macaddress

Aug 01, 2021

Contents:

CHAPTER 1

Introduction

Media access control (MAC) addresses play an important role in local-area networks. They also pack a lot of information into 48-bit hexadecimal strings!

The macaddress library makes it easy to evaluate the properties of MAC addresses and the extended identifiers of which they are subclasses.

CHAPTER 2

Installing macaddress

macaddress is available on GitHub at https://github.com/critical-path/macaddress.

If you do not have pip version 18.1 or higher, then run the following command from your shell.

```
[user@host ~]$ sudo pip install --upgrade pip
```

To install macaddress with test-related dependencies, run the following command from your shell.

```
[user@host ~]$ sudo pip install --editable git+https://github.com/critical-path/
↪macaddress.git#egg=macaddress[test]
```

To install it without test-related dependencies, run the following command from your shell.

```
[user@host ~]$ sudo pip install git+https://github.com/critical-path/macaddress.git
```

(If necessary, replace `pip` with `pip3`.)

CHAPTER 3

Using macaddress

While macaddress contains multiple classes, the only one with which you need to interact directly is `MediaAccessControlAddress`.

Import `MediaAccessControlAddress`.

```
>>> from macaddress import MediaAccessControlAddress
```

Instantiate `MediaAccessControlAddress` by passing in a MAC address in plain, hyphen, colon, or dot notation.

```
>>> mac = MediaAccessControlAddress("a0b1c2d3e4f5")
```

```
>>> mac = MediaAccessControlAddress("a0-b1-c2-d3-e4-f5")
```

```
>>> mac = MediaAccessControlAddress("a0:b1:c2:d3:e4:f5")
```

```
>>> mac = MediaAccessControlAddress("a0b1.c2d3.e4f5")
```

To determine whether the MAC address is a broadcast, a multicast (layer-two), or a unicast address, access its `is_broadcast`, `is_multicast`, and `is_unicast` properties.

```
>>> print(mac.is_broadcast)
False
```

```
>>> print(mac.is_multicast)
False
```

```
>>> print(mac.is_unicast)
True
```

To determine whether the MAC address is a universally-administered address (UAA) or a locally-administered address (LAA), access its `is_uaa` and `is_laa` properties.

```
>>> print(mac.is_uaa)
True
```

```
>>> print(mac.is_laa)
False
```

To work with the MAC address's octets, access its `octets` property, which contains six `Octet` objects.

```
>>> print(mac.octets)
[Octet('a0'), Octet('b1'), Octet('c2'), Octet('d3'), Octet('e4'), Octet('f5')]
```

To determine whether the MAC address is an extended unique identifier (EUI), an extended local identifier (ELI), or unknown, access its `type` property.

```
>>> print(mac.type)
unique
```

To determine whether the MAC address has an organizationally-unique identifier (OUI) or a company ID (CID), access its `has_oui` and `has_cid` properties.

```
>>> print(mac.has_oui)
True
```

```
>>> print(mac.has_cid)
False
```

To view the decimal equivalent of the MAC address, access its `decimal` property.

```
>>> print(mac.decimal)
176685338322165
```

To view the binary equivalent of the MAC address, access its `binary` and `reverse_binary` properties. With `binary`, the most-significant digit of each octet appears first. With `reverse_binary`, the least-significant digit of each octet appears first.

```
>>> print(mac.binary)
101000001011000111000010110100111111001001111010 1
```

```
>>> print(mac.reverse_binary)
000001011000110101000011110010110010011110101111
```

To return the MAC address's two "fragments," call the `to_fragments` method. For an EUI, this means the 24-bit OUI as the first fragment and the remaining interface-specific bits as the second fragment. For an ELI, this means the 24-bit CID as the first fragment and the remaining interface-specific bits as the second fragment.

```
>>> fragments = mac.to_fragments()
>>> print(fragments)
('a0b1c2', 'd3e4f5')
```

