Addressing, Protocol Families and Socket Types

A *socket* is one endpoint of a communication channel used by programs to pass data back and forth locally or across the Internet. Sockets have two primary properties controlling the way they send data: the *address family* controls the OSI network layer protocol used and the *socket type* controls the transport layer protocol.

Python supports three address families. The most common, `AF_INET`, is used for IPv4 Internet addressing. IPv4 addresses are four bytes long and are usually represented as a sequence of four numbers, one per octet, separated by dots (e.g., `10.1.1.5` and `127.0.0.1`). These values are more commonly referred to as "IP addresses." Almost all Internet networking is done using IP version 4 at this time.

`AF_INET6` is used for IPv6 Internet addressing. IPv6 is the "next generation" version of the Internet protocol, and supports 128-bit addresses, traffic shaping, and routing features not available under IPv4. Adoption of IPv6 continues to grow, especially with the proliferation of cloud computing and the extra devices being added to the network because of Internet-of-things projects.

`AF_UNIX` is the address family for Unix Domain Sockets (UDS), an inter-process communication protocol available on POSIX-compliant systems. The implementation of UDS typically allows the operating system to pass data directly from process to process, without going through the network stack. This is more efficient than using `AF_INET`, but because the file system is used as the namespace for addressing, UDS is restricted to processes on the same system. The appeal of using UDS over other IPC mechanisms such as named pipes or shared memory is that the programming interface is the same as for IP networking, so the application can take advantage of efficient communication when running on a single host, but use the same code when sending data across the network.

> **Note**
>
> The `AF_UNIX` constant is only defined on systems where UDS is supported.

The socket type is usually either `SOCK_DGRAM` for message-oriented datagram transport or `SOCK_STREAM` for stream-oriented transport. Datagram sockets are most often associated with UDP, the *user datagram protocol*. They provide unreliable delivery of individual messages. Stream-oriented sockets are associated with TCP, *transmission control protocol*. They provide byte streams between the client and server, ensuring message delivery or failure notification through timeout management, retransmission, and other features.

Most application protocols that deliver a large amount of data, such as HTTP, are built on top of TCP because it makes it simpler to create complex applications when message ordering and delivery is handled automatically. UDP is commonly used for protocols where order is less important (since the messages are self-contained and often small, such as name look-ups via DNS), or for *multicasting* (sending the same data to several hosts). Both UDP and TCP can be used with either IPv4 or IPv6 addressing.

> **Note**
>
> Python's `socket` module supports other socket types but they are less commonly used, so are not covered here. Refer to the standard library documentation for more details.

Looking up Hosts on the Network

`socket` includes functions to interface with the domain name services on the network so a program can convert the host name of a server into its numerical network address. Applications do not need to convert addresses explicitly before using them to connect to a server, but it can be useful when reporting errors to include the numerical address as well as the name value being used.

To find the official name of the current host, use `gethostname()`.

```
# socket_gethostname.py

import socket

print(socket.gethostname())
```

The name returned will depend on the network settings for the current system, and may change if it is on a different network (such as a laptop attached to a wireless LAN).

```
$ python3 socket_gethostname.py

apu.hellfly.net
```

Use `gethostbyname()` to consult the operating system hostname resolution API and convert the name of a server to its numerical address.

```
# socket_gethostbyname.py

import socket

HOSTS = [
    'apu',
    'pymotw.com',
    'www.python.org',
    'nosuchname',
]

for host in HOSTS:
    try:
        print('{} : {}'.format(host, socket.gethostbyname(host)))
    except socket.error as msg:
        print('{} : {}'.format(host, msg))
```

If the DNS configuration of the current system includes one or more domains in the search, the name argument does not need to be a fully qualified name (i.e., it does not need to include the domain name as well as the base hostname). If the name cannot be found, an exception of type `socket.error` is raised.

```
$ python3 socket_gethostbyname.py

apu : 10.9.0.10
pymotw.com : 66.33.211.242
www.python.org : 151.101.32.223
nosuchname : [Errno 8] nodename nor servname provided, or not
known
```

For access to more naming information about a server, use `gethostbyname_ex()`. It returns the canonical hostname of the server, any aliases, and all of the available IP addresses that can be used to reach it.

```
# socket_gethostbyname_ex.py

import socket

HOSTS = [
    'apu',
    'pymotw.com',
    'www.python.org',
    'nosuchname',
]

for host in HOSTS:
    print(host)
    try:
        name, aliases, addresses = socket.gethostbyname_ex(host)
        print('  Hostname:', name)
        print('  Aliases :', aliases)
        print(' Addresses:', addresses)
    except socket.error as msg:
        print('ERROR:', msg)
    print()
```

Having all known IP addresses for a server lets a client implement its own load balancing or fail-over algorithms.

```
$ python3 socket_gethostbyname_ex.py

apu
```

```
apu
  Hostname: apu.hellfly.net
  Aliases : ['apu']
 Addresses: ['10.9.0.10']

pymotw.com
  Hostname: pymotw.com
  Aliases : []
 Addresses: ['66.33.211.242']

www.python.org
  Hostname: prod.python.map.fastlylb.net
  Aliases : ['www.python.org', 'python.map.fastly.net']
 Addresses: ['151.101.32.223']

nosuchname
ERROR: [Errno 8] nodename nor servname provided, or not known
```

