Unicode & Character Encodings in Python: A Painless Guide – Real Python

Real Python

Handling character encodings in Python or any other language can at times seem painful. Places such as Stack Overflow have thousands of questions stemming from confusion over exceptions like `UnicodeDecodeError` and `UnicodeEncodeError`. This tutorial is designed to clear the `Exception` fog and illustrate that working with text and binary data in Python 3 can be a smooth experience. Python's Unicode support is strong and robust, but it takes some time to master.

This tutorial is different because it's not language-agnostic but instead deliberately Python-centric. You'll still get a language-agnostic primer, but you'll then dive into illustrations in Python, with text-heavy paragraphs kept to a minimum. You'll see how to use concepts of character encodings in live Python code.

By the end of this tutorial, you'll:

- Get conceptual overviews on character encodings and numbering systems
- Understand how encoding comes into play with Python's `str` and `bytes`
- Know about support in Python for numbering systems through its various forms of `int` literals
- Be familiar with Python's built-in functions related to character encodings and numbering systems

Character encoding and numbering systems are so closely connected that they need to be covered in the same tutorial or else the treatment of either would be totally inadequate.

Note: This article is Python 3-centric. Specifically, all code examples in this tutorial were generated from a CPython 3.7.2 shell, although all minor versions of Python 3 should behave (mostly) the same in their treatment of text.

If you're still using Python 2 and are intimidated by the differences in how Python 2 and Python 3 treat text and binary data, then hopefully this tutorial will help you make the switch.

What's a Character Encoding?

There are tens if not hundreds of character encodings. The best way to start understanding what they are is to cover one of the simplest character encodings, ASCII.

Whether you're self-taught or have a formal computer science background, chances are you've seen an ASCII table once or twice. ASCII is a good place to start learning about character encoding because it is a small and contained encoding. (Too small, as it turns out.)

It encompasses the following:

- **Lowercase English letters**: *a* through *z*
- **Uppercase English letters**: *A* through *Z*
- **Some punctuation and symbols**: `"$"` and `"!"`, to name a couple
- **Whitespace characters**: an actual space (`" "`), as well as a newline, carriage return, horizontal tab, vertical tab, and a few others
- **Some non-printable characters**: characters such as backspace, `"\b"`, that can't be printed literally in the way that the letter *A* can

So what is a more formal definition of a character encoding?

At a very high level, it's a way of translating characters (such as letters, punctuation, symbols, whitespace, and control characters) to integers and ultimately to bits. Each character can be encoded to a unique sequence of bits. Don't worry if you're shaky on the concept of bits, because we'll get to them shortly.

The various categories outlined represent groups of characters. Each single character has a corresponding **code point**, which you can think of as just an integer. Characters are segmented into different ranges within the ASCII table:

Code Point Range	Class
0 through 31	Control/non-printable characters
32 through 64	Punctuation, symbols, numbers, and space
65 through 90	Uppercase English alphabet letters
91 through 96	Additional graphemes, such as `[` and `\`
97 through 122	Lowercase English alphabet letters
123 through 126	Additional graphemes, such as `{` and `|`
127	Control/non-printable character (`DEL`)

The entire ASCII table contains 128 characters. This table captures the complete **character set** that ASCII permits. If you don't see a character here, then you simply can't express it as printed text under the ASCII encoding scheme.

Code Point	Character (Name)	Code Point	Character (Name)
0	NUL (Null)	64	`@`
1	SOH (Start of Heading)	65	`A`
2	STX (Start of Text)	66	`B`
3	ETX (End of Text)	67	`C`
4	EOT (End of Transmission)	68	`D`
5	ENQ (Enquiry)	69	`E`
6	ACK (Acknowledgment)	70	`F`

Code Point	Character (Name)	Code Point	Character (Name)
7	BEL (Bell)	71	G
8	BS (Backspace)	72	H
9	HT (Horizontal Tab)	73	I
10	LF (Line Feed)	74	J
11	VT (Vertical Tab)	75	K
12	FF (Form Feed)	76	L
13	CR (Carriage Return)	77	M
14	SO (Shift Out)	78	N
15	SI (Shift In)	79	O
16	DLE (Data Link Escape)	80	P
17	DC1 (Device Control 1)	81	Q
18	DC2 (Device Control 2)	82	R
19	DC3 (Device Control 3)	83	S
20	DC4 (Device Control 4)	84	T
21	NAK (Negative Acknowledgment)	85	U
22	SYN (Synchronous Idle)	86	V
23	ETB (End of Transmission Block)	87	W
24	CAN (Cancel)	88	X
25	EM (End of Medium)	89	Y
26	SUB (Substitute)	90	Z
27	ESC (Escape)	91	[
28	FS (File Separator)	92	\
29	GS (Group Separator)	93	]
30	RS (Record Separator)	94	^
31	US (Unit Separator)	95	_
32	SP (Space)	96	`
33	!	97	a
34	"	98	b
35	#	99	c
36	$	100	d
37	%	101	e
38	&	102	f
39	'	103	g
40	(	104	h
41	)	105	i
42	*	106	j
43	+	107	k
44	,	108	l
45	-	109	m
46	.	110	n
47	/	111	o
48	0	112	p
49	1	113	q
50	2	114	r
51	3	115	s
52	4	116	t
53	5	117	u
54	6	118	v
55	7	119	w
56	8	120	x
57	9	121	y
58	:	122	z
59	;	123	{
60	<	124	|
61	=	125	}
62	>	126	~
63	?	127	DEL (delete)

The `string` Module

Python's `string` module is a convenient one-stop-shop for string constants that fall in ASCII's character set.

Here's the core of the module in all its glory:

```
# From lib/python3.7/string.py

whitespace = ' \t\n\r\v\f'
ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
```

```
ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
ascii_letters = ascii_lowercase + ascii_uppercase
digits = '0123456789'
hexdigits = digits + 'abcdef' + 'ABCDEF'
octdigits = '01234567'
punctuation = r"""!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"""
printable = digits + ascii_letters + punctuation + whitespace
```

Most of these constants should be self-documenting in their identifier name. We'll cover what `hexdigits` and `octdigits` are shortly.

You can use these constants for everyday string manipulation:

```
>>>

>>> import string

>>> s = "What's wrong with ASCII?!?!?"
>>> s.rstrip(string.punctuation)
'What's wrong with ASCII'
```

Note: `string.printable` includes all of `string.whitespace`. This disagrees slightly with another method for testing whether a character is considered printable, namely `str.isprintable()`, which will tell you that none of `{'\v', '\n', '\r', '\f', '\t'}` are considered printable.

The subtle difference is because of definition: `str.isprintable()` considers something printable if "all of its characters are considered printable in `repr()`."

A Bit of a Refresher

Now is a good time for a short refresher on the **bit**, the most fundamental unit of information that a computer knows.

A bit is a signal that has only two possible states. There are different ways of symbolically representing a bit that all mean the same thing:

- 0 or 1
- "yes" or "no"
- `True` or `False`
- "on" or "off"

Our ASCII table from the previous section uses what you and I would just call numbers (0 through 127), but what are more precisely called numbers in base 10 (decimal).

You can also express each of these base-10 numbers with a sequence of bits (base 2). Here are the binary versions of 0 through 10 in decimal:

Decimal	Binary (Compact)	Binary (Padded Form)
0	0	00000000
1	1	00000001
2	10	00000010
3	11	00000011
4	100	00000100
5	101	00000101
6	110	00000110
7	111	00000111
8	1000	00001000
9	1001	00001001
10	1010	00001010

Notice that as the decimal number *n* increases, you need more **significant bits** to represent the character set up to and including that number.

Here's a handy way to represent ASCII strings as sequences of bits in Python. Each character from the ASCII string gets pseudo-encoded into 8 bits, with spaces in between the 8-bit sequences that each represent a single character:

```
>>>

>>> def make_bitseq(s: str) -> str:
...     if not s.isascii():
...         raise ValueError("ASCII only allowed")
...     return " ".join(f"{ord(i):08b}" for i in s)

>>> make_bitseq("bits")
'01100010 01101001 01110100 01110011'

>>> make_bitseq("CAPS")
'01000011 01000001 01010000 01010011'

>>> make_bitseq("$25.43")
'00100100 00110010 00110101 00101110 00110100 00110011'

>>> make_bitseq("~5")
'01111110 00110101'
```

Note: `.isascii()` was introduced in Python 3.7.

The f-string `f"{ord(i):08b}"` uses Python's Format Specification Mini-Language, which is a way of specifying formatting for replacement

fields in format strings:

- The left side of the colon, `ord(i)`, is the actual object whose value will be formatted and inserted into the output. Using `ord()` gives you the base-10 code point for a single `str` character.
- The right hand side of the colon is the format specifier. `08` means *width 8, 0 padded*, and the `b` functions as a sign to output the resulting number in base 2 (binary).

This trick is mainly just for fun, and it will fail very badly for any character that you don't see present in the ASCII table. We'll discuss how other encodings fix this problem later on.

We Need More Bits!

There's a critically important formula that's related to the definition of a bit. Given a number of bits, *n*, the number of distinct possible values that can be represented in *n* bits is 2^n:

```
def n_possible_values(nbits: int) -> int:
    return 2 ** nbits
```

Here's what that means:

- 1 bit will let you express $2^1 == 2$ possible values.
- 8 bits will let you express $2^8 == 256$ possible values.
- 64 bits will let you express $2^{64} == 18{,}446{,}744{,}073{,}709{,}551{,}616$ possible values.

There's a corollary to this formula: given a range of distinct possible values, how can we find the number of bits, *n*, that is required for the range to be fully represented? What you're trying to solve for is *n* in the equation $2^n = x$ (where you already know *x*).

Here's what that works out to:

```
>>>

>>> from math import ceil, log

>>> def n_bits_required(nvalues: int) -> int:
...     return ceil(log(nvalues) / log(2))

>>> n_bits_required(256)
8
```

The reason that you need to use a ceiling in `n_bits_required()` is to account for values that are not clean powers of 2. Say you need to store a character set of 110 characters total. Naively, this should take `log(110) / log(2) == 6.781` bits, but there's no such thing as 0.781 bits. 110 values will require 7 bits, not 6, with the final slots being unneeded:

```
>>>

>>> n_bits_required(110)
7
```

All of this serves to to prove one concept: ASCII is, strictly speaking, a 7-bit code. The ASCII table that you saw above contains 128 code points and characters, 0 through 127 inclusive. This requires 7 bits:

```
>>>

>>> n_bits_required(128)  # 0 through 127
7
>>> n_possible_values(7)
128
```

The issue with this is that modern computers don't store much of anything in 7-bit slots. They traffic in units of 8 bits, conventionally known as a **byte**.

Note: Throughout this tutorial, I assume that a byte refers to 8 bits, as it has since the 1960s, rather than some other unit of storage. You are free to call this an *octet* if you prefer.

This means that the storage space used by ASCII is half-empty. If it's not clear why this is, think back to the decimal-to-binary table from above. You *can* express the numbers 0 and 1 with just 1 bit, or you can use 8 bits to express them as 00000000 and 00000001, respectively.

You *can* express the numbers 0 through 3 with just 2 bits, or 00 through 11, or you can use 8 bits to express them as 00000000, 00000001, 00000010, and 00000011, respectively. The highest ASCII code point, 127, requires only 7 significant bits.

Knowing this, you can see that `make_bitseq()` converts ASCII strings into a `str` representation of bytes, where every character consumes one byte:

```
>>>

>>> make_bitseq("bits")
'01100010 01101001 01110100 01110011'
```

ASCII's underutilization of the 8-bit bytes offered by modern computers led to a family of conflicting, informalized encodings that each specified additional characters to be used with the remaining 128 available code points allowed in an 8-bit character encoding scheme.

Not only did these different encodings clash with each other, but each one of them was by itself still a grossly incomplete representation of the world's characters, regardless of the fact that they made use of one additional bit.

Over the years, one character encoding mega-scheme came to rule them all. However, before we get there, let's talk for a minute about numbering systems, which are a fundamental underpinning of character encoding schemes.

Covering All the Bases: Other Number Systems

In the discussion of ASCII above, you saw that each character maps to an integer in the range 0 through 127.

This range of numbers is expressed in decimal (base 10). It's the way that you, me, and the rest of us humans are used to counting, for no reason more complicated than that we have 10 fingers.

But there are other numbering systems as well that are especially prevalent throughout the CPython source code. While the "underlying number" is the same, all numbering systems are just different ways of expressing the same number.

If I asked you what number the string `"11"` represents, you'd be right to give me a strange look before answering that it represents eleven.

However, this string representation can express different underlying numbers in different numbering systems. In addition to decimal, the alternatives include the following common numbering systems:

- **Binary**: base 2
- **Octal**: base 8
- **Hexadecimal (hex)**: base 16

But what does it mean for us to say that, in a certain numbering system, numbers are represented in base *N*?

Here is the best way that I know of to articulate what this means: it's the number of fingers that you'd count on in that system.

If you want a much fuller but still gentle introduction to numbering systems, Charles Petzold's *Code* is an incredibly cool book that explores the foundations of computer code in detail.

One way to demonstrate how different numbering systems interpret the same thing is with Python's `int()` constructor. If you pass a `str` to `int()`, Python will assume by default that the string expresses a number in base 10 unless you tell it otherwise:

```
>>>

>>> int('11')
11
>>> int('11', base=10)  # 10 is already default
11
>>> int('11', base=2)  # Binary
3
>>> int('11', base=8)  # Octal
9
>>> int('11', base=16)  # Hex
17
```

There's a more common way of telling Python that your integer is typed in a base other than 10. Python accepts **literal** forms of each of the 3 alternative numbering systems above:

Type of Literal	Prefix	Example
n/a	n/a	`11`
Binary literal	`0b` or `0B`	`0b11`
Octal literal	`0o` or `0O`	`0o11`
Hex literal	`0x` or `0X`	`0x11`

All of these are sub-forms of **integer literals**. You can see that these produce the same results, respectively, as the calls to `int()` with non-default `base` values. They're all just `int` to Python:

```
>>>

>>> 11
11
>>> 0b11  # Binary literal
3
>>> 0o11  # Octal literal
9
>>> 0x11  # Hex literal
17
```

Here's how you could type the binary, octal, and hexadecimal equivalents of the decimal numbers 0 through 20. Any of these are perfectly valid in a Python interpreter shell or source code, and all work out to be of type `int`:

Decimal	Binary	Octal	Hex
0	0b0	0o0	0x0
1	0b1	0o1	0x1
2	0b10	0o2	0x2
3	0b11	0o3	0x3
4	0b100	0o4	0x4
5	0b101	0o5	0x5
6	0b110	0o6	0x6
7	0b111	0o7	0x7
8	0b1000	0o10	0x8
9	0b1001	0o11	0x9
10	0b1010	0o12	0xa

Decimal	Binary	Octal	Hex
11	0b1011	0o13	0xb
12	0b1100	0o14	0xc
13	0b1101	0o15	0xd
14	0b1110	0o16	0xe
15	0b1111	0o17	0xf
16	0b10000	0o20	0x10
17	0b10001	0o21	0x11
18	0b10010	0o22	0x12
19	0b10011	0o23	0x13
20	0b10100	0o24	0x14

It's amazing just how prevalent these expressions are in the Python Standard Library. If you want to see for yourself, navigate to wherever your `lib/python3.7/` directory sits, and check out the use of hex literals like this:

```
$ grep -nri --include "*\.py" -e "\b0x" lib/python3.7
```

This should work on any Unix system that has `grep`. You could use `"\b0o"` to search for octal literals or "\b0b" to search for binary literals.

What's the argument for using these alternate `int` literal syntaxes? In short, it's because 2, 8, and 16 are all powers of 2, while 10 is not. These three alternate number systems occasionally offer a way for expressing values in a computer-friendly manner. For example, the number 65536 or 2^{16}, is just 10000 in hexadecimal, or `0x10000` as a Python hexadecimal literal.

Enter Unicode

As you saw, the problem with ASCII is that it's not nearly a big enough set of characters to accommodate the world's set of languages, dialects, symbols, and glyphs. (It's not even big enough for English alone.)

Unicode fundamentally serves the same purpose as ASCII, but it just encompasses a way, way, *way* bigger set of code points. There are a handful of encodings that emerged chronologically between ASCII and Unicode, but they are not really worth mentioning just yet because Unicode and one of its encoding schemes, UTF-8, has become so predominantly used.

Think of Unicode as a massive version of the ASCII table—one that has 1,114,112 possible code points. That's 0 through 1,114,111, or 0 through $17 * (2^{16}) - 1$, or `0x10ffff` hexadecimal. In fact, ASCII is a perfect subset of Unicode. The first 128 characters in the Unicode table correspond precisely to the ASCII characters that you'd reasonably expect them to.

In the interest of being technically exacting, **Unicode itself is *not* an encoding**. Rather, Unicode is **implemented** by different character encodings, which you'll see soon. Unicode is better thought of as a map (something like a `dict`) or a 2-column database table. It maps characters (like `"a"`, `"¢"`, or even `"✇"`) to distinct, positive integers. A character encoding needs to offer a bit more.

Unicode contains virtually every character that you can imagine, including additional non-printable ones too. One of my favorites is the pesky right-to-left mark, which has code point 8207 and is used in text with both left-to-right and right-to-left language scripts, such as an article containing both English and Arabic paragraphs.

Note: The world of character encodings is one of many fine-grained technical details over which some people love to nitpick about. One such detail is that only 1,111,998 of the Unicode code points are actually usable, due to a couple of archaic reasons.

Unicode vs UTF-8

It didn't take long for people to realize that all of the world's characters could not be packed into one byte each. It's evident from this that modern, more comprehensive encodings would need to use multiple bytes to encode some characters.

You also saw above that Unicode is not technically a full-blown character encoding. Why is that?

There is one thing that Unicode doesn't tell you: 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 and vice versa.

Unicode is an abstract encoding standard, not an encoding. That's where UTF-8 and other encoding schemes come into play. The Unicode standard (a map of characters to code points) defines several different encodings from its single character set.

UTF-8 as well as its lesser-used cousins, UTF-16 and UTF-32, are encoding formats for representing Unicode characters as binary data of one or more bytes per character. We'll discuss UTF-16 and UTF-32 in a moment, but UTF-8 has taken the largest share of the pie by far.

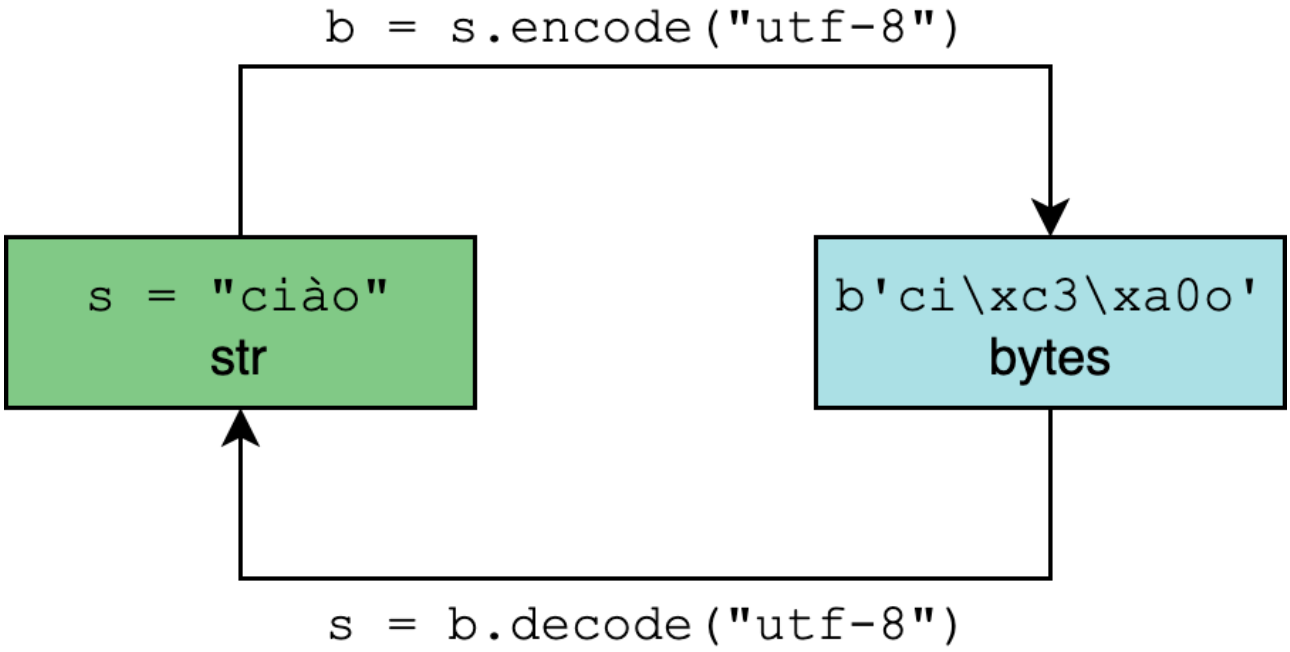
That brings us to a definition that is long overdue. What does it mean, formally, to **encode** and **decode**?

Encoding and Decoding in Python 3

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

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

Encoding and decoding is the process of going from one to the other:



Encoding vs decoding (Image: Real Python)

In `.encode()` and `.decode()`, the `encoding` parameter is `"utf-8"` by default, though it's generally safer and more unambiguous to specify it:

```
>>>

>>> "résumé".encode("utf-8")
b'r\xc3\xa9sum\xc3\xa9'
>>> "El Niño".encode("utf-8")
b'El Ni\xc3\xb1o'

>>> b"r\xc3\xa9sum\xc3\xa9".decode("utf-8")
'résumé'
>>> b"El Ni\xc3\xb1o".decode("utf-8")
'El Niño'
```

The results of `str.encode()` is a `bytes` object. Both bytes literals (such as `b"r\xc3\xa9sum\xc3\xa9"`) and the representations of bytes permit only ASCII characters.

This is why, when calling `"El Niño".encode("utf-8")`, the ASCII-compatible `"El"` is allowed to be represented as it is, but the *n* with tilde is escaped to `"\xc3\xb1"`. That messy-looking sequence represents two bytes, `0xc3` and `0xb1` in hex:

```
>>>

>>> " ".join(f"{i:08b}" for i in (0xc3, 0xb1))
'11000011 10110001'
```

That is, the character `ñ` requires two bytes for its binary representation under UTF-8.

Note: If you type `help(str.encode)`, you'll probably see a default of `encoding='utf-8'`. Be careful about excluding this and just using `"résumé".encode()`, because the default may be different in Windows prior to Python 3.6.

Python 3: All-In on Unicode

Python 3 is all-in on Unicode and UTF-8 specifically. Here's what that means:

- Python 3 source code is assumed to be UTF-8 by default. This means that you don't need `# -*- coding: UTF-8 -*-` at the top of `.py` files in Python 3.
- All text (`str`) is Unicode by default. Encoded Unicode text is represented as binary data (`bytes`). The `str` type can contain any literal Unicode character, such as `"Δv / Δt"`, all of which will be stored as Unicode.
- Anything from the Unicode character set is kosher in identifiers, meaning `résumé = "~/Documents/resume.pdf"` is valid if this strikes your fancy.
- Python's `re` module defaults to the `re.UNICODE` flag rather than `re.ASCII`. This means, for instance, that `r"\w"` matches Unicode word characters, not just ASCII letters.
- The default `encoding` in `str.encode()` and `bytes.decode()` is UTF-8.

There is one other property that is more nuanced, which is that the default `encoding` to the built-in `open()` is platform-dependent and depends on the value of `locale.getpreferredencoding()`:

```
>>>

>>> # Mac OS X High Sierra
>>> import locale
>>> locale.getpreferredencoding()
```

```
'UTF-8'

>>> # Windows Server 2012; other Windows builds may use UTF-16
>>> import locale
>>> locale.getpreferredencoding()
'cp1252'
```

Again, the lesson here is to be careful about making assumptions when it comes to the universality of UTF-8, even if it is the predominant encoding. It never hurts to be explicit in your code.

One Byte, Two Bytes, Three Bytes, Four

A crucial feature is that UTF-8 is a **variable-length encoding**. It’s tempting to gloss over what this means, but it’s worth delving into.

Think back to the section on ASCII. Everything in extended-ASCII-land demands at most one byte of space. You can quickly prove this with the following generator expression:

```
>>>

>>> all(len(chr(i).encode("ascii")) == 1 for i in range(128))
True
```

UTF-8 is quite different. A given Unicode character can occupy anywhere from one to four bytes. Here’s an example of a single Unicode character taking up four bytes:

```
>>>

>>> ibrow = "🤨"
>>> len(ibrow)
1
>>> ibrow.encode("utf-8")
b'\xf0\x9f\xa4\xa8'
>>> len(ibrow.encode("utf-8"))
4

>>> # Calling list() on a bytes object gives you
>>> # the decimal value for each byte
>>> list(b'\xf0\x9f\xa4\xa8')
[240, 159, 164, 168]
```

This is a subtle but important feature of `len()`:

- The length of a single Unicode character as a Python `str` will *always* be 1, no matter how many bytes it occupies.
- The length of the same character encoded to `bytes` will be anywhere between 1 and 4.

The table below summarizes what general types of characters fit into each byte-length bucket:

Decimal Range	Hex Range	What’s Included	Examples
0 to 127	`"\u0000"` to `"\u007F"`	U.S. ASCII	`"A"`, `"\n"`, `"7"`, `"&"`
128 to 2047	`"\u0080"` to `"\u07FF"`	Most Latinic alphabets*	`"ę"`, `"±"`, `"ƌ"`, `"ñ"`
2048 to 65535	`"\u0800"` to `"\uFFFF"`	Additional parts of the multilingual plane (BMP)**	`"ത"`, `"ᄇ"`, `"ᮈ"`, `"‰"`
65536 to 1114111	`"\U00010000"` to `"\U0010FFFF"`	Other***	`"𝕂"`, `"𐀀"`, `"😓"`, `"🂲"`,

*Such as English, Arabic, Greek, and Irish
**A huge array of languages and symbols—mostly Chinese, Japanese, and Korean by volume (also ASCII and Latin alphabets)
***Additional Chinese, Japanese, Korean, and Vietnamese characters, plus more symbols and emojis

Note: In the interest of not losing sight of the big picture, there is an additional set of technical features of UTF-8 that aren’t covered here because they are rarely visible to a Python user.

For instance, UTF-8 actually uses prefix codes that indicate the number of bytes in a sequence. This enables a decoder to tell what bytes belong together in a variable-length encoding, and lets the first byte serve as an indicator of the number of bytes in the coming sequence.

Wikipedia’s UTF-8 article does not shy away from technical detail, and there is always the official Unicode Standard for your reading enjoyment as well.

What About UTF-16 and UTF-32?

Let’s get back to two other encoding variants, UTF-16 and UTF-32.

The difference between these and UTF-8 is substantial in practice. Here’s an example of how major the difference is with a round-trip conversion:

```
>>>

>>> letters = "αβγδ"
>>> rawdata = letters.encode("utf-8")
>>> rawdata.decode("utf-8")
'αβγδ'
>>> rawdata.decode("utf-16")  # 😟
'뇎닎돎듎'
```

In this case, encoding four Greek letters with UTF-8 and then decoding back to text in UTF-16 would produce a text `str` that is in a completely different language (Korean).

Glaringly wrong results like this are possible when the same encoding isn’t used bidirectionally. Two variations of decoding the same `bytes`

object may produce results that aren't even in the same language.

This table summarizes the range or number of bytes under UTF-8, UTF-16, and UTF-32:

Encoding	Bytes Per Character (Inclusive)	Variable Length
UTF-8	1 to 4	Yes
UTF-16	2 to 4	Yes
UTF-32	4	No

One other curious aspect of the UTF family is that UTF-8 will not *always* take up less space than UTF-16. That may seem mathematically counterintuitive, but it's quite possible:

```
>>>

>>> text = "記者 鄭啟源 羅智堅"
>>> len(text.encode("utf-8"))
26
>>> len(text.encode("utf-16"))
22
```

The reason for this is that the code points in the range `U+0800` through `U+FFFF` (2048 through 65535 in decimal) take up three bytes in UTF-8 versus only two in UTF-16.

I'm not by any means recommending that you jump aboard the UTF-16 train, regardless of whether or not you operate in a language whose characters are commonly in this range. Among other reasons, one of the strong arguments for using UTF-8 is that, in the world of encoding, it's a great idea to blend in with the crowd.

Not to mention, it's 2019: computer memory is cheap, so saving 4 bytes by going out of your way to use UTF-16 is arguably not worth it.

Python's Built-In Functions

You've made it through the hard part. Time to use what you've seen thus far in Python.

Python has a group of built-in functions that relate in some way to numbering systems and character encoding:

- `ascii()`
- `bin()`
- `bytes()`
- `chr()`
- `hex()`
- `int()`
- `oct()`
- `ord()`
- `str()`

These can be logically grouped together based on their purpose:

- **`ascii()`, `bin()`, `hex()`, and `oct()`** are for obtaining a different representation of an input. Each one produces a `str`. The first, `ascii()`, produces an ASCII only representation of an object, with non-ASCII characters escaped. The remaining three give binary, hexadecimal, and octal representations of an integer, respectively. These are only *representations*, not a fundamental change in the input.
- **`bytes()`, `str()`, and `int()`** are class constructors for their respective types, `bytes`, `str`, and `int`. They each offer ways of coercing the input into the desired type. For instance, as you saw earlier, while `int(11.0)` is probably more common, you might also see `int('11', base=16)`.
- **`ord()` and `chr()`** are inverses of each other in that `ord()` converts a `str` character to its base-10 code point, while `chr()` does the opposite.

Here's a more detailed look at each of these nine functions:

Function	Signature	Accepts	Return Type	Purpose
`ascii()`	`ascii(obj)`	Varies	`str`	ASCII only representation of an object, with non-ASCII characters escaped
`bin()`	`bin(number)`	`number: int`	`str`	Binary representation of an integer, with the prefix `"0b"`

`bytes()`	`bytes(iterable_of_ints)`
	`bytes(s, enc[, errors])`
	`bytes(bytes_or_buffer)`
	`bytes([i])`
		Varies
		`i: int`
			`bytes`	Coerce (convert) the input to `bytes`, raw binary data

`chr()`	`chr(i)`	`i>=0` `i<=1114111`	`str`	Convert an integer code point to a single Unicode character
`hex()`	`hex(number)`	`number: int`	`str`	Hexadecimal representation of an integer, with the prefix `"0x"`
`int()`	`int([x])` `int(x, base=10)`	Varies	`int`	Coerce (convert) the input to `int`
`oct()`	`oct(number)`	`number: int`	`str`	Octal representation of an integer, with the prefix `"0o"`

Function	Signature	Accepts	Return Type	Purpose
`ord()`	`ord(c)`	`c: str` `len(c) == 1`	`int`	Convert a single Unicode character to its integer code point
`str()`	`str(object='')` `str(b[, enc[, errors]])`	Varies	`str`	Coerce (convert) the input to `str`, text

You can expand the section below to see some examples of each function.

`ascii()` gives you an ASCII-only representation of an object, with non-ASCII characters escaped:

```
>>>

>>> ascii("abcdefg")
"'abcdefg'"

>>> ascii("jalepeño")
"'jalepe\\xf1o'"

>>> ascii((1, 2, 3))
'(1, 2, 3)'

>>> ascii(0xc0ffee)  # Hex literal (int)
'12648430'
```

`bin()` gives you a binary representation of an integer, with the prefix `"0b"`:

```
>>>

>>> bin(0)
'0b0'

>>> bin(400)
'0b110010000'

>>> bin(0xc0ffee)  # Hex literal (int)
'0b110000001111111111101110'

>>> [bin(i) for i in [1, 2, 4, 8, 16]]  # `int` + list comprehension
['0b1', '0b10', '0b100', '0b1000', '0b10000']
```

`bytes()` coerces the input to `bytes`, representing raw binary data:

```
>>>

>>> # Iterable of ints
>>> bytes((104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100))
b'hello world'

>>> bytes(range(97, 123))  # Iterable of ints
b'abcdefghijklmnopqrstuvwxyz'

>>> bytes("real 🐊", "utf-8")  # String + encoding
b'real \xf0\x9f\x90\x8d'

>>> bytes(10)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

>>> bytes.fromhex('c0 ff ee')
b'\xc0\xff\xee'

>>> bytes.fromhex("72 65 61 6c 70 79 74 68 6f 6e")
b'realpython'
```

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

```
>>>

>>> chr(97)
'a'

>>> chr(7048)
'𛈈'

>>> chr(1114111)
'\U0010ffff'

>>> chr(0x10FFFF)  # Hex literal (int)
'\U0010ffff'

>>> chr(0b01100100)  # Binary literal (int)
'd'
```

`hex()` gives the hexadecimal representation of an integer, with the prefix `"0x"`:

```
>>>

>>> hex(100)
'0x64'

>>> [hex(i) for i in [1, 2, 4, 8, 16]]
```

```
['0x1', '0x2', '0x4', '0x8', '0x10']

>>> [hex(i) for i in range(16)]
['0x0', '0x1', '0x2', '0x3', '0x4', '0x5', '0x6', '0x7',
 '0x8', '0x9', '0xa', '0xb', '0xc', '0xd', '0xe', '0xf']
```

`int()` coerces the input to `int`, optionally interpreting the input in a given base:

```
>>>

>>> int(11.0)
11

>>> int('11')
11

>>> int('11', base=2)
3

>>> int('11', base=8)
9

>>> int('11', base=16)
17

>>> int(0xc0ffee - 1.0)
12648429

>>> int.from_bytes(b"\x0f", "little")
15

>>> int.from_bytes(b'\xc0\xff\xee', "big")
12648430
```

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

```
>>>

>>> ord("a")
97

>>> ord("ę")
281

>>> ord("")
7048

>>> [ord(i) for i in "hello world"]
[104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]
```

`str()` coerces the input to `str`, representing text:

```
>>>

>>> str("str of string")
'str of string'

>>> str(5)
'5'

>>> str([1, 2, 3, 4])  # Like [1, 2, 3, 4].__str__(), but use str()
'[1, 2, 3, 4]'

>>> str(b"\xc2\xbc cup of flour", "utf-8")
'¼ cup of flour'

>>> str(0xc0ffee)
'12648430'
```

Python String Literals: Ways to Skin a Cat

Rather than using the `str()` constructor, it's commonplace to type a `str` literally:

```
>>>

>>> meal = "shrimp and grits"
```

That may seem easy enough. But the interesting side of things is that, because Python 3 is Unicode-centric through and through, you can "type" Unicode characters that you probably won't even find on your keyboard. You can copy and paste this right into a Python 3 interpreter shell:

```
>>>

>>> alphabet = 'αβγδεζηθικλμνξοπρςστυφχψ'
>>> print(alphabet)
αβγδεζηθικλμνξοπρςστυφχψ
```

Besides placing the actual, unescaped Unicode characters in the console, there are other ways to type Unicode strings as well.

One of the densest sections of Python's documentation is the portion on lexical analysis, specifically the section on string and bytes literals. Personally, I had to read this section about one, two, or maybe nine times for it to really sink in.

Part of what it says is that there are up to six ways that Python will allow you to type the same Unicode character.

The first and most common way is to type the character itself literally, as you've already seen. The tough part with this method is finding the actual keystrokes. That's where the other methods for getting and representing characters come into play. Here's the full list:

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"`

Here's some proof and validation of the above:

```
>>>

>>> (
...     "a" ==
...     "\x61" ==
...     "\N{LATIN SMALL LETTER A}" ==
...     "\u0061" ==
...     "\U00000061"
... )
True
```

Now, there are two main caveats:

1. Not all of these forms work for all characters. The hex representation of the integer 300 is `0x012c`, which simply isn't going to fit into the 2-hex-digit escape code `"\xhh"`. The highest code point that you can squeeze into this escape sequence is `"\xff"` (`"ÿ"`). Similarly for `"\ooo"`, it will only work up to `"\777"` (`"ǿ"`).

2. For `\xhh`, `\uxxxx`, and `\Uxxxxxxxx`, exactly as many digits are required as are shown in these examples. This can throw you for a loop because of the way that Unicode tables conventionally display the codes for characters, with a leading `U+` and variable number of hex characters. They key is that Unicode tables most often do not zero-pad these codes.

For instance, if you consult unicode-table.com for information on the Gothic letter faihu (or fehu), `"𐍆"`, you'll see that it is listed as having the code `U+10346`.

How do you put this into `"\uxxxx"` or `"\Uxxxxxxxx"`? Well, you can't fit it in `"\uxxxx"` because it's a 4-byte character, and to use `"\Uxxxxxxxx"` to represent this character, you'll need to left-pad the sequence:

This also means that the `"\Uxxxxxxxx"` form is the only escape sequence that is capable of holding *any* Unicode character.

Note: Here's a short function to convert strings that look like `"U+10346"` into something Python can work with. It uses `str.zfill()`:

```
>>>

>>> def make_uchr(code: str):
...     return chr(int(code.lstrip("U+").zfill(8), 16))
>>> make_uchr("U+10346")
'𐍆'
>>> make_uchr("U+0026")
'&'
```

Other Encodings Available in Python

So far, you've seen four character encodings:

1. ASCII
2. UTF-8
3. UTF-16
4. UTF-32

There are a ton of other ones out there.

One example is Latin-1 (also called ISO-8859-1), which is technically the default for the Hypertext Transfer Protocol (HTTP), per RFC 2616. Windows has its own Latin-1 variant called cp1252.

Note: ISO-8859-1 is still very much present out in the wild. The `requests` library follows RFC 2616 "to the letter" in using it as the default encoding for the content of an HTTP/HTTPS response. If the word "text" is found in the `Content-Type` header, and no other encoding is specified, then `requests` will use ISO-8859-1.

The complete list of accepted encodings is buried way down in the documentation for the `codecs` module, which is part of Python's Standard Library.

There's one more useful recognized encoding to be aware of, which is `"unicode-escape"`. If you have a decoded `str` and want to quickly get a representation of its escaped Unicode literal, then you can specify this encoding in `.encode()`:

```
>>>

>>> alef = chr(1575)  # Or "\u0627"
>>> alef_hamza = chr(1571)  # Or "\u0623"
>>> alef, alef_hamza
('ا' ,'أ')
```

```
>>> alef.encode("unicode-escape")
b'\\u0627'
>>> alef_hamza.encode("unicode-escape")
b'\\u0623'
```

You Know What They Say About Assumptions...

Just because Python makes the assumption of UTF-8 encoding for files and code that *you* generate doesn't mean that you, the programmer, should operate with the same assumption for external data.

Let's say that again because it's a rule to live by: when you receive binary data (bytes) from a third party source, whether it be from a file or over a network, the best practice is to check that the data specifies an encoding. If it doesn't, then it's on you to ask.

All I/O happens in bytes, not text, and bytes are just ones and zeros to a computer until you tell it otherwise by informing it of an encoding.

Here's an example of where things can go wrong. You're subscribed to an API that sends you a recipe of the day, which you receive in `bytes` and have always decoded using `.decode("utf-8")` with no problem. On this particular day, part of the recipe looks like this:

```
>>>

>>> data = b"\xbc cup of flour"
```

It looks as if the recipe calls for some flour, but we don't know how much:

```
>>>

>>> data.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xbc in position 0: invalid start byte
```

Uh oh. There's that pesky `UnicodeDecodeError` that can bite you when you make assumptions about encoding. You check with the API host. Lo and behold, the data is actually sent over encoded in Latin-1:

```
>>>

>>> data.decode("latin-1")
'¼ cup of flour'
```

There we go. In Latin-1, every character fits into a single byte, whereas the "¼" character takes up two bytes in UTF-8 (`"\xc2\xbc"`).

The lesson here is that it can be dangerous to assume the encoding of any data that is handed off to you. It's *usually* UTF-8 these days, but it's the small percentage of cases where it's not that will blow things up.

If you really do need to abandon ship and guess an encoding, then have a look at the `chardet` library, which uses methodology from Mozilla to make an educated guess about ambiguously encoded text. That said, a tool like `chardet` should be your last resort, not your first.

Odds and Ends: `unicodedata`

We would be remiss not to mention `unicodedata` from the Python Standard Library, which lets you interact with and do lookups on the Unicode Character Database (UCD):

```
>>>

>>> import unicodedata

>>> unicodedata.name("€")
'EURO SIGN'
>>> unicodedata.lookup("EURO SIGN")
'€'
```

Wrapping Up

In this article, you've decoded the wide and imposing subject of character encoding in Python.

You've covered a lot of ground here:

- Fundamental concepts of character encodings and numbering systems
- Integer, binary, octal, hex, str, and bytes literals in Python
- Python's built-in functions related to character encoding and numbering systems
- Python 3's treatment of text versus binary data

Now, go forth and encode!

Resources

For even more detail about the topics covered here, check out these resources:

- **Joel Spolsky:** The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
- **David Zentgraf:** What every programmer absolutely, positively needs to know about encodings and character sets to work with text
- **Mozilla:** A composite approach to language/encoding detection
- **Wikipedia:** UTF-8

- **John Skeet:** Unicode and .NET
- **Charles Petzold:** *Code: The Hidden Language of Computer Hardware and Software*
- **Network Working Group, RFC 3629:** UTF-8, a transformation format of ISO 10646
- **Unicode Technical Standard #18:** Unicode Regular Expressions

The Python docs have two pages on the subject:

- What's New in Python 3.0
- Unicode HOWTO