

---

# ipaddress モジュールの紹介

リリース 3.12.9

Guido van Rossum and the Python development team

4 月 08, 2025

## 目次

1	アドレス/ネットワーク/インターフェースオブジェクトの作成	2
1.1	IP バージョンについての注意	2
1.2	IP ホストアドレス	2
1.3	ネットワークを定義する	3
1.4	ホストインターフェース	4
2	アドレス/ネットワーク/インターフェースオブジェクト	4
3	アドレスのリストとしてのネットワーク	6
4	比較	6
5	他のモジュールで IP アドレスを使用する	7
6	インスタンスの生成に失敗した場合にさらに詳細を取得する	7

---

author  
Peter Moody

author  
Nick Coghlan

### 概要

このドキュメントは `ipaddress` モジュールへのやさしい入門を提供するのが目的です。これは主にまだ IP ネットワーキングの用語に詳しくないユーザーを対象としていますが、`ipaddress` がどのように IP ネットワークアドレス処理の概念を表現しているかをざっと見るのにも役立つかもしれません。

# 1 アドレス/ネットワーク/インターフェースオブジェクトの作成

`ipaddress` は IP アドレスを検査したり操作したりするためのモジュールであるため、まずはオブジェクトを作成したほうが良いでしょう。文字列および整数を使用して `ipaddress` を使用してオブジェクトを作成できます。

## 1.1 IP バージョンについての注意

IP アドレッシングに詳しくない読者にとっては、インターネットプロトコル (IP) が現在バージョン 4 プロトコルからバージョン 6 へと移行している途中であることを知っておくのが重要です。この移行はプロトコルのバージョン 4 が世界中の需要に対応するために十分なアドレスを提供できないことが大きな理由で起こっており、特にインターネットへ直接接続する機器が増えている状況を踏まえたものです。

上述した 2 つのプロトコルバージョンの違いの詳細を説明するのはこの入門ドキュメントの対象範囲を超えていますが、読者は少なくともこれらの 2 つのバージョンが存在することと、どちらかのバージョンを強制的に使用することが必要になることもあるということを認識する必要があります。

## 1.2 IP ホストアドレス

アドレスは多くの場合「ホストアドレス」と呼ばれ、IP アドレッシングを扱う場合の最も基本的な単位です。アドレスを作成する単純な方法は `ipaddress.ip_address()` ファクトリー関数を使用することで、この方法では渡された値をもとに IPv4 か IPv6 アドレスのどちらかを自動的に決定します:

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

アドレスは整数から直接作成することも可能です。32 ビットに収まる値は IPv4 アドレスであるとみなされます:

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

IPv4 または IPv6 アドレスの使用を強制するには、適切なクラスを直接呼び出す必要があります。これは特に IPv6 アドレスを小さな整数で作成する場合に役立ちます:

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

### 1.3 ネットワークを定義する

ホストアドレスは通常 IP ネットワークへ統合され、`ipaddress` はネットワーク定義の作成、検査、操作の手段を提供します。IP ネットワークオブジェクトはネットワークの一部であるホストアドレスの範囲を定義する文字列から生成されます。この情報の最も単純な形式は「ネットワークアドレス/ネットワークプレフィックス」のペアであり、プレフィックスは先頭のビット数として定義され、アドレスがネットワークの一部であるかどうか、それらのビットに期待される値がネットワークアドレスに定義されているかを判断するために比較されます。

アドレスについては、正しい IP バージョンを自動的に判断するファクトリー関数が提供されます:

```
>>> 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')
```

ネットワークオブジェクトにはホストビットを設定することができません。これについての実践的な効果は `192.0.2.1/24` がネットワークを表さないということです。次の項でより詳しく説明しますが、ネットワーク内 IP 表記は一般的に特定のネットワーク上のネットワークインターフェースを表すのに使用されるため、このような定義はインターフェースオブジェクトと呼ばれます。

デフォルトでは、ホストビットセットの指定でネットワークオブジェクトを作成しようとする `ValueError` が発生する結果となります。`strict=False` フラグをコンストラクタに渡すと、追加分のビットをゼロとして強制的に処理するように要求できます:

```
>>> 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')
```

文字列形式では劇的に柔軟性を提供できる一方、ネットワークはちょうどホストアドレスのように整数で定義することもできます。この場合、ネットワークは整数により特定できる単一のアドレスのみを含むものとして処理され、ネットワークプレフィックスはネットワークアドレスの全体を含みます:

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

アドレスについては、ファクトリー関数を呼ぶ代わりにクラスコンストラクタを直接呼ぶことで、作成対象のネットワークを特定の種類に強制することができます。

## 1.4 ホストインターフェース

直前に述べたように、特定のネットワーク上のアドレスを示したい場合には、アドレスもネットワーククラスも十分ではありません。192.0.2.1/24 のような表記は「192.0.2.0/24 のネットワーク上の 192.0.2.1 のホスト」の短縮形として通常ネットワークエンジニアやファイアウォールやルータ用のツールを書く人が使用し、`ipaddress` はそれに応じて特定のネットワークとアドレスを関連づけるハイブリッドクラス集を提供します。アドレス部分がネットワークアドレスであることに制限されないことを除き、作成用のインターフェースはネットワーク定義用のインターフェースと同一です。

```
>>> 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')
```

整数入力 (ネットワークと同様に) 受け入れられ、適切なコンストラクタを直接呼び出すことで特定の IP バージョンの使用を強制できます。

## 2 アドレス/ネットワーク/インターフェースオブジェクト

さて、IPv(4|6)(アドレス|ネットワーク|インターフェース) オブジェクトをわざわざ作成したので、多分それについて情報がほしいことでしょう。`ipaddress` はこの作業を簡単に直観的にできるようにします。

IP バージョンを抽出する:

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

インターフェースからネットワークを取得する:

```
>>> 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')
```

個別なアドレスがいくつネットワークに存在するかを調べる:

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4.num_addresses
256
```

(次のページに続く)

(前のページからの続き)

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

「使用可能」なネットワーク上のアドレスを順番にたどる:

```
>>> 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
```

ネットマスクを取得 (例: ネットワークプレフィックスに応じてビットを設定) したりホストマスク (ネットマスクの一部ではない任意のビット) を取得する:

```
>>> 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')
```

アドレスを展開したり圧縮したりする:

```
>>> 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'
```

IPv4 は展開や圧縮には対応していませんが、それでも関連したオブジェクトが適切なプロパティを提供し、バージョンに対して中立なコードが IPv4 アドレスを正しく処理する一方で IPv6 向けには最も簡潔または最も詳細な形式が使用されることを保証します。

### 3 アドレスのリストとしてのネットワーク

ネットワークをリストとして扱うと便利な場合があります。これは、ネットワークを次のようにインデックス付けすることが可能なことを意味します:

```
>>> 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')
```

以下のようなリストのメンバーシップの判定用文法が使用可能なように、ネットワークオブジェクト自身が適応します:

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

ネットワークプレフィックスに基づき、所属しているかどうかの判定が効果的にできます:

```
>>> 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 比較

`ipaddress` はオブジェクトの比較をするための単純で、かつ願わくば直観的な方法を意味が通じるように提供します:

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

異なるバージョンや異なるタイプのオブジェクトを比較しようとすると `TypeError` 例外が発生します。

## 5 他のモジュールで IP アドレスを使用する

IP アドレスを使用する他のモジュール (`socket` など) は通常このモジュールからオブジェクトを直接受け付けません。直接渡すのではなく、他のモジュールが受け付ける整数や文字列に強制的に変換する必要があります:

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

## 6 インスタンスの生成に失敗した場合にさらに詳細を取得する

上述したバージョンに関知しないファクトリー関数を使用してアドレス/ネットワーク/インターフェースオブジェクトを作成する場合、渡された値がそのタイプのオブジェクトとして認識されなかったと単純に通知する汎用的なメッセージとともに、どのエラーでも `ValueError` として報告されます。特定のエラーが提供されないのは、拒否された理由をより詳細に提供するために、値が IPv4 または IPv6 である **はず** かどうかを知る必要があるためです。

追加の詳細を得ると役立つ場合のユースケースに対応するため、個別のクラスのコンストラクタは実際に `ValueError` のサブクラスである `ipaddress.AddressValueError` と `ipaddress.NetmaskValueError` を発生させて具体的に定義のどの部分のパーズに失敗したのかを示します。

クラスコンストラクタを直接使用する場合にはこれらのエラーメッセージはさらに非常に詳細になっています。例えば:

```
>>> 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
```

しかし、モジュールについて固有の上記例外のどちらも `ValueError` を親クラスとして持ち、エラーの特定のタイプに興味がない場合でも、以下のようにコードを記述できます:

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