

03 - Cadenas en Python 3

Diego Andrés Álvarez Marín
Profesor Asociado

Universidad Nacional de Colombia
Sede Manizales

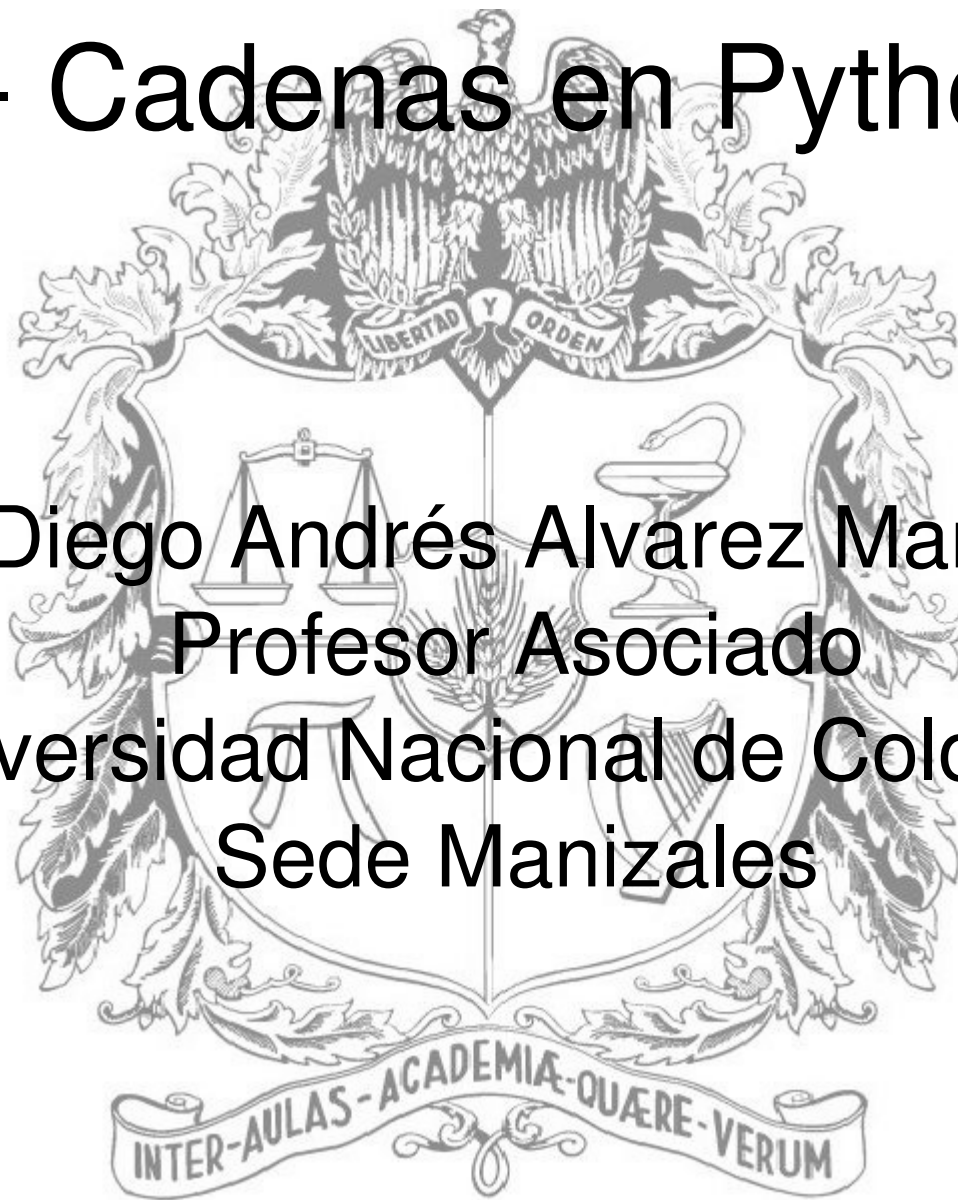


Tabla de contenido

- Definición de cadenas
- Cadenas de varias líneas
- Secuencias de escape
- Raw strings
- ASCII vs UNICODE
- UTF-8
- f-strings
- Indexado y slicing
- Referencias a cadenas, copias de cadenas
- Variables “intern”

Tabla de contenido

Operadores

., +, *

[:, [i], [i:j], [i:j:k]

is, is not

in, not in

=, ==, !=

<, >, <=, >=

Funciones:

len()

id()

chr(), ord()

int(), float(), str()

Métodos:

lower(), upper(),
title(), replace(),
find(), join(),
split(),
startswith(),
endswith(), ljust(),
rjust(), center(),
zfill(), isdigit(),
isupper(),
islower(),
isalpha()

Tipos de datos

Los tipos de datos determinan el conjunto de valores que un objeto puede tomar y las operaciones que se pueden realizar con ellas.

- Tipos de datos escalares:
 - Números enteros, flotantes, complejos, fraccionarios, lógicos(booleanos)
- Tipos de datos secuenciales:
 - Secuencias de bytes, **cadenas**
- Tipos de datos estructurados:
 - Listas (lists): secuencias ordenadas de valores
 - Tuplas (tuples): secuencias inmutables de valores ordenados
 - Conjuntos (sets): conjunto no ordenado de valores
 - Diccionarios (dictionaries): conjunto no ordenado de valores, que tienen una “llave” que los identifican
- Objetos: módulos, funciones, clases, métodos, archivos, código compilado, etc.
- “Constantes”

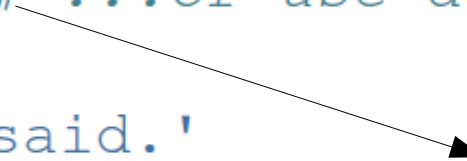
Secuencias de bytes

- Cadenas: son secuencias de caracteres UNICODE
- Bytes y arrays de bytes: por ejemplo, una imagen .jpg, un .mp3, etc. Para estos dos tipos de datos se invita al lector a consultar <https://realpython.com/python-strings/>

Cadenas

En Python existe el tipo de dato cadena, pero no existe el tipo de dato char (que si existe en C). Lo más parecido a un char de lenguaje C sería entonces una cadena de longitud 1.

```
>>> 'spam eggs'    # single quotes
'spam eggs'
>>> 'doesn\'t'     # use \' to escape the single quote...
"doesn't"
>>> "doesn't"      # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

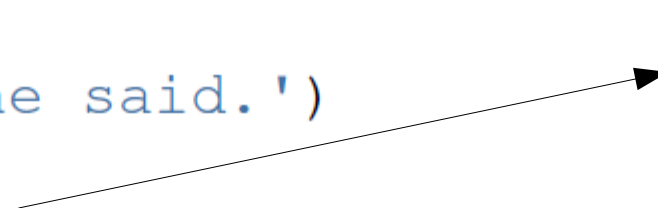


Los comentarios en Python empiezan con #

Se puede utilizar comillas simples o comillas dobles indistintamente⁶

Definiendo cadenas

```
>>> "Isn't," she said.  
"Isn't," she said.  
>>> print("Isn't," she said.)  
Isn't," she said.  
>>> s = 'First line.\nSecond line.' # \n means newline  
>>> s # without print(), \n is included in the output  
'First line.\nSecond line.'  
>>> print(s) # with print(), \n produces a new line  
First line.  
Second line.
```



`\n` significa nueva línea

```

>>> print("""\
Este es un texto de
varios reglones""")
Este es un texto de
varios reglones
>>> print('''\
Este es un texto de
varios reglones''')
Este es un texto de
varios reglones
>>> print("""
Este es un texto de
varios reglones
""")
Este es un texto de
varios reglones

>>> print('''
Este es un texto de
varios reglones
''')
Este es un texto de
varios reglones

>>> |

```

Cadenas en varias líneas

Las comillas simples o dobles se usan indistintamente

```

>>> texto = ('Este es '
             'un texto '
             'de varias '
             'líneas.')

