Un descripteur est une classe qu’on instancie comme attribut d’une autre classe pour faire office de setter et de getter sur cet attribut. Le descripteur doit implémenter les méthodes \_\_get\_\_ et \_\_set\_\_ qui seront exécutées quand on essaye d’assigner ou lire l’attribut. Respecter cette signature, c’est adopter ce qu’on appelle pompeusement le “descriptor protocol”.

Si le principe vous rappelle les [propriétés](http://docs.python.org/2/library/functions.html#property), c’est normal, les propriétés sont implémentées en utilisant des descripteurs.

Exemple balot et complètement arbitraire :

|  |
| --- |
| **class** JeSuisUnDescripteurEtJeVousEmmerde(object):    *# les noms des attributs sont des conventions*  **def** \_\_get\_\_(self, obj, objtype):  **return** obj, objtype    **def** \_\_set\_\_(self, obj, value):  **print** obj, value      **class** JeSuisUneClasseNormaleEtJeVousAime(object):    ze\_attribute = JeSuisUnDescripteurEtJeVousEmmerde()      >>> objet\_affecteux = JeSuisUneClasseNormaleEtJeVousAime()  >>> **print** objet\_affecteux.ze\_attribute  <\_\_main\_\_.JeSuisUneClasseNormaleEtJeVousAime object at 0x1cb20d0> <**class** '\_\_main\_\_.JeSuisUneClasseNormaleEtJeVousAime'>  >>> objet\_affecteux.ze\_attribute = 'dtc '  <\_\_main\_\_.JeSuisUneClasseNormaleEtJeVousAime object at 0x1cedf10> dtc |

Vous noterez que obj est donc toujours l’instance de l’objet qui possède l’attribut sur lequel on agit. Dans \_\_get\_\_, objtype est la classe de cet objet. Dans \_\_set\_\_, value est la nouvelle valeur qu’on assigne à l’objet.

Vous allez me dire: pourquoi utiliser les descriptors plutôt que les properties ?

D’abord, les descripteurs sont des unités de code indépendantes. Vous pouvez faire un module avec vos descripteurs, et les distribuer en tant que lib. Donc c’est réutilisable. Ensuite, vous n’êtes pas limités à votre méthode en cours, vous avez accès à tout l’arsenal de la programmation OO.

Par exemple, si vous pouvez faire un descriptor d’alerte, qui envoie un signal à tous les abonnés pour cette valeur:

|  |
| --- |
| **class** SignalDescriptor(object):    abonnements = {}    @classmethod  **def** previens\_moi(cls, obj, attr, callback):  cls.abonnements.setdefault(obj, {}).setdefault(attr, set()).add(callback)    **def** \_\_init\_\_(self, nom, valeur\_initiale=None):  self.nom = nom  self.valeur = valeur\_initiale    **def** \_\_get\_\_(self, obj, objtype):  **for** callback **in** self.abonnements.get(obj, {}).get(self.nom, ()):  callback('get', obj, self.nom, self.valeur)  **return** self.valeur    **def** \_\_set\_\_(self, obj, valeur):  **for** callback **in** self.abonnements.get(obj, {}).get(self.nom, ()):  callback('set', obj, self.nom, self.valeur, valeur)  self.valeur = valeur |

Et voilà, vous pouvez distribuer ça sur Github, c’est plug and play.

Par exemple, pour créer un objet Joueur sur lequel on veut monitorer le nombre de crédits :

|  |
| --- |
| **class** Joueur(object):    credits = SignalDescriptor("credits", 0) |

On l’utilise normalement:

|  |
| --- |
| >>> j = Joueur()  >>> j.credits  0  >>> j.credits = 15  >>> j.credits  15  >>> j.credits += 5  >>> j.credits  20 |

Mais si on rajoute un abonné :

|  |
| --- |
| **def** monitorer\_credits(action, obj, attribut, valeur\_actuelle, nouvelle\_valeur=None):    **if** action == 'set':  **print** "Les crédits ont changé:"  **else**:  **print** "Les crédits ont été consultés:"  **print** action, obj, attribut, valeur\_actuelle, nouvelle\_valeur    >>> SignalDescriptor.previens\_moi(j, 'credits', monitorer\_credits) |

Alors à chaque action sur les crédits, tous les abonnés sont appelés :

|  |
| --- |
| >>> j.credits  Les crédits ont été consultés:  get <\_\_main\_\_.Joueur object at 0x1f6b190> credits 20 None  20  >>> j.credits = -20  Les crédits ont changé:  set <\_\_main\_\_.Joueur object at 0x1f6b190> credits 20 -20  >>> j.credits -= 10 *# get ET set*  Les crédits ont été consultés:  get <\_\_main\_\_.Joueur object at 0x1f6b190> credits -20 None  Les crédits ont changé:  set <\_\_main\_\_.Joueur object at 0x1f6b190> credits -20 -30 |

