NOHAR

Zeste de savoir

Le pattern Dispatcher en Python

12 août 2019

Table des matières

Le *Dispatcher* est un *design pattern* que j'aime beaucoup utiliser en Python car il permet de concevoir un programme de façon **événementielle**, c'est-à-dire comme une collection de *callbacks* qui sont utilisés en réaction à des *évènements*.

Il est très utile lorsque l'on veut donner à l'utilisateur de sa classe un moyen générique d'en personnaliser le comportement. On peut penser, par exemple, à un *bot* dont on voudrait qu'il réagisse à des commandes simples sur un canal de chat, ou un framework web, dans lequel on associe des actions à des URL, ou même un *Visitor*, dont on se sert pour traverser de grandes arborescences d'objets pour traiter ces objets de façon adéquate selon leur type, comme les arbres syntaxiques (AST) dans un compilateur.

En fait, ces dernières années, j'ai tellement passé de temps à réimplémenter ce *pattern* chaque fois que j'en ai eu besoin, que j'ai fini par le raffiner au point de chercher une façon de l'ajouter à n'importe quelle classe juste en héritant du *mixin-qui-va-bien*. Et c'est cette implémentation que je vais détailler au travers de ce tutoriel.

1. Un exemple pour commencer : un chatter bot

Imaginons que nous voulions programmer un *bot* pour un système de chat quelconque. Celui-ci pourrait réagir facilement à des commandes que l'on lui enverrait depuis un client de chat à la façon des bots IRC. Par exemple, si on lui envoie la commande `!ping`, le bot répondra `pong`, et si on lui envoie la commande `!date`, il nous retournera la date et l'heure.

C'est le genre de programme qu'un pattern Dispatcher rend tout à fait élégant à programmer : typiquement, dans ce genre de programme, on veut **séparer le code des différentes commandes** de celui qui est responsable d'administrer le bot et sa communication avec le réseau, par exemple.

Voici comment on pourrait réaliser cette partie de notre bot :

```python
from contextlib import suppress

class ChatBot:
    _callbacks = {}

    def react(self, message):
        """
        Process a message.
        Commands are assumed to follow this syntax:

            !<command> [<args>...]

        When a command is identified, find the corresponding callback and
        let it handle the command.

        """
        if not message.startswith('!'):
            return

        cmd, *args = message.split()
        cmd = cmd.lstrip('!')
        with suppress(KeyError):
            self._callbacks[cmd](self, *args)

    @classmethod
    def register_cmd(cls, callback):
        """
        Decorator to register a new callback.

        A callback should have the same name as its corresponding command.
        It takes an instance of the calling ChatBot object as its first
        argument and may accept an arbitrary number of positional arguments.

        """
        cls._callbacks[callback.__name__] = callback
        return callback
```

Notre `ChatBot` consiste en fait uniquement en ces deux méthodes :

— Le décorateur `@ChatBot.register_cmd` va servir à définir de nouveaux callbacks et les enregistrer auprès de la classe ChatBot, dans l'attribut de classe `_callbacks`,
— La méthode `bot.react(message)` identifie les messages qui contiennent des commandes, et exécute les commandes lorsqu'il trouve le callback correspondant.

Les commandes, quant à elles, sont définies *hors* de la classe. Comme ceci :

```python
@ChatBot.register_cmd
def ping(bot, *args):
    """
    Usage: !ping

    Answer "pong!".
    """
    print("pong!")

@ChatBot.register_cmd
def list_cmds(bot, *args):
    """
    Usage: !list_cmds

    List all available commands.
    """
    print("Available commands: ")
    print(', '.join(sorted(bot._callbacks.keys())))


@ChatBot.register_cmd
def help(bot, cmd=None, *args):
    """
    Usage: !help [command]

    Show help about commands.
    """
    if not cmd:
        list_cmds(bot)

            print("\nType '!help <command>' to get help about specific commands
        return
    cmd = bot._callbacks.get(cmd, None)
    if cmd is None:
        return
    doc = cmd.__doc__
    if not doc:
        return
    print('\n'.join(line.strip() for line in
        doc.strip().split('\n')))
```

Comme vous le voyez, il suffit de créer une fonction qui prend une instance de la classe `ChatBot` en premier argument pour implémenter une commande, et de lui appliquer le décorateur `@ChatBot.register_cmd` pour l'ajouter au robot. Notez que j'utilise ici une petite astuce qui consiste à se servir des *docstrings* de ces callbacks pour générer les messages d'aide du robot.

