8.3. collections — Container datatypes

Source code: Lib/collections/__init__.py

This module implements specialized container datatypes providing alternatives to Python's general purpose built-in containers, dict, list, set, and tuple.

namedtuple()	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

Changed in version 3.3: Moved Collections Abstract Base Classes to the collections.abc module. For backwards compatibility, they continue to be visible in this module as well.

8.3.1. ChainMap objects

New in version 3.3.

A ChainMap class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls.

The class can be used to simulate nested scopes and is useful in templating.

class collections.ChainMap(*maps)

A ChainMap groups multiple dicts or other mappings together to create a single, updateable view. If no *maps* are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.

The underlying mappings are stored in a list. That list is public and can be accessed or updated using the *maps* attribute. There is no other state.

Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.

A ChainMap incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in ChainMap.

All of the usual dictionary methods are supported. In addition, there is a *maps* attribute, a method for creating new subcontexts, and a property for accessing all but the first mapping:

maps

A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

new_child(m=None)

Returns a new ChainMap containing a new map followed by all of the maps in the current instance. If m is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to d.new_child() is equivalent to: ChainMap({}}, *d.maps). This method is used for creating subcontexts that can be updated without altering values in any of the parent mappings.

Changed in version 3.4: The optional m parameter was added.

parents

Property returning a new ChainMap containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search. Use cases are similar to those for the nonlocal keyword used in nested scopes. The use cases also parallel those for the built-in super() function. A reference to d.parents is equivalent to: ChainMap(*d.maps[1:]).

See also:

- The MultiContext class in the Enthought CodeTools package has options to support writing to any mapping in the chain.
- Django's Context class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the new_child() method and the parents() property.
- The Nested Contexts recipe has options to control whether writes and other mutations apply only to the first mapping or to any mapping in the chain.
- A greatly simplified read-only version of Chainmap.

8.3.1.1. ChainMap Examples and Recipes

This section shows various approaches to working with chained maps.

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace).items() if v}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the ChainMap class to simulate nested contexts:

```
c = ChainMap() # Create root context
d = c.new child() # Create nested child context
e = c.new_child() # Child of c, independent from d
e.maps[0] # Current context dictionary -- like Python's locals()
e.maps[-1]
                # Root context -- like Python's globals()
e.parents
                # Enclosing context chain -- like Python's nonlocals
d['x']
              # Get first key in the chain of contexts
d['x'] = 1
              # Set value in current context
del d['x']
              # Delete from current context
list(d)
             # All nested values
k in d
              # Check all nested values
```

```
len(d) # Number of nested values
d.items() # All nested items
dict(d) # Flatten into a regular dictionary
```

The ChainMap class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```
class DeepChainMap(ChainMap):
  'Variant of ChainMap that allows direct updates to inner scopes'
  def ___setitem___(self, key, value):
    for mapping in self.maps:
      if key in mapping:
        mapping[key] = value
        return
    self.maps[0][key] = value
  def ___delitem___(self, key):
    for mapping in self.maps:
      if key in mapping:
        del mapping[key]
        return
    raise KeyError(key)
>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lio
>>> d['lion'] = 'orange' # update an existing key two levels down
>>> d['snake'] = 'red' # new keys get added to the topmost dict
>>> del d['elephant'] # remove an existing key one level down
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange
```

8.3.2. Counter objects

A counter tool is provided to support convenient and rapid tallies. For example:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
... cnt[word] += 1
```

class collections. Counter([iterable-or-mapping])

A Counter is a dict subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The Counter class is similar to bags or multisets in other languages.

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```
>>> c = Counter() # a new, empty counter
>>> c = Counter('gallahad') # a new counter from an iteral
>>> c = Counter({'red': 4, 'blue': 2}) # a new counter from a ma
>>> c = Counter(cats=4, dogs=8) # a new counter from keyw
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a KeyError:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon'] # count of a missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use del to remove it entirely:

```
>>> c['sausage'] = 0 # counter entry with a zero count with a zero
```

New in version 3.1.

