Unicode and Byte Strings

Overview

- Many programmers, including most beginners, deal with simple forms of text like ASCII, they can happily work with Python's basic str string type and its associated operations and don't need to come to grips with more advanced string concepts.
- In fact, such programmers can often ignore the string changes in Python 3.X and continue to use strings as they may have in the past.
- On the other hand, many other programmers deal with more specialized types of data: non-ASCII character sets, image file contents, and so on.
- For those programmers, and others who may someday join them, in this module we're going to fill in the rest of the Python string story and look at some more advanced concepts in Python's string model.

String Changes in 3.X

- One of the most noticeable changes in the Python 3.X line is the **mutation** of string object types.
- In a nutshell, 2.X's **str** and **unicode** types have morphed into 3.X's **bytes** and **str** types, and a new mutable **bytearray** type has been added.
- The bytearray type is technically available in Python 2.6 and 2.7 too (though not earlier), but it's a back-port from 3.X and does not as clearly distinguish between text and binary content in 2.X.
- Especially if you process data that is either Unicode or binary in nature, these changes can have substantial impacts on your code.

Who's affected?

- If you deal with non-ASCII Unicode text—for instance, in the context of internationalized domains like the Web, or the results of some XML and JSON parsers and databases - you will find support for text encodings to be different in 3.X, but also probably more direct, accessible, and seamless than in 2.X.
- If you deal with binary data - for example, in the form of image or audio files or packed data processed with the struct module - you will need to understand 3.X's new bytes object and 3.X's different and sharper distinction between text and binary data and files.

Who's affected?

- If you fall into neither of the prior two categories, you can generally use strings in 3.X much as you would in 2.X, with the general str string type, text files, and all the familiar string operations we studied earlier. Your strings will be encoded and decoded by 3.X using your platform's default encoding (e.g., ASCII, or UTF-8 on Windows in the U.S. - sys.getdefaultencoding gives your default if you care to check), but you probably won't notice.

In other words, if your text is always ASCII, you can get by with normal string objects and text files and can avoid most of the following story for now.

Character Encoding Schemes - ASCII

- Character sets are standards that assign integer codes to individual characters so they can be represented in computer memory.
- The ASCII standard, for example, was created in the U.S., and it defines many U.S. programmers' notion of text strings.
- ASCII defines character codes from 0 through 127 and allows each character to be stored in one 8-bit byte, only 7 bits of which are used.
- For example, the ASCII standard maps the character 'a' to the integer value 97 (0x61 in hex), which can be stored in a single byte in memory and files.

```
>>> ord('a')          # 'a' is a byte with binary value 97 in ASCII (and others)
97
>>> hex(97)
'0x61'
>>> chr(97)           # Binary value 97 stands for character 'a'
'a'
```

Character Encoding Schemes – Latin-1

- Sometimes one byte per character isn't enough, though. Various symbols and accented characters, for instance, do not fit into the range of possible characters defined by ASCII.
- To accommodate special characters, some standards use all the possible values in an 8-bit byte, 0 through 255, to represent characters, and assign the values 128 through 255 (outside ASCII's range) to special characters.
- One such standard, known as the Latin-1 character set, is widely used in Western Europe. In Latin-1, character codes above 127 are assigned to accented and otherwise special characters.

```
>>> 0xC4
196
>>> chr(196)                    # Python 3.X result form shown
'Ä'
```

Character Encoding Schemes - Unicode

- Still, some alphabets define so many characters that it is impossible to represent each of them as one byte.
- Unicode allows more flexibility.
- Unicode text is sometimes referred to as “wide-character” strings, because characters may be represented with multiple bytes if needed.
- Unicode is typically used in internationalized programs, to represent European, Asian, and other non-English character sets that have more characters than 8-bit bytes can represent.

Character Encoding Schemes - Unicode

- To store such rich text in computer memory, we say that characters are translated to and from raw bytes using an encoding - the rules for translating a string of Unicode characters to a sequence of bytes and extracting a string from a sequence of bytes.
- Encoding is the process of translating a string of characters into its raw bytes form, according to a desired encoding name.
- Decoding is the process of translating a raw string of bytes into its character string form, according to its encoding name.

Character Encoding Schemes - Unicode

- That is, we encode from string to raw bytes, and decode from raw bytes to string.
- To scripts, decoded strings are just characters in memory, but may be encoded into a variety of byte string representations when stored on files, transferred over networks, embedded in documents and databases, and so on.
- For some encodings, the translation process is trivial - ASCII and Latin-1, for instance, map each character to a fixed-size single byte, so no translation work is required.
- For other encodings, the mapping can be more complex and yield multiple bytes per character, even for simple 8-bit forms of text.