Ce qu'il est important de noter ici, c'est que les développeurs n'ont plus du tout besoin de se soucier du fonctionnement interne du bot lorsque leur but est simplement de lui ajouter de nouveaux comportements. Ils peuvent se concentrer uniquement sur l'écriture du callback et de

sa doc.

Vérifions son fonctionnement dans la console :

```
>>> bot = ChatBot()
>>> bot.react("Hello !")
>>> bot.react("!help")
Available commands:
help, list_cmds, ping

Type '!help <command>' to get help about specific commands
>>> bot.react("!help ping")
Usage: !ping

Answer "pong!".
>>> bot.react("!ping")
pong!
```

En somme, le fait que cette classe `ChatBot` soit conçue comme un dispatcher permet de découpler complètement l'implémentation des commandes auxquelles il doit réagir de tout le reste de la classe, ce qui le rend facile à maintenir et à étendre.

2. Un dispatcher générique

Bon, c'est bien joli de créer un dispatcher en deux méthodes, mais il faut quand même avouer que ces méthodes ne sont pas évidentes à se rappeler. Ce serait plutôt cool qu'on puisse créer un dispatcher juste en héritant de la *classe-qui-va-bien*, pour ne pas risquer de se tromper en le codant. D'ailleurs, le top, ce serait qu'il existe un *mixin*, qui puisse apporter la totalité de cette fonctionnalité à n'importe quelle classe.

Et ce n'est pas trivial, figurez-vous !

2.1. Anatomie d'un dispatcher

Avant d'aller plus loin, il faut caractériser ce qui définit un dispatcher. Concrètement, on a besoin d'au moins trois éléments :

— **Une structure** (un *mapping*, comme un dictionnaire) **dans laquelle on référence les callbacks**.
— **Une méthode de classe permettant d'enregistrer un nouveau callback**, et de préférence, qui se décline en deux versions : une version *explicite* et une version *décorateur*. Cette méthode s'appellerait `register`.
— **Une méthode permettant de récupérer un callback au moyen de sa clé**. Cette méthode (d'instance) s'appellerait `dispatch`.

2.2. Cassons-nous les dents sur une première implémentation…

On pourrait partir bille en tête en définissant notre *mixin* de façon directe, comme ceci :

```
class WrongDispatcher:
    __callbacks__ = {}

    @classmethod
    def set_callback(cls, key, cbk):
        cls.__callbacks__[key] = cbk
        return cbk

    @classmethod
    def register(cls, key):
        def wrapper(cbk):
            return cls.set_callback(key, cbk)
        return wrapper

    @property
    def dispatcher(self):
        return self.__callbacks__

    def dispatch(self, key, default=None):
        return self.__callbacks__.get(key, default)
```

Après tout, cette classe implémente *exactement* l'anatomie du Dispatcher telle que nous l'avons décrite à l'instant. Après un rapide test, on pourrait même croire que ça marche :

```
>>> class A(WrongDispatcher):
...     pass
...
>>> @A.register('foo')
... def foo(): pass
...
>>> a = A()
>>> a.dispatcher
{'foo': <function foo at 0x7f637e4b1950>}
>>> a.dispatch('foo')
<function foo at 0x7f637e4b1950>
```

Mais regardez ce qu'il se passe lorsque l'on crée un second dispatcher :

```
>>> class B(WrongDispatcher):
...     pass
...
```

```
>>> @B.register('bar')
... def bar(): pass
...
>>> b = B()
>>> b.dispatch('bar')
<function bar at 0x7f637e4b19d8>
>>> b.dispatch('foo')
<function foo at 0x7f637e4b1950>   # <---- WTF?!
```

Que vient faire dans `B` le callback `foo` de `A` ?!

En fait, le problème, c'est que ces deux classes ont rempli *le même dictionnaire de callbacks*, hérité de leur classe mère commune :

```
>>> WrongDispatcher.__callbacks__
{'bar': <function bar at 0x7f637e4b19d8>,
 'foo': <function foo at 0x7f637e4b1950>}
```

Ça, c'est embêtant : les dispatchers se marchent sur les pompes !