>>> texto
'Este es un texto de varias líneas.'
>>> print(texto)
Este es un texto de varias líneas.
>>>

>>> print('Este \
es un texto \
de una sola \
línea')
Este es un texto de una sola línea

```


Sangrado en cadenas de varias líneas

```
1 def escribir_carta():
2     print('''Estimada Ana,
3     ¿Como estás? Espero que muy bien.
4     Escribeme pronto que quiero saber de ti.
5     Con cariño,
6     Jorge''')
7     print('P.S. Te extraño mucho.')
8
9 escribir_carta() # esta línea no pertenece a la función
```

```
Line: 11 of 11 Col: 1    LINE  INS
daalvarez@eredron:~ > python3 02_cadenas.py
Estimada Ana,
¿Como estás? Espero que muy bien.
Escribeme pronto que quiero saber de ti.
Con cariño,
Jorge
P.S. Te extraño mucho.
daalvarez@eredron:~ > █
```

Observe que en las cadenas de varias líneas (multi-line strings) no se conserva la indentación del bloque que contiene la cadena.

Secuencias de escape

Secuencia de escape
para carácter de control

Resultado

decimal	hexadec
7	\x07
8	\x08
12	\x0C
10	\x0A
13	\x0D
9	\x09
11	\x0B

\a Carácter de «campana» (BEL)

\b «Espacio atrás» (BS)

\f Alimentación de formulario (FF)

\n Salto de línea (LF)

\r Retorno de carro (CR)

\t Tabulador horizontal (TAB)

\v Tabulador vertical (VT)

\ooo Carácter cuyo código en octal es *ooo*

\xhh Carácter cuyo código en hexadecimal es *hh*

```
>>> ord('Z')
90
>>> oct(90)
'0o132'
>>> print('\132')
Z
```



Otras secuencias de escape

Resultado

\\ Carácter barra invertida (\)

\' Comilla simple (')

\" Comilla doble (")

\ y salto de línea Se ignora (para expresar una cadena en varias líneas)

Secuencias de escape

```
>>> print('12345678'*5)
1234567812345678123456781234567812345678
>>> print('1\t2\tyxyz\t3\t\rst\t\t\t')
1      2\tyxyz  3      rst      ttt
>>> cad = 'Una\ncadena\nde\nvarias\tlíneas'
>>> cad
'Una\ncadena\nde\nvarias\tlíneas'
>>> print(cad)
Una
cadena
de
varias  líneas
>>>

>>> ord('\n')
10
>>> oct(10)
'0o12'
>>> hex(10)
'0xa'
>>> cad = 'Una\ncadena\012de\xavarias\tlíneas'
SyntaxError: (unicode error) 'unicodeescape' codec can't decode
bytes in position 17-19: truncated \xXX escape
>>> cad = 'Una\ncadena\012de\x0avarias\tlíneas'
>>> cad
'Una\ncadena\nde\nvarias\tlíneas'
>>> print(cad)
Una
cadena
de
varias  líneas
>>> |
```

Aquí está el error

Es cero 12 no "o" 12

Secuencias de escape

Archivo: 03_ejemplo_contador_v1.py

```
1 from time import sleep
2 import sys
3
4 tiempo = 0.05 #segundos
5
6 # Observe como colocar varios comandos en una sola línea. Esto de todos
7 # modos no se recomienda en Python:
8 # Ver: https://www.python.org/dev/peps/pep-0008/
9 # Aquí simplemente lo hago porque mejora la claridad del código
10
11 ▼ for i in range(101):
12     print('%3d%% \\' % i, end = ''); sys.stdout.flush(); sleep(tiempo)
13     print('\b|', end = ''); sys.stdout.flush(); sleep(tiempo)
14     print('\b/', end = ''); sys.stdout.flush(); sleep(tiempo)
15     print('\b-', end = ''); sys.stdout.flush(); sleep(tiempo)
16     print('\r', end = '');
17
18 # Aquí toca poner el sys.stdout.flush(), porque normalmente el buffer
19 # de salida solo se imprime cuando aparece un \n
20
21 print('FIN!\a')
```

Secuencias de escape

A partir de Python 3.4, el siguiente código es válido:

Archivo: 03_ejemplo_contador_v2.py

```
1 from time import sleep
2
3 tiempo = 0.05 #segundos
4
5 # Observe como colocar varios comandos en una sola línea. Esto de todos
6 # modos no se recomienda en Python:
7 # Ver: https://www.python.org/dev/peps/pep-0008/
8 # Aquí simplemente lo hago porque mejora la claridad del código
9
10 ▼ for i in range(101):
11     print('%3d%% \\' % i, end = '', flush = True); sleep(tiempo)
12     print('\b|', end = '', flush = True); sleep(tiempo)
13     print('\b/', end = '', flush = True); sleep(tiempo)
14     print('\b-', end = '', flush = True); sleep(tiempo)
15     print('\r', end = '', flush = True)
16
17 # Aquí toca poner el "flush = True", porque normalmente el buffer
18 # de salida solo se imprime cuando aparece un \n
19
20 print('FIN!\a')
```

Raw strings

```
>>> print("C:\programas\nombre")
```

```
C:\programas
```

```
ombre
```

```
>>> print("C:\\programas\\nombre")
```

```
C:\\programas\\nombre
```

```
>>> print(r"C:\programas\nombre")
```

```
C:\programas\nombre
```

→ **Raw string**: imprime textualmente la cadena de texto

Escribiendo comandos super largos en varias líneas

Es usual entre los programadores que los códigos no sobrepasen la línea 80. Por lo tanto, a veces es necesario dividir el código en varias líneas. Esto se puede hacer con la ayuda del \

