La dynamique des attributs

Emmenez-moi au bout de l'attr

Antoine (entwanne) Rozo

La

CONNAISSANCE

POUR TOUS & SANS PÉPINS

Nos contenus sont gratuits

Et notre mascotte est pulpeuse !

Notre communauté est sympathique et disponible

Zeste de savoir

zestedesavoir.com

La dynamique des attributs

- Comprendre le stockage et l'accès aux attributs en Python
- Mettre en place des attributs dynamiques sur nos objets
- https://zestedesavoir.com/tutoriels/954/notions-de-python-avancees/

Attributs en Python

Attributs en Python

- Les attributs permettent d'associer des données à un objet

```
dana.attr = 10

dana.attr

del dana.attr
```

Fonctions `setattr`, `getattr` et `delattr`

- Ces opérations élémentaires correspondent à des fonctions Python

```
setattr(dana, 'foo', 'bar')

getattr(dana, 'foo')

delattr(dana, 'foo')
```

Fonction `hasattr`

- Une fonction supplémentaire permet de tester la présence d'un attribut

```
hasattr(dana, 'foo')
```

Stockage des attributs

- Les objets Python possèdent un attribut spécial, `__dict__`
- Il s'agit d'un dictionnaire qui stocke toutes les données de l'objet

```
dana.__dict__
```

- Ce dictionnaire est utilisé lors de l'accès à un attribut

```
dana.__dict__['foo']
```

Method Resolution Order (*MRO*)

Method Resolution Order

- L'accès à un attribut ne se contente pas d'explorer le `__dict__` de l'objet
- Sont aussi analysés celui du type, et de tous les types parents
- L'ordre d'évaluation des types est défini par le *MRO*

Method Resolution Order

```
class A:
    foo = 'A.foo'
    bar = 'A.bar'
    baz = 'A.baz'

class B(A):
    bar = 'B.bar'

b = B()
b.baz = 'b.baz'

b.foo, b.bar, b.baz
```

Method Resolution Order

- On peut connaître le *MRO* d'une classe en faisant appel à sa méthode `mro`

```
B.mro()
```

Method Resolution Order

- Celui-ci est surtout utile lors d'héritages multiples, il se base sur l'algorithme C3
- Il permet de linéariser la hiérarchie des classes parentes

```
class P1:
    foo = 'P1.foo'

class P2:
    foo = 'P2.foo'
    bar = 'P2.bar'

class C(P1, P2):
    pass

C.mro()

C.foo, C.bar
```

Method Resolution Order

```
object.mro()
```

```
class A: pass
```

```
A.mro()
```

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

```
B.mro()
```

```
class C: pass
```

```
C.mro()
```

Method Resolution Order

```
class D(A, C): pass

D.mro()

class E(B, C): pass

E.mro()

class F(D, E): pass

F.mro()

class G(E, D): pass

G.mro()
```

MRO de l'impossible

```
class H(A, B): pass

class H(B, A): pass

H.mro()
```

Attributs et méthodes spéciales

__getattr__ et __getattribute__

- Des méthodes spéciales sont impliquées dans la recherche des attributs d'un objet
- Lors de l'accès à un attribut, la méthode `__getattribute__` est appelée
- C'est celle-ci qui s'occupe par défaut d'explorer les dictionnaires d'attributs

```
def __getattribute__(self, name):
    if name in self.__dict__:
        return self.__dict__[name]
    for cls in type(self).mro():
        if name in cls.__dict__:
            return cls.__dict__[name]
    raise AttributeError
```

__getattr__ et __getattribute__

```
class Temperature:
    def __init__(self, celsius=0):
        self.celsius = celsius

    def __getattribute__(self, name):
        print(f"Récupération de l'attribut {name}")
        if name == 'fahrenheit':
            return self.celsius * 1.8 + 32
        return super().__getattribute__(name)

t = Temperature(25)
t.celsius

t.fahrenheit
```

Pièges de `__getattribute__`

- Attention aux cas de récursions infinies

```
class WTF:
    def __getattribute__(self, name):
        return self.__dict__[name]

wtf = WTF()
wtf.foo = 0

wtf.foo
```

