Python Dictionaries

Table of contents

Creating and indexing 1
`keys`, `values`, `items` 2
`clear` and `copy` 2
`get` 3
Warning: keys must be immutable 3

> **Definition (Dictionary).** A *dictionary* in Python is a data-structure that stores `tuple(key, value)` pairings. Python has a "magic" hash function to find a key among the tuples fast. Dictionaries are **not ordered** and are **mutable**.

Hash tables are a candidate for the most important/useful data-structure in computer science.

The dictionary is not a new idea so much as a generalised index: see Lecture 4B's hash-function section[1] for two hand-rolled encodings, and the figure there showing a key being turned into a list position by two different hash functions. `dict` is what you get when Python supplies that hash function for you, for keys of any (immutable) type. The reason a dictionary has no ordering is precisely that the keys can be of mixed type.

Creating and indexing

```
>>> h = {
...     "red": ["apple", "firetrucks", "cars"],
...     "yellow": ["banana", "cars"],
...     "blue": ["sky", "cars"]
...     }
>>> h["blue"]
['sky', 'cars']
>>> h["green"]
```

[1] courses/csse1001/2025-08-18-non-primitive-data.pdf

```
KeyError: 'green'
>>> h["green"] = ["leaves"]   # fine, because we're assigning not retrieving
```

An empty dictionary is `dict()` (or `{}`).

keys, values, items

```
>>> h.keys()
dict_keys(['red', 'yellow', 'blue'])
>>> h.values()
dict_values([['apple', 'firetrucks', 'cars'], ['banana', 'cars'], ['sky', 'cars']])
>>> h.items()
dict_items([('red', ['apple', 'firetrucks', 'cars']), ('yellow', ['banana', 'cars']), ('blue
```

`items()` is the one that shows the definition literally: each pairing comes back as a `(key, value)` tuple.

clear and copy

```
>>> f = h.copy()
>>> h.clear()
>>> h["blue"]
KeyError: 'blue'
>>> f["blue"]
['sky', 'cars']
```

Like list's `.copy()` (see python-lists[2]), dict's `.copy()` is only a **shallow copy** — mutable values are still shared:

```
>>> f = h.copy()
>>> h["blue"].append("windex")
>>> h["blue"]
['sky', 'cars', 'windex']
>>> f["blue"]
['sky', 'cars', 'windex']   # shallow copy -- same underlying list
```

[2] courses/csse1001/python-lists.pdf

Read the two transcripts together: `clear()` empties the dictionary `h` without touching `f`, because `f` is a different dictionary object — but `h["blue"].append(...)` reaches *through* both dictionaries to the one list they share. Same shallow/deep split as the shallow-copy figure in python-lists[3].

`get`

`.get(key)` is a safer alternative to `h[key]` — it returns `None` instead of raising `KeyError` if the key is missing:

```
>>> h.get("red")
['apple', 'firetrucks', 'cars']
>>> h["green"]
KeyError: 'green'
>>> h.get("green")
None                          # safer, because no error is thrown here
```

Without a method, the same "avoid a `KeyError`" pattern is written explicitly:

```
>>> if key in table:          # avoiding KeyError
...     table[key] += 1
... else:
...     table[key] = 0
```

This is a sketch of the membership-test idiom, not a working counter. As a *frequency counter* the `else` branch would have to be `table[key] = 1` — a key seen for the first time has been seen once, not zero times.

Warning: keys must be immutable

The **keys** of a dictionary **must** be an **immutable** type (see python-primitive-data-types[4] and python-lists[5]):

[3] courses/csse1001/python-lists.pdf
[4] courses/csse1001/python-primitive-data-types.pdf
[5] courses/csse1001/python-lists.pdf

```
>>> d = dict()
>>> d[ [1,2,3] ] = 1
TypeError: unhashable type: 'list'
>>> d[ (1,2,3) ] = 1        # tuples are immutable
>>> h = dict()
>>> d[ h ] = 1
TypeError: unhashable type: 'dict'
```

"Unhashable" is the error message's way of saying the same thing the definition does: a key has to be something the hash function can turn into a fixed location, and a value that can change underneath you cannot be.