Byte Histogram in Python 3

Justus Perlwitz

2015-11-30

Contents

I had this curious thought the other day: what is the byte value distribution in binary files, such as an executable? Take for example `/bin/echo` on OS X 10.11.1.

What we want to find out more precisely is: how likely will a certain byte value appear in a file. And how does it look like when we visualize that distribution?

This is going to be a field day thanks to a few handy classes and methods in the Python 3 standard library. Let's get started!

1 Reading the file

While reading binary files, we need to make sure that we open a file in binary mode. This can be accomplished by passing the `"rb"` (open for reading and enable binary mode) flag to the open function.

This will keep the Python runtime from decoding the contents of the file and instead allows us to access the raw bytes. Otherwise we would get a UTF-8 decoding error, as our binary file `/bin/echo` contains bytes that cannot be decoded into valid UTF-8 characters.

This is one of the biggest changes in Python 3, compared to Python 2. The way byte strings and unicode strings are being handled can be a challenge when porting code from Python 2 to 3, but makes things a lot easier once it's done. Bytestrings, which are either of type `bytes` or `bytearray` in Python 3, have a different intent than UTF-8 strings, which are of type `str`. While bytestrings are, as their name says, for handling binary data, such as executables, serialized files and more, UTF-8 strings are for handling natural languages. This could be an HTML template using German with lots of umlauts, or a regular expression looking through texts written in Japanese.

We also make use of the `open()` context manager, which is a good habit for several reasons. It allows open files and their file descriptors to be garbage collected immediately after leaving the context manager and it makes exception handling easier, as the context manager closes file descriptors even in the presence of exceptions and passes the exceptions on after all resources have been freed.

Note also that byte-wise access

```
with open("/bin/echo", "rb") as fd:
    contents = fd.read()
```

So much thought has gone into just two lines of Python functionality. That's what I like about the language. It is well designed for these purposes.

2 Counting bytes

Now, we are going to make use of the `Counter` class in the `collections` module. The `Counter` class allows us to easily count the occurrence of individual elements in a iterable object, such as the byte contents of a file. Let's get going.

```
from collections import Counter
c = Counter(contents)
```

What we have now is a `Counter` instance, that stores the absolute frequency of the same items over the whole `contents` bytes list. The `Counter` class supports basic operations like retrieving the most common elements or subtracting one `Counter` instance from another. The most common byte can be retrieved like so

```
for value, frequency in c.most_common(10):
    print("0x{:02x}: {}".format(value, frequency))
```

```
0x00: 12309
0x01: 215
0x30: 149
0x65: 134
0x74: 130
0x20: 127
0x69: 108
0x03: 100
0x72: 97
0x06: 94
```

It's interesting to see that the `0x00` byte is the most frequent value. I am not knowledgeable on the topic of executable file formats, but I suspect it is due to the padding of the file's segments.

3 Getting to the probability distribution

What we want to get to now is not only the absolute distribution, i.e., how many times a certain byte value occurs, but the probability distribution. With the probability distribution we know instead what the likelihood is that the next byte we read from `/bin/echo` has a certain value.

In order to get to a probability distribution, we need to do some calculations. Therefore, I went ahead and created a helper method, that uses a `Counter` object and the length of the file contents to adjust every byte value count by the number of bytes in the file. Inside of the helper method there is a generator function that loops over a `Counter` instance and applies the calculation.

As you will see we need to wrap the generator inside of a dict, so that the `Counter` does not count the frequency of value-frequency tuples, which are all going to be unique, but instead it will apply a dict to itself and allow us to use all the additional functionality that the `Counter` class offers.

```
def probability_distribution(content):
    def _helper():
        absolute_distribution = Counter(content)
        length = len(content)
        for value, frequency in absolute_distribution.items():
            yield int(value), float(frequency) / length
    return Counter(dict(_helper()))

c = probability_distribution(contents)
```

Now, if we want to see the most common elements, we can do the same as before:

```
print("Probability Distribution")
for value, frequency in c.most_common(10):
    print("0x{:02x}: {:.04f}".format(value, frequency))
```

```
Probability Distribution
0x00: 0.6826
0x01: 0.0119
0x30: 0.0083
0x65: 0.0074
```

```
0x74: 0.0072
0x20: 0.0070
0x69: 0.0060
0x03: 0.0055
0x72: 0.0054
0x06: 0.0052
```

The advantage with this way of displaying a value distribution is that we can compare it to other files more easily compared to the absolute distribution as file lengths can vary, but the distribution of certain values might not. This could for example apply to padding bytes.

For example, if we run our script on `/bin/pwd` instead, we will have the following probability distribution:

```
Probability Distribution
0x00: 0.6911
0x01: 0.0120
0x30: 0.0081
0x74: 0.0078
0x65: 0.0074
0x20: 0.0072
0x5f: 0.0060
0x69: 0.0058
0x70: 0.0054
0x03: 0.0053
```

We can see that the first 6 byte values appear in `/bin/echo` as well, in a slightly different ranking.

For the great finale we will visualize the distribution in the terminal.

```
max_prob = c.most_common(1)[0][1]
for value, frequency in c.most_common():
    print("0x{:02x}: {}".format(value, " " * int(frequency * 80/max_prob)))
```

```
0x00:
0x01:
0x30:
0x74:
0x65:
0x20:
0x5f:
...
```

That's not too interesting. If we ignore the 0 byte, things look a lot more interesting.

```
c = probability_distribution(list(filter(bool, contents)))

max_prob = c.most_common(1)[0][1]

for value, frequency in c.most_common():
    print("0x{:02x}: {}".format(value, " " * int(frequency * 80/max_prob)))
```

```
0x01:
0x30:
0x74:
0x65:
0x20:
0x5f:
0x69:
0x70:
0x03:
```

```
0x06:
0x72:
0xff:
0x04:
0x63:
0x02:
0x61:
0x6e:
0x6f:
0x31:
...
```

That looks really interesting! Note that in both cases we scaled the maximum probability, so that the value with the maximum probability will be shown with 80 bar characters.