Non sequential data structures:

Dictionaries:

The dict type is not only widely used in our programs but also a fundamental part of the Python implementation. Module namespaces, class and instance attributes, and function keyword arguments are some of the fundamental constructs where dictionaries are deployed. Because of their crucial role, Python dicts are highly optimized. Hash tables are the engines behind Python’s high-performance dicts. The built-in functions live in \_\_builtins\_\_.\_\_dict\_\_.

>>> for i in \_\_builtins\_\_.\_\_dict\_\_:

... print(i)

...

\_\_name\_\_

\_\_doc\_\_

\_\_package\_\_

\_\_loader\_\_

\_\_spec\_\_

\_\_build\_class\_\_

\_\_import\_\_

( . . . )

int

list

map

object

range

reversed

set

slice

staticmethod

str

super

tuple

type  
( . . . )

abs

open

quit

exit

copyright

credits

license

help

>>>

The generic Mapping Types:

Just like we saw with the sequences, the mapping types, the dict and the set type, they come from the collections module, not from collections.abc. Also just like with the sequences, we can determine the behavior and the main functions that they support by considering which features do they inherit from their parent superclasses:

Mutable Mapping and its superclasses from collections.abc

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Container | | \_\_contains\_\_ | | |  | | --- | | Mapping | | \_\_getitem\_\_ | | \_\_contains\_\_ | | \_\_eq\_\_ | | \_\_ne\_\_ | | get() | | items() | | keys() | | values() | | |  | | --- | | Mutable Mapping | | \_\_setitem\_\_ | | \_\_delitem\_\_ | | clear() | | pop() | | popitem() | | setdefault() | | update() | |
| |  | | --- | | Iterable | | \_\_iter\_\_ | |
| |  | | --- | | Sized | | \_\_len\_\_ | |

Implementations of specialized mappings often extend dict or collections.User Dict, instead of these ABCs. The main value of the ABCs is documenting and formalizing the minimal interfaces for mappings, and serving as criteria for isinstance tests in code that needs to support mappings in a broad sense:

>>> my\_dict = {}

>>> isinstance(my\_dict, abc.Mapping)

True

Using isinstance is better than checking whether a function argument is of dict type, because then alternative mapping types can be used. All mapping types in the standard library use the basic dict in their implementation, so they share the limitation that the keys must be hashable (the values need not be hashable, only the keys).

The generic Mapping Types:

Here is part of the definition of hashable from the Python Glossary:

An object is hashable if it has a hash value which never changes during its lifetime (it needs a \_\_hash\_\_() method) and can be compared to other objects (it needs an \_\_eq\_\_() method). Hashable objects which compare equal must have the same hash value. […]

The atomic immutable types (str, bytes, numeric types) are all hashable. A frozen set is always hashable because its elements must be hashable by definition. A tuple is hashable only if all its items are hashable. See tuples tt, tl, and tf:

Hashable: X

>>>

>>> tt = (1, 2, (30, 40))

>>> hash(tt)

8027212646858338501

>>> tl = (1, 2, [30, 40])

>>> hash(tl)

Traceback (most recent call last):

File "", line 1, in <module>

TypeError: unhashable type: 'list'

>>> tf = (1, 2, frozenset([30, 40]))

>>> hash(tf)

-4118419923444501110

>>>

User-defined types are hashable by default because their hash value is their id() and they all compare not equal. If an object implements a custom \_\_eq\_\_ that takes into account its internal state, it may be hashable only if all its attributes are immutable.

How to build a dict type:

Given the ground rules, you can basically build a dictionary with any pair of values, in any kind of structure, if and only if, the first value, the one that will be passed as the key, is an immutable value and it can be hashable.

Dict Type building X

>>>

>>> 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

Dict Comprehensions:

In addition to the literal syntax and the flexible dict constructor, we can use dict com‐ prehensions to build dictionaries. Since Python 2.7, the syntax of listcomps and genexps was applied to dict comprehensions (and set comprehensions as well, which we’ll soon visit). A dictcomp builds a dict instance by producing key:value pair from any iterable.

