James K. Tauber

Character Encoding of Classical Languages

Abstract: Underlying any processing and analysis of texts is the need to represent the individual characters that make up those texts. For the first few decades, scholars pioneering digital classical philology had to adopt various workaround for dealing with the various scripts of historical languages on systems that were never intended for anything but English. The Unicode Standard addresses many of the issues with character encoding across the world's writing systems, including those used by historical languages, but its practical use in digital classical philology is not without challenges. This chapter will start with a conceptual overview of character coding systems and the Unicode Standard in particular but will discuss practical issues relating to the input, interchange, processing and display of classical texts. As well as providing guidelines for interoperability in text representation, various aspects of text processing at the character level will be covered including normalisation, search, regular expressions, collation, and alignment.

Introduction

The representation of texts electronically must be grounded in the representation of individual characters in those texts and it is for this reason that character encoding is a foundational part of digital philology.

In this chapter we will look at the character encoding of classical texts with an emphasis on Unicode. I will provide a conceptual introduction and brief history to illustrate the development of those concepts. To avoid being too abstract, however, I will give examples relevant to Ancient Greek as well as demonstrate certain processing characteristics of Unicode via snippets of Python. I include discussion of things to watch out for and common pitfalls.

Preliminaries and history

The idea of encoding the letters of the alphabet using combinations of a much smaller set of symbols goes back centuries. Francis Bacon developed a secret

James K. Tauber, Eldarion

https://doi.org/10.1515/9783110599572-009

code that represented each letter as a sequence of just two symbols. This approach foreshadowed both the telegraph and the computer where letters, numbers, and other characters are encoded as sequences of 1s and 0s.

In 1870, Émile Baudot developed a code whose descendants became the standard for the telegraph, used right up until the introduction of ASCII. Baudot's code, which had some similarities to Bacon's handwritten cipher, was a fixed-length encoding where each letter was represented by a sequence of five binary digits – 1s and 0s or "bits" – enabling the encoding of 32 different characters at a time. In fact, Baudot's code supported 64 different characters, by having two sets of 32 characters and reserving one character in each set to mean "switch to the other set".

In 1963, the 7-bit American Standard Code for Information Interchange (ASCII) was released and quickly became widely adopted. The extra codes that seven bits afforded meant a certain number of codes could be used, not for printable characters but as control codes for printers to indicate line breaks, page breaks, tabs, and so on.

As computers increasingly adopted ASCII, however, there was a problem. The set of characters supported was fine for the US but was inadequate for languages in Western Europe, to say nothing of scripts such as Greek or Cyrillic. The East Asian languages, with their large inventories of ideographs were out of the question.

The first work around, where only a handful of necessary characters were missing, was to replace lesser-used characters. Some French-speaking countries used a variant of ASCII that replaced { and } with é and è respectively. Countries like Greece replaced the entire repertoire of letters with their own (forming the ELOT 927 standard). These work arounds meant you always had to be aware of what particular character set was being used.

Over time, as more computer systems adopted the 8-bit "byte", it became helpful to use all 256 codes that this enabled. Various character sets were developed that kept the first 128 codes identical to ASCII with the other 128 available for extra characters. This meant ASCII characters (including the control characters) could be transmitted without worrying about the particular variant being used. It also meant a full 128 extra characters were possible.

A number of variants adopting this approach were standardised as the ISO-8859 family. Each was compatible with ASCII for the first 128 codes but specified a different set of additional 128 characters. ISO 8859-1 (also known as Latin-1) extended ASCII with most of the characters needed for the languages of Western Europe. ISO 8859-2 did the same for Central and Eastern European countries using a Latin alphabet. ISO 8859-5 provided Cyrillic, ISO 8859-7 provided monotonic Greek, and ISO 8859-8 provided modern Hebrew.

As well as those ratified by standards organisations, additional character encodings were introduced by various operating system vendors. Still used to this day is Microsoft Windows CP-1252, a variant of Latin-1.

Alongside this explosion of 8-bit character encodings, countries such as China, Japan and Korea developed their own character encodings that used two bytes to support the larger set of characters they required.

Unicode history

In late 1987, developers at Apple and Xerox started working on a single "Unicode" to rule them all. The name was intended to evoke the idea of being universal in coverage of the world's writing system, uniform in structure, and with each character assigned a unique code position.

It was determined a fixed-width 16-bit encoding (allowing up to 65,536 characters) would be sufficient for unifying existing character encoding standards.

