Ten thousand meters

Diving deep, flying high to see why

about blog materials

Python behind the scenes #9: how Python strings work

Published: Sun 21 February 2021
By *Victor Skvortsov*
tags: Python behind the scenes Python CPython

In 1991 Guido van Rossum released the first version of the Python programming language. About that time the world began to witness a major change in how computer systems represent written language. The internalization of the Internet increased the demand to support different writing systems, and the Unicode Standard was developed to meet this demand. Unicode defined a universal character set able to represent any written language, various non-alphanumeric symbols and, eventually, emoji 😀. Python wasn't designed with Unicode in mind, but it evolved towards Unicode support during the years. The major change happened when Python got a built-in support for Unicode strings – the `unicode` type that later became the `str` type in Python 3. Python strings have been proven to be a convenient way to work with text in the Unicode age. Today we'll see how they work behind the scenes.

Note: In this post I'm referring to CPython 3.9. Some implementation details will certainly change as CPython evolves. I'll try to keep track of important changes and add update notes.

The scope of this post

This post doesn't try to cover all aspects of text encoding in relation to Python. You see, programming language designers have to make several text encoding decisions because they have to answer the following questions:

- How to talk to the external world (the encodings of command-line parameters, environment variables, standard streams and the file system).
- How to read the source code (the encoding of source files).
- How to represent text internally (the encoding of strings).

This post focuses on the last problem. But before we dive into the internals of Python strings, let's briefly discuss the problem of text encoding on a real life example and clarify what Unicode really is.

The essence of text encoding

You see this text as a sequence of characters rendered by your browser and displayed on your screen. I see this text as the same sequence of characters as I type it into my editor. In order for us to see the same thing, your browser and my editor must be able to represent the same set of characters, that is, they must agree on a **character set**. They also need to choose some, possibly different, ways to represent the text internally to be able to work with it. For example, they may choose to map each character to a unit consisting of one or more bytes and represent the text as a sequence of those units. Such a mapping is usually referred to as a **character encoding**. A character encoding is also crucial for our communication. Your browser and my web server must agree on how to **encode** text into bytes and **decode** text from bytes, since bytes is what they transmit to talk to each other.

The character set that your browser and my editor use is Unicode. Unicode is able to represent English as well as any other written language you can think of (文言, Čeština, Ελληνικά, עברית, हिन्दी), 日本語, Português, Русский) and thousands of miscellaneous symbols (£, ⅐, ↳, ∭, ⌘, , ♫, 🧑‍💻, 🍺) . My web server sends this text as a part of the HTML page in the UTF-8 encoding. You browser knows which encoding was used to encode the text because the `Content-Type` HTTP header declares the encoding:

```
Content-Type: text/html; charset=utf-8
```

