ELOQUENT PYTHON PROGRAMMING

STRUKTUR DATA

Struktur data bawaan Python beragam, seperti fixed array, array, doubly linked list, hashmap, dengan interface yang beragam pula yaitu:

1. tuple, daftar beragam objek dengan urutan indeks, anggotanya tidak dapat diedit (immutable)
2. list, daftar beragam objek dengan urutan indeks, anggota dapat diubah, ditambah, dikurangi
3. dict, kumpulan pasangan kunci&nilai, kunci tidak dapat diedit, nilai dapat diedit, anggota dapat dibuang, ditambah
4. set, kumpulan objek tanpa duplikat, tanpa urutan
5. str, deretan karakter, dengan urutan indeks, tidak dapat diedit

lalu ada beberapa tambahan dari pustaka sertaan

1. array
2. double end queue deque
3. namedtuple

In [1]:
```
from array import array
from collections import deque, namedtuple

for cls in (tuple, list, dict, set, str, array, deque):
    kandungan = [k for k in dir(cls) if '_' not in k]  # list comprehension
    print(cls,':')
    print(kandungan)
    print('~'*20)
print()
```

```
<class 'tuple'> :
['count', 'index']
~~~~~~~~~~~~~~~~~~~~
<class 'list'> :
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reve
rse', 'sort']
~~~~~~~~~~~~~~~~~~~~
<class 'dict'> :
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'u
pdate', 'values']
~~~~~~~~~~~~~~~~~~~~
<class 'set'> :
['add', 'clear', 'copy', 'difference', 'discard', 'intersection', 'isdisjoint', 'issubse
t', 'issuperset', 'pop', 'remove', 'union', 'update']
~~~~~~~~~~~~~~~~~~~~
<class 'str'> :
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'fin
d', 'format', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isident
ifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'r
eplace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitli
nes', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
~~~~~~~~~~~~~~~~~~~~
```

```
<class 'array.array'> :
['append', 'byteswap', 'count', 'extend', 'frombytes', 'fromfile', 'fromlist', 'fromunic
ode', 'index', 'insert', 'itemsize', 'pop', 'remove', 'reverse', 'tobytes', 'tofile', 't
olist', 'tounicode', 'typecode']
~~~~~~~~~~~~~~~~~~~~
<class 'collections.deque'> :
['append', 'appendleft', 'clear', 'copy', 'count', 'extend', 'extendleft', 'index', 'ins
ert', 'maxlen', 'pop', 'popleft', 'remove', 'reverse', 'rotate']
~~~~~~~~~~~~~~~~~~~~
```

Source code dari class namedtuple yang dapat dilihat dengan fungsi inspect.getsource(), sbb:

In [ ]:
```
from inspect import getsource
print(getsource(namedtuple))
```

TUPLE

Tuple sudah banyak sekali dijelaskan dari sudut interface atau spesifikasinya, tapi jarang dibahas dari sisi implementasi struktur datanya. Materi mengenai struktur data dapat dirujuk dari kursus terbuka "MIT OCW 6.006 Introduction to Algorithms".
Struktur tuple tidak mendukung operasi append, atau insert sehingga mengesankan bahwa ia adalah objek container fixed array berisi C structure pointers yang masing masing menunjuk ke objek objek anggota tuple. Dengan elemen pointers yang seragam, sama ukurannya, misalnya dalam sistem 64 bit digunakan 8 bytes, maka pada struktur fixed array kita dapat mengakses elemen anggota secara langsung tidak tergantung dari banyaknya anggota, seperti mengakses random access memory, alamat elemen pada indeks k, adalah alamat indeks awal ditambah size pointer dikalikan indeks k, atau pada notasi big-O adalah O(1). Fixed array memiliki kelemahan pada operasi insert yang memerlukan pergeseran elemen.
Lain lagi dengan struktur link list, akses anggota elemen menjadi lebih lambat, linear O(n), karena harus ditelusuri linknya satu per satu, walaupun untuk operasi append/insert ringan, O(1).
Karakteristik Tuple lebih condong ke arah fixed array, artinya untuk operasi dengan data immutable kita mendapat *akses yang cepat, ringan, hemat memori* dengan menggunakan tuple.

Tuple adalah immutable dalam artian anggotanya tidak dapat diedit, ditambah, tetapi tuple dapat dibuat lagi dengan anggota yang baru.

In [2]:
```
t1 = ((2021, 'lulus sekolah'),)
t1 = t1 + ((2022, 'mulai kerja'),)
print(t1)
```

