UNICODE HANDLING

Practice Session – Week 11

What is a character?

IT IS NOT A BIT.

It is a **unit of information** like:
Letter,
Digit,
Period,
Punctuation,
Math symbols,
Control characters - typically not visible.

TEXT is a sequence of characters.

What is a bit?

A unit of information in computing and digital communications.
A bit is simply a “0” or a “1”.
8 Bits = 1 Byte
11001100 → 1 byte
11110000 → 1 byte
11110000 11001100 → 2 bytes
...

A need for one unique representation of characters.

Your friend writes a program where for each day of the week he assigns a numeric value:
Monday = 1
Tuesday =2
Wednesday = 3
….
Sunday = 7

You also write a program where you assign each day of the week to a numeric value:
Monday = 7
Tuesday = 6
Wednesday = 5
…..
Sunday = 1

Your friend tells you to meet at 15 o'clock at day 7 of the week. (Day 7 for him means Sunday)
However, for you day 7 means Monday. A misunderstanding has come up.
Here comes the idea for a universal representation of characters.

ASCII - American Standard Code for Information Interchange

- Established in 1968
- Characters are mapped to numeric codes
- It represents a character in 7 bits
- Since there are 2 possible values for a bit (0 or 1), it means we have 2^7 = 128 possible combinations.
- ASCII maps value from 0 until 127 to 128 characters.

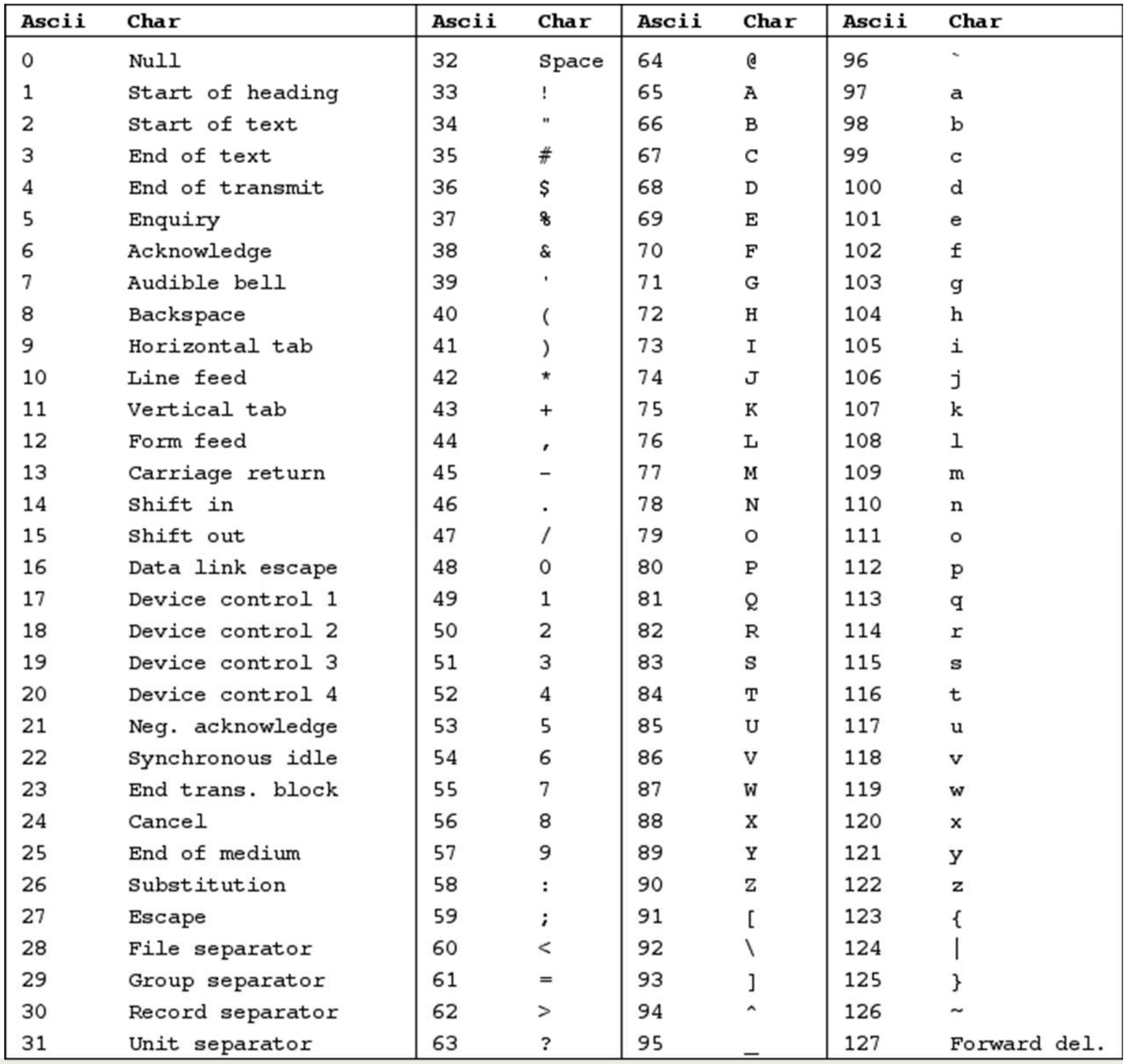
For example, ‘a’ in ASCII has a code point 97, ‘b’ has a code point of 98 and so on..