Counter objects support three methods beyond those available for all dictionaries:

elements()

Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element's count is less than one, elements() will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'b', 'b']
```

most_common([n])

Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or None, most_common() returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily:

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

subtract([iterable-or-mapping])

Elements are subtracted from an *iterable* or from another *mapping* (or counter). Like dict.update() but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)

>>> d = Counter(a=1, b=2, c=3, d=4)

>>> c.subtract(d)

>>> c

Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

New in version 3.2.

The usual dictionary methods are available for Counter objects except for two which work differently for counters.

fromkeys(iterable)

This class method is not implemented for Counter objects.

update([iterable-or-mapping])

Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like dict.update() but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of (key, value) pairs.

Common patterns for working with Counter objects:

```
sum(c.values())
                         # total of all counts
c.clear()
                    # reset all counts
        ) # reset all counts
# list unique elements
list(c)
set(c)
                   # convert to a set
dict(c)
                   # convert to a regular dictionary
                      # convert to a list of (elem, cnt) pairs
c.items()
Counter(dict(list of pairs)) # convert from a list of (elem, cnt) pairs
c.most common()[:-n-1:-1] # n least common elements
                   # remove zero and negative counts
+0
```

Several mathematical operations are provided for combining Counter objects to produce multisets (counters that have counts greater than zero). Addition and subtraction combine counters by adding or subtracting the counts of corresponding elements. Intersection and union return the minimum and maximum of corresponding counts. Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

Unary addition and subtraction are shortcuts for adding an empty counter or subtracting from an empty counter.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

New in version 3.3: Added support for unary plus, unary minus, and in-place multiset operations.

Note: Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The Counter class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you *could* store anything in the value field.
- The most_common() method requires only that the values be orderable.
- For in-place operations such as c[key] += 1, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for update() and subtract() which allow negative and zero values for both inputs and outputs.
- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.
- The elements() method requires integer counts. It ignores zero and negative counts.

See also:

- Bag class in Smalltalk.
- Wikipedia entry for Multisets.
- C++ multisets tutorial with examples.
- For mathematical operations on multisets and their use cases, see *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19.*
- To enumerate all distinct multisets of a given size over a given set of elements, see itertools.combinations_with_replacement():

map(Counter, combinations_with_replacement('ABC', 2)) -> AA AB AC BB BC CC

8.3.3. deque objects

class collections.deque([iterable[, maxlen]])

Returns a new deque object initialized left-to-right (using append()) with data from *iterable*. If *iterable* is not specified, the new deque is empty.

Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(o) and insert(o, v) operations which change both the size and position of the underlying data representation.

If *maxlen* is not specified or is *None*, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end. Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

Deque objects support the following methods:

append(x)

Add *x* to the right side of the deque.

appendleft(x)

Add *x* to the left side of the deque.

clear()

Remove all elements from the deque leaving it with length 0.

copy()

Create a shallow copy of the deque.

New in version 3.5.

count(x)

Count the number of deque elements equal to x.

New in version 3.2.

extend(*iterable*)

Extend the right side of the deque by appending elements from the iterable argument.

extendleft(iterable)

Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

index(x[, start[, stop]])

Return the position of *x* in the deque (at or after index *start* and before index *stop*). Returns the first match or raises ValueError if not found.

New in version 3.5.

insert(i, x)

Insert x into the deque at position i.

If the insertion would cause a bounded deque to grow beyond *maxlen*, an IndexError is raised.

New in version 3.5.

pop()

Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

popleft()

Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

remove(*value*)

Remove the first occurrence of *value*. If not found, raises a ValueError.

reverse()

Reverse the elements of the deque in-place and then return None.

New in version 3.2.

rotate(n)

Rotate the deque n steps to the right. If n is negative, rotate to the left. Rotating one step to the right is equivalent to: d.appendleft(d.pop()).

Deque objects also provide one read-only attribute:

maxlen

Maximum size of a deque or *None* if unbounded.

New in version 3.1.

In addition to the above, deques support iteration, pickling, len(d), reversed(d), copy.copy(d), copy.deepcopy(d), membership testing with the in operator, and subscript references such as d[-1]. Indexed access is O(1) at both ends but slows to O(n) in the middle. For fast random access, use lists instead.

```
Starting in version 3.5, deques support __add__(), __mul__(), and __imul__().
```

Example:

```
>>>
>>> from collections import deque
>>> d = deque('ghi') # make a new deque with three items
>>> for elem in d: # iterate over the deque's elements
... print(elem.upper())
G
Η
>>> d.append('j') # add a new entry to the right side
>>> d.appendleft('f') # add a new entry to the left side
                           # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])
                             # return and remove the rightmost item
>>> d.pop()
>>> d.popleft()
                               # return and remove the leftmost item
1 f 1
>>> list(d)
                             # list the contents of the deque
['g', 'h', 'i']
>>> d[0]
                             # peek at leftmost item
'g'
>>> d[-1]
                             # peek at rightmost item
۱i۱
>>> list(reversed(d))
                                   # list the contents of a deque in reverse
['i', 'h', 'g']
```

```
>>> 'h' in d # search the deque
True
>>> d.extend('jkl') # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
                  # right rotation
>>> d.rotate(1)
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)
                         # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> deque(reversed(d))
                             # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()
                        # empty the deque
            # cannot pop from an empty deque
>>> d.pop()
Traceback (most recent call last):
 File "<pyshell#6>", line 1, in -toplevel-
    d.pop()
IndexError: pop from an empty deque
>>> d.extendleft('abc') # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

8.3.3.1. deque Recipes

This section shows various approaches to working with deques.