```
((2021, 'lulus sekolah'), (2022, 'mulai kerja'))
```

Pada contoh di atas tuple t1 bukan di edit tetapi nama t1 dipakai lagi, dikaitkan dengan objek yang baru yaitu gabungan objek t1 yang sebelumnya dengan suatu ekspresi baru. Kita dapat memastikan dengan melihat id objeknya, yaitu alamat objek yang berbeda. (implementasi CPython id = alamat)

In [3]:
```
t1 = ((2021, 'lulus sekolah'),)
old_id = id(t1)
t1 = t1 + ((2022, 'mulai kerja'),)
new_id = id(t1)
print(old_id, new_id)
```

```
1479522468672 1479521798720
```

In [4]:
```
t1 = ((2021, 'lulus sekolah'),)
obj0 = id(t1[0])
t1 = t1 + ((2022, 'mulai kerja'),)
obj1 = id(t1[0])
print(obj0, obj1)
```

```
1479521717120 1479521717120
```

Kita perlu ingat bahwa operasi assignment dalam Python hanya melibatkan *pointer* sedangkan objek aslinya sendiri tidak berpindah, tidak dicopy. Pada contoh di atas objek elemen pertama tetap digunakan pada tuple yang baru, dilihat dari alamat objeknya yang sama.

In [5]:
```
t2 = t1[:] #tuple aliasing
print(t1 is t2)
```

```
True
```

In [6]:
```
Lst1 = list([1,2,3,4])
Lst2 = Lst1[:]  # list cloning
print(Lst1 is Lst2)
```

```
False
```

Perlu diperhatikan juga bahwa **tuple tidak mendukung cloning**, karena sifatnya yang *immutable* maka tidak ada alasan untuk mendukung cloning, membuat objek baru dengan isi yang sama. Jadi walaupun kita menggunakan sintaks yang biasa untuk membuat cloning pada list, efeknya pada tuple adalah aliasing. (cloning adalah membuat objek berbeda berisikan elemen yang sama, sedangkan aliasing adalah memberi nama yang berbeda untuk objek yang sama)

In [2]:
```
'''
tuple, kumpulan objek yang dapat diiterasi
generator expressions dengan sintaks mirip list comprehension menghasilkan objek genera
unpacking operator asterisk * memiliki efek mengiterasi generator
'''
for klas in (tuple, list,):   # for'each'-in loop dari iterable tuple
    metodas = (m for m in dir(klas) if '_' not in m)   # membentuk suatu generator
    print(metodas)
    print(*metodas,'\n')   # asterisk * unpacking generator, mengeluarkan elemen-2 nya.
```

