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

Method Description

capitalize() Converts the first character to upper case

casefold() Converts string into lower case

center() Returns a centered string

count() Returns the number of times a specified value occurs in a string

encode() Returns an encoded version of the string

endswith() Returns true if the string ends with the specified value

expandtabs() Sets the tab size of the string

find() Searches the string for a specified value and returns the position of where it was found

format() Formats specified values in a string

format\_map() Formats specified values in a string

index() Searches the string for a specified value and returns the position of where it was found

isalnum() Returns True if all characters in the string are alphanumeric

isalpha() Returns True if all characters in the string are in the alphabet

isascii() Returns True if all characters in the string are ascii characters

isdecimal() Returns True if all characters in the string are decimals

isdigit() Returns True if all characters in the string are digits

isidentifier() Returns True if the string is an identifier

islower() Returns True if all characters in the string are lower case

isnumeric() Returns True if all characters in the string are numeric

isprintable() Returns True if all characters in the string are printable

isspace() Returns True if all characters in the string are whitespaces

istitle() Returns True if the string follows the rules of a title

isupper() Returns True if all characters in the string are upper case

join() Converts the elements of an iterable into a string

ljust() Returns a left justified version of the string

lower() Converts a string into lower case

lstrip() Returns a left trim version of the string

maketrans() Returns a translation table to be used in translations

partition() Returns a tuple where the string is parted into three parts

replace() Returns a string where a specified value is replaced with a specified value

rfind() Searches the string for a specified value and returns the last position of where it was found

rindex() Searches the string for a specified value and returns the last position of where it was found

rjust() Returns a right justified version of the string

rpartition() Returns a tuple where the string is parted into three parts

rsplit() Splits the string at the specified separator, and returns a list

rstrip() Returns a right trim version of the string

split() Splits the string at the specified separator, and returns a list

splitlines() Splits the string at line breaks and returns a list

startswith() Returns true if the string starts with the specified value

strip() Returns a trimmed version of the string

swapcase() Swaps cases, lower case becomes upper case and vice versa

title() Converts the first character of each word to upper case

translate() Returns a translated string

upper() Converts a string into upper case

zfill() Fills the string with a specified number of 0 values at the beginning

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

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.

Changed in version 3.11: Added the 'z' option (see also PEP 682).

The ',' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

Changed in version 3.1: Added the ',' option (see also PEP 378).

The '\_' option signals the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type 'd'. For integer presentation types 'b', 'o', 'x', and 'X', underscores will be inserted every 4 digits. For other presentation types, specifying this option is an error.

Changed in version 3.6: Added the '\_' option (see also PEP 515).

When no explicit alignment is given, preceding the width field by a zero ('0') character enables sign-aware zero-padding for numeric types. This is equivalent to a fill character of '0' with an alignment type of '='.

Changed in version 3.10: Preceding the width field by '0' no longer affects the default alignment for strings.

is\_valid()

Returns false if the template has invalid placeholders that will cause substitute() to raise ValueError.

New in version 3.11.

get\_identifiers()¶

Returns a list of the valid identifiers in the template, in the order they first appear, ignoring any invalid identifiers.

New in version 3.11.

braceidpattern – This is like idpattern but describes the pattern for braced placeholders. Defaults to None which means to fall back to idpattern (i.e. the same pattern is used both inside and outside braces). If given, this allows you to define different patterns for braced and unbraced placeholders.

New in version 3.7.

flags – The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is re.IGNORECASE. Note that re.VERBOSE will always be added to the flags, so custom idpatterns must follow conventions for verbose regular expressions.

New in version 3.2.

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

Transliteration is a process of writing the word of one language using similarly pronounced alphabets in other languages. It deals with the pronunciation of words in other languages. Similarly, in computer language, the computer can handle ASCII characters but has problems with non-ASCII characters. There are some times when we are unable to skip non-ASCII characters as it can lead to loss of information. There should be a way to read non-ASCII characters and express them by text in ASCII characters.

