

Ch09-2-Built-in-DataStructures

December 24, 2021

1 Built-in Data Structures and Collections

- all builtin functions are listed here with examples: <https://docs.python.org/3/library/functions.html>

1.1 zip()

- built-in zip class can help us quickly create list of tuples and then a dictionary

```
[1]: help(zip)
```

Help on class zip in module builtins:

```
class zip(object)
| zip(*iterables) --> A zip object yielding tuples until an input is
| exhausted.
|
|     >>> list(zip('abcdefg', range(3), range(4)))
|     [('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]
|
| The zip object yields n-length tuples, where n is the number of iterables
| passed as positional arguments to zip(). The i-th element in every tuple
| comes from the i-th iterable argument to zip(). This continues until the
| shortest argument is exhausted.
|
| Methods defined here:
|
| __getattr__(self, name, /)
|     Return getattr(self, name).
|
| __iter__(self, /)
|     Implement iter(self).
|
| __next__(self, /)
|     Implement next(self).
|
| __reduce__(...)
|     Return state information for pickling.
|
| -----
```

```
| Static methods defined here:
|
| __new__(*args, **kwargs) from builtins.type
|     Create and return a new object.  See help(type) for accurate signature.
```

```
[2]: zdata = zip([1, 2, 3], ('a', 'b', 'c'))
```

```
[3]: alist = list(zdata)
```

```
[4]: alist
```

```
[4]: [(1, 'a'), (2, 'b'), (3, 'c')]
```

```
[5]: # create dict
adict = dict(alist)
print(adict)
```

```
{1: 'a', 2: 'b', 3: 'c'}
```

1.2 exercise

Create a dict that maps lowercase alphabets to integers, e.g., a maps to 1, b maps to 2, ..., z maps to 26 and print it

```
[6]: import string
lettersToDigits = dict(zip(string.ascii_lowercase, range(1, 27)))
```

```
[7]: print(lettersToDigits)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8, 'i': 9, 'j':
10, 'k': 11, 'l': 12, 'm': 13, 'n': 14, 'o': 15, 'p': 16, 'q': 17, 'r': 18, 's':
19, 't': 20, 'u': 21, 'v': 22, 'w': 23, 'x': 24, 'y': 25, 'z': 26}
```

1.3 exercise

Create a dict that maps lowercase alphabets to their corresponding ASCII values , e.g., a maps to 97, b maps to 98, ..., z maps to 122 and print the dictionary in alphabetical order

```
[1]: import string
lettersToDigits = dict(zip(string.ascii_lowercase, range(ord('a'), ord('z')+1)))
```

```
[2]: print(lettersToDigits)
```

```
{'a': 97, 'b': 98, 'c': 99, 'd': 100, 'e': 101, 'f': 102, 'g': 103, 'h': 104,
'i': 105, 'j': 106, 'k': 107, 'l': 108, 'm': 109, 'n': 110, 'o': 111, 'p': 112,
'q': 113, 'r': 114, 's': 115, 't': 116, 'u': 117, 'v': 118, 'w': 119, 'x': 120,
'y': 121, 'z': 122}
```

1.4 Set Types - set, frozenset

- <https://docs.python.org/3/library/stdtypes.html#set>
- as set object is an unordered collection of distinct hashable objects
- set is mutable
- frozenset is immutable

```
[8]: help(set)
```

Help on class set in module builtins:

```
class set(object)
|   set() -> new empty set object
|   set(iterable) -> new set object
|
|   Build an unordered collection of unique elements.
|
|   Methods defined here:
|
|   __and__(self, value, /)
|       Return self&value.
|
|   __contains__(...)
|       x.__contains__(y) <==> y in x.
|
|   __eq__(self, value, /)
|       Return self==value.
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|   __getattr__(self, name, /)
|       Return getattr(self, name).
|
|   __gt__(self, value, /)
|       Return self>value.
|
|   __iand__(self, value, /)
|       Return self&=value.
|
|   __init__(self, /, *args, **kwargs)
|       Initialize self.  See help(type(self)) for accurate signature.
|
|   __ior__(self, value, /)
|       Return self|=value.
|
|   __isub__(self, value, /)
|       Return self-=value.
```

```

|  __iter__(self, /)
|      Implement iter(self).
|
|  __ixor__(self, value, /)
|      Return self^=value.
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __reduce__(...)
|      Return state information for pickling.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(...)
|      S.__sizeof__() -> size of S in memory, in bytes
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __xor__(self, value, /)
|      Return self^value.