```
<generator object <genexpr> at 0x0000018727761A80>
count index

<generator object <genexpr> at 0x00000187277623B0>
append clear copy count extend index insert pop remove reverse sort
```

Iterable ialah suatu objek yang mampu mengembalikan elemen anggotanya satu per satu. Contoh iterable antara lain semua tipe sekuens (seperti list, tuple, str), dan beberapa tipe non-sekuens seperti dict, objek file, dan juga objek dari klas yang memiliki metoda __iter__() atau __getitem__(), yang mengimplementasikan semantik sekuens.

Iterable dapat digunakan bersama for loop atau bersama fungsi lain yang memerlukan sekuens seperti map(), zip().

Generator ialah suatu fungsi yang mengeluarkan elemen di dalamnya satu per satu, dan cenderung lebih hemat memori daripada list yang dibentang isi elemennya. Generator dapat digunakan bersama for loop, atau fungsi next() yang hanya mengeluarkan satu elemen setiap kali dipanggil. (ekspresi bintang, asterisk * juga dapat digunakan dalam kasus tertentu)

DEQUE

Menurut Raymond Hettinger, proses insert elemen di posisi awal pada suatu list menggunakan waktu yang tidak sedikit, O(n), sehingga jika suatu program besar banyak melakukan hal itu maka kinerja keseluruhan akan menurun. Itulah sebabnya dia menambahkan suatu struktur data doubly-linked list, double ended queue atau disingkat deque dengan metoda andalannya *appendleft, extendleft, popleft*, dalam pustaka collections.
Operasi append, pop pada posisi awal/akhir deque memiliki kompleksitas waktu konstan O(1). Tetapi deque kedodoran dalam akses elemen di tengah, O(n), suatu kelemahan bawaan dari struktur doubly-linked list.

In [2]:
```
from time import time
from collections import deque

deck = deque()
start = time()
for n in range(1_000_000):
    deck.appendleft(n)
stop = time()
print(f'deque appendleft {1+deck[0]:,} elemen {stop-start:.2f}')

Lst = list()
nmax = 250_000
start = time()
for n in range(nmax):
    Lst.insert(0,n)
stop = time()
print(f'list insert0 {1+Lst[0]:,} elemen {stop-start:.2f}')
```

```
deque appendleft 1,000,000 elemen 0.12
list insert0 250,000 elemen 14.30
```

DICT

Struktur dari dict adalah hashmap atau sebutan lainnya hashtable. Dict adalah kumpulan objek dengan tatanan berupa pasangan kunci dan nilai.
Kunci pada dictionary haruslah objek yang hashable karena strukturnya tidak memiliki indeks tetapi menggunakan fungsi hash pada key/kunci sebagai 'indeks' elemennya. Objek yang hashable antara lain integer, float, string, tuple yang beranggotakan hashable object. List termasuk objek yang unhashable, tidak dapat digunakan sebagai kunci.
Dalam hashmap/dict kompleksitas pencarian elemen adalah O(1) dan insertion/deletion juga O(1), artinya konstan tidak tergantung pada banyaknya elemen anggota yang terdaftar.
Jika terjadi *collision* atau fungsi hash atas dua atau beberapa key menghasilkan 'indeks hash' yang

sama maka perlu suatu penanganan lagi, misalnya perbaikan fungsi hash, linear probing, quadratic probing, double hash layering, atau menggunakan chaining link list, dan ini akan mempengaruhi performa kompleksitas waktunya.

In [1]:
```
nomor_telp = dict()
nomor_telp[('sam','edw')] = '0888999733x'
nomor_telp[('budi','di amrik')] = '(818) 785 090x'
print(nomor_telp[('sam','edw')])
print(nomor_telp)
```

```
0888999733x
{('sam', 'edw'): '0888999733x', ('budi', 'di amrik'): '(818) 785 090x'}
```

Salah satu penggunaan dict ialah untuk menghitung jumlah pemunculan huruf huruf. Berikut contoh kode tentang anagram, permainan kata, yaitu huruf huruf dalam suatu kata kata diacak menjadi kata kata yang berbeda, misalnya beras|sebar|besar, "eleven plus two" | "twelve plus one", "drum piano asbak"|"opa bandar musik". Kode kita hanya akan menghitung huruf dan menyimpulkan apakah dua ungkapan kata merupakan anagram atau tidak.

In [2]:
```
def anagram(word,words):
    def cc(word):
        d={}
        for c in word.lower().replace(' ',''):
            if c not in d:
                d[c] = 0
            d[c] = d[c] + 1
        return d
    ref = cc(word)
    return [wrd for wrd in words if cc(wrd) == ref]

print(anagram("setec astronomy", ("too many secrets", "my Socrates note")))
print(anagram('beras',('besar','sebar', 'sabar','resah')))
```

