passe-t-il si on mélange `yield` et `return` ?

- La bonne manière de sortir d'un itérateur est soit d'arriver à la fin du bloc de code (fin d'un `while`, fin d'une boucle `for`) ou de faire un `return`, mais jamais de faire `raise StopIteration`
 - Voir la PEP 479
<https://www.python.org/dev/peps/pep-0479/>

Fonction générateur et itérateur

- Les générateurs représentent une manière pratique d'implémenter un itérateur sur un objet (dans une classe)
 - La fonction `__iter__` sur l'objet doit être implémentée comme un générateur (c'est-à-dire la fonction `__iter__` doit retourner une valeur avec `yield`)
 - L'itérateur produit avec `iter(obj)` contient automatiquement les méthodes `__iter__` et `__next__()`

Comment implémenter un itérateur pour notre classe `Mots` ?

- On veut implémenter une classe `Mots` qui prend une phrase dans le constructeur et produit une instance qui permet d'itérer sur les mots
- Cas 1 : on implémente dans `Mots` les méthodes `__iter__()` et `__next__()`
 - `__iter__()` retourne `self`
 - L'instance est l'itérateur, donc un unique itérateur par instance
 - `__next__()` implémente l'itération

```
class Mots():
    def __init__(self, phrase):
        self.phrase = phrase.split()
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count == len(self.phrase):
            raise StopIteration
        self.count = self.count + 1
        return self.phrase[self.count - 1]

>>> m = Mots("une grande phrase")
>>> [x for x in m]
['une', 'grande', 'phrase']
>>> [x for x in m] #il n'y a qu'un itérateur par instance
[]
```

Comment implémenter un itérateur pour notre classe `Mots` ?

- Cas 2 : on implémente dans `Mots` la méthode `__iter__()` qui retourne une instance d'une nouvelle classe `IterMots` qui sera un itérateur pour notre classe `Mots`
 - On implémente dans `IterMots` les méthodes `__iter__()` qui retourne `self` et `__next__()` qui implémente l'itération
 - Chaque itération sur une instance de `Mots` crée un nouvel itérateur

```
class Mots():
    def __init__(self, phrase):
        self.phrase = phrase.split()

    def __iter__(self):
        return IterMots(self.phrase)

class IterMots():
    def __init__(self, phrase):
        self.phrase = phrase
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count == len(self.phrase):
            raise StopIteration
        self.count = self.count + 1
        return self.phrase[self.count - 1]
```



```
>>> m = Mots("une grande phrase")
>>> [x for x in m]
['une', 'grande', 'phrase']
>>> [x.upper() for x in m] #il y a de multiples itérateurs
                             #par instance
['UNE', 'GRANDE', 'PHRASE']
>>> [x.upper() for x in m if 'a' in x]
['GRANDE', 'PHRASE']
```

Comment implémenter un itérateur pour notre classe `Mots` ?

- Cas 3 : on implémente dans `Mots` uniquement `__iter__()` sous forme d'une fonction générateur
 - Chaque itération sur une instance crée une nouvelle fonction génératrice, donc un nouvel itérateur
 - C'est la solution la plus compacte à écrire

```
class Mots():  
    def __init__(self, phrase):  
        self.phrase = phrase.split()  
  
    def __iter__(self):  
        for i in self.phrase:  
            yield i
```

```
>>> m = Mots("une grande phrase")  
>>> [x for x in m]  
['une', 'grande', 'phrase']  
>>> [x for x in m]  
['une', 'grande', 'phrase']
```

Retour sur les générateurs

- L'utilisation des générateurs pour écrire un itérateur pour une classe n'est qu'une toute petite partie de leur utilité
- Les générateurs sont à la base de presque tout en Python 3
 - Permet d'abstraire la difficulté d'un traitement
 - Très performant
 - S'utilise avec tout ce qui itère
 - À la base des coroutines et de la programmation asynchrone

Exemple de générateurs

- Je veux extraire toutes les entêtes de fonctions Python contenues dans tous les fichiers Python d'un répertoire

```

import os
def extract_def_headers(dossier):
    with os.scandir(dossier) as scan:
        for file in scan:
            if file.is_file() and file.name.endswith('.py'):
                try:
                    with open(file) as f:
                        for line in f:
                            if line.strip().startswith('def '):
                                yield line.strip()[4:-1]
                except UnicodeDecodeError as e:
                    print(f"{file.path} is a binary file")

# exemples d'utilisation
for func_name in extract_def_headers('.'):
    print(func_name)

L = [func_name for func_name in extract_def_headers('.') if
func_name.startswith('get')]
print(L)

```

Exemple de générateurs

- On a pas besoin de classes alors ?
 - Si, mais uniquement pour les cas les plus sophistiqués
 - Traitement très complexe
 - Besoin d'héritage
 - Etc.
 - Les générateurs fournissent une solution légère et élégante pour abstraire des traitements itératifs

Comment refactoriser des générateurs

```
import math  
def gen_etrange(n):  
    for i in range(n):  
        yield i**2  
  
    for i in range(n):  
        yield math.sqrt(i)
```

- Comment factoriser le code en vert ?


```
def gen_carre(n):  
    for i in range(n):  
        yield i**2  
  
def gen_sqrt(n):  
    for i in range(n):  
        yield math.sqrt(i)  
  
def gen_etrange(n):  
    for i in gen_carre(n):  
        yield i  
  
    for i in gen_sqrt(n):  
        yield i
```

- Pas très élégant !
 - Il faut utiliser la syntaxe `yield from`
 - Équivalent au code en rouge

```
def gen_etrange(n):  
    yield from gen_carre(n)  
    yield from gen_sqrt(n)
```

- `yield from` est beaucoup plus puissant que cela, on l'utilise pour les coroutines
 - C'est de la délégation

Chaîne de traitements

- Une usage classique des générateurs est de les chaîner pour faire une chaîne de traitement
 - L'idée est de faire de grands traitements sans jamais générer une grande structure de données temporaire

```
def cat_on_file(filename):  
    with open(filename, 'r') as f:  
        for line in f:  
            yield line.strip()
```

yield une valeur, donc
crée un générateur

Prend un générateur
et itère dessus

```
def remove_comments(lines):  
    for line in lines:  
        if not line.startswith('#'):  
            yield line
```

yield une valeur, donc
crée un générateur

```
def get_func_headers(lines):  
    for line in lines:  
        if line.startswith('def') and line.endswith(':'):  
            yield line
```

Prend un générateur
et itère dessus

yield une valeur, donc
crée un générateur

```
all_lines = cat_on_file('formation_avancee.py')
```

générateur



```
all_lines_no_comment = remove_comments(all_lines)
```

générateur

```
all_func = get_func_headers(all_lines_no_comment)
```

générateur

```
for i in all_func:  
    print(i)
```

Pour aller plus loin

Le module itertools

<https://docs.python.org/3/library/itertools.html?module=itertools>

A Curious Course on Coroutines and Concurrency

<http://www.dabeaz.com/coroutines/index.html>

Gestion avancée des attributs

À quoi ça sert ?

- À ajouter une couche de logique lorsque l'on accède, modifie ou efface un attribut d'un objet

Qu'est-ce que la gestion des attributs ?

- Supposons que l'on veuille faire une action spécifique lors de l'accès ou de la modification d'un attribut (validation, etc.)
 - `a.attr = 10`
 - `print(a.attr)`

On a deux solutions

- Créer des méthodes `setattr()` et `getattr()`, mais il faut changer tous les appels à l'attribut `attr`
- Modifier la manière dont Python accède aux attributs

Il y a 3 manières de modifier l'accès aux attributs

- Les propriétés
- Les descripteurs
- Les méthodes `__getattr__`,
`__setattr__` et
`__getattribute__`

Spécifique à un attribut

Intercepte tous les attributs

1. Les propriétés

Les propriétés

```
class C:
    def __init__(self):
        # on utilise self.x pour passer par setx
        # à l'initialisation (au lieu de self._x)
        # pas de risque d'appel récursif ici
        self.x = None

    def getx(self):          # getter appelé par c.x
        print('get x')
        return self._x

    def setx(self, value):  # setter appelé par c.x = value
        print(f'set x to {value}')
        self._x = value

    def delx(self):         # deleter appelé par del c.x
        print('del x')
        del self._x

    # ici on déclare que x est une propriété de C
    x = property(getx, setx, delx, "docstring de 'x'.")
```

Les propriétés

```
>>> inst = C()
>>> inst.x = 10
set x to 10
>>> print(inst.x)
get x
10
>>> del inst.x
del x
>>> help(inst)
Help on C in module __main__ object:
```

```
class C(builtins.object)
|   Methods defined here:
|   ...
|   -----
|   Data descriptors defined here:
|   ...
|   x
|   docstring de 'x'.
```

Les propriétés

- Les propriétés vivent dans la classe et accèdent aux attributs de l'instance par `self`

Les propriétés

- Les propriétés sont héritées par les sous classes

```
>>> class subC(C): pass
```

```
>>> inst = subC()
```

```
>>> inst.x = 10
```

```
set x to 10
```

```
>>> print(inst.x)
```

```
get x
```

```
10
```

```
>>> del inst.x
```

```
del x
```

