Modulo 2 (intermedio): Cadenas

Cómo las computadoras entienden los caracteres individuales

Has escrito algunos programas interesantes desde que comenzó este curso, pero todos ellos han procesado solo un tipo de datos: los numéricos. Como sabes (puedes ver esto en todas partes), muchos datos de la computadora no son números: nombres, apellidos, direcciones, títulos, poemas, documentos científicos, correos electrónicos, sentencias judiciales, confesiones de amor y mucho, mucho más.

Todos estos datos deben ser almacenados, ingresados, emitidos, buscados y transformados por computadoras como cualquier otro dato, sin importar si son caracteres únicos o enciclopedias de múltiples volúmenes.

¿Cómo es esto posible?

¿Cómo puedes hacerlo en Python? Esto es lo que discutiremos ahora. Comencemos con como las computadoras entienden los caracteres individuales.

Las computadoras almacenan los caracteres como números. Cada carácter utilizado por una computadora corresponde a un número único, y viceversa. Esta asignación debe incluir más caracteres de los que podrías esperar. Muchos de ellos son invisibles para los humanos, pero esenciales para las computadoras.

Algunos de estos caracteres se llaman **espacios en blanco**, mientras que otros se nombran **caracteres de control**, porque su propósito es controlar dispositivos de entrada y salida.

Un ejemplo de un espacio en blanco que es completamente invisible a simple vista es un código especial, o un par de códigos (diferentes sistemas operativos pueden tratar este asunto de manera diferente), que se utilizan para marcar el final de las líneas dentro de los archivos de texto.

Las personas no ven este signo (o estos signos), pero pueden observar el efecto de su aplicación donde ven un salto de línea.

Podemos crear prácticamente cualquier cantidad de asignaciones de números con caracteres, pero la vida en un mundo en el que cada tipo de computadora utiliza una codificación de caracteres diferentes no sería muy conveniente. Este sistema ha llevado a la necesidad de introducir un estándar universal y ampliamente aceptado, implementado por (casi) todas las computadoras y sistemas operativos en todo el mundo.

El denominado ASCII (por sus siglas en inglés American Standard Code for Information Interchange). El Código Estándar Americano para Intercambio de Información es el más utilizado, y es posible suponer que casi todos los dispositivos modernos (como computadoras, impresoras, teléfonos móviles, tabletas, etc.) usan este código.

El código proporciona espacio para 256 caracteres diferentes, pero solo nos interesan los primeros 128. Si deseas ver como se construye el código, mira la tabla a continuación. Haz clic en la tabla para ampliarla. Mírala cuidadosamente: hay algunos datos interesantes. Observa el código del carácter más común: el espacio. El cual es el 32.

Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111
(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113

Ahora verifica el código de la letra minúscula a. El cual es 97. Ahora encuentra la A mayúscula. Su código es 65. Ahora calcula la diferencia entre el código de la a y la A. Es igual a 32. Ese es el código del espacio. Interesante, ¿no es así?

También ten en cuenta que las letras están ordenadas en el mismo orden que en el Alfabeto Latino.

I18N

Ahora, el alfabeto latino no es suficiente para toda la humanidad. Los usuarios de ese alfabeto son minoría. Era necesario idear algo más flexible y capaz que ASCII, algo capaz de hacer que todo el software del mundo sea susceptible de **internacionalización**, porque diferentes idiomas usan alfabetos completamente diferentes, y a veces estos alfabetos no son tan simples como el latino.

La palabra internacionalización se acorta comúnmente a **I18N**.

¿Por qué? Observa con cuidado, hay una I al inicio de la palabra, a continuación hay 18 letras diferentes, y una N al final.

A pesar del origen ligeramente humorístico, el término se utiliza oficialmente en muchos documentos y normas.

El **software I18N** es un estándar en los tiempos actuales. Cada programa tiene que ser escrito de una manera que permita su uso en todo el mundo, entre diferentes culturas, idiomas y alfabetos.