```
['too many secrets', 'my Socrates note']
['besar', 'sebar']
```

Baris ke-7 pada contoh di atas menghitung kemunculan setiap huruf dalam dictionary dengan kunci berupa hurufnya dan nilainya diakumulasi. Baris ke-5,6,7 dapat diganti menjadi satu baris menggunakan metoda dict.get() sbb:

In [3]:
```
def anagram(word,words):
    def cc(word):
        d={}
        for c in word.lower().replace(' ',''):
            d[c] = d.get(c,0) + 1
        return d
    ref = cc(word)
    return [wrd for wrd in words if cc(wrd) == ref]

print(anagram("setec astronomy", ("too many secrets", "my Socrates note")))
print(anagram('beras',('besar','sebar', 'sabar','resah')))
```

```
['too many secrets', 'my Socrates note']
['besar', 'sebar']
```

Proses counting dengan dictionary sepertinya banyak dipakai sehingga GvR membuat class khusus bernama Counter dan disematkan dalam pustaka collections. Akhirnya, kode kita dapat dipadatkan menjadi:

In [4]:
```
from collections import Counter
def anagram(word,words):
    cc = lambda w: Counter(w.lower().replace(' ',''))
    ref = cc(word)
    return [wrd for wrd in words if cc(wrd) == ref]

print(anagram("setec astronomy", ("too many secrets", "my Socrates note")))
print(anagram('beras',('besar','sebar', 'sabar','resah')))
```

```
['too many secrets', 'my Socrates note']
['besar', 'sebar']
```

Kita dapat melihat gaya penulisan fungsional ala Python seperti contoh berikut yang melakukan pengecekan apakah dua ungkapan merupakan anagram atau bukan.

In [5]:
```
from collections import Counter
cc = lambda w: Counter(w.lower().replace(' ',''))
is_anagram = lambda w1,w2: cc(w1) == cc(w2)

print(is_anagram('eleven plus two','twelve plus one'))
```

```
True
```

In [6]:
```
import inspect
print(inspect.getsource(Counter))
```

```
class Counter(dict):
    '''Dict subclass for counting hashable items.  Sometimes called a bag
    or multiset.  Elements are stored as dictionary keys and their counts
    are stored as dictionary values.

    >>> c = Counter('abcdeabcdabcaba')  # count elements from a string

    >>> c.most_common(3)                # three most common elements
    [('a', 5), ('b', 4), ('c', 3)]
    >>> sorted(c)                       # list all unique elements
    ['a', 'b', 'c', 'd', 'e']
    >>> ''.join(sorted(c.elements()))   # list elements with repetitions
    'aaaaabbbbcccdde'
    >>> sum(c.values())                 # total of all counts
    15

    >>> c['a']                          # count of letter 'a'
    5
    >>> for elem in 'shazam':           # update counts from an iterable
    ...     c[elem] += 1                # by adding 1 to each element's count
    >>> c['a']                          # now there are seven 'a'
    7
    >>> del c['b']                      # remove all 'b'
    >>> c['b']                          # now there are zero 'b'
    0
```

```
        >>> d = Counter('simsalabim')       # make another counter
        >>> c.update(d)                     # add in the second counter
        >>> c['a']                          # now there are nine 'a'
        9

        >>> c.clear()                       # empty the counter
        >>> c
        Counter()

        Note:  If a count is set to zero or reduced to zero, it will remain
        in the counter until the entry is deleted or the counter is cleared:

        >>> c = Counter('aaabbc')
        >>> c['b'] -= 2                     # reduce the count of 'b' by two
        >>> c.most_common()                 # 'b' is still in, but its count is zero
        [('a', 3), ('c', 1), ('b', 0)]

        '''
        # References:
        #   http://en.wikipedia.org/wiki/Multiset
        #   http://www.gnu.org/software/smalltalk/manual-base/html_node/Bag.html
        #   http://www.demo2s.com/Tutorial/Cpp/0380__set-multiset/Catalog0380__set-multiset.
    htm
        #   http://code.activestate.com/recipes/259174/
        #   Knuth, TAOCP Vol. II section 4.6.3

        def __init__(self, iterable=None, /, **kwds):
            '''Create a new, empty Counter object.  And if given, count elements
            from an input iterable.  Or, initialize the count from another mapping
            of elements to their counts.

