Перечень всех «белых» IP-адресов Интернета

Однажды передо мной встала задача: на произвольном роутере локальной сети выделить интернет-трафик пользователей и перенаправить его по другому маршруту, не трогая при этом локальный трафик.

В принципе банальщина, решить которую можно множеством способов. Например, для локального трафика задать статические маршруты, а шлюзом по умолчанию сделать роутер, куда нужно перенаправить трафик. Схема рабочая, но трудоёмкая. Ведь для каждого роутера придётся вручную корректировать таблицы маршрутизации.

Хотелось сделать универсальный скрипт, с помощью которого можно было бы легко и быстро управлять трафиком. Идея в том, что можно задавать статические маршруты не для локального трафика, а для внешнего, и при этом не трогать шлюз по

умолчанию. Дело осталось за малым – получить перечень всех внешних IP-подсетей, чтобы на их основе строить маршруты.

Ну и как в жизни бывает, автоматизация работы привела к гораздо большим трудозатратам, нежели явная работа руками. Проблема оказалась там, где не ждали – готового перечня внешних IP-подсетей найти не удалось. А раз нет, значит придется делать самому.

Что значит внешние IP-подсети? По-научному это подсети, содержащие IP-адреса глобально доступных узлов, или, как их называют в простонародье, подсети «белых IP-адресов».

Получить искомый перечень можно разными способами:

1. Взять список всех публичных автономных систем (ASN) и извлечь из него искомые IP-подсети.
2. Вычесть из всего адресного пространства (0.0.0.0 – 255.255.255.255) список специальных, глобально недоступных IP-адресов.

Я решил использовать второй способ, как более простой. Дело в том, что специальные адреса определяются стандартами, которые практически неизменны, а список и содержимое автономных систем изменяются с течением времени. Дело осталось за малым – пробежаться по стандартам и выписать искомые адреса.

Первым попавшимся мне стандартом оказался RFC: 3704 (BCP: 84) Ingress Filtering for Multihomed Networks. В нём указано, что интернет-провайдеры не должны маршрутизировать в Интернет «марсианские IP-адреса», включающий в себя адреса:

```
0.0.0.0/8
10.0.0.0/8
127.0.0.0/8
172.16.0.0/12
192.168.0.0/16
224.0.0.0/4
240.0.0.0/4
```

Хорошо, первый блок специальных адресов получен, но есть ощущение, что это далеко не всё. Название следующего стандарта RFC: 3330 Special-Use IPv4 Addresses обнадёживало, что на нем поиски закончатся. Однако, после прочтения мне удалось пополнить список совсем немного:

```
169.254.0.0/16
192.0.2.0/24
192.88.99.0/24
```

198.18.0.0/15

И тут мне стало грустно. Прочитав два стандарта, я понял, что они содержат лишь небольшие порции зарезервированных адресов, и ни один не является исчерпывающим. Например, ни в первом, ни во втором не было диапазона 100.64.0.0/10 (Carrier grade NAT), который также относится к специальным глобально недоступным адресам.

Я решил, что нужно искать не стандарты, а ресурс, где были бы приведены сразу все немаршрутизируемые IP-адреса. Благо, такой ресурс нашёлся. IANA выпускает и поддерживает полный перечень IP-адресов специального назначения (Рисунок 1).

