

---

# ipaddress 모듈에 대한 소개

출시 버전 3.10.18

Guido van Rossum  
and the Python development team

7월 08, 2025

## Contents

1	주소/네트워크/인터페이스 객체 만들기	2
1.1	IP 버전에 대한 참고 사항	2
1.2	IP 호스트 주소	2
1.3	네트워크 정의	2
1.4	호스트 인터페이스	3
2	주소/네트워크/인터페이스 객체 검사	4
3	주소 리스트로서의 네트워크	5
4	비교	5
5	다른 모듈과 함께 IP 주소 사용하기	6
6	인스턴스 생성 실패 시 세부 사항 가져오기	6

---

저자 Peter Moody

저자 Nick Coghlan

### 개요

이 문서에서는 ipaddress 모듈을 간략하게 소개하고자 합니다. 주로 IP 네트워킹 용어에 익숙하지 않은 사용자를 대상으로 하지만, ipaddress가 IP 네트워크 주소 개념을 나타내는 방식에 대한 개요를 원하는 네트워크 엔지니어에게 유용할 수도 있습니다.

# 1 주소/네트워크/인터페이스 객체 만들기

ipaddress는 IP 주소를 검사하고 조작하는 모듈이기 때문에, 가장 먼저 하고 싶어 할 일은 몇몇 객체를 만드는 것입니다. ipaddress를 사용하여 문자열과 정수로 객체를 만들 수 있습니다.

## 1.1 IP 버전에 대한 참고 사항

특히 IP 주소 지정에 익숙하지 않은 독자는, 인터넷 프로토콜(IP)이 현재 프로토콜 버전 4에서 버전 6으로 이동하는 과정에 있음을 아는 것이 중요합니다. 이러한 전환은 주로 프로토콜 버전 4가 전 세계의 요구 사항을 처리할 수 있는 충분한 주소를 제공하지 못하기 때문에 발생하고 있습니다. 특히 인터넷에 직접 연결되는 장치의 수가 증가함에 따라 더욱더 그렇습니다.

프로토콜의 두 버전 간의 차이점에 대한 자세한 설명은 이 소개의 범위를 벗어나지만, 독자는 최소한 이 두 버전이 존재한다는 사실을 알고 있어야 하며, 때로는 한 버전이나 다른 버전을 강제로 사용해야 할 필요가 있습니다.

## 1.2 IP 호스트 주소

주소, 종종 “호스트 주소” 라고 하는 것은 IP 주소 지정으로 작업할 때 가장 기본 단위입니다. 주소를 만드는 가장 간단한 방법은 ipaddress.ip\_address() 팩토리 함수를 사용하는 것인데, 전달된 값을 기반으로 IPv4나 IPv6 주소 중 어느 것을 만들지 자동으로 결정합니다:

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

주소는 정수에서 직접 만들 수도 있습니다. 32비트에 들어맞는 값은 IPv4 주소로 간주합니다:

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

IPv4나 IPv6 주소를 강제로 사용하려면, 해당 클래스를 직접 호출할 수 있습니다. 이것은 작은 정수를 위한 IPv6 주소 생성을 강제하는 데 특히 유용합니다:

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

## 1.3 네트워크 정의

호스트 주소는 대개 IP 네트워크로 그룹화되므로, ipaddress는 네트워크 정의를 만들고, 검사하고, 조작할 방법을 제공합니다. IP 네트워크 객체는 해당 네트워크의 일부인 호스트 주소의 범위를 정의하는 문자열로 만들어집니다. 이 정보의 가장 간단한 형식은 “네트워크 주소/네트워크 접두사” 쌍입니다. 접두어는 주소가 네트워크 일부인지 판별하기 위해 비교되는 선행 비트 수를 정의하고, 네트워크 주소는 그 비트들의 기대되는 값을 정의합니다.

주소의 경우, 정확한 IP 버전을 자동으로 결정하는 팩토리 함수가 제공됩니다:

```
>>> 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')
```

네트워크 객체는 호스트 비트가 설정될 수 없습니다. 이것의 실제 효과는 192.0.2.1/24가 네트워크를 설명하지 않는다는 것입니다. 이러한 정의는 인터페이스 객체라고 불리는데, 그 이유는 주어진 네트워크상의 컴퓨터의 네트워크 인터페이스를 기술하기 위해 네트워크상의 IP(ip-on-a-network) 표기법이 일반적으로 사용되기 때문입니다. 자세한 내용은 다음 절에서 설명합니다.

기본적으로, 호스트 비트가 설정된 네트워크 객체를 만들려고 하면 ValueError가 발생합니다. 추가 비트를 강제로 0으로 변환하도록 요청하려면, 플래그 strict=False를 생성자에 전달할 수 있습니다:

```
>>> 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')
```

문자열 형식은 유연성이 훨씬 뛰어나지만, 호스트 주소와 마찬가지로 정수로 네트워크를 정의할 수도 있습니다. 이 경우, 네트워크는 정수로 식별되는 단일 주소만 포함하는 것으로 간주하므로, 네트워크 접두사는 전체 네트워크 주소를 포함합니다:

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

주소와 마찬가지로, 팩토리 함수를 사용하는 대신 클래스 생성자를 직접 호출하여 특정 종류의 네트워크를 만들 수 있습니다.