On vient d’implémenter une version encapsulée du [pattern observer](http://sametmax.com/le-pattern-observer-en-utilisant-des-decorateurs) dédié à un attribut. On peut faire de nombreuses choses avec les descripteurs: grouper des attributs, les transformer à la volée, les sauvegarder ailleurs (imaginez un objet de config qui sauvegarde automatiquement chaque modification de ses attributs dans un fichier…).

Les méthodes \_\_new\_\_ et \_\_init\_\_ n’ont rien de spécial. Ce sont des méthodes ordinaires. Mais parce qu’elles sont nommées ainsi, Python les détecte et les appelle automatiquement a un moment précis.

Ce moment, c’est ce qui différencie \_\_init\_\_ de \_\_new\_\_.

**\_\_init\_\_ pour initialiser**

\_\_init\_\_ est la méthode qui va être appelée automatiquement après qu’un objet ai été crée. **Ce n’est pas un contructeur du tout**, c’est un initialiseur.

Si vous faîtes ça:

|  |
| --- |
| >>> **class** Premiere(object):  ...  ... **def** \_\_init\_\_(self, prix):  ... **print** "%s euros" % prix  ...  >>> c = Premiere(10000)  10000 euros |

A la ligne c = Premiere(10000), Python va créer une instance de la classe Première(). Il va ensuite immédiatement et automatiquement appeler \_\_init\_\_ en lui passant cette instance en premier argument et les paramètres passés par l’appel Premiere(paramètres). Donc, quand \_\_init\_\_ est appelé, l’objet instancié existe déjà.

On va utiliser \_\_init\_\_ pour initialiser l’objet, c’est à dire pour lui donner son état de départ: changer les attributs, configurer l’objet par rapports aux arguments, etc.

Dans tous les autres langages, on utiliserait le constructeur pour faire ce boulot. Pas en Python.

L’avantage de \_\_init\_\_, c’est qu’il est très facile à manipuler. Il n’y a pas de magie dangereuse dans \_\_init\_\_: on a l’objet tout neuf, et les arguments passés à l’instancitation, on peut donc manipuler l’objet sans se soucier du reste. Ici on attache deux attributs à l’instance self:

|  |
| --- |
| >>> **class** Premiere(object):  ... discount = False  ... **def** \_\_init\_\_(self, prix):  ... self.prix = prix  ... **if** self.prix < 5000:  ... self.discount = True  ...  >>> c = Premiere(10000)  >>> c.discount  False |

Comme en Python les attributs sont dynamiques, on peut attacher un argument même si l’instance ne le déclare pas, et il est créé automatiquement.

En résumé: \_\_init\_\_ est appelé automatiquement APRES la création de l’objet, et on met dedans le code d’initialisation de l’objet (généralement une modification des attributs pour leur donner leur état de départ).

**\_\_new\_\_ pour créer**

\_\_new\_\_ est le vrai constructeur. Pour cette raison, elle doit retourner un objet.

|  |
| --- |
| >>> **class** Premiere(object):  ...  ... **def** \_\_new\_\_(cls, prix):  ... **print** "%s euros" % prix  ... **return** super(Premiere, cls).\_\_new\_\_(cls)  ...  >>> c = Premiere(10000)  10000 euros |

\_\_new\_\_ est appelée AVANT la création de l’objet, car c’est son boulot de créer l’instance et de la retourner. Comme on ne sait pas retourner une instance nous même (enfin si, mais pas dans cet article :-)), on appelle super() pour utiliser la méthode \_\_new\_\_ de object et créer une instance pour cette classe.

L’objet créé sera ensuite passé à \_\_init\_\_ automatiquement par Python.

On utilise rarement \_\_new\_\_. Les deux cas principaux sont:

* si on hérite d’un type immutable (str, int, tuple, etc), \_\_new\_\_ est le seul endroit où on puisse initialiser l’objet.
* dans le cas des métaclasses.

En résumé: \_\_new\_\_ est le vrai constructeur, il est appelé pour créer l’objet, et l’objet ainsi instancié est passé à \_\_init\_\_. Vous n’avez presque aucune raison de vous en servir, c’est vraiment pour les cas particuliers.