```
>>> x = 5
>>> cad = str(123) + \
' es un número y ' + \
('xyz' if x<0 else 'abc') + \
' es una cadena'
>>> print(cad)
123 es un número y abc es una cadena
```

ASCII

Es un código de caracteres basado en el alfabeto latino, tal como se usa en inglés moderno. Fue creado en 1963 por el ANSI como una reimplementación o evolución de los conjuntos de códigos utilizados entonces en telegrafía. Se reconoció como estándar en 1967.

Utiliza 7 bits para representar los caracteres, aunque inicialmente empleaba un bit adicional (bit de paridad) que se usaba para detectar errores en la transmisión.

Contiene 32 caracteres no imprimibles, de los cuales la mayoría son caracteres de control y 95 caracteres imprimibles.

NOTA: a menudo se llama incorrectamente ASCII a otros códigos de caracteres de 8 bits, como el estándar ISO-8859-1, que es una extensión que utiliza 8 bits para proporcionar caracteres adicionales usados en idiomas distintos al inglés, como el español.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

El estándar de codificación de caracteres UNICODE

Se diseñó para facilitar el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas, además de textos clásicos de lenguas muertas. Unicode incluye sistemas de escritura modernos como: árabe, braille, copto, cirílico, griego, sinogramas (hanja coreano, hanzi chino y kanji japonés), silabarios japoneses (hiragana y katakana), hebreo y latino; escrituras históricas extintas, para propósitos académicos, como por ejemplo: cuneiforme, griego antiguo, fenicio y rúnico. Entre los caracteres no alfabéticos incluidos en Unicode se encuentran símbolos musicales y matemáticos, fichas de juegos como el dominó, flechas, iconos, emoticonos, etc.
































































































































































El estándar de codificación de caracteres UNICODE

- Se adoptó como norma ISO en 1993, como una extensión de los códigos ASCII.
- Se tiene espacio para 1'114.112 símbolos posibles (del 0x000000 al 0x10FFFF). Actualmente se usa la versión 11.0 (Junio de 2018). Esta versión define 137.439 símbolos.
- Los puntos de código se representan utilizando notación hexadecimal agregando el prefijo U+. El valor hexadecimal se completa con ceros hasta 4 dígitos hexadecimales cuando es necesario; si es de longitud mayor que 4 dígitos no se agregan ceros.

Los símbolos UNICODE

<http://www.unicode.org/charts/>

- Se escriben como U+XXXX o como U+XXXXXX donde X es un hexadecimal (esto se llama el **code point**).
- En C/Python se representa tal Unicode como \uXXXX (hasta 4 hexadecimales) o como \UXXXXXX (hasta 6 hexadecimales).
- En GNU/Linux se escribe con **Ctrl+Shift+u+codepoint** ENTER. Cuando esto se hace la letra u aparece subrayada. Se debe tener una fuente apropiada instalada.
- En MS Windows: ver la utilidad "**charmap**". Se debe tener una fuente apropiada instalada.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2600																
2610																
2620																
2630																
2640																
2650																
2660																
2670																
2680																
2690																

Unicode

(por falta de fuentes apropiadas no funciona en la consola de Windows)

```
1 print("áÁ éÉ íÍ óÓ úÚ üÜ ñÑ")
2 print("مرحبا");           # Hola en árabe
3 print("Բարեւ Ձեզ");      # Hola en armenio
4 print("こんにちは");      # Hola en japonés
5 print("\u2602 \u265A \u263A");
```

Line: 7 of 7 Col: 1 LINE INS 02_unicode.py UTF-8