But, since this is an American-developed standard it cannot represent accented characters like ‘é’ or ‘Í’.

Upper - case (A-Z)	26
Digits (0-9)	10
Space	1
Punctuation marks (.,?{%)	32
Lower-case (a-z)	26
Control characters (tab, cl, if)	33
TOTAL	128

ASCII examples

A → code point 65 → 01000001
B → code point 66 → 01000010
9 → code point 57 →00111001



Ascii	Char	Ascii	Char	Ascii	Char	Ascii	Char
0	Null	32	Space	64	@	96	`
1	Start of heading	33	!	65	A	97	a
2	Start of text	34	"	66	B	98	b
3	End of text	35	#	67	C	99	c
4	End of transmit	36	$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledge	38	&	70	F	102	f
7	Audible bell	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Horizontal tab	41	)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift in	46	.	78	N	110	n
15	Shift out	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Neg. acknowledge	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End trans. block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitution	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	File separator	60	<	92	\	124	|
29	Group separator	61	=	93	]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	Forward del.

Problems with ASCII?

- In 1980 most computers were 8-bits, meaning we could represent 2^8 = 256 values using a byte (8 bits).
- Since ASCII only went up to 127, different countries assigned values 128 to 255 for different accented characters.
- Different machines had different code which led to problems exchanging files.
- 255 characters are not enough if we want to represent all the characters from Japanese, Chinese, Turkish or basically all alphabets of the world.
- We could use one coding system for Turkish, and another one for Chinese but that means we could not write a Turkish quote in our Chinese text.
- Hence, UNICODE is the solution.

UNICODE

- One Universal Code for every character:
  - No matter what the platform.
  - No matter what the program,
  - No matter what the language

Principles of UNICODE:

- Universality (any language, bidirectional scripts: Hebrew, Arabic)
- Unification (avoid duplicate encoding of characters within scripts across languages. Character code U+0057 “Y” is same in English, German, French, etc.)
- **We can represent 1,114,112 characters.**

WHY WE NEED STANDARDIZATION

Why do we need to unify coding standards?

- **When electronic information is received from one place to another place where the systems use different coding standards, such information may become mis-coded or incorrectly displayed even if code conversion is applied.**

UNICODE AND CHARACTERS

Character is a representation of a single symbol in a piece of text.
UNICODE is a way of defining a set of characters that everyone can agree on.

- It has a huge database of characters.
- Each character is associated with a unique number, called code point.
- A text string is a series of these codepoints, representing the character for each element in the string.

STRING TYPES

Three distinct string types:

- unicode : represents unicode strings (text strings)
- Str: represents byte strings (binary data)- using 8-bits
- Basestring: acts as parent class for both the other string types

With the help of character encoding we can interchange between bytes and characters.