            >>> c = Counter()                           # a new, empty counter
            >>> c = Counter('gallahad')                 # a new counter from an iterable
            >>> c = Counter({'a': 4, 'b': 2})           # a new counter from a mapping
            >>> c = Counter(a=4, b=2)                   # a new counter from keyword args

            '''
            super().__init__()
            self.update(iterable, **kwds)

        def __missing__(self, key):
            'The count of elements not in the Counter is zero.'
            # Needed so that self[missing_item] does not raise KeyError
            return 0

        def total(self):
            'Sum of the counts'
            return sum(self.values())

        def most_common(self, n=None):
            '''List the n most common elements and their counts from the most
            common to the least.  If n is None, then list all element counts.

            >>> Counter('abracadabra').most_common(3)
            [('a', 5), ('b', 2), ('r', 2)]

            '''
            # Emulate Bag.sortedByCount from Smalltalk
            if n is None:
                return sorted(self.items(), key=_itemgetter(1), reverse=True)
```

```
        # Lazy import to speedup Python startup time
        import heapq
        return heapq.nlargest(n, self.items(), key=_itemgetter(1))

    def elements(self):
        '''Iterator over elements repeating each as many times as its count.

        >>> c = Counter('ABCABC')
        >>> sorted(c.elements())
        ['A', 'A', 'B', 'B', 'C', 'C']

        # Knuth's example for prime factors of 1836:  2**2 * 3**3 * 17**1
        >>> prime_factors = Counter({2: 2, 3: 3, 17: 1})
        >>> product = 1
        >>> for factor in prime_factors.elements():     # loop over factors
        ...     product *= factor                       # and multiply them
        >>> product
        1836

        Note, if an element's count has been set to zero or is a negative
        number, elements() will ignore it.

        '''
        # Emulate Bag.do from Smalltalk and Multiset.begin from C++.
        return _chain.from_iterable(_starmap(_repeat, self.items()))

    # Override dict methods where necessary

    @classmethod
    def fromkeys(cls, iterable, v=None):
        # There is no equivalent method for counters because the semantics
        # would be ambiguous in cases such as Counter.fromkeys('aaabbc', v=2).
        # Initializing counters to zero values isn't necessary because zero
        # is already the default value for counter lookups.  Initializing
        # to one is easily accomplished with Counter(set(iterable)).  For
        # more exotic cases, create a dictionary first using a dictionary
        # comprehension or dict.fromkeys().
        raise NotImplementedError(
            'Counter.fromkeys() is undefined.  Use Counter(iterable) instead.')

    def update(self, iterable=None, /, **kwds):
        '''Like dict.update() but add counts instead of replacing them.

        Source can be an iterable, a dictionary, or another Counter instance.

        >>> c = Counter('which')
        >>> c.update('witch')           # add elements from another iterable
        >>> d = Counter('watch')
        >>> c.update(d)                 # add elements from another counter
        >>> c['h']                      # four 'h' in which, witch, and watch
        4

        '''
        # The regular dict.update() operation makes no sense here because the
        # replace behavior results in the some of original untouched counts
        # being mixed-in with all of the other counts for a mismash that
        # doesn't have a straight-forward interpretation in most counting
        # contexts.  Instead, we implement straight-addition.  Both the inputs
        # and outputs are allowed to contain zero and negative counts.
```

```
            if iterable is not None:
                if isinstance(iterable, _collections_abc.Mapping):
                    if self:
                        self_get = self.get
                        for elem, count in iterable.items():
                            self[elem] = count + self_get(elem, 0)
                    else:
                        # fast path when counter is empty
                        super().update(iterable)
                else:
                    _count_elements(self, iterable)
            if kwds:
                self.update(kwds)

        def subtract(self, iterable=None, /, **kwds):
            '''Like dict.update() but subtracts counts instead of replacing them.
            Counts can be reduced below zero.  Both the inputs and outputs are
            allowed to contain zero and negative counts.

