Introducción al modulo ipaddress

Versión 3.10.13

Guido van Rossum
and the Python development team

noviembre 14, 2023

Python Software Foundation
Email: docs@python.org

Índice general

autor Peter Moody

autor Nick Coghlan

Descripción

Este documento tiene como objetivo proporcionar una introducción apacible al módulo `ipaddress`. Está dirigido principalmente a los usuarios que no están familiarizados con la terminología IP de redes, pero también puede ser útil para los ingenieros de red que quieren una visión general de cómo `ipaddress` representa los conceptos de direccionamiento IP de red.

1 Creando objetos Dirección/Red/Interfaz (*Address/Network/Interface*)

Siendo `ipaddress` un módulo para inspeccionar y manipular direcciones IP, la primer cosa que usted querrá hacer es crear algunos objetos. Puede utilizar `ipaddress` para crear objetos a partir de cadenas de caracteres y enteros.

1.1 Nota sobre versiones IP

Para los lectores que no están particularmente familiarizados con el direccionamiento IP, es importante saber que el Protocolo de Internet (IP) se encuentra actualmente en el proceso de pasar de la versión 4 del protocolo a la versión 6. Esta transición se produce en gran parte porque la versión 4 del protocolo no proporciona suficientes direcciones para satisfacer las necesidades de todo el mundo, especialmente dado el creciente número de dispositivos con conexiones directas a Internet.

Explicando los detalles de las diferencias entre las dos versiones del protocolo está más allá del alcance de esta introducción, pero los lectores deben al menos ser conscientes de que estas dos versiones existen, y a veces será necesario forzar el uso de una versión u otra.

1.2 Direcciones IP del Host

Las direcciones, a menudo denominadas «direcciones de host» son la unidad más básica cuando se trabaja con direccionamiento IP. La forma más sencilla de crear direcciones es usar la función de fabrica `ipaddress.ip_address()`, que determina automáticamente si se crea una dirección IPv4 o IPv6 en función del valor pasado:

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

Las direcciones también se pueden crear directamente a partir de enteros. Los valores que caben dentro de 32 bits se asume que son direcciones IPv4:

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

Para forzar el uso de direcciones IPv4 o IPv6, las clases relevantes se pueden invocar directamente. Esto es particularmente útil para forzar la creación de direcciones IPv6 para enteros pequeños:

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

1.3 Definiendo Redes

Las direcciones de host generalmente se agrupan en redes IP, por lo que `ipaddress` proporciona una forma de crear, inspeccionar y manipular definiciones de red. Los objetos de red IP se construyen a partir de cadenas que definen el rango de direcciones de host que forman parte de esa red. La forma más simple para esa información es un par de «dirección de red/prefijo de red», donde el prefijo define el número de bits iniciales que se comparan para determinar si una dirección es parte de la red y la dirección de red define el valor esperado de esos bits.

En cuanto a las direcciones, se proporciona una función de fábrica que determina automáticamente la versión IP correcta:

```
>>> 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')
```

Los objetos de red no pueden tener ningún bit de host establecido. El efecto práctico de esto es que `192.0.2.1/24` no describe una red. Tales definiciones se conocen como objetos de interfaz ya que la notación *ip-on-a-network* se usa comúnmente para describir interfaces de red de un ordenador en una red determinada y se describen más adelante en la siguiente sección.

De forma predeterminada, al intentar crear un objeto de red con los bits de host establecidos, se lanzará un `ValueError`. Para solicitar que los bits adicionales se coaccionen a cero, el flag `strict=False` se puede pasar al constructor:

```
>>> 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')
```

Si bien la forma de cadena de caracteres ofrece mucha más flexibilidad, las redes también se pueden definir con enteros, al igual que las direcciones de host. En este caso, se considera que la red contiene solo la dirección única identificada por el entero, por lo que el prefijo de red incluye toda la dirección de red:

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

Al igual que con las direcciones, la creación de un tipo particular de red se puede forzar llamando directamente al constructor de clase en lugar de usar la función de fábrica.