El código ASCII emplea ocho bits para cada signo. Ocho bits significan 256 caracteres diferentes. Los

primeros 128 se usan para el alfabeto latino estándar (tanto en mayúsculas como en minúsculas). ¿Es posible colocar todos los otros caracteres utilizados en todo el mundo a los 128 lugares restantes?

No, no lo es.

Puntos de código y páginas de códigos

Necesitamos un nuevo término: un **punto de código**.

Un punto de código **es un numero que compone un caracter**. Por ejemplo, el 32 es un punto de código que compone un espacio en codificación ASCII. Podemos decir que el código ASCII estándar consta de 128 puntos de código.

Como el ASCII estándar ocupa 128 de 256 puntos de código posibles, solo puedes hacer uso de los 128 restantes.

No es suficiente para todos los idiomas posibles, pero puede ser suficiente para un idioma o para un pequeño grupo de idiomas similares.

¿Se puede **establecer la mitad superior de los puntos de código de manera diferente para diferentes idiomas**? Si, por supuesto. A tal concepto se le denomina una página de códigos.

Una página de códigos es un **estándar para usar los 128 puntos de código superiores para almacenar caracteres específicos**. Por ejemplo, hay diferentes páginas de códigos para Europa Occidental y Europa del Este, alfabetos cirílicos y griegos, idiomas árabe y hebreo, etc.

Esto significa que el mismo punto de código puede formar diferentes caracteres cuando se usa en diferentes páginas de códigos.

Por ejemplo, el punto de código 200 forma una Č (una letra usada por algunas lenguas eslavas) cuando lo utiliza la página de códigos ISO/IEC 8859-2, pero forma un Ш (una letra cirílica) cuando es usado por la página de códigos ISO/IEC 8859-5.

En consecuencia, para determinar el significado de un punto de código específico, debes conocer la página de códigos de destino.

En otras palabras, los puntos de código derivados del código de páginas son ambiguos.

Unicode

Las páginas de códigos ayudaron a la industria informática a resolver problemas de I18N durante algún tiempo, pero pronto resultó que no serían una solución permanente.

El concepto que resolvió el problema a largo plazo fue el Unicode.

Unicode asigna caracteres únicos (letras, guiones, ideogramas, etc.) a más de un millón de puntos de código. Los primeros 128 puntos de código Unicode son idénticos a ASCII, y los primeros 256 puntos de código Unicode son idénticos a la página de códigos ISO/IEC 8859-1 (una página de códigos diseñada para idiomas de Europa occidental).

UCS-4

El estándar Unicode no dice nada sobre como codificar y almacenar los caracteres en la memoria y los archivos.

Solo nombra todos los caracteres disponibles y los asigna a planos (un grupo de caracteres de origen, aplicación o naturaleza similares).

Existe más de un estándar que describe las técnicas utilizadas para implementar Unicode en computadoras y sistemas de almacenamiento informáticos reales. El más general de ellos es UCS-4.

El nombre proviene de Universal Character Set (Conjunto de Caracteres Universales).

UCS-4 emplea 32 bits (cuatro bytes) para almacenar cada carácter, y el código es solo el número único de los puntos de código Unicode. Un archivo que contiene texto codificado UCS-4 puede comenzar con un BOM (byte order mark - marca de orden de bytes), una combinación no imprimible de bits que anuncia la naturaleza del contenido del archivo. Algunas utilidades pueden requerirlo.

Como puedes ver, UCS-4 es un estándar bastante derrochador: aumenta el tamaño de un texto cuatro veces en comparación con el estándar ASCII. Afortunadamente, hay formas más inteligentes de codificar textos Unicode.

UTF-8

Uno de los más utilizados es **UTF-8**.

El nombre se deriva de **Unicode Transformation Format** (Formato de Transformación Unicode).

