

python / cpython Public
[Code](#)
[Issues](#)
5k+
[Pull requests](#)
2.2k
[Actions](#)
[Projects](#)
[Security and q](#)

Commit 4398b78


serhiy-storchaka authored on May 26, 2025 · ✓ 24 / 26 · Verified

[3.12] [gh-133767](#): Fix use-after-free in the unicode-escape decoder with an error handler ([GH-129648](#)) ([GH-133944](#)) ([#134337](#))

If the error handler is used, a new bytes object is created to set as the object attribute of UnicodeDecodeError, and that bytes object then replaces the original data. A pointer to the decoded data will become invalid after destroying that temporary bytes object. So we need other way to return the first invalid escape from `_PyUnicode_DecodeUnicodeEscapeInternal()`.

`_PyBytes_DecodeEscape()` does not have such issue, because it does not use the error handlers registry, but it should be changed for compatibility with `_PyUnicode_DecodeUnicodeEscapeInternal()`.

(cherry picked from commit [9f69a58](#))

(cherry picked from commit [6279eb8](#))

[3.12](#) ([#134337](#)) · [v3.12.13](#) ... [v3.12.11](#)

1 parent [310cd89](#) commit 4398b78

8 files changed

+194 -57 ■■■■■
[↑ Top](#)



- Include/cpython

- bytesobject.h

- unicodeobject.h






- Lib/test

- test_codeccallbacks.py

- test_codecscs.py

- Misc/NEWS.d/next/Security

- 2025-05-09-20-22-54.gh-issue-133767.kN2i3Q.rst

- v  Objects
 -  bytesobject.c
 -  unicodeobject.c
- v  Parser
 -  string_parser.c

 Search within code 

v Include/cpython/bytesobject.h ...

```

↑... @@ -25,6 +25,10 @@ PyAPI_FUNC(PyObject*) _PyBytes_FromHex(
25 25     int use_bytearray);
26 26
27 27     /* Helper for PyBytes_DecodeEscape that detects invalid escape chars. */
28 + PyAPI_FUNC(PyObject*) _PyBytes_DecodeEscape2(const char *, Py_ssize_t,
29 +                                               const char *,
30 +                                               int *, const char **);
31 + // Export for binary compatibility.
28 32     PyAPI_FUNC(PyObject *) _PyBytes_DecodeEscape(const char *, Py_ssize_t,
29 33                                               const char *, const char **);
30 34

```

v Include/cpython/unicodeobject.h ...

```

↑... @@ -684,6 +684,19 @@ PyAPI_FUNC(PyObject*)
_PyUnicode_DecodeUnicodeEscapeStateful(
684 684 );
685 685     /* Helper for PyUnicode_DecodeUnicodeEscape that detects invalid escape
686 686     chars. */
687 + PyAPI_FUNC(PyObject*) _PyUnicode_DecodeUnicodeEscapeInternal2(
688 +     const char *string,     /* Unicode-Escape encoded string */
689 +     Py_ssize_t length,     /* size of string */
690 +     const char *errors,     /* error handling */
691 +     Py_ssize_t *consumed,   /* bytes consumed */
692 +     int *first_invalid_escape_char, /* on return, if not -1, contain the first
693 +                                     invalid escaped char (<= 0xff) or
694 +                                     invalid
694 +                                     octal escape (> 0xff) in string. */

```

```

695 +     const char **first_invalid_escape_ptr); /* on return, if not NULL, may
696 +         point to the first invalid escaped
697 +         char in string.
698 +         May be NULL if errors is not NULL. */
699 + // Export for binary compatibility.
687 700 PyAPI_FUNC(PyObject*) _PyUnicode_DecodeUnicodeEscapeInternal(
688 701     const char *string, /* Unicode-Escape encoded string */
689 702     Py_ssize_t length, /* size of string */

