EDUFY

Python dictionaries: keys, values and everything inside

Maximilian Bahrami · 11 November 2026

Store values under names instead of positions: make dictionaries, read values (even from lists, tuples and dictionaries inside them), add and change keys, and see every key, value and pair with keys(), values() and items().

A list keeps things in order, and you find them by their position: fruits[0], fruits[1]... But often you don't want "the item at position 2". You want "the price of an apple", or "this user's email". That's what a **dictionary** is for. It stores **key-value pairs**: every value has a name, its **key**, and you get the value back by that name, without needing to know where it is. In this lesson you'll make dictionaries, read values from them (even from lists and dictionaries inside them), add and change values, and list all the keys and values. Press **Run** on any example, then **Edit** to change it.

A key and its value

A dictionary is written in **curly braces** { }. Inside, each pair is a **key**, a **colon** :, and its **value**. **Commas** separate the pairs:

```
{'key1': 'value1', 'key2': 'value2'}
```

To get a value, write the dictionary's name and the **key in square brackets**:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}

print(my_dict['key1'])
print(my_dict['key2'])
```

my_dict['key1'] means "the value stored under the key 'key1'", and that's 'value1'.

A real example: prices

Keys are usually words that describe the value. Here each fruit is a key, and its price is the value:

```
prices = {'apple': 400, 'orange': 250}

print(prices['apple'])
print(prices['orange'])
```

You never had to know that the apple came first. You asked for 'apple', and you got its price. With a list you'd need to remember that the apple's price is at position 0, the orange's at position 1, and so on. That gets hard very fast.

A list or a dictionary?

Both hold many values. The difference is **how you get a value back**:

	List	Dictionary
Written with	[ ]	{ } with key: value
You get a value by its...	**position** (index): fruits[0]	**key** (name): prices['apple']
Positions and slices	yes: fruits[0], fruits[1:3]	no: there are no positions
Use it for	things in an order: steps, a queue, a top 10	things with a name: prices, settings, a person's details

A good test: if you'd describe the data with "first, second, third...", use a list. If you'd describe it with "the price of...", "the name of...", "the age of...", use a dictionary.

Can you count on the order?

> A dictionary has no positions, so you can't ask for "the first item", and slicing doesn't work. When you print one, Python shows the pairs in the order you added them, but you always get a value by its **key**, never by where it is.

Anything can go inside

A value can be anything: a number, a string, a **list**, a **tuple**, even **another dictionary**:

```
stuff = {'k1': 200, 'k2': ['apple', 'orange'], 'k3': ('time', 'year'), 'k4': {'insideKey': 100}}

print(stuff['k1'])
print(stuff['k2'])
print(stuff['k3'])
print(stuff['k4'])
```

To reach something **inside** a value, add more brackets. Python reads them **from left to right**, one step at a time.

A dictionary inside a dictionary:

```
stuff = {'k1': 200, 'k2': ['apple', 'orange'], 'k3': ('time', 'year'), 'k4': {'insideKey': 100}}

print(stuff['k4']['insideKey'])
```

1. stuff['k4'] is the inner dictionary {'insideKey': 100}.
2. ['insideKey'] then takes the value from **that** dictionary: 100.

A tuple inside a dictionary:

```
stuff = {'k1': 200, 'k2': ['apple', 'orange'], 'k3': ('time', 'year'), 'k4': {'insideKey': 100}}

print(stuff['k3'][1])
```

1. stuff['k3'] is the tuple ('time', 'year').
2. [1] is a **position** now, because a tuple is ordered: position 1 is 'year'.

A list inside a dictionary, and then a string method:

```
stuff = {'k1': 200, 'k2': ['apple', 'orange'], 'k3': ('time', 'year'), 'k4': {'insideKey': 100}}

print(stuff['k2'][0].upper())
```

1. stuff['k2'] is the list ['apple', 'orange'].
2. [0] is the first item: 'apple'.
3. .upper() works on that string: 'APPLE'.

This is what makes Python so flexible: whatever a step gives you (a list, a tuple, a dictionary, a string), you can keep going with what that type can do.

Adding a new key

To add a pair, give a new key a value with =:

```
d = {'k1': 2, 'k2': 3}
d['k3'] = 3

print(d)
```

There was no 'k3' before, so Python adds it. The dictionary now has three pairs.

Changing a value

The same = on a key that **already exists** replaces its value:

```
d = {'k1': 2, 'k2': 3}
d['k2'] = 'new value'

print(d)
```

Each key appears **only once** in a dictionary. So d['k2'] = ... never makes a second 'k2'.

It replaces the old value. The new value doesn't have to be the same type: here a number became a string.

Adding and changing look exactly the same. The only difference is whether the key was already there.

All the keys, all the values, all the pairs

Three methods show you what's inside:

```
d = {'k1': 2, 'k2': 'new value', 'k3': 3}

print(d.keys()) # only the keys
print(d.values()) # only the values
print(d.items()) # the pairs, each as (key, value)
```

- d.keys() gives dict_keys(['k1', 'k2', 'k3']): every key.
- d.values() gives dict_values([2, 'new value', 3]): every value.
- d.items() gives dict_items([('k1', 2), ('k2', 'new value'), ('k3', 3)]): every pair, each as a tuple (key, value).

The dict_keys(...) around the result just tells you what kind of collection it is. What matters is what's inside the brackets.

Common mistakes

A colon between the pairs

Wrong

```
my_dict = {'key1': 'value1': 'key2': 'value2'}
```

Right

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
```

> The **colon** goes between a key and its value, the **comma** between two pairs. With a colon in the wrong place, Python stops with a SyntaxError.