Dict Comprehension X

>>>

>>> DIAL\_CODES = [

... (86, 'China'),

... (91, 'India'),

... (1, 'United States'),

... (62, 'Indonesia'),

... (55, 'Brazil'),

... (92, 'Pakistan'),

... (880, 'Bangladesh'),

... (234, 'Nigeria'),

... (7, 'Russia'),

... (81, 'Japan'),

... ]

>>> country\_code = {country: code for code, country in DIAL\_CODES}

>>> country\_code

{'China': 86, 'India': 91, 'Bangladesh': 880, 'United States': 1,

'Pakistan': 92, 'Japan': 81, 'Russia': 7, 'Brazil': 55, 'Nigeria':

234, 'Indonesia': 62}

>>> {code: country.upper() for country, code in country\_code.items()

... if code < 66}

{1: 'UNITED STATES', 55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA'}True

Other types of dictionaries:

The basic and standard dictionaries offered by python can do a great deal, but sometimes we need our dictionaries to do more, and more efficient. That’s why we have form collections two more types of dictionaries that solve different types of problems for different types of situations. These are the defaultdict, and the OrederedDict.

Methods and attributes found in dictionaries

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **dict** | | **defaultdict** | **OrederedDict** | |  |
| d.clear() | | **•** | **•** | **•** | Remove all items | |
| d.\_\_contains\_\_(k) | | **•** | **•** | **•** | k in d | |
| d.copy() | | **•** | **•** | **•** | Shallow copy | |
| d.\_\_copy\_\_() | |  | **•** |  | Support for copy.copy | |
| d.default\_factory | |  | **•** |  | Callable invoked by \_\_missing\_\_ to set missing values | |
| d.\_\_delitem\_\_(k) | | **•** | **•** | **•** | del d[k]—remove item with key k | |
| d.fromkeys(it, [initial]) | | **•** | **•** | **•** | New mapping from keys in iterable, with optional initial value (defaults to None) | |
| d.get(k, [default]) | | **•** | **•** | **•** | Get item with key k, return default or None if missing | |
| d.\_\_getitem\_\_(k) | | **•** | **•** | **•** | d[k]—get item with key k | |
| d.items() | | **•** | **•** | **•** | Get view over items—(key, value) pairs | |
| d.iter() | | **•** | **•** | **•** | Get iterator over keys | |
| d.keys() | | **•** | **•** | **•** | Get view over keys | |
| d.\_\_len\_\_() | | **•** | **•** | **•** | len(d)—number of items | |
| d.\_\_missing\_\_(k) | |  | **•** |  | Called when \_\_getitem\_\_ cannot find the key | |
| d.move\_to\_end(k, [last]) | |  |  | **•** | Move k first or last position (last is True by default) | |
| d.pop(k, [default]) | | **•** | **•** | **•** | Remove and return value at k, or default or None if missing | |
| d.popitem() | | **•** | **•** | **•** | Remove and return an arbitrary (key, value) item | |
| d.\_\_reversed\_\_() | |  |  | **•** | Get iterator for keys from last to first inserted | |
| d.setdefault(k,[default]) | | **•** | **•** | **•** | If k in d, return d[k]; else set d[k] = default and return it | |
| d.\_\_setitem\_\_(k, v) | | **•** | **•** | **•** | d[k] = v—put v at k | |
| d.update(m, [\*\*kargs]) | | **•** | **•** | **•** | Update d with items from mapping or iterable of (key, value) pairs | |
| d.values() | | **•** | **•** | **•** | Get view over values | |

default\_factory is not a method, but a callable instance attribute set by the end user when default dict is instantiated.

OrderedDict.popitem() removes the first item inserted (FIFO); an optional last argument, if set to True, pops the last item (LIFO).

Handling Missing keys with setdefault:

In line with the fail-fast philosophy, dict access with d[k] raises an error when k is not an existing key. Every Pythonista knows that d.get(k, default) is an alternative to d[k] whenever a default value is more convenient than handling KeyError. However, when updating the value found (if it is mutable), using either \_\_getitem\_\_ or get is awkward and inefficient. We basically use the setdefault because using the normal get method, the script does 3 searches instead of one, three iterations, which as we know, can be a very expensive operation to repeat. In the other hand, the setdefault method does it all in the first iteration.