UNICODE OR BYTE str

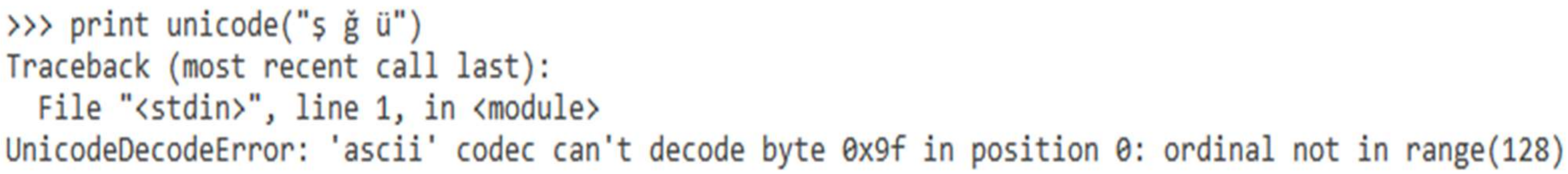
UNICODE strings: used when dealing with text manipulations such as:

Finding the number of characters
Cutting the string

BYTE string: used when devices need to deal with concrete implementation of what bytes represent the abstract characters::

Dealing with Input/Output
Reading to/ from the disk
Printing to a terminal

UNICODE in PYTHON



```
>>> print unicode("ş ğ ü")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0x9f in position 0: ordinal not in range(128)
```

Since default encoding is ASCII and there is no valid representation of the Turkish letters: ş ğ ü in ASCII we will get an UnicodeDecodeError, meaning that python does not know how to decode that character, since it it out of the 128 range.

ENCODING

Rules for translating a UNICODE string into a sequence of bytes are called an **encoding.** Or simply, it is a set of rules that assign numeric values to each text character.

Encoding default is ASCII on most platforms.

sys.setdefaultencoding() → can be used to set the default encoding to whatever you want. It may be impractical, since many apps, may have to deal with different text encodings in different places.

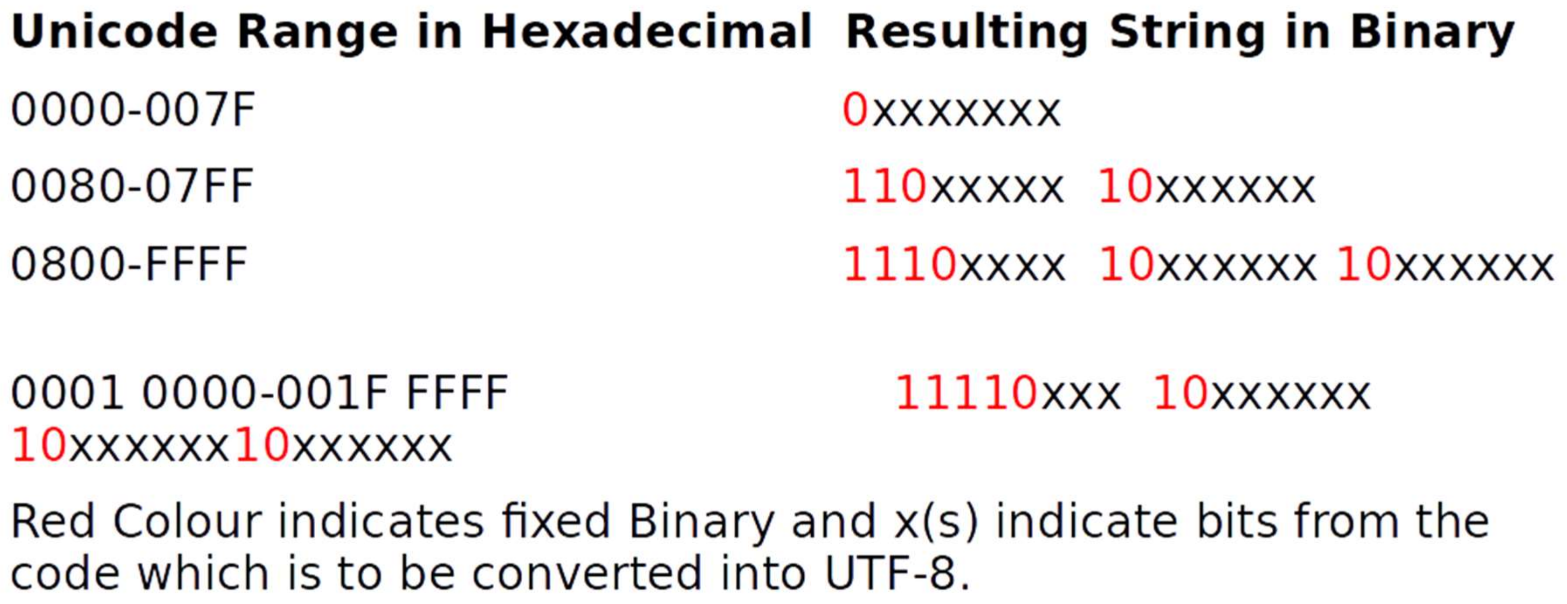
UTF - 8 ENCODING

UTF-8 is one of the most commonly used encodings. Stands for Unicode Transformation Format
UTF-8 is an extension of ASCII, means that the first 128 code points are assigned to the same characters as in ASCII.
UTF- 8 has dynamic length, meaning that it takes from 1 to 4 bytes to represent characters.

