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

Python has several built-in functions associated with the [string data type](https://www.digitalocean.com/community/tutorials/an-introduction-to-working-with-strings-in-python-3). These functions let us easily modify and manipulate strings. We can think of functions as being actions that we perform on elements of our code. Built-in functions are those that are defined in the Python programming language and are readily available for us to use.

## Making Strings Upper and Lower Case

The functions str.upper() and str.lower() will return a string with all the letters of an original string converted to upper- or lower-case letters. Because strings are immutable data types, the returned string will be a new string. Any characters in the string that are not letters will not be changed.

ss = "Sammy Shark"

print(ss.upper())

| **Method** | **True if** |
| --- | --- |
| str.isalnum() | String consists of only alphanumeric characters (no symbols) |
| str.isalpha() | String consists of only alphabetic characters (no symbols) |
| str.islower() | String’s alphabetic characters are all lower case |
| str.isnumeric() | String consists of only numeric characters |
| str.isspace() | String consists of only whitespace characters |
| str.istitle() | String is in title case |
| str.isupper() | String’s alphabetic characters are all upper case |

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

String constants

The constants defined in this module are:

**string.ascii\_letters**

**The concatenation of the [ascii\_lowercase](https://docs.python.org/3/library/string.html" \l "string.ascii_lowercase" \o "string.ascii_lowercase) and [ascii\_uppercase](https://docs.python.org/3/library/string.html" \l "string.ascii_uppercase" \o "string.ascii_uppercase) constants described below. This value is not locale-dependent.**

**string.ascii\_lowercase**

**The lowercase letters 'abcdefghijklmnopqrstuvwxyz'. This value is not locale-dependent and will not change.**

**string.ascii\_uppercase**

**The uppercase letters 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. This value is not locale-dependent and will not change.**

**string.digits**

**The string '0123456789'.**

**string.hexdigits**

**The string '0123456789abcdefABCDEF'.**

**string.octdigits**

**The string '01234567'.**

**string.punctuation**

**String of ASCII characters which are considered punctuation characters in the C locale: !"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~.**

**string.printable**

**String of ASCII characters which are considered printable. This is a combination of**[**digits**](https://docs.python.org/3/library/string.html#string.digits)**, [ascii\_letters](https://docs.python.org/3/library/string.html" \l "string.ascii_letters" \o "string.ascii_letters),**[**punctuation**](https://docs.python.org/3/library/string.html#string.punctuation)**, and**[**whitespace**](https://docs.python.org/3/library/string.html#string.whitespace)**.**

**string.whitespace**

**A string containing all ASCII characters that are considered whitespace. This includes the characters space, tab, linefeed, return, formfeed, and vertical tab.**

The *sign* option is only valid for number types, and can be one of the following:

| **Option** | **Meaning** |
| --- | --- |
| '+' | indicates that a sign should be used for both positive as well as negative numbers. |
| '-' | indicates that a sign should be used only for negative numbers (this is the default behavior). |
| space | indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers. |

The 'z' option coerces negative zero floating-point values to positive zero after rounding to the format precision. This option is only valid for floating-point presentation types.

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

The clear cut way to trim this string (as I understand Python) is simply to say the string is in a variable called s, we get:

s.replace('Â ', '')

That should do the trick. But of course it complains that the non-ASCII character '\xc2' in file blabla.py is not encoded.

The code:

f = urllib.urlopen(url)

soup = BeautifulSoup(f)

s = soup.find('div', {'id':'main\_count'})

#making a print 's' here goes well. it shows 6Â 918Â 417Â 712

s.replace('Â ','')

save\_main\_count(s)

It gets no further than s.replace...

You actually only need # coding: utf-8. -\*- is not for decoration, but you are unlikely to ever need it. I think it was there for old shells.

**Example**

>>> unicode\_string = u"hello aåbäcö"

>>> unicode\_string.encode("ascii", "ignore")

'hello abc'

### **Example (Python 3)**

s = b'6\xc2\xa0918\xc2\xa0417\xc2\xa0712'

print(s.decode('latin-1')) # incorrectly decoded

u = s.decode('utf8') # correctly decoded

print(u)

print(u.replace('\N{NO-BREAK SPACE}','\_'))

print(u.replace('\xa0','-')) # \xa0 is Unicode for NO-BREAK SPACE

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

The two file types may look the same on the surface, but they encode data differently. While both binary and text files contain data stored as a series of bits (binary values of 1s and 0s), the bits in text files represent characters, while the bits in binary files represent custom data.

