like `gb2312` or `Shift_JIS`, but not UTF-8footnote:[On this topic, a useful answer on StackOverflow is Difference between MBCS and UTF-8 on Windows..

On GNU/Linux and OSX all of these encodings are set to UTF-8 by default, and have been for several years, so I/O handles all Unicode characters. On Windows, not only are different encodings used in the same system, but they are usually codepages like `'cp850'` or `'cp1252'` that support only ASCII with 127 additional characters that are not the same from one encoding to the other. Therefore, Windows users are far more likely to face encoding errors unless they are extra careful.

To summarize, the most important encoding setting is that returned by `locale.getpreferredencoding()`: it is the default for opening text files and for `sys.stdout/stdin/stderr` when they are redirected to files. However, the documentation reads (in part):

> `locale.getpreferredencoding(do_setlocale=True)`
>
> Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess. […]

Therefore, the best advice about encoding defaults is: do not rely on them.

If you follow the advice of the Unicode sandwich and always are explicit about the encodings in your programs, you will avoid a lot of pain. Unfortunately Unicode is painful even if you get your `bytes` correctly converted to `str`. The next two sections cover subjects that are simple in ASCII-land, but get quite complex on planet Unicode: text normalization — i.e. converting text to a uniform representation for comparisons — and sorting.

Normalizing Unicode for saner comparisons

String comparisons are complicated by the fact that Unicode has combining characters: diacritics and other marks that attach to the preceding character, appearing as one when printed.

For example, the word "café" may be composed in two ways, using 4 or 5 code points, but the result looks exactly the same:

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

The code point U+0301 is the `COMBINING ACUTE ACCENT`. Using it after "e" renders "é". In the Unicode standard, sequences like `'é'` and `'e\u0301'` are called "canonical equivalents", and applications are supposed to treat them as the same. But Python sees two different sequences of code points, and considers them not equal.

The solution is to use Unicode normalization, provided by the `unicodedata.normalize` function. The first argument to that function is one of four strings: `'NFC'`, `'NFD'`, `'NFKC'` and `'NFKD'`. Let's start with the first two.

NFC (Normalization Form C) composes the code points to produce the shortest equivalent string, while NFD decomposes, expanding composed characters into base characters and separate combining characters. Both of these normalizations make comparisons work as expected:

```
>>> from unicodedata import normalize
>>> s1 = 'café'  # composed "e" with acute accent
>>> s2 = 'cafe\u0301'  # decomposed "e" and acute accent
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

Western keyboards usually generate composed characters, so text typed by users will be in NFC by default. But to be safe it may be good to sanitize strings with `normalize('NFC', user_text)` before saving. NFC is also the normalization form recommended by the W3C in Character Model for the World Wide Web: String Matching and Searching.

Some single characters are normalized by NFC into another single character. The symbol for the ohm Ω unit of electrical resistance is normalized to the Greek uppercase omega. They are visually identical, but they compare unequal so it is essential to normalize to avoid surprises:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

The letter K in the acronym for the other two normalization forms — NFKC and NFKD — stands for "compatibility". These are stronger forms of normalization, affecting the so called "compatibility characters". Although one goal of Unicode is to have a single "canonical" code point for each character, some characters appear more than once for compatibility with preexisting standards. For example, the micro sign, `'µ'` (`U+00B5`) was added to Unicode to support round-trip conversion to `latin1`, even though the same character is part of the Greek alphabet with code point is `U+03BC` (`GREEK SMALL LETTER MU`). So, the micro sign is considered a "compatibility character".

In the NFKC and NFKD forms, each compatibility character is replaced by a "compatibility decomposition" of one or more characters that are considered a "preferred" representation, even if there is some formatting loss — ideally, the formatting should be the responsibility of external markup, not part of Unicode. To exemplify, the compatibility decomposition of the one half fraction `'½'` (`U+00BD`) is the sequence of three characters `'1/2'`, and the compatibility decomposition of the micro sign `'µ'` (`U+00B5`) is the lowercase mu `'μ'` (`U+03BC`)[12].

Here is how the NFKC works in practice:

```
>>> from unicodedata import normalize, name
>>> half = '½'
>>> normalize('NFKC', half)
'1/2'
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'µ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('µ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

Although `'1/2'` is a reasonable substitute for `'½'`, and the micro sign is really a lowercase Greek mu, converting `'4²'` to `'42'` changes the meaning[13]. An application could store `'4²'` as `'4<sup>2</sup>'`, but the `normalize` function knows nothing about formatting. Therefore, NFKC or NFKD may lose or distort information, but they can produce

12. Curiously, the micro sign is considered a "compatibility character" but the ohm symbol is not. The end result is that NFC doesn't touch the micro sign but changes the ohm symbol to capital omega, while NFKC and NFKD change both the ohm and the micro into other characters.