## 1.4 호스트 인터페이스

위에서 언급했듯이, 특정 네트워크상의 주소를 설명해야 하는 경우, 주소로도 네트워크 클래스로도 충분하지 않습니다. 192.0.2.1/24와 같은 표기법은 네트워크 엔지니어와 방화벽과 라우터 용 도구를 작성하는 사람들이 “네트워크 192.0.2.0/24 상의 호스트 192.0.2.1”의 줄임말로 많이 사용합니다. 따라서, ipaddress는 주소를 특정 네트워크와 결합하는 혼성 클래스 집합을 제공합니다. 생성을 위한 인터페이스는 주소 부분이 네트워크 주소로 제한되지 않는 것을 제외하고는 네트워크 객체를 정의하는 것과 같습니다.

```
>>> 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')
```

정수 입력이 받아들여지고 (네트워크처럼), 특정 IP 버전의 사용은 관련 생성자를 직접 호출함으로써 강제될 수 있습니다.

## 2 주소/네트워크/인터페이스 객체 검사

여러분은 IPv(4/6)(Address|Network|Interface) 객체를 만드는 데 어려움을 겪었다면, 아마도 이에 대한 정보를 얻고자 할 것입니다. `ipaddress`는 이 작업을 쉽고 직관적으로 만들려고 합니다.

IP 버전 추출하기:

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

인터페이스에서 네트워크 얻기:

```
>>> 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')
```

네트워크에 있는 개별 주소의 개수 찾기:

```
>>> 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
```

네트워크에서 “사용 가능한” 주소 이터레이트하기:

```
>>> 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
```

넷 마스크(`netmask`)(즉, 네트워크 접두사에 해당하는 비트들)나 호스트 마스크(`hostmask`)(넷 마스크에 포함되지 않은 비트들) 얻기:

```
>>> 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')
```

주소를 펼치거나 압축하기:

```
>>> 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'
```

IPv4는 펄치기와 압축을 지원하지 않지만, 연관된 객체는 여전히 관련 프로퍼티를 제공하므로 버전 중립적인 코드가 IPv4 주소를 올바르게 처리하면서도 IPv6 주소에 대해 가장 간결하거나 가장 자세한 형식을 쉽게 사용할 수 있습니다.

### 3 주소 리스트로서의 네트워크

네트워크를 리스트로 취급하는 것이 때로 유용합니다. 즉, 다음과 같이 인덱싱할 수 있습니다:

```
>>> 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')
```

이것은 또한 네트워크 객체가 다음과 같은 리스트 멤버십 테스트 문법을 사용하는 데 적합하다는 것을 의미합니다:

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

포함 테스트는 네트워크 접두어를 기반으로 효율적으로 수행됩니다:

```
>>> 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 비교

ipaddress는 의미가 있는 곳에서 객체를 비교하는 간단하고 직관적인 방법을 제공합니다:

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

다른 버전이나 다른 형의 객체를 비교하려고 하면 `TypeError` 예외가 발생합니다.

## 5 다른 모듈과 함께 IP 주소 사용하기

IP 주소를 사용하는 다른 모듈(가령 socket)은 일반적으로 이 모듈의 객체를 직접 받아들이지 않습니다. 대신, 다른 모듈이 받아들일 수 있는 정수나 문자열로 강제 변환되어야 합니다:

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

## 6 인스턴스 생성 실패 시 세부 사항 가져오기

버전에 구애받지 않는 팩토리 함수를 사용하여 주소/네트워크/인터페이스 객체를 만들 때, 단순히 전달된 값이 해당 형의 객체로 인식되지 않는다는 일반 에러 메시지와 함께 예러가 ValueError로 보고됩니다. 구체적인 예러가 없는 이유는 거부된 이유에 대한 자세한 정보를 제공하기 위해서는 값이 IPv4나 IPv6 중 어느 것으로 가정되는지를 알아야 하기 때문입니다.

이 추가 세부 정보를 액세스하는 것이 유용한 사용 사례를 지원하기 위해, 개별 클래스 생성자는 실제로 ValueError 서브 클래스 ipaddress.AddressValueError와 ipaddress.NetmaskValueError를 발생시켜 정의의 어느 부분에서 구문 분석하는 데 실패했는지 정확히 가리킵니다.

에러 메시지는 클래스 생성자를 직접 사용할 때 훨씬 자세해집니다. 예를 들어:

```
>>> 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
```

그러나, 두 모듈 특정 예외 모두 부모 클래스로 ValueError를 가지므로, 특정 유형의 예러에 관심이 없다면, 여전히 다음과 같은 코드를 작성할 수 있습니다:

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