Unicode 1.0 was published in 1991. There was a competing international standard being developed at the same time but it was eventually agreed that the international standard, ISO 10646, would synchronise their character codes with Unicode and this synchronisation continues to this day.

In 1996, Unicode 2.0 was released and, through the use of surrogate pairs (see below) it expanded the available codespace 17-fold from 65,536 positions to 1,114,112, largely to accommodate archaic and historical writing systems that were not originally envisaged to need encoding.

Since then there have been regular releases of Unicode and, at the time of writing, Unicode 12.0 is being prepared.

Other non-Unicode approaches

Although Unicode is the only reasonable choice for digital philology nowadays, other approaches to the representation of characters in historical languages were used before the widespread adoption of Unicode. As they may still be encountered in legacy data, it is worth mentioning two.

Custom fonts

One approach common before the broad adoption of Unicode and still seen in older online resources is the use of custom fonts that place non-standard

glyphs in place of those assumed by the character encoding. This approach relied on the recipient having the necessary font and it being used at the appropriate time.

While this was a usable workaround for display purposes, the conflation between character encoding and font choice made processing much more difficult, particularly due to the inconsistent reuse of codes between fonts.

BetaCode

Pioneering work in digital classical philology was already taking place at a time when computers were only capable of encoding Latin characters, numerals, and some basic punctuation.

BetaCode was developed by David Packard in the late 1970s to enable the representation of Greek characters with this limited character set.

BetaCode is not a character encoding system in the way we mean here as it is not about assigning numerical codes to characters and then encoding them as bits. Instead, BetaCode is essentially a transliteration scheme that maps Greek characters and diacritic marks to the available Latin characters (which may then be represented using any number of character encoding systems). For example, an omega with rough breathing, a circumflex and iota subscript would be transliterated `W(=|`.

In practice, some resources are inconsistent in their ordering of diacritics in BetaCode and care must be taken, particularly when converting to Unicode.

Unicode fundamentals

In any digital processing of text, we are potentially interested in a variety of textual elements. We may care about words, sentences, paragraphs, chapters, or entire works. We may care about elements smaller than a word: syllables, and individual letters (with or without diacritic marks such as accents).

Which elements we care about, how we demarcate them, how we determine their equivalence to one another will depend on not only the specific language but the particular task at hand. A search may want to ignore any distinction between uppercase and lowercase letters, or the accentuation but those are important in running text. Running text may not display vowel length but a dictionary might.

The goal of a character encoding system is to provide a set of fundamental units on top of which larger text elements can be built. These fundamental

units, called *characters* are assigned numerical *character codes*. Any text element then becomes representable as a sequence of these character codes.

The first step of any character encoding system is to select which characters need to represented. This collection of characters is referred to as the *character repertoire*. Some space of numbered code positions (or *code points*) is then defined and characters are assigned to these points. This effectively assigns each character a numerical code. There are then a number of ways of encoding this numerical code as a string of one or more bytes.

Beyond all this, Unicode provides data about each character, necessary for its rendering and processing, as well as various algorithms for text processing that make use of these data.

Character repertoires and code spaces

As of version 12.0, Unicode has a repertoire of 137,929 characters. This covers not only characters from scripts in modern use but also many archaic and historic scripts. Ongoing work ensures increasing coverage.

The original Unicode specification only had a codespace of 65,536 points (representable with 16-bit numbers) but since 2.0, Unicode has supported 1,114,112 (17 × 65,536) points to which characters may be assigned.

A code position (also known as a *code point*) is really just an integer and the Unicode codespace consists of the range of integers from 0 to 1,114,111 (10FFFF in hex).

Some are reserved for future use, private use, or use in surrogate pairs (see below). A few are marked as never to be used. The majority of code points, however, are intended for *graphic characters*, that is characters to be displayed. A small number are for control codes (mostly due to legacy from ASCII).

Once a character from the repertoire is assigned a code point in the codespace, it is called an *encoded character* or *coded character*. We can refer to a particular character at a code point with `U+XXXX` where `XXXX` is the code point in hexadecimal.

In Python 3, the built-in `ord()` function will return the code-point of a one-character string.

```
>>> ord('α')
945
```

or as a hex string:

```
>>> hex(ord('α'))
'0x3b1'
```

Given an integer, `chr()` will return the character at that code-point.

```
>>> chr(0x3B1)
'α'
```

Outside of a specific programming language, we would say that 'α' is U+03B1. Within a Python string, characters may be referred to by their (hexadecimal) code point using `\uXXXX`.

```
>>> 'alpha is \u03b1'
'alpha is α'
```

Characters versus glyphs

Selecting a character repertoire means deciding what counts as a distinct "character".