13. This could lead some to believe that 16 is The Answer to the Ultimate Question of Life, The Universe, and Everything.

convenient intermediate representations for searching and indexing: users may be pleased that a search for `'1/2 inch'` also finds documents containing `'½ inch'`.

NFKC and NFKD normalization should be applied with care and only in special cases — e.g. search and indexing — and not for permanent storage, as these transformations cause data loss.

When preparing text for searching or indexing, another operation is useful: case folding, our next subject.

Case folding

Case folding is essentially converting all text to lowercase, with some additional transformations. It is supported by the `str.casefold()` method (new in Python 3.3).

For any string `s` containing only `latin1` characters, `s.casefold()` produces the same result as `s.lower()`, with only two exceptions: the micro sign `'µ'` is changed to the Greek lower case mu (which looks the same in most fonts) and the German Eszett or "sharp s" (ß) becomes "ss".

```
>>> micro = 'µ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('µ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

As of Python 3.4 there are 116 code points for which `str.casefold()` and `str.lower()` return different results. That's 0.11% of a total of 110,122 named characters in Unicode 6.3.

As usual with anything related to Unicode, case folding is a complicated issue with plenty of linguistic special cases, but the Python core team made an effort to provide a solution that hopefully works for most users.

In the next couple of sections, we'll put our normalization knowledge to use developing utility functions.

Utility functions for normalized text matching

As we've seen, NFC and NFD are safe to use and allow sensible comparisons between Unicode strings. NFC is the best normalized form for most applications. `str.casefold()` is the way to go for case-insensitive comparisons.

If you work with text in many languages, a pair of functions like `nfc_equal` and `fold_equal` in Example 4-13 are useful additions to your toolbox.

Example 4-13. `normeq.py`: normalized Unicode string comparison.

```
"""
Utility functions for normalized Unicode string comparison.

Using Normal Form C, case sensitive:

    >>> s1 = 'café'
    >>> s2 = 'cafe\u0301'
    >>> s1 == s2
    False
    >>> nfc_equal(s1, s2)
    True
    >>> nfc_equal('A', 'a')
    False

Using Normal Form C with case folding:

    >>> s3 = 'Straße'
    >>> s4 = 'strasse'
    >>> s3 == s4
    False
    >>> nfc_equal(s3, s4)
    False
    >>> fold_equal(s3, s4)
    True
    >>> fold_equal(s1, s2)
    True
    >>> fold_equal('A', 'a')
    True

"""

from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

Beyond Unicode normalization and case folding — both part of the Unicode standard — sometimes it makes sense to apply deeper transformations, like changing `'café'` into `'cafe'`. We'll see when and how in the next section.

Extreme "normalization": taking out diacritics

The Google Search secret sauce involves many tricks, but one of them apparently is ignoring diacritics (e.g. accents, cedillas etc.), at least in some contexts. Removing diacritics is not a proper form of normalization because if often changes the meaning of words and may produce false positives when searching. But it helps coping with some facts of life: people sometimes are lazy or ignorant about the correct use of diacritics, and spelling rules change over time, meaning that accents come and go in living languages.

Outside of searching, getting rid of diacritics also makes for more readable URLs, at least in Latin based languages. Take a look at the URL for the English language Wikipedia article about the city of São Paulo:

```
http://en.wikipedia.org/wiki/S%C3%A3o_Paulo
```

The *%C3%A3* part is the URL-escaped, UTF-8 rendering of the single letter "ã" ("a" with tilde). The following is much friendlier, even if it is not the right spelling:

```
http://en.wikipedia.org/wiki/Sao_Paulo
```

To remove all diacritics from a `str`, you can use a function like Example 4-14.

Example 4-14. Function to remove all combining marks (module `sanitize.py`)

```
import unicodedata
import string


def shave_marks(txt):
    """Remove all diacritic marks"""
    norm_txt = unicodedata.normalize('NFD', txt)   ❶
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c))   ❷
    return unicodedata.normalize('NFC', shaved)   ❸
```

❶ Decompose all characters into base characters and combining marks.

❷ Filter out all combining marks.

❸ Recompose all characters.

Example 4-15 shows a couple of uses of `shave_marks`.

Example 4-15. Two examples using `shave_marks` from Example 4-14.

```
>>> order = '“Herr Voß: • ½ cup of Œtker™ caffè latte • bowl of açaí.”'
>>> shave_marks(order)
'“Herr Voß: • ½ cup of Œtker™ caffe latte • bowl of acai.”'  ❶
>>> Greek = 'Zéφυρος, Zéfiro'
>>> shave_marks(Greek)
'Zεφυρος, Zefiro'  ❷
```

❶ Only the letters “è”, “ç” and “í” were replaced.

❷ Both “έ” and “é” were replaced.