```
daalvarez@eredron:~ > python3 02_unicode.py
áÁ éÉ íÍ óÓ úÚ üÜ ñÑ
مرحبا
Բարեւ Ձեզ
こんにちは
☂ 🏰 😊
daalvarez@eredron:~ >
```

Nota: estos caracteres especiales requieren más de un byte para su almacenamiento

De ASCII a UNICODE

Hace algunos años era corriente codificar cada carácter con un byte (ocho bits). La tabla que establecía la correspondencia entre carácter y valor numérico era la denominada tabla ASCII, de la que hemos hablado brevemente en el primer capítulo. La letra a, por ejemplo, tiene valor numérico 97. En realidad no codificaba 256 símbolos, sino solo 128: los que correspondían a valores numéricos entre 0 y 127. Esta tabla, diseñada en 1968, era problemática en países como el nuestro, pues no recogía los caracteres acentuados o la letra ñ. Con 256 caracteres, que es lo que podemos codificar con 8 bits, era imposible tener un juego completo válido para todos los países del mundo. De hecho, ni siquiera para todos los países europeos.

Cada sistema informático extendió la tabla ASCII a su gusto. Usualmente se añadían caracteres a los 128 valores no usados por la tabla ASCII. Esto trajo multitud de problemas a la hora de intercambiar archivos de texto entre sistemas. En los años 90, una comisión estandarizó tablas adaptadas a diferentes dominios lingüísticos. La tabla apropiada para Europa occidental era la denominada ISO-8859-1, también conocida como IsoLatin1 o Latin1. Pronto esta tabla se quedó corta: el símbolo del euro no estaba contemplado en ella. La tabla ISO-8859-15 ampliaba la ISO-8859-1 para recoger el símbolo del euro.

El problema de codificar la información textual estaba lejos de quedar satisfactoriamente resuelto si había que recurrir a multitud de tablas de 256 caracteres. Piénsese en que era imposible, por ejemplo, incluir en un único fichero de texto un fragmento en español con otro en japonés.

Surgió entonces una codificación capaz de resolver el problema definitivamente: la codificación Unicode. Unicode empezó planteando que cada carácter debía codificarse con 16 bits y no con solo 8. Esto hacía que hubiera 65536 códigos disponibles. Como seguían siendo insuficientes para representar cualquier carácter de cualquier lengua, Unicode definió codificaciones con número de bits variable que permitieran, mediante sucesivas extensiones, dar cuenta de cualquier alfabeto existente.

UTF-8

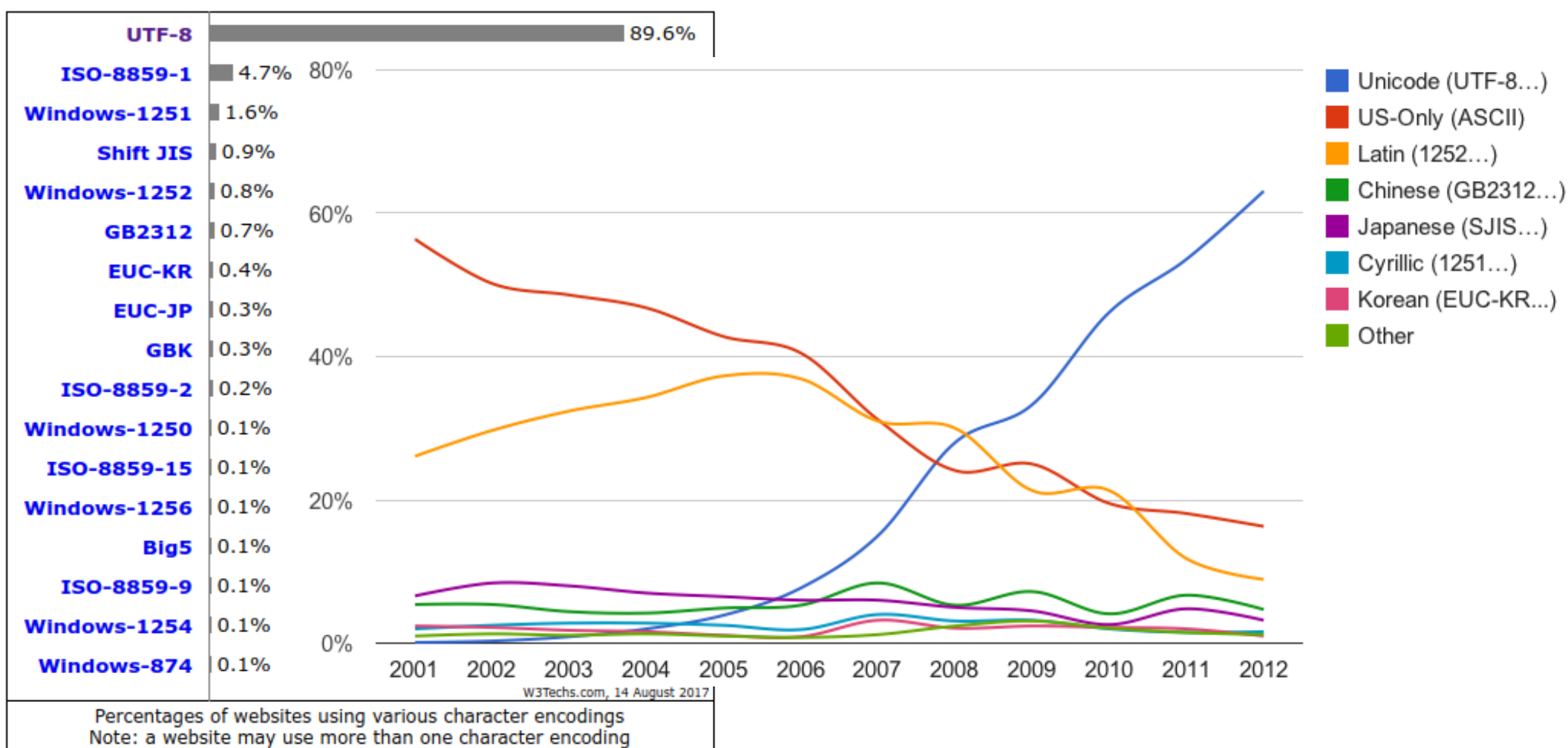
UTF-8 es un formato de codificación de caracteres Unicode utilizando símbolos de longitud variable. Este fue creado en 1993 por Ken Thompson (uno de los padres de UNIX).

Características principales:

- Es capaz de representar cualquier carácter Unicode.
- Usa símbolos de longitud variable (de 1 a 4 bytes por carácter Unicode).
- Incluye la especificación ASCII, por lo que cualquier mensaje ASCII se representa sin cambios.
- Es la codificación de caracteres dominante en Internet. En Enero de 2019, 92.9% de todas las páginas web estaban codificadas con UTF-8.
- (Desventaja) Un texto escrito en un lenguaje asiático almacenado en UTF-8 ocupa muchos más bytes que si se almacena con otra codificación (por ejemplo la UTF-16).

Otras codificaciones de caracteres: ISO/IEC 8859-1, Shift JIS, GB 2312, Windows-1252, etc.

Uso de los sistemas de codificación



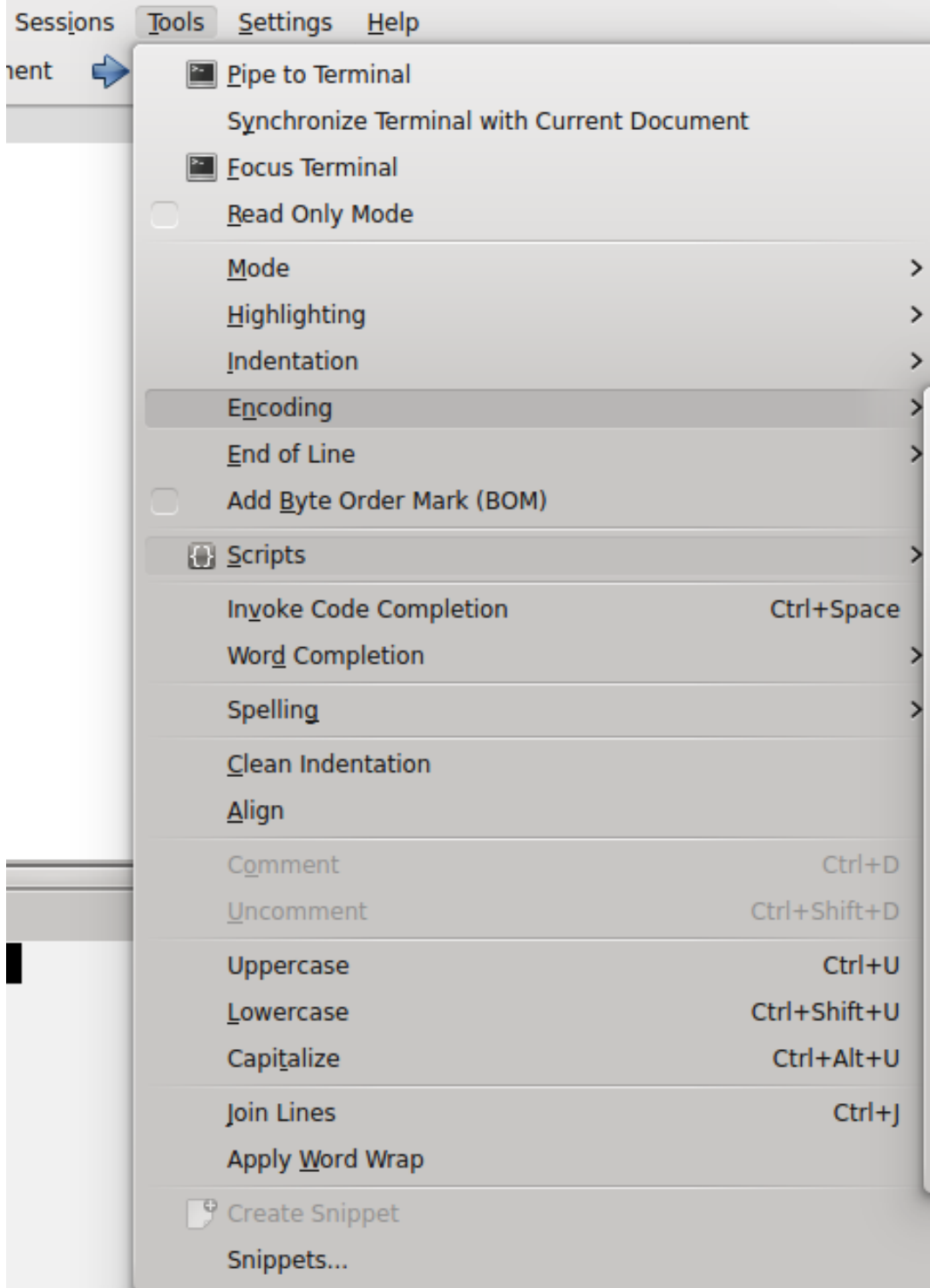
Fuente:

<https://googleblog.blogspot.com.co/2012/02/unicode-over-60-percent-of-web.html>

https://w3techs.com/technologies/overview/character_encoding/all

Selección de la codificación de caracteres UTF-8

Por favor cambie la codificación de caracteres a UTF-8 en su IDE favorito, especialmente si usa Windows.



Los comandos chr() y ord()