Address Block	Name	RFC	Allocation Date	Termination Date	Source	Destination	Forwardable	Globally Reachable	Reserved-by-Protocol
0.0.0.0/8	"This network"	[RFC791], Section 3.2	1981-09	N/A	True	False	False	False	True
0.0.0.0/32	"This host on this network"	[RFC1122], Section 3.2.1.3	1981-09	N/A	True	False	False	False	True
10.0.0.0/8	Private-Use	[RFC1918]	1996-02	N/A	True	True	True	False	False
100.64.0.0/10	Shared Address Space	[RFC6598]	2012-04	N/A	True	True	True	False	False
127.0.0.0/8	Loopback	[RFC1122], Section 3.2.1.3	1981-09	N/A	False [1]	False [1]	False [1]	False [1]	True
169.254.0.0/16	Link Local	[RFC3927]	2005-05	N/A	True	True	False	False	True
172.16.0.0/12	Private-Use	[RFC1918]	1996-02	N/A	True	True	True	False	False
192.0.0.0/24 [2]	IETF Protocol Assignments	[RFC6890], Section 2.1	2010-01	N/A	False	False	False	False	False
192.0.0.0/29	IPv4 Service Continuity Prefix	[RFC7335]	2011-06	N/A	True	True	True	False	False
192.0.0.8/32	IPv4 dummy address	[RFC7600]	2015-03	N/A	True	False	False	False	False
192.0.0.9/32	Port Control Protocol Anycast	[RFC7723]	2015-10	N/A	True	True	True	True	False
192.0.0.10/32	Traversal Using Relays around NAT Anycast	[RFC8155]	2017-02	N/A	True	True	True	True	False
192.0.0.170/32, 192.0.0.171/32	NAT64/DNS64 Discovery	[RFC8880][RFC7050], Section 2.2	2013-02	N/A	False	False	False	False	True
192.0.2.0/24	Documentation (TEST-NET-1)	[RFC5737]	2010-01	N/A	False	False	False	False	False
192.31.196.0/24	AS112-v4	[RFC7535]	2014-12	N/A	True	True	True	True	False
192.52.193.0/24	AMT	[RFC7450]	2014-12	N/A	True	True	True	True	False
192.88.99.0/24	Deprecated (6to4 Relay Anycast)	[RFC7526]	2001-06	2015-03					
192.88.99.2/32	6a44-relay anycast address	[RFC6751]	2012-10	N/A	True	True	True	False	False
192.168.0.0/16	Private-Use	[RFC1918]	1996-02	N/A	True	True	True	False	False
192.175.48.0/24	Direct Delegation AS112 Service	[RFC7534]	1996-01	N/A	True	True	True	True	False
198.18.0.0/15	Benchmarking	[RFC2544]	1999-03	N/A	True	True	True	False	False
198.51.100.0/24	Documentation (TEST-NET-2)	[RFC5737]	2010-01	N/A	False	False	False	False	False
203.0.113.0/24	Documentation (TEST-NET-3)	[RFC5737]	2010-01	N/A	False	False	False	False	False
240.0.0.0/4	Reserved	[RFC1112], Section 4	1989-08	N/A	False	False	False	False	True
255.255.255.255/32	Limited Broadcast	[RFC8190] [RFC919], Section 7	1984-10	N/A	False	True	False	False	True

Footnotes

[1] Several protocols have been granted exceptions to this rule. For examples, see [RFC8029] and [RFC5884].

[2] Not useable unless by virtue of a more specific reservation.

Рисунок 1

Это именно то, что нужно. Здесь интересующие меня адреса отмечены, как «Globally Reachable: FALSE».

Но, как обычно, не обошлось без нескольких ложек дёгтя.

1. Сеть 192.0.0.0/24 является глобально недоступной с двумя исключениями: 192.0.0.9/32 (Port Control Protocol Anycast) и 192.0.0.10/32 (Traversal Using Relays around NAT Anycast).
2. Список не содержит подсеть 224.0.0.0/4 зарезервированную (RFC: 1112 Host Extensions for IP Multicasting) под многоадресный трафик.

Соответственно, передо мной встал вопрос, что делать с этими адресами. С одной стороны, RFC указывает, что 192.0.0.9 и 192.0.0.10 являются глобально доступными, с другой стороны, они являются служебными и, соответственно, ни один реальный

сервер в Интернете не должен иметь их в качестве адреса сетевого интерфейса. Такая же проблема и с многоадресным трафиком. Он может ограничиваться только локальной сетью, а может и выходить за её пределы.

Осознавая тот факт, что красивого универсального решения тут быть не может, я решил:

1. Не делать исключений для 192.0.0.9 и 192.0.0.10 и всю подсеть 192.0.0.0/24 считать глобально недоступной.
2. Подсеть многоадресной рассылки 224.0.0.0/4 также считать глобально недоступной.

Источник данных я нашёл, войну с перфекционизмом провёл, теперь нужно заняться расчётами. Поигравшись полчаса, я понял, что строить искомый перечень вручную – путь в никуда. Во-первых, очень трудоёмко, во-вторых, могут появиться ошибки. Следовательно, нужно автоматизировать.

Для этого я написал Python3 библиотеку **ipaddressspace**, опирающуюся на стандартную ipadrress, и позволяющую проводить математические операции над пространствами IP-адресов с учётом особенностей алгебры множеств.