Approach 1:

This approach is related to the inbuilt library unidecode. This library helps Transliterating non-ASCII characters in Python. It provides an unidecode() method that takes Unicode data and tries to represent it in ASCII. This method automatically determines scripting language and transliterates it accordingly. It accepts unicode string values and returns a transliteration in string format.

Steps:

* Import unidecode library
* Call unidecode() method with input text

Code:

# Import unidecode module from unidecode

from unidecode import unidecode

# Get transliteration for following

# non-ASCII text (Unicode string)

print(unidecode(u'ko\u017eu\u0161\u010dek'))

# Get transliteration for following

# non-ASCII text (Devanagari)

print(unidecode("आप नीचे अपनी भाषा और इनपुट उपकरण चुनें और लिखना आरंभ करें"))

# Get transliteration for following

# non-ASCII text (Chinese)

print(unidecode("谢谢你"))

# Get transliteration for following

# non-ASCII text (Japanese)

print(unidecode("ありがとう。"))

# Get transliteration for following

# non-ASCII text (Russian)

print(unidecode("улыбаться Владимир Путин"))

Approach 2:

This approach deals with building a structure that will help in transliteration. In this, Unicode values of non-ASCII characters are labeled with related ASCII values from here provides a list of scripts. Unicode’s value for letters in each script is provided therewith representable ASCII character. Wikipedia also has a collection of Unicode values and respective transliteration.

Steps:

Create Dictionary having Unicode values as keys and ASCII representation as values

Transliterate each letter in the text using that dictionary.

For example, if we want transliteration of the Russian language, we will take all letters in Cyrillic script as Russian uses Cyrillic. Then for each letter, its Unicode value and ASCII representation are used to create a dictionary. And, then this dictionary is used to transliterate given text.

For example, some letters of Devanagari script are taken and a dictionary is created. Further, it is used on small text for transliteration.

Code:

# Create devanagari transliteration dictionary

devanagari\_translit\_dict = {

'\u0905': 'A', '\u0906': 'AA', '\u0907': 'I', '\u0908': 'II',

'\u0909': 'U', '\u090A': 'UU', '\u090F': 'E', '\u0910': 'AI',

'\u0913': 'O', '\u0914': 'AU', '\u0915': 'K', '\u0916': 'KH',

'\u0917': 'G', '\u0918': 'GH', '\u0919': 'NG', '\u091A': 'C',

'\u091B': 'CH', '\u091C': 'J', '\u091D': 'JH', '\u091E': 'NY',

'\u091F': 'TT', '\u0920': 'TTH', '\u0921': 'DD', '\u0922': 'DDH',

'\u0923': 'NN', '\u0924': 'T', '\u0925': 'TH', '\u0926': 'D',

'\u0927': 'DH', '\u0928': 'N', '\u092A': 'P', '\u092B': 'PH',

'\u092C': 'B', '\u092D': 'BH', '\u092E': 'M', '\u092F': 'Y',

'\u0930': 'R', '\u0932': 'L', '\u0933': 'LL', '\u0935': 'V',

'\u0936': 'SH', '\u0937': 'SS', '\u0938': 'S', '\u0939': 'H',

'\u093E': 'AA', '\u093F': 'I', '\u0940': 'II', '\u0941': 'U',

'\u0942': 'UU', '\u0947': 'E', '\u0948': 'AI', '\u094B': 'O',

'\u094C': 'AU', '\u094D': '', '\u0902': 'n'}

# Define function transliterating text

def transliterate(text, translit\_dict):

new\_word = ''

for letter in text:

new\_letter = ''

if letter in translit\_dict:

new\_letter = translit\_dict[letter]

else:

new\_letter = letter

new\_word += new\_letter

return new\_word

# Input text in devanagari

text = "आप नीचे अपनी भाषा और इनपुट उपकरण चुनें और लिखना आरंभ करें"

# Obtain Transliterated text for given input text

transliterated\_text = transliterate(text, devanagari\_translit\_dict)

print(transliterated\_text)

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

Computers store every file as a collection of 0s and 1s i.e., in binary form. Therefore, every file is basically just a series of bytes stored one after the other. There are mainly two types of data files — text file and binary file. A text file consists of human readable characters, which can be opened by any text editor. On the other hand, binary files are made up of non-human readable characters and symbols, which require specific programs to access its contents.