```
>>> ord('A')
65
>>> ord('1')
49
>>> ord('a')
97
>>> ord('ñ')
241
>>> ord('Ñ')
209
>>> ord('á')
225
>>> chr(65)
'A'
>>> chr(65+32)
'a'
>>> chr(225)
'á'
>>> ord('⇒')
8652
>>> chr(0x21cc)
'⇒'
>>> chr(8652)
'⇒'
```

- **chr(i)** retorna el carácter Unicode correspondiente al código entrado
 $0 \leq i \leq 0x10ffff$
- **ord(c)** retorna el código Unicode correspondiente al carácter “c” entrado
 $0 \leq \text{ord}(c) \leq 0x10ffff$

Comparación de secuencias

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types with `<` or `>` is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a `TypeError` exception.

Ordenamiento lexicográfico de cadenas

(se aplica de igual forma a listas)

The comparison uses **lexicographical ordering**: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. Los diccionarios se ordenan de esta forma.

http://es.wikipedia.org/wiki/Orden_lexicogr%C3%A1fico

Una aplicación más general del orden lexicográfico es al comparar **cadenas de caracteres**. Distinto al caso para los productos cartesianos n-arios mencionados arriba, las cadenas de caracteres no poseen longitud fija. Usando la misma idea de definición recursiva que para el caso anterior, ahora debemos considerar el que una secuencia puede ser más larga que la otra, y que por lo tanto termine de recorrerse mientras que todavía quedan caracteres en la otra.

La secuencia más corta a considerar será la **cadena vacía** ϵ , es decir:

$$\forall b \in \Sigma^* : \epsilon \leq b$$

Así, la definición recursiva queda:

$$\forall [a_1 \dots a_m], [b_1 \dots b_n] \in \Sigma^* \setminus \{\epsilon\} : [a_1 \dots a_m] \leq [b_1 \dots b_n] \Leftrightarrow a_1 < b_1 \vee (a_1 = b_1 \wedge [a_2 \dots a_m] \leq [b_2 \dots b_n])$$

Comparación de cadenas

```
>>> 'a' < 'aa'
True
>>> 'b' < 'aa'
False
>>> 'A' < 'aa'
True
>>> 'a' < 'Aa'
False
>>> 'abajo' < 'arriba'
True
>>> 'Abajo' < 'arriba'
True
>>> 'Abajo' < 'Arriba'
True
>>> 'abajo' < 'Arriba'
False
>>> 'ábaco' < 'arriba'
False
>>> 'Casa' < 'Arreglar'
False
```

```
>>> ord('a')
97
>>> ord('b')
98
>>> ord('A')
65
>>> ord('C')
67
>>> ord('á')
225
```

El ordenamiento de cadenas se hace de acuerdo con el código Unicode de los caracteres.

Los operadores + y * con cadenas

```
>>> # Las cadenas de texto se pueden concatenar con el
>>> # operador + y repetirse con el operador *
>>> 3*'abc' + 'xyz'
'abcabcabcxyz'
>>>
>>> # dos cadenas encerradas entre comillas simples o dobles
>>> # una al lado de la otra se concatenan inmediatamente
>>> "Buenos" "días"
'Buenosdías'
>>>
>>> # Este truco no sirve con variables o expresiones
>>> Saludo = "Buenos"
>>> Saludo "días"
SyntaxError: invalid syntax
>>> 3*'abc' 'xyz'
'abcxyzabcxyzabcxyz'
>>> 'abc'*3 'xyz'
SyntaxError: invalid syntax
```

Aquí si funcionó. Aunque es recomendable que sea explícito y coloque el +

```
>>> # Por lo tanto si quiere concatenar utilice el +
>>> Saludo + " " + 'días'
'Buenos días'
>>>
```

La cadena vacía

```
>>> vacia = '' # La cadena vacía
```

```
>>> len(vacia)
```

```
0
```

```
>>> print(vacia)
```

```
>>> print('xyz' + vacia + 'xyz')
```

```
xyzxyz
```

```
>>>
```

```
>>>
```

```
>>> espacio = ' ' # El espacio
```

```
>>> len(espacio)
```

```
1
```

```
>>> print('xyz' + espacio + 'xyz')
```

```
xyz xyz
```

```
>>>
```



```
>>> int('noventa')
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    int('noventa')
ValueError: invalid literal for int() with base 10: 'noventa'
```

```
>>> int(90)
90
>>> int('90')
90
>>> int(' 90 ')
90
>>> type(_)
<class 'int'>
>>> 3 + int(' 90 ')
93
>>> float('90')
90.0
>>> type(_)
<class 'float'>
>>> float(' 120.3523 ')
120.3523
>>> int('120.3523')
```

```
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    int('120.3523')
ValueError: invalid literal for int() with base 10: '120.3523'
```

```
>>> str(90)
'90'
>>> str(90.3452)
'90.3452'
>>> type(_)
<class 'str'>
```

Convirtiendo cadenas a números y viceversa

```
>>> str(49.2)
'49.2'
>>> str(3+4j)
'(3+4j)'
>>> str(3 + 29)
'32'
>>> str('foo')
'foo'
```

```
>>> int('42\n')
42
>>> int('    42    \n')
42
```

Interpolación variables en cadenas (f-strings)

Python

```
>>> n = 20
>>> m = 25
>>> prod = n * m
>>> print(f'The product of {n} and {m} is {prod}')
```

The product of 20 and 25 is 500

Este mecanismo se introdujo en Python 3.6

Any of Python's three quoting mechanisms can be used to define an f-string:

Python

```
>>> var = 'Bark'

>>> print(f'A dog says {var}!')
```

A dog says Bark!

```
>>> print(f"A dog says {var}!")
```

A dog says Bark!

```
>>> print(f'''A dog says {var}!''')
```

A dog says Bark!

Más información en: <https://realpython.com/python-f-strings/>

Métodos con cadenas

```
>>> dir(str)
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

La documentación de las funciones de cadenas se encuentra en:
<https://docs.python.org/3/library/stdtypes.html#string-methods>

Los métodos lower(), upper() y title()

```
>>> cadena = 'Hola, ¿cómo estás?'
>>> cadena.lower()
'hola, ¿cómo estás?'
>>> cadena
'Hola, ¿cómo estás?'
>>> saludo = 'Hola, ¿cómo estás?'.upper()
>>> saludo
'HOLA, ¿CÓMO ESTÁS?'
>>> upper('Hola, ¿cómo estás?')
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    upper('Hola, ¿cómo estás?')
NameError: name 'upper' is not defined
>>> 'Hola, ¿cómo estás?'.upper().lower()
'hola, ¿cómo estás?'
>>> 'Hola, ¿cómo estás?'.lower().upper()
'HOLA, ¿CÓMO ESTÁS?'
>>> 'pepito pérez'.title()
'Pepito Pérez'
>>> 'PEPITO PÉREZ'.title()
'Pepito Pérez'
>>> |
```

Los métodos replace() y find()

```
>>> 'un pequeño ejemplo'.replace('pequeño', 'gran')
'un gran ejemplo'
>>> una_cadena = 'un pequeño ejemplo'.replace('o', '-')
>>> una_cadena
'un pequeñ- ejempl-'
>>> |
```

```
>>> cad = 'Un gran ejemplo'
>>> cad.find('gran')
3
>>> cad.find('Gran')
-1
>>> cad.find('z')
-1
>>> cad.find('ejem')
8
>>> |
```

El método split()

```
>>> help(str.split)
```

```
Help on method descriptor:
```

```
split(...)
```

```
S.split(sep=None, maxsplit=-1) -> list of strings
```

Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

```
>>> txt = "Esto es una cadena de texto."
```

```
>>> txt.split()
```

```
['Esto', 'es', 'una', 'cadena', 'de', 'texto.']
```

```
>>> txt.split(' ')
```

```
['Esto', 'es', 'una', 'cadena', 'de', 'texto.']
```

```
>>> txt.split('a')
```

```
['Esto es un', ' c', 'den', ' de texto.']
```

```
>>> txt.split(' ', 3)
```

```
['Esto', 'es', 'una', 'cadena de texto.']
```