```

```

| add(...)
|     Add an element to a set.
|
|     This has no effect if the element is already present.
|
| clear(...)
|     Remove all elements from this set.
|
| copy(...)
|     Return a shallow copy of a set.
|
| difference(...)
|     Return the difference of two or more sets as a new set.
|
|     (i.e. all elements that are in this set but not the others.)
|
| difference_update(...)
|     Remove all elements of another set from this set.
|
| discard(...)
|     Remove an element from a set if it is a member.
|
|     If the element is not a member, do nothing.
|
| intersection(...)
|     Return the intersection of two sets as a new set.
|
|     (i.e. all elements that are in both sets.)
|
| intersection_update(...)
|     Update a set with the intersection of itself and another.
|
| isdisjoint(...)
|     Return True if two sets have a null intersection.
|
| issubset(...)
|     Report whether another set contains this set.
|
| issuperset(...)
|     Report whether this set contains another set.
|
| pop(...)
|     Remove and return an arbitrary set element.
|     Raises KeyError if the set is empty.
|
| remove(...)
|     Remove an element from a set; it must be a member.

```

```

|
|     If the element is not a member, raise a KeyError.
|
| symmetric_difference(...)
|     Return the symmetric difference of two sets as a new set.
|
|     (i.e. all elements that are in exactly one of the sets.)
|
| symmetric_difference_update(...)
|     Update a set with the symmetric difference of itself and another.
|
| union(...)
|     Return the union of sets as a new set.
|
|     (i.e. all elements that are in either set.)
|
| update(...)
|     Update a set with the union of itself and others.
|
| -----
| Static methods defined here:
|
| __new__(*args, **kwargs) from builtins.type
|     Create and return a new object.  See help(type) for accurate signature.
|
| -----
| Data and other attributes defined here:
|
| __hash__ = None

```

```

[9]: # create aset from a list
aset = set([1, 2, 1, 3, 'hello', 'hi', 3])

```

```

[10]: # check the length of aset
len(aset)

```

[10]: 5

```

[11]: print(aset)

```

```

{1, 2, 3, 'hello', 'hi'}

```

```

[12]: # membership test
'hi' in aset

```

[12]: True

```
[13]: 'Hi' in aset
```

```
[13]: False
```

```
[14]: # see all the methods in set  
help(set)
```

Help on class set in module builtins:

```
class set(object)
|   set() -> new empty set object
|   set(iterable) -> new set object
|
|   Build an unordered collection of unique elements.
|
|   Methods defined here:
|
|   __and__(self, value, /)
|       Return self&value.
|
|   __contains__(...)
|       x.__contains__(y) <==> y in x.
|
|   __eq__(self, value, /)
|       Return self==value.
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|   __getattr__(self, name, /)
|       Return getattr(self, name).
|
|   __gt__(self, value, /)
|       Return self>value.
|
|   __iand__(self, value, /)
|       Return self&=value.
|
|   __init__(self, /, *args, **kwargs)
|       Initialize self. See help(type(self)) for accurate signature.
|
|   __ior__(self, value, /)
|       Return self|=value.
|
|   __isub__(self, value, /)
|       Return self-=value.
|
```

```

|  __iter__(self, /)
|      Implement iter(self).
|
|  __ixor__(self, value, /)
|      Return self^=value.
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __reduce__(...)
|      Return state information for pickling.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(...)
|      S.__sizeof__() -> size of S in memory, in bytes
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __xor__(self, value, /)
|      Return self^value.
|

```



```

| add(...)
|     Add an element to a set.
|
|     This has no effect if the element is already present.
|
| clear(...)
|     Remove all elements from this set.
|
| copy(...)
|     Return a shallow copy of a set.
|
| difference(...)
|     Return the difference of two or more sets as a new set.
|
|     (i.e. all elements that are in this set but not the others.)
|
| difference_update(...)
|     Remove all elements of another set from this set.
|
| discard(...)
|     Remove an element from a set if it is a member.
|
|     If the element is not a member, do nothing.
|
| intersection(...)
|     Return the intersection of two sets as a new set.
|
|     (i.e. all elements that are in both sets.)
|
| intersection_update(...)
|     Update a set with the intersection of itself and another.
|
| isdisjoint(...)
|     Return True if two sets have a null intersection.
|
| issubset(...)
|     Report whether another set contains this set.
|
| issuperset(...)
|     Report whether this set contains another set.
|
| pop(...)
|     Remove and return an arbitrary set element.
|     Raises KeyError if the set is empty.
|
| remove(...)
|     Remove an element from a set; it must be a member.
|

