PYTHON

YT0744

Johan Van Bauwel
E-mail: johan.vanbauwel@thomasmore.be
Office: A114

DICTIONARIES AND STRUCTURED DATA

Like a list, a **dictionary** is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers.

Indexes for dictionaries are called **keys**, and a key with its associated value is called a **key-value pair**.

In code, a dictionary is typed with braces, {}

DICTIONARIES AND STRUCTURED DATA

Enter the following into the interactive shell:

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This **assigns a dictionary** to the myCat variable. This dictionary’s keys are 'size', 'color', and 'disposition'.
The values for these keys are 'fat', 'gray', and 'loud', respectively.
You can **access these values through their keys**:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

DICTIONARIES VS. LISTS

Unlike lists, items in dictionaries are unordered.
The first item in a list named spam would be spam[0]. But there is no “first” item in a dictionary.

While the **order of items** matters for determining whether two lists are the same, **it does not matter** in what order the key-value pairs are typed in a dictionary.

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False

>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

DICTIONARIES VS. LISTS

Because **dictionaries** are not ordered, they **can’t be sliced like lists**. Trying to access a key that does not exist in a dictionary will result in a **KeyError error message**

Though dictionaries are not ordered, the fact that you can **have arbitrary values** for the keys allows you to organize your data in powerful ways. E.g. a birthday calendar with the names as keys and the birthdays as values:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
spam['color']
KeyError: 'color'
```

DICTIONARIES VS. LISTS

```
birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

while True:
    print('Enter a name: (blank to quit)')
    name = input()
    if name == '':
        break

    if name in birthdays:
        print(birthdays[name] + ' is the birthday of ' + name)
    else:
        print('I do not have birthday information for ' + name)
        print('What is their birthday?')
        bday = input()
        birthdays[name] = bday
        print('Birthday database updated.')
```

DICTIONARIES VS. LISTS

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

All the data you enter in this program is forgotten when the program terminates.
You'll learn how to save data to files on the hard drive in Chapter 8.

THE KEYS(), VALUES(), AND ITEMS() METHODS

There are three **dictionary methods** that will return list-like values of the dictionary’s keys, values, or both keys and values:

- keys()
- values()
- items()

The values returned by these methods are **not true lists**:
They **cannot be Modified.**

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

THE KEYS(), VALUES(), AND ITEMS() METHODS

Here, a **for loop** iterates over each of the values in the spam dictionary.
A for loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():
            print(k)
color
Age
```

```
>>> for i in spam.items():
           print(i)

('color', 'red')
('age', 42)
```

$\longleftarrow$ Tuples!

THE KEYS(), VALUES(), AND ITEMS() METHODS

If you want a **true list** from one of these methods, pass its list-like return value to the list() function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
        print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age']. You can also use the **multiple assignment trick** in a for loop to assign the key and value to separate variables.

CHECK WHETHER A KEY OR VALUE EXISTS

Recall from the previous chapter that the *in* and *not in* operators can check whether a **value exists** in a list.
Enter the following into the interactive shell:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

THE GET() METHOD

It’s tedious to check whether a key exists in a dictionary before accessing that key’s value.

Dictionaries have a **get() method** that takes two arguments:
- the *key* of the value to retrieve
- a *fallback value* to return if that key does not exist.

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no 'eggs' key in the picnicItems dictionary, the **default value 0** is returned by the get() method. Without using get(), the code would have caused an error message. (KeyError)

THE SETDEFAULT() METHOD

Set a value in a dictionary for a certain key **only if that key does not already have a value.**

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}

>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

- **The method returns the value 'black**' **because this is now the value** set for the key 'color'.
- When spam.setdefault('color', 'white') is called next, the value for that key is not changed to 'white' because spam already has a key named 'color'

THE SETDEFAULT() METHOD

The setdefault() method is a **nice shortcut to ensure that a key exists**.

Here is a short program that counts the number of occurrences of each letter in a string.
Open the file editor window and enter the following code,
saving it as characterCount.py

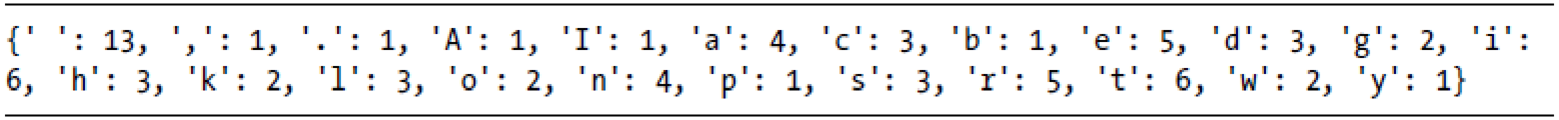
```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

THE SETDEFAULT() METHOD

The program loops over each character in the message variable's string, counting how often each character appears.

The setdefault() method call
ensures that the key is in the count dictionary (with a default value of 0) so the program doesn't throw a KeyError error when
count[character] += 1 is executed.

When you run this program, the output will look like this:



```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

ADDING KEYS & VALUES

Direct adding of new value

```
d = {‘apple':‘green'}
print(d)                    Output: {‘apple':‘green'}
d[‘lemon'] = ‘yellow’
print(d)                    Output: {‘apple':‘green‘, ‘lemon’:’yellow’}
```

Using **update()** method to add key-values pairs into dictionary

```
d = {‘apple':‘green'}
print(d)                    Output: {‘apple':‘green'}
d.update({'lemon':'yellow'})
print(d)                    Output: {‘apple':‘green‘, ‘lemon’:’yellow’}
```

REMOVE KEY/VALUE

Remove a key/value by key using **del**:

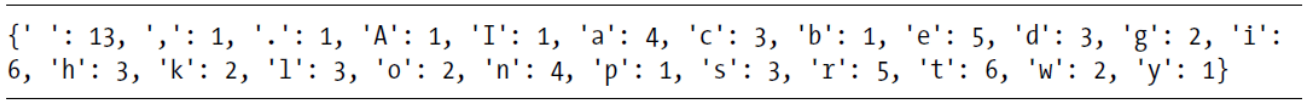
```
d= {'apple':1,'beer':2,'chips':3}
print(d)                    Output:{'apple':1,'beer':2,'chips':3}
if ‘beer' in d:
        del d[‘beer']
print(d)                    Output:{'apple':1,'chips':3}
```

PRETTY PRINTING

If you import the **pprint module** into your programs, you’ll have access to the **pprint()** and **pformat() functions** that will “pretty print” a dictionary’s values.

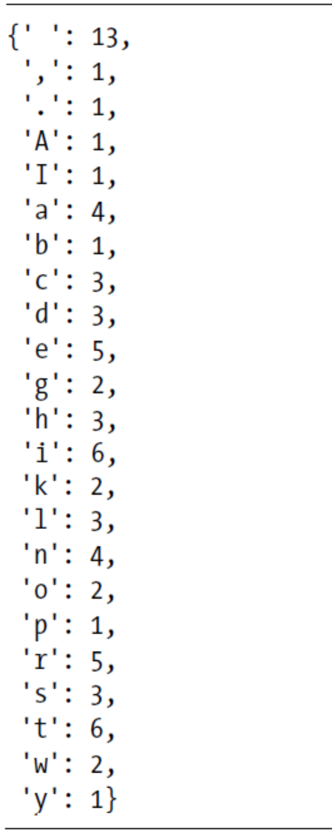
Example:

```
for character in message:
        count.setdefault(character, 0)
        count[character] = count[character] + 1
print(count)
```



```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

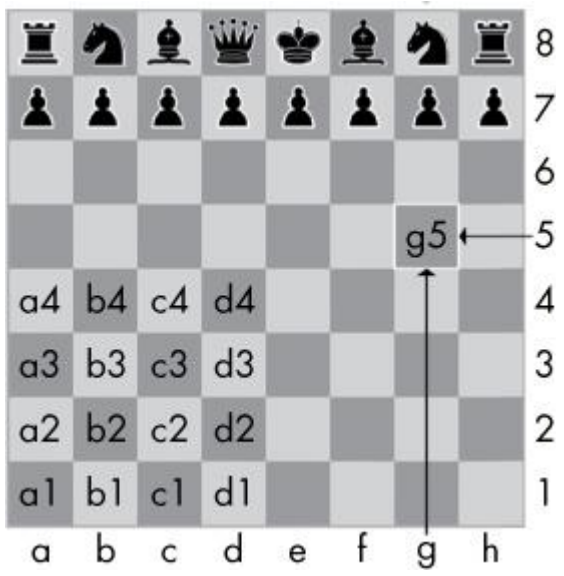
```
import pprint
for character in message:
        count.setdefault(character, 0)
        count[character] = count[character] + 1
pprint.pprint(count)
```



```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

REAL-WORLD THINGS

Even before the Internet, it was possible to play a game of chess with someone on the other side of the world.
Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move.
To do this, the players needed a way to unambiguously describe the state of the board and their moves.
In algebraic chess notation, the spaces on the chessboard are identified by a number and letter coordinate, as in the figure.



you can use it to unambiguously describe a game of chess without needing to be in front of a chessboard. In fact, you don’t even need a physical chess set if you have a good memory.

Computers have good memories. A program on a modern computer can easily store billions of strings like '2. Nf3 Nc6'. This is how computers can play chess without having a physical chessboard.

PRACTICE QUESTIONS

1. What does the code for an empty dictionary look like?

2. What does a dictionary value with a key 'foo' and a value 42 look like?

3. What is the main difference between a dictionary and a list?

4. What happens if you try to access spam['foo'] if spam is {'bar': 100}?

PRACTICE QUESTIONS

5. If a dictionary is stored in spam, what is the difference between the expressions ***'cat' in spam*** and ***'cat' in spam.keys()***?

6. If a dictionary is stored in spam, what is the difference between the expressions ***'cat' in spam*** and ***'cat' in spam.values()***?

7. What is a shortcut for the following code?

```
if 'color' not in spam:
    spam['color'] = 'black'
```

8. What module and function can be used to “pretty print” dictionary values?

Q & A

- Questions?