Les propriétés

- Le getter retourne la valeur retournée lors de l'accès à l'attribut
- Le setter et le deleter retournent `None`
- Les 4 arguments de `property` sont optionnels
 - Si le setter, getter ou deleter n'est pas spécifié, l'opération correspondante est interdite
 - Par défaut, la doctring est celle du getter

Les propriétés et décorateurs

- Il existe un décorateur `@property` qui décore le getter
 - On doit appeler dans ce cas le getter du nom de l'attribut, e.g., `attr`
 - On peut ensuite décorer le setter et le deleter avec `@attr.setter` et `@attr.deleter`

Les propriétés et décorateurs

```
class C:
```

```
    def __init__(self):  
        self._x = None
```

```
@property
```

```
def x(self):                                # appelé par c.x  
    'docstring for property x'  
    print('get x')  
    return self._x
```

```
@x.setter
```

```
def x(self, value):                        # appelé par c.x = value  
    print(f'set x to {value}')
```

```
    self._x = value
```

```
@x.deleter
```

```
def x(self):                                # appelé par del c.x  
    print('del x')  
    del self._x
```

Avec la syntaxe du décorateur,
les trois méthodes ont le nom
de l'attribut

```
>>> help(C)
Help on C in module __main__ object:
```

```
class C(builtins.object)
|   Methods defined here:
|
|   __init__(self)
|       Initialize self.  See help(type(self)) ...
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
|
|   x
|       docstring for property x
```

2. Les descripteurs

<https://docs.python.org/3/howto/descriptor.html>

Les descripteurs

- Un descripteur est une classe qui détermine le comportement lors de l'accès, l'affectation et l'effacement d'un attribut
- Très puissant, utilisé pour implémenter
 - Les propriétés
 - Les méthodes statiques et de classe
 - Les méthodes bound et unbound
 - `super()`
 - Etc.

```
class TraceAccessX:
    'docstring for descriptor TraceAccessX'
    def __init__(self):
        self.nb_get = 0

    def __get__(self, inst, insttype):
        self.nb_get = self.nb_get + 1
        print(f'get {self.nb_get} times')
        return inst._x

    def __set__(self, inst, val):
        print('set x')
        inst._x = val

    def __delete__(self, inst):
        print('deleting x...')
        del inst._x
```

```
class C:
    def __init__(self):
        self.x = 0
    x = TraceAccessX()
```

Une classe avec au moins une méthode `__get__`, `__set__` ou `__delete__` est un descripteur


```
class TraceAccessX:
    'docstring for descriptor TraceAccessX'
    def __init__(self):
        self.nb_get = 0

    def __get__(self, inst, insttype):
        self.nb_get = self.nb_get + 1
        print(f'get {self.nb_get} times')
        return inst._x

    def __set__(self, inst, val):
        print('set x')
        inst._x = val

    def __delete__(self, inst):
        print('deleting x...')
        del inst._x

class C:
    def __init__(self):
        self.x = 0
    x = TraceAccessX()
```

```
class TraceAccessX:
    'docstring for descriptor TraceAccessX'
    def __init__(self):
        self.nb_get = 0

    def __get__(self, inst, insttype):
        self.nb_get = self.nb_get + 1
        print(f'get {self.nb_get} times')
        return inst._x

    def __set__(self, inst, val):
        print('set x')
        inst._x = val

    def __delete__(self, inst):
        print('deleting x...')
        del inst._x

class C:
    def __init__(self):
        self.x = 0

x = TraceAccessX()
```

The diagram illustrates the interaction between the `TraceAccessX` descriptor and the `C` class. Red arrows trace the flow of attribute access:

- `C.__init__` calls `self.x`, which triggers `TraceAccessX.__get__(self, inst, insttype)`.
- `C.__init__` calls `self.x = 0`, which triggers `TraceAccessX.__set__(self, inst, val)`.
- `C.__init__` calls `del self.x`, which triggers `TraceAccessX.__delete__(self, inst)`.

The diagram also shows the state of `C.x` at each step:

- Initially, `C.x` is 0.
- After `self.x = 0`, `C.x` is set to 10.
- After `del self.x`, `C.x` is deleted.

```
class TraceAccessX:
    'docstring for descriptor TraceAccessX'
    def __init__(self):
        self.nb_get = 0

    def __get__(self, inst, insttype):
        self.nb_get = self.nb_get + 1
        print(f'get {self.nb_get} times')
        return inst.x

    def __set__(self, inst, val):
        print('set x')
        inst.x = val

    def __delete__(self, inst):
        print('deleting x...')
        del inst.x

class C:
    def __init__(self):
        self.x = 0
    x = TraceAccessX()
```

L'attribut d'instance ne doit pas avoir le même nom que le descripteur. Sinon, il y a un appel récursif lors de l'appel de `__set__`

```
>>> c = C()
>>> c.__dict__
{'_x': 0}
>>> C.__dict__
mappingproxy({'x': <__main__.TraceAccessX object at
0x031AF050>, '__init__': <function C.__init__ at
0x03417A08>, '__module__': '__main__', '__doc__': None,
'__dict__': <attribute '__dict__' of 'C' objects>,
'__weakref__': <attribute '__weakref__' of 'C' objects>})
>>> c.x
get 1 times
0
>>> c.x = 10
set x
>>> c.x
get 2 times
10
```

```

>>> help(C)
Help on class C in module __main__:

class C(builtins.object)
|   Methods defined here:
|
|   __init__(self)
|       Initialize self.  See help(type(self)) ...
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if
defined)
|
|   x
|       docstring for descriptor TraceAccessX

```

```
>>> del c.x
deleting x...
>>> c.x
get 3 times
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    c.x
  File "C:/Users/alegout/Desktop/test.py", line 8, in
__get__
    return inst._x
AttributeError: 'C' object has no attribute '_x'
```

Stockage des attributs

- Il y a une instance du descripteur par attribut de la classe C
- L'instance du descripteur peut stocker ses propres attributs qui seront partagés par toutes les instances de la classe C
- Un descripteur peut accéder aux attributs des instances
- Le descripteur peut modifier les attributs stockés dans l'instance

Où stocker les attributs ?

- Dans le descripteur si l'information doit être partagée par toutes les instances de C
- Dans l'instance de C si l'information doit être différente pour chaque instance de C

Que se passe-t-il si l'attribut de l'instance a le même nom que le descripteur (l'attribut de la classe) ?

- Le descripteur doit préempter la règle de recherche des attributs sinon il ne serait jamais appelé dans ce cas puisque l'attribut d'instance est prioritaire sur l'attribut de classe

Data et non-data descripteurs

- Un data descripteur implémente `__get__()` et `__set__()`
 - Il **sera prioritaire** sur un attribut d'instance du même nom

```
class TraceAccessX:
    def __get__(self, inst, insttype):
        print('get x')
        return inst._x

    def __set__(self, inst, val):
        print('set x')
        inst._x = val

class C:
    def __init__(self):
        self.x = 0
    x = TraceAccessX()
```

```
>>> c = C()  
>>> c.x  
get x  
0  
>>> c.x  
get x  
0  
>>> c.x = 10  
set x  
>>> c.x  
get x  
10
```

Data et non-data descripteurs

- Un non-data descripteur implémente uniquement `__get__()`
 - Il **ne sera pas prioritaire** sur un attribut d'instance du même nom
- C'est notamment le mécanisme utilisé pour implémenter
 - Les méthodes `bounds`
 - Les `staticmethod` et `classmethod`

```
class TraceAccessX:
    def __get__(self, inst, insttype):
        print('get x')
        return inst._x
```

```
class C:
    def __init__(self):
        self._x = 0
    x = TraceAccessX()
```

```
>>> c = C()
```

```
>>> c.x
```

```
get x
```

```
0
```

```
>>> c.x = 10
```

```
>>> c.x
```

```
10
```

```
>>> del c.x
```

```
>>> c.x
```

```
get x
```

```
0
```

```
class C:
    def ma_methode(self):
        pass

inst = C()

# Les objets Function sont des non-data descriptor. Lorsqu'une
# fonction est appelée depuis une classe, le non-data descriptor
# retourne une fonction classique
# https://docs.python.org/3/howto/descriptor.html#functions-and-methods
>>> C.ma_methode
<function __main__.C.ma_methode(self)>get x

# Lorsqu'elle est appelée depuis une instance, le non-data
# descriptor retourne une méthode bound
>>> inst.ma_methode
<bound method C.ma_methode of <__main__.C object at 0x07640>>

# Mais, si l'on crée un attribut ma_methode dans l'instance
# inst, il va préempter l'accès à la méthode bound. C'est le
# comportement attendu et la raison de la logique derrière
# les non-data descriptor
>>> inst.ma_methode = 25
>>> inst.ma_methode
25
```

Attention aux appels récursifs

- Si une variable `x` est un descripteur et que dans le descripteur on appelle ou on affecte `self.x`, il va y avoir un appel récursif
- C'est pourquoi on préfère utiliser dans le descripteur `self. x`


```
class TraceAccessX:
    def __get__(self, inst, insttype):
        print('get x')
        return inst.x

    def __set__(self, inst, val):
        print('set x')
        inst.x = val

class C:
    def __init__(self):
        self.x = 0      # À la création de l'instance
        x = TraceAccessX() # il y a un appel récursif

>>> c = C()
...
RecursionError: maximum recursion depth exceeded
```

Comment rendre un descripteur read-only ?

