


ХОЧУ ПОМОЧЬ  
ПРОЕКТУ

Модуль ipaddress в Python




CW cutwoodshop.ru

РЕКЛАМА

3D карта мира из дерева по специальной цене!

5,0 ★ Рейтинг организации ⓘ

Купить



С / Модуль ipaddress в Python

Создание объектов IPv4 и IPv6 адресами, сетями и интерфейсами

Модуль предоставляет возможности для создания, управления и работы с адресами и сетями IPv4 и IPv6. Он упрощает выполнение различных задач, связанных с IP-адресами, включая проверку того, принадлежит ли адрес определенной подсети, итерацию по всем хостам в определенной подсети, проверку того, представляет ли адрес или определение сети и т. д.

Содержание:

- Создание объектов IP-адреса, сети и интерфейса;
  - Определение IP-адресов;
  - Определение IP-сетей;
  - Определение IP-интерфейса;
- Операции с объектами IP-адреса, сети и интерфейса.;
  - Объект сети как список IP-адресов;
  - Сравнение IP-адресов;
  - Преобразование объекта IP-адреса в строку и число;
- Информация при сбое создания IP-адреса, сети и интерфейса.

Создание объектов IP-адреса, сети и интерфейса.

Так как ipaddress - это модуль для проверки и управления IP-адресами, первое, что нужно сделать, это создать несколько объектов. Объекты модуля ipaddress можно создавать из [строк](#) и [целых чисел](#).

*Примечание* к версиям IP адресов: Для пользователей, которые не знакомы с IP-адресацией, важно знать, что Интернет-протокол в настоящее время находится в процессе перехода от версии 4 к версии 6. Этот переход происходит в основном потому, что версия 4 протокола не предоставляет достаточного количества адресов для удовлетворения потребностей всего мира, особенно учитывая растущее число устройств с прямым подключением к Интернету.

Определение IP-адресов.

Адреса, часто называемые "адресами хостов", являются основной единицей при работе с IP-адресацией. Самый простой способ создания адресов - использовать [фабричную функцию ipaddress.ip\\_address\(\)](#), которая автоматически определяет, следует ли создавать адрес IPv4 или IPv6 на основе переданного значения:

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

Адреса также можно создавать непосредственно из целых чисел. Предполагается, что значения, которые соответствуют 32 битам, являются адресами IPv4:

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

Чтобы принудительно использовать адреса IPv4 или IPv6, соответствующие классы могут быть вызваны напрямую. Такое поведение полезно для принудительного создания адресов IPv6 для небольших целых чисел:

```
>>> ipaddress.ip_address(1)
# IPv4Address('127.0.0.1')
>>> 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')
```

Объекты IP-сети не могут иметь никаких битов хоста. На практике это означает, что запись 192.0.2.1/24 не описывает сеть. Такие определения называются [объектами сетевых интерфейсов](#), поскольку нотация *ip-on-a-network* обычно используется для описания сетевых интерфейсов компьютера в данной сети.

По умолчанию попытка создать сетевой объект с установленными битами хоста приведет к возникновению [ValueError](#). Чтобы запросить приведение дополнительных битов к нулю, в конструктор необходимо передать флаг `strict=False`:

```
>>> ipaddress.ip_network('192.0.2.1/24')
# 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')
```

Хотя строковая форма предлагает значительно большую гибкость, объекты сети также можно определять с помощью целых чисел, как и [адреса хостов](#). В этом случае считается, что сеть содержит только один адрес, обозначенный целым числом, поэтому префикс сети включает весь сетевой адрес:

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

Как и в случае с адресами, определенный тип сети может быть принудительно создан прямым вызовом конструктора класса вместо использования фабричной функции.

## Определение IP-интерфейса.

Если нужно описать адрес в конкретной сети, ни объекта адреса, ни объекта сети недостаточно. Обозначение типа 192.0.2.1/24, используется сетевыми инженерами для определения правил маршрутизаторов и расшифровывается как "хост 192.0.2.1 в сети 192.0.2.0/24". Соответственно, модуль ipaddress предоставляет набор гибридных классов, которые связывают адрес с определенной сетью. Интерфейс для создания идентичен интерфейсу для определения объектов сети, за исключением того, что часть адреса не ограничивается объектом сетевого адреса.

```
>>> 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')
```

Вверх

Для создания интерфейса могут использоваться целочисленные входные данные (как в случае с сетями), так же создание определенного типа интерфейса хоста может быть создано прямым вызовом конструктора соответствующего класса

## Операции с объектами IP-адреса, сети и интерфейса.

И Р-адреса, сети или интерфейса, и вероятно, необходимо получить информацию о нем. Мо простым и интуитивно понятным.