Pour éviter cela, il faut que les classes filles aient *leur propre dictionnaire de callbacks à elles*. Regardez :

```
>>> class C(WrongDispatcher):
...     __callbacks__ = {}   # on crée un nouveau dictionnaire
...                          # de callbacks
...
>>> @C.register('baz')
... def baz(): pass
...
>>> C.__callbacks__
{'baz': <function baz at 0x7f637e4b1a60>}
>>> WrongDispatcher.__callbacks__
{'bar': <function bar at 0x7f637e4b19d8>,
 'foo': <function foo at 0x7f637e4b1950>}
```

Cette fois, le callback que l'on a enregistré dans `C` n'a contaminé personne, parce qu'on a eu la présence d'esprit de créer un nouveau dictionnaire `__callbacks__` dans la déclaration de la classe `C`.

C'est vraiment pas pratique ! On ne va quand même pas demander à l'utilisateur de rajouter manuellement cet attribut chaque fois qu'il hérite de notre mixin ou d'une de ses filles...

Eh bien j'ai une bonne et une mauvaise nouvelle :

— La bonne nouvelle, c'est qu'il existe une solution propre à ce problème.
— La mauvaise nouvelle, c'est que cette solution consiste à implémenter la création de l'attribut de classe `__callbacks__` *dans le constructeur d'une métaclasse*...

2.3. Cassons-nous la tête sur la seconde implémentation...

En voilà un mot qui fait peur : **métaclasse** !

Je vous rassure tout de suite, je ne vais pas me lancer dans un cours théorique sur les métaclasses. D'abord parce que je trouve ça plutôt ennuyeux, et ensuite, parce que, comme probablement 99% des gens (qui n'oseront jamais l'avouer publiquement) : je n'ai jamais vraiment pigé aucun cours sur les métaclasses. Il a fallu que j'expérimente tout seul pour ça.

Par contre, je vais essayer de vous montrer simplement **ce que c'est**, et **pourquoi on a besoin d'en créer une ici**.

Bon, prenons notre problème à nous : nous avons besoin que lorsque l'on définit une classe particulière (un dispatcher), un nouvel attribut soit créé et ajouté automatiquement à cette classe, attribut que j'ai appelé `__callbacks__` un peu plus haut.

En fait, ça ressemble beaucoup à ce qui se passe lorsque l'on définit n'importe quelle classe. Par exemple si je définis une classe `Foo` comme ceci :

```
1 class Foo:
2     def some_method(self):
3         pass
```

Et que j'affiche tous ses attributs dans la console...

```
1 >>> dir(Foo)
2 ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
      '__eq__',
3 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
      '__init__',
4 '__le__', '__lt__', '__module__', '__ne__', '__new__',
      '__reduce__',
5 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
      '__str__',
6 '__subclasshook__', '__weakref__', 'some_method']
```

On s'aperçoit que cette classe `Foo` a quand même un paquet d'attributs cachés ! Par exemple, l'attribut `__dict__` contient une référence sur l'unique méthode de notre classe :

```
1 >>> Foo.__dict__['some_method']
2 <function Foo.some_method at 0x7f637e4b1b70>
```

Cet attribut `__dict__`, il a bien fallu qu'il soit créé et initialisé quelque part, lorsque Python a fini de parser la classe `Foo`, il a **construit** un objet `Foo` avec des attributs spéciaux.

Vu qu'on peut manipuler `Foo` dans la console, on peut lui demander son type :

```
>>> type(Foo)
<class 'type'>
```

Bon, accrochez-vous parce que c'est là que ça devient rigolo. Le type de `Foo` est `type`. Ça veut donc dire que `type` est une... classe, et que lorsqu'on l'instancie, on obtient des classes, comme `Foo`.

Une classe que l'on instancie pour obtenir une classe, on appelle ça une **métaclasse**. Et la métaclasse principale de Python, est `type`. En somme, pour résumer : **c'est dans le constructeur de `type` que la classe `Foo` a gagné ses attributs, comme `Foo.__dict__`.**

Nous, notre but, c'est d'ajouter un nouveau dictionnaire `__callbacks__` en attribut de la classe. Un attribut du même genre que `__dict__`. Il est donc naturel que nous rajoutions cet attribut dans le *constructeur de la métaclasse* de notre dispatcher. Il suffit donc de **surcharger** le constructeur de la (méta)classe `type`, pour y ajouter notre tambouille. On en profitera d'ailleurs pour déplacer toutes nos `@classmethod` dans la définition de la métaclasse.