El concepto es muy inteligente. **UTF-8 emplea tantos bits para cada uno de los puntos de código como realmente necesita para representarlos**.

Por ejemplo:

- Todos los caracteres latinos (y todos los caracteres ASCII estándar) ocupan ocho bits.
- Los caracteres no latinos ocupan 16 bits.
- Los ideógrafos CJK (China-Japón-Corea) ocupan 24 bits.

Debido a las características del método utilizado por UTF-8 para almacenar los puntos de código, no es necesario usar el BOM, pero algunas de las herramientas lo buscan al leer el archivo, y muchos editores lo configuran durante el guardado.

Python 3 es totalmente compatible con Unicode y UTF-8:

- Puedes usar caracteres codificados Unicode / UTF-8 para nombrar variables y otras entidades.
- Puedes usarlos durante todas las entradas y salidas.

Esto significa que Python3 está completamente Internacionalizado.

Puntos Clave

1. Las computadoras almacenan caracteres como números. Hay más de una forma posible de codificar caracteres, pero solo algunas de ellas ganaron popularidad en todo el mundo y se usan comúnmente en TI: estas son **ASCII** (se emplea principalmente para codificar el alfabeto latino y algunos de sus derivados) y **UNICODE** (capaz de codificar prácticamente todos los alfabetos que utilizan los seres humanos).

2. Un número correspondiente a un carácter en particular se llama **punto de código**.

3. UNICODE utiliza diferentes formas de codificación cuando se trata de almacenar los caracteres usando archivos o memoria de computadora: dos de ellas son **UCS-4** y **UTF-8** (esta última es la más común ya que desperdicia menos espacio de memoria).

BOM (Byte Order Mark), Una Marca de Orden de Bytes es una combinación especial de bits que anuncia la codificación utilizada por el contenido de un archivo (por ejemplo, UCS-4 o UTF-B).

Cadenas: una breve reseña

Hagamos un breve repaso de la naturaleza de las cadenas en Python.

En primer lugar, las cadenas de Python (o simplemente cadenas, ya que no vamos a discutir las cadenas de ningún otro lenguaje) son secuencias inmutables.

Es muy importante tener en cuenta esto, porque significa que debes esperar un comportamiento familiar.

Analicemos el código en el editor para entender de lo qué estamos hablando:

Observa el Ejemplo 1. La función len() empleada en las cadenas devuelve el número de caracteres que contiene el argumento. La salida del código es 2. Cualquier cadena puede estar vacía. Si es el caso, su longitud es 0 como en el Ejemplo 2.

No olvides que la diagonal invertida (\) empleada como un carácter de escape, no esta incluida en la longitud total de la cadena. El código en el Ejemplo 3, da como salida un 3. Ejecuta los tres ejemplos de código y verifícalo.

```
# Ejemplo 1

word = 'by'
print(len(word))


# Ejemplo 2

empty = ''
print(len(empty))


# Ejemplo 3

i_am = 'I\'m'
print(len(i_am))
```

Cadenas multilínea

Ahora es un muy buen momento para mostrarte otra forma de especificar cadenas dentro del código fuente de Python. Ten en cuenta que la sintaxis que ya conoces no te permitirá usar una cadena que ocupe más de una línea de texto.

Por esta razón, el código siguiente es erróneo:

```
multiline = 'Línea #1
Línea #2'

print(len(multiline))
```

Afortunadamente, para este tipo de cadenas, Python ofrece una sintaxis simple, conveniente y separada.

```
multiline = '''Línea #1
Línea #2'''

print(len(multiline))
```

Como puedes ver, la cadena comienza con **tres apóstrofes**, no uno. El mismo apóstrofe triplicado se usa para terminar la cadena.

El número de líneas de texto dentro de una cadena de este tipo es arbitrario.

La salida del código es *17*.

Cuenta los caracteres con cuidado. ¿Es este resultado correcto o no? Se ve bien a primera vista, pero cuando cuentas los caracteres, no lo es.