            Source can be an iterable, a dictionary, or another Counter instance.

            >>> c = Counter('which')
            >>> c.subtract('witch')             # subtract elements from another iterable
            >>> c.subtract(Counter('watch'))    # subtract elements from another counter
            >>> c['h']                          # 2 in which, minus 1 in witch, minus 1 in w
        atch
            0
            >>> c['w']                          # 1 in which, minus 1 in witch, minus 1 in w
        atch
            -1

            '''
            if iterable is not None:
                self_get = self.get
                if isinstance(iterable, _collections_abc.Mapping):
                    for elem, count in iterable.items():
                        self[elem] = self_get(elem, 0) - count
                else:
                    for elem in iterable:
                        self[elem] = self_get(elem, 0) - 1
            if kwds:
                self.subtract(kwds)

        def copy(self):
            'Return a shallow copy.'
            return self.__class__(self)

        def __reduce__(self):
            return self.__class__, (dict(self),)

        def __delitem__(self, elem):
            'Like dict.__delitem__() but does not raise KeyError for missing values.'
            if elem in self:
                super().__delitem__(elem)

        def __eq__(self, other):
            'True if all counts agree. Missing counts are treated as zero.'
            if not isinstance(other, Counter):
                return NotImplemented
```

```
        return all(self[e] == other[e] for c in (self, other) for e in c)

    def __ne__(self, other):
        'True if any counts disagree. Missing counts are treated as zero.'
        if not isinstance(other, Counter):
            return NotImplemented
        return not self == other

    def __le__(self, other):
        'True if all counts in self are a subset of those in other.'
        if not isinstance(other, Counter):
            return NotImplemented
        return all(self[e] <= other[e] for c in (self, other) for e in c)

    def __lt__(self, other):
        'True if all counts in self are a proper subset of those in other.'
        if not isinstance(other, Counter):
            return NotImplemented
        return self <= other and self != other

    def __ge__(self, other):
        'True if all counts in self are a superset of those in other.'
        if not isinstance(other, Counter):
            return NotImplemented
        return all(self[e] >= other[e] for c in (self, other) for e in c)

    def __gt__(self, other):
        'True if all counts in self are a proper superset of those in other.'
        if not isinstance(other, Counter):
            return NotImplemented
        return self >= other and self != other

    def __repr__(self):
        if not self:
            return f'{self.__class__.__name__}()'
        try:
            # dict() preserves the ordering returned by most_common()
            d = dict(self.most_common())
        except TypeError:
            # handle case where values are not orderable
            d = dict(self)
        return f'{self.__class__.__name__}({d!r})'

    # Multiset-style mathematical operations discussed in:
    #       Knuth TAOCP Volume II section 4.6.3 exercise 19
    #       and at http://en.wikipedia.org/wiki/Multiset
    #
    # Outputs guaranteed to only include positive counts.
    #
    # To strip negative and zero counts, add-in an empty counter:
    #       c += Counter()
    #
    # When the multiplicities are all zero or one, multiset operations
    # are guaranteed to be equivalent to the corresponding operations
    # for regular sets.
    #     Given counter multisets such as:
    #         cp = Counter(a=1, b=0, c=1)
    #         cq = Counter(c=1, d=0, e=1)
    #     The corresponding regular sets would be:
    #         sp = {'a', 'c'}
```

```
#         sq = {'c', 'e'}
#     All of the following relations would hold:
#         set(cp + cq) == sp | sq
#         set(cp - cq) == sp - sq
#         set(cp | cq) == sp | sq
#         set(cp & cq) == sp & sq
#         (cp == cq) == (sp == sq)
#         (cp != cq) == (sp != sq)
#         (cp <= cq) == (sp <= sq)
#         (cp < cq) == (sp < sq)
#         (cp >= cq) == (sp >= sq)
#         (cp > cq) == (sp > sq)

def __add__(self, other):
    '''Add counts from two counters.

