4.10. Mapping Types — dict

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary. (For other containers see the built-in list, set, and tuple classes, and the collections module.)

A dictionary’s keys are almost arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (such as 1 and 1.0) then they can be used interchangeably to index the same dictionary entry. (Note however, that since computers store floating-point numbers as approximations it is usually unwise to use them as dictionary keys.)

Dictionaries can be created by placing a comma-separated list of key: value pairs within braces, for example: {'jack': 4098, 'sjoerd': 4127} or {4098: 'jack', 4127: 'sjoerd'}, or by the dict constructor.

class dict(\*\*kwarg)class dict(mapping, \*\*kwarg)class dict(iterable, \*\*kwarg)

Return a new dictionary initialized from an optional positional argument and a possibly empty set of keyword arguments.

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument. If a key being added is already present, the value from the keyword argument replaces the value from the positional argument.

To illustrate, the following examples all return a dictionary equal to {"one": 1, "two": 2, "three": 3}:

>>>>>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

>>> a == b == c == d == e

True

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

len(d)

Return the number of items in the dictionary d.

d[key]

Return the item of d with key key. Raises a KeyError if key is not in the map.

If a subclass of dict defines a method \_\_missing\_\_(), if the key key is not present, the d[key] operation calls that method with the key key as argument. The d[key] operation then returns or raises whatever is returned or raised by the \_\_missing\_\_(key) call if the key is not present. No other operations or methods invoke \_\_missing\_\_(). If \_\_missing\_\_() is not defined, KeyError is raised. \_\_missing\_\_() must be a method; it cannot be an instance variable:

>>>>>> class Counter(dict):

... def \_\_missing\_\_(self, key):

... return 0

>>> c = Counter()

>>> c['red']

0

>>> c['red'] += 1

>>> c['red']

1

See collections.Counter for a complete implementation including other methods helpful for accumulating and managing tallies.

d[key] = value

Set d[key] to value.

del d[key]

Remove d[key] from d. Raises a KeyError if key is not in the map.

key in d

Return True if d has a key key, else False.

key not in d

Equivalent to not key in d.

iter(d)

Return an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()).

clear()

Remove all items from the dictionary.

copy()

Return a shallow copy of the dictionary.

classmethod fromkeys(seq[, value])

Create a new dictionary with keys from seq and values set to value.

fromkeys() is a class method that returns a new dictionary. value defaults to None.

get(key[, default])

Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

items()

Return a new view of the dictionary’s items ((key, value) pairs). See the documentation of view objects.

keys()

Return a new view of the dictionary’s keys. See the documentation of view objects.

pop(key[, default])

If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

popitem()

Remove and return an arbitrary (key, value) pair from the dictionary.

popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling popitem() raises a KeyError.

setdefault(key[, default])

If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None.

update([other])

Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None.

update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: d.update(red=1, blue=2).

values()

Return a new view of the dictionary’s values. See the documentation of view objects.

See also:

types.MappingProxyType can be used to create a read-only view of a dict.

4.10.1. Dictionary view objects

The objects returned by dict.keys(), dict.values() and dict.items() are view objects. They provide a dynamic view on the dictionary’s entries, which means that when the dictionary changes, the view reflects these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests:

len(dictview)

Return the number of entries in the dictionary.

iter(dictview)

Return an iterator over the keys, values or items (represented as tuples of (key, value)) in the dictionary.

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary’s history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of (value, key) pairs using zip(): pairs = zip(d.values(), d.keys()). Another way to create the same list is pairs = [(v, k) for (k, v) in d.items()].

Iterating views while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.

x in dictview

Return True if x is in the underlying dictionary’s keys, values or items (in the latter case, x should be a (key, value) tuple).

Keys views are set-like since their entries are unique and hashable. If all values are hashable, so that (key, value) pairs are unique and hashable, then the items view is also set-like. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class collections.abc.Set are available (for example, ==, <, or ^).

An example of dictionary view usage:

>>>>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}

>>> keys = dishes.keys()

>>> values = dishes.values()

>>> # iteration

>>> n = 0

>>> for val in values:

... n += val

>>> print(n)

504

>>> # keys and values are iterated over in the same order

>>> list(keys)

['eggs', 'bacon', 'sausage', 'spam']

>>> list(values)

[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes

>>> del dishes['eggs']

>>> del dishes['sausage']

>>> list(keys)

['spam', 'bacon']

>>> # set operations

>>> keys & {'eggs', 'bacon', 'salad'}

{'bacon'}

>>> keys ^ {'sausage', 'juice'}

{'juice', 'sausage', 'bacon', 'spam'}