Data Formats and Encodings

This section of notes is about how to use Python to work with various data formats and encodings. There are many online tools that are also useful. A nice one is CyberChef (https://gchq.github.io/CyberChef/).

Encodings

An encoding of a character is a way to represent it as a number. One of the oldest standards is ASCII. It developed as a way to represent keys from old teletype machines. Here are a few ASCII codes: A=65, B=66, C=67, a=97, b=98, 0=48, 1=49. Most common encodings agree with ASCII for characters up through around 127, but after that they vary. There are literally hundreds of different encodings in use.

Unicode is the most important attempt to come up with a standard. Unicode itself has several encodings, the most important of which is probably UTF-8. It uses 8-bits to encode the first 127 values (which coincide with ASCII), and a variable number of bytes for everything else. Most websites use UTF-8, and strings in Python 3 are UTF-8 by default. As of this writing, there are over 144,000 characters representable in UTF-8.

Encodings in Python

In Python to get the UTF-8 character code associated with a character, use the `ord` function. For instance, `ord(A)` will return 65. To go the other way, use the `chr` function. For instance, `chr(65)` is `A`.

An important object in Python is the `bytes` object. It is a way to represent data as a raw stream of bytes, which is important in networking, cryptography, and in various built-in Python functions. For instance, when you send data over a TCP connection using Python, the data needs to be encoded first. If you have a string `s` and want to turn it into a bytes object, you can convert it using `s.encode()`. This encodes it using UTF-8. You can also create a bytes object directly by appending a `b` to the front of the string before the quote. Here are some examples:

```
s = 'abc'
obj = s.encode()
obj2 = b'abc'
```

A bytes object is a little like an array of bytes, but it's special. For instance, if `s = b'ABC'`, then `s[0]` returns 65 and `list(s)` returns `[65,66,67]`.

You can also start with an array and turn it into a bytes object. For instance, `bytes([65,66,67,250])` will create a bytes object that Python displays as `b'ABC\xfa'`. When displaying bytes object, Python tries to display each individual byte as its ASCII equivalent. Not all byte values from 0 to 127 have a displayable character, and nothing greater than 127 has a standard ASCII value, so Python displays those with the notation `\x**`, where the two stars are the byte's value in hex. For instance, the 250 byte above is displayed as `\xfa` since 250 is fa in hex.

To turn a bytes object into an ordinary string, use the `decode` method. For example, if `s = b'abc'`, then `s.decode()` will return the ordinary string `'123'`.

As mentioned, there are hundreds of different encodings. If you need to work with them, the `encode` and `decode` methods take parameters to indicate the encoding type. Python's `open` method for opening files also takes a parameter to specify the encoding if you need to open a file that was encoded in one of the more unusual encodings. Here are a few examples:

```
s = 'abc'
s.encode('ascii')
s.encode('iso_8859-1')
file = open('somefilename.txt', encoding='iso_8859-1')
```

Encodings usually aren't a problem until they are. For instance, some of the files I edit are stored in ASCII format. When I copy text in from a Word document, Word doesn't use a the standard apostrophe character `'`. It uses a slanted one with character code 8217. If I forget that and don't change it, that character ends up getting interpreted in weird ways by other programs, like web browsers, and the output ends up looking funny.

Number systems

In computers, you will often need to work with data in binary or hex format, and, on rare occasions, data in octal format. Binary is a base 2 number system, using just the digits 0 and 1. Hexadecimal is a base 16 system using the numbers 0-9 as well as the letters a-f as its 16 digits. Octal uses digits 0-7. Here are some quick examples:

```
a = hex(x) # convert decimal integer x into hexadecimal
a = oct(x) # convert decimal integer x into octal
a = bin(x) # convert decimal integer x into binary
```

To indicate a number is a hexadecimal number, precede it by `0x`. For instance, `a = 0x45fa` will cause `a` to hold the hexadecimal value 45fa. When printed, Python will show the decimal value, which is 17914. For binary numbers, precede them by `0b`, like `0b1010`. For octal, use `0o`. To convert a string holding a number from a base into decimal, use commands like below.

```
int('4a89', 16) # convert from hex to decimal
int('1101', 2)  # convert from binary to decimal
```

You could also just enter in `0x4a89` and `0b1101` and press enter at the Python shell in order to do the conversion.

Base 64

Base 64 is a commonly used encoding. It uses the digits 0-9, uppercase letters A-Z, lowercase letters a-z, the `+` symbol, and the `/` symbol as its 64 digits. You will also sometimes see either `=` or `==` at the end of a base 64 encoded value. Base 64 is widely used on the internet.

In Python, import the `base64` and use the `b64encode` and `b64decode` functions. Note that when encoding, the string must first be encoded before using base 64. Here are some examples:

```
a = b64encode(b'this is a test')
s = 'this is a test'
b = b64encode(s.encode())
```

Python bytes object and hex

A Python `bytes` object, as we've seen, acts like an array of bytes. It's sometimes handy to convert those bytes into a stream of hex characters. Use the `hex` method of the `bytes` object to do that. For instance, `b'hello'.hex()` will return the string of hex digits `'68656c6c6f'`.

To go the other way, use `bytes.fromhex`. For instance, `bytes.fromhex('68656c6c6f')` will convert the hex string from the previous paragraph into a bytes object.