Character Encoding Schemes - Unicode

- The widely used UTF-8 encoding, for example, allows a wide range of characters to be represented by employing a variable-sized number of bytes scheme.
- Character codes less than 128 are represented as a single byte; codes between 128 and 0x7ff (2047) are turned into 2 bytes, where each byte has a value between 128 and 255; and codes above 0x7ff are turned into 3- or 4-byte sequences having values between 128 and 255.
- This keeps simple ASCII strings compact, sidesteps byte ordering issues, and avoids null (zero value) bytes that can cause problems for C libraries and networking.

Character Encoding Schemes

- Because their encodings' character maps assign characters to the same codes for compatibility, ASCII is a subset of both Latin-1 and UTF-8.
- That is, a valid ASCII character string is also a valid Latin-1- and UTF-8-encoded string.
- For example, every ASCII file is a valid UTF-8 file, because the ASCII character set is a 7-bit subset of UTF-8.

Character Encoding Schemes

- Conversely, the UTF-8 encoding is binary compatible with ASCII, but only for character codes less than 128.
- Latin-1 and UTF-8 simply allow for additional characters: Latin-1 for characters mapped to values 128 through 255 within a byte, and UTF-8 for characters that may be represented with multiple bytes.

Character Encoding Schemes

- Other encodings allow for richer character sets in different ways.
- UTF-16 and UTF-32, for example, format text with a fixed-size 2 and 4 bytes per each character scheme, respectively, even for characters that could otherwise fit in a single byte.
- Some encodings may also insert prefixes that identify byte ordering.

```
>>> S = 'ni'
>>> S.encode('ascii'), S.encode('latin1'), S.encode('utf8')
(b'ni', b'ni', b'ni')

>>> S.encode('utf16'), len(S.encode('utf16'))
(b'\xff\xfen\x00i\x00', 6)

>>> S.encode('utf32'), len(S.encode('utf32'))
(b'\xff\xfe\x00\x00n\x00\x00\x00i\x00\x00\x00', 12)
```

Character Encoding Schemes

- To Python programmers, encodings are specified as strings containing the encoding's name.
- Python comes with roughly 100 different encodings; see the Python library reference for a complete list.
- Importing the module encodings and running help(encodings) shows you many encoding names as well; some are implemented in Python, and some in C.
- Some encodings have multiple names, too; for example, latin-1, iso_8859_1, and 8859 are all synonyms for the same encoding, Latin-1.

How Python Stores Strings in Memory

- In memory, Python always stores decoded text strings in an encoding-neutral format, which may or may not use multiple bytes for each character.
- All text processing occurs in this uniform internal format.
- Text is translated to and from an encoding-specific format only when it is transferred to or from external text files, byte strings, or APIs with specific encoding requirements.
- Once in memory, though, strings have no encoding.
- They are just the string object.

Python 3.2 and earlier

- Through Python 3.2, strings are stored internally in fixed-length UTF-16 (roughly, UCS-2) format with 2 bytes per character, unless Python is configured to use 4 bytes per character (UCS-4).

Python 3.3 and later

- Python 3.3 and later instead use a variable-length scheme with 1, 2, or 4 bytes per character, depending on a string's content.
- The size is chosen based upon the character with the largest Unicode ordinal value in the represented string.
- This scheme allows a space-efficient representation in common cases, but also allows for full UCS-4 on all platforms.

Python's String Types

- Python 2.X has a general string type for representing binary data and simple 8-bit text like ASCII, along with a specific type for representing richer Unicode text:
  - str for representing 8-bit text and binary data
  - unicode for representing decoded Unicode text
- Python 3.X comes with three string object types - one for textual data and two for binary data:
  - str for representing decoded Unicode text (including ASCII
  - bytes for representing binary data (including encoded text)
  - bytearray, a mutable flavor of the bytes type

Why the different string types?

- All three string types in 3.X support similar operation sets, but they have different roles.
- The main goal behind this change in 3.X was to merge the normal and Unicode string types of 2.X into a single string type that supports both simple and Unicode text: developers wanted to remove the 2.X string dichotomy and make Unicode processing more natural.
- Given that ASCII and other 8-bit text is really a simple kind of Unicode, this convergence seems logically sound.

How it’s being done?