El método join()

```
>>> help(str.join)
```

```
Help on method descriptor:
```

```
join(...)
```

```
    S.join(iterable) -> str
```

```
    Return a string which is the concatenation of the strings in the
    iterable.  The separator between elements is S.
```

```
>>> paises = ['Argentina', 'Uruguay', 'Chile', 'Colombia']
```

```
>>> ', '.join(paises)
```

```
'Argentina, Uruguay, Chile, Colombia'
```

```
>>> '---'.join(paises)
```

```
'Argentina---Uruguay---Chile---Colombia'
```

```
>>> print('\n'.join(paises))
```

```
Argentina
```

```
Uruguay
```

```
Chile
```

```
Colombia
```

```
>>> '\n'.join(paises)
```

```
'Argentina\nUruguay\nChile\nColombia'
```

Cadenas y listas

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> a += ['corge']
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux']
>>> a += 'corge'
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'c', 'o', 'r', 'g', 'e']
```


Buscando una cadena dentro de otra cadena

```
In [4]: 'arbol' in 'No hay árbol que el viento no haya sacudido.'  
Out[4]: False
```

```
In [5]: 'árbol' in 'No hay árbol que el viento no haya sacudido.'  
Out[5]: True
```

```
In [6]: 'viento' in 'No hay árbol que el viento no haya sacudido.'  
Out[6]: True
```

```
In [7]: 'fruta' not in 'No hay árbol que el viento no haya sacudido.'  
Out[7]: True
```

```
In [8]: 'viento' not in 'No hay árbol que el viento no haya sacudido.'  
Out[8]: False
```

Algunas operaciones con cadenas

```
>>> s1 = 'Hola'
>>> len(s1) # la función len() retorna la longitud de la cadena
4
>>> s2 = 'Quiero comer un pastel de chocolate'
>>> s2.split()
['Quiero', 'comer', 'un', 'pastel', 'de', 'chocolate']
```

```
>>> x = 'Hola'
>>> x.startswith('h')
False
>>> x.startswith('H')
True
>>> x.endswith('a')
True
>>> x.endswith('A')
False
>>> x.lower().startswith('h')
True
>>> x.lower().endswith('a')
True
```

```
>>> 'Hola' + ' a ' + 'todos'
'Hola a todos'
>>> 'xyz' * 5
'xyzxyzxyzxyzxyz'
>>> 5 * 'xyz'
'xyzxyzxyzxyzxyz'
>>> int('123')
123
```

str.ljust(), str.rjust(), str.center()

```
>>> 'Hola'.rjust(10)
'      Hola'
>>> 'Hola'.rjust(10, '.')
'.....Hola'
>>> 'Hola'.ljust(10) + 'x'
'Hola      x'
>>> 'Hola'.ljust(10, '.') + 'x'
'Hola.....x'
>>> 'Hola'.center(10) + 'x'
'   Hola   x'
>>> 'Hola'.center(10, '.') + 'x'
'...Hola...x'
>>> L = 'cadena de texto'
>>> S = 'x' + L.center(20, '.') + 'x'
>>> S
'x..cadena de texto...x'
>>> 'x' + L.center(20, '.') + 'x'
'x..cadena de texto...x'
>>> 'x' + L.center(5, '.') + 'x'
'xcadena de textox'
```

str.zfill()

```
>>> help (str.zfill)
Help on method_descriptor:
```

```
zfill(...)
    S.zfill(width) -> str
```

Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

```
>>> '123'.zfill(10)
'0000000123'
>>> '-123'.zfill(10)
'-000000123'
>>> '123.32'.zfill(10)
'0000123.32'
>>> '-123.32'.zfill(10)
'-000123.32'
```

El método isdigit()

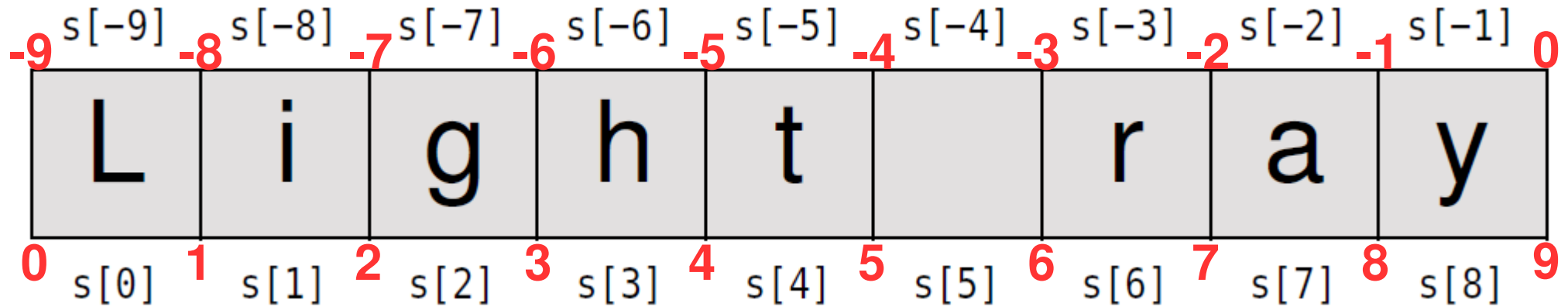
```
>>> '32'.isdigit()
True
>>> 'cuarenta'.isdigit()
False
>>> ''.isdigit()
False
>>> x = 10
>>> x.isdigit()
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    x.isdigit()
AttributeError: 'int' object has no attribute 'isdigit'
>>> x = '10'
>>> x.isdigit()
True
```

Otros métodos con cadenas

```
>>> 'xyz'.isalpha()
True
>>> 'x1y2z3'.isalpha()
False
>>> 'xyz '.isalpha()
False
>>> ''.isalpha()
False
>>> 'xyz '.isupper()
False
>>> 'XYZ '.isupper()
True
>>> 'xyz '.islower()
True
>>> 'XYZ '.islower()
False
>>> '123'.islower()
False
>>> ''.islower()
False
>>> ''.isupper()
False
```

- **S.isalpha()** retorna True si TODOS los caracteres de S están en el rango a..z, A..Z, de lo contrario retorna False
- **S.isupper()** retorna True si TODOS los caracteres de S son mayúsculas, de lo contrario retorna False
- **S.islower()** retorna True si TODOS los caracteres de S son minúsculas, de lo contrario retorna False

Indexado y slicing de cadenas



Indexado

```
>>> s = 'Light ray'
>>> s[0]
'L'
>>> s[8]
'y'
>>> s[-9]
'L'
>>> s[-1]
'y'
>>>
```

Índice

Slicing

```
>>> s = 'Light ray'
>>> s[0:2]
'Li'
>>> s[2:5]
'ght'
>>> s[:2]
'Li'
>>> s[4:]
't ray'
>>> s[-2:]
'ay'
>>> s[:2] + s[4:]
'Lit ray'
>>> |
```

Observe que el indexado de cadenas empieza en 0, tal y como en lenguaje C.

Indexado y slicing de cadenas