```



Lib/test/test_codeccallbacks.py



@@ -1,6 +1,7 @@

```

1 1 import codecs
2 2 import html.entities
3 3 import itertools
4 + import re
4 5 import sys
5 6 import unicodedata
6 7 import unittest

```



@@ -1124,7 +1125,7 @@ def test_bug828737(self):

```

1124 1125         text = 'abc<def>ghi'*n
1125 1126         text.translate(charmap)
1126 1127

```

```

1127 -     def test_mutatingdecodehandler(self):
1128 +     def test_mutating_decode_handler(self):

```

```

1128 1129         baddata = [
1129 1130             ("ascii", b"\xff"),
1130 1131             ("utf-7", b"++"),

```



@@ -1159,6 +1160,42 @@ def mutating(exc):

```

1159 1160         for (encoding, data) in baddata:
1160 1161             self.assertEqual(data.decode(encoding, "test.mutating"),
1161 1162                 "\u4242")

```

```

1163 +     def test_mutating_decode_handler_unicode_escape(self):
1164 +         decode = codecs.unicode_escape_decode
1165 +         def mutating(exc):
1166 +             if isinstance(exc, UnicodeDecodeError):
1167 +                 r = data.get(exc.object[:exc.end])

```

```

1168 +         if r is not None:
1169 +             exc.object = r[0] + exc.object[exc.end:]
1170 +             return ('\u0404', r[1])
1171 +             raise AssertionError("don't know how to handle %r" % exc)
1172 +
1173 +     codecs.register_error('test.mutating2', mutating)
1174 +     data = {
1175 +         br'\x0': (b'\\', 0),
1176 +         br'\x3': (b'xxx\\', 3),
1177 +         br'\x5': (b'x\\', 1),
1178 +     }
1179 +     def check(input, expected, msg):
1180 +         with self.assertWarns(DeprecationWarning) as cm:
1181 +             self.assertEqual(decode(input, 'test.mutating2'), (expected,
1182 + len(input)))
1182 +             self.assertIn(msg, str(cm.warning))
1183 +
1184 +         check(br'\x0n\z', '\u0404n\\z', r"invalid escape sequence '\z'")
1185 +         check(br'\x0n\501', '\u0404n\u0141', r"invalid octal escape sequence
1186 + '\501'")
1186 +         check(br'\x0z', '\u0404\\z', r"invalid escape sequence '\z'")
1187 +
1188 +         check(br'\x3n\zr', '\u0404n\\zr', r"invalid escape sequence '\z'")
1189 +         check(br'\x3zr', '\u0404\\zr', r"invalid escape sequence '\z'")
1190 +         check(br'\x3z5', '\u0404\\z5', r"invalid escape sequence '\z'")
1191 +         check(memoryview(br'\x3z5x')[:-1], '\u0404\\z5', r"invalid escape
1192 + sequence '\z'")
1192 +         check(memoryview(br'\x3z5xy')[:-2], '\u0404\\z5', r"invalid escape
1193 + sequence '\z'")
1193 +
1194 +         check(br'\x5n\z', '\u0404n\\z', r"invalid escape sequence '\z'")
1195 +         check(br'\x5n\501', '\u0404n\u0141', r"invalid octal escape sequence
1196 + '\501'")
1196 +         check(br'\x5z', '\u0404\\z', r"invalid escape sequence '\z'")
1197 +         check(memoryview(br'\x5zy')[:-1], '\u0404\\z', r"invalid escape
1198 + sequence '\z'")

```

```
1162 1199         # issue32583
```

```
1163 1200         def test_crashing_decode_handler(self):
```

```
1164 1201             # better generating one more character to fill the extra space slot
```