- To achieve this, 3.X stores text in a redefined str type - an immutable sequence of characters (not necessarily bytes), which may contain either simple text such as ASCII whose character values fit in single bytes, or richer character set text such as UTF-8 whose character values may require multiple bytes.
- Strings processed by your script with this type are stored generically in memory and are encoded to and decoded from byte strings per either the platform Unicode default or an explicit encoding name.
- This allows scripts to translate text to different encoding schemes, both in memory and when transferring to and from files.

Binary data

- While 3.X's new str type does achieve the desired string/unicode merging, many programs still need to process raw binary data that is not encoded per any text format.
- Image and audio files, as well as packed data used to interface with devices or C programs you might process with Python's struct module, fall into this category.
- Because Unicode strings are decoded from bytes, they cannot be used to represent bytes.

Binary data

- To support processing of such truly binary data, a new string type, bytes, also was introduced - an immutable sequence of 8-bit integers representing absolute byte values, which prints as ASCII characters when possible.
- Though a distinct object type, bytes supports almost all the same operations that the str type does; this includes string methods, sequence operations, and even re module pattern matching, but not string formatting.
- In 2.X, the general str type fills this binary data role, because its strings are just sequences of bytes; the separate unicode type handles richer text strings.

Binary data

- In more detail, a 3.X bytes object really is a sequence of small integers, each of which is in the range 0 through 255; indexing a bytes returns an int, slicing one returns another bytes, and running the list built-in on one returns a list of integers, not characters.
- When processed with operations that assume characters, though, the contents of bytes objects are assumed to be ASCII-encoded bytes (e.g., the isalpha method assumes each byte is an ASCII character code).
- Further, bytes objects are printed as character strings instead of integers for convenience.

bytearray

- While they were at it, Python developers also added a bytearray type in 3.X. bytearray is a variant of bytes that is mutable and so supports in-place changes.
- It supports the usual string operations that str and bytes do, as well as many of the same in-place change operations as lists (e.g., the append and extend methods, and assignment to indexes).
- This can be useful both for truly binary data and simple types of text. Assuming your text strings can be treated as raw 8-bit bytes (e.g., ASCII or Latin-1 text), bytearray finally adds direct in-place mutability for text data - something not possible without conversion to a mutable type in Python 2.X, and not supported by Python 3.X's str or bytes.

Python 2.x and 3.x

- Although Python 2.X and 3.X offer much the same functionality, they package it differently.
- In fact, the mapping from 2.X to 3.X string types is not completely direct - 2.X's str equates to both str and bytes in 3.X, and 3.X's str equates to both str and unicode in 2.X.
- Moreover, the mutability of 3.X's bytearray is unique.

Text Files

- When a file is opened in text mode, reading its data automatically decodes its content and returns it as a str; writing takes a str and automatically encodes it before transferring it to the file.
- Both reads and writes translate per a platform default or a provided encoding name.
- Text-mode files also support universal end-of-line translation and additional encoding specification arguments.
- Depending on the encoding name, text files may also automatically process the byte order mark sequence at the start of a file (more on this momentarily).

Binary Files

- When a file is opened in binary mode by adding a b (lowercase only) to the mode-string argument in the built-in open call, reading its data does not decode it in any way but simply returns its content raw and unchanged, as a bytes object; writing similarly takes a bytes object and transfers it to the file unchanged.
- Binary-mode files also accept a bytearray object for the content to be written to the file.

Text or Binary?

- If you are processing image files, data transferred over networks, packed binary data whose content you must extract, or some device data streams, chances are good that you will want to deal with it using bytes and binary-mode files.
- You might also opt for bytearray if you wish to update the data without making copies of it in memory.
- If instead you are processing something that is textual in nature, such as program output, HTML, email content, or CSV or XML files, you'll probably want to use str and text-mode files.

Python 3.X String Literals

- Python 3.X string objects originate when you call a built-in function such as str or bytes, read a file created by calling open (described in the next section), or code literal syntax in your script.
- For the latter, a new literal form, b'xxx' (and equivalently, B'xxx') is used to create bytes objects in 3.X, and you may create bytearray objects by calling the bytearray function, with a variety of possible arguments.

```
C:\code> C:\python33\python
>>> B = b'spam'                  # 3.X bytes literal make a bytes object (8-bit bytes)
>>> S = 'eggs'                   # 3.X str literal makes a Unicode text string

>>> type(B), type(S)
(<class 'bytes'>, <class 'str'>)

>>> B                            # bytes: sequence of int, prints as character string
b'spam'
>>> S
'eggs'
```