A key design principle of the Unicode Standard is that characters are not the same as glyphs. A *character* is an abstract element in a writing system whereas a *glyph* is a specific shape rendered on screen or in print. The shape of a lowercase 'a' in a particular font is a glyph but the abstract idea of a "lowercase a" is a character.

Unicode is fundamentally concerned with characters, not glyphs, and while the Unicode code charts give example glyphs to provide guidance on what is meant by a particular character, there is nothing in Unicode that describes the precise shape, size, or orientation of glyphs.

The reasons for why Unicode draws the distinctions it does with certain characters, while conflating others, are complex and there are notions of equivalence and compatibility between characters that will be discussed later. Both purity and practicality have a part to play and it must be remembered that a major goal of Unicode was to unify existing character encoding systems which may or may not have had exactly the same design philosophy of Unicode.

But to give some flavour of the challenges, consider the following.

A Latin capital A and Greek capital Alpha might look identical in a font, but are they the same character? Is the µ used for the unit prefix micro- the same as the Greek letter? Is a superscript digit the same character as the subscript version? Is a final sigma in Greek just a variant glyph or a different character from the normal sigma? What about the differing initial, medial and final forms of Arabic letters? Is 'é' a single character or is the acute accent a separate character added to the 'e'? If the latter, is that acute the same acute used in the Greek 'έ' ?

The answers Unicode has for these questions will depend on legacy encoding, whether characters are used in the same writing system or not, whether there would be visual confusion between two characters, whether the characters behave differently in case folding or text segmentation or sorting, and so on.

The structure of the Unicode codespace

There is a structure to the space of code points to which characters are assigned and we provide a brief overview of that structure here.

Planes and blocks

As mentioned earlier, in 1996, Unicode extended the codespace with an additional 16 x 65,536 code points on top of the original 65,536. Each chunk of 65,536 points is called a *plane*. The original space of code points, representable with just a single 16-bit number, is now referred to as the *Basic Multilingual Plane* (BMP) or Plane 0. The other 16 planes are collectively referred to as the *Supplementary Planes*.

Within a plane, code positions are grouped into blocks. Blocks are just an organisational device and not necessarily an indication of language, script, or any specific character properties. A sample of the blocks most relevant to classical philology include:

Basic Latin (ASCII)	0000–007F
Latin-1 Supplement	0080–00FF
Spacing Modifier Letters	02B0–02FF
Combining Diacritical Marks	0300–036F
Greek	0370–03FF
Greek Extended	1F00–1FFF
General Punctuation	2000–206F

although many others exist for the other scripts of historical languages.

Private use area

To accommodate the encoding of characters not currently (or ever to be) represented in Unicode, there are a number of code points designated as a *Private Use Area* (PUA).

The area from U+E000 to U+F8FF (consisting of 6,400 code points) as well as the entirety of Planes 15 and 16 are designated as Private Use Areas.

Communities can assign otherwise unsupported characters to points in the private use area by agreement among themselves without fear of clashing with any official assignments.

Private use areas are used by everything from scholars working on obscure writing systems or transcriptions of manuscripts with rare symbols to conlangers wanting to exchange electronic texts in their favourite constructed language.

Surrogate pairs

The area from U+D800 to U+DFFF is designated for use in surrogate pairs which is a mechanism used by the UTF-16 encoding form to address the supplementary planes with two 16-bit code units while still only using a single 16-bit code unit for characters in the BMP. This is described in the section below on UTF-16.

Diacritics and modifying marks

Many writing systems involve marks that, in some way, modify a base character. For example, the grave accent in è, the diaeresis in ï, the ogonek in ǫ, the macron in ā, or the rough breathing and acute accent in ᾅ.

The combinatorial possibilities quickly explode, especially when one considers that modifying marks may combine (as in the case with the alpha above). If each combination was assigned to its own code point, thousands of code points would be necessary. So the approach taken by Unicode is to separately assign each base character and modifying mark to their own code point and a sequence of two or more characters is used to convey a base with its diacritics.

Now there are some exceptions to this. Some combinations, for legacy reasons, have a dedicated code point. This is true for Greek because of existing character encoding systems that Unicode was incorporating. It's important to recognise this an exception, though, and even in the case of Greek, not all combinations are expressed in this way.