Lib/test/test_codex.py



```

@@ -1196,23 +1196,39 @@ def test_escape(self):
1196 1196         check(br"[\1010]", b"[A0]")
1197 1197         check(br"[\x41]", b"[A]")
1198 1198         check(br"[\x410]", b"[A0]")
1199  +
1200  +         def test_warnings(self):
1201  +             decode = codecs.escape_decode
1202  +             check = coding_checker(self, decode)
1199 1203         for i in range(97, 123):
1200 1204             b = bytes([i])
1201 1205             if b not in b'abfnrtvx':
1202  -                 with self.assertWarns(DeprecationWarning):
1206  +                 with self.assertWarnsRegex(DeprecationWarning,
1207  +                     r"invalid escape sequence '\\%c" % i):
1203 1208                     check(b"\" + b, b"\" + b)
1204  -                 with self.assertWarns(DeprecationWarning):
1209  +                 with self.assertWarnsRegex(DeprecationWarning,
1210  +                     r"invalid escape sequence '\\%c" % (i-32)):
1205 1211                     check(b"\" + b.upper(), b"\" + b.upper())
1206  -                 with self.assertWarns(DeprecationWarning):
1212  +                 with self.assertWarnsRegex(DeprecationWarning,
1213  +                     r"invalid escape sequence '\\8"):
1207 1214                     check(br"\8", b"\"8")
1208 1215                     with self.assertWarns(DeprecationWarning):
1209 1216                     check(br"\9", b"\"9")
1210  -                 with self.assertWarns(DeprecationWarning):
1217  +                 with self.assertWarnsRegex(DeprecationWarning,
1218  +                     r"invalid escape sequence '\\\xfa") as cm:
1211 1219                     check(b"\"xfa", b"\"xfa")
1212 1220                     for i in range(0o400, 0o1000):
1213  -                 with self.assertWarns(DeprecationWarning):
1221  +                 with self.assertWarnsRegex(DeprecationWarning,
1222  +                     r"invalid octal escape sequence '\\%o" % i):
1214 1223                     check(rb'\%o' % i, bytes([i & 0o377]))
1215 1224
1225  +                 with self.assertWarnsRegex(DeprecationWarning,
1226  +                     r"invalid escape sequence '\\z"):

```

1227	+	self.assertEqual(decode(br'\x\z', 'ignore'), (b'\z', 4))
1228	+	with self.assertWarnsRegex(DeprecationWarning,
1229	+	r"invalid octal escape sequence '\\501"):
1230	+	self.assertEqual(decode(br'\x\501', 'ignore'), (b'A', 6))
1231	+	
1216	1232	def test_errors(self):
1217	1233	decode = codecs.escape_decode
1218	1234	self.assertRaises(ValueError, decode, br"\x")
		@@ -2479,24 +2495,40 @@ def test_escape_decode(self):
2479	2495	check(br"[\x410]", "[A0]")
2480	2496	check(br"\u20ac", "\u20ac")
2481	2497	check(br"\U0001d120", "\U0001d120")
2498	+	
2499	+	def test_decode_warnings(self):
2500	+	decode = codecs.unicode_escape_decode
2501	+	check = coding_checker(self, decode)
2482	2502	for i in range(97, 123):
2483	2503	b = bytes([i])
2484	2504	if b not in b'abfnrtuvx':
2485	-	with self.assertWarns(DeprecationWarning):
2505	+	with self.assertWarnsRegex(DeprecationWarning,
2506	+	r"invalid escape sequence '\\%c" % i):
2486	2507	check(b"\" + b, "\"" + chr(i))
2487	2508	if b.upper() not in b'UN':
2488	-	with self.assertWarns(DeprecationWarning):
2509	+	with self.assertWarnsRegex(DeprecationWarning,
2510	+	r"invalid escape sequence '\\%c" % (i-32)):
2489	2511	check(b"\" + b.upper(), "\"" + chr(i-32))
2490	-	with self.assertWarns(DeprecationWarning):
2512	+	with self.assertWarnsRegex(DeprecationWarning,
2513	+	r"invalid escape sequence '\\8"):
2491	2514	check(br"\8", "\\8")
2492	2515	with self.assertWarns(DeprecationWarning):
2493	2516	check(br"\9", "\\9")
2494	-	with self.assertWarns(DeprecationWarning):
2517	+	with self.assertWarnsRegex(DeprecationWarning,
2518	+	r"invalid escape sequence '\\\xfa") as cm:
2495	2519	check(b"\\\xfa", "\\xfa")
2496	2520	for i in range(0o400, 0o1000):

```

2497 -         with self.assertWarns(DeprecationWarning):
2521 +         with self.assertWarnsRegex(DeprecationWarning,
2522 +             r"invalid octal escape sequence '\\%o'" % i):
2498 2523             check(rb'\%o' % i, chr(i))
2499 2524
2525 +         with self.assertWarnsRegex(DeprecationWarning,
2526 +             r"invalid escape sequence '\\z'"):
2527 +             self.assertEqual(decode(br'\x\z', 'ignore'), ('\z', 4))
2528 +         with self.assertWarnsRegex(DeprecationWarning,
2529 +             r"invalid octal escape sequence '\\501'"):
2530 +             self.assertEqual(decode(br'\x\501', 'ignore'), ('\u0141', 6))
2531 +
2500 2532     def test_decode_errors(self):
2501 2533         decode = codecs.unicode_escape_decode
2502 2534         for c, d in (b'x', 2), (b'u', 4), (b'U', 4):