ipaddressspace.py

```
#!/usr/bin/env python3

"""IPv4 address space maniplation toolkit.

Features:
1. Converting IP address ranges to and from IP networks
2. Addition and subtraction of IP address ranges and address spaces

#Example 1. Show start and end IP addresses for an IP network
r = IPv4Range(ipaddress.IPv4Network("192.137.234.16/31"))
print(r)

#Output:
#192.137.234.16 - 192.137.234.17

#Example 2. Show IP networks covering a range of IP addresses
r = IPv4Range(ipaddress.IPv4Address("1.0.0.1"),
              ipaddress.IPv4Address("1.0.0.8"))
for net in r.networks():
    print(net)

#Output:
#1.0.0.1/32
#1.0.0.2/31
#1.0.0.4/30
#1.0.0.8/32

#Example 3. Addition of address spaces
s1 = IPv4AddressSpace()
s1.append(ipaddress.IPv4Network("1.0.0.0/27"))
```

```
s1 += IPv4Range(ipaddress.IPv4Address("1.0.0.33"),
                ipaddress.IPv4Address("1.0.0.255"))
print(s1)

#Output:
#1.0.0.0 - 1.0.0.31
#1.0.0.33 - 1.0.0.255

s1 += ipaddress.IPv4Address("1.0.0.32")
print(s1)

#Output
#1.0.0.0 - 1.0.0.255

# Example 4. Calculation of the difference between address spaces
s2 = IPv4AddressSpace(ipaddress.IPv4Network("0.0.0.0/0"))
s2.remove(ipaddress.IPv4Network("192.168.0.0/16"))
print(s2)

#Output:
#0.0.0.0 - 192.167.255.255
#192.169.0.0 - 255.255.255.255

s3 = IPv4AddressSpace(ipaddress.IPv4Network("0.0.0.0/0"))
s3.remove(ipaddress.IPv4Network("10.0.0.0/8"))
print(s3)
#Output:
#0.0.0.0 - 9.255.255.255
#11.0.0.0 - 255.255.255.255

s4 = s2-s3
print(s4)
#Output:
#10.0.0.0 - 10.255.255.255

#Example 5. Show IP networks spanning the IP address space
s5 = IPv4AddressSpace(ipaddress.IPv4Network("0.0.0.0/0"))
s5 -= ipaddress.IPv4Network("10.0.0.0/8")
s5 -= ipaddress.IPv4Network("64.0.0.0/2")

print(s5)
#Output
#0.0.0.0 - 9.255.255.255
#11.0.0.0 - 63.255.255.255
#128.0.0.0 - 255.255.255.255

for net in s5.networks():
    print(net)
#Output
#0.0.0.0/5
#8.0.0.0/7
#11.0.0.0/8
#12.0.0.0/6
#16.0.0.0/4
#32.0.0.0/3
#128.0.0.0/1

Licensed under the MIT License (https://opensource.org/license/mit).
(c) 2025, imbasoft.ru
"""

import ipaddress


class IPv4Range:
```

```
    """A range of IPv4 addresses defined by a start IP and an end IP address.

    The object is immutable after creation, changing attributes is not allowed.

    Attributes:
        IP_ADDRESS_BITS_COUNT (int): a constant that specifies the number
            of bits in an IPv4 address
        IP_ADDRESS_MAX_VALUE (int): a constant that specifies int form of
            max IPv4 address (255.255.255.255)
        start_ip (ipaddress.IPv4Address) - start address of range. Must be
            always <= end_ip. Can't be None.
        end_ip (ipaddress.IPv4Address) - end address of range. Must be
            always => start_ip. Can't be None.
    """

    IP_ADDRESS_BITS_COUNT = 32
    IP_ADDRESS_MAX_VALUE = (1 << IP_ADDRESS_BITS_COUNT) - 1

    def __init__(self, *args):
        """
        Object constrution overloaded with args.

        Raises:
            ValueError: if the arguments are wrong: invalid types, start_ip >
                end_ip and etc.
        """
        # one argument can be only ipaddress.IPv4Network
        if 1 == len(args):
            ip_subnet = args[0]
            if not isinstance(ip_subnet, ipaddress.IPv4Network):
                raise ValueError(
                    "Single argument must be an IP address.IPv4Network"
                )
            self.__start_ip = ip_subnet.network_address
            self.__end_ip = ip_subnet.broadcast_address

        # two arguments: first is start IP address, second last IP address
        elif 2 == len(args):
            if not isinstance(args[0], ipaddress.IPv4Address):
                raise ValueError(
                    "First paramater must be an ipaddress.IPv4Address"
                )
            if not isinstance(args[1], ipaddress.IPv4Address):
                raise ValueError(
                    "Second paramater must be an ipaddress.IPv4Address"
                )
            if args[0] > args[1]:
                raise ValueError(
                    "First paramater must be less than or equal to Second"
                )
            self.__start_ip = args[0]
            self.__end_ip = args[1]
        else:
            raise ValueError("Incorrect number of parameters")

    @property
    def start_ip(self):
        """ipaddress.IPv4Address: Start of IP range."""
        return self.__start_ip

    @property
    def end_ip(self):
        """ipaddress.IPv4Address: End of IP range."""
        return self.__end_ip
```

```
    def __eq__(self, param):
        """Compare object with IPv4Range, IPv4Network, IPv4Address.

        Args:
            param (IPv4Range | IPv4Network | IPv4Address):
                object to compare with

        Raises:
            ValueError: if the number or type of parameters is incorrect
        """
        if isinstance(param, IPv4Range):
            if self.start_ip == param.start_ip and self.end_ip == param.end_ip:
                return True
        elif isinstance(param, ipaddress.IPv4Address):
            if self.start_ip == param and self.end_ip == param:
                return True
        elif isinstance(param, ipaddress.IPv4Network):
            if (
                self.start_ip == param.network_address
                and self.end_ip == param.broadcast_address
            ):
                return True
        else:
            raise ValueError(
                "Only IPv4Range, IPv4Address, IPv4Network "
                "types are allowed to compare"
            )
        return False

    def __str__(self):
        """Print form of object."""
        return f"{self.start_ip} - {self.end_ip}"

    def __iter__(self):
        """Unpack start_ip and end_ip from object."""
        yield self.__start_ip
        yield self.__end_ip

    def _get_non_zero_bits_positions(self, param):
        """Return non-zero bits positions in parameter.

        Args:
            param (int): the value in which to find the positions
                of non-zero bits.

        Return:
            (int, int): A tuple, where the first value is the position of
                the high-order non-zero bit, and the second is
                the position of the low-order non-zero bit.

        Examples:
            _get_non_zero_bits_positions(0b110) -> (3,2)
        """
        low_pos = 0
        hi_pos = 0

        index = 1
        while param != 0:
            remainder = param % 2
            if remainder != 0:
                if 0 == low_pos:
                    low_pos = index
                    hi_pos = index
                hi_pos = max (hi_pos, index)
            param //= 2
```

```
            index += 1
        return (hi_pos, low_pos)

    def networks(self):
        """Return a list of IPv4 networks that cover a range of IP addresses.

        The list contains a minimum number of the largest IP networks.

        Return:
            [ipaddress.IPv4Network] - list of IP networks

        """
        start_addr = int(self.__start_ip)
        end_addr = int(self.__end_ip)
        result_list = []

        # this is need becouse network must contain last address
        # of ip range
        end_addr += 1

        # start work
        current_addr = start_addr
        go_next = True
        while go_next:
            low_bit_pos = self._get_non_zero_bits_positions(current_addr)[-1]
            if 0 == low_bit_pos:
                low_bit_pos = IPv4Range.IP_ADDRESS_BITS_COUNT + 1

            # IPv4Range.__IPAdressBitsCount + 1 needed becouse range() must
            # contain IPv4Range.__IPAdressBitsCount as last index
            mask_max = IPv4Range.IP_ADDRESS_BITS_COUNT + 1

            # A smaller mask covers a larger range of IP addresses.
            # The mask cannot be larger than the last non-zero bit in
            # the IP address.
            mask_min = IPv4Range.IP_ADDRESS_BITS_COUNT + 1 - low_bit_pos

            # Looking for an IP mask that most fully covers the specified
            # IP range. Starting with a largest available mask (/0, /1, ... )
            # and ends with a small one (..., /31, /32)
            # If the mask is too big, skip it and go to a smaller one.
            for current_mask in range(mask_min, mask_max):
                mask_add = IPv4Range.IP_ADDRESS_MAX_VALUE >> current_mask
                max_addr = current_addr + mask_add

                if (max_addr + 1) == end_addr:
                    go_next = False
                    break
                if max_addr < end_addr:
                    break

            network_str = (
                str(ipaddress.IPv4Address(current_addr)) + "/"
                + str(current_mask)
            )
            current_addr = max_addr + 1
            result_list.append(ipaddress.IPv4Network(network_str))
        return result_list


class IPv4AddressSpace:
    """List of IPv4Ranges and manipulation methods.

    Attributes:
        __ip_ranges ([IPv4Range]): an internal list of IP address ranges
```

```
        describing the IP address space.
"""

def __init__(self, param=None):
    """Object creation is overloaded with various types of parameters.

    Args:
        param (ipaddress.IPv4Network | IPv4Range | IPv4AddressSpace):
            used to create IP ranges
    Raises:
        ValueError: for unsupported paramter types
    """
    # empty constructor - default
    if param is None:
        self.clear()
    else:
        if isinstance(param, ipaddress.IPv4Network):
            self.__ip_ranges = [IPv4Range(param)]
        elif isinstance(param, IPv4Range):
            self.__ip_ranges = [param]
        elif isinstance(param, IPv4AddressSpace):
            self.__ip_ranges = param.ip_ranges
        else:
            raise ValueError("Unsupported parameter type")

@property
def ip_ranges(self):
    """[IPv4Range]: Address space.

    All IPv4Range's in address space sorted by IP address in ascending
    order. There are no overlaps or duplicates here.
    Use addition or subtraction to manipulate address space.
    """
    return self.__ip_ranges

def _set_ip_ranges(self, param):
    """Private address space setter."""
    if isinstance(param, list):
        # empty list
        if not param:
            self.clear()
        else:
            last_int_ip = 0
            for item in param:
                # type checking
                if not isinstance(item, IPv4Range):
                    raise ValueError("Invalid item in list")
                # IPv4Range's must not overlap and must be
                # sorted in ascending order.
                if (last_int_ip > int(item.start_ip)
                   or last_int_ip > int(item.end_ip)):
                    raise ValueError("IPv4Range not ordered")
                # Add 1 because the next range should not overlap
                # the previous one.
                last_int_ip = int(item.end_ip) + 1
            # check passed. assign list
            self.__ip_ranges = param
    else:
        raise ValueError("Unsupported parameter type")

def clear(self):
    """Clear internal list of IPv4Range's."""
    self.__ip_ranges = []

def is_empty(self):
```

```
        """Check for elements in the IPv4Range list.

        Return:
            (bool): True if the elements exists, or false if it does not.
        """
        if self.__ip_ranges is None:
            return True
        if 0 == len(self.__ip_ranges):
            return True
        return False

    def __iter__(self):
        """Return iterator of internal IPv4Range list."""
        return iter(self.__ip_ranges)

    def __str__(self):
        """Show IPv4Range's in printable form."""
        result = ""
        for ip_range in self.__ip_ranges:
            if result:
                result += "\n"
            result += f"{ip_range}"
        return result

    def _sum(self, param):
        """Private function for summing address spaces.

        The function is overloaded with various parameter types.
        Parameters of any type are converted internally into the
        IPv4AddressSpace. The function returns a list with the minimum number
        of largest IPv4Range's. After the function executes, the number of
        IPv4Range in the internal list may differ from the returned value.

        Args:
            param (ipaddress.IPv4Address | ipaddress.IPv4Network,
                IPv4Range | IPv4AddressSpace): data to add to the internal
                list of IPv4Ranges

        Returns:
            [IPv4Range]: The result of adding IPv4Range's

        Raises:
            ValueError: if an unsupported parameter type is used
        """
        if isinstance(param, ipaddress.IPv4Address):
            added_address_space = IPv4AddressSpace(IPv4Range(param, param))
        elif isinstance(param, ipaddress.IPv4Network):
            added_address_space = IPv4AddressSpace(param)
        elif isinstance(param, IPv4Range):
            added_address_space = IPv4AddressSpace(param)
        elif isinstance(param, IPv4AddressSpace):
            added_address_space = param
        else:
            raise ValueError("Unsupported parameter type")

        combinet_list_of_ranges = (self.ip_ranges
                                   + added_address_space.ip_ranges)
        if not combinet_list_of_ranges:
            return []

        intervals = [
            (int(start_ip), int(end_ip))
            for start_ip, end_ip in combinet_list_of_ranges
        ]
        # Sort by StartIP and ignorig EndIP
```

```
        intervals.sort(key=lambda x: x[0])

        result_ip_ranges = []

        prev_int_start_ip, prev_int_end_ip = intervals[0]

        for current_start, current_end in intervals[1:]:
            # Two ranges intersect or are adjacent,
            # so expand their endpoint.
            if prev_int_end_ip + 1 >= current_start:
                prev_int_end_ip = max(prev_int_end_ip, current_end)
            else:
                result_ip_ranges.append(
                    IPv4Range(
                        ipaddress.IPv4Address(prev_int_start_ip),
                        ipaddress.IPv4Address(prev_int_end_ip),
                    )
                )
                prev_int_start_ip = current_start
                prev_int_end_ip = current_end

        result_ip_ranges.append(
            IPv4Range(
                ipaddress.IPv4Address(prev_int_start_ip),
                ipaddress.IPv4Address(prev_int_end_ip),
            )
        )
        return result_ip_ranges

    def _subtract(self, param):
        """Private function for subtracting address spaces.

        The function is overloaded with various parameter types.
        Parameters of any type are converted internally into the
        IPv4AddressSpace. The function returns a list with the minimum number
        of largest IPv4Range's. After the function executes, the number of
        IPv4Range in the internal list may differ from the returned value.

        Args:
            param (ipaddress.IPv4Address | ipaddress.IPv4Network,
                   IPv4Range | IPv4AddressSpace): data to subtract from the
                   internal list of IPv4Range's

        Returns:
            [IPv4Range]: The result of subtracting address spaces. May be None
                if there are no IP addresses left.

        Raises:
            ValueError: if an unsupported parameter type is used
        """
        if isinstance(param, ipaddress.IPv4Address):
            sub_address_space = IPv4AddressSpace(IPv4Range(param, param))
        elif isinstance(param, ipaddress.IPv4Network):
            sub_address_space = IPv4AddressSpace(param)
        elif isinstance(param, IPv4Range):
            sub_address_space = IPv4AddressSpace(param)
        elif isinstance(param, IPv4AddressSpace):
            sub_address_space = param
        else:
            raise ValueError("Unsupported parameter type")

        if self.is_empty():
            return []

        base = self.__ip_ranges
```

```
        # Convert internal IPRanges to a list of tuples (events) where:
        # - first member is IP converted to int
        # - second member is type of IP:
        # --  1 = start of the base range
        # -- -1 = end of the base range
        # --  2 = start of the subtractive range
        # -- -2 = end of the subtractive range
        events = []

        for start_ip, end_ip in base:
            start = int(start_ip)
            end = int(end_ip)
            events.append((start, 1))
            events.append((end + 1, -1))

        for start_ip, end_ip in sub_address_space:
            start = int(start_ip)
            end = int(end_ip)
            events.append((start, -2))
            events.append((end + 1, 2))

        # Sort events by IP and type. Smallest IP first.
        # If IPs are equal then count type order:
        # substractive end, base start, base end, substractive start
        events.sort(key=lambda x: (x[0], -x[1]))

        # Possible events order after sort:
        # base_start => base_end   | sub_start  | sub_end
        # base_end   => base_start | sub_start  | sub_end
        # sub_start  => sub_end    | base_start | base_end
        # sub_end    => sub_start  | base_start | base_end

        # First event can be: base_start | sub_start

        # Valid subtraction results:
        # 1. base start  | base end
        # 2. base start  | (sub_start - 1)
        # 3. (sub end+1) | base_end
        # We must find these transitions of events

        # When we find the start of a range, the next event will
        # always be the end of the range. Since the range is contiguous
        # and has no gaps

        result = []
        active = 0
        start = None

        for current_int_ip, delta in events:
            if active > 0 and start is not None:
                # save found range
                # -1 becouse all ends incremented by one
                result.append((start, current_int_ip - 1))

            active += delta

            if active > 0 and start is None:
                start = current_int_ip
            elif active <= 0:
                start = None

        # Create IPRanges list
        return [
            IPv4Range(ipaddress.IPv4Address(s), ipaddress.IPv4Address(e))
```

```
            for s, e in result
            if s <= e
        ]

    def __add__(self, param):
        """Addition of address spaces.

        The function is overloaded with various parameter types.
        Parameters of any type are converted internally into the
        IPv4AddressSpace. Returns an IPv4AddressSpace containing the sum
        of the internal list of IPv4Range's with parameters. The result
        consists of the minimum number of the largest IPv4Range's.

        Args:
            param (ipaddress.IPv4Address | ipaddress.IPv4Network,