To return the MAC address in different notations, call the `to_plain_notation`, `to_hyphen_notation`, `to_colon_notation`, and `to_dot_notation` methods.

```
>>> plain = mac.to_plain_notation()
>>> print(plain)
a0b1c2d3e4f5
```

```
>>> hyphen = mac.to_hyphen_notation()
>>> print(hyphen)
a0-b1-c2-d3-e4-f5
```

```
>>> colon = mac.to_colon_notation()
>>> print(colon)
a0:b1:c2:d3:e4:f5
```

```
>>> dot = mac.to_dot_notation()
>>> print(dot)
a0b1.c2d3.e4f5
```

CHAPTER 4

Patterns for macaddress

4.1 Create a range of MAC addresses

```
# Import `pprint.pprint` and `macaddress.MediaAccessControlAddress`.

>>> from pprint import pprint
>>> from macaddress import MediaAccessControlAddress

# Identify the start and end of the range.

>>> start_mac = MediaAccessControlAddress("a0b1c2d3e4f5")
>>> end_mac = MediaAccessControlAddress("a0b1c2d3e4ff")

# Create a list containing one `MediaAccessControlAddress` object
# for each address in the range.

>>> mac_range = [
...   MediaAccessControlAddress(format(decimal, "x"))
...   for decimal in range(start_mac.decimal, end_mac.decimal + 1)
... ]

# Do something useful with the results, such as returning
# the colon notation of each MAC address in the list.

>>> colons = [
...     mac.to_colon_notation() for mac in mac_range
... ]
>>> pprint(colons)
["a0:b1:c2:d3:e4:f5",
 "a0:b1:c2:d3:e4:f6",
 "a0:b1:c2:d3:e4:f7",
 "a0:b1:c2:d3:e4:f8",
 "a0:b1:c2:d3:e4:f9",
 "a0:b1:c2:d3:e4:fa",
```

(continues on next page)

(continued from previous page)

```
 "a0:b1:c2:d3:e4:fb",
 "a0:b1:c2:d3:e4:fc",
 "a0:b1:c2:d3:e4:fd",
 "a0:b1:c2:d3:e4:fe",
 "a0:b1:c2:d3:e4:ff"]
```

4.2 Map-reduce a list of MAC addresses

```
# Import `functools.reduce`, `pprint.pprint`, and
# `macaddress.MediaAccessControlAddress`.

>>> from functools import reduce
>>> from pprint import pprint
>>> from macaddress import MediaAccessControlAddress

# Define `transform`, which is our map function.

>>> def transform(mac, attributes):
...     transformed = {}
...     transformed[mac.normalized] = {}
...     for attribute in attributes:
...         transformed[mac.normalized][attribute] = getattr(mac, attribute)
...     return transformed
...

# Define `fold`, which is our reduce function.

>>> def fold(current_mac, next_mac):
...     for key, value in next_mac.items():
...         if key in current_mac:
...             pass
...         else:
...             current_mac[key] = value
...     return current_mac
...

# Define `map_reduce`, which calls `functools.reduce`, `transform`, and `fold`.

>>> def map_reduce(macs, attributes):
...     return reduce(fold, [transform(mac, attributes) for mac in macs])
...

# Identify addresses of interest.

>>> addresses = [
...     "a0:b1:c2:d3:e4:f5",
...     "a0:b1:c2:d3:e4:f6",
...     "a0:b1:c2:d3:e4:f7",
...     "a0:b1:c2:d3:e4:f8",
...     "a0:b1:c2:d3:e4:f9",
...     "a0:b1:c2:d3:e4:fa",
...     "a0:b1:c2:d3:e4:fb",
...     "a0:b1:c2:d3:e4:fc",
...     "a0:b1:c2:d3:e4:fd",
```

(continues on next page)