Mappings with Flexible Key Lookup:

Sometimes it is convenient to have mappings that return some made-up value when a missing key is searched. There are two main approaches to this: one is to use a default dict instead of a plain dict. The other is to subclass dict or any other mapping type and add a \_\_missing\_\_ method.

The default\_factory of a defaultdict is only invoked to pro‐ vide default values for \_\_getitem\_\_ calls, and not for the other methods. For example, if dd is a defaultdict, and k is a missing key, dd[k] will call the default\_factory to create a default value, but dd.get(k) still returns None.

defaultdict: Another Take on Missing Keys:

A defaultdict is configured to create items on demand whenever a missing key is searched. Here is how it works: when instantiating a defaultdict, you provide a callable that is used to produce a default value whenever \_\_getitem\_\_ is passed a nonexistent key argument.

For example, given an empty defaultdict created as dd = defaultdict(list), if 'new-key' is not in dd, the expression dd['new-key'] does the following steps:

1. Calls list() to create a new list.

2. Inserts the list into dd using 'new-key' as key.

3. Returns a reference to that list.

The StrKeyDict0 :

Suppose you’d like a mapping where keys are converted to str when looked up. A concrete use case is the Pingo.io project, where a programmable board with GPIO pins (e.g., the Raspberry Pi or the Arduino) is represented by a board object with a board.pins attribute, which is a mapping of physical pin locations to pin objects, and the physical location may be just a number or a string like "A0" or "P9\_12". For consistency, it is desirable that all keys in board.pins are strings, but it is also convenient that looking up my\_arduino.pin[13] works as well, so beginners are not tripped when they want to blink the LED on pin 13 of their Arduinos.

class StrKeyDict0(dict):

    def \_\_missing\_\_(self, key):

        if isinstance(key, str):

            raise KeyError(key)

        return self[str(key)]

    def get(self, key, default=None):

        try:

            return self[key]

        except KeyError:

            return default

    def \_\_contains\_\_(self, key):

    return key in self.keys() or str(key) in self.keys()

The \_\_missing\_\_ method:

Underlying the way mappings deal with missing keys is the aptly named \_\_missing\_\_ method. This method is not defined in the base dict class, but dict is aware of it: if you subclass dict and provide a \_\_missing\_\_ method, the standard dict.\_\_getitem\_\_ will call it whenever a key is not found, instead of raising KeyError.

The \_\_missing\_\_ method is just called by \_\_getitem\_\_ (i.e., for the d[k] operator). The presence of a \_\_missing\_\_ method has no effect on the behavior of other methods that look up keys, such as get or \_\_contains\_\_ (which implements the in operator). This is why the default\_factory of defaultdict works only with \_\_getitem\_\_, as noted in the warning at the end of the previous section.

Variations of dict:

*collections.OrderedDict*

Maintains keys in insertion order, allowing iteration over items in a predictable order. The popitem method of an OrderedDict pops the first item by default, but if called as my\_odict.popitem(last=True), it pops the last item added.

*collections.ChainMap*

Holds a list of mappings that can be searched as one. The lookup is performed on each mapping in order and succeeds if the key is found in any of them. This is useful to interpreters for languages with nested scopes, where each mapping represents a scope context. The “ChainMap objects” section of the collections docs has several examples of ChainMap usage, including this snippet inspired by the basic rules of variable lookup in Python:

import builtins

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

>>>

*collections.Counter*

A mapping that holds an integer count for each key. Updating an existing key adds to its count. This can be used to count instances of hashable objects (the keys) or as a multiset—a set that can hold several occurrences of each element. Counter implements the + and - operators to combine tallies, and other useful methods such as most\_common([n]), which returns an ordered list of tuples with the n most common items and their counts; see the documentation. Here is Counter used to count letters in words:

>>> ct = collections.Counter('abracadabra')

>>> ct

Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})

>>> ct.update('aaaaazzz')

>>> ct

Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})

>>> ct.most\_common(2)

[('a', 10), ('z', 3)]

>>>

Subclassing UserDict:

The term subclassing refers to create a class that inherits from another class. In this case, we inherit from UserDict instead of inheriting from the plain dict. We do this because most of the time is much easier to inherit from UserDict in order to make the code faster, this is because most of the methods that we will implement they are already created in the UserDict. Also, when using the plain dict, we might also break it or not implement the methods the right way.

