In [18]:
```
import unicodedata
```

ASCII and Unicode Strings

Python has two types of "strings": byte strings and unicode strings.

In fact, byte strings are used to represent multiple different kinds of data:

- ASCII strings
- arbitrary binary arrays of bytes
- UTF-8 encoded unicode (as well as other encodings)

These distinctions are not marked in the type.
In contrast, Unicode strings are always just strings.

Syntax to note is:

- strings are written as `"..."`
- unicode strings are written as `u"..."`
- non-printable characters can be *escaped* in various ways
  - `\x10` - byte 16 (hexadecimal)
  - `\10` - byte 8 (octal)
  - `\t` `\n` `\r` etc. - special characters
- inside Unicode
  - `\uxxxx` 16 bit unicode character
  - `\Uxxxxxxxx` 32 bit unicode character

In [82]:
```
"\10"
```

Out[82]:
```
'\\d10'
```

In [19]:
```
type("Hello, World")
```

Out[19]:
```
str
```

In [20]:
```
type(u"Hallo, wie gähhhtsch?")
```

Out[20]:
```
unicode
```

The functions `ord` and `unichr` convert individual characters.

In [2]:
```
unichr(77)
```

Out[2]:
```
u'M'
```

In [127]:
```
ord(u"□")
```

Out[127]:
```
12356
```

We refer to the number (integer) of a Unicode character as its *codepoint*.

Unicode is really primarily an assignment of codepoints to characters and their properties.

(The second important part of Unicode is encodings; we look at those below.)

The function `unicode` converts a string to a unicode string.

In [128]:
```
unicode("abc")
```

Out[128]:
```
u'abc'
```

You can use `str` to convert from Unicode to a string, but it won't work for strings that can't be represented in ASCII. You really need a *codec* (coder-decoder); see below.

In [129]:
```
str(u"abc")
```

Out[129]:
```
'abc'
```

In [130]:
```
str(u"äbc")
```

```
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-130-3d6f994274e9> in <module>()
----> 1 str(u"äbc")

UnicodeEncodeError: 'ascii' codec can't encode character u'\xe4' in position 0:
ordinal not in range(128)
```

Note that there is a diference between displaying output and printing it; the former uses the `repr` method, that tries to represent code in an ASCII/source friendly way independent of encoding.

In [123]:
```
print repr(unichr(0x200))
unichr(0x200)
```

```
u'\u0200'
```

Out[123]:
```
u'\u0200'
```

In [124]:
```
print unichr(0x200)
```

```
Ȁ
```

Let's get an impression of what characters exist in Unicode.

