Editor

Dictionaries

Section 21.12

Completed

< back to module

Skip straight to editor/terminal

A dictionary in Python is similar to a list in that it's a collection of "things" we can store together. Where it differs from a list is the format in which that information is stored and retrieved.

If we wanted to represent Agent Q's closed and open case statistics, in a list we'd probably store it like this:

```
case_stats = [21, 12, 8]

print('Total cases: ' + str(case_stats[0]))
print('Solved cases: ' + str(case_stats[1]))
print('Unsolved cases: ' + str(case_stats[2]))
```

This code generates the following output:

```
Total cases: 21
Solved cases: 12
Unsolved cases: 8
```

This seems straightforward enough, but it's pretty fragile code. What if the list gets reordered or reversed? Or if a new stat gets added into the list? Then we'd have to update all our print functions that call on a specific index of our list for their data.

This is the kind of problem a dictionary is made for. It allows us to associate information together in helpful pairs called "key-value pairs", which helps in making our data more understandable as well as making it easier for us to get precisely the information we want from it without worrying about *where* it is in the list.

Using a dictionary, we'd represent the case statistics like this:

```
case_stats = {'total': 21, 'solved': 12, 'unsolved': 8}
```

Notice the difference between a list and a dictionary? The list uses square bracket notation `[]` whereas a dictionary uses curly bracket notation `{}`. The dictionary also uses a "key: value" notation: `'total'`, `'solved'`, and `'unsolved'` are all keys, and `20`, `12`, and `8` are all values.

The format for dictionaries is always the same: `key: value`. In the example above, since our key is a string, we've also wrapped our key in quotes (') while the values here are integers so they don't need quotes. Strings are always in quotes, even in dictionaries.

You can also use an integer as the key and a string as the value: this is a completely valid dictionary as well, though perhaps not as useful for us in this case (and we'll see why later in this section). But the below example won't cause an error. Keys and values are pretty flexible parts of a dictionary.

```
case_stats = {21: 'total', 12: 'solved', 8: 'unsolved'}
```

So how do we get data from our dictionary into our output?

```
case_stats = {'total': 21, 'solved': 12, 'unsolved': 8}

print('Total cases: ' + str(case_stats['total']))
print('Solved cases: ' + str(case_stats['solved']))
print('Unsolved cases: ' + str(case_stats['unsolved']))
```

Instead of using the index number to find an item in our dictionary, we tell our output what *key* to look for, and it will output the *value* it finds associated with that key.

This is great, because now it doesn't matter what order our items are stored in: as long as it has the same key, we'll always get the value we're looking for.

These kinds of data structures are used in a lot of programming languages, especially high-level ones. Sometimes they go by different names: in PHP these are called "associative arrays", in Ruby they're known as "hashes", and in Java they're known as "maps". But they all behave more or less the same, and follow a similar structure of key-value pairing in the syntax.

Lists inside dictionaries

Like with lists, we can store different types of data within a dictionary. Let's add some more information to our dictionary to describe Agent Q's case statistics and try out using a few more different data types.

```
case_stats = {'total': 21, 'solved': 12, 'unsolved': 8, 'month': 'June',
'percent_solved': 57.14, 'types': ['forensics', 'cryptography', 'web app']}

print('Month: ' + str(case_stats['month']))
print('Total cases: ' + str(case_stats['total']))
print('Solved cases: ' + str(case_stats['solved']))
print('Unsolved cases: ' + str(case_stats['unsolved']))
print('Percent solved: ' + str(case_stats['percent_solved']) + '%')
print('Types of cases: ')
print('\t' + str(case_stats['types'][0]))
print('\t' + str(case_stats['types'][1]))
print('\t' + str(case_stats['types'][2]))
```

The above code gives us the following output:

```
Month: June
Total cases: 21
Solved cases: 12
Unsolved cases: 8
Percent solved: 57.14%
Types of cases:
    forensics
    cryptography
    web app
```

Notice how we've embedded a list inside our dictionary linked to the key `types`? We can access the individual items in this list using the list index, which we've done here where we've called `case_stats['types'][0]`. This tells the computer to look for the variable called 'case_stats' which is a dictionary, and within that dictionary look for the key 'types', and within that list look for the item at index 0, which is 'forensics' in our example above.