(continued from previous page)

```
...     "a0:b1:c2:d3:e4:fe",
...     "a0:b1:c2:d3:e4:ff"
... ]

# Create a list containing one `MediaAccessControlAddress` object
# for each address of interest.

>>> macs = [
...     MediaAccessControlAddress(address) for address in addresses
... ]

# Create a list with attributes of interest.

>>> attributes = [
...     "is_unicast",
...     "is_uaa"
... ]

# Call `map_reduce`, passing in the lists of `MediaAccessControlAddress`
# objects and attributes.

>>> mapped_reduced = map_reduce(macs, attributes)
>>> pprint(mapped_reduced)
{"a0b1c2d3e4f5": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4f6": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4f7": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4f8": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4f9": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4fa": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4fb": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4fc": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4fd": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4fe": {"is_uaa": True, "is_unicast": True},
 "a0b1c2d3e4ff": {"is_uaa": True, "is_unicast": True}}
```

4.3 Serialize the attributes of a MAC address

```
# Import `json.dumps`.

>>> from json import dumps

# Identify the addresses and attributes of interest.

>>> unserialized = {
...     "a0b1c2d3e4f5": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4f6": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4f7": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4f8": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4f9": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4fa": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4fb": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4fc": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4fd": {"is_uaa": True, "is_unicast": True},
...     "a0b1c2d3e4fe": {"is_uaa": True, "is_unicast": True},
```

(continues on next page)

(continued from previous page)

```
...     "a0b1c2d3e4ff": {"is_uaa": True, "is_unicast": True}
... }

# Call `json.dumps` on the unserialized addresses.

>>> serialized = dumps(unserialized, indent=2)
>>> print(serialized)
{
  "a0b1c2d3e4f5": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4f6": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4f7": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4f8": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4f9": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4fa": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4fb": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4fc": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4fd": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4fe": {
    "is_uaa": true,
    "is_unicast": true
  },
  "a0b1c2d3e4ff": {
    "is_uaa": true,
    "is_unicast": true
  }
}
```

CHAPTER 5

Testing macaddress

To conduct testing, run the following commands from your shell.

```
[user@host macaddress]$ flake8 --count --ignore E125 macaddress
[user@host macaddress]$ pytest --cov --cov-report=term-missing
```

CHAPTER 6

Other languages

The macaddress library is also available in the following languages:

- JavaScript
- Ruby
- Rust

CHAPTER 7

macaddress

7.1 macaddress package

7.1.1 macaddress.macaddress module

This module includes MediaAccessControlAddress and AddressError.

exception macaddress.macaddress.**AddressError**
Bases: Exception

MediaAccessControlAddress raises AddressError if instantiated with an invalid argument.

Parameters **message** (*str*) – A human-readable error message.

class macaddress.macaddress.**MediaAccessControlAddress**(*address*)
Bases: *macaddress.ei48.ExtendedIdentifier48*

MediaAccessControlAddress makes it easy to work with media access control (MAC) addresses.

is_broadcast
Whether the MAC address is a broadcast address.

"ffffffffffff" = broadcast.

Type bool

is_multicast
Whether the MAC address is a multicast address (layer-two multicast, not layer-three multicast).

The least-significant bit in the first octet of a MAC address determines whether it is a multicast or a unicast.

1 = multicast.

Type bool

is_unicast
Whether the MAC address is a unicast address.

The least-significant bit in the first octet of a MAC address determines whether it is a multicast or a unicast.

0 = unicast.

Type bool

is_uaa
Whether the MAC address is a universally-administered address (UAA).

The second-least-significant bit in the first octet of a MAC address determines whether it is a UAA or an LAA.

0 = UAA.

Type bool

is_laa
Whether the MAC address is a locally-administered address (LAA).

The second-least-significant bit in the first octet of a MAC address determines whether it is a UAA or an LAA.

1 = LAA.

Type bool

7.1.2 macaddress.ei48 module

This module includes ExtendedIdentifier48 and IdentifierError.

exception macaddress.ei48.**IdentifierError**
Bases: Exception

ExtendedIdentifier48 raises IdentifierError if instantiated with an invalid argument.