In [17]:

```
for i in range(0x0100,0x4000,256):
    print "%4x"%i,"".join([unichr(j) for j in range(i,i+256,7)])
```

```
 100 ĀćĎěĜġĪıĸĿņōŔśŢũŰŷžƅƌƓƚơƨƯƶƽǄǋǒǙǠǧǮǵǼ
 200 ȀȇȎȕȜȣȪȱȸȿɆɍɔɛɢɩɰɷɾʅʌʓʚʡʨʯʶʽ˄ˋ˒˙ˠ˧ˮ˵˼
 300 ̷̨̼̜̣̪̱̘̟̦̭̻̀̇̎̃̊̑͂͆́̕̚͢͡;΅ΌΓΚΡΨίζντϋϒϙϠϧϮϵϼ
 400 ЀЇЎЕМУЪбипцэєћѢѩѰѷѾ҅ҌғҚҡҨүҶҽӄӋӒәӠӧӮӵӼ
 500 ԃԊԑԘԟԦ□ԱԸԿՆՍ՜գժձոտ□֊□□□ְַ֢֩□
 600 □٥□□□□□ۊ□ڵڌټپخذرسشغقف□٢٩□ي□طظ□□□
 700 □܇ܐܛܢܩܰ□□□□□□□□□□□□□□ߕ□□
 800 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
 900 ऀइऎकजणपरसि□ॐॗज़॥३॰□ॾঅঌওচডনযশ়ৃ□ো□□ৠ১৮৵□
 a00 □ਇ□ਕਜਣਪ□ਸਿ□ੁ□ਖ਼□੩□□ੲઅઌઓચડનયશ઼ૃ□ો□□ૠ૧૮□□
 b00 □□□□□□□□□□□□□□□□□□□□□□□□□அ□ஓச□ஙயம□□ொ□□□□கஅஔ□
 c00 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
 d00 □ഇഎകജണപറസി□□□□ൗ□□□ൡ൨൯□ශඍලටඪදබලහ□ා□ෙ□□□□□□
 e00 □งฎดมรสั฿ๆ□๏□□□□□□□□ບມ□ໂ໇□□ ໆ ໐□□□□□□
 f00 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□
1000 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ႷႾჅ□ႽႸ□გჳ
1100 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1200 ሀ□ሎሕሜሣሪሱሸሿቆቍቔቛቢቩተቷቾኅኌ□□ኡከኯ኶ኽዄዋዒዙዠዧዮድዼ
1300 ጃጊ጑ጘጟጦጭጴጻፂፉፐፗ□፥፬፳፺□□□□□□ᎰᎷᎾᏅᏌᏓᏚᏡᏨᏯ□□
1400 □ᐇᐎᐕᐜᐣᐪᐱᐸᐿᑆᑍᑔᑛᑢᑩᑰᑷᑾᒅᒌᒓᒚᒡᒨᒯᒶᒽᓄᓋᓒᓙᓠᓧᓮᓵᓼ
1500 ᔃᔊᔑᔘᔟᔦᔭᔴᔻᕂᕉᕐᕗᕞᕥᕬᕳᕺᖁᖈᖏᖖᖝᖤᖫᖲᖹᗀᗇᗎᗕᗜᗣᗪᗱᗸᗿ
1600 ᘆᘍᘔᘛᘢᘩᘰᘷᘾᙅᙌᙓᙚᙡᙨᙯᙶᙽᚄᚋᚒᚙᚠᚧᚮᚵᚼᛃᛊᛑᛘᛟᛦ᛭ᛴ□□
1700 □□□□□□□ᜣᜪᜱ□□□□□□□□□□□□□□□□□□□□□□□□□□□
1800 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ᣁᣈᣏᣖᣝᣤᣫᣲ□
1900 □□□□□□□□□□□□ᥔᥛᥢᥩᥰ□□□□□□□□□□□□□□□□□□□□
1a00 ᨀᨇᨎᨕ□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1b00 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1c00 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
1d00 ᴀᴇᴎᴕᴜᴣᴱᴸᴿᵆᵍᵔᵛᵢᵩᵰᵷᵾᶅᶌᶓᶚᶡᶨᶯᶶᶽ□□□□□□□□□□
1e00 ḀḇḎḕḜḣḪḱḸḿṆṍṔṛṢṩṰṷṾẅẌẓẚạẨẮặẼỉỎỗỞụỬỳỺ
1f00 ἀἇἎἕ□ἣἪἱἸἿ□ὍὔὛὢὩὰὶ□ᾃᾊᾑᾘᾟᾦᾭᾴ ῂΈῐῗ□ῥῬ□Ὼ
2000  ‐‗‘‥‬‹‼₃‿⁇⁎□⁰⁷⁾₅₌ₓₚ₡₨₯₶□□□□□□
2100 ℀℅ℇℎℕ℞ℤ℥Kℱ⅂⅃ⅅⅆ⅓⅔⅗⅞ⅢⅩⅰⅷⅽↄ□↓↚↡↨↯↶↽⇄⇋⇒⇙⇠⇧⇮⇵⇼
2200 ∀∇∎∕∜∣∪∱∸∿≆≍≔≛≢≩≰≷≾⊅⊌⊓⊚⊡⊨⊯⊶⊽⋄⋋⋒⋙⋠⋧⋮⋵⋼
2300 ⌀⌇⌎⌕⌜⌣□□⌸⌿⍆⍍⍔⍛⍢⍩⍰⍷⍾⎅⎌⎓⎚⎡⎨⎯⎶⎽□□⏙⏠□□□
2400 ␀␇␎␕□□□□□□⑊□□□⑩□□□□□□□□□□□□□ⒶⒽⓄⓋⓒⓙⓠⓧ□□□
2500 ─┇┎┕├┣┪┱┸┿╆╍╔╛╢╩╰╷╾▅▌▓▚□▨▯▶▽◄○◒◙◠◧◮◵◼
2600 ☀☇☎☕☜☣☪☱☸☿♆♍♔♛♢♩♰♷♾⚅⚌⚓⚚⚡⚨⚯⚶⚽□□□□□□□□□□□
2700 □✇✎✕✜✣✪✱✸✿❆❍□❛❢❩❰❷❾➅➌➓➚➡➨➯➶➽⟄⟋⟒⟙⟠⟧⟮⟵⟼
2800 ⠀⠇⠎⠕⠜⠣⠪⠱⠸⠿⡆⡍⡔⡛⡢⡩⡰⡷⡾⢅⢌⢓⢚⢡⢨⢯⢶⢽⣄⣋⣒⣙⣠⣧⣮⣵⣼
2900 ⤀⤇⤎⤕⤜⤣⤪⤱⤸⤿⥆⥍⥔⥛⥢⥩⥰⥷⥾⦅⦌⦓⦚⦡⦨⦯⦶⦽⧄⧋⧒⧙⧠⧧⧮⧵⧼
2a00 ⨀⨇⨎⨕⨜⨣⨪⨱⨸⨿⩆⩍⩔⩛⩢⩩⩰⩷⩾⪅⪌⪓⪚⪡⪨⪯⪶⪽⫄⫋⫒⫙⫠⫧⫮⫵⫼
2b00 ⬀⬇⬎⬕⬜⬣⬪⬱⬸⬿⭆⭍□⭛□□□□□□□□□□□□□□□□□□□□□□□□
2c00 ⰀⰇⰎⰕⰜⰣⰪⰱⰸⰿⱆⱍⱔⱛ□ⱩⱰⱷⱾⲅⲌⲓⲚⲡⲨⲯⲶⲽ□□□ⳙ□□□ⳮ□□□
2d00 ⴀⴇⴎⴕⴜⴣ□ⴰⴷⴾⵅⵌⵓⵚⵡ□□□□□□□□□□□□□□□□□□□□□ⷠ
2e00 □□□□□□⸪□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
2f00 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3000 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3100 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3200 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3300 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3400 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3500 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3600 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3700 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3800 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3900 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
3a00 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

Encoding and Decoding

There are three kind of "strings" you have to think about:

- ASCII - the traditional US-English character set, used for most programming
  - 128 code points
  - 7 bits per character
- Unicode - the full character set representing all characters in the world
  - 110000 characters in 100 scripts
- UTF-8 - an encoding of Unicode
  - encoded Unicode with 8 bits per character

Important property:

The UTF-8, ASCII, and Unicode are all "the same" if all the characters are in the ASCII character set.

In Python, encoding and decoding is performed via the `encode` and `decode` methods. They take a *codec* name as an argument (`ascii` or `utf-8` are the only ones that are relevant to us), plus an optional argument saying what should happen if a string is not de/encodable.