The 3.X **bytes** object is actually a sequence of short integers, though it prints its content as characters whenever possible:

```
>>> B[0], S[0]                          # Indexing returns an int for bytes, str for str
(115, 'e')
>>> B[1:], S[1:]                        # Slicing makes another bytes or str object
(b'pam', 'ggs')
>>> list(B), list(S)
([115, 112, 97, 109], ['e', 'g', 'g', 's'])      # bytes is really 8-bit small ints
```

The bytes object is also immutable, just like str (though bytearray, described later, is not); you cannot assign a str, bytes, or integer to an offset of a bytes object.

```
>>> B[0] = 'x'                                  # Both are immutable
TypeError: 'bytes' object does not support item assignment
>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment
```

Finally, note that the `bytes` literal's `b` or `B` prefix also works for any string literal form, including triple-quoted blocks, though you get back a string of raw bytes that may or may not map to characters:

```
>>> # bytes prefix works on single, double, triple quotes, raw
>>> B = B"""
... xxxx
... yyyy
... """
...
>>> B
b'\nxxxx\nyyyy\n'
```

String Type Conversions

- Although Python 2.X allowed str and unicode type objects to be mixed in expressions (when the str contained only 7-bit ASCII text), 3.X draws a much sharper distinction - str and bytes type objects never mix automatically in expressions and never are converted to one another automatically when passed to functions.
- A function that expects an argument to be a str object won't generally accept a bytes, and vice versa.

String Type Conversions

- Because of this, Python 3.X basically requires that you commit to one type or the other, or perform manual, explicit conversions when needed:
  - str.encode() and bytes(S, encoding) translate a string to its raw bytes form and create an encoded bytes from a decoded str in the process.
  - bytes.decode() and str(B, encoding) translate raw bytes into its string form and create a decoded str from an encoded bytes in the process.

```
>>> S = 'eggs'
>>> S.encode()                          # str->bytes: encode text into raw bytes
b'eggs'
>>> bytes(S, encoding='ascii')          # str->bytes, alternative
b'eggs'

>>> B = b'spam'
>>> B.decode()                          # bytes->str: decode raw bytes into text
'spam'
>>> str(B, encoding='ascii')            # bytes->str, alternative
'spam'
```

```
>>> import sys
>>> sys.platform                          # Underlying platform
'win32'
>>> sys.getdefaultencoding()              # Default encoding for str here
'utf-8'

>>> bytes(S)
TypeError: string argument without an encoding

>>> str(B)                                # str without encoding
"b'spam'"                                 # A print string, not conversion!
>>> len(str(B))
7
>>> len(str(B, encoding='ascii'))         # Use encoding to convert to str
4
```

Coding ASCII Text

```
C:\code> C:\python33\python
>>> ord('X')                # 'X' is binary code point value 88 in the default encoding

88
>>> chr(88)                 # 88 stands for character 'X'
'X'

>>> S = 'XYZ'               # A Unicode string of ASCII text
>>> S
'XYZ'
>>> len(S)                  # Three characters long
3
>>> [ord(c) for c in S]     # Three characters with integer ordinal values
[88, 89, 90]
```

Coding Non-ASCII Text

```
>>> chr(0xc4)                  # 0xC4, 0xE8: characters outside ASCII's range
'Ä'
>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'             # Single 8-bit value hex escapes: two digits
>>> S
'Äè'

>>> S = '\u00c4\u00e8'         # 16-bit Unicode escapes: four digits each
>>> S
'Äè'
>>> len(S)                     # Two characters long (not number of bytes!)
2
```

Encoding and Decoding Non-ASCII text

```
>>> S = '\u00c4\u00e8'                    # Non-ASCII text string, two characters long
>>> S
'Äè'
>>> len(S)
2

>>> S.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1:
ordinal not in range(128)
```

Encoding and Decoding Non-ASCII text

```
>>> S.encode('latin-1')                    # 1 byte per character when encoded
b'\xc4\xe8'

>>> S.encode('utf-8')                      # 2 bytes per character when encoded
b'\xc3\x84\xc3\xa8'

>>> len(S.encode('latin-1'))               # 2 bytes in latin-1, 4 in utf-8
2
>>> len(S.encode('utf-8'))
4
```