```
>>> s = 'Light ray'
```

```
>>> s[20]
```

```
Traceback (most recent call last):
```

```
File "<pyshell#31>", line 1, in <module>
```

```
s[20]
```

```
IndexError: string index out of range
```

```
>>> s[4:20]
```

```
't ray'
```

```
>>> s[20:]
```

```
''
```

```
>>> s[2] = 'J'
```

```
Traceback (most recent call last):
```

```
File "<pyshell#34>", line 1, in <module>
```

```
s[2] = 'J'
```

```
TypeError: 'str' object does not support item assignment
```

```
>>> s[2:] = 'xyz'
```

```
Traceback (most recent call last):
```

```
File "<pyshell#35>", line 1, in <module>
```

```
s[2:] = 'xyz'
```

```
TypeError: 'str' object does not support item assignment
```

```
>>> |
```

Indexando fuera del
rango produce error

Hacer slicing fuera del
rango es válido

Las cadenas no se
pueden modificar una vez
creadas: son **inmutables**

```
>>> 'Hola a todos'[2]
'l'
>>> 'Hola a todos'[7]
't'
```


Indexado y slicing de cadenas

- Slicing = corte.
- El `:` se conoce como el operador de corte (slicing operator)

```
>>> cad = 'abcdefghijklmnopqrstuvwxyz'
```

```
>>> cad[2:len(cad):3]
```

```
'cfilorux'
```

```
>>> cad[5:len(cad):2]
```

```
'fhjlnprtvxz'
```

```
>>> cad[::-2]
```

```
'zxvtrpnljhfdb'
```

```
>>> cad[3::-1]
```

```
'dcba'
```

```
>>> cad[20::-1]
```

```
'utsrqponmlkjihgfedcba'
```

```
>>> 'Hola a todos'[2:-4]
```

```
'la a t'
```

Reemplazando una porción de una cadena

```
>>> C = 'Hoy es un día muy especial'
>>> C[9:-13]
' día'
>>> C[9:-13] = 'a mañana' # genera un error ya que las cadenas son inmutables
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    C[9:-13] = 'a mañana' # genera un error ya que las cadenas son inmutables
TypeError: 'str' object does not support item assignment
>>>
>>> C = C[:9] + 'a mañana' + C[-13:]
>>> C
'Hoy es una mañana muy especial'
>>> |
```

Recorrido de cadenas

```
>>> cad = 'Una cadena'
>>> for i in range(len(cad)):
    print (i, cad[i])
```

```
0 U
1 n
2 a
3
4 c
5 a
6 d
7 e
8 n
9 a

>>> for i in range(len(cad)):
    print(cad[len(cad)-i-1])
```

a
n
e
d
a
c

a
n
U

```
>>> |
```

```
>>> cad = 'Una cadena'
>>> for letra in cad:
    print(letra)
```

U
n
a

c
a
d
e
n
a

```

1 cadena = input('Entre una cadena = ')
2 i = int(input('Entre un número = '))
3 j = int(input('Entre otro número = '))
4
5 subcadena = ''
6 ▼ for k in range(i,j):
7     subcadena += cadena[k]
8
9 print('La subcadena entre {0} y {1} es "{2}"'.format(i, j, subcadena))
10 print('La subcadena entre {0} y {1} es "{2}"'.format(i, j, cadena[i:j]))

```

Line: 12 of 14 Col: 1 LINE INS

02_subcadena.py

daalvarez@eredron:~ > cd /home/daalvarez

daalvarez@eredron:~ > python3 02_subcadena.py

Entre una cadena = Piensa como piensan los sabios, mas habla como habla la gente sencilla.

Entre un número = 3

Entre otro número = 20

La subcadena entre 3 y 20 es "nsa como piensan "

La subcadena entre 3 y 20 es "nsa como piensan "

daalvarez@eredron:~ > █

Referencias y punteros

Un **puntero** es la dirección de memoria de un objeto. Cada objeto en memoria se almacena en una dirección única. Una **referencia** se puede entender como un puntero que apunta a un objeto.

El comando id()

Retorna la “identificación” (la cédula de ciudadanía) de un objeto. Esto es, un entero que se garantiza que es único y constante para este objeto durante su vida útil.

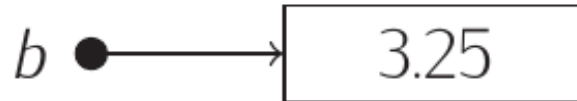
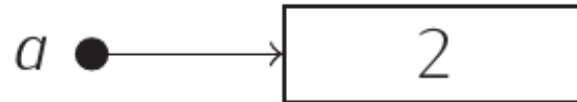
Dos objetos pueden tener el mismo id().

En CPython id() retorna la dirección de memoria del objeto.

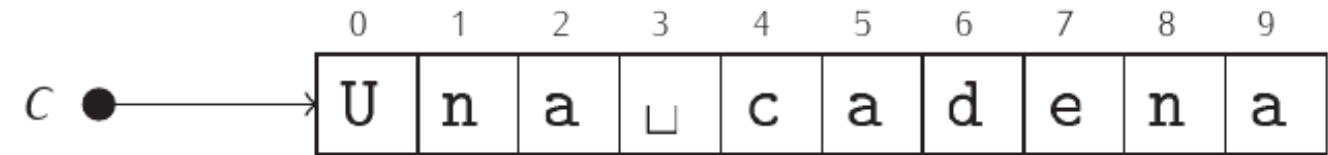
```
>>> x = 3
>>> id(x)
138405152
>>> hex(id(x)) # esta es la dirección de memoria de la variable 'x'
'0x83fe520'
```

Referencias a cadenas

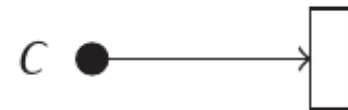
```
>>> a = 2
>>> b = 3.25
```



