Introduction au module ipaddress

Version 3.7.4

Guido van Rossum
and the Python development team

septembre 07, 2019

Python Software Foundation
Email : docs@python.org

Table des matières

auteur Peter Moody

auteur Nick Coghlan

Aperçu

Ce document vise à fournir une introduction rapide au module `ipaddress`. Il est destiné aux utilisateurs qui ne sont pas familiers avec la terminologie des réseaux IP, mais il peut aussi être utile aux ingénieurs réseaux qui cherchent un aperçu de la représentation des concepts d'adressage IP avec le module `ipaddress`.

1 Créer un objet Adresse/Réseau/Interface

Vu que `ipaddress` est un module pour inspecter et manipuler des adresses IP, la première chose que vous voudrez faire est de créer quelques objets. Vous pouvez utiliser `ipaddress` pour créer des objets à partir de chaînes de caractères et d'entiers.

1.1 Note sur les versions d'IP

Pour les lecteurs qui ne sont pas particulièrement familiers avec l'adressage IP il est important de savoir que le protocole IP est actuellement en évolution de la version 4 du protocole vers la version 6. Cette transition est largement due au fait que la version 4 du protocole ne fournit pas assez d'adresses pour gérer les besoins du monde entier, particulièrement à cause de la croissance des périphériques directement connectés à Internet.

Expliquer les détails des différences entre les deux versions du protocole est au-delà du périmètre de cette introduction, mais les lecteurs doivent au moins être avertis de l'existence de ces deux versions et qu'il sera nécessaire de forcer l'utilisation d'une version ou de l'autre.

1.2 Adresses IP des hôtes

Les adresses, souvent dénommées "adresses hôtes" sont les unités les plus basiques quand on travaille avec l'adressage IP. Le moyen le plus simple de créer des adresses est d'utiliser la fonction de fabrication `ipaddress.ip_address()` qui va automatiquement déterminer quoi créer entre une adresse IPv4 ou une adresse IPv6 en fonction de la valeur qui est transmise :

```
>>> ipaddress.ip_address('192.0.2.1')
IPv4Address('192.0.2.1')
>>> ipaddress.ip_address('2001:DB8::1')
IPv6Address('2001:db8::1')
```

Les adresses peuvent être créées directement depuis des entiers. Les valeurs qui correspondent à des entiers 32 bits sont assimilées à des adresses IPv4 :

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.ip_address(42540766411282592856903984951653826561)
IPv6Address('2001:db8::1')
```

Pour forcer l'utilisation d'IPv4 ou d'IPv6, la classe appropriée peut être invoquée directement. C'est particulièrement utile pour forcer la création d'adresse IPv6 pour de petits entiers :

```
>>> ipaddress.ip_address(1)
IPv4Address('0.0.0.1')
>>> ipaddress.IPv4Address(1)
IPv4Address('0.0.0.1')
>>> ipaddress.IPv6Address(1)
IPv6Address('::1')
```

1.3 Définir des réseaux

Les adresses hôtes sont souvent regroupées dans des réseaux IP, donc `ipaddress` fournit un moyen de créer, d'inspecter et de manipuler les définitions des réseaux. Les objets correspondants aux réseaux IP sont construits à partir de chaînes de caractères qui définissent le périmètre des adresses hôtes qui font partie de ce réseau. La forme la plus simple de cette

information est un couple ”adresse réseau/préfixe réseau” , où le préfixe définit le nombre de bits de tête qui sont comparés pour déterminer si l'adresse fait ou ne fait pas partie de ce réseau et l'adresse réseau définit la valeur attendue pour ces bits.

Tout comme pour les adresses, une fonction de fabrication est disponible et détermine automatiquement la version correcte d'IP :

```
>>> ipaddress.ip_network('192.0.2.0/24')
IPv4Network('192.0.2.0/24')
>>> ipaddress.ip_network('2001:db8::0/96')
IPv6Network('2001:db8::/96')
```

Il est interdit pour des objets réseaux d'avoir des bits affectés à leurs hôtes mis à 1. Ainsi la chaine de caractères `192.0.2.1/24` ne peut définir un réseau. Ces objets réseaux sont aussi appelés objets d'interfaces car la notation `adresse ip / réseau` est couramment utilisée pour décrire les interfaces réseau d'un ordinateur sur un réseau donné (nous les détaillons plus loin dans cette section).

Par défaut, tenter de créer un objet réseau avec des bits d'hôtes mis à 1 lève une `ValueError`. Pour demander que les bits supplémentaires soient plutôt forcés à zéro, l'attribut `strict=False` peut être passé au constructeur :