Note that UserDict does not inherit from dict, but has an internal dict instance, called data, which holds the actual items. This avoids undesired recursion when coding special methods like \_\_setitem\_\_ and simplifies the coding of \_\_contains\_\_.

StrKeyDict0 with UserDict:

import collections

class StrKeyDict0(dict):

    def \_\_missing\_\_(self, key):

        if isinstance(key, str):

            raise KeyError(key)

        return self[str(key)]

    def \_\_contains\_\_(self, key):

        return str(key) in self.data

    def \_\_setitem\_\_(self, key, item):

        self.data[str(key)] = item

The main mapping methods(\_\_setitem\_\_ and \_\_getitem\_\_):

MutableMapping.update

This is a very powerful method that is also used whenever we use the \_\_init\_\_ method to initialize a class and load the instance from other mappings, form iterables of (key, value) pairs, and keyword arguments. Also because it uses self[key] = value to add the items, this means that the update method is ultimately using the \_\_setitem\_\_ method.

Mapping.get

This method comes from Mapping, this is because it doesn’t need not be mutable to just get the item from the map. This method ultimately calls the \_\_getitem\_\_ method, and by modifying this one, we modify the get method.

Immutable Mapping:

The mapping types provided by the standard library are all mutable, but you may need to guarantee that a user cannot change a mapping by mistake. A concrete use case can be found, again, in the Pingo.io project I described in “The \_\_missing\_\_ Method”, the board.pins mapping represents the physical GPIO pins on the device. As such, it’s nice to prevent inadvertent updates to board.pins because the hardware can’t possibly be changed via software, so any change in the mapping would make it inconsistent with the physical reality of the device.

MappingProxyType :

Since Python 3.3, the types module provides a wrapper class called MappingProxy Type, which, given a mapping, returns a mappingproxy instance that is a read-only but dynamic view of the original mapping. This means that updates to the original mapping can be seen in the mappingproxy, but changes cannot be made through it.

Output X

>>>

>>> from types import MappingProxyType

>>> d = {1: 'A'}

>>> d\_proxy = MappingProxyType(d)

>>> d\_proxy

mappingproxy({1: 'A'})

>>> d\_proxy[1]

'A'

>>> d\_proxy[2] = 'x'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'mappingproxy' object does not support item assignment

>>> d[2] = 'B'

>>> d\_proxy

mappingproxy({1: 'A', 2: 'B'})

>>>d\_proxy[2]

'B'

>>>

The Set Theory:

Sets are relatively new to python and people don’t really use them a lot, even with all the practical uses that they have. Set type is a mutable Mapping type, but also, there is another type of set called frozenset, which is an immutable mapping type, and the set’s brother.

The most characteristic feature of a set is that they can any kind of immutable objects,. This is why the items being held in a set, must by definition be hashable, this means mutable.

Also, a set can only hold one object referring to the same instance, this means, no repeated objects:

>>> l = ['spam', 'spam', 'eggs', 'spam']

>>> set(l)

{'eggs', 'spam'}

>>>

This ability of the sets is not the only thing that makes them super valuable, what is really remarkable about sets, are the mathematical operations that can be performed with them, which are the same as in mathematics. This means that we can find the union, the intersection between others.

The speed of sets:

When we say that the set type can perform certain operations faster than a list or a tuple for example, we mean it. We can’t really see it unless we use at least 10M of elements, floating elements, to be precise, which we will do later

Take a look at the bytecode for the two operations, as output by dis.dis (the disassembler function):

>>> from dis import dis

>>> dis('{1}')

1 0 LOAD\_CONST 0 (1)

3 BUILD\_SET 1

6 RETURN\_VALUE

>>> dis('set([1])')

1 0 LOAD\_NAME 0 (set)

3 LOAD\_CONST 0 (1)

6 BUILD\_LIST 1

9 CALL\_FUNCTION 1 (1 positional, 0 keyword pair)

12 RETURN\_VALUE

Set Literals:

The syntax of set literals—{1}, {1, 2}, etc.—looks exactly like the math notation, with one important exception: there’s no literal notation for the empty set, so we must remember to write set(). To create an empty set, you should use the constructor without an argument: set(). If you write {}, you’re creating an empty dict—this hasn’t changed.