```



...5-05-09-20-22-54.gh-issue-133767.kN2i3Q.rst



...

... @@ -0,0 +1,2 @@

```

1 + Fix use-after-free in the "unicode-escape" decoder with a non-"strict" error
2 + handler.

```

Objects/bytesobject.c

...



```

@@ -1048,10 +1048,11 @@ _PyBytes_FormatEx(const char *format, Py_ssize_t
format_len,

```

```

1048 1048     }
1049 1049
1050 1050     /* Unescape a backslash-escaped string. */
1051 - PyObject *_PyBytes_DecodeEscape(const char *s,
1051 + PyObject *_PyBytes_DecodeEscape2(const char *s,
1052 1052                                     Py_ssize_t len,
1053 1053                                     const char *errors,
1054 -                                     const char **first_invalid_escape)
1054 +                                     int *first_invalid_escape_char,
1055 +                                     const char **first_invalid_escape_ptr)
1055 1056     {
1056 1057         int c;
1057 1058         char *p;
@@ -1065,7 +1066,8 @@ PyObject *_PyBytes_DecodeEscape(const char *s,

```



```

1065 1066         return NULL;
1066 1067     writer.overallocate = 1;
1067 1068
1068 -     *first_invalid_escape = NULL;
1069 +     *first_invalid_escape_char = -1;
1070 +     *first_invalid_escape_ptr = NULL;
1069 1071
1070 1072     end = s + len;
1071 1073     while (s < end) {
1072 1074         @@ -1103,9 +1105,10 @@ PyObject *_PyBytes_DecodeEscape(const char *s,
1073 1075             c = (c<<3) + *s++ - '0';
1074 1076         }
1075 1077         if (c > 0377) {
1076 -             if (*first_invalid_escape == NULL) {
1077 -                 *first_invalid_escape = s-3; /* Back up 3 chars, since
1078 -                 we've
1079 -                 already incremented s. */
1080 +             if (*first_invalid_escape_char == -1) {
1081 +                 *first_invalid_escape_char = c;
1082 +                 /* Back up 3 chars, since we've already incremented s. */
1083 +                 *first_invalid_escape_ptr = s - 3;
1084             }
1085         }
1086         *p++ = c;
1087         @@ -1146,9 +1149,10 @@ PyObject *_PyBytes_DecodeEscape(const char *s,
1088         break;
1089         default:
1090 -             if (*first_invalid_escape == NULL) {
1091 -                 *first_invalid_escape = s-1; /* Back up one char, since we've
1092 -                 already incremented s. */
1093 +             if (*first_invalid_escape_char == -1) {
1094 +                 *first_invalid_escape_char = (unsigned char)s[-1];
1095 +                 /* Back up one char, since we've already incremented s. */
1096 +                 *first_invalid_escape_ptr = s - 1;
1097             }
1098         *p++ = '\\';
1099         s--;