- Contrairement aux propriétés, l'absence de la méthode `__set__` ne rend pas le non-data descripteur read-only
- Il faut définir une méthode `__set__` qui retourne l'exception `AttributeError`

```
class TraceAccessX:
    def __get__(self, inst, insttype):
        print('get x')
        return inst._x

    def __set__(self, inst, val):
        raise AttributeError('cannot set attribut')

class C:
    def __init__(self, const):
        self._x = const
    X = TraceAccessX()
```

```
>>> c = C(25)
>>> print(c.X)
get x
25
```

```
>>> c.X = 10
```

```
AttributeError          Traceback (most recent call last)
<ipython-input-12-3ad54634f457> in <module>()
----> 1 c.x = 10
```

```
<ipython-input-10-8d0f3426d224> in __set__(self, inst,
val)
```

```
5
6     def __set__(self, inst, val):
----> 7         raise AttributeError('cannot set
attribut')
8
9 class C:
```

```
AttributeError: cannot set
```

Exemples de descripteurs

- Property

<https://docs.python.org/3/howto/descriptor.html#properties>

- Méthodes bound

<https://docs.python.org/3/howto/descriptor.html#functions-and-methods>

- Méthodes statiques

<https://docs.python.org/3/howto/descriptor.html#static-methods>

- More examples

<https://docs.python.org/3/howto/descriptor.html#primer>

3. Les méthodes `__getattr__`,
`__setattr__` et
`__getattribute__`

`__getattr__`

- Appelée par `ints.name` si l'attribut `name` n'est pas trouvé le long de l'arbre d'héritage de `inst`
- Retourne une valeur ou `AttributeError`

```
class Traced:
    def __getattr__(self, name):
        print(f'getattr called: Get: {name}')
```



```
>>> t = Traced()
>>> t.x
getattr called: Get: x
>>> t.y = 10
>>> t.y
10
```



```
# implement an automatic method name conversion to handle
# different method spelling conventions
class House:
    def __init__(self, members):
        self.members = members

    # we accept, CamelCase and snake_case convention
    def __getattr__(self, attr):
        return getattr(self, attr.lower().replace('_', ''))

    def isempty(self):
        return len(self.members) == 0

h = House(['alice', 'bob', 'eve'])

>>> h.isempty(), h.is_empty(), h.isEmpty()
(False, False, False)
```

`__getattribute__`

- Toujours appelée par `inst.name` que `name` existe ou qu'il n'existe pas dans l'arbre d'héritage de `inst`
- `__getattr__` ne sera pas appelé si `__getattribute__` est également défini

`__getattrute__`

- **Retourne une valeur ou**
`AttributeError`

```
class Traced:
    def __getattr__(self, name):
        print(f'getattr called: Get: {name}')

>>> t = Traced()
>>> t.x
getattr called: Get: x
>>> t.y = 10
>>> t.y
getattr called: Get: y
>>>
```

Subtilités de

`__getattr__`

- Comme `__getattr__` est toujours appelée, si l'attribut existe généralement on souhaite le retourner

Subtilités de `__getattr__`

- Mais `getattr(self, name)` va créer un appel récursif, il faut donc router l'appel de `__getattr__` en dehors de la classe dans laquelle cette méthode est définie
- on utilise toujours `object.__getattr__(self, name)`

```
class Traced:
    def __getattribute__(self, name):
        print(f'getattribute called: Get: {name}')
        return getattr(self, name) #appel récursif de
                                    #__getattribute__
```

```
>>> t = Traced()
```

```
>>> t.x
```

```
...
```

```
RecursionError: maximum recursion depth exceeded while
calling a Python object
```

```
class Traced:
    def __getattr__(self, name):
        print(f'getattr called: Get: {name}')
        return object.__getattr__(self, name)

>>> t = Traced()
>>> t.x
getattr called: Get: x

Traceback (most recent call last):
  File "<pyshell#120>", line 1, in <module>
    t.x
  File "C:/Users/alegout/Desktop/test.py", line 4, in
__getattr__
    return object.__getattr__(self, name)
AttributeError: 'Traced' object has no attribute 'x'
>>>
```


Subtilités de

`__getattr__`

- Comme toute la logique de recherche des descripteurs est implémentée dans `object.__getattr__` si on surcharge cette méthode sans l'appeler explicitement sur `objet`, on désactive les descripteurs

Subtilités de

`__getattr__`

- Les appels **implicites** des méthodes utilisées pour la surcharge des opérateurs ne sont pas routés par

`__getattr__`

- Même si `__getattr__` est définie pour `obj`, `len(obj)` ou `print(obj)` ne vont pas passer par `__getattr__`

```
class C:
    def __len__(self):
        return 10
    def __getattr__(*args):
        print("__getattr__ de la class C appelé")
        return object.__getattr__(*args)

c = C()
print("appel explicite")
c.__len__() # appel explicite
print("appel implicite")
len(c)      # appel implicite
>>>
appel explicite
__getattr__ de la class C appelé
appel implicite
```

```
# We can access the keys of the dict using the
# notation d.key instead of d['key']
# Used by pandas dataframe to access columns, it is very
# dangerous because keys preempt over methods!

class EasyDict(dict):
    def __getattr__(self, attr):
        if (not attr.endswith('__')) and (attr in self):
            return self[attr]
        else:
            return object.__getattr__(self, attr)

d = EasyDict({'a': 1, 'b': 2})
>>> d['a'], d.a
(1, 1)

>>> d.keys()
dict_keys(['a', 'b'])

>>> d['keys'] = 20
>>> d.keys() # we look for the dict key, not the method
TypeError: 'int' object is not callable
```

`__setattr__`

- Toujours appelée par
`inst.name = value`
- Si `__setattr__` veut affecter une valeur à l'instance on ne peut pas utiliser

`setattr(self, name, value)`

parce que ça crée un appel récursif

`__setattr__`

- On route l'appel en dehors de la classe en utilisant

`object.__setattr__(self, name, value)`

```
class Traced:
    def __setattr__(self, name, value):
        print(f'setattr called Set: {name} {value}')
        object.__setattr__(self, name, value)

>>> t = Traced()
>>> t.x = 10
setattr called Set: x 10
>>> t.x
10
```

```

def watch_variables(var_list):
    """Usage: @watch_variables(['var1', 'var2'])"""
    def _decorator(cls):
        def _setattr(self, name, value):
            if name in var_list:
                print(f'the attribute {name} has been set to {value}')
            return object.__setattr__(self, name, value)

        # we change the __setattr__ method of the decorated class
        cls.__setattr__ = _setattr
        return cls
    return _decorator

@watch_variables(['spam', 'beans'])
class MyClass(object):
    def __init__(self, spam, beans):
        self.spam = spam
        self.beans = beans
        self.chicken = 12

>>> b = MyClass(1, 2)

the attribute spam has been set to 1
the attribute beans has been set to 2

```


Subtilités de `__setattr__`

- Comme toute la logique de modification des descripteurs est implémentée dans `object.__setattr__` si on surcharge cette méthode sans l'appeler explicitement sur `objet`, on désactive les descripteurs

Différence avec les propriétés et les descripteurs

- Une propriété ou un descripteur est lié à un attribut particulier
- Les méthodes `__getattr__`, `__getattribute__` et `__setattr__` sont appelées pour tous les attributs

Comment les descripteurs fonctionnent sous le capot ?

