Understanding Data

Unicode Character Encoding with Python

Bases in Python

The int() constructor can be used to create integers from different bases.

```
In [1]: int('11')
Out [1]: 11
In [2]: int('11', base=10)
Out [2]: 11
In [3]: int('11', base=2)
Out [3]: 3
In [4]: int('11', base=8)
Out [4]: 9
In [5]: a = int('11', base=16)
In [6]: print(a + 3)
20
```

Bases in Python

Alternatively, we can use binary, octal or hexadecimal **integer literal** to represent integers of different bases.

```
In [1]: 0b11 # binary literal
Out [1]: 3
In [2]: 0o11 # octal literal
Out [2]: 9
In [2]: 0x11 # hexadecimal literal
Out [2]: 17
In [2]: a = 0x11 + 0b11
In [2]: print(a)
20
```

Unicode

Unicode is a 2-column database table or map that assigns every character(like "a", "¢", or even "℔") a positive integer or **code point**. It has a total of 1,114,112 characters.

In fact, ASCII is a perfect subset of Unicode. The first 128 characters in the Unicode table correspond precisely to the ASCII characters.

Unicode, however, is **not** a character encoding. It doesn't tell you how to get actual bits from text—just code points. It doesn't tell you enough about how to convert text to binary data(0 and 1) and vice versa.

A **character encoding** specifies a set of rules to translate the integers(code points) from the Unicode table into a sequence of bits(0 and 1) that can then be processed/stored by the computer.

Unicode

Two Python functions that are useful in mapping between the Unicode character set to its code points are chr() and ord(). These functions are inverses.

chr() converts an integer code point to a single Unicode character.

```
In [1]: chr(97)
Out [1]: 'a'
In [2]: chr(128512)
Out [2]: '😀'
```

ord() converts a single Unicode character to its integer code point.

```
In [3]: ord('😀')
Out [1]: 128512
```

Unicode Encodings

From the Unicode character set, there are several ways of encoding each character into a sequence of one or more bytes.

UTF-8, UTF-16 and UTF-32 are three examples of different Unicode encodings. UTF stands for Unicode Transformation Format.

But UTF-8 by far is the most widely used encoding(95% of all webpages, most emails services and programs).

Bytes

A **bytes** object is a sequence of immutable integers of single bytes. Each byte is 8 bits so you can think of a byte as an integer with values 0 <= x < 256.

Since ASCII characters have code points in the range 0 – 127, it is common to use ASCII characters to represent bytes in this range.

For bytes in the range 128 – 255, hexadecimal digits can be used. Remember that 2 hexadecimal digits correspond exactly to one byte. The escape sequence "\x" is added to the digits to represent the 8-bit hexadecimal.

For example, "\xf3" represents the hexadecimal 0xf3(15*16+3 = 243 in decimal).

Bytes Literals

A **bytes** object is a sequence of immutable integers of single bytes. A bytes literal is similar to a string literal except a "b" prefix is append to it.

```
In [1]: a = b"Aa\xf3" # bytes literal
In [2]: a[0]
Out [2]: 65  # A = 65 in ASCII
In [3]: a[1]
Out [3]: 97 # a = 97 in ASCII
In [4]: a[2]
Out [4]: 243 # \xf3 is a single byte = 243 in decimal
In [5]: a[3]
index out of range
```

str vs bytes

Python 3's **str** type is meant to represent human-readable text and can contain any Unicode character.

```
In [1]: s1 = "résumé" # string literal
In [2]: s2 = "🤨" # string literal
In [3]: s3 = "記者 鄭啟源 羅智堅" # string literal
```

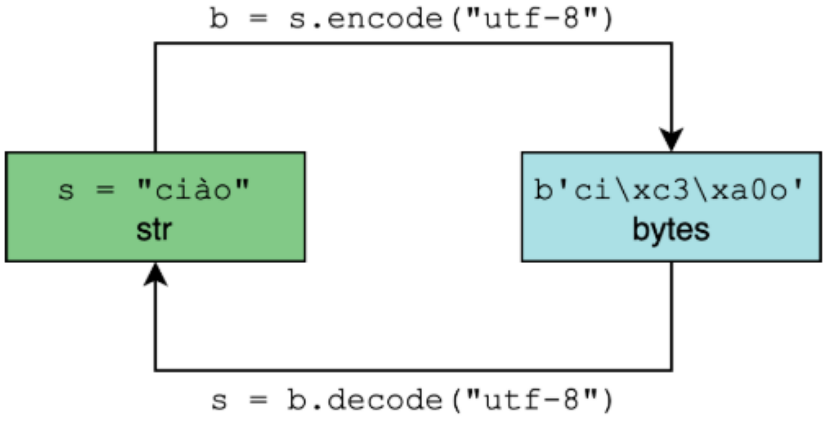
The **bytes** type, conversely, represents binary data, or sequences of raw bytes, that do not intrinsically have an encoding attached to it.

```
In [4]: a = b"r\xc3\xa9sum\xc3\xa9" # bytes literal
```

Decoding and **encoding** is the process of going between str and bytes.

Encoding/Decoding

à needs two bytes \xc3\xa0 using UTF-8 encoding.



Encoding vs decoding (Image: Real Python)

```
In [1]: s = "ciào" # string literal
In [2]: b = s.encode("utf-8") # bytes literal
In [3]: print(b)
b'ci\xc3\xa0o'
In [3]: print(b.decode("utf8")
ciào
```

Note that both "utf-8" and "utf8" work.

How to type Unicode characters

You can type Unicode characters such as β or 者 or 🤔 by using special programs, special language keyboard or copy and paste from another source.

For example, if I press "e" on my keyboard and hold it down, I get the following little window that allow me to select which letter I want:

Programmatically, Python gives you many ways of typing Unicode using escape sequences in your Python code.

How to type Unicode characters

Escape Sequence	Meaning	How To Express "a"
"\ooo"	Character with octal value ooo	"\141"
"\xhh"	Character with hex value hh	"\x61"
"\N{name}"	Character named name in the Unicode database	"\N{LATIN SMALL LETTER A}"
"\uxxxx"	Character with 16-bit (2-byte) hex value xxxx	"\u0061"
"\Uxxxxxxxx"	Character with 32-bit (4-byte) hex value xxxxxxxx	"\U00000061"

Note: Not all of the above forms work for all characters. For example, big code points may require more than 16-bit(2-byte) so "\xhh" and "\uxxxx" may not be enough. But "\Uxxxxxxxx" will work with ALL Unicode characters by using zero padding.

Example

1) Google the symbol "crossed swords".

Unicode Character "⚔" (U+2694)

Since 2694 used 4 hex digits. Prefix "\u" to get escape sequence "\u2694".

```
In [1]: "\u2694"
Out [1]: '⚔'

In [2]: 0x2694  # hex literal
Out [2]: 9876  # decimal literal equivalent

In [3]: chr(9876)  # code point to character conversion
Out [3]: '⚔'
```

Example

The same sequence of bytes may represent different characters depending on the encoding.

```
In [1]: letters = "αβγδ"

In [2]: rawdata = letters.encode("utf-8")  # bytes
In [3]: rawdata.decode("utf-8")
Out [3]: αβγδ

In [4]: rawdata.decode("utf-16")     # 😧

Out [4]: '뇎닎돎듎'
```

Questions

1) Suppose that you wrote "hi" in a text file(file.txt) and saved it on your computer using UTF8 encoding. How many bytes in memory do you need?

```
In [1]: len("hi".encode("utf8"))
Out [3]: 2
```

2) Resave "hi" using UTF-16 as file2.txt. How many bytes in memory do you need for this new file? Display the actual sequence of bytes stored in your hard drive for this file.

```
In [2]: len("hi".encode("utf16"))
Out [2]: 6
In [3]: "hi".encode("utf16")
Out [3]: b'\xff\xfeh\x00i\x00'
```

References

Unicode & Character Encodings in Python: A Painless Guide, Real Python, https://realpython.com/python-encodings-guide/

The Official Documentation for Python. https://docs.python.org/3