Rules of UTF -8:

- If code point is < 128, it’s represented by the corresponding byte value (ASCII).
- If code point is between 128 and 2047, it’s turned into two byte values between 128 and 225.
- If code point is between 2047 and 65535, it’s turned into three byte sequences.
- If code point is between 65535 and 1114111, it’s turned into four byte sequences.

UTF - 8 ENCODING



Unicode Range in Hexadecimal	Resulting String in Binary
0000-007F	0xxxxxxx
0080-07FF	110xxxxx 10xxxxxx
0800-FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-001F FFFF	11110xxx 10xxxxxx 10xxxxxx10xxxxxx

Red Colour indicates fixed Binary and x(s) indicate bits from the code which is to be converted into UTF-8.

IMPORTANT METHODS

The ord() function tells us the numeric value of a simple ASCII character

The unichr() function takes an integer and returns a unicode string that contains the corresponding code point

```
>>> print( ord( ‘H’ ), unichr( 72 ) )
( 72 u’H’ )
>>> s.decode(encoding)
```

- <type ‘str’> to <type ‘unicode’>

```
>>> u.encode(encoding)
```

- <type ‘unicode’> to <type ‘str’>

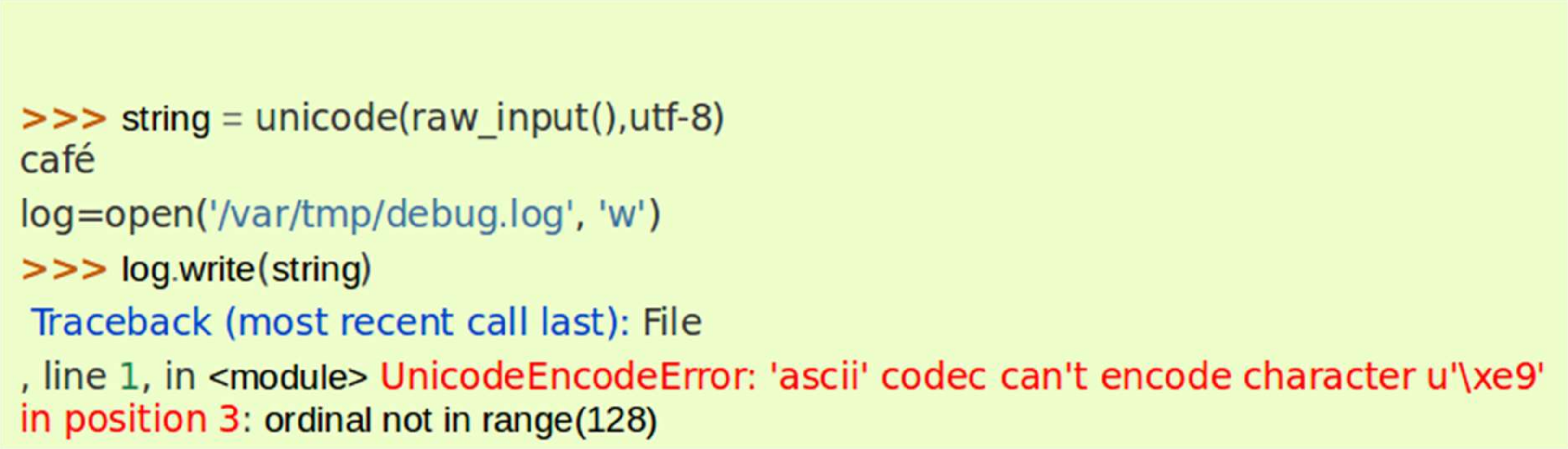
Inconsistent error 1

When entering:
>>> “The quick brown fox jumped over the lazy dog.”
No error is encountered.
However, when entering:
>>> “İyiyim, teşekkür ederim!”
We see an exception error.