Encoding and Decoding Non-ASCII text

```
>>> B = b'\xc4\xe8'                    # Text encoded per Latin-1
>>> B
b'\xc4\xe8'
>>> len(B)                             # 2 raw bytes, two encoded characters
2
>>> B.decode('latin-1')                # Decode to text per Latin-1
'Äè'

>>> B = b'\xc3\x84\xc3\xa8'            # Text encoded per UTF-8
>>> len(B)                             # 4 raw bytes, two encoded characters
4
>>> B.decode('utf-8')                  # Decode to text per UTF-8
'Äè'
>>> len(B.decode('utf-8'))             # Two Unicode characters in memory
2
```

Byte String Literals: Encoded Text

- Two cautions here too.
- First, Python 3.X allows special characters to be coded with both hex and Unicode escapes in str strings, but only with hex escapes in bytes strings - Unicode escape sequences are silently taken verbatim in bytes literals, not as escapes. In fact, bytes must be decoded to str strings to print their non-ASCII characters properly.
- Second, bytes literals require characters either to be ASCII characters or, if their values are greater than 127, to be escaped; str stings, on the other hand, allow literals containing any character in the source character set - which, defaults to UTF-8 unless an encoding declaration is given in the source file.

```
>>> S = 'A\xC4B\xE8C'                    # 3.X: str recognizes hex and Unicode escapes
>>> S
'AÄBèC'
>>> S = 'A\u00C4B\U000000E8C'
>>> S
'AÄBèC'

>>> B = b'A\xC4B\xE8C'                   # bytes recognizes hex but not Unicode
>>> B
b'A\xc4B\xe8C'
>>> B = b'A\u00C4B\U000000E8C'           # Escape sequences taken literally!
>>> B
b'A\\u00C4B\\U000000E8C'

>>> B = b'A\xC4B\xE8C'                   # Use hex escapes for bytes
>>> B                                    # Prints non-ASCII as hex
b'A\xc4B\xe8C'
>>> print(B)
b'A\xc4B\xe8C'
>>> B.decode('latin-1')                  # Decode as latin-1 to interpret as text
'AÄBèC'
```

```
>>> S = 'AÄBèC'                          # Chars from UTF-8 if no encoding declaration
>>> S
'AÄBèC'

>>> B = b'AÄBèC'
SyntaxError: bytes can only contain ASCII literal characters.

>>> B = b'A\xC4B\xE8C'                   # Chars must be ASCII, or escapes
>>> B
b'A\xc4B\xe8C'
>>> B.decode('latin-1')
'AÄBèC'

>>> S.encode()                           # Source code encoded per UTF-8 by default
b'A\xc3\x84B\xc3\xa8C'                   # Uses system default to encode, unless passed
>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\xa8C'

>>> B.decode()                           # Raw bytes do not correspond to utf-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: ...
```

Converting Encodings

```
>>> B = b'A\xc3\x84B\xc3\xa8C'          # Text encoded in UTF-8 format originally
>>> S = B.decode('utf-8')               # Decode to Unicode text per UTF-8
>>> S
'AÄBèC'

>>> T = S.encode('cp500')               # Convert to encoded bytes per EBCDIC
>>> T
b'\xc1c\xc2T\xc3'

>>> U = T.decode('cp500')               # Convert back to Unicode per EBCDIC
>>> U
'AÄBèC'

>>> U.encode()                          # Per default utf-8 encoding again
b'A\xc3\x84B\xc3\xa8C'
```

Source File Character Set Encoding Declarations

- To interpret the content of strings you code and hence embed within the text of your script files, Python uses the UTF-8 encoding by default, but it allows you to change this to support arbitrary character sets by including a comment that names your desired encoding.
- The comment must be of this form and must appear as either the first or second line in your script in either Python 2.X or 3.X

```
# -*- coding: latin-1 -*-
```

```
# -*- coding: latin-1 -*-
# Any of the following string literal forms work in latin-1.
# Changing the encoding above to either ascii or utf-8 fails,
# because the 0xc4 and 0xe8 in myStr1 are not valid in either.

myStr1 = 'aÄBèC'

myStr2 = 'A\u00c4B\U000000e8C'

myStr3 = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'

import sys
print('Default encoding:', sys.getdefaultencoding())

for aStr in myStr1, myStr2, myStr3:
    print('{0}, strlen={1}, '.format(aStr, len(aStr)), end='')

    bytes1 = aStr.encode()              # Per default utf-8: 2 bytes for non-ASCII
    bytes2 = aStr.encode('latin-1')     # One byte per char
   #bytes3 = aStr.encode('ascii')       # ASCII fails: outside 0..127 range

    print('byteslen1={0}, byteslen2={1}'.format(len(bytes1), len(bytes2)))
```

```
C:\code> C:\python33\python text.py
Default encoding: utf-8
aÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
```

Using 3.X bytes Objects

- The 3.X bytes object is a sequence of small integers, each of which is in the range 0 through 255, that happens to print as ASCII characters when displayed.
- It supports sequence operations and most of the same methods available on str objects (and present in 2.X's str type).
- However, bytes does not support the for mat method or the % formatting expression, and you cannot mix and match bytes and str type objects without explicit conversions - you generally will use all str type objects and text files for text data, and all bytes type objects and binary files for binary data.