Notation for set( ):

>>> s = {1}

>>> type(s)

<class 'set'>

>>> s

{1}

>>> s.pop()

1>>> s

set()

There is no special syntax to represent frozenset literals—they must be created by calling the constructor. The standard string representation in Python 3 looks like a frozenset constructor call. Note the output in the console session:

Notation for frozenset( ):

>>> frozenset(range(10))

frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

Set Comprehensions:

Set Comprehensions or setcomps are basically made the same way that listcomps, because we don’t need a pair (key, value) like in a dictionary. We can see it in action in the next example:

Build a set of Latin-1 characters that have the word “SIGN” in theirUnicode names

>>> from unicodedata import name

>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i),'')}

{'§', '=', '¢', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©',

'°', '+', '÷', '±', '>', '¬', '®', '%'}

>>>

Set Comprehensions:

Here we can clearly see all pf the methods that are inherited from left to right from the superclasses from collections.abc and the mathematical operations-like methods like \_\_and\_\_, and \_\_or\_\_.

Mutable Set and its superclasses from collections.abc

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Container | | \_\_contains\_\_ | | |  | | --- | | Set | | isdisjoint | | \_\_le\_\_ | | \_\_lt\_\_ | | \_\_gt\_\_ | | \_\_ge\_\_ | | \_\_eq\_\_ | | \_\_ne\_\_ | | \_\_and\_\_ | | \_\_or\_\_ | | \_\_sub\_\_ | | \_\_xor\_\_ | | |  | | --- | | Mutable Set | | add | | discard | | remove | | pop | | clear | | \_\_ior\_\_ | | \_\_iand\_\_ | | \_\_ixor\_\_ | | \_\_isub\_\_ | |
| |  | | --- | | Iterable | | \_\_iter\_\_ | |
| |  | | --- | | Sized | | \_\_len\_\_ | |

Mathematical set operations

|  |  |  |  |
| --- | --- | --- | --- |
| Math Symbol | Python operator | Method | Description |
| S ∩ Z | s & z | s & z s.\_\_and\_\_(z) | Intersection of s and z |
|  | z & s | z & s s.\_\_rand\_\_(z) | Reversed & operator |
|  |  | s.intersection(it,…) | Intersection of s and all sets built from iterables it, etc. |
|  | s &= z | s.\_\_iand\_\_(z) | s updated with intersection of s and z |
|  |  | s.intersection\_update(it,…) | s updated with intersection of s and all sets built from iterables it, etc. |
|  |  |  |  |
| S ∪ Z | s | z | s.\_\_or\_\_(z) | Union of s and z |
|  | z | s | s.\_\_ror\_\_(z) | Reversed | |
|  |  | s.union(it, …) | Union of s and all sets built from iterables it, etc. |
|  | s |= z | z s.\_\_ior\_\_(z) | s updated with union of s and z |
|  |  | s.update(it, …) | S updated with union of s and all sets built from iterables it, etc. |
|  |  |  |  |
| S \ Z | s - z | s.\_\_sub\_\_(z) | Relative complement or difference between s and z |
|  | z - s | s.\_\_rsub\_\_(z) | Reversed - operator |
|  |  | s.difference(it, …) | Difference between s and all sets built from iterables it, etc. |
|  | s -= z | s.\_\_isub\_\_(z) | s updated with difference between s and z |
|  |  | s.difference\_update(it, …) | s updated with difference between s and all sets built from iterables it, etc. |
|  |  | s.symmetric\_difference(it) | Complement of s & set(it) |
|  |  |  |  |
| S ∆ Z | s ^ z | s.\_\_xor\_\_(z) | Symmetric difference (the complement of the intersection s & z) |
|  | z ^ s | s.\_\_rxor\_\_(z) | Reversed ^ operator |
|  |  | s.symmetric\_differ  ence\_update(it, …) | S updated with symmetric difference of s and all sets built from iterables it, etc. |
|  | s ^= z | s.\_\_ixor\_\_(z) | s updated with symmetric difference of s and z |