```
>>> ipaddress.ip_network('192.0.2.1/24')
Traceback (most recent call last):
  ...
ValueError: 192.0.2.1/24 has host bits set
>>> ipaddress.ip_network('192.0.2.1/24', strict=False)
IPv4Network('192.0.2.0/24')
```

Alors que la forme textuelle offre davantage de flexibilité les réseaux peuvent aussi être définis avec des entiers, tout comme les adresses hôtes. Dans ce cas le réseau est considéré comme contenant uniquement l'adresse identifiée par l'entier, donc le préfixe réseau inclut l'adresse du réseau complet :

```
>>> ipaddress.ip_network(3221225984)
IPv4Network('192.0.2.0/32')
>>> ipaddress.ip_network(42540766411282592856903984951653826560)
IPv6Network('2001:db8::/128')
```

Comme avec les adresses, la création d'un type de réseau particulier peut être forcée en appelant directement le constructeur de la classe plutôt que d'utiliser la fonction de fabrication.

1.4 Interfaces des hôtes

Comme mentionné ci-dessus, si vous avez besoin de décrire une adresse sur un réseau particulier, ni l'adresse ni les classes réseaux ne sont suffisantes. Les notations comme `192.0.2.1/24` sont généralement utilisées par les ingénieurs réseaux et les personnes qui écrivent des outils pour les pare-feu et les routeurs comme raccourci pour ” l'hôte `192.0.2.1` sur le réseau `192.0.2.0/24`”, par conséquent, `ipaddress` fournit un ensemble de classes hybrides qui associent une adresse à un réseau particulier. L'interface pour la création est identique à celle pour la définition des objets réseaux, excepté que la partie adresse n'est pas contrainte d'être une adresse réseau.

```
>>> ipaddress.ip_interface('192.0.2.1/24')
IPv4Interface('192.0.2.1/24')
>>> ipaddress.ip_interface('2001:db8::1/96')
IPv6Interface('2001:db8::1/96')
```

Les entiers sont acceptés en entrée (comme avec les réseaux) et l'utilisation d'une version d'IP peut être forcée en appelant directement le constructeur adapté.

2 Inspecter les objets Address/Network/Interface

Vous vous êtes donné la peine de créer un objet (adresselréseaulinterface)IPv(4|6), donc vous voudrez probablement des informations sur celui-ci. `ipaddress` essaie de rendre ceci facile et intuitif.

Extraire la version du protocole IP

```
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> addr6 = ipaddress.ip_address('2001:db8::1')
>>> addr6.version
6
>>> addr4.version
4
```

Obtenir le réseau à partir de l'interface :

```
>>> host4 = ipaddress.ip_interface('192.0.2.1/24')
>>> host4.network
IPv4Network('192.0.2.0/24')
>>> host6 = ipaddress.ip_interface('2001:db8::1/96')
>>> host6.network
IPv6Network('2001:db8::/96')
```

Trouver combien d'adresses individuelles sont dans un réseau :

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.num_addresses
256
>>> net6 = ipaddress.ip_network('2001:db8::0/96')
>>> net6.num_addresses
4294967296
```

Itération sur chacune des adresses "utilisables" d'un réseau :

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> for x in net4.hosts():
...     print(x)
192.0.2.1
192.0.2.2
192.0.2.3
192.0.2.4
...
192.0.2.252
192.0.2.253
192.0.2.254
```

Obtenir le masque réseau (définit les bits correspondant au préfixe du réseau) ou le masque de l'hôte (tous les bits qui ne sont pas dans le masque du réseau) :

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.netmask
IPv4Address('255.255.255.0')
>>> net4.hostmask
IPv4Address('0.0.0.255')
>>> net6 = ipaddress.ip_network('2001:db8::0/96')
>>> net6.netmask
IPv6Address('ffff:ffff:ffff:ffff:ffff:ffff::')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> net6.hostmask
IPv6Address('::ffff:ffff')
```

Éclater ou compresser l'adresse :

```
>>> addr6.exploded
'2001:0db8:0000:0000:0000:0000:0000:0001'
>>> addr6.compressed
'2001:db8::1'
>>> net6.exploded
'2001:0db8:0000:0000:0000:0000:0000:0000/96'
>>> net6.compressed
'2001:db8::/96'
```

Alors que IPv4 ne gère pas l'éclatement ou la compression, les objets associés fournissent toujours les propriétés adaptées pour que du code, le plus neutre possible vis-à-vis de la version, puisse facilement s'assurer que la forme la plus concise ou la plus verbeuse utilisée pour des adresses IPv6 va aussi fonctionner pour gérer des adresses IPv4.