Bounded length deques provide functionality similar to the tail filter in Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
    return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

The rotate() method provides a way to implement deque slicing and deletion. For example, a pure Python implementation of del d[n] relies on the rotate() method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

To implement deque slicing, use a similar approach applying rotate() to bring a target element to the left side of the deque. Remove old entries with popleft(), add new entries with extend(), and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as dup, drop, swap, over, pick, rot, and roll.

8.3.4. defaultdict objects

class collections. defaultdict([default_factory[, ...]])

Returns a new dictionary-like object. defaultdict is a subclass of the built-in dict class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the dict class and is not documented here.

The first argument provides the initial value for the default_factory attribute; it defaults to None. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

defaultdict objects support the following method in addition to the standard dict operations:

__missing__(key)

If the default_factory attribute is None, this raises a KeyError exception with the *key* as argument.

If default_factory is not None, it is called without arguments to provide a default value for the given *key*, this value is inserted in the dictionary for the *key*, and returned.

If calling default_factory raises an exception this exception is propagated unchanged.

This method is called by the <u>__getitem__()</u> method of the dict class when the requested key is not found; whatever it returns or raises is then returned or raised by <u>__getitem__()</u>.

Note that __missing__() is *not* called for any operations besides __getitem__(). This means that get() will, like normal dictionaries, return None as a default rather than using default_factory.

defaultdict objects support the following instance variable:

default_factory

This attribute is used by the <u>missing</u>() method; it is initialized from the first argument to the constructor, if present, or to None, if absent.

8.3.4.1. defaultdict Examples

Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]>
>>> d = defaultdict(list)
>>> for k, v in s:
... d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the default_factory function which returns an empty list. The list.append() operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the list.append() operation adds another value to the list. This technique is simpler and faster than an equivalent technique using dict.setdefault():

```
>>> d = {}
>>> for k, v in s:
... d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the default_factory to int makes the defaultdict useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
... d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the default_factory function calls int() to supply a default count

of zero. The increment operation then builds up the count for each letter.

The function int() which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```
>>> def constant_factory(value):
... return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the default_factory to set makes the defaultdict useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', >>> d = defaultdict(set)
>>> for k, v in s:
... d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5. namedtuple() Factory Function for Tuples with Named Fields

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index.

collections. **namedtuple**(*typename*, *field_names*, *verbose=False*, *rename=False*)

Returns a new tuple subclass named typename. The new subclass is used to create tuple-like objects that have fields

accessible by attribute lookup as well as being indexable and iterable. Instances of the subclass also have a helpful docstring (with typename and field_names) and a helpful __repr__() method which lists the tuple contents in a name=value format.

The *field_names* are a single string with each fieldname separated by whitespace and/or commas, for example 'x y' or 'x, y'. Alternatively, *field_names* can be a sequence of strings such as ['x', 'y'].

Any valid Python identifier may be used for a fieldname except for names starting with an underscore. Valid identifiers consist of letters, digits, and underscores but do not start with a digit or underscore and cannot be a keyword such as *class*, *for*, *return*, *global*, *pass*, or *raise*.

If *rename* is true, invalid fieldnames are automatically replaced with positional names. For example, ['abc', 'def', 'ghi', 'abc'] is converted to ['abc', '_1', 'ghi', '_3'], eliminating the keyword def and the duplicate fieldname abc.

If *verbose* is true, the class definition is printed after it is built. This option is outdated; instead, it is simpler to print the source attribute.

Named tuple instances do not have per-instance dictionaries, so they are lightweight and require no more memory than regular tuples.

Changed in version 3.1: Added support for rename.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22) # instantiate with positional or keyword argu
>>> p[o] + p[1] # indexable like the plain tuple (11, 22)

33
>>> x, y = p # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y # fields also accessible by name
```

```
33
>>> p # readable __repr__ with a name=value style
Point(x=11, y=22)
```

Named tuples are especially useful for assigning field names to result tuples returned by the csv or sqlite3 modules:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title
import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employerint(emp.name, emp.title))

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FRC for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

```
classmethod somenamedtuple. _make(iterable)
```

Class method that makes a new instance from an existing sequence or iterable.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple._asdict()

Return a new OrderedDict which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

Changed in version 3.1: Returns an OrderedDict instead of a regular dict.

somenamedtuple._replace(kwargs)