A text file can be understood as a sequence of characters consisting of alphabets, numbers and other special symbols. Files with extensions like .txt, .py, .csv, etc. are some examples of text files. When we open a text file using a text editor (e.g., Notepad), we see several lines of text. However, the file contents are not stored in such a way internally. Rather, they are stored in sequence of bytes consisting of 0s and 1s. In ASCII, UNICODE or any other encoding scheme, the value of each character of the text file is stored as bytes. So, while opening a text file, the text editor translates each ASCII value and shows us the equivalent character that is readable by the human being. For example, the ASCII value 65 (binary equivalent 1000001) will be displayed by a text editor as the letter ‘A’ since the number 65 in ASCII character set represents ‘A’. Each line of a text file is terminated by a special character, called the End of Line (EOL). For example, the default EOL character in Python is the newline (\n). However, other characters can be used to indicate EOL. When a text editor or a program interpreter encounters the ASCII equivalent of the EOL character, it displays the remaining file contents starting from a new line. Contents in a text file are usually separated by whitespace, but comma (,) and tab (\t) are also commonly used to separate values in a text file.

Binary files are also stored in terms of bytes (0s and 1s), but unlike text files, these bytes do not represent the ASCII values of characters. Rather, they represent the actual content such as image, audio, video, compressed versions of other files, executable files, etc. These files are not human readable. Thus, trying to open a binary file using a text editor will show some garbage values. We need specific software to read or write the contents of a binary file. Binary files are stored in a computer in a sequence of bytes. Even a single bit change can corrupt the file and make it unreadable to the supporting application. Also, it is difficult to remove any error which may occur in the binary file as the stored contents are not human readable. We can read and write both text and binary files through Python programs.

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

The Binary Option:

One alternative that is always available is to open files in binary mode and process them as bytes rather than as text. This can work in many cases, especially those where the ASCII markers are embedded in genuinely arbitrary binary data.

However, for both “text data with unknown encoding” and “text data with known encoding, but potentially containing encoding errors”, it is often preferable to get them into a form that can be handled as text strings. In particular, some APIs that accept both bytes and text may be very strict about the encoding of the bytes they accept (for example, the urllib.urlparse module accepts only pure ASCII data for processing as bytes, but will happily process text strings containing non-ASCII code points).

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

Files in an ASCII compatible encoding, best effort is acceptable:

Use case: the files to be processed are in an ASCII compatible encoding, but you don’t know exactly which one. All files must be processed without triggering any exceptions, but some risk of data corruption is deemed acceptable (e.g. collating log files from multiple sources where some data errors are acceptable, so long as the logs remain largely intact).

Approach: use the “latin-1” encoding to map byte values directly to the first 256 Unicode code points. This is the closest equivalent Python 3 offers to the permissive Python 2 text handling model.

Example: f = open(fname, encoding="latin-1")

Consequences:

* data will not be corrupted if it is simply read in, processed as ASCII text, and written back out again.
* will never raise UnicodeDecodeError when reading data
* will still raise UnicodeEncodeError if codepoints above 0xFF (e.g. smart quotes copied from a word processing program) are added to the text string before it is encoded back to bytes. To prevent such errors, use the backslashreplace error handler (or one of the other error handlers that replaces Unicode code points without a representation in the target encoding with sequences of ASCII code points).
* data corruption may occur if the source data is in an ASCII incompatible encoding (e.g. UTF-16)
* corruption may occur if data is written back out using an encoding other than latin-1
* corruption may occur if the non-ASCII elements of the string are modified directly (e.g. for a variable width encoding like UTF-8 that has been decoded as latin-1 instead, slicing the string at an arbitrary point may split a multi-byte character into two pieces)

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

ASCII and Unicode are two popular encoding schemes. ASCII encodes symbols, digits, letters, etc., whereas Unicode encodes special texts from different languages, letters, symbols, etc.