```

```

@@ -1162,23 +1166,37 @@ PyObject *_PyBytes_DecodeEscape(const char *s,
1162 1166     return NULL;
1163 1167 }
1164 1168
1169 + // Export for binary compatibility.
1170 + PyObject *_PyBytes_DecodeEscape(const char *s,
1171 +                               Py_ssize_t len,
1172 +                               const char *errors,
1173 +                               const char **first_invalid_escape)
1174 + {
1175 +     int first_invalid_escape_char;
1176 +     return _PyBytes_DecodeEscape2(
1177 +         s, len, errors,
1178 +         &first_invalid_escape_char,
1179 +         first_invalid_escape);
1180 + }
1181 +
1182 PyObject *_PyBytes_DecodeEscape(const char *s,
1183                               Py_ssize_t len,
1184                               const char *errors,
1185                               Py_ssize_t Py_UNUSED(unicode),
1186                               const char *Py_UNUSED(recode_encoding))
1187 {
1188 +     int first_invalid_escape_char;
1189 +     const char *first_invalid_escape_ptr;
1190 +     PyObject *result = _PyBytes_DecodeEscape2(s, len, errors,
1191 +                                               &first_invalid_escape_char,
1192 +                                               &first_invalid_escape_ptr);
1193
1194     if (result == NULL)
1195         return NULL;
1196
1197     if (first_invalid_escape != NULL) {
1198         unsigned char c = *first_invalid_escape;
1199         if ('4' <= c && c <= '7') {
1200             if (first_invalid_escape_char != -1) {
1201                 if (first_invalid_escape_char > 0xff) {
1202                     if (PyErr_WarnFormat(PyExc_DeprecationWarning, 1,
1203                                         "invalid octal escape sequence '\\%.3s'",

```

1181	-	first_invalid_escape) < 0)
1198	+	"invalid octal escape sequence '\\%0'",
1199	+	first_invalid_escape_char) < 0)
1182	1200	{
1183	1201	Py_DECREF(result);
1184	1202	return NULL;
↕		@@ -1187,7 +1205,7 @@ PyObject *PyBytes_DecodeEscape(const char *s,
1187	1205	else {
1188	1206	if (PyErr_WarnFormat(PyExc_DeprecationWarning, 1,
1189	1207	"invalid escape sequence '\\%c'",
1190	-	c) < 0)
1208	+	first_invalid_escape_char) < 0)
1191	1209	{
1192	1210	Py_DECREF(result);
1193	1211	return NULL;
↓		