                IPv4Range | IPv4AddressSpace): data to subtract to the internal
                list of IPv4Range's

        Returns:
            [IPv4Range]: The result of adding IPv4Range's

        Raises:
            ValueError: if an unsupported parameter type is used
        """
        result_ip_address_space = IPv4AddressSpace()
        result_ip_address_space._set_ip_ranges(self._sum(param))
        return result_ip_address_space

    def append(self, param):
        """Add new address space to the internal list of IPv4Range's.

        The function is overloaded with various parameter types.
        Parameters of any type are converted internally into the
        IPv4AddressSpace. After the function is executed, the internal
        IPv4Range list will contain the sum of itself with the parameter.
        The number of elements in the internal IPv4Range list may also change.

        Args:
            param (ipaddress.IPv4Address | ipaddress.IPv4Network,
                IPv4Range | IPv4AddressSpace): data to add to the internal
                list of IPv4Range's

        Raises:
            ValueError: if an unsupported parameter type is used
        """
        self._set_ip_ranges(self._sum(param))

    def __sub__(self, param):
        """Subtraction of address spaces.

        The function is overloaded with various parameter types.
        Parameters of any type are converted internally into the
        IPv4AddressSpace. The function returns a list with the minimum number
        of largest IPv4Range's. After the function executes, the number of
        IPv4Range in the internal list may differ from the returned value.

        Args:
            param (ipaddress.IPv4Address | ipaddress.IPv4Network,
                    IPv4Range | IPv4AddressSpace): data to subtract from the
                    internal list of IPv4Range's

        Returns:
            [IPv4Range]: The result of subtracting address spaces. May be None
                if there are no IP addresses left.
```

```
        Raises:
            ValueError: if an unsupported parameter type is used
        """
        result_ip_address_space = IPv4AddressSpace()
        result_ip_address_space._set_ip_ranges(self._subtract(param))
        return result_ip_address_space

    def remove(self, param):
        """Subtraction of address spaces.

        The function is overloaded with various parameter types.
        Parameters of any type are converted internally into the
        IPv4AddressSpace. After executing the function, the internal IPv4Range
        list will contain the difference between itself and the parameter.
        The number of elements in the internal IPv4Range list may also change
        and become zero.

        Args:
            param (ipaddress.IPv4Address | ipaddress.IPv4Network,
                    IPv4Range | IPv4AddressSpace): data to subtract from the
                    internal list of IPv4Range's

        Raises:
            ValueError: if an unsupported parameter type is used
        """
        self._set_ip_ranges(self._subtract(param))

    def networks(self):
        """Get list of IPv4 networks that span the address space.

        Return:
            [ipaddress.IPv4Network]: The resulting list. May be empty.

        Raises:
            ValueError: if an unsupported parameter type is used
        """
        network_list = []
        for r in self.__ip_ranges:
            network_list.extend(r.networks())
        return network_list

    @staticmethod
    def load_from_file(filename):
        """Return IPv4AddressSpace loaded from file.

        File format:
        # - comment
        Each line can contain:
        IPv4 adress. Ex. 1.2.3.4
        IPv4 subnet. Ex. 1.2.3.4/8 or 1.2.3.4/255.0.0.0
        IPv4 range. Ex. 1.2.3.4-1.2.3.5

        Args:
            param (filename): name of file to load.

        Return:
            IPv4AddressSpace: loaded from file

        Examples:
            ia = IPv4AddressSpace.LoadFromFile("file.txt")

        Raises:
            ValueError: if error parsing file
            FileNotFoundError: if file i/o errors
            OSError: if file i/o errors
```

```
        """
        ip_address_space = IPv4AddressSpace()
        line_from_file = ""

        with open(filename, "r", encoding="utf-8") as file:
            line_succesfuly_decoded = True
            while True:
                if not line_succesfuly_decoded:
                    raise ValueError(f"Error parsing line {line_from_file}")
                # read line from file
                line_from_file = file.readline()
                # stop if line empty
                if not line_from_file:
                    break

                # removing comments
                spleated_list_from_line = line_from_file.split("#")
                uncommented_str = spleated_list_from_line[0]
                uncommented_str = uncommented_str.strip()
                if not uncommented_str: # drop full commented lines
                    continue

                # pasrsing line
                # is line IPv4Address?
                try:
                    ip_addr = ipaddress.IPv4Address(uncommented_str)
                    ip_address_space += ip_addr
                    continue
                except ValueError:
                    pass

                # is line IPv4Network?
                try:
                    ip_network = ipaddress.IPv4Network(uncommented_str,False)
                    ip_address_space += ip_network
                    continue
                except ValueError:
                    pass

                # is line IPv4Range?
                try:
                    split_result = uncommented_str.split("-")
                    if 2 != len(split_result):
                        raise ValueError(
                          "String can't be splited by '-' into 2 parts"
                        )
                    start_ip = ipaddress.IPv4Address(split_result[0].strip())
                    end_ip = ipaddress.IPv4Address(split_result[1].strip())
                    ip_address_space += IPv4Range(start_ip, end_ip)
                    continue
                except ValueError:
                    pass
                line_succesfuly_decoded = False
        return ip_address_space
```

