lec14_binary

December 3, 2018

```
In [1]: %%javascript
        IPython.notebook.events.off('checkpoint_created.Notebook');
        IPython.notebook.events.off('notebook_saved.Notebook');

<IPython.core.display.Javascript object>
```

1 Working with Binary Data in Python

The `str` type is an abstraction for representing textual data. Each character in a `str` object represents a Unicode character. It assumes a particular structure of the underlying binary data.

However, this means that the `str` type cannot be used to represent arbitrary data. For example:

```
In [ ]: with open('hanson.txt') as f:
            contents = f.read()

        print(contents)

In [ ]: with open('Havana.mp3') as f:
            contents = f.read()
```

1.1 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:

Note that `bytes` objects, in general, are 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 lab 9!).

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

        print(x)
```

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.

```
In [ ]: print(x[0])
        print()
        for i in y:
            print(i)
```

We can also open files containing binary data, by passing `'rb'` to the `open` function:

```
In [ ]: with open('Havana.mp3', 'rb') as f:
            contents = f.read()

        print(contents[:100])
        print()
        for i in range(10):
            print(contents[i])
```

We can similarly *write* binary data to a file by opening the file in `'wb'` mode.

1.2 Converting Between `bytes` and `str`

The conversion between a raw sequence of bytes (`bytes`) and a "character string" (`str`) is described by an *encoding* (which specifies how each character is represented as a sequence of bytes).

There are many different possible encodings for string data. The most commonly used encodings are `ASCII` (which represents characters as 7-bit sequences, and so can only represent a small subset of possible characters), and `UTF-8` (where each character is represented by at most four bytes, and which can represent many more characters as a result).

```
In [ ]: x = "Fugänger"
        u8 = x.encode('utf-8')
        print(u8)
        print()
        u16 = x.encode('utf-16')
        print(u16)
```

When decoding (converting from a bytestring to a character string), we need to know what encoding was used to create the binary data.

```
In [ ]: x = "Fugänger"
        u8 = x.encode('utf-8')
        print(u8)
        print()
        u16 = x.encode('utf-16')
        print(u16)
```

```
In [ ]: print(u8.decode('utf-8'))
```

1.3 Byte Arrays

`bytearray` is the *mutable* equivalent to `bytes` (much like `list` vs `tuple`). On occasion, this mutable structure might be preferable to an immutable structure (because certain operations might be faster, or might only be possible, on one type).

```
In [ ]: ba = bytearray()
        ba.extend(u8)
        print(ba)
```

```
In [ ]: ba.extend('übergänge'.encode('utf-8'))
        print(ba)
```

```
In [ ]: print(ba.decode('utf-8'))
```