Set comparison operators and methods that return a bool

|  |  |  |  |
| --- | --- | --- | --- |
| Math Symbol | Python operator | Method | Description |
|  |  | s.isdisjoint(z) | s and z are disjoint (have no common elements) |
| e ∈ S | e in s | s.\_\_contains\_\_(e) | Element e is a member of s |
| S ⊆ Z | s <= z | s.\_\_le\_\_(z) | s is a subset of the z set |
|  |  | s.issubset(it) | s is a subset of the set built from the iterable it |
| S ⊂ Z | s < z | s.\_\_lt\_\_(z) | s is a proper subset of the z set |
| S ⊇ Z | s >= z | s.\_\_ge\_\_(z) | s is a superset of the z set |
|  |  | s.issuperset(it) | s is a superset of the set built from the iterable it |
| S ⊃ Z | s > z | s.\_\_gt\_\_(z) | s is a proper superset of the z set |

Additional set methods

|  |  |  |  |
| --- | --- | --- | --- |
|  | **set** | **frozenset** |  |
| s.add(e) | **•** |  | Add element e to s |
| s.clear() | **•** |  | Remove all elements of s |
| s.copy() | **•** | **•** | Shallow copy of s |
| s.discard(e) | **•** |  | Remove element e from s if it is present |
| s.\_\_iter\_\_() | **•** | **•** | Get iterator over s |
| s.\_\_len\_\_() | **•** | **•** | len(s) |
| s.pop() | **•** |  | Remove and return an element from s, raising KeyError if s is empty |
| s.remove(e) | **•** |  | Remove element e from s, raising KeyError if e not in s |

Dict and Set under the hood:

Understanding how Python dictionaries and sets are implemented using hash tables is helpful to make sense of their strengths and limitations.

Here are some questions this section will answer:

• How efficient are Python dict and set?

• Why are they unordered?

• Why can’t we use any Python object as a dict key or set element?

• Why does the order of the dict keys or set elements depend on insertion order, and may change during the lifetime of the structure?

• Why is it bad to add items to a dict or set while iterating through it?

To motivate the study of hash tables, we start by showcasing the amazing performance of dict and set with a simple test involving millions of items.

A performance experiment:

Total time for using in operator to search for 1,000 keys in haystacks of 5 sizes, stored as dicts, sets, and lists on a Core i7 laptop running Python 3.4.0

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Len of haystack | Factor | dict time | Factor | set time | Factor | set& time | Factor | List time | Factor |
| 1,000 | 1x | 0.000202s | 1.00x | 0.000143s | 1.00x | 0.000087s | 1.00x | 0.010556s | 1.00x |
| 10,000 | 10x | 0.000140s | 0.69x | 0.000147s | 1.03x | 0.000092s | 1.06x | 0.086586s | 8.20x |
| 100,000 | 100x | 0.000228s | 1.13x | 0.000241s | 1.69x | 0.000163s | 1.87x | 0.871560s | 82.57x |
| 1,000,000 | 1000x | 0.000290s | 1.44x | 0.000332s | 2.32x | 0.000250s | 2.87x | 9.189616s | 870.56x |
| 10,000,000 | 10000x | 0.000337s | 1.67x | 0.000387s | 2.71x | 0.000314s | 3.61x | 97.948056s | 9,278.90x |

Hash Tables in Dictionaries:

A hash table is a sparse array (i.e., an array that always has empty cells). In standard data structure texts, the cells in a hash table are often called “buckets.” In a dict hash table, there is a bucket for each item, and it contains two fields: a reference to the key and a reference to the value of the item. Because all buckets have the same size, access to an individual bucket is done by offset.

Python tries to keep at least 1/3 of the buckets empty; if the hash table becomes too crowded, it is copied to a new location with room for more buckets. To put an item in a hash table, the first step is to calculate the hash value of the item key, which is done with the hash() built-in function, explained next.

Hashes and equality:

The hash() built-in function works directly with built-in types and falls back to calling \_\_hash\_\_ for user-defined types. If two objects compare equal, their hash values must also be equal, otherwise the hash table algorithm does not work. For example, because 1 == 1.0 is true, hash(1) == hash(1.0) must also be true, even though the internal representation of an int and a float are very different.