1.4 Interfaces de Host

Como se mencionó anteriormente, si necesita describir una dirección en una red en particular, ni la dirección ni las clases de red son suficientes. La notación como `192.0.2.1/24` es comúnmente utilizada por los ingenieros de red y las personas que escriben herramientas para cortafuegos y enrutadores como abreviatura de «el host `192.0.2.1` en la red `192.0.2.0/24`», En consecuencia, `ipaddress` proporciona un conjunto de clases híbridas que asocian una dirección con una red en particular. La interfaz para la creación es idéntica a la de definir objetos de red, excepto que la parte de dirección no está restringida a ser una dirección de red.

```
>>> 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')
```

Se aceptan entradas enteras (como con las redes), y el uso de una versión IP particular se puede forzar llamando directamente al constructor relevante.

2 Inspeccionando objetos Dirección/Red/Interfaz (*Address/Network/Interface*)

Se ha tomado la molestia de crear un objeto *IPv(4|6)(Address|Network|Interface)*, por lo que probablemente desee obtener información al respecto. `ipaddress` intenta hacer esto fácil e intuitivo.

Extrayendo la versión IP:

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

Obteniendo la red desde una interfaz:

```
>>> 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')
```

Averiguando cuántas direcciones individuales hay en una red:

```
>>> 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
```

Iterando a través de las direcciones «utilizables» en una red:

```
>>> 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
```

Obteniendo la máscara de red (es decir, establecer bits correspondientes al prefijo de red) o la máscara de host (cualquier bits que no forme parte de la máscara de red):

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.netmask
```

(continué en la próxima página)

(proviene de la página anterior)

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

Expandiendo o comprimiendo la dirección:

```
>>> 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'
```

Si bien IPv4 no admite expansión o compresión, los objetos asociados aún proporcionan las propiedades relevantes para que el código neutral de la versión pueda garantizar fácilmente que se use la forma más concisa o detallada para las direcciones IPv6 mientras se maneja correctamente las direcciones IPv4.

3 Redes como listas de direcciones

A veces es útil tratar las redes como listas. Esto significa que es posible indexarlas de esta manera:

```
>>> 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')
```

También significa que los objetos de red se prestan a usar la sintaxis del test de lista de membresía como esta:

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

Las pruebas de contención se realizan de manera eficiente según el prefijo de red:

```
>>> 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 Comparaciones

`ipaddress` proporciona algunas formas simples e intuitivas de comparar objetos, donde esto tiene sentido:

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

Se genera una excepción `TypeError` si intenta comparar objetos de diferentes versiones o tipos diferentes.

5 Uso de direcciones IP con otros módulos

Otros módulos que usan direcciones IP (como `socket`) generalmente no aceptarán directamente objetos de este módulo. En su lugar, deben ser forzados a un entero o una cadena que el otro módulo deberá aceptar:

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

6 Obtener más detalles cuando se produce un error en la creación de instancias

Al crear objetos de dirección/red/interfaz utilizando las funciones de fábrica independientes de la versión, cualquier error se informará como `ValueError` con un mensaje de error genérico que simplemente dice que el valor pasado no se reconoció como un objeto de ese tipo. La falta de un error específico se debe a que es necesario saber si *se supone* que el valor es IPv4 o IPv6 para poder proporcionar más detalles sobre por qué se ha rechazado.

Para admitir casos de uso en los que es útil tener acceso a este detalle adicional, los constructores de clase individuales en realidad lanza las subclases `ValueError` `ipaddress.AddressValueError` y `ipaddress.NetmaskValueError` para indicar exactamente qué parte de la definición no se pudo analizar correctamente.

Los mensajes de error son significativamente más detallados cuando se usan los constructores de clase directamente. Por ejemplo:

```
>>> 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
```

Sin embargo, ambas excepciones específicas del módulo tienen `ValueError` como su clase principal, por lo que si no le preocupa el tipo particular de error, aún puede escribir código como el siguiente:

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