__getattr__ et __getattribute__

- __getattr__ est appelée lorsqu'un attribut n'est pas trouvé par __getattribute__
- Elle permet plus facilement de gérer des attributs dynamiques en plus des existants

```
class Temperature:
    def __init__(self, celsius=0):
        self.celsius = celsius

    def __getattr__(self, name):
        if name == 'fahrenheit':
            return self.celsius * 1.8 + 32
        raise AttributeError(name)

t = Temperature(25)
t.celsius

t.fahrenheit
```

`__setattr__` et `__delattr__`

- Ces méthodes sont appelées respectivement pour l'écriture et la suppression d'un attribut
- Elles sont appelées dans tous les cas, pour tous les attributs

__setattr__ et __delattr__

```
class Temperature:
    def __init__(self, celsius=0):
        self.celsius = celsius

    def __getattr__(self, name):
        if name == 'fahrenheit':
            return self.celsius * 1.8 + 32
        raise AttributeError(name)

    def __setattr__(self, name, value):
        if name == 'fahrenheit':
            self.celsius = (value - 32) / 1.8
        else:
            super().__setattr__(name, value)

t = Temperature()
t.fahrenheit = 100
t.celsius
```

Pièges de `__setattr__`

- Attention encore aux récursions infinies

```
class WTF:
    def __setattr__(self, name, value):
        super().__setattr__(name, value)
        self.last_attribute_modified = name

wtf = WTF()
wtf.foo = 0
```

Pièges de __setattr__

- Et aux appels par l'initialiseur

```
class WTF:
    def __init__(self, path, prefix=''):
        self.path = path
        self.prefix = prefix

    def __setattr__(self, name, value):
        if name == 'path':
            self.path = value + self.suffix
        else:
            super().__setattr__(self, name, value)

wtf = WTF('foo', '/tmp/')
```

Attributs et méthodes spéciales

- En raison des potentiels bugs décrits précédemment, évitez au maximum d'avoir recours à ces méthodes
- Elles sont de plus complexes à utiliser car nécessitent de traiter tous les attributs un à un
- Heureusement Python nous offre d'autres facilités pour gérer des attributs dynamiques

Propriétés

Propriétés

- Les propriétés permettent de simplifier l'usage d'attributs dynamiques
- Elles associent des fonctions de récupération, de modification et de suppression à un nom d'attribut
- On associe une propriété à un nom d'attribut en la définissant comme attribut de classe

Propriétés

```
class Temperature:
    def __init__(self, celsius=0):
        self.celsius = celsius

    def _get_fahrenheit(self):
        return self.celsius * 1.8 + 32

    def _set_fahrenheit(self, value):
        self.celsius = (value - 32) / 1.8

    fahrenheit = property(_get_fahrenheit, _set_fahrenheit)

t = Temperature()
t.fahrenheit = 100
t.celsius
```

Décorateur @property

- property peut aussi s'utiliser comme un décorateur
- Le nom de l'attribut découle alors du nom du *getter*

```
class Temperature:
    def __init__(self, celsius=0):
        self.celsius = celsius

    @property
    def fahrenheit(self):
        return self.celsius * 1.8 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) / 1.8

t = Temperature()
t.fahrenheit = 100
t.celsius
```

Propriétés en lecture seule

- Le *getter* peut être implémenté sans le *setter*

```
class Rect:
    def __init__(self, width, height):
        self.width = width
        self.height = height
    @property
    def perimeter(self):
        return 2 * (self.width + self.height)
    @property
    def area(self):
        return self.width * self.height

rect = Rect(10, 20)

rect.perimeter

rect.area
```

Descripteurs

Descripteurs

- Les propriétés sont un sous-ensemble des descripteurs
- Un descripteur est un objet spécial qui permet de régir le comportement d'un attribut
- Il possède pour cela des méthodes `__get__`, `__set__` et `__delete__`

Descripteurs

- Le descripteur est instancié une seule fois pour toute la classe
- Ses méthodes spéciales sont appelées lors des différents accès à l'attribut
- L'objet duquel on accède à l'attribut est alors passé en paramètre