It can be said that ASCII is a subset of the Unicode encoding scheme

Unicode has several encoding formats, two of which are UTF-7 and UTF-8, which use 7 bits and 8 bits, respectively, to represent characters that are otherwise difficult to store in memory. ASCII also uses 7 and 8 bits for the representation of characters. A large number of characters used around the world which cannot be encoded by using 8-bit representation led to the creation of UTF-16 and UTF-32 encoding formats under Unicode encoding. Thus, ASCII is a subset of the Unicode encoding scheme.

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

The field\_name itself begins with an arg\_name that is either a number or a keyword. If it’s a number, it refers to a positional argument, and if it’s a keyword, it refers to a named keyword argument. If the numerical arg\_names in a format string are 0, 1, 2, … in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, … will be automatically inserted in that order. Because arg\_name is not quote-delimited, it is not possible to specify arbitrary dictionary keys (e.g., the strings '10' or ':-]') within a format string. The arg\_name can be followed by any number of index or attribute expressions. An expression of the form '.name' selects the named attribute using getattr(), while an expression of the form '[index]' does an index lookup using \_\_getitem\_\_().

Changed in version 3.1: The positional argument specifiers can be omitted for str.format(), so '{} {}'.format(a, b) is equivalent to '{0} {1}'.format(a, b).

Changed in version 3.4: The positional argument specifiers can be omitted for Formatter.

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.

Changed in version 3.11: Added the 'z' option (see also PEP 682).

The '#' option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float and complex types. For integers, when binary, octal, or hexadecimal output is used, this option adds the respective prefix '0b', '0o', '0x', or '0X' to the output value. For float and complex the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

The ',' option signals the use of a comma for a thousands separator. For a locale aware separator, use the 'n' integer presentation type instead.

Changed in version 3.1: Added the ',' option (see also PEP 378).

The '\_' option signals the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type 'd'. For integer presentation types 'b', 'o', 'x', and 'X', underscores will be inserted every 4 digits. For other presentation types, specifying this option is an error.

Changed in version 3.6: Added the '\_' option (see also PEP 515).

width is a decimal integer defining the minimum total field width, including any prefixes, separators, and other formatting characters. If not specified, then the field width will be determined by the content.

When no explicit alignment is given, preceding the width field by a zero ('0') character enables sign-aware zero-padding for numeric types. This is equivalent to a fill character of '0' with an alignment type of '='.

Changed in version 3.10: Preceding the width field by '0' no longer affects the default alignment for strings.

is\_valid()

Returns false if the template has invalid placeholders that will cause substitute() to raise ValueError.

New in version 3.11.

get\_identifiers()

Returns a list of the valid identifiers in the template, in the order they first appear, ignoring any invalid identifiers.

New in version 3.11.

dpattern – This is the regular expression describing the pattern for non-braced placeholders. The default value is the regular expression (?a:[\_a-z][\_a-z0-9]\*). If this is given and braceidpattern is None this pattern will also apply to braced placeholders.

Note Since default flags is re.IGNORECASE, pattern [a-z] can match with some non-ASCII characters. That’s why we use the local a flag here.

Changed in version 3.7: braceidpattern can be used to define separate patterns used inside and outside the braces.

braceidpattern – This is like idpattern but describes the pattern for braced placeholders. Defaults to None which means to fall back to idpattern (i.e. the same pattern is used both inside and outside braces). If given, this allows you to define different patterns for braced and unbraced placeholders.

New in version 3.7.

flags – The regular expression flags that will be applied when compiling the regular expression used for recognizing substitutions. The default value is re.IGNORECASE. Note that re.VERBOSE will always be added to the flags, so custom idpatterns must follow conventions for verbose regular expressions.

New in version 3.2.