Why? When we write non-ASCII characters into our strings, we need to handle the conversion manually, since the mechanism that converts between the two types is only able to deal with ASCII characters.

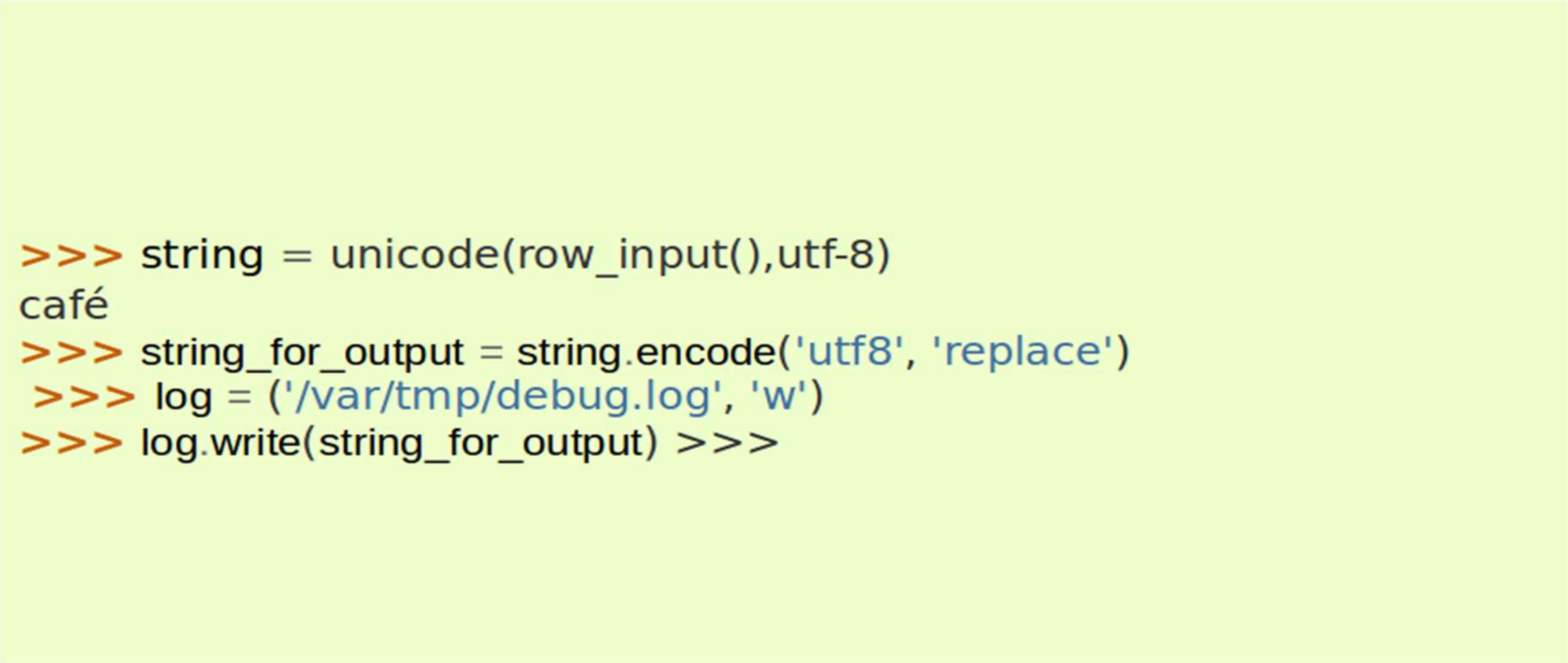
Example

Anytime you output text to the terminal or to a file, the text has to be converted into a byte str. Python will try to implicitly convert from unicode to byte str..but it will throw an exception if the bytes are non-ASCII:



```
>>> string = unicode(raw_input(),utf-8)
café
log=open('/var/tmp/debug.log', 'w')
>>> log.write(string)
 Traceback (most recent call last): File
, line 1, in <module> UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9'
in position 3: ordinal not in range(128)
```

Solution



```
>>> string = unicode(row_input(),utf-8)
café
>>> string_for_output = string.encode('utf8', 'replace')
 >>> log = ('/var/tmp/debug.log', 'w')
>>> log.write(string_for_output) >>>
```