```
In [61]: u"abc".encode("ascii")
```

```
Out[61]: 'abc'
```

```
In [66]: u"äbc".encode("ascii","replace")
```

```
Out[66]: '?bc'
```

Let's look at a non-ASCII character. As you can see here, the German umlaut "ä" turns into a two character sequence when encoded in UTF-8. Each character has the high bit set. You can look up the exact encoding scheme online.

```
In [67]: u"äbc".encode("utf-8")
```

```
Out[67]: '\xc3\xa4bc'
```

Some more unusual characters are encoded as four byte sequences in UTF-8.

```
In [68]: u"□".encode("utf-8")
```

```
Out[68]: '\xf0\x9d\x8d\xa2'
```

Note that although four bytes (i.e. 32 bits) are used for encoding this character, its codepoint is only 119650. That's because only a few bits are used from each 8 bit code.

```
In [83]: ord(u"□")
```

```
Out[83]: 119650
```

Of course, we can also decode.

In [86]:
```
print '\xc3\xa4bc'.decode("utf-8")
```

```
äbc
```

You get an error message if the decoding is not possible.

In [87]:
```
print '\xc3\xa4bc'.decode("ascii")
```

```
---------------------------------------------------------------------------
UnicodeDecodeError                        Traceback (most recent call last)
<ipython-input-87-d1315b67a57b> in <module>()
----> 1 print '\xc3\xa4bc'.decode("ascii")

UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 0: ordinal
not in range(128)
```

File I/O

Standard file descriptors in Python cannot encode/decode UTF-8.

In [73]:
```
with open("temp","w") as stream: stream.write(u"Käse und Brot")
```

```
---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-73-3a350ad9897f> in <module>()
----> 1 with open("temp","w") as stream: stream.write(u"Käse und Brot")

UnicodeEncodeError: 'ascii' codec can't encode character u'\xe4' in position 1:
ordinal not in range(128)
```

To read and write Unicode, use the `codecs.open` function. It returns a standard file object, but it does the right kind of encoding/decoding for UTF-8.

In [76]:
```
import codecs
with codecs.open("temp","w","utf-8") as stream: stream.write(u"Käse und Brot")
```

In [77]:
```
with codecs.open("temp","r","utf-8") as stream: print stream.read()
```