```

```

|         If the element is not a member, raise a KeyError.
|
| symmetric_difference(...)
|     Return the symmetric difference of two sets as a new set.
|
|     (i.e. all elements that are in exactly one of the sets.)
|
| symmetric_difference_update(...)
|     Update a set with the symmetric difference of itself and another.
|
| union(...)
|     Return the union of sets as a new set.
|
|     (i.e. all elements that are in either set.)
|
| update(...)
|     Update a set with the union of itself and others.
|
| -----
| Static methods defined here:
|
| __new__(*args, **kwargs) from builtins.type
|     Create and return a new object.  See help(type) for accurate signature.
|
| -----
| Data and other attributes defined here:
|
| __hash__ = None

```

```
[15]: aset.add(100)
```

```
[16]: aset
```

```
[16]: {1, 100, 2, 3, 'hello', 'hi'}
```

```
[17]: # add 100 again; no effect as 100 already is a member of aset
      aset.add(100)
```

```
[18]: aset
```

```
[18]: {1, 100, 2, 3, 'hello', 'hi'}
```

```
[19]: bset = frozenset(aset)
```

```
[20]: bset
```

```
[20]: frozenset({1, 100, 2, 3, 'hello', 'hi'})
```

```
[21]: help(frozenset)
```

Help on class frozenset in module builtins:

```
class frozenset(object)
|   frozenset() -> empty frozenset object
|   frozenset(iterable) -> frozenset object
|
|   Build an immutable unordered collection of unique elements.
|
|   Methods defined here:
|
|   __and__(self, value, /)
|       Return self&value.
|
|   __contains__(...)
|       x.__contains__(y) <==> y in x.
|
|   __eq__(self, value, /)
|       Return self==value.
|
|   __ge__(self, value, /)
|       Return self>=value.
|
|   __getattr__(self, name, /)
|       Return getattr(self, name).
|
|   __gt__(self, value, /)
|       Return self>value.
|
|   __hash__(self, /)
|       Return hash(self).
|
|   __iter__(self, /)
|       Implement iter(self).
|
|   __le__(self, value, /)
|       Return self<=value.
|
|   __len__(self, /)
|       Return len(self).
|
|   __lt__(self, value, /)
|       Return self<value.
|
```

```

|  __ne__(self, value, /)
|      Return self!=value.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __reduce__(...)
|      Return state information for pickling.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __sizeof__(...)
|      S.__sizeof__() -> size of S in memory, in bytes
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  copy(...)
|      Return a shallow copy of a set.
|
|  difference(...)
|      Return the difference of two or more sets as a new set.
|
|      (i.e. all elements that are in this set but not the others.)
|
|  intersection(...)
|      Return the intersection of two sets as a new set.
|
|      (i.e. all elements that are in both sets.)
|
|  isdisjoint(...)
|      Return True if two sets have a null intersection.

```

```

|
|  issubset(...)
|      Report whether another set contains this set.
|
|  issuperset(...)
|      Report whether this set contains another set.
|
|  symmetric_difference(...)
|      Return the symmetric difference of two sets as a new set.
|
|      (i.e. all elements that are in exactly one of the sets.)
|
|  union(...)
|      Return the union of sets as a new set.
|
|      (i.e. all elements that are in either set.)
|
|  -----
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.