Use `getfqdn()` to convert a partial name to a fully qualified domain name.

```
# socket_getfqdn.py

import socket

for host in ['apu', 'pymotw.com']:
    print('{:>10} : {}'.format(host, socket.getfqdn(host)))
```

The name returned will not necessarily match the input argument in any way if the input is an alias, such as `www` is here.

```
$ python3 socket_getfqdn.py

       apu : apu.hellfly.net
pymotw.com : apache2-echo.catalina.dreamhost.com
```

When the address of a server is available, use `gethostbyaddr()` to do a "reverse" lookup for the name.

```
# socket_gethostbyaddr.py

import socket

hostname, aliases, addresses = socket.gethostbyaddr('10.9.0.10')

print('Hostname :', hostname)
print('Aliases  :', aliases)
print('Addresses:', addresses)
```

The return value is a tuple containing the full hostname, any aliases, and all IP addresses associated with the name.

```
$ python3 socket_gethostbyaddr.py

Hostname : apu.hellfly.net
Aliases  : ['apu']
Addresses: ['10.9.0.10']
```

Finding Service Information

In addition to an IP address, each socket address includes an integer *port number*. Many applications can run on the same host, listening on a single IP address, but only one socket at a time can use a port at that address. The combination of IP address, protocol, and port number uniquely identify a communication channel and ensure that messages sent through a socket arrive at the correct destination.

Some of the port numbers are pre-allocated for a specific protocol. For example, communication between email servers using SMTP occurs over port number 25 using TCP, and web clients and servers use port 80 for HTTP. The port numbers for network services with standardized names can be looked up with `getservbyname()`.

```
# socket_getservbyname.py

import socket
```

```
from urllib.parse import urlparse

URLS = [
    'http://www.python.org',
    'https://www.mybank.com',
    'ftp://prep.ai.mit.edu',
    'gopher://gopher.micro.umn.edu',
    'smtp://mail.example.com',
    'imap://mail.example.com',
    'imaps://mail.example.com',
    'pop3://pop.example.com',
    'pop3s://pop.example.com',
]

for url in URLS:
    parsed_url = urlparse(url)
    port = socket.getservbyname(parsed_url.scheme)
    print('{:>6} : {}'.format(parsed_url.scheme, port))
```

Although a standardized service is unlikely to change ports, looking up the value with a system call instead of hard-coding it is more flexible when new services are added in the future.

```
$ python3 socket_getservbyname.py

  http : 80
 https : 443
   ftp : 21
gopher : 70
  smtp : 25
  imap : 143
 imaps : 993
  pop3 : 110
 pop3s : 995
```

To reverse the service port lookup, use `getservbyport()`.

```
# socket_getservbyport.py

import socket
from urllib.parse import urlunparse

for port in [80, 443, 21, 70, 25, 143, 993, 110, 995]:
    url = '{}://example.com/'.format(socket.getservbyport(port))
    print(url)
```

The reverse lookup is useful for constructing URLs to services from arbitrary addresses.

```
$ python3 socket_getservbyport.py

http://example.com/
https://example.com/
ftp://example.com/
gopher://example.com/
smtp://example.com/
imap://example.com/
imaps://example.com/
pop3://example.com/
pop3s://example.com/
```

The number assigned to a transport protocol can be retrieved with `getprotobyname()`.

```
# socket_getprotobyname.py

import socket


def get_constants(prefix):
    """Create a dictionary mapping socket module
    constants to their names.
```

```
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
    }


protocols = get_constants('IPPROTO_')

for name in ['icmp', 'udp', 'tcp']:
    proto_num = socket.getprotobyname(name)
    const_name = protocols[proto_num]
    print('{:>4} -> {:2d} (socket.{:<12} = {:2d})'.format(
        name, proto_num, const_name,
        getattr(socket, const_name)))
```

The values for protocol numbers are standardized, and defined as constants in `socket` with the prefix `IPPROTO_`.

```
$ python3 socket_getprotobyname.py

icmp ->  1 (socket.IPPROTO_ICMP =  1)
 udp -> 17 (socket.IPPROTO_UDP  = 17)
 tcp ->  6 (socket.IPPROTO_TCP  =  6)
```

Looking Up Server Addresses

`getaddrinfo()` converts the basic address of a service into a list of tuples with all of the information necessary to make a connection. The contents of each tuple will vary, containing different network families or protocols.

```
# socket_getaddrinfo.py

import socket


def get_constants(prefix):
    """Create a dictionary mapping socket module
    constants to their names.
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
    }


families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')

for response in socket.getaddrinfo('www.python.org', 'http'):

    # Unpack the response tuple
    family, socktype, proto, canonname, sockaddr = response

    print('Family        :', families[family])
    print('Type          :', types[socktype])
    print('Protocol      :', protocols[proto])
    print('Canonical name:', canonname)
    print('Socket address:', sockaddr)
    print()
```