The function `shave_marks` from Example 4-14 works all right, but maybe it goes too far. Often the reason to remove diacritics is to change Latin text to pure ASCII, but `shave_marks` also changes non-Latin characters — like Greek letters — which will never become ASCII just by losing their accents. So it makes sense to analyze each base character and to remove attached marks only if the base character is a letter from the Latin alphabet. This is what Example 4-16 does.

Example 4-16. Function to remove combining marks from Latin characters. `import` statements are omitted as this is part of the `sanitize.py` module from Example 4-14.

```
def shave_marks_latin(txt):
    """Remove all diacritic marks from Latin base characters"""
    norm_txt = unicodedata.normalize('NFD', txt)  ❶
    latin_base = False
    keepers = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base:  ❷
            continue  # ignore diacritic on Latin base char
        keepers.append(c)  ❸
        # if it isn't combining char, it's a new base char
        if not unicodedata.combining(c):  ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(keepers)
    return unicodedata.normalize('NFC', shaved)  ❺
```

❶ Decompose all characters into base characters and combining marks.

❷ Skip over combining marks when base character is Latin.

❸ Otherwise, keep current character.

❹ Detect new base character and determine if it's Latin.

❺ Recompose all characters.

An even more radical step would be to replace common symbols in Western texts, like curly quotes, em-dashes, bullets etc. into `ASCII` equivalents. This is what the function `asciize` does in Example 4-17.

Example 4-17. Transform some Western typographical symbols into `ASCII`. This snippet is also part of `sanitize.py` from Example 4-14.

```
single_map = str.maketrans("""‚ƒ„†ˆ‹‘’“”•–—˜›""",    ❶
                           """'f"*^<''""---~>""")

multi_map = str.maketrans({    ❷
    '€': '<euro>',
    '…': '...',
    'Œ': 'OE',
    '™': '(TM)',
    'œ': 'oe',
    '‰': '<per mille>',
    '‡': '**',
})

multi_map.update(single_map)    ❸


def dewinize(txt):
    """Replace Win1252 symbols with ASCII chars or sequences"""
    return txt.translate(multi_map)    ❹


def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt))      ❺
    no_marks = no_marks.replace('ß', 'ss')           ❻
    return unicodedata.normalize('NFKC', no_marks)   ❼
```

❶ Build mapping table for char to char replacement.

❷ Build mapping table for char to string replacement.

❸ Merge mapping tables.

❹ `dewinize` does not affect `ASCII` or `latin1` text, only the Microsoft additions in to `latin1` in `cp1252`.

❺ Apply `dewinize` and remove diacritical marks.

❻ Replace the Eszett with "ss" (we are not using case fold here because we want to preserve the case).

❼ Apply NFKC normalization to compose characters with their compatibility code points.

Example 4-18 shows `asciize` in use.

Example 4-18. Two examples using `asciize` from Example 4-17.

```
>>> order = '“Herr Voß: • ½ cup of Œtker™ caffè latte • bowl of açaí.”'
>>> dewinize(order)
'"Herr Voß: - ½ cup of OEtker(TM) caffè latte - bowl of açaí."'  ❶
```

```
>>> asciize(order)
'"Herr Voss: - 1/2 cup of OEtker(TM) caffe latte - bowl of acai."'  ❷
```

❶ `dewinize` replaces curly quotes, bullets, and ™ (trade mark symbol).

❷ `asciize` applies `dewinize`, drops diacritics and replaces the `'ß'`.

Different languages have their own rules for removing diacritics. For example, Germans change the `'ü'` into `'ue'`. Our `asciize` function is not as refined, so it may or not be suitable for your language. It works acceptably for Portuguese, though.

To summarize, the functions in `sanitize.py` go way beyond standard normalization and perform deep surgery on the text, with a good chance of changing its meaning. Only you can decide whether to go so far, knowing the target language, your users and how the transformed text will be used.

This wraps up our discussion of normalizing Unicode text.

The next Unicode matter to sort out is… sorting.

Sorting Unicode text

Python sorts sequences of any type by comparing the items in each sequence one by one. For strings, this means comparing the code points. Unfortunately, this produces unacceptable results for anyone who uses non-ASCII characters.

Consider sorting a list of fruits grown in Brazil:

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

Sorting rules vary for different locales, but in Portuguese and many languages that use the Latin alphabet, accents and cedillas rarely make a difference when sorting[14]. So "cajá" is sorted as "caja", and must come before "caju".

The sorted `fruits` list should be:

```
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

The standard way to sort non-ASCII text in Python is to use the `locale.strxfrm` function which, according to the `locale` module docs, "transforms a string to one that can be used in locale-aware comparisons".

14. Diacritics affect sorting only in the rare case when they are the only difference between two words — in that case the word with a diacritic is sorted after the plain word.