```

```
[22]: intersection = bset.intersection(aset)
```

```
[23]: intersection
```

```
[23]: frozenset({1, 100, 2, 3, 'hello', 'hi'})
```

```
[24]: cset = aset.copy()
```

```
[25]: cset.add(500)
```

```
[26]: print(cset.intersection(aset))
```

```
{1, 2, 3, 100, 'hello', 'hi'}
```

```
[27]: cset.union(aset)
```

```
[27]: {1, 100, 2, 3, 500, 'hello', 'hi'}
```

1.5 Collections

<https://docs.python.org/3/library/collections.html#module-collections>

1.6 deque

- list-like container with fast appends and pops on either end

```
[29]: from collections import deque
```

```
[30]: a = deque([10, 20, 30])
```

```
[31]: # add 1 to the right side of the queue  
a.append(1)
```

```
[32]: a
```

```
[32]: deque([10, 20, 30, 1])
```

```
[34]: # add -1 to the left side of the queue  
a.appendleft(-1)
```

```
[35]: a
```

```
[35]: deque([-1, 10, 20, 30, 1])
```

```
[33]: help(deque)
```

Help on class deque in module collections:

```
class deque(builtins.object)  
| deque([iterable[, maxlen]]) --> deque object  
|  
| A list-like sequence optimized for data accesses near its endpoints.  
|  
| Methods defined here:  
|  
| __add__(self, value, /)  
|     Return self+value.  
|  
| __bool__(self, /)  
|     self != 0  
|  
| __contains__(self, key, /)  
|     Return key in self.  
|  
| __copy__(...)  
|     Return a shallow copy of a deque.  
|  
| __delitem__(self, key, /)  
|     Delete self[key].  
|
```

```

|  __eq__(self, value, /)
|      Return self==value.
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattr__(self, name, /)
|      Return getattr(self, name).
|
|  __getitem__(self, key, /)
|      Return self[key].
|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iadd__(self, value, /)
|      Implement self+=value.
|
|  __imul__(self, value, /)
|      Implement self*=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __reduce__(...)
|      Return state information for pickling.
|
|  __repr__(self, /)
|      Return repr(self).
|

```

```

|  __reversed__(...)
|      D.__reversed__() -- return a reverse iterator over the deque
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(...)
|      D.__sizeof__() -- size of D in memory, in bytes
|
|  append(...)
|      Add an element to the right side of the deque.
|
|  appendleft(...)
|      Add an element to the left side of the deque.
|
|  clear(...)
|      Remove all elements from the deque.
|
|  copy(...)
|      Return a shallow copy of a deque.
|
|  count(...)
|      D.count(value) -> integer -- return number of occurrences of value
|
|  extend(...)
|      Extend the right side of the deque with elements from the iterable
|
|  extendleft(...)
|      Extend the left side of the deque with elements from the iterable
|
|  index(...)
|      D.index(value, [start, [stop]]) -> integer -- return first index of
value.
|      Raises ValueError if the value is not present.
|
|  insert(...)
|      D.insert(index, object) -- insert object before index
|
|  pop(...)
|      Remove and return the rightmost element.
|
|  popleft(...)
|      Remove and return the leftmost element.
|
|  remove(...)

```



```

|         D.remove(value) -- remove first occurrence of value.
|
|     reverse(...)
|         D.reverse() -- reverse *IN PLACE*
|
|     rotate(...)
|         Rotate the deque n steps to the right (default n=1).  If n is negative,
rotates left.
|
|     -----
|     Static methods defined here:
|
|     __new__(*args, **kwargs) from builtins.type
|         Create and return a new object.  See help(type) for accurate signature.
|
|     -----
|     Data descriptors defined here:
|
|     maxlen
|         maximum size of a deque or None if unbounded
|
|     -----
|     Data and other attributes defined here:
|
|     __hash__ = None

```

1.7 defaultdict

- dict subclass that calls a factory function to supply missing values

```
[36]: from collections import defaultdict
```

```
[37]: dd = defaultdict(int) # uses 0 value to supply for missing key
```

```
[38]: # increment value of key 'a' by 1
dd['a'] += 1
```

```
[39]: dd
```

```
[39]: defaultdict(int, {'a': 1})
```

1.8 OrderedDict

- <https://docs.python.org/3/library/collections.html#collections.OrderedDict>
- dict subclass that remembers the order entries were added
- from Python 3.6 dict works as OrderedDict to some extent
- remembers the order the keys were last inserted

- if a new entry overwrites an existing entry, the original insertion position is changed and moved to the end
 - application in generating Most Recently Used (MRU) and LRU caches
- important method:

```
popitem(last=True)
```

- returns and removes a (key, value) pair
- the pairs are returned in LIFO order if last is true or FIFO order if false.