Also, to be effective as hash table indexes, hash values should scatter around the index space as much as possible. This means that, ideally, objects that are similar but not equal should have hash values that differ widely. . Note how the hashes of 1 and 1.0 are the same, but those of 1.0001, 1.0002, and 1.0003 are very different.

Comparing hash bit patterns of 1, 1.0001, 1.0002, and 1.0003 on a 32-bit build of Python

32-bit Python build

1 00000000000000000000000000000001

!= 0

1.0 00000000000000000000000000000001

------------------------------------------------

1.0 00000000000000000000000000000001

! !!! ! !! ! ! ! ! !! !!! != 16

1.0001 00101110101101010000101011011101

------------------------------------------------

1.0001 00101110101101010000101011011101

!!! !!!! !!!!! !!!!! !! ! != 20

1.0002 01011101011010100001010110111001

------------------------------------------------

1.0002 01011101011010100001010110111001

! ! ! !!! ! ! !! ! ! ! !!!! != 17

1.0003 00001100000111110010000010010110

------------------------------------------------

The hash table algorithm:

To fetch the value at my\_dict[search\_key], Python calls hash(search\_key) to obtain the hash value of search\_key and uses the least significant bits of that number as an offset to look up a bucket in the hash table (the number of bits used depends on the current size of the table). If the found bucket is empty, KeyError is raised. Otherwise, the found bucket has an item—a found\_key: found\_value pair—and then Python checks whether search\_key == found\_key. If they match, that was the item sought: found\_value is returned.

However, if search\_key and found\_key do not match, this is a hash collision. This hap‐ pens because a hash function maps arbitrary objects to a small number of bits, and—in addition—the hash table is indexed with a subset of those bits. In order to resolve the collision, the algorithm then takes different bits in the hash, massages them in a particular way, and uses the result as an offset to look up a different bucket.7 If that is empty, KeyError is raised; if not, either the keys match and the item value is returned, or the collision resolution process is repeated.

Calculate **hash** from **key**

Use other parts of **hash** to locate a different hash table row

hash

collision

no

Equal **keys?**

Empty **bucket?**

Use part of **hash** to locate a **bucket** in hash table

no

yes

yes

Return **value** from **bucket**

Raise **KeyError**

Practical Consequences of How dict Works:

*In the following subsections, we’ll discuss the limitations and benefits that the under‐ lying hash table implementation brings to dict usage.*

Keys must be hashable objects

An object is hashable if all of these requirements are met:

1. It supports the hash() function via a hash() method that always returns the same value over the lifetime of the object.

2. It supports equality via an eq() method.

3. If a == b is True then hash(a) == hash(b) must also be True.

User-defined types are hashable by default because their hash value is their id() and they all compare not equal. If you implement a class with a custom \_\_eq\_\_ method, you must also implement a suitable \_\_hash\_\_, because you must always make sure that if a == b is True then hash(a) == hash(b) is also True. Otherwise, you are breaking an invariant of the hash table algorithm, with the grave consequence that dicts and sets will not deal reliably with your objects. If a custom \_\_eq\_\_ depends on mutable state, then \_\_hash\_\_ must raise TypeError with a message like unhashable type: 'MyClass'.

dicts have significant memory overhead

Because a dict uses a hash table internally, and hash tables must be sparse to work, they are not space efficient. For example, if you are handling a large quantity of records, it makes sense to store them in a list of tuples or named tuples instead of using a list of dictionaries in JSON style, with one dict per record. Replacing dicts with tuples reduces the memory usage in two ways: by removing the overhead of one hash table per record and by not storing the field names again with each record. For user-defined types, the \_\_slots\_\_ class attribute changes the storage of instance attributes from a dict to a tuple in each instance.

Key search is very fast

The dict implementation is an example of trading space for time: dictionaries have significant memory overhead, but they provide fast access regardless of the size of the dictionary—as long as it fits in memory. Also, when we increased the size of a dict from 1,000 to 10,000,000 elements, the time to search grew by a factor of 2.8, from 0.000163s to 0.000456s. The latter figure means we could search more than 2 million keys per second in a dict with 10 million items.