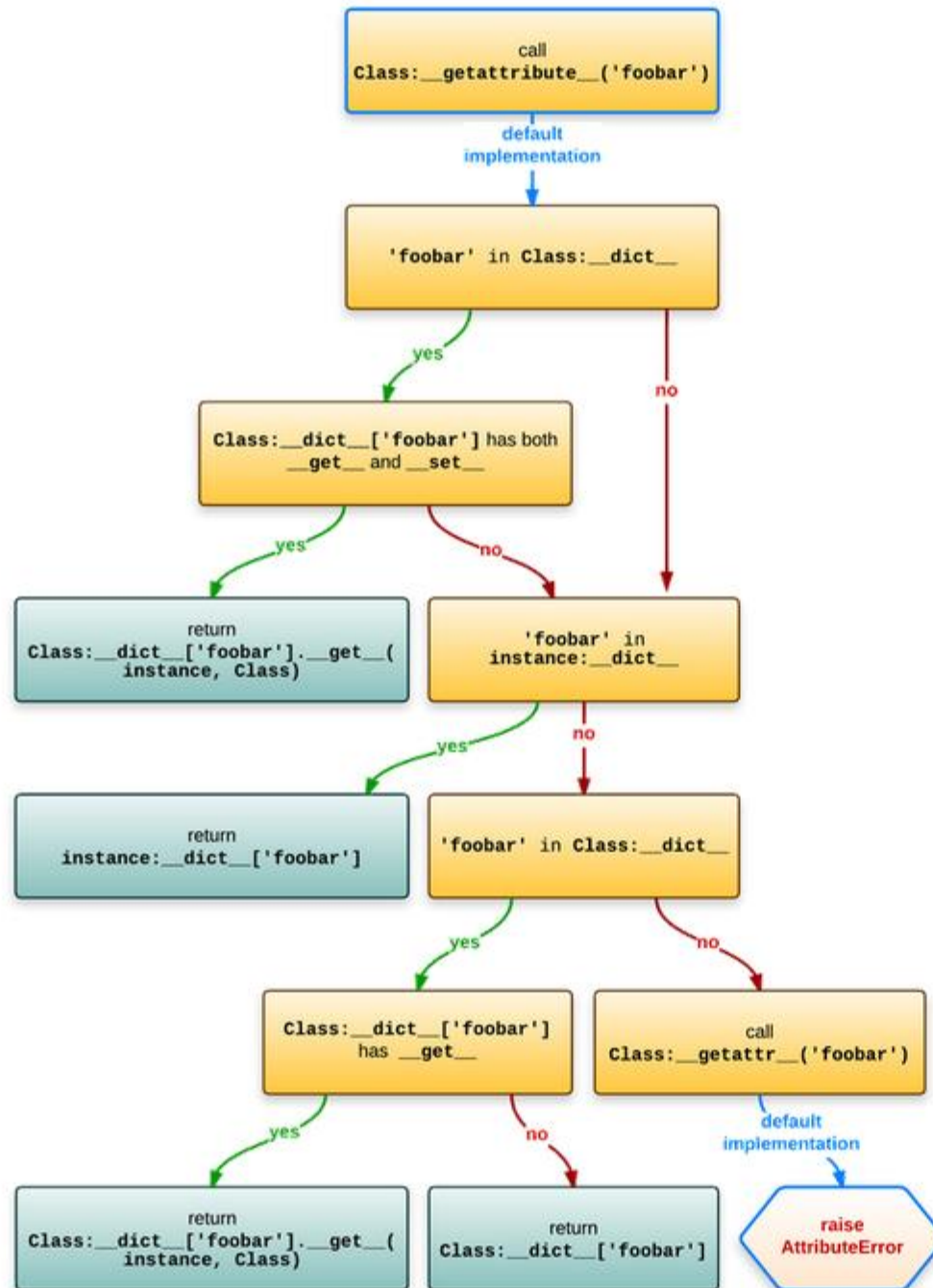
- Tout est implémenté dans les méthodes

`type.__getattr__` (pour la recherche d'attributs pour les classes) et

`object.__getattr__` (pour la recherche d'attributs pour les instances)

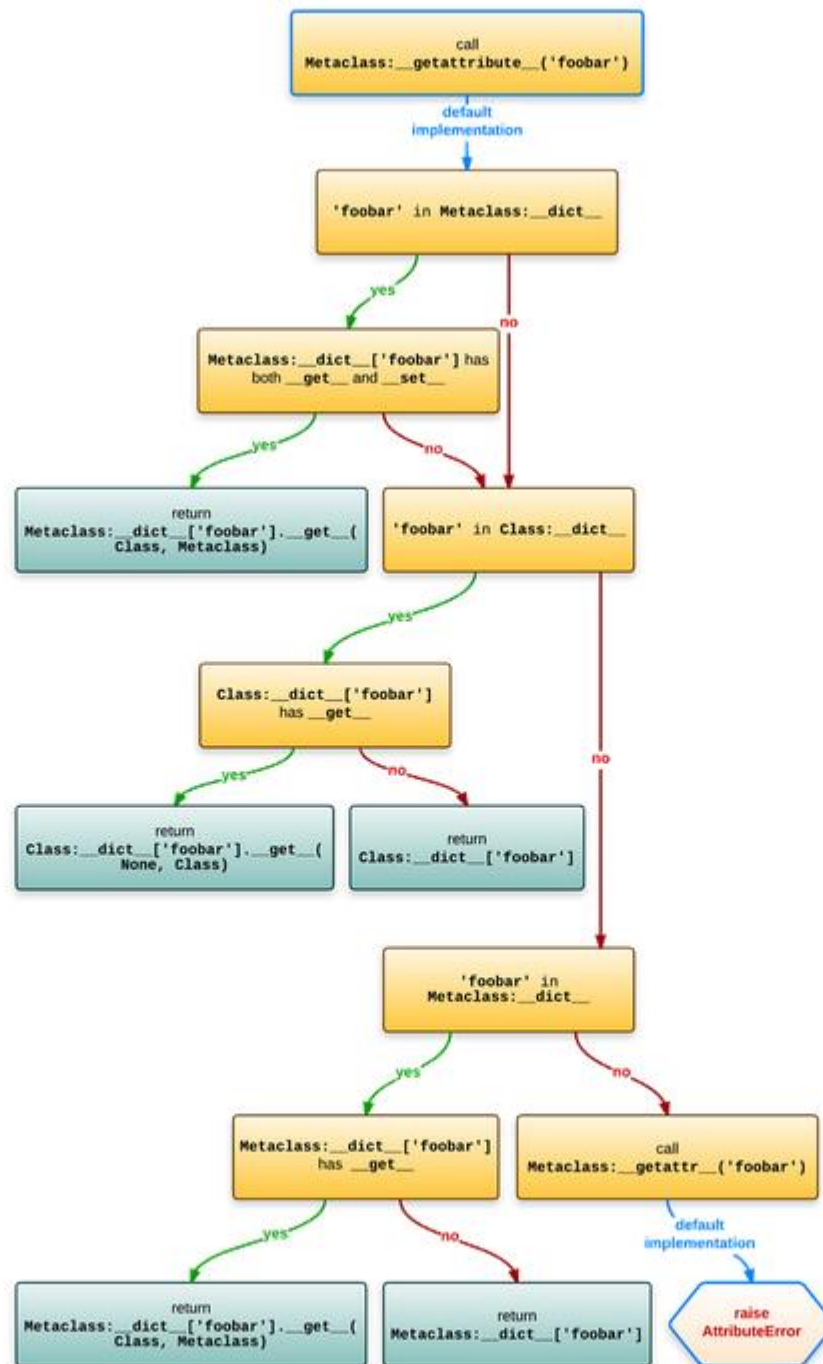
Pour les instances

- Toute la logique est implémentée dans `object.__getattr__`
 - Comme la recherche d'attributs est un appel implicite de méthodes spéciales, on saute l'instance est on cherche `__getattr__` dans la classe puis le long de l'arbre d'héritage
 - Si `__getattr__` est surchargée le long de l'arbre, il faut obligatoirement appeler `object.__getattr__` si on veut que la logique des descripteurs s'applique



Pour les classes

- Toute la logique est implémentée dans `type.__getattr__`
- Comme la recherche d'attributs est un appel implicite de méthodes spéciales, on saute l'instance est on cherche `__getattr__` dans la métaclasse puis le long de l'arbre d'héritage des métaclasses
- Si `__getattr__` est surchargée le long de l'arbre, il faut obligatoirement appeler `type.__getattr__` si on veut que la logique des descripteurs s'applique



Pour aller plus loin

<https://docs.python.org/3/howto/descriptor.html>

<http://sametmax2.com/les-descripteurs-en-python/>

Pour aller plus loin

Understanding Python metaclasses

<https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/>

David Beazley: Python 3 Metaprogramming

<http://www.dabeaz.com/py3meta/index.html>

La méthode `__new__`

Héritage des types builtins

- Utile lorsqu'on veut créer un objet qui ressemble à un builtin sans avoir à réimplémenter tous les comportements donnés par les builtins. Par exemple
 - Une liste d'un type donné
 - Ça ressemble à une liste builtin
 - Une température décimale avec son unité
 - Ça ressemble à un float

Héritage des types builtins

- Comme les mutables et les immuables ne sont pas initialisés au même moment, on n'en hérite pas de la même manière
 - Pour les mutables, tout se passe dans `__init__()`
 - Pour les immuables, tout se passe dans `__new__()`

Comment hériter d'un mutable ?

- Il suffit d'implémenter la méthode `__init__`
 - On a accès à self, on peut donc directement modifier l'objet
 - On peut également appeler la méthode `__init__` sur la super classe (recommandé)

```
# let's start with the easy case : mutable inheritance
class TypedList(list):
    def __init__(self, x, type=int):
        a = [type(e) for e in x]
        # Initialize self with the correctly types list
        # Could have used self.extend(a) instead
        list.__init__(self, a)

    # Modify a builtin behavior :
    # Positive indexing starts at 1
    def __getitem__(self, n):
        if n > 0:
            ind = n - 1
        elif n < 0:
            ind = n
        else:
            raise IndexError('does not support 0 index')
        return list.__getitem__(self, ind)

    # Create a new behavior
    def to_float(self):
        return [float(e) for e in self]
```

```
>>> a = TypedList([1, 2.0, 3.3])
>>> print(a)
[1, 2, 3]
>>> a[1]
1
>>> a[-1]
3
>>> a.to_float()
[1.0, 2.0, 3.0]
```


Comment hériter d'un immuable ?

- Le problème est que lorsque `__init__` est appelée, l'objet existe déjà
 - Or, on ne peut pas modifier un immuable
 - Comment faire alors ?
- `__new__()` est une méthode spéciale appelée avant de créer l'instance, donc avant `__init__()`
 - C'est le vrai constructeur
 - Il doit retourner l'instance

`__new__` est une méthode très spéciale

- Elle est statique, mais n'a pas besoin d'être explicitement déclarée comme telle avec `staticmethod(__new__)`
- Elle doit explicitement appeler `__new__` de la super classe (sinon, l'instance n'est pas créée)
- `__new__` doit retourner une instance de la classe sinon `__init__` n'est pas appelé sur l'instance

La méthode `__new__` ()

- `__new__` (`cls`, ...) reçoit comme arguments
 - La classe `cls` qui doit être instanciée
 - Les arguments passés au constructeur de la classe (lors de l'appel de la classe)
 - Ces arguments ne sont pas nécessairement utilisés
 - On ne peut pas dans `__new__` modifier les arguments qui seront passés à `__init__` (car c'est `type.__call__` qui les passe à `__init__`)
 - Il faut pour cela implémenter `__call__` dans une métaclasse

Deux problèmes concrets lorsqu'on hérite d'un immuable ?

- Problème 1
 - Je veux contrôler la valeur de mon objet à la création
 - Par exemple valider une plage de valeurs possibles ou modifier les valeurs passées au constructeur
 - Le constructeur `__new__` de ma classe va fixer la valeur (de `self`) et je ne pourrai plus la changer dans le `__init__`

```

class NegativeFloatError(Exception):
    pass

class PositiveFloat(float):
    def __new__(cls, val):
        if val >= 0:
            return float.__new__(cls, val)
        else:
            raise NegativeFloatError(
                f"cannot create a PositiveFloat for {val}")

# PositiveFloat support all float operations
>>> PositiveFloat(2) + PositiveFloat(10) + 4.2
16.2

# but it adds an extra check
>>> PositiveFloat(-1)
-----
NegativeFloatError                                Traceback (most recent call last)
...
NegativeFloatError: cannot create a PositiveFloat for -1

```

```

class CamelCaseString(str):
    """
    crée automatiquement un chaîne en camel case
    si elle est underscored: test_str => TestStr
    """
    def __new__(cls, my_str):
        no_underscore = " ".join(my_str.split('_'))
        camelCase = "".join(no_underscore.title().split())
        # Other option
        # camelCase = my_str.replace('_', ' ').title().replace(' ', '')
        return str.__new__(cls, camelCase)

c = CamelCaseString("a_test_string")

# CamelCaseString supports all string operations
>>> print(c)
'ATestString'

>>> c.endswith('ing')
True

>>> c + 'Spam'
'ATestStringSpam'

```

Deux problèmes concrets lorsqu'on hérite d'un immuable ?

- Problème 2
 - Je veux passer plus de paramètres que prévu dans le constructeur de la classe immuable héritée
 - On redéfinit `__new__` et `__init__` pour changer la signature du constructeur et permettre de passer d'autres paramètres
 - `__new__` retourne l'instance
 - `__init__` ajoute des attributs à la classe
 - On peut le faire directement dans `__new__`, mais c'est une bonne pratique de le faire dans `__init__`

```

class Temperature(float):
    Celcius = 'c'
    Kelvin = 'k'
    def __new__(cls, temp, unit=None):
        instance = float.__new__(cls, temp)
        return instance

    def __init__(self, temp, unit=None):
        if (unit == None) or (unit.lower() == Temperature.Celcius):
            self.unit = Temperature.Celcius
        elif unit.lower() == Temperature.Kelvin:
            self.unit = Temperature.Kelvin
        else:
            raise Exception('Unsupported temperature unit')

    def __repr__(self):
        symbol = '\u2103' if self.unit == Temperature.Celcius else
'\u212a'
        return f'{float.__repr__(self)}{symbol}'

    def __add__(self, obj):
        if self.unit == obj.unit:
            new_temp = float.__add__(self,obj)
            return Temperature(new_temp, self.unit)
        else:
            raise Exception('cannot add temperatures in different units')

```



```
a = Temperature(15)
b = Temperature(12)
c = Temperature(200, Temperature.Kelvin)
```

```
>>> a, b, c
(15.0°C, 12.0°C, 200.0K)
```

```
>>> a + b
27.0°C
```

```
>>> a + c
```

```
-----
Exception                                Traceback (most recent call last)
...
Exception: cannot add temperatures in different units
```

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__new__

Version simplifiée, la version
complète est dans la
discussion sur les métaclasses

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__new__

Implémentation par défaut
qui ne fait rien

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__new__

La vraie machinerie qui
alloue la mémoire et crée
les instances des classes est
dans `object.__new__`

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__new__

Implémenté en Python et
créé au chargement du
module

classe

__init__

__new__

Lorsqu'on appelle la
classe

```
>>> classe()
```

pour créer
l'instance, ça appelle
`type.__call__`

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

classe

__call__

__init__

__new__

__init__

__new__

type.__call__
appelle

classe.__new__

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

classe

__call__

__init__

__new__

__init__

__new__

`object.__new__`
crée réellement
l'instance, il faut
donc l'appeler
explicitement depuis
la classe, ou
implicitement en
appelant `__new__`
sur la super classe

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

classe

__call__

__init__

__new__

__init__

__new__

Si

`classe.__new__`
retourne une instance
de classe, alors

`type.__call__`
appelle

`classe.__init__`

En pratique, quand appelle-t-on __new__ et __init__ ?

- On initialise toujours avec __init__ si on n'a pas une bonne raison d'utiliser __new__
- On utilise __new__ lorsqu'on doit
 - initialiser un objet immuable
 - en effet, un immuable est créé par __new__ et ne peut plus être modifié par __init__
 - ou contrôler la création des instances
 - On peut aussi utiliser une métaclasse

Pour aller plus loin

- <https://docs.python.org/3/reference/datamodel.html#object.new>
- <http://sametmax2.com/la-difference-entre-new-et-init-en-python/>

Les métaclasses

Tout est un objet en Python

Mais tous les objets n'ont pas les mêmes propriétés

- Métaclasses
- Classes
- Instances

Quelle est la super classe de toutes les classes ?

`object` est la super classe de toutes les classes

métaclasse

classe

instance

`object`

```
class C:
    pass
>>> t = C.__bases__
>>> t
(<class 'object'>,)
>>> t[0].__bases__
()
>>>
object.__bases__
()
```

`object` est la super classe de toutes les classes

métaclasse

classe

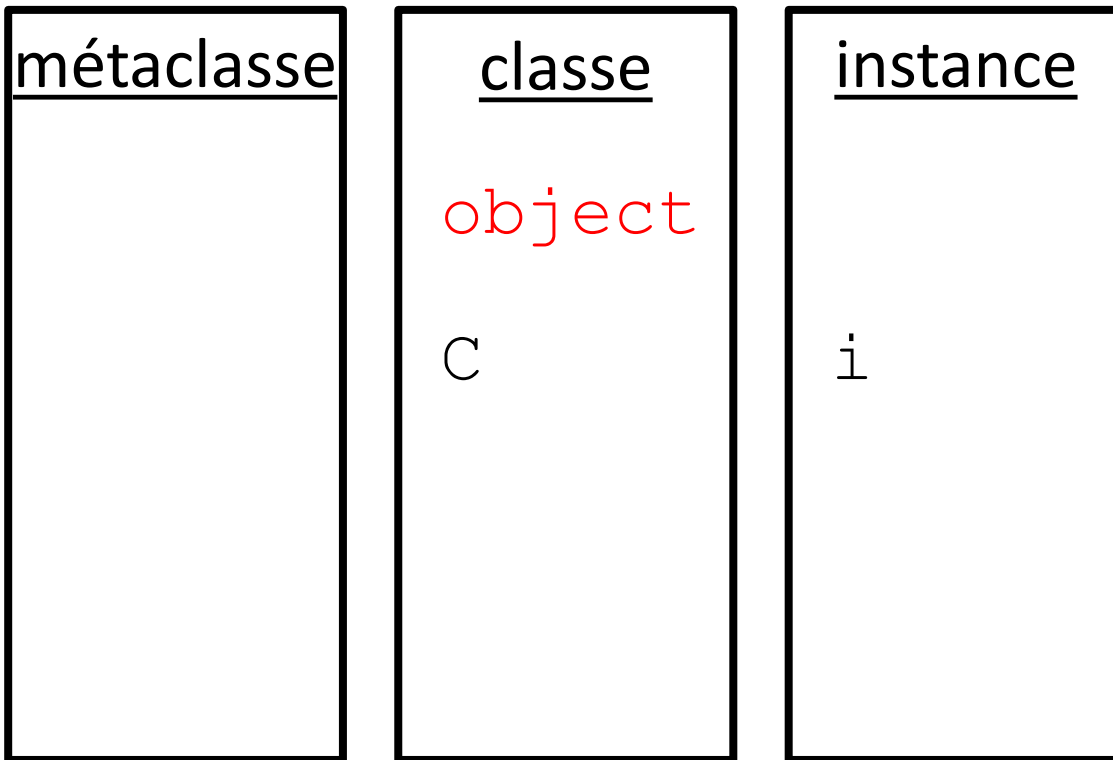
instance

`object`

```
>>> int.__bases__  
(<class' object'>,)  
>>> dict.__bases__  
(<class' object'>,)  
>>> str.__bases__  
(<class' object'>,)
```


Quelle différence entre classe et instance ?