Method Calls

```
C:\code> C:\python33\python

# Attributes in str but not bytes
>>> set(dir('abc')) - set(dir(b'abc'))
{'isdecimal', '__mod__', '__rmod__', 'format_map', 'isprintable',
'casefold', 'format', 'isnumeric', 'isidentifier', 'encode'}

# Attributes in bytes but not str
>>> set(dir(b'abc')) - set(dir('abc'))
{'decode', 'fromhex'}
```

```
>>> B = b'spam'                      # b'...' bytes literal
>>> B.find(b'pa')
1

>>> B.replace(b'pa', b'XY')          # bytes methods expect bytes arguments
b'sXYm'

>>> B.split(b'pa')                   # bytes methods return bytes results
[b's', b'm']

>>> B
b'spam'
>>> B[0] = 'x'
TypeError: 'bytes' object does not support item assignment
```

```
>>> '%s' % 99
'99'
>>> b'%s' % 99
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'

>>> '{0}'.format(99)
'99'
>>> b'{0}'.format(99)
AttributeError: 'bytes' object has no attribute 'format'
```

Sequence Operations

```
>>> B = b'spam'                  # A sequence of small ints
>>> B                            # Prints as ASCII characters (and/or hex escapes)
b'spam'

>>> B[0]                         # Indexing yields an int
115
>>> B[-1]
109

>>> chr(B[0])                    # Show character for int
's'


>>> list(B)                      # Show all the byte's int values
[115, 112, 97, 109]

>>> B[1:], B[:-1]
(b'pam', b'spa')
>>> len(B)
4
>>> B + b'lmn'
b'spamlmn'
>>> B * 4
b'spamspamspamspam'
```

Other Ways to Make bytes Objects

```
>>> B = b'abc'                      # Literal
>>> B
b'abc'

>>> B = bytes('abc', 'ascii')       # Constructor with encoding name
>>> B
b'abc'

>>> ord('a')
97
>>> B = bytes([97, 98, 99])         # Integer iterable
>>> B
b'abc'

>>> B = 'spam'.encode()             # str.encode() (or bytes())
>>> B
b'spam'
>>>
>>> S = B.decode()                  # bytes.decode() (or str())
>>> S
'spam'
```

Mixing String Types

```
# Must pass expected types to function and method calls

>>> B = b'spam'

>>> B.replace('pa', 'XY')
TypeError: expected an object with the buffer interface

>>> B.replace(b'pa', b'XY')
b'sXYm'

>>> B = B'spam'
>>> B.replace(bytes('pa'), bytes('xy'))
TypeError: string argument without an encoding

>>> B.replace(bytes('pa', 'ascii'), bytes('xy', 'utf-8'))
b'sxym'
```

Mixing String Types

```
# Must convert manually in 3.X mixed-type expressions

>>> b'ab' + 'cd'
TypeError: can't concat bytes to str

>>> b'ab'.decode() + 'cd'                    # bytes to str
'abcd'
>>> b'ab' + 'cd'.encode()                    # str to bytes
b'abcd'
>>> b'ab' + bytes('cd', 'ascii')             # str to bytes
b'abcd'
```

Using 3.X/2.6+ bytearray Objects

- Python 3.X grew a third string type, though - bytearray, a mutable sequence of integers in the range 0 through 255, which is a mutable variant of bytes.
- As such, it supports the same string methods and sequence operations as bytes, as well as many of the mutable in-place-change operations supported by lists.
- Bytearrays support in-place changes to both truly binary data as well as simple forms of text such as ASCII, which can be represented with 1 byte per character (richer Unicode text generally requires Unicode strings, which are still immutable).
- The bytear ray type is also available in Python 2.6 and 2.7 as a back-port from 3.X, but it does not enforce the strict text/binary distinction there that it does in 3.X.