3 Réseaux en tant que listes d'adresses

Il est parfois utile de traiter les réseaux en tant que listes. Cela signifie qu'il est possible de les indexer comme ça :

```
>>> net4[1]
IPv4Address('192.0.2.1')
>>> net4[-1]
IPv4Address('192.0.2.255')
>>> net6[1]
IPv6Address('2001:db8::1')
>>> net6[-1]
IPv6Address('2001:db8::ffff:ffff')
```

Cela signifie aussi que les objets réseaux se prêtent bien à l'utilisation de la syntaxe suivante pour le test d'appartenance à la liste :

```
if address in network:
    # do something
```

En se basant sur le préfixe réseau on peut efficacement tester l'appartenance :

```
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> addr4 in ipaddress.ip_network('192.0.2.0/24')
True
>>> addr4 in ipaddress.ip_network('192.0.3.0/24')
False
```

4 Comparaisons

`ipaddress` fournit des moyens simples et intuitifs (du moins nous l'espérons) pour comparer les objets, quand cela fait sens :

```
>>> ipaddress.ip_address('192.0.2.1') < ipaddress.ip_address('192.0.2.2')
True
```

Une exception `TypeError` est levée si vous tentez de comparer des objets de différentes versions ou de types différents.

5 Utiliser des adresse IP avec d'autre modules

Les autres modules qui utilisent des adresses IP (comme `socket`) n'acceptent généralement pas les objets de ce module directement. Au lieu de cela, ils doivent être convertis en entiers ou en chaînes de caractères que l'autre module va accepter :

```
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> str(addr4)
'192.0.2.1'
>>> int(addr4)
3221225985
```

6 Obtenir plus de détails lors de l'échec de la création de l'instance

Lors de la création des objets Adresse/Réseau/Interface en utilisant les fonctions de fabrication agnostiques à la version, n'importe quelle erreur va être signalée en tant que `ValueError` avec un message d'erreur générique qui dit simplement que la valeur entrée n'a pas été reconnue en tant qu'objet de ce type. Pour fournir plus de détails sur la cause du rejet, il faudrait reconnaitre si la valeur est *supposée* être une adresse IPv4 ou IPv6.

Pour gérer les cas d'usage où il est utile d'avoir accès à ces détails, les constructeurs individuels des classes lèvent actuellement les sous-classes de `ValueError`, `ipaddress.AddressValueError` et `ipaddress.NetmaskValueError` pour indiquer précisément quelle partie de la définition n'a pas pu être correctement traitée.

Les messages d'erreur sont particulièrement plus détaillés lors de l'utilisation directe du constructeur. Par exemple :

```
>>> ipaddress.ip_address("192.168.0.256")
Traceback (most recent call last):
  ...
ValueError: '192.168.0.256' does not appear to be an IPv4 or IPv6 address
>>> ipaddress.IPv4Address("192.168.0.256")
Traceback (most recent call last):
  ...
ipaddress.AddressValueError: Octet 256 (> 255) not permitted in '192.168.0.256'

>>> ipaddress.ip_network("192.168.0.1/64")
Traceback (most recent call last):
  ...
ValueError: '192.168.0.1/64' does not appear to be an IPv4 or IPv6 network
>>> ipaddress.IPv4Network("192.168.0.1/64")
Traceback (most recent call last):
  ...
ipaddress.NetmaskValueError: '64' is not a valid netmask
```

Cependant, les exceptions spécifiques des deux modules ont `ValueError` comme classe parent ; donc si vous n'êtes pas intéressé par le type particulier d'erreur remontée, vous pouvez écrire votre code comme suit :

```
try:
    network = ipaddress.IPv4Network(address)
except ValueError:
    print('address/netmask is invalid for IPv4:', address)
```