    >>> Counter('abbb') + Counter('bcc')
    Counter({'b': 4, 'c': 2, 'a': 1})

    '''
    if not isinstance(other, Counter):
        return NotImplemented
    result = Counter()
    for elem, count in self.items():
        newcount = count + other[elem]
        if newcount > 0:
            result[elem] = newcount
    for elem, count in other.items():
        if elem not in self and count > 0:
            result[elem] = count
    return result

def __sub__(self, other):
    ''' Subtract count, but keep only results with positive counts.

    >>> Counter('abbbc') - Counter('bccd')
    Counter({'b': 2, 'a': 1})

    '''
    if not isinstance(other, Counter):
        return NotImplemented
    result = Counter()
    for elem, count in self.items():
        newcount = count - other[elem]
        if newcount > 0:
            result[elem] = newcount
    for elem, count in other.items():
        if elem not in self and count < 0:
            result[elem] = 0 - count
    return result

def __or__(self, other):
    '''Union is the maximum of value in either of the input counters.

    >>> Counter('abbb') | Counter('bcc')
    Counter({'b': 3, 'c': 2, 'a': 1})

    '''
    if not isinstance(other, Counter):
        return NotImplemented
```

```
            result = Counter()
            for elem, count in self.items():
                other_count = other[elem]
                newcount = other_count if count < other_count else count
                if newcount > 0:
                    result[elem] = newcount
            for elem, count in other.items():
                if elem not in self and count > 0:
                    result[elem] = count
            return result

        def __and__(self, other):
            ''' Intersection is the minimum of corresponding counts.

            >>> Counter('abbb') & Counter('bcc')
            Counter({'b': 1})

            '''
            if not isinstance(other, Counter):
                return NotImplemented
            result = Counter()
            for elem, count in self.items():
                other_count = other[elem]
                newcount = count if count < other_count else other_count
                if newcount > 0:
                    result[elem] = newcount
            return result

        def __pos__(self):
            'Adds an empty counter, effectively stripping negative and zero counts'
            result = Counter()
            for elem, count in self.items():
                if count > 0:
                    result[elem] = count
            return result

        def __neg__(self):
            '''Subtracts from an empty counter.  Strips positive and zero counts,
            and flips the sign on negative counts.

            '''
            result = Counter()
            for elem, count in self.items():
                if count < 0:
                    result[elem] = 0 - count
            return result

        def _keep_positive(self):
            '''Internal method to strip elements with a negative or zero count'''
            nonpositive = [elem for elem, count in self.items() if not count > 0]
            for elem in nonpositive:
                del self[elem]
            return self

        def __iadd__(self, other):
            '''Inplace add from another counter, keeping only positive counts.

            >>> c = Counter('abbb')
            >>> c += Counter('bcc')
            >>> c
```

```
        Counter({'b': 4, 'c': 2, 'a': 1})

        '''
        for elem, count in other.items():
            self[elem] += count
        return self._keep_positive()

    def __isub__(self, other):
        '''Inplace subtract counter, but keep only results with positive counts.

        >>> c = Counter('abbbc')
        >>> c -= Counter('bccd')
        >>> c
        Counter({'b': 2, 'a': 1})

        '''
        for elem, count in other.items():
            self[elem] -= count
        return self._keep_positive()

    def __ior__(self, other):
        '''Inplace union is the maximum of value from either counter.

        >>> c = Counter('abbb')
        >>> c |= Counter('bcc')
        >>> c
        Counter({'b': 3, 'c': 2, 'a': 1})

        '''
        for elem, other_count in other.items():
            count = self[elem]
            if other_count > count:
                self[elem] = other_count
        return self._keep_positive()

    def __iand__(self, other):
        '''Inplace intersection is the minimum of corresponding counts.

        >>> c = Counter('abbb')
        >>> c &= Counter('bcc')
        >>> c
        Counter({'b': 1})

        '''
        for elem, count in self.items():
            other_count = other[elem]
            if other_count < count:
                self[elem] = other_count
        return self._keep_positive()
```