bytearrays in Action

```
# Creation in 2.6/2.7: a mutable sequence of small (0..255) ints

>>> S = 'spam'
>>> C = bytearray(S)                    # A back-port from 3.X in 2.6+
>>> C                                   # b'..' == '..' in 2.6+ (str)
bytearray(b'spam')
```

bytearrays in Action

```
# Creation in 3.X: text/binary do not mix

>>> S = 'spam'
>>> C = bytearray(S)
TypeError: string argument without an encoding

>>> C = bytearray(S, 'latin1')          # A content-specific type in 3.X
>>> C
bytearray(b'spam')

>>> B = b'spam'                         # b'..' != '..' in 3.X (bytes/str)
>>> C = bytearray(B)
>>> C
bytearray(b'spam')
```

bytearrays in Action

```
# Mutable, but must assign ints, not strings

>>> C[0]
115

>>> C[0] = 'x'                          # This and the next work in 2.6/2.7
TypeError: an integer is required
>>> C[0] = b'x'
TypeError: an integer is required

>>> C[0] = ord('x')                     # Use ord() to get a character's ordinal
>>> C

bytearray(b'xpam')

>>> C[1] = b'Y'[0]                      # Or index a byte string
>>> C
bytearray(b'xYam')
```

byte vs bytearray

```
# in bytes but not bytearray
>>> set(dir(b'abc')) - set(dir(bytearray(b'abc')))
{'__getnewargs__'}

# in bytearray but not bytes
>>> set(dir(bytearray(b'abc'))) - set(dir(b'abc'))
{'__iadd__', 'reverse', '__setitem__', 'extend', 'copy', '__alloc__',
'__delitem__', '__imul__', 'remove', 'clear', 'insert', 'append', 'pop'}
```

Text File Basics

```
C:\code> C:\python33\python
# Basic text files (and strings) work the same as in 2.X

>>> file = open('temp', 'w')
>>> size = file.write('abc\n')          # Returns number of characters written
>>> file.close()                        # Manual close to flush output buffer

>>> file = open('temp')                 # Default mode is "r" (== "rt"): text input
>>> text = file.read()
>>> text
'abc\n'
>>> print(text)
abc
```

Text and Binary Modes

```
C:\code> C:\python27\python
>>> open('temp', 'w').write('abd\n')          # Write in text mode: adds \r
>>> open('temp', 'r').read()                  # Read in text mode: drops \r
'abd\n'
>>> open('temp', 'rb').read()                 # Read in binary mode: verbatim
'abd\r\n'

>>> open('temp', 'wb').write('abc\n')         # Write in binary mode
>>> open('temp', 'r').read()                  # \n not expanded to \r\n
'abc\n'
>>> open('temp', 'rb').read()
'abc\n'
```

Text and Binary Modes

```
C:\code> C:\python33\python
# Write and read a text file
>>> open('temp', 'w').write('abc\n')          # Text mode output, provide a str
4
>>> open('temp', 'r').read()                  # Text mode input, returns a str
'abc\n'
>>> open('temp', 'rb').read()                 # Binary mode input, returns a bytes
b'abc\r\n'
```

Type and Content Mismatches in 3.X

```
# Types are not flexible for file content
>>> open('temp', 'w').write('abc\n')          # Text mode makes and requires str
4
>>> open('temp', 'w').write(b'abc\n')
TypeError: must be str, not bytes

>>> open('temp', 'wb').write(b'abc\n')        # Binary mode makes and requires bytes
4
>>> open('temp', 'wb').write('abc\n')
TypeError: 'str' does not support the buffer interface
```

Reading and Writing Unicode in 3.X

```
C:\code> C:\python33\python
>>> S = 'A\xc4B\xe8C'          # Five-character decoded string, non-ASCII
>>> S
'AÄBèC'
>>> len(S)
5
```

Manual encoding

```
# Encode manually with methods
>>> L = S.encode('latin-1')        # 5 bytes when encoded as latin-1
>>> L
b'A\xc4B\xe8C'
>>> len(L)
5

>>> U = S.encode('utf-8')          # 7 bytes when encoded as utf-8
>>> U
b'A\xc3\x84B\xc3\xa8C'
>>> len(U)
7
```

File output encoding

```
# Encoding automatically when written
>>> open('latindata', 'w', encoding='latin-1').write(S)      # Write as latin-1
5
>>> open('utf8data', 'w', encoding='utf-8').write(S)         # Write as utf-8
5

>>> open('latindata', 'rb').read()                           # Read raw bytes
b'A\xc4B\xe8C'

>>> open('utf8data', 'rb').read()                            # Different in files
b'A\xc3\x84B\xc3\xa8C'
```

File input decoding

```
# Decoding automatically when read
>>> open('latindata', 'r', encoding='latin-1').read()        # Decoded on input
'AÄBèC'
>>> open('utf8data', 'r', encoding='utf-8').read()           # Per encoding type
'AÄBèC'

>>> X = open('latindata', 'rb').read()                       # Manual decoding:
>>> X.decode('latin-1')                                      # Not necessary
'AÄBèC'
>>> X = open('utf8data', 'rb').read()
>>> X.decode()                                                    # UTF-8 is default
'AÄBèC'
```

Decoding mismatches

```
>>> file = open(r'C:\Python33\python.exe', 'r')
>>> text = file.read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2: ...

>>> file = open(r'C:\Python33\python.exe', 'rb')
>>> data = file.read()
>>> data[:20]
b'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00'
```

Handling the BOM in 3.X

- Some encoding schemes store a special byte order marker (BOM) sequence at the start of files, to specify data endianness (which end of a string of bits is most significant to its value) or declare the encoding type.
- Python both skips this marker on input and writes it on output if the encoding name implies it, but we sometimes must use a specific encoding name to force BOM processing explicitly.

Handling the BOM in 3.X

- For example, in the UTF-16 and UTF-32 encodings, the BOM specifies big- or little-endian format.
- A UTF-8 text file may also include a BOM, but this isn't guaranteed, and serves only to declare that it is UTF-8 in general.
- When reading and writing data using these encoding schemes, Python automatically skips or writes the BOM if it is either implied by a general encoding name, or if you provide a more specific encoding name to force the issue.

Dropping the BOM in Notepad

- Let's make some files with BOMs to see how this works in practice.
- When you save a text file in Windows Notepad, you can specify its encoding type in a drop-down list - simple ASCII text, UTF-8, or little- or big-endian UTF-16.
- If a two-line text file named spam.txt is saved in Notepad as the encoding type ANSI, for instance, it's written as simple ASCII text without a BOM. When this file is read in binary mode in Python, we can see the actual bytes stored in the file.
- When it's read as text, Python performs end-of-line translation by default; we can also decode it as explicit UTF-8 text since ASCII is a subset of this scheme (and UTF-8 is Python 3.X's default encoding)

```
C:\code> C:\python33\python                # File saved in Notepad
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
>>> open('spam.txt', 'rb').read()          # ASCII (UTF-8) text file
b'spam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()           # Text mode translates line end
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'spam\nSPAM\n'
```

If this file is instead saved as *UTF-8* in Notepad, it is prepended with a 3-byte UTF-8 BOM sequence, and we need to give a more specific encoding name ("utf-8-sig") to force Python to skip the marker:

```
>>> open('spam.txt', 'rb').read()          # UTF-8 with 3-byte BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'
>>> open('spam.txt', 'r').read()
'ï»¿spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'\ufeffspam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-8-sig').read()
'spam\nSPAM\n'
```

If the file is stored as *Unicode big endian* in Notepad, we get UTF-16-format data in the file, with 2-byte (16-bit) characters prepended with a 2-byte BOM sequence—the encoding name "utf-16" in Python skips the BOM because it is implied (since all UTF-16 files have a BOM), and "utf-16-be" handles the big-endian format but does not skip the BOM (the second of the following fails to print on older Pythons):

```
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00s\x00p\x00a\x00m\x00\r\x00\n\x00S\x00P\x00A\x00M\x00\r\x00\n'
>>> open('spam.txt', 'r').read()
'\xfeÿ\x00s\x00p\x00a\x00m\x00\n\x00\n\x00S\x00P\x00A\x00M\x00\n\x00\n'
>>> open('spam.txt', 'r', encoding='utf-16').read()
'spam\nSPAM\n'
>>> open('spam.txt', 'r', encoding='utf-16-be').read()
'\ufeffspam\nSPAM\n'
```

Dropping the BOM in Python

```
>>> open('temp.txt', 'w', encoding='utf-8').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                          # No BOM
b'spam\r\nSPAM\r\n'

>>> open('temp.txt', 'w', encoding='utf-8-sig').write('spam\nSPAM\n')
10
>>> open('temp.txt', 'rb').read()                          # Wrote BOM
b'\xef\xbb\xbfspam\r\nSPAM\r\n'

>>> open('temp.txt', 'r').read()
'ï»¿spam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()         # Keeps BOM
'\ufeffspam\nSPAM\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()     # Skips BOM
'spam\nSPAM\n'
```