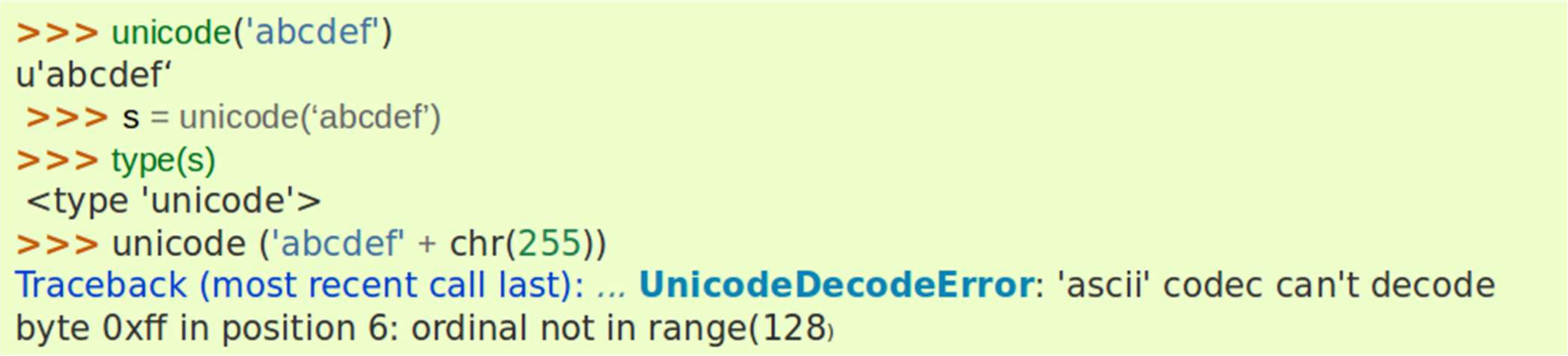
What happens when you use print??

Since the **terminal is a file-like object** it should raise an exception if you do not encode the string.

```
>>> string = unicode( raw_input(), utf-8)
café
>>> print string.encode(‘utf-8’ , ‘replace’)
café
```