**Text Files**

Text files are more restrictive than binary files since they can only contain textual data. However, unlike binary files, they are less likely to become corrupted. While a small error in a binary file may make it unreadable, a small error in a text file may simply show up once the file has been opened. This is one of reasons Microsoft switched to a compressed text-based XML format for the Office 2007 file types.  
  
Text files may be saved in either a plain text (.TXT) format and rich text (.RTF) format. A typical plain text file contains several lines of text that are each followed by an End-of-Line (EOL) character. An End-of-File (EOF) marker is placed after the final character, which signals the end of the file. Rich text files use a similar file structure, but may also include text styles, such as bold and italics, as well as page formatting information. Both plain text and rich text files include a (character encoding| characterencoding) scheme that determines how the characters are interpreted and what characters can be displayed.  
  
Since text files use a simple, standard format, many programs are capable of reading and editing text files. Common text editors include Microsoft Notepad and WordPad, which are bundled with Windows, and Apple TextEdit, which is included with Mac OS X.

**Binary Files**  
  
Binary files typically contain a sequence of bytes, or ordered groupings of eight bits. When creating a custom file format for a program, a developer arranges these bytes into a format that stores the necessary information for the application. Binary file formats may include multiple types of data in the same file, such as image, video, and audio data. This data can be interpreted by supporting programs, but will show up as garbled text in a text editor. Below is an example of a .PNG image file opened in an image viewer and a text editor.  
Image Viewer     Text Editor  
Flower - Image Viewer     Flower - Text Editor  
  
As you can see, the image viewer recognizes the binary data and displays the picture. When the image is opened in a text editor, the binary data is converted to unrecognizable text. However, you may notice that some of the text is readable. This is because the PNG format includes small sections for storing textual data. The text editor, while not designed to read this file format, still displays this text when the file is opened. Many other binary file types include sections of readable text as well. Therefore, it may be possible to find out some information about an unknown binary file type by opening it in a text editor.  
  
Binary files often contain headers, which are bytes of data at the beginning of a file that identifies the file's contents. Headers often include the file type and other descriptive information. For example, in the image above, the "PNG" text indicates the file is a PNG image. If a file has invalid header information, software programs may not open the file or they may report that the file is corrupted.

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

**What’s a Character Encoding?**

There are tens if not hundreds of character encodings. The best way to start understanding what they are is to cover one of the simplest character encodings, ASCII.

Whether you’re self-taught or have a formal computer science background, chances are you’ve seen an ASCII table once or twice. ASCII is a good place to start learning about character encoding because it is a small and contained encoding. (Too small, as it turns out.)

It encompasses the following:

* **Lowercase English letters**: *a* through *z*
* **Uppercase English letters**: *A* through *Z*
* **Some punctuation and symbols**: "$" and "!", to name a couple
* **Whitespace characters**: an actual space (" "), as well as a newline, carriage return, horizontal tab, vertical tab, and a few others
* **Some non-printable characters**: characters such as backspace, "\b", that can’t be printed literally in the way that the letter *A* can

So what is a more formal definition of a character encoding?

At a very high level, it’s a way of translating characters (such as letters, punctuation, symbols, whitespace, and control characters) to integers and ultimately to bits. Each character can be encoded to a unique sequence of bits. Don’t worry if you’re shaky on the concept of bits, because we’ll get to them shortly.

The various categories outlined represent groups of characters. Each single character has a corresponding **code point**, which you can think of as just an integer. Characters are segmented into different ranges within the ASCII table:

| **Code Point Range** | **Class** |
| --- | --- |
| 0 through 31 | Control/non-printable characters |
| 32 through 64 | Punctuation, symbols, numbers, and space |
| 65 through 90 | Uppercase English alphabet letters |
| 91 through 96 | Additional graphemes, such as [ and \ |
| 97 through 122 | Lowercase English alphabet letters |
| 123 through 126 | Additional graphemes, such as { and | |
| 127 | Control/non-printable character (DEL) |

The entire ASCII table contains 128 characters. This table captures the complete **character set** that ASCII permits. If you don’t see a character here, then you simply can’t express it as printed text under the ASCII encoding scheme.