This does, though, lead to different sequences of Unicode character essentially meaning the same thing. For this reason, and as will be discussed below, Unicode has notions of equivalency and character sequences can be normalised for comparison purposes.

Unicode character database

The *Unicode Character Database* provides a wide range of properties for each character useful in various processing and rendering applications. The database gives a name for the character, the block the character is in and, in many cases, the script the character is part of.

In addition, the database provides information on case mappings, directionality, decompositions, and more. This information is vital to many of the algorithms that accompany the Unicode Standard and help define how to divide words and break lines, sort text, format numbers, fold cases, handle bidirectional text, optionally ignore diacritics when searching and so on.

The Unicode Character Database is provided in a machine readable form for download. Some of the more commonly used information is also directly available from Python via the `unicodedata` standard library module.

The `name()` function in the `unicodedata` module returns the name of the given character:

```
>>>import unicodedata
>>> unicodedata.name('α')
'GREEK SMALL LETTER ALPHA'
```

General character categories

Every assigned character in Unicode has a *general category*. This is one of the properties defined for each character in the Unicode Character Database.

The top-level general categories are Letter (L), Mark (M), Number (N), Punctuation (P), Symbol (S), Separator (Z), and Other (C). Each general category is split into further subcategories.

Characters in the M category represent diacritics and are also referred to as *combining characters*. Any graphic character that is not a combining character is said to be a *base character*. Multiple combining characters may be used with a single base character, but the base character always comes first.

Nonspacing marks

Within the combining characters, there is a subset referred to as the *nonspacing marks*. These are marks like smooth breathing, the acute or the macron that wouldn't normally take up any extra width with a fixed-width font. Nonspacing

marks have a general category of Mn unless they fully enclose their base character, in which case they have a general category of Me.

An example of general categories

Consider the following text:

Il 1.1 μῆνιν

Here is each character's code point (in hexadecimal) and category:

I	l		1	.	1		μ	ῆ		ν	ι	ν
49	6C	20	31	2E	31	20	03BC	03B7	0342	03BD	03B9	03BD
Lu	Ll	Zs	Nd	Po	Nd	Zs	Ll	Ll	Mn	Ll	Ll	Ll

Where Lu = uppercase letter, Ll = lowercase letter, Zs = space separator, Nd = decimal digit, and Mn = nonspacing mark. Note that the ῆ is represented by two characters: η, a lowercase letter (Ll) and the circumflex, a nonspacing mark (Mn).

The function `category()` in unicodedata will return the category of the given character.

```
>>> unicodedata.category(",")
'Po'
```

Encoding forms

At the core of Unicode is the assignment of characters to code points, which effectively assigns each character in the repertoire a unique integer representation. But our ultimate goal, at least in terms of storage and transmission, is how to represent characters in terms of bits.

There are three primary mappings-to-bits, called *encoding forms*, that Unicode provides: UTF-32, UTF-16, and UTF-8. These respectively use sequences of 32-bit, 16-bit, and 8-bit *code units* to represent character sequences.

All three encoding forms are capable of representing all code points in Unicode but have different advantages in different contexts. The follow table demonstrates the different encoding forms applied to characters from different parts of the code space:

		UTF-32	UTF-16	UTF-8
a	U+0061	00000061	0061	61
æ	U+00E6	000000E6	00E6	C3 A6
α	U+03B1	000003B1	03B1	CE B1
ἄ (pre-composed)	U+1F04	00001F04	1F04	E1 BC 84
(Linear B 'A')	U+10000	00010000	D800 DC00	F0 90 80 80

One may occasionally encounter references to UCS-2 or UCS-4. UCS-4 is functionally equivalent to UTF-32 and UCS-2 is a obsolete subset of UTF-16.

UTF-32

UTF-32 is the simplest encoding form for Unicode as it is a fixed-width, 32-bit representation of a code point with no conversion necessary.

UTF-16

UTF-16 is optimised for the BMP and will use a single 16-bit code unit (that is, two bytes) for all characters in that plane. Other planes are still accessible using pairs of 16-bit units (that is, four bytes) known as surrogate pairs.

UTF-16 is at most the same size as UTF-32 and, in the vast majority of cases (namely with characters on the BMP), is half the size. For this reason, UTF-16 is almost always to be preferred over UTF-32.

Surrogate pairs are a pair of code units that, if treated as code points in isolation, are never used. But as a pair, they can be converted to a code point outside the basic plane.