This program demonstrates how to look up the connection information for `www.python.org`.

```
$ python3 socket_getaddrinfo.py

Family        : AF_INET
Type          : SOCK_DGRAM
Protocol      : IPPROTO_UDP
```

```
Canonical name:
Socket address: ('151.101.32.223', 80)

Family        : AF_INET
Type          : SOCK_STREAM
Protocol      : IPPROTO_TCP
Canonical name:
Socket address: ('151.101.32.223', 80)

Family        : AF_INET6
Type          : SOCK_DGRAM
Protocol      : IPPROTO_UDP
Canonical name:
Socket address: ('2a04:4e42:8::223', 80, 0, 0)

Family        : AF_INET6
Type          : SOCK_STREAM
Protocol      : IPPROTO_TCP
Canonical name:
Socket address: ('2a04:4e42:8::223', 80, 0, 0)
```

`getaddrinfo()` takes several arguments for filtering the result list. The `host` and `port` values given in the example are required arguments. The optional arguments are `family`, `socktype`, `proto`, and `flags`. The optional values should be either `0` or one of the constants defined by `socket`.

```
# socket_getaddrinfo_extra_args.py
```

```
import socket


def get_constants(prefix):
    """Create a dictionary mapping socket module
    constants to their names.
    """
    return {
        getattr(socket, n): n
        for n in dir(socket)
        if n.startswith(prefix)
    }


families = get_constants('AF_')
types = get_constants('SOCK_')
protocols = get_constants('IPPROTO_')

responses = socket.getaddrinfo(
    host='www.python.org',
    port='http',
    family=socket.AF_INET,
    type=socket.SOCK_STREAM,
    proto=socket.IPPROTO_TCP,
    flags=socket.AI_CANONNAME,
)

for response in responses:
    # Unpack the response tuple
    family, socktype, proto, canonname, sockaddr = response

    print('Family        :', families[family])
    print('Type          :', types[socktype])
    print('Protocol      :', protocols[proto])
    print('Canonical name:', canonname)
    print('Socket address:', sockaddr)
    print()
```

Since `flags` includes `AI_CANONNAME`, the canonical name of the server, which may be different from the value used for the lookup if the host has any aliases, is included in the results this time. Without the flag, the canonical name value is left empty.

```
$ python3 socket_getaddrinfo_extra_args.py
```

```
Family        : AF_INET
Type          : SOCK_STREAM
Protocol      : IPPROTO_TCP
Canonical name: prod.python.map.fastlylb.net
Socket address: ('151.101.32.223', 80)
```

IP Address Representations

Network programs written in C use the data type `struct sockaddr` to represent IP addresses as binary values (instead of the string addresses usually found in Python programs). To convert IPv4 addresses between the Python representation and the C representation, use `inet_aton()` and `inet_ntoa()`.

```
# socket_address_packing.py

import binascii
import socket
import struct
import sys

for string_address in ['192.168.1.1', '127.0.0.1']:
    packed = socket.inet_aton(string_address)
    print('Original:', string_address)
    print('Packed  :', binascii.hexlify(packed))
    print('Unpacked:', socket.inet_ntoa(packed))
    print()
```

The four bytes in the packed format can be passed to C libraries, transmitted safely over the network, or saved to a database compactly.

```
$ python3 socket_address_packing.py

Original: 192.168.1.1
Packed  : b'c0a80101'
Unpacked: 192.168.1.1

Original: 127.0.0.1
Packed  : b'7f000001'
Unpacked: 127.0.0.1
```

The related functions `inet_pton()` and `inet_ntop()` work with both IPv4 and IPv6 addresses, producing the appropriate format based on the address family parameter passed in.

```
# socket_ipv6_address_packing.py

import binascii
import socket
import struct
import sys

string_address = '2002:ac10:10a:1234:21e:52ff:fe74:40e'
packed = socket.inet_pton(socket.AF_INET6, string_address)

print('Original:', string_address)
print('Packed  :', binascii.hexlify(packed))
print('Unpacked:', socket.inet_ntop(socket.AF_INET6, packed))
```

An IPv6 address is already a hexadecimal value, so converting the packed version to a series of hex digits produces a string similar to the original value.

```
$ python3 socket_ipv6_address_packing.py

Original: 2002:ac10:10a:1234:21e:52ff:fe74:40e
Packed  : b'2002ac10010a1234021e52fffe74040e'
Unpacked: 2002:ac10:10a:1234:21e:52ff:fe74:40e
```

See also

- Wikipedia: IPv6 – Article discussing Internet Protocol Version 6 (IPv6)

- Wikipedia: IPv6 – Article discussing Internet Protocol version 6 (IPv6).
- Wikipedia: OSI Networking Model – Article describing the seven layer model of networking implementation.
- Assigned Internet Protocol Numbers – List of standard protocol names and numbers.

socket — Network Communication

TCP/IP Client and Server

Quick Links

This page was last updated 2017-01-01.

Navigation

socket — Network Communication
TCP/IP Client and Server

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example