Línea #1 contiene ocho caracteres. Las dos líneas juntas contienen 16 caracteres. ¿Perdimos un carácter? ¿Dónde? ¿Cómo?

No, no lo hicimos.

El carácter que falta es simplemente invisible: **es un espacio en blanco**. Se encuentra entre las dos líneas de texto.

Se denota como: **\n**.

¿Lo recuerdas? Es un carácter especial (de control) utilizado para **forzar un avance de línea**. No puedes verlo, pero cuenta.

Las cadenas multilínea pueden ser delimitadas también por **comillas triples**, como aqui:

multiline = «»«Línea #1 Línea #2»«»

print(len(multiline))

Elige el método que sea más cómodo. Ambos funcionan igual.

Operaciones con cadenas

Al igual que otros tipos de datos, las cadenas tienen su propio conjunto de operaciones permitidas, aunque son bastante limitadas en comparación con los números.

En general, las cadenas pueden ser:

- Concatenadas (unidas).
- Replicadas.

La primera operación la realiza el operador **+** (toma en cuenta que no es una adición o suma) mientras que la segunda por el operador * (toma en cuenta de nuevo que no es una multiplicación).

La capacidad de usar el mismo operador en tipos de datos completamente diferentes (como números o cadenas) se llama **overloading - sobrecarga** (debido a que el operador está sobrecargado con diferentes tareas).

Analiza el ejemplo:

- El operador + es empleado en dos o más cadenas y produce una nueva cadena que contiene todos los caracteres de sus argumentos (nota: el orden es relevante aquí, en contraste con su versión numérica, la cual es conmutativa).
- El operador * necesita una cadena y un número como argumentos; en este caso, el orden no importa: puedes poner el número antes de la cadena, o viceversa, el resultado será el mismo: una nueva cadena creada por la enésima replicación de la cadena del argumento.

```
str1 = 'a'
str2 = 'b'

print(str1 + str2)
print(str2 + str1)
print(5 * 'a')
print('b' * 4)
```

El fragmento de código produce el siguiente resultado:

```
ab
ba
aaaaa
bbbb
```

Nota: Los atajos de los operadores anteriores también son aplicables para las cadenas (+= y *  .

Operaciones con cadenas: ord()

Si deseas **saber el valor del punto de código ASCII/UNICODE de un carácter específico**, puedes usar la función **ord()** (proveniente de ordinal).

La función necesita **una cadena de un carácter como argumento** - incumplir este requisito provoca una excepción TypeError, y devuelve un número que representa el punto de código del argumento.

```
# # Demostración de la función ord ().

char_1 = 'a'
char_2 = ' '  # space

print(ord(char_1))
print(ord(char_2))
```

Las salida del fragmento de código es:

```
97
32
```

Ahora asigna diferentes valores a ch1 y ch2, por ejemplo, α (letra griega alfa), y ę (una letra en el alfabeto polaco); luego ejecuta el código y ve qué resultado produce. Realiza tus propios experimentos.

Operaciones con cadenas: chr()

Si conoces el punto de código (número) y deseas obtener el carácter correspondiente, puedes usar la función llamada **chr()**.

La función **toma un punto de código y devuelve su carácter**.

Invocándolo con un argumento inválido (por ejemplo, un punto de código negativo o inválido) provoca las excepciones ValueError o TypeError.

```
# Demostración de la función chr.

print(chr(97))
print(chr(945))
```

Ejecuta el código en el editor, su salida es la siguiente:

```
a
α
```

Nota:

- chr(ord(x)) == x
- ord(chr(x)) == x

Cadenas como secuencias: indexación

Ya dijimos antes que las **cadenas de Python son secuencias**. Es hora de mostrarte lo que significa realmente.

Las cadenas no son listas, pero **pueden ser tratadas como tal en muchos casos**.