```
>>> c = 'Una cadena'
```



```
>>> c = ''
```

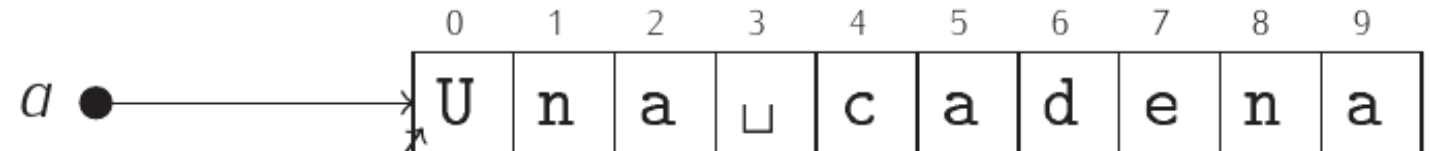


```
>>> a = 'Una cadena'
```

```
>>> b = a
```

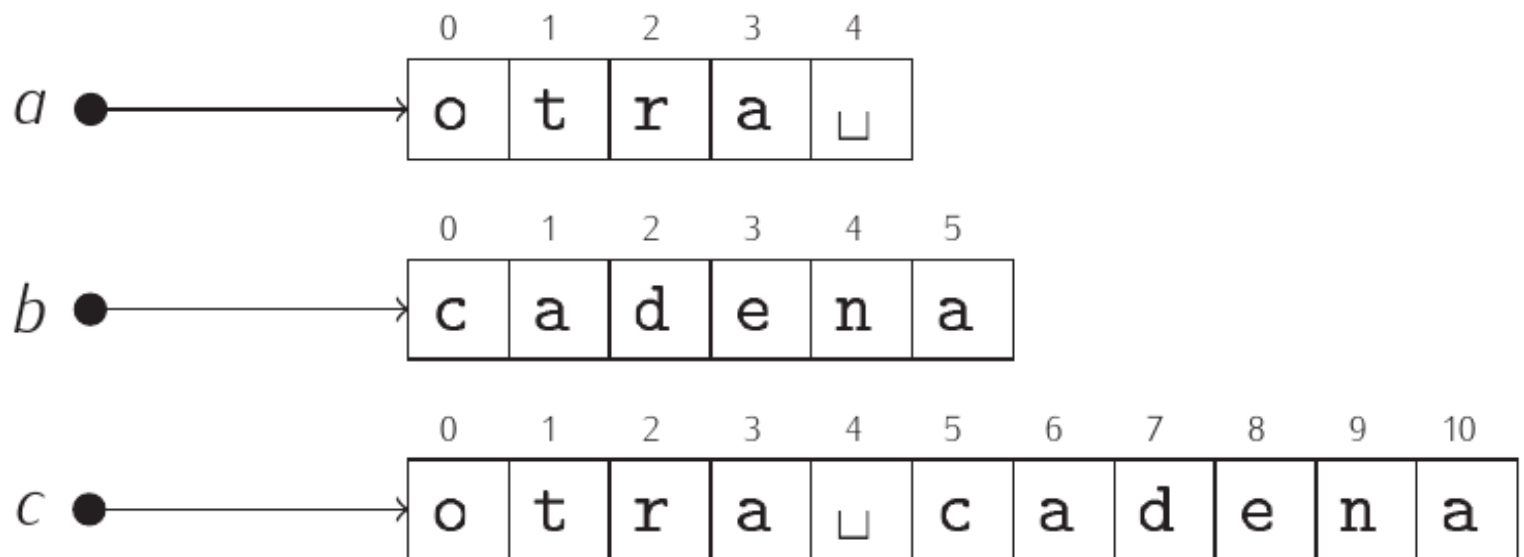
```
>>> hex(id(a))
'0x7fa87d51c030'
```

```
>>> hex(id(b))
'0x7fa87d51c030'
```



Tanto *a* como *b* apuntan a la misma cadena. Al asignar *b*=*a* únicamente se está copiando su referencia y no cada uno de los caracteres que la componen.

```
>>> a = 'otra '  
>>> b = 'cadena'  
>>> c = a+b  
>>>  
>>> hex(id(a))  
'0x7fa87d50da78'  
>>> hex(id(b))  
'0x7fa87d50dab0'  
>>> hex(id(c))  
'0x7fa881f66eb0'
```




```

>>> a = 'Una cadena muy muy larga'
>>> hex(id(a))
'0xb71e89f8'
>>> a = a+'.'
>>> hex(id(a))
'0xb71e8a30'

```



Variables “intern” (recluídas)

The reason this works interactively is that (most) string literals are **interned** by default. From Wikipedia:

Interned strings speed up string comparisons, which are sometimes a performance bottleneck in applications (such as compilers and dynamic programming language runtimes) that rely heavily on hash tables with string keys. Without interning, checking that two different strings are equal involves examining every character of both strings. This is slow for several reasons: it is inherently $O(n)$ in the length of the strings; it typically requires reads from several regions of memory, which take time; and the reads fills up the processor cache, meaning there is less cache available for other needs. With interned strings, a simple object identity test suffices after the original intern operation; this is typically implemented as a pointer equality test, normally just a single machine instruction with no memory reference at all.

So, when you have two string literals (words that are literally typed into your program source code, surrounded by quotation marks) in your program that have the same value, the Python compiler will automatically intern the strings, making them both stored at the same memory location. (Note that this doesn't *always* happen, and the rules for when this happens are quite convoluted, so please don't rely on this behavior in production code!)

Since in your interactive session both strings are actually stored in the same memory location, they have the same *identity*, so the `is` operator works as expected. But if you construct a string by some other method (even if that string contains *exactly* the same characters), then the string may be *equal*, but it is not *the same string* -- that is, it has a different *identity*, because it is stored in a different place in memory.

[share](#) [edit](#)

answered Oct 1 '09 at 16:02



[Daniel Pryden](#)

24k ● 3 ● 37 ● 84

http://en.wikipedia.org/wiki/String_interning

```
>>> a = 'xyz'
>>> b = 'xyz'
>>> id(a)
139712166587056
>>> id(b)
139712166587056
>>> a = 100* '-'
>>> b = 100* '-'
>>> id(a)
139712189879992
>>> id(b)
139712189880144
>>> a = 1
>>> b = 1
>>> id(a)
10455040
>>> id(b)
10455040
>>> a = 1000
>>> b = 1000
>>> id(a)
139712190830192
>>> id(b)
139712166332496
```

Variables “intern” (recluídas)

```
>>> a = 100*'-'  
>>> b = 100*'-'  
>>> id(a)  
139683138135736  
>>> id(b)  
139683138135888
```

Intern \In*tern"\, v. t. [F. interne. See {Intern}, a.]
1. To put for safe keeping in the interior of a place or country; to confine to one locality; as, to intern troops which have fled for refuge to a neutral country.
[1913 Webster]

```
>>> import sys  
>>> a = sys.intern(100*'-')  
>>> b = sys.intern(100*'-')  
>>> id(a)  
139683138136040  
>>> id(b)  
139683138136040  
>>> help(sys.intern)
```

```
>>> 'hell' + 'o' is 'hello'  
True
```

Help on built-in function intern in module sys:

```
intern(...)  
    intern(string) -> string
```

``Intern'' the given string. This enters the string in the (global) table of interned strings whose purpose is to speed up dictionary lookups. Return the string itself or the previously interned string object with the same value.

Variables “intern” (recluídas)

Python does this, so does Java, and so do C and C++ when compiling in optimized modes. If you use two identical strings, instead of wasting memory by creating two string objects, all interned strings with the same contents point to the same memory. This results in the Python "*is*" operator returning True because two strings with the same contents are pointing at the same string object. This is only useful for memory savings though. You cannot rely on it to test for string equality, because the various interpreters and compilers cannot always do it.

Ayuda de “cadenas”

- Todas las funciones y operaciones que se pueden realizar con cadenas se encuentran e

<https://docs.python.org/3/library/stdtypes.html#str>

- O escribiendo en la línea de comandos de Python:

```
>>> help(str)
```

Tamaño de las variables en memoria

Varían de implementación a implementación del interpretador, por lo que no se puede fiar de este número para los cálculos.

```
>>> import sys
>>> x = 123
>>> sys.getsizeof(x)
28
>>> sys.getsizeof(123)
28
>>> sys.getsizeof('abc')
52
>>> sys.getsizeof('abcd')
53
>>> x = 2**1000
>>> sys.getsizeof(x)
160
>>> sys.getsizeof([1, 2, 3, 'r'])
96
```

Referencias

- Wikipedia
- <http://www.inventwithpython.com/>
- <http://www.diveintopython3.net/>
- Documentación de Python:
 - <https://docs.python.org/3/tutorial/index.html>
 - <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>
- <https://realpython.com/python-data-types/>
- <https://realpython.com/python-strings/>