The first in the pair (the *high-surrogate*) must be in the range U+D800 to U+DBFF and the second in the pair (the *low-surrogate*) must be in the range U+DC00 to U+DFFF.

A choice of one of the 1,024 numbers in the high-surrogate range and one of the 1,024 numbers in the low-surrogate range gives 1,024 x 1,024 = 1,048,576 addressable code points, which is the size of the code space in the supplementary planes.

UTF-8

UTF-8 is a variable-width encoding which, although capable of representing the entire Unicode codespace, is optimised for ASCII. A text containing just 7-bit ASCII characters will take one byte per character under UTF-8. This is at the expense of some characters (those from U+800 on up), taking more bytes than UTF-16.

Not only are code points from U+0000 to U+007F represented as one byte in UTF-8, but they are done so in a way that is identical to ASCII. This makes UTF-8 backwards compatible with ASCII. Any valid ASCII sequence is valid UTF-8.

Code points from U+0080 through U+07FF are representable with two bytes, U+0800 through U+FFFF with three bytes, and the supplementary planes with four bytes.

Endianness and the byte order mark

When dealing with code units of more than one byte, there is always the question of whether the individual bytes are big-endian or little-endian (the so-called "endianness"). This means that UTF-16 and UTF-32 encoding forms actually come in two variants. Unicode has a helpful way to give an internal hint as to the endianness used in a sequence.

The code point U+FEFF is assigned a zero width, no-break space. This is a character which basically has no effect. If a system decoding, say, UTF-16 got the endianness wrong, a U+FEFF would come across (incorrectly) as U+FFFE. The latter is a Unicode noncharacter. In other words, the code point is not assigned a character and never will be. That effectively means that if a decoding system encounters a U+FFFE, it must be assuming the wrong endianness.

By putting the U+FEFF character at the start of a UTF-16 or UTF-32 file, you effectively are letting the decoder whether the byte sequence is big or little ending. For this reason, the character is also known as the Byte Order Mark (BOM).

Equivalence, compatibility, and normalization

As mentioned earlier, there is a certain amount of complexity to the question of whether two things should be considered the same character (or sequence of characters) or not. The Unicode Standard has the notion of *equivalence* to formally define that, at least in certain contexts, some sequences of code points should be treated as representing the same abstract character.

Unicode distinguishes two types of equivalence: *canonical* and *compatibility*. Canonical equivalence means that two (sequences of) code points should essentially be considered the same. That is, they should render the same, be processed the same, and be substitutable for one another. Compatibility equivalent means that there may be differences in appearance and in how they should be processed in some situations, but that there are other situations where they could be considered equivalent.

Canonical equivalence

One relevant example of canonical equivalence in sequences with a base character and one or more combining characters. If a pre-composed character exists, it is canonically equivalent to the decomposition into a combining sequence.

For example ἅ, in its pre-composed form, is assigned to U+1F04 and given the name, "GREEK SMALL LETTER ALPHA WITH PSILI AND OXIA". It is canonically equivalent to:

U+03B1	U+0313	U+0301
GREEK SMALL LETTER ALPHA	COMBINING COMMA ABOVE	COMBINING ACUTE ACCENT

The latter is referred to as the *canonical decomposition* of U+1F04.

Note that, in this case, order matters. The sequence U+03B1 U+0301 U+0313 is not equivalent to U+03B1 U+0313 U+301 (and hence not equivalent to U+1F04). The order of the two combining characters matters because they attach to the same place on the base character. The place of attachment is indicated by the *combining class* property.

With something like ᾷ, though, the sequence U+03B1 U+0342 U+0345 is equivalent to U+03B1 U+0345 U+0342 because the two combining characters attach in different places.

Compatibility equivalence

Compatibility equivalence is a weaker condition. Ligatures are generally compatibility equivalent to their separate-letter versions but not canonically equivalent. U+00B5, the code point for the unit prefix micro-, has a *compatibility decomposition* to U+03BC, the Greek mu, but not a canonical decomposition. Superscript and subscript digits (which have their own code points) also have a compatibility decomposition to the corresponding plain digits.

Normalization

If character sequences are canonically equivalent then, in almost all cases you want them to be considered equal when processing. If character sequences are compatibility equivalent, you may also want them to be considered equal. The Unicode Normalization Algorithm transforms strings into a form, called a *normalization form*, such that equivalent strings will have the same form. Once in a normalization form, testing equivalence is therefore just a matter of binary comparison.