1.9 Counter

- one of the applications of dict is to keep count of certain keys (e.g., word histogram)
- can use Counter – dict subclass for counting hashable objects
- in case of a tie, Counter remembers the order of the key

```
[3]: from collections import Counter
```

```
[4]: c = Counter('apple') # a new counter from an iterable
```

```
[5]: c
```

```
[5]: Counter({'a': 1, 'p': 2, 'l': 1, 'e': 1})
```

```
[6]: # counter from iterable
d = Counter(['apple', 'apple', 'ball'])
```

```
[7]: d
```

```
[7]: Counter({'apple': 2, 'ball': 1})
```

```
[8]: e = Counter({'apple': 10, 'ball': 20}) # counter from mapping
```

```
[9]: e
```

```
[9]: Counter({'apple': 10, 'ball': 20})
```

```
[10]: f = c+e
```

```
[11]: f
```

```
[11]: Counter({'a': 1, 'p': 2, 'l': 1, 'e': 1, 'apple': 10, 'ball': 20})
```

```
[12]: f = f+d
```

```
[13]: f
```

```
[13]: Counter({'a': 1, 'p': 2, 'l': 1, 'e': 1, 'apple': 12, 'ball': 21})
```

```
[14]: f.most_common()
```

```
[14]: [('ball', 21), ('apple', 12), ('p', 2), ('a', 1), ('l', 1), ('e', 1)]
```

```
[15]: help(Counter)
```

Help on class Counter in module collections:

```
class Counter(builtins.dict)
|   Counter(iterable=None, /, **kwds)
|
|   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('abcdeababcdabacaba') # 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
|   'aaaaabbbbccccdde'
|   >>> 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)]
|
| Method resolution order:
|     Counter
|     builtins.dict
|     builtins.object
|
| Methods defined here:
|
| __add__(self, other)
|     Add counts from two counters.
|
|     >>> Counter('abbb') + Counter('bcc')
|     Counter({'b': 4, 'c': 2, 'a': 1})
|
| __and__(self, other)
|     Intersection is the minimum of corresponding counts.
|
|     >>> Counter('abbb') & Counter('bcc')
|     Counter({'b': 1})
|
| __delitem__(self, elem)
|     Like dict.__delitem__() but does not raise KeyError for missing values.
|
| __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})
|
| __iand__(self, other)
|     Inplace intersection is the minimum of corresponding counts.
|
|     >>> c = Counter('abbb')
|     >>> c &= Counter('bcc')
|     >>> c
|     Counter({'b': 1})
|
| __init__(self, iterable=None, /, **kwargs)
|     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
|
|     __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})
|
|     __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})
|
|     __missing__(self, key)
|         The count of elements not in the Counter is zero.
|
|     __neg__(self)
|         Subtracts from an empty counter. Strips positive and zero counts,
|         and flips the sign on negative counts.
|
|     __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})
|
|     __pos__(self)
|         Adds an empty counter, effectively stripping negative and zero counts
|
|     __reduce__(self)
|         Helper for pickle.
|
|     __repr__(self)
|         Return repr(self).
|
|     __sub__(self, other)

```

```

|         Subtract count, but keep only results with positive counts.
|
|         >>> Counter('abbbc') - Counter('bccd')
|         Counter({'b': 2, 'a': 1})
|
|     copy(self)
|         Return a shallow copy.
|
|     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.
|
|     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)]
|
|     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 watch
|         0

```

```

|         >>> c['w']                                # 1 in which, minus 1 in witch,
minus 1 in watch
|         -1
|
|         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
|
|         -----
|         Class methods defined here:
|
|         fromkeys(iterable, v=None) from builtins.type
|             Create a new dictionary with keys from iterable and values set to value.
|
|         -----
|         Data descriptors defined here:
|
|         __dict__
|             dictionary for instance variables (if defined)
|
|         __weakref__
|             list of weak references to the object (if defined)
|
|         -----
|         Methods inherited from builtins.dict:
|
|         __contains__(self, key, /)
|             True if the dictionary has the specified key, else False.
|
|         __eq__(self, value, /)
|             Return self==value.
|
|         __ge__(self, value, /)
|             Return self>=value.
|
|         __getattr__(self, name, /)
|             Return getattr(self, name).
|
|         __getitem__(...)
|             x.__getitem__(y) <==> x[y]