Allons, courage. C'est pas si terrible que ça. J'ai même commenté le code en français, pour une fois :

```
class SingleDispatcherMeta(type):
    def __new__(mcs, name, bases, attrs):
        # Juste avant que la classe ne soit définie par Python.

        # on lui ajoute un dictionnaire de callbacks
        callbacks = dict()
        attrs['__callbacks__'] = callbacks

        # et une property "dispatcher" qui retourne ce
            dictionnaire.
        attrs['dispatcher'] = property(lambda obj: callbacks)

        return super().__new__(mcs, name, bases, attrs)

    # Enregistre un nouveau callback auprès du dispatcher
    def set_callback(cls, key, callback):
        cls.__callbacks__[key] = callback
        return callback

    # Pareil, mais avec un decorator @Class.register(key)
    def register(cls, key):
        def wrapper(callback):
            return cls.set_callback(key, callback)
        return wrapper


# Le mixin en lui-même.
class SingleDispatcher(metaclass=SingleDispatcherMeta):
```

```
28      # Retourne le callback associé à la clé key, s'il existe.
29      def dispatch(self, key, default=None):
30          return self.dispatcher.get(key, default)
```

Constatons que cela fonctionne :

```
1  >>> class A(SingleDispatcher):
2  ...     pass
3  ...
4  >>> class B(SingleDispatcher):
5  ...     pass
6  ...
7  >>> @A.register('foo')
8  ... def foo(): pass
9  ...
10 >>> @B.register('bar')
11 ... def bar(): pass
12 ...
13 >>> a = A()
14 >>> b = B()
15 >>> a.dispatcher
16 {'foo': <function foo at 0x7f4b26875ae8>}
17 >>> b.dispatcher
18 {'bar': <function bar at 0x7f4b26875b70>}
```

Ici, on se rend compte que les callbacks de `A` et de `B` sont bien enregistrés dans des dictionnaires bien distincts. Mission accomplie.

2.4. Retour sur notre exemple

Voici à quoi ressemble la classe `ChatBot` de notre exemple en utilisant cette métaclasse. Notez que l'on réimplémente quand même le décorateur `@ChatBot.register_cmd`, puisque celui-ci utilise automatiquement le nom de la fonction comme clé pour identifier les callbacks.

```
1  class ChatBot(SingleDispatcher):
2      @classmethod
3      def register_cmd(cls, callback):
4          return cls.set_callback(callback.__name__, callback)
5
6      def react(self, message):
7          if not message.startswith('!'):
8              return
9          cmd, *args = message.split()
10         cmd = cmd.lstrip('!')
11
```

```
12          handler = self.dispatch(cmd, lambda *_: None)
13          handler(self, *args)
14
15  # ... l'enregistrement des callbacks se fait de la même manière ...
```

Plutôt cool, non ?

3. Un dispatcher générique... et extensible !

Bien, nous avons un pattern que nous pouvons utiliser en héritant d'une classe spéciale et nous pouvons même hériter d'un `SingleDispatcher` *sans risquer de contaminer son dictionnaire de callbacks*. C'est pas mal, mais cette solution n'est qu'à moitié satisfaisante.

En fait, lorsque l'on hérite d'une classe, intuitivement, on s'attend à hériter de tous ses comportements, et justement en Python, le principal intérêt de l'héritage, c'est de ne pas avoir besoin de les recoder. Sauf que ce n'est pas le cas lorsque l'on hérite d'un `SingleDispatcher` : on gagne ses méthodes et son fonctionnement général, mais **on n'a plus accès à ses callbacks**.

Si on reprend l'exemple de notre `ChatBot`, ça veut dire que si un développeur décide d'hériter de `ChatBot`, il se retrouve obligé de réimplémenter et réenregistrer la commande `help` qui gère la doc, par exemple, alors qu'on peut tout à fait se dire que ce callback sera utile à tous les bots qui seront implémentés de la sorte…

Bref, le but, c'est que lorsque l'on hérite d'un tel dispatcher, on puisse également hériter de tous ses callbacks.

En somme, si `A` est un dispatcher et que `B` hérite de `A` :

— `B` doit répondre par défaut à tous les signaux de `A`, avec le même comportement par défaut,
— Les nouveaux callbacks ajoutés à `A` doivent être accessibles à `B`,
— Les nouveaux callbacks ajoutés à `B` ne doivent concerner que `B` et ses classes filles.