Если библиотеку можно использовать где угодно, хоть на Web-сайте, то мне здесь и сейчас необходима небольшая консольная утилита, позволяющая выполнить необходимые расчёты. В результате я написал

ipcalc.py.

```
import sys
import ipaddressspace

# skip sys.argv[0] — scrpit name.
args = sys.argv[1:]

print("IPv4 address space calculator. (c) imbasoft.ru. Freeware. MIT License")
if 0 == len(args):
    print("Usage: ipcalc.py file1 +/- file2 +/- file3 ... [--network]")
    print("Example: ipcalc.py iaspace1.txt + iaspace2.txt - iaspace3.txt")
    exit(0)

file_expected = True
network_output = False
result_ip_address_space = ipaddressspace.IPv4AddressSpace()
last_operator = "+"

for arg in args:
    if arg in ('+', '-'):
        if file_expected:
            print("Error parsing command line. File expected")
            exit(1)
        file_expected = True
        last_operator = arg
        continue
    if "--network" == arg:
      network_output = True
      continue
    else:
        if not file_expected:
            print("Error parsing command line. Operator expected")
            exit(1)

        ia_from_file = None
        try:
            ia_from_file=ipaddressspace.IPv4AddressSpace.load_from_file(arg)
        except:
            print(f"Error parsing file: {arg}")
            exit(1)
        if "+" == last_operator:
            result_ip_address_space += ia_from_file
        if "-" == last_operator:
            result_ip_address_space -= ia_from_file
        file_expected = False

print("Result:")
if (network_output):
    for net in result_ip_address_space.networks():
        print(net)
else:
    print(result_ip_address_space)
```