Return a new instance of the named tuple replacing specified fields with new values:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)
>>> for partnum, record in inventory.items():
... inventory[partnum] = record._replace(price=newprices[partnum])
```

somenamedtuple.__source

A string with the pure Python source code used to create the named tuple class. The source makes the named tuple self-documenting. It can be printed, executed using exec(), or saved to a file and imported.

New in version 3.3.

somenamedtuple._fields

Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
>>> p._fields # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

To retrieve a field whose name is stored in a string, use the getattr() function:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-staroperator (as described in Unpacking Argument Lists):

```
>>> d = { 'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

Since a named tuple is a regular Python class, it is easy to add or change functionality with a subclass. Here is how to add a calculated field and a fixed-width print format:

```
>>> class Point(namedtuple('Point', 'x y')):
... __slots__ = ()
... @property
... def hypot(self):
... return (self.x ** 2 + self.y ** 2) ** 0.5
... def __str__(self):
... return 'Point: x=%6.3f y=%6.3f hypot=%6.3f' % (self.x, self.y)

>>> for p in Point(3, 4), Point(14, 5/7):
```

```
>>> for p in Point(3, 4), Point(14, 5/7):
... print(p)
Point: x= 3.000 y= 4.000 hypot= 5.000
Point: x=14.000 y= 0.714 hypot=14.018
```

The subclass shown above sets __slots__ to an empty tuple. This helps keep memory requirements low by preventing the creation of instance dictionaries.

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the _fields attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

Docstrings can be customized by making direct assignments to the __doc__ fields:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

Changed in version 3.5: Property docstrings became writeable.

Default values can be implemented by using _replace() to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
>>> janes_account = default_account._replace(owner='Jane')
```

See also:

 Recipe for named tuple abstract base class with a metaclass mix-in by Jan Kaliszewski. Besides providing an abstract base class for named tuples, it also supports an alternate metaclass-based constructor that is convenient for use cases where named tuples are being subclassed.

8.3.6. OrderedDict objects

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

class collections.OrderedDict([items])

Return an instance of a dict subclass, supporting the usual dict methods. An *OrderedDict* is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

popitem(last=True)

The popitem() method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO order if false.

move_to_end(key, last=True)

Move an existing *key* to either end of an ordered dictionary. The item is moved to the right end if *last* is true (the default) or to the beginning if *last* is false. Raises KeyError if the *key* does not exist:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

New in version 3.2.

In addition to the usual mapping methods, ordered dictionaries also support reverse iteration using reversed().

Equality tests between OrderedDict objects are order-sensitive and are implemented as list(od1.items())==list(od2.items()). Equality tests between OrderedDict objects and other Mapping objects are order-insensitive like regular dictionaries. This allows OrderedDict objects to be substituted anywhere a regular dictionary is used.

The OrderedDict constructor and update() method both accept keyword arguments, but their order is lost because Python's function call semantics pass in keyword arguments using a regular unordered dictionary.

Changed in version 3.5: The items, keys, and values views of OrderedDict now support reverse iteration using reversed().

8.3.6.1. OrderedDict Examples and Recipes

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[o]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[o])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

It is also straight-forward to create an ordered dictionary variant that 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:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

def ___setitem___(self, key, value):
    if key in self:
        del self[key]
    OrderedDict.__setitem___(self, key, value)
```

An ordered dictionary can be combined with the Counter class so that the counter remembers the order elements are first encountered:

```
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountere

def __repr__(self):
    return '%s(%r)' % (self.__class__.__name___, OrderedDict(self)

def __reduce__(self):
    return self.__class___, (OrderedDict(self),)
```

8.3.7. UserDict objects

The class, UserDict acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from dict; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

class collections. UserDict([initialdata])

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the data attribute of UserDict instances. If *initialdata* is provided, data is initialized with its contents; note that a reference to *initialdata* will not be kept, allowing it be used for other purposes.

In addition to supporting the methods and operations of mappings, UserDict instances provide the following attribute:

data

A real dictionary used to store the contents of the UserDict class.

8.3.8. UserList objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and

override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from list; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections. UserList([/ist])

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the data attribute of UserList instances. The instance's contents are initially set to a copy of *list*, defaulting to the empty list []. *list* can be any iterable, for example a real Python list or a UserList object.

In addition to supporting the methods and operations of mutable sequences, <u>UserList</u> instances provide the following attribute:

data

A real list object used to store the contents of the UserList class.

Subclassing requirements: Subclasses of UserList are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

8.3.9. UserString objects

The class, UserString acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from str; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections. UserString([sequence])

Class that simulates a string or a Unicode string object. The instance's content is kept in a regular string object, which is accessible via the data attribute of UserString instances. The instance's contents are initially set to a copy of *sequence*. The *sequence* can be an instance of bytes, str, UserString (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in str() function.

Changed in version 3.5: New methods __getnewargs__, __rmod__, casefold, format_map, isprintable, and maketrans.