En fait, la bibliothèque standard fournit une structure, relativement méconnue, qui répond parfaitement à ce problème. J'ai nommé `collections.ChainMap` . Voyons comment ça marche.

Imaginons que nous ayons un dictionnaire de callbacks pour `A`. Au lieu que ce soit un simple dictionnaire, ce sera un `ChainMap` (par défaut, les deux se comportent exactement pareil) :

```
1  >>> A = ChainMap()
2  >>> A['spam'] = 'a_spam'
3  >>> A['eggs'] = 'a_eggs'
4  >>> A
5  ChainMap({'eggs': 'a_eggs', 'spam': 'a_spam'})
```

Maintenant, si nous voulons créer le dictionnaire de callbacks pour `B`, il suffit d'*ajouter* les dictionnaires mappés par `A` dans le `ChainMap` de `B` :

```
>>> B = ChainMap()
>>> B.maps.extend(A.maps)
>>> B
ChainMap({}, {'eggs': 'a_eggs', 'spam': 'a_spam'})
```

Ajoutons à `B` un nouveau callback (`bacon`) et surchargeons son callback `spam` :

```
>>> B['bacon'] = 'b_bacon'
>>> B['spam'] = 'b_spam'
>>> B
ChainMap({'spam': 'b_spam', 'bacon': 'b_bacon'},
         {'eggs': 'a_eggs', 'spam': 'a_spam'})
>>> for key, callback in B.items():
...     print("{}: {}".format(key, callback))
...
eggs: a_eggs
spam: b_spam
bacon: b_bacon
```

Vérifions que les callbacks de `A` n'ont pas bougé :

```
>>> for key, callback in A.items():
...     print("{}: {}".format(key, callback))
...
eggs: a_eggs
spam: a_spam
```

Maintenant vérifions que les nouveaux callbacks ajoutés à `A` deviennent automatiquement accessibles à `B` :

```
>>> A['homard'] = 'a_homard'
>>> for key, callback in B.items():
...     print("{}: {}".format(key, callback))
...
eggs: a_eggs
homard: a_homard
spam: b_spam
bacon: b_bacon
```

Je ne sais pas vous, mais personnellement, même après plusieurs années de carrière, je reste ébahi devant la capacité de la bibliothèque standard à résoudre notre problème avant même qu'il ne se pose…

3. *Un dispatcher générique... et extensible !*

Notre problème d'héritage est complètement résolu. On n'a plus qu'à implémenter la métaclasse, et on aura fini.

```
class DispatcherMeta(type):
    def __new__(mcs, name, bases, attrs):
        # Juste avant que la classe ne soit définie par Python

        # On construit le dictionnaire de callbacks en héritant de
            ceux des
        # classes mères
        callbacks = ChainMap()
        maps = callbacks.maps
        for base in bases:
            if isinstance(base, DispatcherMeta):
                maps.extend(base.__callbacks__.maps)

        # Comme avant, on ajoute le dictionnaire de callbacks et
        # la property "dispatcher" pour y accéder
        attrs['__callbacks__'] = callbacks
        attrs['dispatcher'] = property(lambda obj: callbacks)
        cls = super().__new__(mcs, name, bases, attrs)
        return cls

    def set_callback(cls, key, callback):
        cls.__callbacks__[key] = callback
        return callback

    def register(cls, key):
        def wrapper(callback):
            return cls.set_callback(key, callback)
        return wrapper


class Dispatcher(metaclass=DispatcherMeta):
    def dispatch(self, key, default=None):
        return self.dispatcher.get(key, default)
```

Vérifions que ce nouveau dispatcher se comporte comme prévu :

```
>>> class A(Dispatcher): pass
...
>>> @A.register('spam')
... def spam(): pass
...
>>> @A.register('eggs')
... def eggs(): pass
...
```

```
>>> class B(A): pass
...
>>> b = B()
>>> @B.register('bacon')
... def bacon(): pass
...
>>> @B.register('spam')
... def spam_override(): pass
...
>>> @A.register('homard')
... def homard(): pass
...
>>> for key, val in b.dispatcher.items():
...     print(key, val)
...
eggs <function eggs at 0x7fe513411268>
spam <function spam_override at 0x7fe513411400>
bacon <function bacon at 0x7fe513411378>
homard <function homard at 0x7fe513411488>
```

Victoire !