∨	Objects/unicodeobject.c	...
↕		@@ -6046,13 +6046,15 @@ PyUnicode_AsUTF16String(PyObject *unicode)
6046	6046	/* --- Unicode Escape Codec -----
		*/
6047	6047	
6048	6048	PyObject *
6049	-	_PyUnicode_DecodeUnicodeEscapeInternal(const char *s,
6049	+	_PyUnicode_DecodeUnicodeEscapeInternal2(const char *s,
6050	6050	Py_ssize_t size,
6051	6051	const char *errors,
6052	6052	Py_ssize_t *consumed,
6053	-	const char **first_invalid_escape)
6053	+	int *first_invalid_escape_char,
6054	+	const char **first_invalid_escape_ptr)
6054	6055	{
6055	6056	const char *starts = s;
6057	+	const char *initial_starts = starts;
6056	6058	_PyUnicodeWriter writer;
6057	6059	const char *end;
6058	6060	PyObject *errorHandler = NULL;
↕		@@ -6061,7 +6063,8 @@ _PyUnicode_DecodeUnicodeEscapeInternal(const char
		*s,
6061	6063	PyInterpreterState *interp = _PyInterpreterState_Get();

```

6062 6064
6063 6065 // so we can remember if we've seen an invalid escape char or not
6064 - *first_invalid_escape = NULL;
6066 + *first_invalid_escape_char = -1;
6067 + *first_invalid_escape_ptr = NULL;
6065 6068
6066 6069 if (size == 0) {
6067 6070 if (consumed) {
@@ -6149,9 +6152,12 @@ _PyUnicode_DecodeUnicodeEscapeInternal(const char
*s,
6149 6152 }
6150 6153 }
6151 6154 if (ch > 0377) {
6152 - if (*first_invalid_escape == NULL) {
6153 - *first_invalid_escape = s-3; /* Back up 3 chars, since
we've
6154 - already incremented s. */
6155 + if (*first_invalid_escape_char == -1) {
6156 + *first_invalid_escape_char = ch;
6157 + if (starts == initial_starts) {
6158 + /* Back up 3 chars, since we've already incremented
s. */
6159 + *first_invalid_escape_ptr = s - 3;
6160 + }
6155 6161 }
6156 6162 }
6157 6163 WRITE_CHAR(ch);
@@ -6252,9 +6258,12 @@ _PyUnicode_DecodeUnicodeEscapeInternal(const char
*s,
6252 6258 goto error;
6253 6259
6254 6260 default:
6255 - if (*first_invalid_escape == NULL) {
6256 - *first_invalid_escape = s-1; /* Back up one char, since we've
6257 - already incremented s. */
6261 + if (*first_invalid_escape_char == -1) {
6262 + *first_invalid_escape_char = c;
6263 + if (starts == initial_starts) {
6264 + /* Back up one char, since we've already incremented s.
*/

```

```

6265 +             *first_invalid_escape_ptr = s - 1;
6266 +         }
6258 6267     }
6259 6268         WRITE_ASCII_CHAR('\');
6260 6269         WRITE_CHAR(c);
6293 6302     return NULL;
6294 6303 }
6295 6304
6305 + // Export for binary compatibility.
6306 + PyObject *
6307 + _PyUnicode_DecodeUnicodeEscapeInternal(const char *s,
6308 +                                         Py_ssize_t size,
6309 +                                         const char *errors,
6310 +                                         Py_ssize_t *consumed,
6311 +                                         const char **first_invalid_escape)
6312 + {
6313 +     int first_invalid_escape_char;
6314 +     return _PyUnicode_DecodeUnicodeEscapeInternal2(
6315 +         s, size, errors, consumed,
6316 +         &first_invalid_escape_char,
6317 +         first_invalid_escape);
6318 + }
6319 +
6296 6320 PyObject *
6297 6321 _PyUnicode_DecodeUnicodeEscapeStateful(const char *s,
6298 6322                                         Py_ssize_t size,
6299 6323                                         const char *errors,
6300 6324                                         Py_ssize_t *consumed)
6301 6325 {
6302 -     const char *first_invalid_escape;
6303 -     PyObject *result = _PyUnicode_DecodeUnicodeEscapeInternal(s, size,
6326 +     int first_invalid_escape_char;
6327 +     const char *first_invalid_escape_ptr;
6328 +     PyObject *result = _PyUnicode_DecodeUnicodeEscapeInternal2(s, size,
6304 6329                                         consumed,
6305 -                                         &first_invalid_escape);