Even if you save this HTML page locally, your browser will still be able to detect its encoding because the encoding is specified in the HTML itself:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <!-- ... -->
</html>
```

This may seem absurd to you. How can a browser decode the HTML to read the encoding if it doesn't know the encoding yet? This is usually not a problem in practice because the beginning of an HTML page contains only ASCII characters and most encodings used on the web encode ASCII characters in the same way. Check out the HTML standard to learn more about the algorithm that browsers use to determine the encoding.

Note that the HTTP header and the HTML metatag specify "charset", i.e. a character set. This may seem confusing since UTF-8 is not a character set. What they really specify is a character encoding. The two terms are often used interchangeably because character encodings typically imply a character set of the same name. For example, the ASCII character encoding implies the ASCII character set. The Unicode Standard fixes the terminology by giving precise definitions to all important terms. We'll study them, but before, let's discuss why and how the Unicode project began.

The road to Unicode

Before the adoption of Unicode, most computer systems used the ASCII character encoding that encodes a set of 128 characters using a 7-bit pattern to encode each character. ASCII was sufficient to deal with English texts but that's about it. Other character encodings were developed to support more languages. Most of them extended ASCII to 256 characters and used one byte to encode each character. For example, the ISO 8859 standard defined a family of 15 such character encodings. Among them were:

- Latin Western European ISO 8859-1 (German, French, Portuguese, Italian, etc.)
- Central European ISO 8859-2 (Polish, Croatian, Czech, Slovak, etc.)
- Latin/Cyrillic ISO 8859-5 (Russian, Serbian, Ukrainian, etc.)
- Latin/Arabic ISO 8859-6
- Latin/Greek ISO 8859-7.

Multi-lingual software had to handle many different character encodings. This complicated things a lot. Another problem was to choose the right encoding to decode text. Failing to do so resulted in a garbled text known as mojibake. For example, if you encode the Russian word for mojibake "кракозябры" using the KOI-8 encoding and decode it using ISO 8859-1, you'll get "ËÒÁËÏÚÑÂÒÙ".

The problems with different character encodings are not gone completely. Nevertheless, it became much more easier to write multi-lingual software nowadays. This is due to two independent initiatives that began in the late 1980s. One was ISO 10646, an international standard, and the other was Unicode, a project organized by a group of software companies. Both projects had the same goal: to replace hundreds of conflicting character encodings with a single universal one that covers all languages in widespread use. They quickly realized that having two different universal character sets wouldn't help achieve the goal, so in 1991 the Universal Coded Character Set (UCS) defined by ISO 10646 and Unicode's character set were unified. Today the projects define essentially the same character encoding model. Nevertheless, both continue to exist. The difference between them is that the Unicode Standard has a greater scope:

> *The assignment of characters is only a small fraction of what the Unicode Standard and its associated specifications provide. The specifications give programmers extensive descriptions and a vast amount of data about the handling of text, including how to:*
>
> - *divide words and break lines*
> - *sort text in different languages*
> - *format numbers, dates, times, and other elements appropriate to different locales*
> - *display text for languages whose written form flows from right to left, such as Arabic or Hebrew*
> - *display text in which the written form splits, combines, and reorders, such as for the languages of South Asia*
> - *deal with security concerns regarding the many look-alike characters from writing systems around the world*

The most important thing that we need to understand about Unicode is how it encodes characters.

Unicode basics

Unicode defines **characters** as smallest components of written language that have semantic value. This means that such units as diacritical marks are considered to be characters on their own. Multiple Unicode characters can be combined to produce what visually looks like a single character. Such combinations of characters are called **grapheme clusters** in Unicode. For example, the string "á" is a grapheme cluster that consists of two characters: the Latin letter "a" and the acute accent "´". Unicode encodes some grapheme clusters as separate characters as well, but does that solely for compatibility with legacy encodings. Due to combining characters, Unicode can represent all sorts of grapheme clusters such as "ä́" and, at the same time, keep the character set relatively simple.

Unicode characters are abstract. The standard doesn't care about the exact shape a character takes when it's rendered. The shape, called a **glyph**, is considered to be a concern of a font designer. The connection between characters and glyphs can be quite complicated. Multiple characters can merge into a single glyph. A single character can be rendered as multiple glyphs. And how characters map to glyphs can depend on the context. Check out the Unicode Technical Report #17 for examples.

Unicode doesn't map characters to bytes directly. It does the mapping in two steps:

1. The **coded character set** maps characters to code points.
2. A **character encoding form**, such as UTF-8, maps code points to sequences of code units, where each code unit is a sequence of one or more bytes.

The Unicode coded character set is what we usually mean when we say Unicode. It's the same thing as the UCS defined by ISO 10646. The word "coded" means that it's not actually a set but a mapping. This mapping assigns a code point to each character in the character set. A **code point** is just an integer in the range [0, 1114111], which is written as U+0000..U+10FFFF in the Unicode hexadecimal notation and is called a **code space**. The current Unicode 13.0 assigns code points to 143,859 characters.

Technically, the coded character set is a collection of entries. Each entry defines a character and assigns a code point to it by specifying three pieces of information:

- the code point value
- the name of the character; and

- a representative glyph.

For example, the entry for the letter "b" looks like this: (U+0062, LATIN SMALL LETTER B, b).

The standard also specifies various character properties such as whether the character is a letter, a numeral or some other symbol, whether it's written from left-to-right or from right-to-left and whether it's an uppercase letter, lowercase letter or doesn't have a case at all. All this information is contained in the Unicode Character Database. We can query this database from Python using the `unicodedata` standard module.

If we encode some text with the coded character set, what we get is a sequence of code points. Such a sequence is called a **Unicode string**. This is an appropriate level of abstraction to do text processing. Computers, however, know nothing about code points, so code points must be encoded to bytes. Unicode defines three character encoding forms to do that: UTF-8, UTF-16 and UTF-32. Each is capable of encoding the whole code space but has its own strengths and weaknesses.

UTF-32 is the most straightforward encoding form. Each code point is represented by a code unit of 32 bits. For example, the code point U+01F193 is encoded as `0x0001F193`. The main advantage of UTF-32, besides simplicity, is that it's a fixed-width encoding form, i.e. each code point corresponds to a fixed number of code units (in this case – one). This allows fast code point indexing: we can access the nth code point of a UTF-32-encoded string in constant time.

Originally, Unicode defined only one encoding form that represented each code point by a code unit of 16 bits. It was possible to encode the whole code space using this encoding form because the code space was smaller and consisted of 2^16 = 65,536 code points. Over the time, Unicode people realized that 65,536 code points were not enough to cover all written language and extended the code space to 1,114,112 code points. The problem was that new code points, which constituted the range U+010000..U+10FFFF, could not be represented by a 16-bit code unit. Unicode solved this problem by encoding each new code points with a pair of 16-bit code units, called a **surrogate pair**. Two unassigned ranges of code points were reserved to be used only in surrogate pairs: U+D800..U+DBFF for higher parts of surrogate pairs and U+DC00..U+DFFF for lower parts of surrogate pairs. Each of these ranges consists of 1024 code points, so they can be used to encode 1024 × 1024 = 1,048,576 code points. This encoding form that uses one 16-bit code unit to encode code points in the range U+0000..U+FFFF and two 16-bit code units to encode code points in the range U+010000..U+10FFFF became known as UTF-16. Its original version is a part of the ISO 10646 standard and is called UCS-2. The only difference between UTF-16 and UCS-2 is that UCS-2 doesn't support surrogate pairs and is only capable of encoding code points in the range U+0000..U+FFFF known as the Basic Multilingual Plane (BMP). The ISO 10646 standard also defines the UCS-4 encoding form, which is effectively the same thing as UTF-32.

UTF-32 and UTF-16 are widely used for representing Unicode strings in programs. They are, however, not very suitable for text storage and transmission. The first problem is that they are space-inefficient. This is especially true when a text that consists mostly of ASCII characters is encoded using the UTF-32 encoding form. The second problem is that bytes within a code unit can be arranged in a little-endian or big-endian order, so UTF-32 and UTF-16 come in two flavors each. The special code point called the byte order mark (BOM) is often added to the beginning of a text to specify the endianness. And the proper handling of BOMs adds complexity. The UTF-8 encoding form doesn't have these issues. It represents each code point by a sequence of one, two, three or four bytes. The leading bits of the first byte indicate the length of the sequence. Other bytes always have the form `0b10xxxxxx` to distinguish them from the first byte. The following table shows what sequences of each length look like and what ranges of code points they encode:

Range	Byte 1	Byte 2	Byte 3	Byte 4
U+0000..U+007F	`0b0xxxxxxx`			
U+0080..U+07FF	`0b110xxxxx`	`0b10xxxxxx`		
U+0800..U+FFFF	`0b1110xxxx`	`0b10xxxxxx`	`0b10xxxxxx`	
U+010000..U+10FFFF	`0b11110xxx`	`0b10xxxxxx`	`0b10xxxxxx`	`0b10xxxxxx`

To encode a code point, we choose an appropriate template from the table above and replace xs in it with the binary representation of a code point. An appropriate template is the shortest template that is capable of encoding the code point. The binary representation of a code point is aligned to the right, and the leading xs are replaced with 0s.

Note that UTF-8 represents all ASCII characters using just one byte, so that any ASCII-encoded text is also a UTF-8-encoded text. This feature is one of the reasons why UTF-8 gained adoption and became the most dominant encoding on the web.

This section should give us a basic idea of how Unicode works. If you want to learn more about Unicode, I really recommend reading the first few chapters of the Unicode Standard.

A brief history of Python strings

The way Python strings work today is very different from the way Python strings worked when Python was first released. This aspect of the language changed significantly multiple times. To better understand why modern Python strings work the way they do, let's take a quick look into the past.

Initially, Python had one built-in type to represent strings – the `str` type. It was not the `str` type we know today. Python strings were byte strings, that is, sequences of bytes, and worked similar to how `bytes` objects work in Python 3. This is in contrast to Python 3 strings that are Unicode strings.

Since byte strings were sequences of bytes, they were used to represent all kinds of data: sequences of ASCII characters, UTF-8-encoded texts and arbitrary arrays of bytes. Byte strings themselves didn't hold any information about the encoding. It was up to a program to interpret the values. For example, we could put a UTF-8-encoded text into a byte string, print it to the stdout and see the actual Unicode characters if the terminal encoding was UTF-8:

```
$ python2.7
>>> s = '\xe2\x9c\x85'
>>> print(s)
✅
```

Though byte strings were sequences of bytes, they were called strings for a reason. The reason is that Python provided string methods for byte strings, such as `str.split()` and `str.upper()`. Think about what the `str.upper()` method should do on a sequence of bytes. It doesn't make sense to take a byte and convert it to an uppercase variant because bytes don't have case. It starts make sense if we assume that the sequence of bytes is a text in some encoding. That's exactly what Python did. The assumed encoding depended on the current locale. Typically, it was ASCII. But we could change the locale, so that string methods began to work on non-ASCII-encoded text:

```
$ python2.7
>>> s = '\xef\xe8\xf2\xee\xed' # Russian 'питон' in the encoding windows-1251
>>> '\xef\xe8\xf2\xee\xed'.upper() # does nothing since characters are non-ascii
'\xef\xe8\xf2\xee\xed'
>>> import locale
>>> locale.setlocale(locale.LC_ALL , 'ru_RU.CP1251')
'ru_RU.CP1251'
>>> '\xef\xe8\xf2\xee\xed'.upper() # converts to uppercase
'\xcf\xc8\xd2\xce\xcd'
>>> print('\xef\xe8\xf2\xee\xed'.upper().decode('windows-1251')) # let's print it
ПИТОН
```

The implementation of this logic relied on the C standard library. It worked for 8-bit fixed-width encodings but didn't work for UTF-8 or any other Unicode encoding. In short, Python had no Unicode strings back then.

Then the `unicode` type was introduced. This happened before Python 2 when PEPs hadn't existed yet. The change was only later described in PEP 100. The instances of `unicode` were true Unicode strings, that is, sequences of code points (or, if you prefer, sequences of Unicode characters). They worked much like strings we have today:

```
$ python2.7
>>> s = u'питон' # note unicode literal
>>> s # each element is a code point
u'\u043f\u0438\u0442\u043e\u043d'
>>> s[1] # can index code points
u'\u0438'
>>> print(s.upper()) # string methods work
ПИТОН
```

Python used the UCS-2 encoding to represent Unicode strings internally. UCS-2 was capable of encoding all the code points that were assigned at that moment. But then Unicode assigned first code points outside the Basic Multilingual Plane, and UCS-2 could no longer encode all the code points. Python switched from UCS-2 to UTF-16. Now any code point outside the Basic Multilingual Plane could be represented by a surrogate pair. This caused another problem. Since UTF-16 is a variable-width encoding, getting the nth code point of a string requires scanning the string until that code point is found. Python supported indexing into a string in constant time and didn't want to lose that. So, what happened is that Unicode objects seized to be true Unicode strings and became sequence of code units. This had the following consequences:

```
$ python2.7
>>> u'hello'[4] # indexing is still supported and works fast
u'o'
>>> len(u'😀') # but length of a character outside BMP is 2
2
>>> u'😀'[1] # and indexing returns code units, not code points
u'\ude00'
```

PEP 261 tried to revive true Unicode strings. It introduced a compile-time option that enabled the UCS-4 encoding. Now Python had two distinct builds: a "narrow" build and a "wide" build. The choice of the build affected the way Unicode objects worked. UCS-4 could not replace UTF-16 altogether because of its space-inefficiency, so both had to coexist. Internally, Unicode object was represented as an array of `Py_UNICODE` elements. The `Py_UNICODE` type was set to `wchar_t` if the size of `wchar_t` was compatible with the build. Otherwise, it was set to either `unsigned short` (UTF-16) or `unsigned long` (UCS-4).

In the meantime, Python developers focused their attention on another source of confusion: the coexistence of byte strings and Unicode strings. There were several problems with this. For example, it was possible to mix two types:

```
>>> "I'm str" + u" and I'm unicode"
u"I'm str and I'm unicode"
```

Unless it wasn't:

```
>>> "I'm str \x80" + u" and I'm unicode"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 8: ordinal not in range(128)
```

The famous Python 3.0 release renamed the `unicode` type to the `str` type and replaced the old `str` type with the `bytes` type. The essence of this change is summarized in the release notes:

> *The biggest difference with the 2.x situation is that any attempt to mix text and data in Python 3.0 raises `TypeError`, whereas if you were to mix Unicode and 8-bit strings in Python 2.x, it would work if the 8-bit string happened to contain only 7-bit (ASCII) bytes, but you would get `UnicodeDecodeError` if it contained non-ASCII values. This value-specific behavior has caused numerous sad faces over the years.*

Python strings became the Python strings we know today with the release of Python 3.3. PEP 393 got rid of "narrow" and "wide" builds and introduced the flexible string representation. This representation made Python strings true Unicode strings without exceptions. Its essence can be summarized as follows. Three different fixed-width encodings are used to represent strings: UCS-1, UCS-2 and UCS-4. Which encoding is used for a given string depends on the largest code point in that string:

- If all code points are in the range U+0000..U+00FF, then UCS-1 is used. UCS-1 encodes code points in that range with one byte and does not encode other code points at all. It's equivalent to the Latin-1 (ISO 8859-1) encoding.
- If all code points are in the range U+0000..U+FFFF and at least one code point is in the range U+0100..U+FFFF, then UCS-2 is used.
- Finally, if at least one code point is in the range U+010000..U+10FFFF, then UCS-4 is used.

In addition to this, CPython distinguishes the case when a string contains only ASCII characters. Such strings are encoded using UCS-1 but stored in a special way. Let's take a look at the actual code to understand the details.

Meet modern Python strings

CPython uses three structs to represent strings: `PyASCIIObject`, `PyCompactUnicodeObject` and `PyUnicodeObject`. The second one extends the first one, and the third one extends the second one:

```
typedef struct {
  PyObject_HEAD
  Py_ssize_t length;
  Py_hash_t hash;
  struct {
      unsigned int interned:2;
      unsigned int kind:2;
      unsigned int compact:1;
      unsigned int ascii:1;
      unsigned int ready:1;
  } state;
  wchar_t *wstr;
} PyASCIIObject;

typedef struct {
  PyASCIIObject _base;
  Py_ssize_t utf8_length;
  char *utf8;
  Py_ssize_t wstr_length;
} PyCompactUnicodeObject;

typedef struct {
  PyCompactUnicodeObject _base;
  union {
      void *any;
      Py_UCS1 *latin1;
      Py_UCS2 *ucs2;
      Py_UCS4 *ucs4;
  } data;
} PyUnicodeObject;
```

Why do we need all these structs? Recall that CPython provides the Python/C API that allows writing C extensions. In particular, it provides a set of functions to work with strings. Many of these functions expose the internal representation of strings, so PEP 393 could not get rid of the old representation without breaking C extensions. One of the reasons why the current representation of strings is more compilcated than it should be is because CPython continues to provide the old API. For example, it provides the `PyUnicode_AsUnicode()` function that returns the `Py_UNICODE*` representation of a string.

Let's first see how CPython represents strings created using the new API. These are called "canonical" strings. They include all the strings that we create when we write Python code. The `PyASCIIObject` struct is used to represent ASCII-only strings. The buffer that holds a string is not a part of the struct but immediately follows it. The allocation is done at once like this:

```
obj = (PyObject *) PyObject_MALLOC(struct_size + (size + 1) * char_size);
```

The `PyCompactUnicodeObject` struct is used to represent all other Unicode strings. The buffer is allocated in the same way right after the struct. Only `struct_size` is different and `char_size` can be `1`, `2` or `4`.

The reason why both `PyASCIIObject` and `PyCompactUnicodeObject` exist is because of an optimization. It's often necessary to get a UTF-8 representation of a string. If a string is an ASCII-only string, then CPython can simply return the data stored in the buffer. But otherwise, CPython has to perform a conversion from the current encoding to UTF-8. The `utf8` field of `PyCompactUnicodeObject` is used to store the cached UTF-8 representation. This representation is not always cached. The special API function `PyUnicode_AsUTF8AndSize()` should be called when the cache is needed.

If someone requests the old `Py_UNICODE*` representation of a "canonical" string, then CPython may need to perform a conversion. Similarly to `utf8`, the `wstr` field of `PyASCIIObject` is used to store the cached `Py_UNICODE*` representation.

The old API allowed creating strings with a `NULL` buffer and filling the buffer afterwards. Today the strings created in this way are called "legacy" strings. They are represented by the `PyUnicodeObject` struct. Initially, they have only the `Py_UNICODE*` representation. The `wstr` field is used to hold it. The users of the API must call the `PyUnicode_READY()` function on "legacy" strings to make them work with the new API. This function stores the canonical (USC-1, UCS-2 or UCS-4) representation of a string in the `data` field of `PyUnicodeObject`.

The old API is still supported but deprecated. PEP 623 lays down a plan to remove it in Python 3.12.

Perhaps the most interesting question about the flexible string representation is how to get it. Typically, a string is created by decoding a sequence of bytes using some encoding. This is how the parser creates strings from string literals. This is how contents of a file become strings. And this is what happens when we call the `decode()` method of a `bytes` object. In all of these cases Python uses the UTF-8 encoding by default, so let's discuss the algorithm that decodes a UTF-8-encoded text to a Python string. It's not immediately obvious how to implement such an algorithm because CPython needs to choose an appropriate struct and encoding to represent the string (ASCII, UCS-1, UCS-2 or UCS-4), and it must decode all the code points to do that. One solution would be to read the input twice: the first time to determine the largest code point in the input and the second time to convert the input from the UTF-8 encoding to the chosen internal encoding. This is not what CPython does. It tries to be optimistic and initially creates an instance of `PyASCIIObject` to represent the string. If it encounters a non-ASCII character as it reads the input, it creates an instance of `PyCompactUnicodeObject`, chooses the next most compact encoding that is capable of representing the character and converts the already decoded prefix to the new encoding. This way, it reads the input once but may change the internal representation up to three times. The algorithm is implemented in the `unicode_decode_utf8()` function in `Objects/unicodeobject.c`.

There is a lot more to say about Python strings. The implementation of string methods, such as `str.find()` and `str.join()`, is an interesting topic, but it probably deserves a separate port. Another topic worth discussing is string interning. We'll cover it when we take a look at how Python dictionaries work. This post focuses on how CPython implements strings and it won't be complete if we don't discuss alternative ways to implement strings in a programming language, so that's what we'll do now.

How other Python implementations represent strings

The flexible string representation is quite complex, so you might wonder whether other Python implementations, such as PyPy and MicroPython, use it. The short answer is: they don't. In fact, I'm not aware of any other language, not to say about Python implementation, that takes CPython's approach.

MicroPython uses UTF-8 for the string representation. Strings are true Unicode strings just like in CPython. Code point indexing is supported but implemented by scanning the string, so it takes $O(n)$ time to access the nth code point.

PyPy uses UTF-8 too. But it does code point indexing in constant time. The trick is simple. Here's how you can do it. Think about a UTF-8 representation as a sequence of blocks, each block (with the possible exception of the last) containing 64 code points. Create an array of integers such that ith element of the array is a starting byte position of the ith block. Then the nth code point of a string can be found as follows:

```
def get_code_point(buffer, n):
    block_num, code_point_in_block = divmod(n, 64)
    block_start_byte = block_index[block_num]
    return seek_forward(buffer[block_start_byte:], code_point_in_block)
```

This message on the pypy-dev mailing list explains the algorithm in more detail.

MicroPython and PyPy have to implement the same strings that CPython implements in order to stay compatible with it. But other languages have different views on what a string should be in the first place. It's especially interesting to look at those languages that were designed with Unicode in mind. This is the focus of the next section.

How strings work in other languages

C

The most primitive form of a string data type is an array of bytes. Python 2 strings are an example of this approach. It comes from C where strings are represented as arrays of `char`. The C standard library provides a set of functions like `toupper()` and `isspace()` that take bytes and treat them as characters in the encoding specified by the current locale. This allows working with encodings that use one byte per character. To support other encodings, the `wchar_t` type was introduced in the C90 standard. Unlike `char`, `wchar_t` is guaranteed to be large enough to represent all characters in any encoding specified by any supported locale. For example, if some locale specifies the UTF-8 encoding, then `wchar_t` must be large enough to represent all Unicode code points. The problem with `wchar_t` is that it is platform-dependent and its width can be as small as 8 bits. The C11 standard addressed this problem and introduced the `char16_t` and `char32_t` types that can be used to represent code units of UTF-16 and UTF-32 respectively in a platform-independent way. Chapter 5 of the Unicode Standard discusses Unicode data types in C in more detail.

Go

In Go, a string is a read-only slice of bytes, i.e. an array of bytes along with the number of bytes in the array. A string may hold arbitrary bytes just like an array of `char` in C, and indexing into a string returns a byte. Nevertheless, Go provides decent Unicode support. First, Go source code is always UTF-8. This means that string literals are valid UTF-8 sequences. Second, iterating over a string with the `for` loop yields Unicode code points. There is a separate type to represent code points – the `rune` type. Third, the standard library provides functions to work with Unicode. For example, we can use the `ValidString()` function provided by the `unicode/utf8` package to check whether a given string is a valid UTF-8 sequence. To learn more about strings in Go, check out this excellent article written by Rob Pike.

Rust

Rust provides several string types. The main string type, called `str`, is used to represent UTF-8-encoded text. A string is a slice of bytes that cannot hold arbitrary bytes but only a valid UTF-8 sequence. Attempt to create a string from a sequence of bytes that is not a valid UTF-8 sequence results in an error. Indexing into a string by an integer is not supported. The docs give a reasoning for that:

> *Indexing is intended to be a constant-time operation, but UTF-8 encoding does not allow us to do this. Furthermore, it's not clear what sort of thing the index should return: a byte, a codepoint, or a grapheme cluster. The `bytes` and `chars` methods return iterators over the first two, respectively.*

The iteration is the way to access code points. Nevertheless, it's possible to index into a string by a range, like `&string[0..4]`. This operation returns a substring consisting of bytes in the specified range. If the substring is not a valid UTF-8 sequence, the program will crash. It's always possible to access individual bytes of a string by converting it to a byte slice first. To learn more about strings in Rust, check out Chapter 8 of the Rust Programming Language book.

Swift

Swift takes the most radical approach when it comes to Unicode support. A string in Swift is a sequence of Unicode grapheme clusters, that is, a sequence of human-perceived characters. The `count` property returns the number of grapheme clusters:

```
let str = "\u{65}\u{301}"
print(str)
print(str.count)

// Output:
// é
// 1
```

And iterating over a string yields grapheme clusters:

```
let str = "Cluster:\u{1112}\u{1161}\u{11AB} "
for c in str {
    print(c, terminator:" ")
}

// Output:
// C l u s t e r : 한
```

To implement such behavior, a language must be able to detect boundaries of grapheme clusters. The Unicode Standard Annex #29 describes how to do that algorithmically.

Internally, a string is stored in the UTF-8 encoding. Indexing into a string by an integer is not supported. There is an API, though, that allows accessing grapheme clusters by indices:

```
let str = "Swift";
let c = str[str.index(str.startIndex, offsetBy: 3)]
print(c)

// Output:
// f
```

It looks intentionally clumsy to remind programmers about the expensiveness of the operation. To learn more about strings in Swift, check out the Language Guide.

Conclusion

In the modern world of programming, the word "string" means Unicode data. Programmers should be aware of how Unicode works, and language designers should provide the right abstraction to deal with it. Python strings are sequences of Unicode code points. The flexible string representation allows indexing into a string in constant time and, at the same time, tries to keep strings relatively compact. This approach seems to work well for Python because accessing elements of a string is easy, and in most cases programmers don't even think whether those elements should be characters or grapheme clusters. Modern languages, such as Go, Rust and Swift, questioned whether indexing into a string is important at all. They give us an idea of what the best approach for implementing strings may look like: represent strings internally as UTF-8 sequences and provide a set of iterators that yield bytes, code units, code points and grapheme clusters. Python evolves. Will it gravitate towards this approach in the future?

The implementation of built-in types is a fascinating topic. It's always interesting and useful to know how things you constantly deal with actually work. This is especially true of Python dictionaries. They are not only extensively used by programmers but also underlie important features of the language. Next time we'll see how they work.

If you have any questions, comments or suggestions, feel free to contact me at victor@tenthousandmeters.com