Descripteurs

```
class Fahrenheit:
    def __get__(self, instance, owner):
        return instance.celsius * 1.8 + 32

    def __set__(self, instance, value):
        instance.celsius = (value - 32) / 1.8

class Temperature:
    def __init__(self, celsius=0):
        self.celsius = 0

    fahrenheit = Fahrenheit()

t = Temperature()
t.fahrenheit = 100
t.celsius
```

Méthode `__get__` des descripteurs

- Quel est donc ce paramètre `owner` de la méthode `__get__` ?
- Un descripteur peut-être récupéré depuis la classe et non depuis une instance de cette classe
- Dans ce cas, le paramètre `instance` vaudra `None`, et `owner` référence toujours la classe utilisée

Méthode __get__ des descripteurs

```
class Descriptor:
    def __get__(self, instance, owner):
        if instance is None:
            return f'Attribute of class {owner}'
        return f'Attribute of {instance}'

class C:
    attr = Descriptor()

C.attr

obj = C()
obj.attr
```

Descripteurs

- Ce comportement n'est valable que pour le `__get__`
- En effet, la redéfinition et la suppression de l'attribut de classe doivent toujours être possibles

```
C.attr = 'foo'

del C.attr
```

Méthode __set_name__

- Depuis Python 3.6, les descripteurs peuvent aussi comporter une méthode __set_name__ appelée lorsqu'ils sont définis dans une classe

```
class cachedescriptor:
    def __init__(self, func):
        self.func = func

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, inst, owner):
        if inst is None:
            return self
        if self.name not in inst.__dict__:
            inst.__dict__[self.name] = self.func(inst)
        return inst.__dict__[self.name]
```

Méthode __set_name__

```
class Calculation:
    @cachedescriptor
    def result(self):
        print('Complex calculation')
        ...
        return 0

calc = Calculation()

calc.result
```

Propriétés

- Implémentation simple des propriétés (ne gère pas l'utilisation en décorateurs)

```
class my_property:
    def __init__(self, fget, fset, fdel):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
    def __get__(self, instance, owner):
        return self.fget(instance)
    def __set__(self, instance, value):
        return self.fset(instance, value)
    def __delete__(self, instance):
        return self.fdel(instance)
```

Méthodes

Méthodes

- Derrière leur apparente simplicité, les méthodes sont en fait des descripteurs
- C'est ce qui explique la différence entre méthodes et *bound methods*

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

C.method

c = C()
c.method
```

Méthodes

- Une méthode est en alors un descripteur autour d'une fonction
- Ce descripteur réagit différemment suivant si la méthode est accédée depuis la classe ou l'une de ses instances

```
from functools import partial

class Method:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        if instance is None:
            return self.func
        return partial(self.func, instance)
```

Méthodes

- Les méthodes de classe fonctionnent de la même manière en utilisant l'`owner`

```
class ClassMethod:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        return partial(self.func, owner)
```

- Les méthodes statiques sont les plus simples et ne dépendent d'aucun descripteur

```
class StaticMethod:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        return self.func
```

Slots

Slots

- Tous les objets ne possèdent pas de `__dict__`
- Il est possible d'optimiser le stockage des attributs en définissant des slots au niveau de la classe
- Cela évite l'instanciation d'un dictionnaire mais empêche de définir des attributs non déclarés

```
class Point:
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(3, 4)
p.x, p.y

p.z = 1
```

Slots

- Les classes utilisant des slots restent compatibles avec les mécanismes d'attributs dynamiques

```
class Point:
    __slots__ = ('x', 'y')

    def __init__(self, x, y):
        self.x = x
        self.y = y

    @property
    def distance(self):
        return (self.x**2 + self.y**2)**0.5

p = Point(3, 4)
p.distance
```

Conclusion

Python 3.7 : Module et `__getattr__`

- Depuis Python 3.7, les modules peuvent aussi définir une méthode spéciale `__getattr__`
- Ils permettent plus facilement de gérer des attributs dynamiques au niveau d'un module

Conclusion

- *Complex is better than complicated*
- *Slides* : https://entwanne.github.io/presentation_attributs_dynamiques_python/pres.slides.html
- https://zestedesavoir.com/tutoriels/954/notions-de-python-avancees/

Antoine (entwanne) Rozo

Questions ?