```
Käse und Brot
```

In [78]:
```
!cat -v temp
```

```
KM-CM-$se und Brot
```

Unicode Data

The Unicode consortium defines a lot of information associated with each codepoint. In Python, you can query this

information using the `unicodedata` library.

In [ ]:
```
import unicodedata
```

Information about each codepoint includes:

- the full name of the character
- the block it is from
- a two letter category (e.g., "Ll" for Letter, lower case)
- whether it is a combining character
- the writing direction (BIDI)
- what it decomposes into, if anything
- whether it is a mirror of another character
- the version of Unicode that defines it
- additional information about some characters, like the numerical value of digits

In [110]:
```
unicodedata.name(u"ä")
```

Out[110]:
```
'LATIN SMALL LETTER A WITH
DIAERESIS'
```

In [115]:
```
unicodedata.category(u"ä")
```

Out[115]:
```
'Ll'
```

In [111]:
```
ord(u"ä")
```

Out[111]:
```
228
```

In [112]:
```
unicodedata.decimal(u"3")
```

Out[112]:
```
3
```

In [113]:
```
unicodedata.numeric(u"□")
```

Out[113]:
```
4.0
```

In [114]:
```
unicodedata.category(u"ß")
```

Out[114]:
```
'Ll'
```

Decomposition and Normalization

Various letters are really just combined forms of separate parts.

For example, the letter "ä" can be viewed as a combination of the letter "a" with the diacritic "¨".

The Unicode consortium hasn't been consistent about how to represent these, so the same letter as it appears on the screen can be represented in two different ways.

In [120]:
```
print u'\u00e4'
print u'a\u0308'
```

```
ä
ä
```

Although these strings look the same, they are represented differently.

In [121]:
```
u'\u00e4'==u'a\u0308'
```

Out[121]:
```
False
```

Unicodedata can decompose characters.

In [30]:
```
unicodedata.decomposition(u"ä")
```

Out[30]:
```
'0061 0308'
```

More generally, it can normalize a string into one of four forms:

- NFD - decomposed by canonical equivalence, combining characters arranged in specific order
- NFC - decomposed and then recomposed by canonical equivalence
- NFKD - decomposed by compatibility, combining characters arranged in specific order
- NFKC - decomposed by compatibility, then recomposed by canonical equivalence

What does this mean?

- canonical equivalence: same appearance and same meaning when printed
- compatible: distinct appearance but usually the same meaning

For example, "ff" as a ligature is compatible with the two letters "ff", but not canonically equivalent (since they look different).

- NFC - best compatibility with conversions from legacy encodings
- NFKC - preferred for identifiers, best security
- NFD - easier to process
- NFKD - easier to process

Yes, unfortunately, you do need to worry about this.

In [37]:
```
for n in ["NFC","NFKC","NFD","NFKD"]:
    s = unicodedata.normalize(n,u"ä")
    print n,repr(s),s
```

```
NFC u'\xe4' ä
NFKC u'\xe4' ä
NFD u'a\u0308' ä
NFKD u'a\u0308' ä
```

In [106]:
```
print u"r\u0308"
```

```
r̈
```

In [107]:
```
print u"+\u0308"
```

```
+¨
```

In [108]:
```
print u"\u0308"
```
¨

In [109]:
```
print u" \u0308"
```
¨

Ligatures

Many languages have ligatures. In some languages and scripts (e.g., German), ligatures like "ä" and "ß" have become letters in their own right. In other scripts, ligatures are just different presentations depending on the context a character appears in; in those cases, Unicode does not represent ligatures as separate code points.

Here is an example in Arabic. Note how the string looks very different when printed a character at a time vs. when printed as a word (also note that Arabic is a right-to-left language):

In [91]:
```
s = u"كتاب"
```

In [92]:
```
print s
```
كتاب

In [95]:
```
for c in s: print c,
print
```
ك ت ا ب

In contrast, the "ffi" ligature in English has its own Unicode codepoint.

In [96]:
```
s = u"a\ufb03ne"
```

In [97]:
```
print s
```
affine

In [98]:
```
for c in s: print c,
print
```
a ffi n e

In [102]:
```
s==u"affine"
```
Out[102]: False

In [103]:
```
unicodedata.normalize("NFKD",s)
```
Out[103]: u'affine'

In [122]:
```
unicodedata.normalize("NFKD",s)==unicodedata.normalize("NFKD",u"affine")
```

Out[122]: True

In [ ]:
```

```