Theory: Creating bytes

31 minutes 0 / 5 problems solved

In general, you can do with bytes objects most of the operations that you can perform on strings. Most probably, though, that the only thing you'll actually ever need to do with bytes is to simply create them from objects of other data types, and then convert them back. Why do `bytes` even exist then, you may ask? The thing is, they may be useful in dealing with relatively low-level applications, where binary data is needed. http and socket modules are good examples: sending and receiving data through sockets is only possible by means of the bytes data type.

Thus, we're going to learn now how to perform the conversion of data types when it all comes to `bytes`.

§1. b prefix

The most obvious way to create a bytes object is to write a string with a `b` prefix. But be careful: this works only for characters that can be encoded by a single byte, i.e. characters whose Unicode code point lies between 0 and 255 inclusively: ASCII and extended ASCII characters. A string containing characters other than these will lead to a syntax error.

```
hello = b'hello bytes'
print(hello)  # b'hello bytes'

chinese_hello = b'你好，世界'  # SyntaxError
```

What is more, in *bytes literals* only ASCII characters are permitted: that is, Latin alphabet, digits, and some basic punctuation characters. Other binary values (extended ASCII, with the Unicode code point between 127 and 255) must be entered into bytes literals using the appropriate escape sequence. Consider the example below:

```
regular = b'Unicode'
print(regular)  # b'Unicode'

strange = b'Ûñïçødê'  # SyntaxError
# if we want to have 'Ûñïçødê' in bytes, we must write:
strange2 = b'\xdb\xf1\xef\xe7\xf8d\xea'

# above, only 'd' remains the same because its Unicode code point equals 100 (whic
h is less than 127)
```

It doesn't seem very convenient, right? Let's take a look at some more flexible ways of creating bytes.

§2. bytes()

You can still convert such strings into bytes by means of other ways, though. The first of them is `bytes()` function, whose arguments are the string to be converted and the name of the output encoding.

```
chinese_hello = bytes('你好，世界', encoding='utf-8')

print(chinese_hello)  # b'\xe4\xbd\xa0\xe5\xa5\xbd\xef\xbc\x8c\xe4\xb8\x96\xe7\x95
\x8c'
```

You can approach the task of creating a bytes object from a different angle, namely, by using integers. If you need only characters whose Unicode code points lie between 0 and 255, the same function `bytes()` can do the job. It takes a list of integers between 0 and 255 and converts them to a bytes sequence. You don't need to specify the encoding in this case:

Current topic:

Creating bytes ...

Topic depends on:

✕ Bytes basics ...

Topic is required for:

Socket module ...

Table of contents:

```
int_to_bytes = bytes([104, 105])
print(int_to_bytes)  # b'hi'
```

However, if you put a single integer (equal to zero or greater) directly as an argument of the function `bytes()`, you won't get any errors and you'll still create a bytes object. The difference is that it'll be a string of zero bytes, and it'll have the length equal to the specified integer.

```
zero_bytes = bytes(4)
print(zero_bytes)  # b'\x00\x00\x00\x00'
```

In case you want to change the data later, use the mutable counterpart of `bytes()` — `bytearray()`. It returns a bytearray object that has most of the methods that the bytes type has.

Apart from this universal way of creating bytes objects, there are also some that are used specifically for strings or integers. More details in the next sections.

§3. encode()

The string method for creating bytes is called `encode()`. It's basically the same operation as `bytes()`, but under a different skin. The default encoding is `utf-8`, but you can specify it, if needed.

```
chinese_hello = '你好，世界'.encode()
chinese_hello_enc = '你好，世界'.encode('utf-8')
print(chinese_hello == chinese_hello_enc)  # True
```

Now that we've learned to convert strings to bytes, let's see what we can do with integers.

§4. to_bytes()

As for conversion of any separate integer of every possible value, the integer method `to_bytes()` can be applied. Two arguments, apart from the integer itself, are required: the number of bytes to be used for representing the integer, and the **byteorder**, either `'little'` or `'big'`, specifying the order in which the bytes should be printed: from the least significant byte to the most significant one in case of `'little'`, vice versa for `'big'`. If the given integer represents an ASCII character, it will be shown (like in the first example below), and if not, an encoded sequence representing the character will be printed (examples two and three).

```
first_number = (100).to_bytes(1, byteorder='little')
print(first_number)  # b'd'

second_number = (1024).to_bytes(2, byteorder='little')
print(second_number)  # b'\x00\x04'

third_number = (1024).to_bytes(2, byteorder='big')
print(third_number)  # b'\x04\x00'
```

> Don't forget to put the integer you are converting in parentheses. The absence of parentheses will cause a syntax error.

Alright, now that we've learned how to create bytes, it's high time to learn how to "undo" them into more familiar data types!

§5. decode()

There are reverse procedures for converting bytes to strings and integers. For strings, these are `str()` with an obligatory argument specifying encoding, and method `decode()` (as with `encode()`, the default encoding here is `utf-8`).

```
bye_bytes = b'bye bytes'
hello_str = str(bye_bytes, encoding='utf-8')
hello_another_str = bye_bytes.decode()
print(hello_str == hello_another_str)  # True
```

Don't forget that this method only works for strings. There's something different for integers.

§6. from_bytes()

This all is a bit more tricky with integers, since the method `from_bytes()` should be called in a following fashion (only `byteorder` argument is required):

```
int_to_bytes = (1024).to_bytes(2, 'little')
print(int_to_bytes)  # b'\x00\x04'

bytes_to_int = int.from_bytes(int_to_bytes, 'little')
print(bytes_to_int)  # 1024
```

So, make sure to use `int` class to call this method.

§7. Conclusions

Let's sum up what we have learned in the topic:

- `bytes()` can be used both for converting strings and lists of integers to bytes, or creating strings of zero bytes
- a byte string can be simply created by means of writing it with `b` prefix
- `encode()` is for converting strings, `to_bytes()` serves for converting integers
- `decode()` can be applied for converting bytes to strings
- `from_bytes()` helps you to convert a bytes object to an integer.

Report a typo

144 users liked this theory. **25** didn't like it. **What about you?**

😍 🙂 😐 ☹️ 😡

Start practicing

Comments (10) Hints (0) Useful links (0) Show discussion