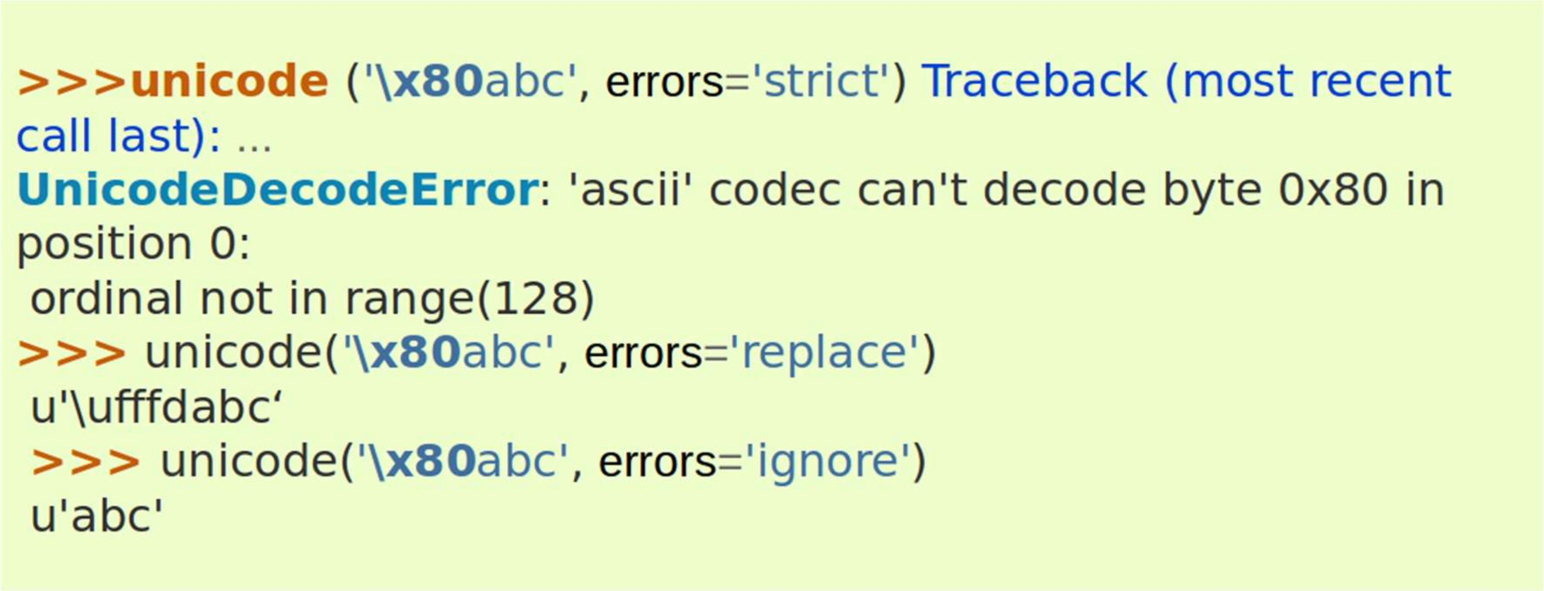
UNICODE TYPE

unicode() constructor has the signature:
unicode (string,[encoding, errors])
All of its arguments should be 8-bit strings. The first argument is converted to Unicode using the specified encoding; if no encoding is specified, the ASCII encoding is used for the conversion meaning that characters greater than 127 will be treated as errors:



```
>>> unicode('abcdef')
u'abcdef‘
 >>> s = unicode(‘abcdef’)
>>> type(s)
 <type 'unicode'>
>>> unicode ('abcdef' + chr(255))
Traceback (most recent call last): ... UnicodeDecodeError: 'ascii' codec can't decode
byte 0xff in position 6: ordinal not in range(128)
```

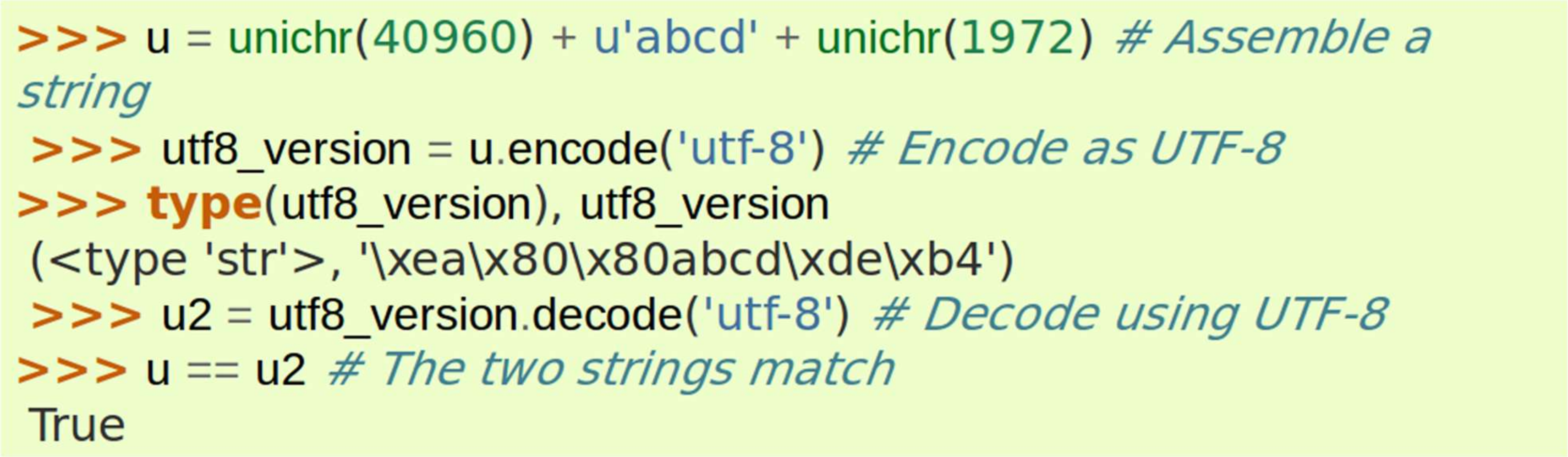
UNICODE ERROR PARAMETERS



```
>>>unicode ('\x80abc', errors='strict') Traceback (most recent
call last): ...
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in
position 0:
 ordinal not in range(128)
>>> unicode('\x80abc', errors='replace')
 u'\ufffdabc‘
 >>> unicode('\x80abc', errors='ignore')
 u'abc'
```