Parameters **message** (*str*) – A human-readable error message.

class macaddress.ei48.**ExtendedIdentifier48**(*identifier*)
Bases: object

ExtendedIdentifier48 makes it easy to work with the IEEE's 48-bit extended unique identifiers (EUI) and extended local identifiers (ELI).

The first 24 or 36 bits of an EUI is called an organizationally- unique identifier (OUI), while the first 24 or 36 bits of an ELI is called a company ID (CID).

Visit the IEEE's website for more information on EUIs and ELIs.

Helpful link: https://standards.ieee.org/products-services/regauth/tut/index.html

original
The hexadecimal identifier passed in by the user.

Type str

normalized
The hexadecimal identifier after replacing all uppercase letters with lowercase letters and removing all hypens, colons, and dots.

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *a0b1c2d3e4f5*.

Type str

is_valid
Whether the user passed in a valid hexadecimal identifier.

Type bool

octets
Each of the hexadecimal identifier's six octets.

Type list

first_octet
The hexadecimal identifier's first octet.

Type *Octet*

type
The hexadecimal identifier's type, where type is unique, local, or unknown.

The two least-significant bits in the first octet of an extended identifier determine whether it is an EUI.

00 = unique.

The four least-signficant bits in the first octet of an extended identifier determine whether it is an ELI.

1010 = local.

Type str

has_oui
Whether the hexadecimal identifier has an OUI.

If the identifier is an EUI, then it has an OUI.

Type bool

has_cid
Whether the hexadecimal identifier has a CID.

If the identifier is an ELI, then it has a CID.

Type bool

decimal
The decimal equivalent of the hexadecimal digits passed in by the user.

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *176685338322165*.

Type int

binary
The binary equivalent of the hexadecimal identifier passed in by the user. *The most-significant digit of each octet appears first.*

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *101000001011000111000010110100111110010011110101*.

Type str

reverse_binary
The reverse-binary equivalent of the hexadecimal identifier passed in by the user. *The least-significant digit of each octet appears first.*

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *000001011000110101000011110010110010011110101111*.

Type str

Parameters **identifier** (*str*) – Twelve hexadecimal digits (0-9, A-F, or a-f).

Raises *IdentifierError*

to_fragments(*bits=24*)
Returns the hexadecimal identifier's two "fragments."

For an EUI, this means the 24- or 36-bit OUI as the first fragment and the remaining device- or object-specific bits as the second fragment.

For an ELI, this means the 24- or 36-bit CID as the first fragment and the remaining device- or object-specific bits as the second fragment.

For example, if the user passes in *A0-B1-C2-D3-E4-F5* and calls this method with either *bits=24* or no keyword argument, then ExtendedIdentifier48 will return *(a0b1c2, d3e4f5)*.

If the user passes in *A0-B1-C2-D3-E4-F5* and calls this method with *bits=36*, then ExtendedIdentifier48 will return *(a0b1c2d3e, 4f5)*.

Parameters **bits** (*int*) – The number of bits for the OUI or CID.

The default value is 24.

to_plain_notation()
Returns the hexadecimal identifier in plain notation (for example, *a0b1c2d3e4f5*).

to_hyphen_notation()
Returns the hexadecimal identifier in hyphen notation (for example, *a0-b1-c2-d3-e4-f5*).

to_colon_notation()
Returns the hexadecimal identifier in colon notation (for example, *a0:b1:c2:d3:e4:f5*).

to_dot_notation()
Returns the hexadecimal identifier in dot notation (for example, *a0b1.c2d3.e4f5*).

7.1.3 macaddress.octet module

This module includes Octet and OctetError.

exception macaddress.octet.**OctetError**
Bases: Exception

Octet raises OctetError if instantiated with an invalid argument.