Из

```
>>> host4 = ipaddress.ip_address('192.0.2.1')
>>> host6 = ipaddress.ip_address('2001:db8::1')
>>>
#
>>>
#
```

По

```
>>> net4 = ipaddress.ip_interface('192.0.2.1/24')
>>>
#
>>> host6 = ipaddress.ip_interface('2001:db8::1/96')
>>> host6.network
# IPv6Network('2001:db8::/96')
```

Узнаем, сколько отдельных адресов в сети:

```
>>> 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
```

Перебор *полезных* адресов в сети:

```
>>> 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
```

Получение сетевой маски (т. е. установки битов, соответствующих префиксу сети) или маски хоста (любых битов, не являющихся частью сетевой маски):

```
>>> 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')
```

Расширение или сжатие IPv6-адреса:

Вверх



```
>>> addr6.exploded
# '2001:0db8:0000:0000:0000:0000:0000:0001'
>>> addr6.compressed
# '2001:db8::1'
>>> addr6.broadcast
# '2001:db8::ffff:ffff:ffff:ffff'
>>> addr6.broadcast_exploded
# '2001:db8:0000:0000:0000:0000:0000:ffff'
```

Хотя расширение или сжатие, связанные объекты по-прежнему предоставляют соответствующие свойства, но по версии код может легко гарантировать, что для адресов IPv6 используется наиболее корректная форма, при этом адреса IPv4, так же обрабатывая правильно.

## Оформление списка IP-адресов.

По умолчанию предоставляется объект сети как списка адресов. Это означает, что их можно индексировать, как и любой список:

```
>>> net4 = ipaddress.ip_network('192.0.2.0/24')
>>> net4[0]
# IPv4Address('192.0.2.1')
>>> net4[-1]
# IPv4Address('192.0.2.255')

>>> net6 = ipaddress.ip_network('2001:db8::0/96')
>>> net6[1]
# IPv6Address('2001:db8::1')
>>> net6[-1]
# IPv6Address('2001:db8::ffff:ffff:ffff:ffff')
```

## Проверка вхождения/членства IP-адреса в сеть.

Сетевые объекты можно проверять на вхождение/членство в списке следующим образом:

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

Вхождение/членство IP-адреса эффективно выполняется на основе префикса сети:

```
>>> 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
```

## Сравнение IP-адресов.

Модуль ipaddress предоставляет несколько простых, интуитивно понятных способов сравнения объектов, где это имеет смысл:

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

Если производится попытка сравнить объекты разных версий или разных типов, то возникает [исключение TypeError](#).

## Преобразование объекта IP-адреса в строку и число.

Другие модули, использующие IP-адреса (например, [socket](#)), обычно не принимают объекты IP-адреса напрямую. Следовательно они должны быть приведены к целому числу или строке, которые примет другой модуль:

```
>>> addr4 = ipaddress.ip_address('192.0.2.1')
>>> str(addr4)
# '192.0.2.1'
>>> int(addr4)
# 1985
```

Вверх

# Информация при сбое создания экземпляра.

При создании объектов адреса, сети или интерфейса с использованием фабричных функций, не зависящих от версии протокола, любые ошибки будут генерировать исключение `ValueError` с общим сообщением об ошибке (переданное значение не этого типа).

Конкретные исключения, возникающие в названии конкретный протокол, при ошибках создания объекта вызывают исключения `ipaddress.NetmaskValueError` и точно указывают, какую часть определения не удалось преобразовать. Эти исключения более детализированы при прямом использовании конструкторов классов. Например:

```
>>> ipaddress.ip_address('192.168.0.256')
# Traceback (most recent call last):
#   File "...", line 1, in <module>
#     ipaddress.ip_address('192.168.0.256')
# ValueError: '192.168.0.256' does not appear to be an IPv4 or IPv6 address

>>> ipaddress.ip_network('192.168.0.1/256')
# Traceback (most recent call last):
#   File "...", line 1, in <module>
#     ipaddress.ip_network('192.168.0.1/256')
# ValueError: Octet 256 (> 255) not permitted in '192.168.0.256'

>>> ipaddress.ip_network('192.168.0.1/64')
# Traceback (most recent call last):
#   File "...", line 1, in <module>
#     ipaddress.ip_network('192.168.0.1/64')
# 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):
#   File "...", line 1, in <module>
#     ipaddress.IPv4Network('192.168.0.1/64')
# ipaddress.NetmaskValueError: '64' is not a valid netmask
```

Оба специфичных для модуля исключения имеют `ValueError` в качестве родительского класса, поэтому, если не беспокоит конкретный тип ошибки, все равно можно написать код, подобный следующему:

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

Содержание раздела:

- [КРАТКИЙ ОБЗОР МАТЕРИАЛА.](#)
- [Функции `ip\_address`, `ip\_network` и `ip\_interface` модуля `ipaddress`](#)
- [Объекты `IPv4Address\(\)` и `IPv6Address\(\)` модуля `ipaddress`](#)
- [Объекты `IPv4Network\(\)` и `IPv6Network\(\)` модуля `ipaddress`](#)
- [Объекты `IPv4Interface\(\)` и `IPv6Interface\(\)` модуля `ipaddress`](#)
- [Функции уровня модуля `ipaddress`](#)