Decode method

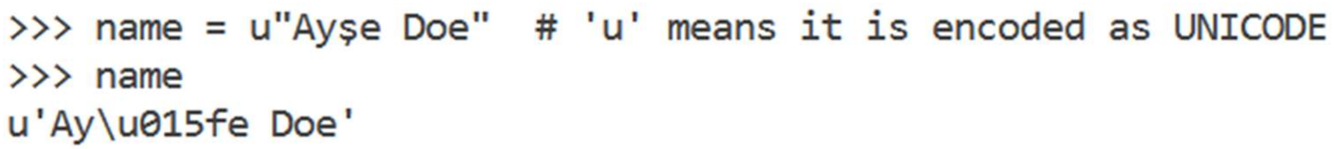
Python’s 8-bit strings have a .decode([encoding],[errors]) method that interprets the string using the given encoding:



```
>>> u = unichr(40960) + u'abcd' + unichr(1972) # Assemble a
string
 >>> utf8_version = u.encode('utf-8') # Encode as UTF-8
>>> type(utf8_version), utf8_version
 (<type 'str'>, '\xea\x80\x80abcd\xde\xb4')
 >>> u2 = utf8_version.decode('utf-8') # Decode using UTF-8
>>> u == u2 # The two strings match
 True
```

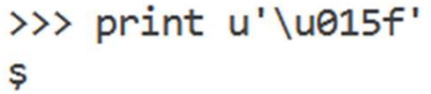
EXAMPLE CODE

Define a variable called ‘name’ and use a non-ASCII character



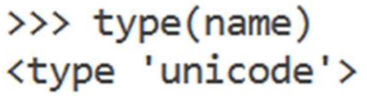
```
>>> name = u"Ayşe Doe"  # 'u' means it is encoded as UNICODE
>>> name
u'Ay\u015fe Doe'
```

‘ş’ becomes ‘\u015f’. What happens if we print ‘\u015f’



```
>>> print u'\u015f'
ş
```

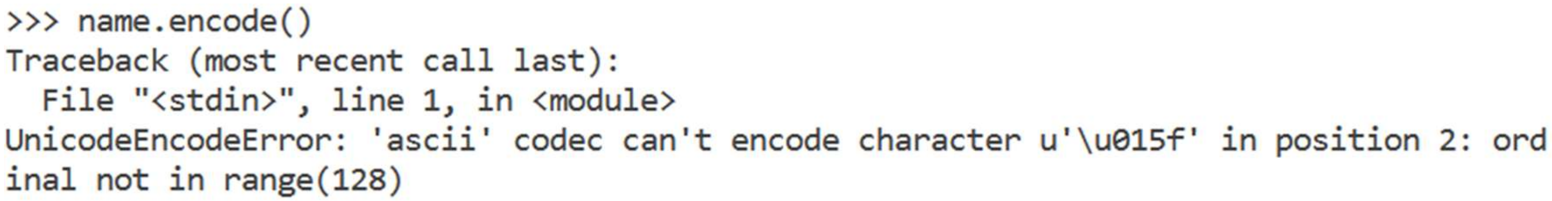
So, we can print unicode characters. But they are still unicode.



```
>>> type(name)
<type 'unicode'>
```

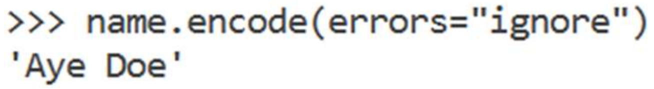
UNICODE TO ASCII

Unicode can be converted into ASCII, but has more characters



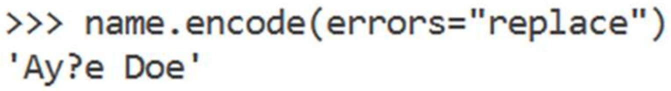
```
>>> name.encode()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\u015f' in position 2: ord
inal not in range(128)
```

‘ş’ can be ignored. (Can’t convert a char to ASCII? Just delete it)



```
>>> name.encode(errors="ignore")
'Aye Doe'
```

Or it can be replaced. (Those ‘?’ you see when you open a file)



```
>>> name.encode(errors="replace")
'Ay?e Doe'
```

For more information on Unicode in Python:
https://docs.python.org/2/howto/unicode.html