Работает она просто. В командной строке указываются файлы и арифметические операторы. Файлы содержат адресные пространства, а арифметические операторы указывают, что необходимо сделать: сложить адресные пространства между собой или вычесть. По умолчанию ответ выдаётся в виде диапазонов IP-адресов (например, 1.2.3.4-1.2.3.10), а если указать ключ «--network», то в виде CIDR-подсетей (например, 10.20.30.0/24).

Для решения моей задачи я сделал два файла:
1.FullSpace.txt, задающий всё доступное адресное пространство.

```
0.0.0.0/0
```

2.SpecialIPs.txt, содержащий перечень зарезервированных IP-адресов:

```
# Based on https://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml
0.0.0.0/8
0.0.0.0/32
10.0.0.0/8
100.64.0.0/10
127.0.0.0/8
169.254.0.0/16
172.16.0.0/12
192.0.0.0/24
192.0.0.0/29
192.0.0.8/32
# The IP address is considered globally unreachable.
192.0.0.9/32
# The IP address is considered globally unreachable.
192.0.0.10/32
192.0.0.170/32
192.0.0.171/32
192.0.2.0/24
192.88.99.2/32
192.168.0.0/16
198.18.0.0/15
198.51.100.0/24
203.0.113.0/24
240.0.0.0/4
255.255.255.255/32

# Manually added multicast subnet
224.0.0.4/4
```

Запустив всё это вместе:

```
python3 ipcalc.py FullSpace.txt - SpecialIPs.txt
```

Я получил перечень белых IP-диапазонов:

```
1.0.0.0 - 9.255.255.255
11.0.0.0 - 100.63.255.255
100.128.0.0 - 126.255.255.255
128.0.0.0 - 169.253.255.255
169.255.0.0 - 172.15.255.255
172.32.0.0 - 191.255.255.255
192.0.1.0 - 192.0.1.255
192.0.3.0 - 192.88.99.1
192.88.99.3 - 192.167.255.255
192.169.0.0 - 198.17.255.255
198.20.0.0 - 198.51.99.255
198.51.101.0 - 203.0.112.255
203.0.114.0 - 223.255.255.255
```

Добавив ключ «--network»

```
python3 ipcalc.py FullSpace.txt - SpecialIPs.txt –network
```

Получил перечень «белых IP-подсетей»:

```
1.0.0.0/8
2.0.0.0/7
4.0.0.0/6
8.0.0.0/7
11.0.0.0/8
12.0.0.0/6
16.0.0.0/4
32.0.0.0/3
64.0.0.0/3
96.0.0.0/6
100.0.0.0/10
100.128.0.0/9
101.0.0.0/8
102.0.0.0/7
104.0.0.0/5
112.0.0.0/5
120.0.0.0/6
124.0.0.0/7
126.0.0.0/8
128.0.0.0/3
160.0.0.0/5
168.0.0.0/8
169.0.0.0/9
169.128.0.0/10
169.192.0.0/11
```

169.224.0.0/12
169.240.0.0/13
169.248.0.0/14
169.252.0.0/15
169.255.0.0/16
170.0.0.0/7
172.0.0.0/12
172.32.0.0/11
172.64.0.0/10
172.128.0.0/9
173.0.0.0/8
174.0.0.0/7
176.0.0.0/4
192.0.1.0/24
192.0.3.0/24
192.0.4.0/22
192.0.8.0/21
192.0.16.0/20
192.0.32.0/19
192.0.64.0/18
192.0.128.0/17
192.1.0.0/16
192.2.0.0/15
192.4.0.0/14
192.8.0.0/13
192.16.0.0/12
192.32.0.0/11
192.64.0.0/12
192.80.0.0/13
192.88.0.0/18
192.88.64.0/19
192.88.96.0/23
192.88.98.0/24
192.88.99.0/31
192.88.99.3/32
192.88.99.4/30
192.88.99.8/29
192.88.99.16/28
192.88.99.32/27
192.88.99.64/26
192.88.99.128/25
192.88.100.0/22
192.88.104.0/21
192.88.112.0/20

192.88.128.0/17
192.89.0.0/16
192.90.0.0/15
192.92.0.0/14
192.96.0.0/11
192.128.0.0/11
192.160.0.0/13
192.169.0.0/16
192.170.0.0/15
192.172.0.0/14
192.176.0.0/12
192.192.0.0/10
193.0.0.0/8
194.0.0.0/7
196.0.0.0/7
198.0.0.0/12
198.16.0.0/15
198.20.0.0/14
198.24.0.0/13
198.32.0.0/12
198.48.0.0/15
198.50.0.0/16
198.51.0.0/18
198.51.64.0/19
198.51.96.0/22
198.51.101.0/24
198.51.102.0/23
198.51.104.0/21
198.51.112.0/20
198.51.128.0/17
198.52.0.0/14
198.56.0.0/13
198.64.0.0/10
198.128.0.0/9
199.0.0.0/8
200.0.0.0/7
202.0.0.0/8
203.0.0.0/18
203.0.64.0/19
203.0.96.0/20
203.0.112.0/24
203.0.114.0/23
203.0.116.0/22
203.0.120.0/21

203.0.128.0/17 203.1.0.0/16 203.2.0.0/15 203.4.0.0/14 203.8.0.0/13 203.16.0.0/12 203.32.0.0/11 203.64.0.0/10 203.128.0.0/9 204.0.0.0/6 208.0.0.0/4