Working with Binary Data in Python

The strings in Python that we are familiar with (the `str` type) consist of a useful abstraction for representing text. In particular, Python strings represent sequences of characters (where each character represents a Unicode character).

These are a useful abstraction for displaying text, but the fact that they exist as an abstraction for the text means that they do not work as a representation for other kinds of data, for example images or audio (since text has a particular struture in its binary data that images or audio won't generally have). Ultimately, what can be stored on disk (or sent over the network) is a series of binary values (1's and 0's). Saving character strings to disk, for example, requires first converting them to raw binary data first.

In this lab, we will be working with binary data in a variety of formats (including not only text, but also images and audio), so we will need a different, lower-level abstraction that lets us work directly with binary data. Conveniently, Python offers just such an abstraction, through *bytestrings* and *bytearrays*.

1) Byte Strings

Python offers a built-in type called `bytes` that represents a sequence of raw binary data. While this does represent a sequence of 0's and 1's; the `bytes` type exposes those values to us not as individual *bits*, but rather as *bytes*, each of which contains 8 bits. As such, we can think of each byte as representing an integer between 0 and 255 (inclusive).

One way to construct bytestrings is directly from arrays of integers, for example:

```
>>> x = bytes([207, 128])
>>> y = bytes([207, 132, 32, 62, 32])
```

If we print the resulting value, we see a different Python syntax for bytestrings (which looks like a regular string, but with a `b` in front:

```
>>> print(x)
b'\xcf\x80'
>>> print(y)
b'\xcf\x84 > '
```

Note that this is not human readable like a regular Python character string. However, it is in a format that makes it easy to directly write the data to disk, or to send it across a network (which will be relevant for this lab!).

Byte strings support indexing like regular strings. When we index into a bytestring, we get the integer value associated with the byte at that index (this will be an integer in the range 0 to 255, inclusive). Similarly, looping over a bytestring gives us integer values.

```
>>> print(x[0])
207
>>> for i in y:
...     print(i)
...
207
132
32
62
32
```

2) Converting Between Character Strings and Bytestrings

As we have mentioned above, Python has two separate internal representations for strings: character strings (`str`) and byte strings (`bytes`). The conversion between these is described by an *encoding* (which specifies how each character is represented as a sequence of bytes).

In order to store a character string on disk, it must first be *encoded*; and to view a human-readable form of a string that has been read from a file on disk, it must be *decoded*.

There are many different possible encodings for string data (that is, given a seequence of abstract characters, there are a number of different ways we could choose to represent those as raw binary data). Two of the most common encodings are `ASCII`, `UTF-8`, and `UTF-16`. As we can see, encoding the same string with two different encodings gives us two different sequences of bytes:

```
>>> x = "Fußgänger"
>>> u8 = x.encode('utf-8')
>>> u8
b'Fu\xc3\x9fg\xc3\xa4nger'
>>> u16 = x.encode('utf-16')
>>> u16
b'\xff\xfeF\x00u\x00\xdf\x00g\x00\xe4\x00n\x00g\x00e\x00r\x00'
```

When decoding, we also need to know which character encoding we are using. Trying to use the wrong encoding will either produce an unexpected answer, or an error:

```
>>> u8.decode('utf-8')
'Fußgänger'
>>> u8.decode('utf-16')
Traceback (most recent call last):
  File "<pyshell#119>", line 1, in <module>
    u8.decode('utf-16')
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x72 in position 10: truncated data
```

3) Reading and Writing Binary Data From Files

When we dealt with files in previous labs, we were making the (unstated) assumption that the files we were working with contained textual data (specifically, textual data encoded using the UTF-8 encoding, which is a broadly-used standard).

Let's see what happens when we try to open a file that does not contain UTF-8-encoded text:

```
>>> with open('Havana.mp3', 'r') as f:
...     x = f.read()
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "/usr/local/lib/python3.7/codecs.py", line 322, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 45: invalid start byte
```

Indeed, we get an exception saying that Python does not know how to convert the data in that file into a human-readable character string! In order to grab the data from the file, we need to tell Python to open the file in *binary read* mode by specifying `'rb'` when opening the file:

```
>>> with open('Havana.mp3', 'rb') as f:
...     x = f.read()
...
```

Having done this, we have the raw binary data available to us in the variable `x` (as a bytestring).

```
>>> len(x)
3471822
>>> x[:100]
b'ID3\x04\x00\x00\x00\x00\x00#TSSE\x00\x00\x00\x0f\x00\x00\x03Lavf57.56.101\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff\xfbP\x00\x
 r\x004\xf9\xa1\x00\x02\x05'
>>> for i in x[:10]:
...     print(i)
...
73
68
51
4
0
0
0
0
```

```
0
35
```

4) Representing Integers with Bytestrings

As we mentioned above, each byte in a bytestring has a representation as an integer in the range 0-255 (inclusive). A few different representations for each possible byte are shown in the table on this page for reference, though it is certainly not necessary to commit that table to memory.

The question might arise, though: what do we do if we want to represent a number bigger than 255? Clearly, one byte isn't enough in that case! So we need a different convention for representing larger integers.

In general, we can use the concatenation of multiple bytes to represent larger integers. In this lab, we choose a "big-endian" representation, where the most-significant byte comes first. By this, we mean that, if we have a byte string containing bytes b_0, b_1, b_2, and b_3 (in that order), this represents an integer:

$$(256^3 \times b_0) + (256^2 \times b_1) + (256 \times b_2) + b_3$$

In theory, we could extend this to an arbitrary number of bytes to represent arbitrarily-large integers. In this lab, though, we chose to represent lengths using 4 bytes only (though this still allows us to represent integers up to 4,294,967,295).

Since we can index into bytestrings to get integer values, it is possible to compute this value directly without a complicated algorithm. Alternatively, you can look up the documentation for the `int.to_bytes` and `int.from_bytes`, which will handle this conversion (in either direction).

5) Byte Arrays

It is worth noting also that the `bytes` type, like regular Python strings, is *immutable*. On occasion (depending on the operation you want to perform), you may wish to work with a *mutable* array of bytes (such that you can mutate it by changing a character in place, by appending/extending values, etc).

To this end, Python gives us the `bytearray` type. You can create a `bytearray` from a bytestring with `bytearray(x)`, or you can create an empty `bytearray` with `bytearray()`.

```
>>> ba = bytearray()
>>> ba.extend(u8)
>>> ba.extend("übergänge".encode('utf-8'))
>>> ba
bytearray(b'Fu\xc3\x9fg\xc3\xa4nger\xc3\xbcberg\xc3\xa4nge')
```

Note that the elements in the `bytearray` need to be valid bytes (integers in the range 0 to 255, inclusive). So you can `extend` a `bytearray` by an iterable of integers. But if you want to append values (or modify values), those values need to be given as integers.

It is worth noting, also, that `bytearray` objects can, in many ways, be treated like `bytes` objects (for example, they provide a `decode` method:

```
>>> ba.decode('utf-8')
'Fußgängerübergänge'
```

But if you need to, you can always convert back to a `bytes` object:

```
>>> x2 = bytes(ba)
```

Just like lists vs tuples, the question of whether you want a byte string or a byte array is heavily dependent on the application (certain operations might only be possible on one type or the other, or certain operations might be more efficient with one type than with the other).