Le type d'une classe est l'objet
`type`



```
class C:
    pass
>>> i = C()
>>> type(i)
<class '__main__.C'>
>>> type(C)
<class 'type'>
```

Le type d'une classe est l'objet `type`

<u>métaclasse</u>	<u>classe</u>	<u>instance</u>
	<code>object</code>	
	<code>C</code>	<code>i</code>
	<code>int</code>	<code>1</code>
	<code>str</code>	<code>'a'</code>

```
>>> type(1)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type('a')
<class 'str'>
>>> type(str)
<class 'type'>
```

Pourquoi le type de toutes les classes est l'objet `type` ?

- Le type est l'objet qui instancie
- L'objet `type` instancie toutes les classes
- C'est une métaclasse

Le type d'une classe est le type de sa super-classe

<u>métaclasse</u>	<u>classe</u>	<u>instance</u>
<code>type</code>	<code>object</code> <code>C</code> <code>int</code> <code>str</code>	<code>i</code> <code>1</code> <code>'a'</code>

```
>>> class C:
    pass
>>> class D(C): pass
>>> type(object)
<class 'type'>
>>> type(C)
<class 'type'>
>>> type(D)
<class 'type'>
```

- La métaclasse `type` instancie les classes
- Les classes instancient les instances
- Toutes les classes héritent de `object`

- On appelle *classe* ou *type* une instance de la métaclasse
- La classe a pour type l'objet métaclasse

- On appelle *instance* une instance d'une classe
- Une instance n'a pas pour type l'objet `type`, mais l'objet classe

- Les classes peuvent avoir des sous-classes, pas les instances
- Les classes peuvent avoir des instances, pas les instances

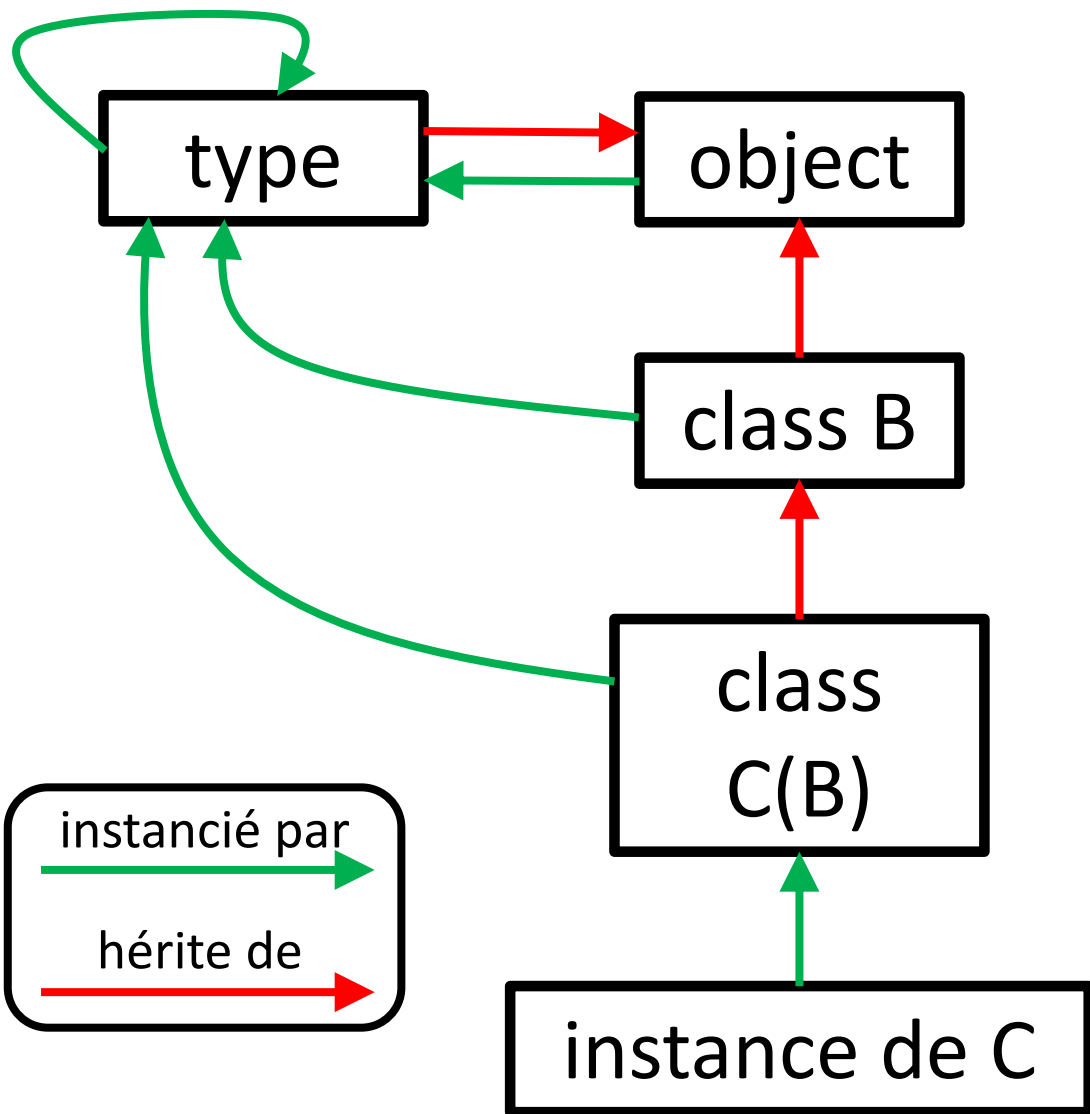
Quel lien entre `type` et `object` ?

<u>métaclasse</u>	<u>classe</u>	<u>instance</u>
type	object	
	C	i
	int	1
	str	'a'

```

>>> type(object)
<class 'type'>
>>> type.__bases__
(<class 'object'>,)
>>> type(type)
<class 'type'>

```



Peut-on écrire nos propres
métaclasses pour instancier les
classes ?

Oui !

Comment écrire une métaclasse ?

En C

- Contrôle total de la création d'objets

Comment écrire une métaclasse ?

En Python en créant une classe qui hérite de l'objet `type`

- Contrôle
 - pré-instanciation
 - initialisation de l'objet classe

Comment la métaclasse crée l'objet classe ?

Code du programmeur

```
class C(object):  
    lst = []  
    def __init__(self):  
        self.x = 10  
    def reset_x(self):  
        self.x = 0
```

Ce que fait l'interpréteur (simplifié)

```
lst = []  
def __init__(self):  
    self.x = 10  
def reset_x(self):  
    self.x = 0  
  
esp = {'__init__' : __init__,  
       'reset_x' : reset_x,  
       'lst' : []}  
  
C = type('C', (object,), esp)
```

Étapes de création d'une classe

- L'interpréteur va
 - Créer un espace de nommage (vide) avec la méthode `type.__prepare__` qui retourne un objet utilisé pour l'espace de nommage
 - Exécuter le bloc de code de la classe pour remplir l'espace de nommage
 - Créer l'objet classe (avec `type()`) en lui associant **une copie** de l'espace de nommage

Étapes de création d'une classe

- La création est un peu plus compliquée.
Tout est expliqué ici

<https://docs.python.org/3/reference/datamodel.html#customizing-class-creation>

Que fait l'appel

`type(name, bases, dict) ?`

- Appel de `__call__` sur l'objet `type`

- Le code de `__call__` est

```
cls = type.__new__(type, name, bases, dict)
type.__init__(cls, name, bases, dict)
```

`__new__` crée l'objet classe

- On peut donc faire des modifications avant la création (espace de nommage, super classes, etc.) ou changer l'objet retourné

`__init__` initialise l'objet classe

- On peut modifier l'objet classe après sa création

Attention : ici on parle de classes, pas d'instances !

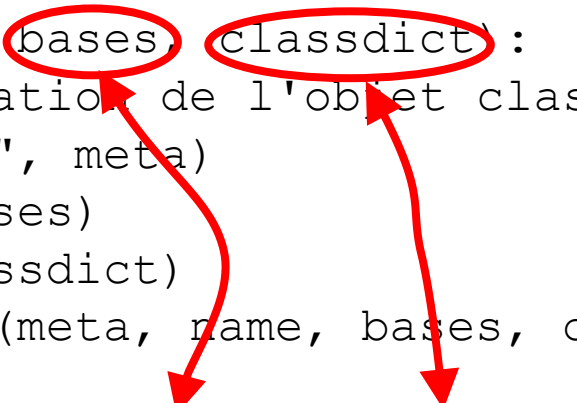
Alors `__new__` ou `__init__` ?