Parameters **message** (*str*) – A human-readable error message.

class macaddress.octet.**Octet**(*digits*)
Bases: object

Octet makes it easy to convert two hexadecimal digits to eight binary or reverse-binary digits.

This is useful when working with the IEEE's extended unique identifiers and extended local identifiers.

original
The hexadecimal digits passed in by the user.

Type str

normalized
The hexadecimal digits after replacing all uppercase letters with lowercase letters.

For example, if the user passes in *A0*, then Octet will return *a0*.

Type str

is_valid
Whether the user passed in valid hexadecimal digits.

Type bool

decimal
The decimal equivalent of the hexadecimal digits passed in by the user.

For example, if the user passes in *A0*, then Octet will return *160*.

Type int

binary
The binary equivalent of the hexadecimal digits passed in by the user. *The most-significant digit appears first.*

For example, if the user passes in *A0*, then Octet will return *10100000*.

Type str

reverse_binary
The reverse-binary equivalent of the hexadecimal digits passed in by the user. *The least-significant digit appears first.*

For example, if the user passes in *A0*, then Octet will return *00000101*.

Type str

Parameters **digits** (*str*) – Two hexadecimal digits (0-9, A-F, or a-f).

Raises *OctetError*

7.1.4 Module contents

The macaddress library makes it easy to work with media access control (MAC) addresses.

class macaddress.**ExtendedIdentifier48**(*identifier*)
Bases: object

ExtendedIdentifier48 makes it easy to work with the IEEE's 48-bit extended unique identifiers (EUI) and extended local identifiers (ELI).

The first 24 or 36 bits of an EUI is called an organizationally- unique identifier (OUI), while the first 24 or 36 bits of an ELI is called a company ID (CID).

Visit the IEEE's website for more information on EUIs and ELIs.

Helpful link: https://standards.ieee.org/products-services/regauth/tut/index.html

original
The hexadecimal identifier passed in by the user.

Type str

normalized
The hexadecimal identifier after replacing all uppercase letters with lowercase letters and removing all hypens, colons, and dots.

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *a0b1c2d3e4f5*.

Type str

is_valid
Whether the user passed in a valid hexadecimal identifier.

Type bool

octets
Each of the hexadecimal identifier's six octets.

Type list

first_octet
The hexadecimal identifier's first octet.

Type *Octet*

type
The hexadecimal identifier's type, where type is unique, local, or unknown.

The two least-significant bits in the first octet of an extended identifier determine whether it is an EUI.

00 = unique.

The four least-signficant bits in the first octet of an extended identifier determine whether it is an ELI.

1010 = local.

Type str

has_oui
Whether the hexadecimal identifier has an OUI.

If the identifier is an EUI, then it has an OUI.

Type bool

has_cid
Whether the hexadecimal identifier has a CID.

If the identifier is an ELI, then it has a CID.

Type bool

decimal
The decimal equivalent of the hexadecimal digits passed in by the user.

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *176685338322165*.

Type int

binary
The binary equivalent of the hexadecimal identifier passed in by the user. *The most-significant digit of each octet appears first.*

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *101000001011000111000010110100111110010011110101*.

Type str

reverse_binary
The reverse-binary equivalent of the hexadecimal identifier passed in by the user. *The least-significant digit of each octet appears first.*

For example, if the user passes in *A0-B1-C2-D3-E4-F5*, then ExtendedIdentifier48 will return *000001011000110101000011110010110010011110101111*.

Type str

Parameters **identifier** (*str*) – Twelve hexadecimal digits (0-9, A-F, or a-f).

Raises *IdentifierError*

to_fragments(*bits=24*)
Returns the hexadecimal identifier's two "fragments."

For an EUI, this means the 24- or 36-bit OUI as the first fragment and the remaining device- or object-specific bits as the second fragment.

For an ELI, this means the 24- or 36-bit CID as the first fragment and the remaining device- or object-specific bits as the second fragment.