Dictionaries inside dictionaries

Unsurprisingly, we can also put dictionaries inside our dictionaries, like so:

```
case_stats = {'month': 'June', 'stats': {'total': 21, 'solved': 12, 'unsolved': 8,
'percent_solved': 57.14}, 'types': ['forensics', 'cryptography', 'web app']}

print('Month: ' + str(case_stats['month']))
print('Total cases: ' + str(case_stats['stats']['total']))
print('Solved cases: ' + str(case_stats['stats']['solved']))
print('Unsolved cases: ' + str(case_stats['stats']['unsolved']))
print('Percent solved: ' + str(case_stats['stats']['percent_solved']) + '%')
print('Types of cases: ')
print('\t' + str(case_stats['types'][0]))
print('\t' + str(case_stats['types'][1]))
print('\t' + str(case_stats['types'][2]))
```

This code will give us the exact same output we saw in our previous example, but gives our dictionary more structure. We can start to see how powerful dictionaries can be when it comes to organizing our data. But we can also see how quickly dictionaries can get large and difficult to read. Since programming languages are for humans, let's use some formatting to make our code easier for us to read so we can see the structure of our dictionary data more clearly.

```
case_stats = {
            'month': 'June',
            'stats': {
                      'total': 21,
                      'solved': 12,
                      'unsolved': 8,
                      'percent_solved': 57.14
                      },
            'types': ['forensics', 'cryptography', 'web app']
            }

print('Month: ' + str(case_stats['month']))
print('Total cases: ' + str(case_stats['stats']['total']))
print('Solved cases: ' + str(case_stats['stats']['solved']))
print('Unsolved cases: ' + str(case_stats['stats']['unsolved']))
print('Percent solved: ' + str(case_stats['stats']['percent_solved']) + '%')
print('Types of cases: ')
print('\t' + str(case_stats['types'][0]))
print('\t' + str(case_stats['types'][1]))
print('\t' + str(case_stats['types'][2]))
```

There we go, that's a *lot* easier for us to read. Any time you're building a dictionary that's more than a few key-value pairs long, it's helpful to format it like this to better understand what you're building.

Dictionaries inside lists

We can also create a list of dictionaries. Let's create a list with some simple case stats for Agent Q, Agent S, and Agent J.

```
case_stats = [
            {'agent': 'Q', 'solved': 12, 'unsolved': 8},
            {'agent': 'S', 'solved': 15, 'unsolved': 5},
            {'agent': 'J', 'solved': 8, 'unsolved': 3}
]

print('Agent Q:')
print('\t Solved cases:' + str(case_stats[0]['solved']))
print('\t Unsolved cases:' + str(case_stats[0]['unsolved']))

print('Agent S:')
print('\t Solved cases:' + str(case_stats[1]['solved']))
print('\t Unsolved cases:' + str(case_stats[1]['unsolved']))

print('Agent J:')
print('\t Solved cases:' + str(case_stats[2]['solved']))
print('\t Unsolved cases:' + str(case_stats[2]['unsolved']))
```

This code gives us the below output:

```
Agent Q:
     Solved cases:12
     Unsolved cases:8
Agent S:
     Solved cases:15
     Unsolved cases:5
Agent J:
     Solved cases:8
     Unsolved cases:3
```

Mixing dictionaries and lists in different ways is incredibly powerful for us. For now, let's concentrate on how we can modify and build dictionaries on the fly.

Modifying dictionaries

So what happens when Agent Q solves another case? Now we need to update her stats. Let's modify our dictionary.

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))

case_stats['month'] = 'July'
case_stats['total'] = 22
case_stats['solved'] = 13

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))
```

It's a new month, so we need to update a few pieces of information in Agent Q's stats. The first is, of course, the month itself. We do that by calling on the `month` key in `case_stats` with `case_stats['month']` and giving it a new value, 'July': `case_stats['month'] = 'July'`. Then we also update 2 more pieces of information in our dictionary following the same technique: the total case number needs to go up by 1, which makes it 22, and the total number of solved cases also needs to go up by 1, which is 13. Agent Q's unsolved cases number didn't change in July, so we can leave that one as it is.

The above code will output:

