Uma introdução ao módulo ipaddress

Release 3.8.20

Guido van Rossum
and the Python development team

setembro 08, 2024

Python Software Foundation
Email: docs@python.org

Sumário

autor Peter Moody

autor Nick Coghlan

Visão Geral

Este documento tem como objetivo prover uma suave introdução ao módulo `ipaddress`. Ele é direcionado primeiramente para usuários que ainda não são familiarizados com a terminologia de rede IP, mas também pode ser útil para engenheiros de rede que querem uma visão geral de como `ipaddress` representa os conceitos de endereçamento de rede IP.

1 Criando objetos de Endereço/Rede/Interface

Como `ipaddress` é um módulo para inspecionar e manipular endereços IP, a primeira coisa que você deseja fazer é criar alguns objetos. Você pode usar `ipaddress` para criar objetos a partir de strings e inteiros.

1.1 Uma observação sobre versões do IP

Para leitores que não estão particularmente familiarizados com endereçamento IP, é importante saber que o Protocolo de Internet está atualmente no processo de mudança da versão 4 para a versão 6 do protocolo. Esta transição está ocorrendo em grande parte porque a versão 4 do protocolo não fornece endereços suficientes para atender as necessidades do mundo todo, especialmente considerando o crescente número de dispositivos com conexões diretas a internet.

Explicar os detalhes das diferenças entre as duas versões do protocolo está além do escopo desta introdução, mas os leitores precisam pelo menos estar cientes de que essas duas versões existem, e às vezes será necessário forçar o uso de uma versão ou do outro.

1.2 Endereços IP de host

Os endereços, muitas vezes chamados de "endereços de host", são a unidade mais básica ao trabalhar com endereçamento IP. A maneira mais simples de criar endereços é usar a função de fábrica `ipaddress.ip_address()`, que determina automaticamente se deve ser criado um endereço IPv4 ou IPv6 com base no valor passado:

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

Os endereços também podem ser criados diretamente a partir de números inteiros. Os valores que cabem em 32 bits são considerados endereços IPv4:

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

Para forçar o uso de endereços IPv4 ou IPv6, as classes relevantes podem ser invocadas diretamente. Isto é particularmente útil para forçar a criação de endereços IPv6 para números inteiros pequenos:

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

1.3 Definindo redes

Endereços de host geralmente são agrupados em redes IP, então `ipaddress` fornece uma maneira de criar, inspecionar e manipular definições de rede. Os objetos de rede IP são construídos a partir de strings que definem o intervalo de endereços de host que fazem parte dessa rede. A forma mais simples para essa informação é um par de "endereço de rede/prefixo de rede", onde o prefixo define o número de bits iniciais que são comparados para determinar se um endereço faz ou não parte da rede e o endereço de rede define o valor esperado daqueles bits.

Quanto aos endereços, é fornecida uma função de fábrica que determina automaticamente a versão correta do 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')
```

Os objetos de rede não podem ter nenhum bit de host definido. O efeito prático disso é que `192.0.2.1/24` não descreve uma rede. Tais definições são chamadas de objetos interface, uma vez que a notação "ip-em-uma-rede" é comumente usada para descrever interfaces de rede de um computador em uma determinada rede e é descrita mais detalhadamente na próxima seção.

Por padrão, tentar criar um objeto rede com bits de host definidos resultará na exceção `ValueError` ser levantada. Para solicitar que os bits adicionais sejam forçados a zero, o sinalizador `strict=False` pode ser passada para o construtor:

```
>>> 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')
```

Embora o formato string ofereça significativamente mais flexibilidade, as redes também podem ser definidas com números inteiros, assim como os endereços de host. Neste caso, considera-se que a rede contém apenas o endereço único identificado pelo número inteiro, portanto o prefixo da rede inclui todo o endereço da rede:

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

Tal como acontece com os endereços, a criação de um tipo específico de rede pode ser forçada chamando diretamente o construtor da classe em vez de usar a função de fábrica.

1.4 Interfaces do host

Conforme mencionado acima, se você precisar descrever um endereço em uma rede específica, nem o endereço nem as classes de rede serão suficientes. Notação como `192.0.2.1/24` é comumente usada por engenheiros de rede e pessoas que escrevem ferramentas para firewalls e roteadores como uma abreviação para "o host `192.0.2.1` na rede `192.0.2.0/24`", Consequentemente, `ipaddress` fornece um conjunto de classes híbridas que associam um endereço a uma rede específica. A interface para criação é idêntica àquela para definição de objetos de rede, exceto que a parte do endereço não está restrita a ser um endereço de rede.

```
>>> 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')
```

Entradas de inteiros são aceitas (como acontece com redes), e o uso de uma versão IP específica pode ser forçado chamando diretamente o construtor relevante.

2 Inspecionando objetos endereço/rede/interface

Você se deu ao trabalho de criar um objeto IPv(4|6)(Address|Network|Interface) e provavelmente deseja obter informações sobre ele. `ipaddress` tenta tornar isso fácil e intuitivo.

Extraindo a versão do IP:

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

Obtendo a rede de uma 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')
```

Descobrindo quantos endereços individuais estão em uma rede:

```
>>> 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 através dos endereços "utilizáveis" em uma rede:

```
>>> 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
```

Obtendo a máscara de rede (ou seja, definir bits correspondentes ao prefixo da rede) ou a máscara de host (qualquer bit que não faça parte da máscara de rede):

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

Explodindo ou compactando o endereço:

```
>>> 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'
```

Embora o IPv4 não tenha suporte a explosão ou compactação, os objetos associados ainda fornecem as propriedades relevantes para que o código de versão neutra possa facilmente garantir que o formato mais conciso ou mais detalhado seja usado para endereços IPv6, ao mesmo tempo que manipula corretamente os endereços IPv4.

3 Networks as lists of Addresses

It's sometimes useful to treat networks as lists. This means it is possible to index them like this:

```
>>> 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')
```

It also means that network objects lend themselves to using the list membership test syntax like this:

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

Containment testing is done efficiently based on the network prefix:

```
>>> 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 Comparações

`ipaddress` provides some simple, hopefully intuitive ways to compare objects, where it makes sense:

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

A `TypeError` exception is raised if you try to compare objects of different versions or different types.

5 Using IP Addresses with other modules

Other modules that use IP addresses (such as `socket`) usually won't accept objects from this module directly. Instead, they must be coerced to an integer or string that the other module will accept:

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

6 Getting more detail when instance creation fails

When creating address/network/interface objects using the version-agnostic factory functions, any errors will be reported as `ValueError` with a generic error message that simply says the passed in value was not recognized as an object of that type. The lack of a specific error is because it's necessary to know whether the value is *supposed* to be IPv4 or IPv6 in order to provide more detail on why it has been rejected.

To support use cases where it is useful to have access to this additional detail, the individual class constructors actually raise the `ValueError` subclasses `ipaddress.AddressValueError` and `ipaddress.NetmaskValueError` to indicate exactly which part of the definition failed to parse correctly.

The error messages are significantly more detailed when using the class constructors directly. For example:

```
>>> 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
```

However, both of the module specific exceptions have `ValueError` as their parent class, so if you're not concerned with the particular type of error, you can still write code like the following:

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