- Ce que je peux faire dans `__new__` et pas dans `__init__`
 - Retourner un autre objet
 - Changer l'arbre d'héritage
 - Changer l'espace de nommage avant la création de l'objet classe (plus facile qu'après)
- Souvent, c'est équivalent d'utiliser l'une ou l'autre, le choix est alors une question de goût
 - `__new__` marche dans tous les cas
 - `__init__` est plus simple

```
class MaMetaClasse(type):
    def __new__(meta, name, bases, classdict):
        print("Avant la creation de l'objet classe", name)
        print("metaclass :", meta)
        print("bases :", bases)
        print("dict :", classdict)
        return type.__new__(meta, name, bases, classdict)

    def __init__(classe, name, bases, classdict):
        type.__init__(classe, name, bases, classdict)
        print("Après la creation de la classe", name)
        print("classe :", classe)
        print("bases :", bases)
        print("dict :", classdict)

class C(metaclass=MaMetaClasse):
    x = 1
```



- Dans `__init__`, la classe est déjà créée, donc si on modifie `bases` ou `classdict` ça n'aura pas d'impact sur la classe, il faut le faire dans `__new__`

```
>>>
```

Avant la creation de l'objet classe C

```
metaclass : <class '__main__.MaMetaClasse'>
```

```
bases : ()
```

```
dict : {'x': 1, '__qualname__': 'C', '__module__': '__main__'}
```

Apres la creation de la classe C

```
classe : <class '__main__.C'>
```

```
bases : ()
```

```
dict : {'x': 1, '__qualname__': 'C', '__module__': '__main__'}
```


Exemple de métaclasse

- Ajouter la classe `BaseOfAll` comme super classe de toutes les classes
- Créer des noms de méthodes en CamelCase automatiquement

```

class UpperAttrMetaclass(type):
    def __new__(meta, clsname, bases, dct):
        new_dct = {}
        for name, val in dct.items():
            if not name.startswith("__"):
                new_dct[name] = val
                camel_name = (name.replace("_", " ")
                              .title().replace(" ", ""))
                new_dct[camel_name] = val
            else:
                new_dct[name] = val

        bases = (BaseOfAll,)
        return type.__new__(meta, clsname, bases, new_dct)

```

```

class BaseOfAll:
    def common_func(self):
        return "in common_func"

```

```

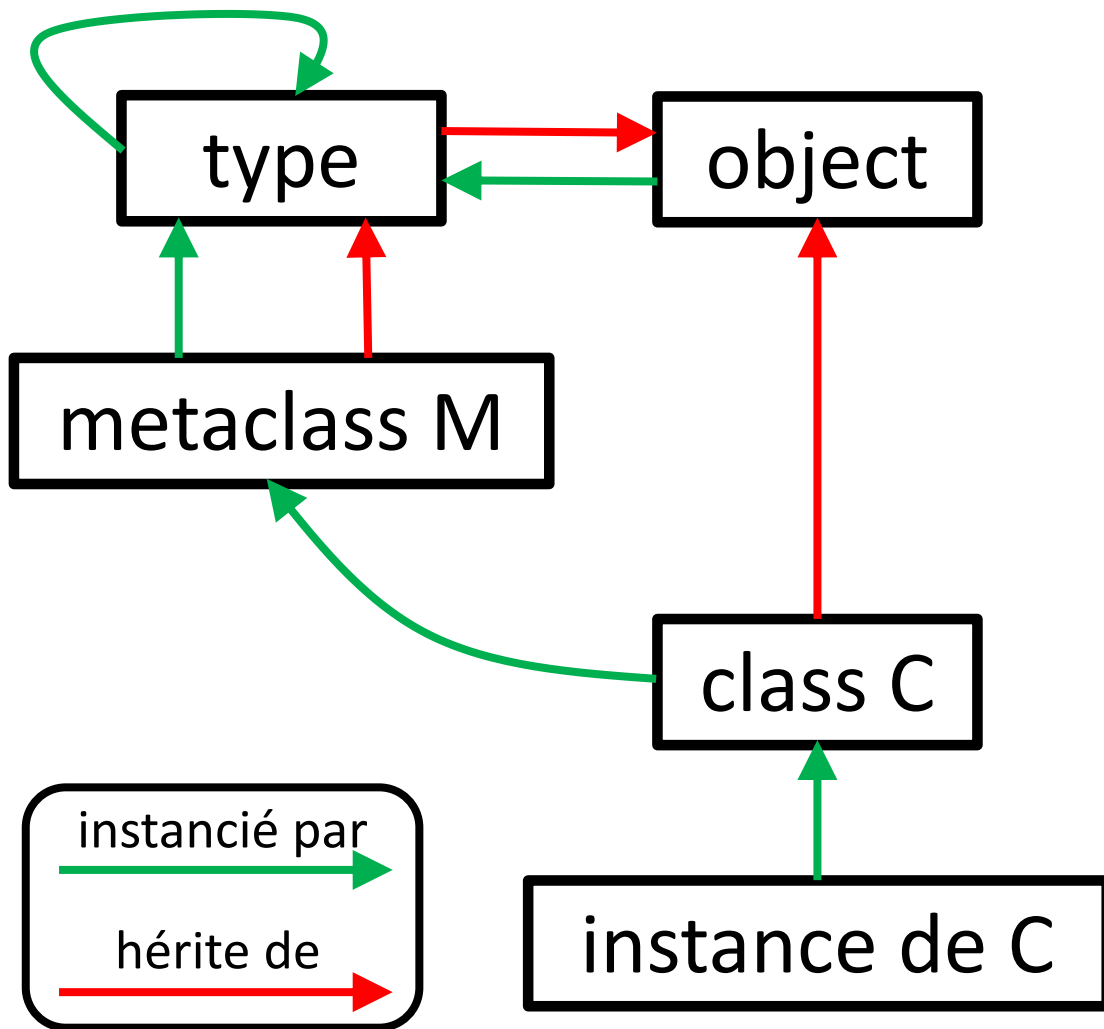
class C(metaclass=UpperAttrMetaclass):
    def func_snake_name(self):
        print('in func')

```

```

print(vars(C), C.__bases__, C().common_func(), sep="\n")

```



Quelle différence entre un décorateur de classe et une métaclasse

- On peut faire essentiellement la même chose avec l'un ou l'autre
 - Dans certains cas, c'est plus facile avec un décorateur, dans d'autres avec une métaclasse
- La différence la plus importante est
 - Le décorateur n'est pas hérité par les sous classes d'une classe décorée
 - La métaclasse est héritée, la métaclasse d'une sous classe est celle de sa super classe

```
class meta(type):
    def __new__(meta, name, bases, classdict):
        return type.__new__(meta, name, bases, classdict)

class A(metaclass=meta):
    pass

class B(A):
    pass

print(f"type de A {type(A)}\ntype de A() {type(A())}\n"
      "type de B {type(B)}\ntype de B() {type(B())}")
```

Héritage, métaclasse et recherche d'attributs

- On cherche les attributs le long de l'arbre d'héritage, mais est-ce que la métaclasse entre quelque part dans l'algorithme de recherche des attributs?
 - C'est simple ! (ou presque)
 - Une classe est une instance de métaclasse, il y a donc une relation d'héritage entre la classe et sa métaclasse, mais cette relation n'est pas propagée aux instances de la classe

Héritage, métaclasse et recherche d'attributs

- `instance.x` sera cherché le long de l'arbre d'héritage mais pas dans les métaclasse
- `class.x` sera cherché le long de l'arbre d'héritage puis dans la métaclasse

```
class meta(type):
    x = y = z = "meta"
    def __new__(meta, name, bases, classdict):
        return type.__new__(meta, name, bases, classdict)
```

```
class A(metaclass=meta):
    x = y = "A"
    pass
```

```
class B(A):
    x = "B"
    pass
```

```
print(f"B.x: {B.x}\nB.y: {B.y}\nB.z: {B.z}\n")
print(f"B().x: {B().x}\nB().y: {B().y}")
print(f"B().z: {B().z}\n") # AttributeError
>>>
```

```
B.x: B
```

```
B.y: A
```

```
B.z: meta
```

```
B().x: B
```

```
B().y: A
```

```
Traceback (most recent call last):
```

```
AttributeError: 'B' object has no attribute 'z'
```


Héritage, métaclasse et recherche d'attributs

- Il y a une exception pour l'appel implicite des méthodes spéciales `__X__`
 - Une méthode spéciale n'est jamais résolue dans l'espace de nommage d'une instance lors d'un appel implicite
 - Un appel implicite sur une instance de classe est résolu directement dans la classe
 - Un appel implicite sur une instance de métaclasse (c'est-à-dire une classe) est résolu directement dans la métaclasse

```
print("creation de la metaclassse")
class MaMetaClasse(type):
    def __new__(meta, name, bases, classdict):
        print("__new__ de la metaclassse")
        return type.__new__(meta, name, bases, classdict)

    def __init__(cls, name, bases, classdict):
        type.__init__(cls, name, bases, classdict)
        print("__init__ de la metaclassse")

    def __call__(cls, *args, **kwargs):
        print("__call__ de la metaclassse")
        return type.__call__(cls, *args, **kwargs)
```

```
print("creation de l'objet classe")
class C(metaclass=MaMetaClasse):
    def __new__(cls):
        print("__new__ de la classe")
        return object.__new__(cls)

    def __init__(self):
        print("__init__ de la classe")

    def __call__(self):
        print("__call__ de la classe")

print("creation de l'instance")
c = C()
print("appel de l'instance")
c()
```

creation de la metaclassse
creation de l'objet classe
__new__ de la metaclassse
__init__ de la metaclassse
creation de l'instance
__call__ de la metaclassse
__new__ de la classe
__init__ de la classe
appel de l'instance
__call__ de la classe

Héritage, métaclasse et recherche d'attributs

- Et il y a une dernière exception à cette exception, `__new__` étant une méthode statique, elle est appelée sur l'objet qui veut créer une instance
 - Sur la classe si on veut créer une instance de classe
 - Sur la métaclasse si on veut créer une classe

