Theory: Bytes basics

1973 users solved this topic. Latest completion was about 2 hours ago.

22 minutes 0 / 5 problems solved

Skip this topic Start practicing

As we already know, a `str` object is a sequence of characters. More than that, these are not just "some" characters: these are Unicode characters. Now it's time to introduce `bytes`, a counterpart of the Python's string. As such, `bytes` data type was introduced in Python 3, and it has a twofold nature: it represents usual strings, easily understandable by a human reader, in the form that makes them more suitable for processing by the computer.

§1. Functions chr() and ord()

To demonstrate the sophisticated nature of this data type, allow us to start from afar and introduce two Python functions: `ord()` and `chr()`. These are inverse functions that serve for converting Unicode characters to their respective code points — and vice versa. More precisely, `ord()` accepts as an argument a string consisting of a single Unicode character, and returns an integer equal to the code point assigned to this character in the Unicode; `chr()` in a similar manner allows you to convert an integer to the corresponding Unicode character.

In the following example you can see that 'a' symbol has a Unicode code point equal to 97:

```
a_character = 'a'
a_code = ord(a_character)
print(a_code)
# The output is 97
a_character = chr(a_code)
print(a_character)
# The output is 'a'
```

By the way, `ord()` works behind the scenes when you perform a **comparison** of two strings! Let's find out more in the next section.

§2. String comparison

In reality, that's the code points of these strings' characters that are being compared one by one. If the code point of the first character in the first string is bigger than the code point of the first character in the second string, then the first string is bigger than the second one.

```
a_character = 'a'
one_character = '1'
print(ord(a_character))
# The output is 97
print(ord(one_character))
# The output is 49
print(a_character > one_character)
# The output is True
```

If the code points of the first characters are equal, the same comparison is performed between the second characters of each string. This process continues until some respective characters are found not to be equal, or until one (or both) of the strings ends.

Current topic:

Bytes basics

Topic depends on:

✕ Unicode

✓ Comparisons Stage 1 16★

Topic is required for:

Creating bytes

Table of contents:

↑ Bytes basics

§1. Functions chr() and ord()

§2. String comparison

§3. Getting familiar with bytes

§4. Conclusions

Feedback & Comments

```
a_character = 'a'
print(ord(a_character))
# The output is 97
b_character = 'b'
print(ord(b_character))
# The output is 98
aa_line = a_character * 2
# 'aa'
ab_line = a_character + b_character

# 'ab'

print(aa_line < ab_line)

# The output is True

# because first characters of the strings are equal

# and the code point of second character of ab_line

# is bigger than that of aa_line
```

As you can see now, there's an easy way of representing string characters as numbers in Python. This is quite important since the computer doesn't know anything about Unicode and letters and symbols used by humans: to store and process information, a computer needs it to be in numeric form. The same applies to the information sent from one computer to another.

Therefore, Python has a special data type whose purpose is to represent strings as a sequence of numbers, and this data type is called `bytes`. Now we are ready to take a closer look!

§3. Getting familiar with bytes

Any `bytes` object is a sequence of integers representing single bytes. Each of these integers has a value **between 0 and 255** (including 0 and 255), so there are 256 possible values, which is not arbitrary: each byte consists of 8 bits, and 2^8 is equal to 256. `bytes` objects, as well as strings, are **immutable**.

What's so peculiar about bytes? Firstly, when printed, they look like a string in quotes with a prefix `b`. This prefix is not a part of the actual data stored in the variable: it simply designates a bytes object. All the information contained in our bytes object is printed between quotes.

```
first_bytes = b'123'
print(first_bytes)
# The output is b'123'
print(len(first_bytes))
# The output is 3
```

Secondly, don't be misled by bytes' string-like appearance: they are only printed in the way that makes them look like usual strings, so that we, human readers, could understand more quickly what's being stored inside the object. Deep inside bytes objects are still sequences of integers, as we've stated above.

```
first_bytes = b'123'
print(first_bytes[0])
# The output is 49
print(first_bytes[1])
# The output is 50
print(first_bytes[2])
# The output is 51
```

Such behavior reflects the dual nature of `bytes`: they contain strings in the numerical form. So, they store data in a form that is suitable for the computer to read, but they are printed in a way that is more convenient for a human.

Why '1' is equal to 49, and '2' to 50? That's where `chr()` and `ord()` come in. As you remember, they convert symbols to numbers and vice versa by means of the Unicode table. So, when you create a bytes object for the

string '1', an integer returned by the `ord()` function for the string '1' is actually being stored within the object. And when you print the object, a character returned by the `chr()` function for this integer is being printed.

```
one_bytes = b'1'
one_ord = ord('1')
print(one_bytes[0])
# The output is 49
print(one_ord)
# The output is 49
```

But items of the bytes objects are not always printed as single characters. They only do so when their value is within a certain range: if the integer value is somewhere **between 32** and **126** or **equal** to any of these numbers, a corresponding Unicode character will be printed. Otherwise, a hexadecimal escape sequence will be used to represent this integer: it'll usually have the form like '\x01' or '\xdf', (i.e., backslash followed by 'x' and two other symbols) with the exception of some sequences like \t, \n, that you probably see often even without using `bytes`.

```
characters = bytes([55, 255])
print(characters)
# The output is b'7\xff'
```

You must have noticed that in the last example a bytes object was created in a way differing from the ones used before. If you wish to know more about all ways of creating bytes objects, check out the next topic named *Creating bytes.*

§4. Conclusions

- Python 3.x has immutable `bytes` data type, used for representing strings as sequences of single bytes
- `bytes` objects consist of integers with values from 0 to 255 (inclusively)
- characters of usual strings are converted to bytes by means of Unicode table that matches every character with its unique numeric code point
- `ord()` built-in function serves for converting a Unicode character to its respective code point. This function also explains the method of string comparison in Python.
- `chr()` built-in function converts an integer, representing a code point in the Unicode table, to the respective character

Report a typo

157 users liked this theory. **13** didn't like it. **What about you?**

Start practicing

Comments (8) Hints (0) Useful links (0) Show discussion