```

```

|
|  __gt__(self, value, /)
|      Return self>value.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __reversed__(self, /)
|      Return a reverse iterator over the dict keys.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(...)
|      D.__sizeof__() -> size of D in memory, in bytes
|
|  clear(...)
|      D.clear() -> None.  Remove all items from D.
|
|  get(self, key, default=None, /)
|      Return the value for key if key is in the dictionary, else default.
|
|  items(...)
|      D.items() -> a set-like object providing a view on D's items
|
|  keys(...)
|      D.keys() -> a set-like object providing a view on D's keys
|
|  pop(...)
|      D.pop(k[,d]) -> v, remove specified key and return the corresponding
value.
|      If key is not found, d is returned if given, otherwise KeyError is
raised
|
|  popitem(self, /)
|      Remove and return a (key, value) pair as a 2-tuple.

```



```

|
|     Pairs are returned in LIFO (last-in, first-out) order.
|     Raises KeyError if the dict is empty.
|
|
|    .setdefault(self, key, default=None, /)
|         Insert key with a value of default if key is not in the dictionary.
|
|         Return the value for key if key is in the dictionary, else default.
|
|     values(...)
|         D.values() -> an object providing a view on D's values
|
|     -----
|     Static methods inherited from builtins.dict:
|
|     __new__(*args, **kwargs) from builtins.type
|         Create and return a new object.  See help(type) for accurate signature.
|
|     -----
|     Data and other attributes inherited from builtins.dict:
|
|     __hash__ = None

```

1.10 heapq

- min/max priority queue
- <https://docs.python.org/3/library/heapq.html>
- heaps are binary trees for which every parent node has a value less than or equal to any of its children
 - min priority queue
- for max priority queue, negate the values of the keys in the priority queue
- use [] list to build heap one element at a time or use `heapify()` function to transform a list into the priority queue

```
[5]: import heapq
```

```
[7]: # build heap one element at a time
heap = []
for i in range(10, 0, -1):
    heapq.heappush(heap, i)
```

```
[8]: # pop the elements from the queue
while heap:
    print('priority:', heapq.heappop(heap))
# essentially is a heapsort with O(nlogn)
```

```
priority: 1
```

```
priority: 2
priority: 3
priority: 4
priority: 5
priority: 6
priority: 7
priority: 8
priority: 9
priority: 10
```

```
[13]: import random
      # sample 10 random integers between 1 and 50
      alist = random.sample(range(1, 50), 10)
```

```
[14]: alist
```

```
[14]: [12, 38, 49, 6, 26, 14, 47, 25, 29, 8]
```

```
[15]: heapq.heapify(alist)
```

```
[16]: alist
```

```
[16]: [6, 8, 14, 25, 12, 49, 47, 38, 29, 26]
```

```
[17]: # pop the elements from the queue
      while alist:
          print('priority:', heapq.heappop(alist))
      # essentially is a heapsort with  $O(n \log n)$ 
```

```
priority: 6
priority: 8
priority: 12
priority: 14
priority: 25
priority: 26
priority: 29
priority: 38
priority: 47
priority: 49
```

1.11 Exercises

1.11.1 Kattis problems

- Some kattis problems that can be solved using Python built-in data structures
 1. sort - <https://open.kattis.com/problems/sort>
 2. Trending Topic - <https://open.kattis.com/problems/trendingtopic>
 3. FizzBuzz2 - <https://open.kattis.com/problems/fizzbuzz2>

4. CD - <https://open.kattis.com/problems/cd>
 - Hint: implement set intersection of sorted list; don't use built-in set as it's slower for Python
5. Keyboardd - <https://open.kattis.com/problems/keyboardd>
 - Hint: two Counters; print the difference
6. Course Scheduling - <https://open.kattis.com/problems/coursescheduling>
 - Hint: Counter of courses, defaultdict(set) of courseToStudents
7. Train Boarding - <https://open.kattis.com/problems/trainboarding>
 - Hint: Counter or List
8. Shopping List - <https://open.kattis.com/problems/shoppinglist>
 - Hint: Use set to keep track of intersection and sort the final list
9. Knigs of the Forest - <https://open.kattis.com/problems/knigsoftheforest>
 - Hint: sort contestants based on year and use priority queue keeping K contestants per year and finding the winner
10. Seven Wonders - <https://open.kattis.com/problems/sevenwonders>
 - Hint: Counter
11. Select Group - <https://open.kattis.com/problems/selectgroup>
 - Stack for RPN parsing and Set

[]: