Q1. In Python 3.X, what are the names and functions of string object types?

Python 3.X has three string types: str (for Unicode text, including ASCII), bytes

(for binary data with absolute byte values), and bytearray (a mutable flavor of

bytes). The str type usually represents content stored on a text file, and the other

two types generally represent content stored on binary files.

Q2. How do the string forms in Python 3.X vary in terms of operations?

Python 3.X’s string types share almost all the same operations: method calls, sequence operations, and even larger tools like pattern matching work the same way. On the other hand, only str supports string formatting operations, and by tear ray has an additional set of operations that perform in-place changes. The str and bytes types also have methods for encoding and decoding text, respectively.

Q3. In 3.X, how do you put non-ASCII Unicode characters in a string?

Non-ASCII Unicode characters can be coded in a string with both hex (\xNN) and Unicode (\uNNNN, \UNNNNNNNN) escapes. On some machines, some non-ASCII characters— certain Latin-1 characters, for example—can also be typed or pasted directly into code, and are interpreted per the UTF-8 default or a source code encoding directive comment.

Q4. In Python 3.X, what are the key differences between text-mode and binary-mode files?

In 3.X, text-mode files assume their file content is Unicode text (even if it’s all

ASCII) and automatically decode when reading and encode when writing. With

binary-mode files, bytes are transferred to and from the file unchanged. The contents of text-mode files are usually represented as str objects in your script, and the contents of binary files are represented as bytes (or bytearray) objects. Textmode files also handle the BOM for certain encoding types and automatically translate end-of-line sequences to and from the single \n character on input and output unless this is explicitly disabled; binary-mode files do not perform either of these steps. Python 2.X uses codecs.open for Unicode files, which encodes and decodes similarly; 2.X’s open only translates line ends in text mode.

Q5. How can you interpret a Unicode text file containing text encoded in a different encoding than your platform's default?

To read files encoded in a different encoding than the default for your platform,

simply pass the name of the file’s encoding to the open built-in in 3.X

(codecs.open() in 2.X); data will be decoded per the specified encoding when it is read from the file. You can also read in binary mode and manually decode the bytes to a string by giving an encoding name, but this involves extra work and is somewhat error-prone for multibyte characters (you may accidentally read a partial character sequence).

Q6. What is the best way to make a Unicode text file in a particular encoding format?

To create a Unicode text file in a specific encoding format, pass the desired encoding name to open in 3.X (codecs.open() in 2.X); strings will be encoded per the desired encoding when they are written to the file. You can also manually encode a string to bytes and write it in binary mode, but this is usually extra work.

Q7. What qualifies ASCII text as a form of Unicode text?

ASCII text is considered to be a kind of Unicode text, because its 7-bit range of

values is a subset of most Unicode encodings. For example, valid ASCII text is also valid Latin-1 text (Latin-1 simply assigns the remaining possible values in an 8-bit byte to additional characters) and valid UTF-8 text (UTF-8 defines a variable-byte scheme for representing more characters, but ASCII characters are still represented with the same codes, in a single byte). This makes Unicode backward-compatible with the mass of ASCII text data in the world (though it also may have limited its options—self-identifying text, for instance, may have been difficult (though BOMs serve much the same role).

Q8. How much of an effect does the change in string types in Python 3.X have on your code?

The impact of Python 3.X’s string types change depends upon the types of strings you use. For scripts that use simple ASCII text on platforms with ASCII-compatible default encodings, the impact is probably minor: the str string type works the same in 2.X and 3.X in this case. Moreover, although string-related tools in the standard library such as re, struct, pickle, and xml may technically use different types in 3.X than in 2.X, the changes are largely irrelevant to most programs because 3.X’s str and bytes and 2.X’s str support almost identical interfaces. If you process Unicode data, the toolset you need has simply moved from 2.X’s unicode and codecs.open() to 3.X’s str and open. If you deal with binary data files, you’ll need to deal with content as bytes objects; since they have a similar interface to 2.X strings, though, the impact should again be minimal. That said, the update of the

book Programming Python for 3.X ran across numerous cases where Unicode’s

mandatory status in 3.X implied changes in standard library APIs—from networking and GUIs, to databases and email. In general, Unicode will probably impact most 3.X users eventually.