There are four Unicode Normalization Forms:

Normalization Form D	NFD	do a canonical decomposition
Normalization Form C	NFC	do a canonical decomposition followed by a canonical composition
Normalization Form KD	NFKD	do a compatibility decomposition
Normalization Form KC	NFKC	do a compatibility decomposition followed by a canonical composition

Normalization can also be used to fully decompose a string or fully compose it.

The NFD form of ἄ U+1F04 “GREEK SMALL LETTER ALPHA WITH PSILI AND OXIA” is, for example, U+03B1 U+0313 U+0301, the base alpha with the smooth breathing and acute as separate combining characters.

The NFC form of U+03B1 U+0313 U+0301 is likewise U+1F04.

The `normalize()` function on `unicodedata` converts Unicode strings to one of the four normalization forms.

```
>>> for character in unicodedata.normalize('NFD', '\u1F04'):
...     print(hex(ord(character)), unicodedata.name(character))
...
0x3b1 GREEK SMALL LETTER ALPHA
0x313 COMBINING COMMA ABOVE
0x301 COMBINING ACUTE ACCENT
>>> unicodedata.normalize('NFC','\u03b1\u0313\u0301') == '\u1F04'
True
```

Whether two strings are canonically equivalent can be determined by comparing either their NFC or NFD normalizations. Whether two strings are compatibility equivalent can be determined by comparing either their NFKC or NFKD normalizations.

Common pitfalls in character encoding

Unknown encoding and mojibake

If a text is interpreted (that is, decoded) assuming a different character encoding from that used to produce it, unexpected characters can result. These unintended characters are referred by the Japanese term *mojibake* which literally means "character changing".

Here is the first line of the *Iliad* when encoded as UTF-8 but treated as Latin-1:

Î¼á¿†Î½Î¹Î½ á¼„ÎµÎ¹Î´Îµ Î¸Îµá½° Î Î·Î»Î·ÏŠÎ¬Î´Îµ á¼ˆÏ‡Î¹Î»á¿†Î¿Ï‚,

One solution is to manually set the character encoding on the receiving side (in, for example, the browser or text editor) to that used by the sender. But given this requires action on the part of the recipient, this is nothing more than a workaround. In many cases, the better solution is to make sure the correct character encoding has been transmitted along with the text. For example, in HTML a `meta charset="utf-8"` element can ensure the correct browser interpretation. XML and JSON can use UTF-8 without any need for explicit declaration.

Missing characters in a font (fall back)

Systems, if asked to render a particular character using a font that does not have a glyph for that character will either display some dummy character or fallback to another font that does have a glyph for the character. If no fonts exist on the system with such a glyph, the system has no choice but to display a dummy character. In the case where a dummy character is used, it is obvious that the font is missing support for the character but in the fallback case, it might not be so clear at first glance but the display will be uneven.

Where this is quite common is the display of Greek and there are two types of problems that can arise if the first font choice does not support all the characters being used.

Firstly there is the case where the first font choice doesn't have the full set of Greek letters but has some subset (often pi, mu, etc). The subset will be displayed in the first font choice with other letters being displayed in the fallback font.

Secondly the first font choice may have Greek support but not the precomposed polytonic Greek characters. In this case, Greek characters without diacritic will use one font and those with another.

Stray look-alikes from another script

This issue is particular pernicious because the text can look fine but process incorrectly. Various Greek characters look similar or identical to Latin characters and it can be visually impossible to pick up when the two have been mixed.

Greek in Unicode

The encoding of Greek in Unicode was initially based on ISO 8859–7 (equivalent to the Greek national standard ELOT 928) which was designed for monotonic Greek with a single "tonos" accent.

The Greek block from U+0370 to U+03FF consists of the letters (both uppercase and lowercase), the vowels with tonos and, where appropriate, with dialytika (diaeresis) and with both tonos and dialytika, punctuation and the numeral signs (keraia).

The block at U+0370 in fact provides all the letterforms for polytonic Greek, just not the breathing, accents (other than tonos), or iota subscript (ypogegrammeni). However, these are all possible via the use of combining characters from the Combining Diacritical Marks block.