4. Le Visitor en douze lignes

Vous connaissez le décorateur `@functools.singledispatch` ?

Celui-ci sert à surcharger le comportement d'une fonction, suivant le type du premier argument de cette fonction. Remarquez qu'à la différence du programme qu'on a pris en exemple jusqu'à maintenant, celui-ci dispatche en fonction d'un **type** de données et plus en fonction d'une chaîne de caractères, et c'est là que le Dispatcher devient vraiment marrant : on peut dispatcher sur n'importe quel objet *hashable*, comme les types de données, les tuples, les chaînes de caractères, les `frozenset`, les `bytes`…

Par exemple, on peut implémenter le pattern *Visitor* de façon extrêmement concise grâce à notre `Dispatcher` :

```
from dispatcher_pattern import Dispatcher

class Visitor(Dispatcher):
    def visit(self, key, *args, **kwargs):
        handler = self.dispatch(type(key))
        if handler:
            return handler(self, key, *args, **kwargs)
        raise RuntimeError("Unknown key: {!r}".format(key))

    @classmethod
```

```
11      def on(cls, type_):
12          return cls.register(type_)
```

Servons-nous de ce visitor pour évaluer les arbres syntaxiques (AST) de simples expressions algébriques. Comme le suivant :

```
1  (40 + 2) * 3
2
3         *
4        / \
5       +   3
6      / \
7    40   2
```

Le Visitor va évaluer cet arbre en le parcourant comme ceci :

```
1        (*)         Le Visitor se trouve sur le noeud *,
2        / \         Il sait que pour évaluer ce noeud, il faut qu'il
             évalue
3       +   3        ses deux opérandes avant de retourner
4      / \           leur produit. Il va donc s'exécuter récursivement
           sur les
5    40   2          deux noeuds fils : + et 3
6
7
8         *          Le Visitor descend sur le noeud +,
9        / \         Comme pour le noeud *, il faut qu'il évalue
             d'abord les deux
10     (+)  3        opérandes avant de retourner leur somme.
11     / \
12   40   2
13
14
15        *          "40" est une valeur constante, le visitor retourne
              cette valeur
16       / \         telle quelle.
17      +   3
18     / \
19   (40)  2
20
21
22        *          Idem pour "2".
23       / \
24      +   3
25     / \
26   40  (2)
```

```
27
28
29          *           De retour dans son évaluation du noeud +, il
                calcule la somme
30        / \           de ses deux opérandes et la retourne.
31      (42) 3
32
33
34          *           Avant de retourner sur le noeud *, le visitor
35        / \           est appelé récursivement par celui-ci sur le "3",
              sur lequel
36       42 (3)         il n'a ici rien à faire.
37
38
39       (126)          Le visitor retourne dans l'évaluation de la
             multiplaction,
40                      et renvoie le produit de 42 et 3.
```

Implémentons les classes qui représenteront les noeuds de l'AST. Rien de difficile, ils doivent juste "décrire" ce qu'ils sont :

```python
class ASTNode:
    pass

class ConstVal(ASTNode):
    def __init__(self, value):
        self.value = value

class BinOp(ASTNode):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinOp):
    pass

class Mul(BinOp):
    pass

class Sub(BinOp):
    pass

class Div(BinOp):
    pass
```

Ici, nous avons défini un AST pouvant contenir :

— des noeuds `ConstVal` : les feuilles de l'arbre,

— les quatre opérations binaires : addition, multiplication, soustraction, division.

Pour évaluer un arbre composé de ces noeuds, on peut utiliser le visitor suivant :

```
class EvalVisitor(Visitor):
    pass

@EvalVisitor.on(ConstVal)
def eval_cval(vtr, node):
    return node.value

@EvalVisitor.on(Add)
def eval_add(vtr, node):
    return vtr.visit(node.left) + vtr.visit(node.right)

@EvalVisitor.on(Mul)
def eval_mul(vtr, node):
    return vtr.visit(node.left) * vtr.visit(node.right)

@EvalVisitor.on(Sub)
def eval_sub(vtr, node):
    return vtr.visit(node.left) - vtr.visit(node.right)

@EvalVisitor.on(Div)
def eval_div(vtr, node):
    return vtr.visit(node.left) / vtr.visit(node.right)
```

Essayons :

```
>>> ast = Mul(Add(ConstVal(40), ConstVal(2)), ConstVal(3))  # (40 +
    2) * 3
>>> vtor = EvalVisitor()
>>> vtor.visit(ast)
126
```

Aucun soucis.