Naming a dictionary list

Wrong

```
list = {'k1': 200, 'k2': ['apple', 'orange']}
```

Right

```
stuff = {'k1': 200, 'k2': ['apple', 'orange']}
```

> list is already a name in Python: it's the list type itself (list("abc") from lesson 1). Use it as a variable and it means your dictionary from then on, so list(...) stops working later in the program. The same goes for dict, str, int and print. Pick a name that says what's inside: prices, user, stuff.

Asking for a position

> **Wrong**

```
prices = {'apple': 400, 'orange': 250}
print(prices[0])
```

> **Right**

```
prices = {'apple': 400, 'orange': 250}
print(prices['apple'])
```

> A dictionary has no positions. Python looks for a **key** called 0, doesn't find one, and stops with KeyError: 0. Always use the key.

A key that isn't there, or is spelled differently

> **Wrong**

```
prices = {'apple': 400, 'orange': 250}
print(prices['Apple'])
```

> **Right**

```
prices = {'apple': 400, 'orange': 250}
print(prices['apple'])
```

> Keys must match exactly, capitals included: 'Apple' and 'apple' are two different keys. A key that doesn't exist gives KeyError: 'Apple'. Check the spelling with prices.keys().

Exercises

1. The fruit shop

> 1. Make a dictionary prices with 'apple' at 400, 'orange' at 250 and 'banana' at 150.
> 2. Print the price of a banana.
> 3. The shop starts selling kiwis: add 'kiwi' at 300.
> 4. Apples are on sale: change their price to 350.
> 5. Print all the keys, then all the values.

Answer

```
prices = {'apple': 400, 'orange': 250, 'banana': 150}
print(prices['banana'])
```

```
prices['kiwi'] = 300
prices['apple'] = 350

print(prices.keys())
print(prices.values())
```

> Adding the kiwi and changing the apple are the same kind of line. 'kiwi' was new, so it was added; 'apple' already existed, so its value was replaced.

2. A student's card

> Here's one student's information in a single dictionary:

```
student = {'name': 'lina', 'grades': [8, 9, 7], 'birthday': (2007, 5), 'address': {'city': 'Lund', 'zip': '22100'}}
```

> Using only brackets (and one string method), print:

> 1. the student's **city**
> 2. their **second** grade
> 3. the **year** they were born
> 4. their name in **capital letters**: LINA

Answer

```
student = {'name': 'lina', 'grades': [8, 9, 7], 'birthday': (2007, 5), 'address': {'city': 'Lund', 'zip': '22100'}}

print(student['address']['city']) # the inner dictionary, then its key
print(student['grades'][1]) # the list, then position 1
print(student['birthday'][0]) # the tuple, then position 0
print(student['name'].upper()) # the string, then .upper()
```

> Each line goes one step at a time, like in the lesson. The key gets you to the value. If the value is a list or a tuple, you use a position; if it's a dictionary, you use a key; if it's a string, you can use a string method.

3. Update the card

> Start from the same student dictionary. The student got an email address and moved to Malmö:

> 1. Add a new key 'email' with the value 'lina@example.com'.
> 2. Change the city inside 'address' to 'Malmö'.
> 3. Print the whole dictionary, and then print student.items().

> Hint: to change a value inside the inner dictionary, go through both keys, like when you read it.

Answer

```
student = {'name': 'lina', 'grades': [8, 9, 7], 'birthday': (2007, 5), 'address': {'city': 'Lund', 'zip': '22100'}}

student['email'] = 'lina@example.com'
student['address']['city'] = 'Malmö'

print(student)
print(student.items())
```

> student['address']['city'] = 'Malmö' reads like the city lookup you did in exercise 2, but with = at the end: first student['address'] reaches the inner dictionary, then ['city'] = ... replaces the value there.

Quick check

What does this print?

```
prices = {'apple': 400, 'orange': 250}
prices['orange'] = 300
print(prices['orange'])
```

- 250
- ✓ 300
- An error, because 'orange' is already there

> 'orange' already exists, so = replaces its value. Each key appears only once, so there's no error and no second 'orange'.

stuff = {'k3': ('time', 'year')}. How do you get 'time'?

- stuff['time']
- ✓ stuff['k3'][0]
- stuff[0]['k3']

> First the key, stuff['k3'], gives the tuple. Then position [0] in the tuple gives 'time'. 'time' is a value, not a key, and the dictionary has no position 0.

You want to store a phone book: names and their phone numbers. Which fits best?

- A list
- ✓ A dictionary

You'll look numbers up **by name**: phones['Ada']. That's exactly what a dictionary's keys are for. With a list you'd have to know at which position each person is.

Key points

- A dictionary stores **key-value pairs** in curly braces: {'apple': 400, 'orange': 250}. A colon between key and value, a comma between pairs.
- You get a value **by its key**: prices['apple']. Lists use positions; dictionaries use names.
- Values can be anything, even lists, tuples and other dictionaries. Reach inside step by step: stuff['k4']['insideKey'], stuff['k3'][1], stuff['k2'][0].upper().
- d['new_key'] = value adds a pair; the same line with an existing key changes its value. Each key appears only once.
- d.keys(), d.values() and d.items() show the keys, the values and the pairs.
- Don't name a variable list or dict, and spell keys exactly.