The relevant Combining Diacritical Marks in the 0300–036F range are:

Code	Char	Unicode Name	Notes
U+0300	◌̀	COMBINING GRAVE ACCENT	Greek varia
U+0301	◌́	COMBINING ACUTE ACCENT	Greek oxia, tonos
U+0304	◌̄	COMBINING MACRON	long vowel
U+0306	◌̆	COMBINING BREVE	short vowel, Greek vrachy
U+0308	◌̈	COMBINING DIAERESIS	Greek dialytika
U+0313	◌̓	COMBINING COMMA ABOVE	Greek psili, smooth breathing mark
U+0314	◌̔	COMBINING REVERSED COMMA ABOVE	Greek dasia, rough breathing mark
U+0342	◌͂	COMBINING GREEK PERISPOMENI	Greek-specific form of circumflex (whether the glyph looks like tilde or inverted breve)
U+0345	◌ͅ	COMBINING GREEK YPOGEGRAMMENI	

Note that there is a COMBINING CIRCUMFLEX ACCENT at U+0302 but that's not what we think of as a circumflex. In Greek we use U+0342, the COMBINING GREEK PERISPOMENI.

There is a COMBINING GREEK DIALYTIKA TONOS at U+0344 but use of this is discouraged in favour of an explicit sequence of U+0308 U+301 (which is the canonical decomposition for U+0344 anyway).

There is a COMBINING GREEK KORONIS at U+0343 but this is canonically equivalent to U+0313 which is preferred.

The Greek Extended block from U+1F00 to U+1FFF exists to provide precomposed polytonic Greek characters. Note, however, that it is entirely possible to do polytonic Greek without this block, using base characters from the "Greek" block at U+0370 along with combining characters outlined above.

The following are the code points preferred for punctuation and the numeral sign commonly found in Greek texts:

Code	Char	Unicode Name	Notes
U+002C	,	COMMA	
U+002E	.	FULL STOP	
U+003B	;	SEMICOLON	preferred over U+037E "GREEK QUESTION MARK"
U+00B7	·	MIDDLE DOT	preferred over U+0387 "GREEK ANO TELEIA"
U+02B9	ʹ	MODIFIER LETTER PRIME	preferred over U+0374 "GREEK NUMERAL SIGN"
U+2019	’	RIGHT SINGLE QUOTATION MARK	preferred over U+02BC "MODIFIER LETTER APOSTROPHE"

Other issues with Unicode Greek

Precomposed vs decomposed characters

Given that almost (but not) all useful combinations of Greek base character with combining characters also have a precomposed equivalent, the question arises: is one preferred over the other?

It should be again noted that the existence of the precomposed characters is for legacy reasons. Without the constraints of existing ELOT standards, Unicode almost certainly would have done away with the precomposed characters.

Many processing tasks (collation, accent-stripping, etc) are more easily done with decomposed characters. Keyboard input can also be easier with decomposed characters. This does place extra burden on fonts and rendering systems but ultimately this is where the burden should lie.

All this said, the storage and transmission of precomposed Greek text is not particularly problematic given decomposition is easily achievable via normalization forms. Many electronic Greek texts normalize to NFC for storage.

Combining accents and vowel length

Running Greek text rarely indicates vowel length but dictionaries may do so. There are no adequate precomposed characters for representing things like the imperfect ἵστημι. Unicode is perfectly capable of representing it with combining characters but input and rendering systems, including the fonts themselves, sometimes lack support.

Tonos vs oxia

During the monotonic reform of the Greek language in the 1980, the number of accents was reduced from three to just one, the tonos. Although the tonos resembled the acute, reformers were keen to distance themselves from the older system and encouraged type designers to make it visually distinct from the acute.

In 1986, the Greek government officially equated the tonos with the acute. Unfortunately, this equivalency did not make it into Unicode until version 3.0 and so we are left with TONOS pre-compositions in the Greek block alongside OXIA pre-composition in the Greek Extended block, both of which decompose to a combining acute.

Since Unicode 3.0, the normalization of OXIA to TONOS should be harmless and from the point of view of processing, it is. Unfortunately there are a number of otherwise excellent fonts that render tonos and acute distinctly. In the eyes of Unicode and the Greek government, however, they are equivalent.

Which phi and which theta?

U+03C6 "GREEK SMALL LETTER PHI" and U+03B8 "GREEK SMALL LETTER THETA" should be used for Greek text (regardless of whether straight or looped style). U+03D5 "GREEK PHI SYMBOL" and U+03D1 "GREEK THETA SYMBOL" are only intended for mathematical symbols.

Apostrophe marking elision

Besides the straight apostrophe at U+0027 (which really only exists for compatibility with ASCII) the characters U+2019 and U+02BC are also apostrophe-like characters.

Despite the name "RIGHT SINGLE QUOTATION MARK", U+2019 is intended for both the quotation mark and an apostrophe when used as punctuation (in English, for example, to mark contractions or the possessive).