5. Bonus : un Visitor qui comprend l'héritage

Un détail qui survient lorsque l'on utilise beaucoup ce genre de dispatchers sur des types de données, est que l'on peut vouloir exploiter le fait que certaines classes héritent d'autres pour définir des actions par défaut. Si vous regardez bien, dans notre ast, les quatre opérations dérivent toutes de la classe `BinOp`. **Et si on créait un visitor dans lequel on pourrait spécifier un callback par défaut de toutes les classes filles d'un type donné ?**

En fait, ce n'est pas très difficile : pour nous faciliter la tâche, toutes les classes en Python ont un attribut `__mro__` (*Method Resolution Order*), qui sert précisément à nous dire dans quel ordre on doit résoudre les appels des méthodes héritées des classes mères. Il nous suffit de faire exactement la même chose que Python en interne, et boucler dessus !

```
class Visitor(Dispatcher):
    def visit(self, key, *args, **kwargs):
        cls = type(key)
        for base in cls.__mro__:
            handler = self.dispatch(base)
            if handler:
                return handler(self, key, *args, **kwargs)
        raise RuntimeError("Unknown key: {!r}".format(cls))

    @classmethod
    def on(cls, type_):
        return cls.register(type_)
```

Pour illustrer ce comportement, implémentons un visiteur qui transforme les expressions en leur représentation textuelle. Commençons par modifier légèrement les opérations binaires :

```
class BinOp(ASTNode):
    symbol = None
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinOp):
    symbol = '+'

class Mul(BinOp):
    symbol = '*'

class Sub(BinOp):
    symbol = '-'

class Div(BinOp):
    symbol = '/'
```

Nous pouvons maintenant écrire un visiteur qui tire parti de l'arbre d'héritage de notre AST :

```
class PrintVisitor(Visitor):
    pass

@PrintVisitor.on(ConstVal)
```

```
5  def print_val(vtr, node):
6      return str(node.value)
7
8  @PrintVisitor.on(BinOp)
9  def print_binop(vtr, node):
10     if node.symbol is None:
11         raise RuntimeError("Unknown BinOp: {!r}".format(node))
12     return "({} {} {})".format(
13         vtr.visit(node.left),
14         node.symbol,
15         vtr.visit(node.right)
16     )
```

Et... c'est tout.

```
1  >>> ast = Mul(Add(ConstVal(40), ConstVal(2)), ConstVal(3))  # (40 +
       2) * 3
2  >>> vtr = PrintVisitor()
3  >>> print(vtr.visit(ast))
4  ((40 + 2) * 3)
```

En somme, grâce à notre *Dispatcher*, nous avons pu implémenter un *Visitor* qui :

— *Est héritable* : si l'on hérite d'une implémentation de `Visitor`, on hérite également de tous ses callbacks, que l'on peut surcharger ou laisser comme callbacks par défaut.
— *Comprend la sémantique de l'héritage* : si le `Visitor` ne trouve pas de callback pour un type donné, il essaye de se rattrapper en cherchant un callback pour tous les parents de ce type.

Autant de petites choses pour en faire plus en tapant le moins de code possible.

Nous venons d'implémenter un *design pattern* parfait pour les flemmards !

En effet, en héritant (même partiellement) de notre `Dispatcher`, nous pouvons ajouter à n'importe quelle classe la possibilité :

— d'enregistrer des callbacks via un simple décorateur,
— callbacks qui peuvent réagir à n'importe quelle clé du moment que celle-ci est *hashable* (qu'elle peut être utilisée comme clé d'un dictionnaire),
— d'implémenter la fonctionnalité métier d'une application en dehors du coeur de la classe,
— de rendre cette fonctionnalité métier parfaitement extensible,
— d'*hériter* proprement de cette classe, et de surcharger ses callbacks sans craindre les effets de bord,
— d'implémenter de façon tout à fait pythonique un autre *pattern*, très utilisé dans les compilateurs : le `Visitor`.

Vous verrez, maintenant que vous savez qu'il existe, ce *pattern* va devenir un de vos meilleurs potes !

Je souhaite remercier tous les membres qui ont participé à la bêta de ce tutoriel. En particulier **Gabbro, entwanne, artragis et yoch** pour leurs remarques constructives.