```

```

6330 +
        &first_invalid_escape_char,
6331 +
        &first_invalid_escape_ptr);
6306 6332         if (result == NULL)
6307 6333         return NULL;
6308 -         if (first_invalid_escape != NULL) {
6309 -             unsigned char c = *first_invalid_escape;
6310 -             if ('4' <= c && c <= '7') {
6334 +             if (first_invalid_escape_char != -1) {
6335 +                 if (first_invalid_escape_char > 0xff) {
6311 6336                     if (PyErr_WarnFormat(PyExc_DeprecationWarning, 1,
6312 -                                     "invalid octal escape sequence '\\%.3s'",
6313 -                                     first_invalid_escape) < 0)
6337 +                                     "invalid octal escape sequence '\\%0'",
6338 +                                     first_invalid_escape_char) < 0)
6314 6339                     {
6315 6340                         Py_DECREF(result);
6316 6341                         return NULL;
@@ -6319,7 +6344,7 @@ _PyUnicode_DecodeUnicodeEscapeStateful(const char
*s,
6319 6344         else {
6320 6345             if (PyErr_WarnFormat(PyExc_DeprecationWarning, 1,
6321 6346                                     "invalid escape sequence '\\%c'",
6322 -                                     c) < 0)
6347 +                                     first_invalid_escape_char) < 0)
6323 6348             {
6324 6349                 Py_DECREF(result);
6325 6350                 return NULL;

```

Parser/string_parser.c

```

@@ -181,15 +181,18 @@ decode_unicode_with_escapes(Parser *parser, const char
*s, size_t len, Token *t)
181 181     len = p - buf;
182 182     s = buf;
183 183
184 -     const char *first_invalid_escape;
185 -     v = _PyUnicode_DecodeUnicodeEscapeInternal(s, len, NULL, NULL,
        &first_invalid_escape);

```

```

184 +     int first_invalid_escape_char;
185 +     const char *first_invalid_escape_ptr;
186 +     v = _PyUnicode_DecodeUnicodeEscapeInternal2(s, (Py_ssize_t)len, NULL, NULL,
187 +                                               &first_invalid_escape_char,
188 +                                               &first_invalid_escape_ptr);
186 189
187 190         // HACK: later we can simply pass the line no, since we don't preserve the
        tokens
188 191         // when we are decoding the string but we preserve the line numbers.
189 -     if (v != NULL && first_invalid_escape != NULL && t != NULL) {
190 -         if (warn_invalid_escape_sequence(parser, s, first_invalid_escape, t) <
        0) {
191 -             /* We have not decref u before because first_invalid_escape points
192 -                inside u. */
192 +     if (v != NULL && first_invalid_escape_ptr != NULL && t != NULL) {
193 +         if (warn_invalid_escape_sequence(parser, s, first_invalid_escape_ptr,
        t) < 0) {
194 +             /* We have not decref u before because first_invalid_escape_ptr
195 +                points inside u. */
193 196                 Py_XDECREF(u);
194 197                 Py_DECREF(v);
195 198                 return NULL;
@@ -202,14 +205,17 @@ decode_unicode_with_escapes(Parser *parser, const char
*s, size_t len, Token *t)
202 205     static PyObject *
203 206     decode_bytes_with_escapes(Parser *p, const char *s, Py_ssize_t len, Token *t)
204 207     {
205 -     const char *first_invalid_escape;
206 -     PyObject *result = _PyBytes_DecodeEscape(s, len, NULL,
        &first_invalid_escape);
208 +     int first_invalid_escape_char;
209 +     const char *first_invalid_escape_ptr;
210 +     PyObject *result = _PyBytes_DecodeEscape2(s, len, NULL,
        &first_invalid_escape_char,
212 +     &first_invalid_escape_ptr);
207 213         if (result == NULL) {
208 214             return NULL;
209 215         }
210 216
211 -     if (first_invalid_escape != NULL) {

```

```
212 -     if (warn_invalid_escape_sequence(p, s, first_invalid_escape, t) < 0) {
217 +     if (first_invalid_escape_ptr != NULL) {
218 +         if (warn_invalid_escape_sequence(p, s, first_invalid_escape_ptr, t) <
    0) {
213 219             Py_DECREF(result);
214 220             return NULL;
215 221         }
    ↓
```

Comments 0