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__new__

__init__

__new__

Lorsqu'on démarre
l'interpréteur, les objets
`type` et `object`
(implémentés en C pour
CPython) sont créés

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__init__

__new__

__new__

Implémentation par défaut
qui ne fait rien

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__init__

__new__

__new__

La vraie machinerie qui
alloue la mémoire et crée
les objets `type.__new__`
pour les classes et
`object.__new__` pour
les instances

Implémenté
en C et créé
au démarrage
de
l'interpréteur

type

object

__call__

__init__

__new__

__init__

__new__

Implémenté en Python et
créé au chargement du
module

métaclasse

__init__

__new__

On n'a pas le contrôle
en Python sur
l'instanciation de
l'objet métaclasse, il
est créé
automatiquement au
chargement du
module

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

__call__

__init__

__new__

__init__

__new__

__init__

__new__

Création de l'objet
classe au
chargement du
module, c'est
`type.__call__`
qui appelle
`__new__` et
`__init__`

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

__call__

__init__

__init__

__init__

__new__

__new__

__new__

Création de l'objet
classe

C'est `type.__new__` qui crée
réellement l'objet classe, il faut
donc l'appeler explicitement
depuis la métaclasse

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

__call__

__init__

__init__

__init__

__new__

__new__

__new__

Création de l'objet
classe

`type.__init__` est la
méthode par défaut qui ne fait
rien, ça n'est pas nécessaire de
l'appeler dans la métaclasse

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

classe

__call__

__call__

__call__

__init__

__init__

__init__

__init__

__new__

__new__

__new__

__new__

Création de
l'instance
par l'appel
de la classe

Option 1, `__call__` appelle directement
`__new__` et `__init__`

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

classe

__call__

__call__

__call__

__init__

__init__

__init__

__init__

__new__

__new__

__new__

__new__

Création de
l'instance
par l'appel
de la classe

C'est `object.__new__` qui crée
réellement l'instance, il faut
donc l'appeler explicitement
depuis la classe

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

classe

__call__

__call__

__call__

__init__

__init__

__init__

__init__

__new__

__new__

__new__

__new__

Création de
l'instance
par l'appel
de la classe

Option 1, `__call__` appelle directement
`__new__` et `__init__`

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

classe

__call__

__call__

__call__

__init__

__init__

__init__

__init__

__new__

__new__

__new__

__new__

Création de
l'instance
par l'appel
de la classe

Option 2, `__call__` appelle `type.__call__` qui
se charge d'appeler `__new__` et `__init__`

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

classe

__call__

__call__

__call__

__init__

__init__

__init__

__init__

__new__

__new__

__new__

__new__

Création de
l'instance
par l'appel
de la classe

C'est `object.__new__` qui crée
réellement l'instance, il faut
donc l'appeler explicitement
depuis la classe

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

type

object

métaclasse

classe

__call__

__call__

__call__

__init__

__init__

__init__

__init__

__new__

__new__

__new__

__new__

Création de
l'instance
par l'appel
de la classe

Si `object.__new__` crée une
instance de la classe alors
`__init__` est appelée

Implémenté
en C et créé
au démarrage
de
l'interpréteur

Implémenté en Python et
créé au chargement du
module

Créé à l'appel
de la classe

type

object

métaclasse

classe

instance

__call__

__call__

__call__

__init__

__init__

__init__

__init__

__new__

__new__

__new__

__new__

__call__ de la classe est appelé à l'appelle de
l'instance

`__call__` métaclasse et classe

- Comment la méthode `type.__call__` sait s'il faut appeler les méthodes `__new__` et `__init__` sur la métaclasse ou sur la classe
 - `__call__` va appeler les méthodes sur l'objet passé comme premier argument

```

class Metaclassse(type):
    def __call__(cls, *t, **d):
        print(f"in Metaclassse __call__ {cls}\n{t}\n{d}")
        return type.__call__(cls, *t, **d)

    def __new__(meta, *t, **d):
        print(f"in Metaclassse.__new__ {meta}\n{t}\n{d}")
        return type.__new__(meta, *t, **d)

    def __init__(self, *t, **d):
        print(f"in Metaclassse.__init__ {self}\n{t}\n{d}")
        type.__init__(self, *t, **d)

class Classe(metaclass = Metaclassse):
    def __new__(cls, *t, **d):
        print(f"in Classe __new__ {cls}\n{t}\n{d}")
        return object.__new__(cls)

    def __init__(self, *t, **d):
        print(f"in Classe __init__ {self}\n{t}\n{d}")

```

```

in Metaclass.__new__ <class '__main__.Metaclass'>
('Classe', (), {'__module__': '__main__', '__qualname__': 'Classe',
 '__new__': <function Classe.__new__ at 0x000001DA54EF17B8>,
 '__init__': <function Classe.__init__ at 0x000001DA54EF1840>})
{}

in Metaclass.__init__ <class '__main__.Classe'>
('Classe', (), {'__module__': '__main__', '__qualname__': 'Classe',
 '__new__': <function Classe.__new__ at 0x000001DA54EF17B8>,
 '__init__': <function Classe.__init__ at 0x000001DA54EF1840>})
{}

>>> Classe(1, 2)

in Metaclass __call__ <class '__main__.Classe'>
(1, 2)
{}
in Classe __new__ <class '__main__.Classe'>
(1, 2)
{}
in Classe __init__ <__main__.Classe object at 0x000001DA54EEE710>
(1, 2)
{}
<__main__.Classe object at 0x000001DA54EEE710>

```

__call__ et métaclassse

- Ça permet de contrôler le processus d'instanciation des instances de classes avant même l'appel de __new__
 - C'est, par exemple, utile pour créer des singletons
 - Ça permet d'hériter un comportement dans les sous-classes
 - À la différence des décorateurs de classe

```

class Singleton(type):
    _instances = {}
    # set it to True to re-initialize the created instance
    # each time we call the class
    init_singleton_on_call = False

    def __call__(cls, *ta, **ka):
        print("in Singleton __call__")
        if cls not in Singleton._instances:
            Singleton._instances[cls] = type.__call__(cls, *ta,
**ka)

            elif cls.init_singleton_on_call:
                Singleton._instances[cls].__init__(*ta, **ka)
        return Singleton._instances[cls]

class SingleInstance(metaclass = Singleton):
    def __new__(cls, *ta, **ka):
        print("in SingleInstance __new__")
        return object.__new__(cls)

    def __init__(self, *ta, **ka):
        print("in SingleInstance __init__")

```



```
class SubSingleInstance(SingleInstance):
    def __new__(cls, *ta, **ka):
        print("in SubSingleInstance __new__")
        return object.__new__(cls)

    def __init__(self, *ta, **ka):
        print("in SubSingleInstance __init__")

s = SingleInstance()
s2 = SingleInstance()
print(s is s2)

sub_s = SubSingleInstance()
sub_s2 = SubSingleInstance()
print(sub_s is sub_s2)
```

```
>>>
in Singleton __call__
in SingleInstance __new__
in SingleInstance __init__
in Singleton __call__
True
in Singleton __call__
in SubSingleInstance __new__
in SubSingleInstance __init__
in Singleton __call__
True
```

Métaclasses en Python 2.x

- En Python 2.x
 - Tout ce que l'on a décrit sur les métaclasses ne fonctionne qu'avec les classes new-style
- En Python 2.x, la syntaxe pour définir une metaclasse est différente de Python 3.x

```
class C(object):  
    __metaclass__ = MaMetaClasse  
    x = 1
```

Pour aller plus loin

- David Beazley: Python 3 Metaprogramming
 - <http://www.dabeaz.com/py3meta/index.html>
- <https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/>
- <https://zestedesavoir.com/tutoriels/954/notions-de-python-avancees/9-metaclasses/>
- <https://openclassrooms.com/courses/apprenez-a-programmer-en-python/les-metaclasses>
- <http://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example/>
- <https://docs.python.org/3/reference/datamodel.html#customizing-class-creation>
- <https://eli.thegreenplace.net/2012/03/23/python-internals-how-callables-work>
- <https://eli.thegreenplace.net/2012/04/16/python-object-creation-sequence>

Et maintenant ?

Pour aller plus loin

- Python : des fondamentaux à l'utilisation du langage
 - Python intermédiaire
- Python : utilisation avancée
 - Python avancé
- Pour devenir expert, beaucoup de pratique...
 - Et quelques sources très intéressantes

Pour aller plus loin

- David Beazley
 - <http://www.dabeaz.com/talks.html>
 - <https://dabeaz-course.github.io/practical-python/>
- PyCon : la conférence sur Python
 - <http://www.pycon.org>
 - <http://pyvideo.org/>

Où est le code source de Python ?

- Le code source est sous GIT
 - <https://github.com/python/cpython>
 - <https://docs.python.org/devguide/setup.html#directory-structure>
- Le guide des développeurs
 - <https://docs.python.org/devguide/index.html>

Où est le code source de Python ?

- Yet another guided tour of CPython, par Guido
 - <https://paper.dropbox.com/doc/Yet-another-guided-tour-of-CPython-XY7KgFGn88zMNivGJ4Jzv>