Por ejemplo, si deseas acceder a cualquiera de los caracteres de una cadena, puedes hacerlo usando **indexación**. Ejecuta el programa:

```
# Indexando cadenas.

the_string = 'silly walks'

for ix in range(len(the_string)):
    print(the_string[ix], end=' ')

print()
```

Ten cuidado, no intentes pasar los límites de la cadena, ya que provocará una excepción.

La salida del ejemplo es:

```
s i l l y   w a l k s
```

Por cierto, los índices negativos también se comportan como se espera. Comprueba esto tú mismo.

Cadenas como secuencias: iterando

El iterar a través de las cadenas funciona también. Observa el siguiente ejemplo:

```
# Iterando a través de una cadena.

the_string = 'silly walks'

for character in the_string:
    print(character, end=' ')

print()
```

La salida es la misma que el ejemplo anterior. Revísalo.

Rebanadas

Todo lo que sabes sobre rebanadas es utilizable.

Hemos reunido algunos ejemplos que muestran cómo funcionan las rebanadas en el mundo de las cadenas. Observa el código en el editor, analízalo y ejecútalo.

No verás nada nuevo en el ejemplo, pero queremos que estés seguro de entender todas las líneas del código.

```
# Rebanadas

alpha = "abdefg"

print(alpha[1:3])
print(alpha[3:])
print(alpha[:3])
print(alpha[3:-2])
print(alpha[-3:4])
print(alpha[::2])
print(alpha[1::2])
```

La salida del código es:

```
bd
efg
abd
e
e
adf
beg
```

Los operadores in y not in

El operador in

El operador **in** no debería sorprenderte cuando se aplica a cadenas, simplemente **comprueba si el argumento izquierdo (una cadena) se puede encontrar en cualquier lugar dentro del argumento**

derecho (otra cadena).

El resultado es simplemente True(Verdadero) o False(Falso).

Observa el ejemplo a continuación. Así es como el operador in funciona:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" in alphabet)
print("F" in alphabet)
print("1" in alphabet)
print("ghi" in alphabet)
print("Xyz" in alphabet)
```

La salida del ejemplo es:

```
True
False
False
True
False
```

El operador not in

Como probablemente puedas deducir, el operador **not in** también es aplicable aquí.

Así es como funciona:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" not in alphabet)
print("F" not in alphabet)
print("1" not in alphabet)
print("ghi" not in alphabet)
print("Xyz" not in alphabet)
```

La salida del ejemplo es:

```
False
True
True
False
True
```

Las cadenas de Python son inmutables

También te hemos dicho que las **cadenas de Python son inmutables**. Esta es una característica muy importante. ¿Qué significa?

Esto significa principalmente que la similitud de cadenas y listas es limitada. No todo lo que puede hacerse con una lista puede hacerse con una cadena.

La primera diferencia importante **no te permite usar la instrucción *del* para eliminar cualquier cosa de**

una cadena.

El ejemplo siguiente no funcionará:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
del alphabet[0]
```

Lo único que puedes hacer con *del* y una cadena es **eliminar la cadena como un todo**. Intenta hacerlo.

Las cadenas de Python **no tienen el método *append()*** - no se pueden expander de ninguna manera.

El siguiente ejemplo es erróneo:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet.append("A")
```

Con la ausencia del método append(), **el método *insert()*** también es ilegal:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet.insert(0, "A")
```

No pienses que la inmutabilidad de una cadena limita tu capacidad de operar con ellas.

La única consecuencia es que debes recordarlo e implementar tu código de una manera ligeramente diferente:

```
alphabet = "bcdefghijklmnopqrstuvwxy"

alphabet = "a" + alphabet
alphabet = alphabet + "z"

print(alphabet)
```

Esta forma de código es totalmente aceptable, funcionará sin doblar las reglas de Python y traerá el alfabeto latino completo a tu pantalla:

```
abcdefghijklmnopqrstuvwxyz
```

Es posible que desees preguntar si el **crear una nueva copia de una cadena cada vez que se modifica su contenido empeora la efectividad del código**.

Sí, lo hace un poco. Sin embargo, no es un problema en lo absoluto.

Operaciones con cadenas: min()

Ahora que comprendes que las cadenas son secuencias, podemos mostrarte algunas capacidades de secuencia menos obvias. Las presentaremos utilizando cadenas, pero no olvides que las listas también pueden adoptar los mismos trucos.

Comenzaremos con la función llamada **min()**.

Esta función **encuentra el elemento mínimo de la secuencia pasada como argumento**. Existe una condición - la secuencia (cadena o lista) **no puede estar vacía**, de lo contrario obtendrás una excepción ValueError.

```
# Demonstrando min() - Ejemplo 1:
```

```
print(min("aAbByYzZ"))


# Demonstrando min() - Ejemplo 2 y 3:
t = 'Los Caballeros Que Dicen "¡Ni!"'
print('[' + min(t) + ']')

t = [0, 1, 2]
print(min(t))
```

El programa Ejemplo 1 da la siguiente salida:

```
A
```

Nota: Es una A mayúscula. ¿Por qué? Recuerda la tabla ASCII, ¿qué letras ocupan las primeras posiciones, mayúsculas o minúsculas?

Hemos preparado dos ejemplos más para analizar: Ejemplos 2 y 3.

Como puedes ver, presentan más que solo cadenas. El resultado esperado se ve de la siguiente manera:

```
[ ]
0
```

Nota: hemos utilizado corchetes para evitar que el espacio se pase por alto en tu pantalla.

Operaciones con cadenas: max()

Del mismo modo, una función llamada **max() encuentra el elemento máximo de la secuencia**.

```
# Demostración de max() - Ejemplo 1:
print(max("aAbByYzZ"))


# Demostración de max() - Ejemplo 2 & 3:
t = 'Los Caballeros Que Dicen "¡Ni!"'
print('[' + max(t) + ']')

t = [0, 1, 2]
print(max(t))
```

La salida del programa es:

```
z
```

Nota: es una z minúscula.

Ahora veamos la función **max()** a los mismos datos del ejemplo anterior. Observa los Ejemplos 2 y 3 en el editor.

La salida esperada es:

```
[ i ]
```

```
2
```

Operaciones con cadenas: el método index()

El método **index()** (es un método, no una función) **busca la secuencia desde el principio, para encontrar el primer elemento del valor especificado en su argumento**.

Nota: el elemento buscado debe aparecer en la secuencia - **su ausencia causará una excepción ValueError**.

El método devuelve el **índice de la primera aparición del argumento** (lo que significa que el resultado más bajo posible es 0, mientras que el más alto es la longitud del argumento decrementado en 1).

```
# Demonstrando el método index():
print("aAbByYzZaA".index("b"))
print("aAbByYzZaA".index("Z"))
print("aAbByYzZaA".index("A"))
```

Por lo tanto, el ejemplo en la salida del editor es:

```
2
7
1
```

Operaciones con cadenas: la función list()

La función **list() toma su argumento (una cadena) y crea una nueva lista que contiene todos los caracteres de la cadena, uno por elemento de la lista**.

Nota: no es estrictamente una función de cadenas - **list()** es capaz de crear una nueva lista de muchas otras entidades (por ejemplo, de tuplas y diccionarios).

```
print(list("abcabc"))
```

La salida es:

```
['a', 'b', 'c', 'a', 'b', 'c']
```

Operaciones con cadenas: el método count()

El método **count()** cuenta todas las apariciones del elemento dentro de la secuencia. La ausencia de tal elemento no causa ningún problema.