For example, if the user passes in *A0-B1-C2-D3-E4-F5* and calls this method with either *bits=24* or no keyword argument, then ExtendedIdentifier48 will return *(a0b1c2, d3e4f5)*.

If the user passes in *A0-B1-C2-D3-E4-F5* and calls this method with *bits=36*, then ExtendedIdentifier48 will return *(a0b1c2d3e, 4f5)*.

Parameters **bits** (*int*) – The number of bits for the OUI or CID.

The default value is 24.

to_plain_notation()
Returns the hexadecimal identifier in plain notation (for example, *a0b1c2d3e4f5*).

to_hyphen_notation()
Returns the hexadecimal identifier in hyphen notation (for example, *a0-b1-c2-d3-e4-f5*).

to_colon_notation()
Returns the hexadecimal identifier in colon notation (for example, *a0:b1:c2:d3:e4:f5*).

to_dot_notation()
Returns the hexadecimal identifier in dot notation (for example, *a0b1.c2d3.e4f5*).

class macaddress.**MediaAccessControlAddress**(*address*)
Bases: *macaddress.ei48.ExtendedIdentifier48*

MediaAccessControlAddress makes it easy to work with media access control (MAC) addresses.

is_broadcast
Whether the MAC address is a broadcast address.

"ffffffffffff" = broadcast.

Type bool

is_multicast
Whether the MAC address is a multicast address (layer-two multicast, not layer-three multicast).

The least-significant bit in the first octet of a MAC address determines whether it is a multicast or a unicast.

1 = multicast.

Type bool

is_unicast
Whether the MAC address is a unicast address.

The least-significant bit in the first octet of a MAC address determines whether it is a multicast or a unicast.

0 = unicast.

Type bool

is_uaa
Whether the MAC address is a universally-administered address (UAA).

The second-least-significant bit in the first octet of a MAC address determines whether it is a UAA or an LAA.

0 = UAA.

Type bool

is_laa
Whether the MAC address is a locally-administered address (LAA).

The second-least-significant bit in the first octet of a MAC address determines whether it is a UAA or an LAA.

1 = LAA.

Type bool

class macaddress.**Octet**(*digits*)
Bases: object

Octet makes it easy to convert two hexadecimal digits to eight binary or reverse-binary digits.

This is useful when working with the IEEE's extended unique identifiers and extended local identifiers.

original
The hexadecimal digits passed in by the user.

Type str

normalized
The hexadecimal digits after replacing all uppercase letters with lowercase letters.

For example, if the user passes in *A0*, then Octet will return *a0*.

Type str

is_valid
Whether the user passed in valid hexadecimal digits.

Type bool

decimal
The decimal equivalent of the hexadecimal digits passed in by the user.

For example, if the user passes in *A0*, then Octet will return *160*.

Type int

binary
The binary equivalent of the hexadecimal digits passed in by the user. *The most-significant digit appears first.*

For example, if the user passes in *A0*, then Octet will return *10100000*.

Type str

reverse_binary
The reverse-binary equivalent of the hexadecimal digits passed in by the user. *The least-significant digit appears first.*

For example, if the user passes in *A0*, then Octet will return *00000101*.

Type str

Parameters **digits** (*str*) – Two hexadecimal digits (0-9, A-F, or a-f).

Raises *OctetError*

exception macaddress.**IdentifierError**
Bases: Exception

ExtendedIdentifier48 raises IdentifierError if instantiated with an invalid argument.

Parameters **message** (*str*) – A human-readable error message.

exception macaddress.**AddressError**
Bases: Exception

MediaAccessControlAddress raises AddressError if instantiated with an invalid argument.

Parameters **message** (*str*) – A human-readable error message.

exception macaddress.**OctetError**
Bases: Exception

Octet raises OctetError if instantiated with an invalid argument.

Parameters **message** (*str*) – A human-readable error message.

CHAPTER 8

Indices

- genindex
- modindex
- search

Python Module Index

m

Index

N

O

R

T