Voici l’ordre d’éxécution:

|  |
| --- |
| >>> **class** Premiere(object):  ... **def** \_\_new\_\_(cls, prix):  ... **print** "\_\_new\_\_"  ... **return** super(Premiere, cls).\_\_new\_\_(cls)  ... **def** \_\_init\_\_(self, \*args):  ... **print** "\_\_init\_\_"    >>> c = Premiere(10000)  \_\_new\_\_  \_\_init\_\_ |

**Exemple d’utilisation de \_\_new\_\_**

Généralement on sait très bien utiliser \_\_init\_\_, mais \_\_new\_\_ est moins évident.

L’usage le plus fréquent de \_\_new\_\_ quand on hérite d’objets immutables. Par exemple, si vous voulez faire un objet Temperature qui hérite de float et qui accepte une unité en plus, ceci ne va pas marcher:

|  |
| --- |
| **class** Temperature(float):    **def** \_\_init\_\_(self, value, unit):    super(Temperature, self).\_\_init\_\_(value)  self.unit = unit    **def** \_\_str\_\_(self):  **return** "%s%s" % (self.value, self.unit)    **print** Temperature(10, '°C')    Traceback (most recent call last):  File "<ipython-input-1-65b676255e09>", line 11, **in** <module>  Temperature(10, '°C')  TypeError: float() takes at most 1 argument (2 given) |

La raison est que du fait de la nature immutable de float, il est initialisé dans \_\_new\_\_, et il n’attend aucune valeur de plus dans \_\_new\_\_, mais on lui passe malgré tout (via Temperature(10, '°C')).

En revanche, ceci va marcher:

|  |
| --- |
| **class** Temperature(float):    **def** \_\_new\_\_(cls, value, unit):    instance = super(Temperature, cls).\_\_new\_\_(cls, value)  instance.unit = unit  **return** instance    **def** \_\_str\_\_(self):  **return** "%s%s" % (super(Temperature, self).\_\_str\_\_(), self.unit)    **print** Temperature(10, '°C')  10.0°C |

Comme on override \_\_new\_\_, on lui donne la possibilité d’accepter une argument de plus.

Un autre exemple serait de vouloir créer une chaîne de caractères qui est toujours en majuscule (ce qui est bien moins utile que l’exemple précédent):

|  |
| --- |
| **class** CapsLockString(str):    **def** \_\_init\_\_(self, value):    **print** value *# et maintenant je fais quoi ?*    **print** CapsLockString('test')  test  test |

Ça ne plantera pas, mais il n’y a rien que nous puissions faire car str est immutable. On ne peut tout simplement pas faire quoique ce soit avec value. Avec \_\_new\_\_, on peut faire quelque chose sur la chaîne intermédiaire:

|  |
| --- |
| **class** CapsLockString(str):    **def** \_\_new\_\_(cls, value):    **return** super(CapsLockString, cls).\_\_new\_\_(cls, value.upper())    **print** CapsLockString('test')  TEST |

Deux chaînes sont en fait créées, une normale, puis une en majuscule retournée par upper() qui va servir de valeur à notre objet (en fait il y en a même 3 dans l’implémentation CPython, c’est pour ça que les notations littérales sont plus rapides que l’usage des classes pour créer des built-in).

\_\_new\_\_ permet donc essentiellement de créer de jolis API. On l’utilise par ailleurs dans les metaclasses, mais ce sera pour un autre article.

Un troisième usage de \_\_new\_\_, assez rare (mais en même temps utiliser \_\_new\_\_ est déjà rare), c’est le pattern factory. Les javaistes le connaissent bien, c’est un motif de conception qui permet de gérer la création d’objets qui peuvent eux même créer des objets, qui créer des objets qui… Bref.

Car en fait \_\_new\_\_ peut retourner n’importe quoi. Il peut [retourner toujours la même instance](http://sametmax.com/usages-et-dangers-du-null-object-pattern-en-python/) pour faire un singleton par exemple. On peut même carrément renvoyer un truc qui n’a rien n’a voir, par exemple une fonction :

|  |
| --- |
| **class** FonctionFactory(object):    **def** \_\_new\_\_(self, value, repeat):    **def** repeater(string=value):    **return** string \* repeat    **return** repeater      >>> function = FonctionFactory('hello', 2) *# création de la fonction*  >>> **print** function()  hellohello  >>> **print** function('bonjour')  bonjourbonjour |

Ici on retourne carrément une fonction, et pas du tout une instance de FonctionFactory() comme prévu. On pourrait faire ceci de manière plus simple avec de la programmation fonctionnelle, mais \_\_new\_\_ permet de bénéficier de tout l’outillage de la POO.