```
Stats for June:
    Total cases: 21
    Solved cases: 12
    Unsolved cases: 8
Stats for July:
    Total cases: 22
    Solved cases: 13
    Unsolved cases: 8
```

There's another way to update the numbers here that will prove very useful to us later, because it means we don't have to do the math ourselves. We might as well let the computer do the arithmetic, and sometimes we don't actually know what the original number was. We just know we want to increase whatever it was by a specific value. In the example above, we want to increase Agent Q's total cases and solved cases both by 1. Let's modify our code to get the computer to do this addition for us.

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))

case_stats['month'] = 'July'
case_stats['total'] = case_stats['total'] + 1
case_stats['solved'] = case_stats['solved'] + 1

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))
```

This will give us exactly the same output as before, but notice how we've modified lines 9 and 10. In line 9 we're setting `case_stats['total']` to *itself plus 1*. The computer will first retrieve the current value of `case_stats['total']`, which is 21, then it will add 1 to that value to get 22, and finally it will set the new value of `case_stats['total']` to be 22.

There's actually a useful short-hand way of specifying exactly this kind of 'self-incrementing' behaviour so we don't have to write out `case_stats['total'] + 1`, which is a bit wordy. We can use the `+=` notation in Python to help us out here.

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))

case_stats['month'] = 'July'
case_stats['total'] += 1
case_stats['solved'] += 1

print('Stats for ' + case_stats['month'] + ':')
print('\tTotal cases: ' + str(case_stats['total']))
print('\tSolved cases: ' + str(case_stats['solved']))
print('\tUnsolved cases: ' + str(case_stats['unsolved']))
```

We've changed lines 9 and 10 to use the `+=` notation, so now instead of the longer statement `case_stats['total'] = case_stats['total'] + 1` we can use our short hand `case_stats['total'] += 1` which does exactly the same thing.

Adding things to dictionaries

Like we did with lists before, we can add things to our dictionary and even start with an empty one if we want. Let's build Agent Q's stats dictionary from the ground up. We'll build up our dictionary slowly and output it at different states so we can watch how it's built.

```
case_stats = {}

case_stats['month'] = 'June'
print(case_stats)

case_stats['total'] = 21
print(case_stats)

case_stats['solved'] = 12
print(case_stats)

case_stats['unsolved'] = 8
print(case_stats)
```

The above code will output:

```
{'month': 'June'}
{'total': 21, 'month': 'June'}
{'solved': 12, 'total': 21, 'month': 'June'}
{'solved': 12, 'unsolved': 8, 'total': 21, 'month': 'June'}
```

Adding a new key-value pair to our dictionary is exactly the same as updating an existing key-value pair. If the computer finds the key in the dictionary, it will update the value. If it doesn't, it will add the key and value to the front of the dictionary. And since order doesn't matter in dictionaries (because we access everything we need with the key) we don't mind what order things are in.

Removing things from dictionaries

If you want to remove information from a dictionary, you can use `del`:

```
case_stats = {'month': 'June', 'total': 21, 'solved': 12, 'unsolved': 8}
print(case_stats)

del case_stats['unsolved']

print(case_stats)
```

In this example, we've printed out our `case_stats` dictionary before and after we've used `del` to remove the 'unsolved' key and its corresponding value, which you can see in the output:

```
{'solved': 12, 'unsolved': 8, 'total': 21, 'month': 'June'}
{'solved': 12, 'total': 21, 'month': 'June'}
```

Pretty straightforward, but remember: once you've deleted a key-value pair from a dictionary it's gone for good!

Time Remaining: 59 mins **Language:** Python 2 | Python 3 | C **Theme:** Dark | Light **Status:** Editor Terminal Server **Server:** Reset **Log:** Download

Steps

Step 1

Create a list and dictionary structure that includes information about your own week. For each day, add information about that day, including things like:

- Whether it's a school day or weekend
- What time you wake up
- Any other regular activities you do regularly on that day, such as piano lessons or sports.
- If you're in school, for each day that is a school day add in information about what classes you have on each day.

There are a few different ways you could structure this using dictionaries and arrays, and you'll likely require both.

Step 2

Once you've got a dictionary and list structure you like, try modifying it with the methods and functions we talked about in this section. How easy is it to change things about your week? Are there any changes to the structure you want to make now that you've tried modifying it?

Editor - file: ~/output.py

```
week = {'Monday': {'weekday':'y', 'wakeup':'7:30', 'activities':'school'}}

print(week['Monday']['wakeup']
```

Save

Terminal - user: CUV7STBgao

```
CUV7STBgao@7bfebac129ee:~$
```

Completed

< back to module