U+02BC "MODIFIER LETTER APOSTROPHE" is intended when either a distinct letter (in many languages representing a glottal stop) or when modifying a base character (often to represent an ejective).

The apostrophe marking elision in Greek (for example, in γένοιτ' ἄν) falls into the former category and so the Unicode Standard prefers U+2019 for that purpose.

There are some resources, however, that use U+02BC. The primary reason for this choice seems to be that some systems doing text segmentation assume U+2019 is a quotation mark and not part of the preceding word. This is a limitation of the text segmentation being used.

The use of U+02BC is therefore a workaround to deal with incorrect word-breaking by tools. From a character coding, perspective, U+2019 is the correct code point to use.

Keyboard input

Operating systems support virtual keyboards or input sources that map the keys pressed on a physical keyboard to alternative code points for entering a particular script.

There are multiple ways these virtual keyboards / input sources support the entry of characters with diacritics.

The first approach involves pressing the key for the base character, followed by a key for the non-spacing combining character.

The second approach involves a *dead key* for the desired diacritic first which puts the keyboard in a state waiting for a legal base character to be entered second. This may result in a precomposed character rather than a combining character sequence.

Different virtual keyboard layouts may offer one or both of these approaches.

Processing Unicode

Stripping diacritics

With a decomposition it is easy to strip diacritics by filtering out particular characters either by exact match (if you want to strip specific diacritics such as accents while keeping, say, breathing intact) or by general category to strip all.

Sorting and collation

Without additional character data, the sorting of text is usually purely on the basis of code points. In other words, if a character's code point is earlier in the codespace, the character sorts earlier. This is rarely appropriate for at least three reasons. Firstly, the layout of characters in the codespace is based on many factors besides what their collation order should be. Secondly, culturally expected collation order is often language-specific. Language communities sharing the same script still may not sort words the same way. Thirdly, sorting often cannot be done by treating characters in isolation. Sorting in many languages involves treating certain sequences of characters as a single unit.

Fortunately, Unicode specifies the Unicode Collation Algorithm (UCA) for this purpose. Information about how characters are to be sorted are represented in a *collation element table*. Particular locales may require a custom collation element table, although Unicode does provide a default, the Default Unicode Collation Element Table (DUCET) which, out-of-the-box, enables the correct sorting of Ancient Greek.

Many tools such as databases and text editors support the UCA via the International Components for Unicode (ICU) C++ library. Wrappers for ICU exist for many programming languages (including Python) but this author has also implemented a pure-Python version of the UCA called PyUCA.

Regular expressions

Regular expressions in most languages have a certain amount of support for Unicode. In Python (and many other languages) a `\d` in a regex will match all digits (category Nd) not just 0–9 and `\w` will match non-Latin characters just as well as Latin.

There is also an external Python library `regex` which provides richer support including matching by properties such as block, script, and general category.

With all processing, it is important to be aware of equivalency issues and using a normalization form is recommendation for most processing.

The future of Unicode

Through the efforts of projects such as the Script Encoding Initiative, the world's remaining uncoded minority scripts have a good chance of eventual representation in the Standard. Since the expansion of the codespace to the supplementary planes, there is opportunity for encoding the historical scripts not yet included. And, in the meantime, the private use areas are available.

Support for Unicode in operating systems, programming languages, text editors, and fonts is widespread with only occasional shortcomings. For the foreseeable future, the Unicode Standard provides the single most stable representation for the characters in digital texts whether modern or historical.

References

Author's Unicode Resources. https://jktauber.com/unicode/ (last access 2019.01.31).
ICU – International Components for Unicode. http://site.icu-project.org/ (last access 2019.01.31).
Nick Nicholas's Greek Unicode Issues. http://www.opoudjis.net/unicode/unicode.html (last access 2019.01.31).
Python Library unicodedata. https://docs.python.org/3/library/unicodedata.html (last access 2019.01.31).
Python Library regex. https://pypi.org/project/regex (last access 2019.01.31).
Python Library pyuca. https://pypi.org/project/pyuca (last access 2019.01.31).
Script Encoding Initiative. http://www.linguistics.berkeley.edu/sei/ (last access 2019.01.31).
The Unicode Standard. http://www.unicode.org/versions/latest/ (last access 2019.01.31).
Unicode FAQ on Greek Language and Script. https://www.unicode.org/faq/greek.html (last access 2019.01.31).