**# From lib/python3.7/string.py**

**whitespace = ' \t\n\r\v\f'**

**ascii\_lowercase = 'abcdefghijklmnopqrstuvwxyz'**

**ascii\_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'**

**ascii\_letters = ascii\_lowercase + ascii\_uppercase**

**digits = '0123456789'**

**hexdigits = digits + 'abcdef' + 'ABCDEF'**

**octdigits = '01234567'**

**punctuation = r"""!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~"""**

**printable = digits + ascii\_letters + punctuation + whitespace**

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

Best practice, in general, use UTF-8 for writing to files (we don't even have to worry about byte-order with utf-8). utf-8 is the most modern and universally usable encoding - it works in all web browsers, most text-editors (see your settings if you have issues) and most terminals/shells.

It has some non-ASCII symbols. How can I convert these safely to symbols that can be used in HTML source?

Currently I'm converting everything to Unicode on the way in, joining it all together in a Python string, then doing:

import codecs

f = codecs.open('out.txt', mode="w", encoding="iso-8859-1")

f.write(all\_html.encode("iso-8859-1", "replace"))

## Example using many Unicode characters

Here's an example that attempts to map every possible character up to three bits wide (4 is the max, but that would be going a bit far) from the digital representation (in integers) to an encoded printable output, along with its name, if possible (put this into a file called uni.py):

from \_\_future\_\_ import print\_function

import io

from unicodedata import name, category

from curses.ascii import controlnames

from collections import Counter

try: # use these if Python 2

unicode\_chr, range = unichr, xrange

except NameError: # Python 3

unicode\_chr = chr

exclude\_categories = set(('Co', 'Cn'))

counts = Counter()

control\_names = dict(enumerate(controlnames))

with io.open('unidata', 'w', encoding='utf-8') as f:

for x in range((2\*\*8)\*\*3):

try:

char = unicode\_chr(x)

except ValueError:

continue # can't map to unicode, try next x

cat = category(char)

counts.update((cat,))

if cat in exclude\_categories:

continue # get rid of noise & greatly shorten result file

try:

uname = name(char)

except ValueError: # probably control character, don't use actual

uname = control\_names.get(x, '')

f.write(u'{0:>6x} {1} {2}\n'.format(x, cat, uname))

else:

f.write(u'{0:>6x} {1} {2} {3}\n'.format(x, cat, char, uname))

# may as well describe the types we logged.

for cat, count in counts.items():

print('{0} chars of category, {1}'.format(count, cat))

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

ASCII == UNICODE? For backward compatibility, the first 128 Unicode characters point to ASCII characters. And since UTF-8 encodes each of those characters using 1-byte. ASCII is essentially just UTF-8, or we can say that ASCII is a subset of Unicode.

**Size –**

1. It is obvious by now that Unicode represents far more characters than ASCII. ASCII uses a 7-bit range to encode just **128** distinct characters. Unicode on the other hand encodes **154** written scripts. And did I mention emoji? Those too.
2. So, we can say that, while Unicode supports a larger range of characters it also takes up a lot more space than ASCII.

**ASCII == UNICODE?**

1. For backward compatibility, the first 128 Unicode characters point to ASCII characters. And since UTF-8 encodes each of those characters using 1-byte.
2. ASCII is essentially just UTF-8, or we can say that ASCII is a subset of Unicode. Vice versa isn’t true.

**Conclusion:**

In conclusion, both Unicode and ASCII are the standards for text encoding, and they hold the utmost significance in modern communications. Both have their advantages and disadvantages, but a more universal solution for encoding will always facilitate and create ease in communication in the future.

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

## Common Stumbling Blocks

This section lists those few changes that are most likely to trip you up if you’re used to Python 2.5.

### **Print Is A Function**

The print statement has been replaced with a [print()](https://docs.python.org/3/library/functions.html#print) function, with keyword arguments to replace most of the special syntax of the old print statement Examples:

Old: print "The answer is", 2\*2

New: print("The answer is", 2\*2)

Old: print x, # Trailing comma suppresses newline

New: print(x, end=" ") # Appends a space instead of a newline

Old: print # Prints a newline

New: print() # You must call the function!

Old: print >>sys.stderr, "fatal error"

New: print("fatal error", file=sys.stderr)

Old: print (x, y) # prints repr((x, y))

New: print((x, y)) # Not the same as print(x, y)!

You can also customize the separator between items, e.g.:

print("There are <", 2\*\*32, "> possibilities!", sep="")

### **Ordering Comparisons**

### **Views And Iterators Instead Of Lists**

### **Integers**

## Overview Of Syntax Changes

### **Operators And Special Methods**

### **Builtins**

## Performance[¶](https://docs.python.org/3/whatsnew/3.0.html#performance)

This are the methods will been used to change string types