Key ordering depends on insertion order

When a hash collision happens, the second key ends up in a position that it would not normally occupy if it had been inserted first. So, a dict built as dict([(key1, value1), (key2, value2)]) compares equal to dict([(key2, value2), (key1, value1)]), but their key ordering may not be the same if the hashes of key1 and key2 collide.

Here we can see the effect of loading three dicts with the same data, just in different order. The resulting dictionaries all compare equal, even if their order is not the same.

DIAL\_CODES = [

 (86, 'China'),

 (91, 'India'),

 (1, 'United States'),

 (62, 'Indonesia'),

 (55, 'Brazil'),

 (92, 'Pakistan'),

 (880, 'Bangladesh'),

 (234, 'Nigeria'),

 (7, 'Russia'),

 (81, 'Japan'),

 ]

d1 = dict(DIAL\_CODES)

print('d1:', d1.keys())

d2 = dict(sorted(DIAL\_CODES))

print('d2:', d2.keys())

d3 = dict(sorted(DIAL\_CODES, key=lambda x:x[1]))

print('d3:', d3.keys())

assert d1 == d2 and d2 == d3

Output X

>>>

d1: dict\_keys([86, 91, 1, 62, 55, 92, 880, 234, 7, 81])

d2: dict\_keys([1, 7, 55, 62, 81, 86, 91, 92, 234, 880])

d3: dict\_keys([880, 55, 86, 91, 62, 81, 234, 92, 7, 1])

>>>

Adding items to a dict may change the order of existing keys

Whenever you add a new item to a dict, the Python interpreter may decide that the hash table of that dictionary needs to grow. This entails building a new, bigger hash table, and adding all current items to the new table. During this process, new (but different) hash collisions may happen, with the result that the keys are likely to be ordered differently in the new hash table. All of this is implementation-dependent, so you cannot reliably predict when it will happen. If you are iterating over the dictionary keys and changing them at the same time, **your loop may not scan all the items as expected** —not even the items that were already in the dictionary before you added to it.

This is why modifying the contents of a dict while iterating through it is a bad idea. If you need to scan and add items to a dictionary, do it in two steps: read the dict from start to finish and collect the needed additions in a second dict. Then update the first one with it.

In Python 3, the .keys(), .items(), and .values() methods re‐ turn dictionary views, which behave more like sets than the lists returned by these methods in Python 2. Such views are also dynamic: they do not replicate the contents of the dict, and they immediately reflect any changes to the dict.

Practical Consequences of How set Works:

The set and frozenset types are also implemented with a hash table, except that each bucket holds only a reference to the element (as if it were a key in a dict, but without a value to go with it). In fact, before set was added to the language, we often used dictionaries with dummy values just to perform fast membership tests on the keys.

Set elements must be hashable objects.

• Sets have a significant memory overhead.

• Membership testing is very efficient.

• Element ordering depends on insertion order.

• Adding elements to a set may change the order of other elements.

Summary:

Dictionaries:

Dictionaries are a keystone of Python. Beyond the basic dict, the standard library offers handy, ready-to-use specialized mappings like defaultdict, OrderedDict, ChainMap, and Counter, all defined in the collections module. The same module also provides the easy-to-extend UserDict class.

The update and setdefault methods:

Two powerful methods available in most mappings are setdefault and update. The setdefault method is used to update items holding mutable values, for example, in a dict of list values, to avoid redundant searches for the same key. The update method allows bulk insertion or overwriting of items from any other mapping, from iterables providing (key, value) pairs and from keyword arguments. Mapping constructors also use update internally, allowing instances to be initialized from mappings, iterables, or keyword arguments.

Mutable Mapping from collections.abc:

The collections.abc module provides the Mapping and MutableMapping abstract base classes for reference and type checking. The little-known MappingProxyType from the types module creates immutable mappings. There are also ABCs for Set and Mutable Set. Sequence Slicing:

Hash tables:

The hash table implementation underlying dict and set is extremely fast. Understand‐ ing its logic explains why items are apparently unordered and may even be reordered behind our backs. There is a price to pay for all this speed, and the price is in memory.