```
print("abcabc".count("b"))
print('abcabc'.count("d"))
```

Observa el segundo ejemplo en el editor. ¿Puedes adivinar su salida?

Es:

```
2
0
```

Las cadenas de Python tienen un número significativo de métodos destinados exclusivamente al procesamiento de caracteres. No esperes que trabajen con otras colecciones. La lista completa se presenta aquí: https://docs.python.org/3.4/library/stdtypes.html#string-methods.

Te mostraremos los que consideramos más útiles.

Puntos Claves

1. Las cadenas de Python son **secuencias inmutables** y se pueden indexar, dividir en rebanadas e iterar como cualquier otra secuencia, además de estar sujetas a los operadores in y not in. Existen dos tipos de cadenas en Python:

- Cadenas de una línea, las cuales no pueden cruzar los límites de una línea, las denotamos usando apóstrofes ('cadena') o comillas («cadena»).
- Cadenas multilínea, que ocupan más de una línea de código fuente, delimitadas por apóstrofes triples:

```
'''
cadena
'''

"""
cadena
"""
```

2. La longitud de una cadena está determinada por la función **len()**. El carácter de escape (\) no es contado. Por ejemplo:

```
print(len("\n\n"))
```

Su salida es 2.

3. Las cadenas pueden ser **concatenadas** usando el operador **+**, y **replicadas** usando el operador ***. Por ejemplo:

```
asterisk = '*'
plus = "+"
decoration = (asterisk + plus) * 4 + asterisk
print(decoration)
```

salida *+*+*+*+*.

4. El par de funciones **chr()** y **ord()** se pueden utilizar para crear un carácter utilizando su punto de código y para determinar un punto de código correspondiente a un carácter. Las dos expresiones siguientes son siempre verdaderas:

```
chr(ord(character)) == character
ord(chr(codepoint)) == codepoint
```

5. Algunas otras funciones que se pueden aplicar a cadenas son:

list(): crea una lista que consta de todos los caracteres de la cadena. **max()**: encuentra el carácter con el punto de código máximo. **min()**: encuentra el carácter con el punto de código mínimo.

6. El método llamado **index()** encuentra el índice de una subcadena dada dentro de la cadena.

El método capitalize()

Veamos algunos métodos estándar de cadenas en Python. Vamos a analizarlos en orden alfabético, cualquier orden tiene tanto desventajas como ventajas, por lo que la elección puede ser aleatoria.

El método **capitalize()** hace exactamente lo que dice - **crea una nueva cadena con los caracteres tomados de la cadena fuente**, pero intenta modificarlos de la siguiente manera:

- **Si el primer carácter dentro de la cadena es una letr**a (nota: el primer carácter es el elemento con un índice igual a 0, no es el primer carácter visible), se convertirá a mayúsculas.
- **Todas las letras restantes de la cadena se convertirán a minúsculas**.

No olvides que:

- La cadena original desde la cual se invoca el método no se cambia de ninguna manera, la inmutabilidad de una cadena debe obedecerse sin reservas.
- La cadena modificada (en mayúscula en este caso) se devuelve como resultado; si no se usa de alguna manera (asígnala a una variable o pásala a una función / método) desaparecerá sin dejar rastro.

Nota: los métodos no tienen que invocarse solo dentro de las variables. Se pueden invocar directamente desde dentro de literales de cadena. Usaremos esa convención regularmente: simplificará los ejemplos, ya que los aspectos más importantes no desaparecerán entre asignaciones innecesarias.

```
# Demostración del método capitalize():
print('aBcD'.capitalize())
```

Esto es lo que imprime:

```
Abcd
```

Prueba algunos ejemplos más avanzados y verifíca su salida:

```
print("Alpha".capitalize())
print('ALPHA'.capitalize())
print(' Alpha'.capitalize())
print('123'.capitalize())
print("αβγδ".capitalize())
```

El método center()

La variante de un parámetro del método **center() genera una copia de la cadena original, tratando de centrarla dentro de un campo de un ancho especificado**.

El centrado se realiza realmente al **agregar algunos espacios antes y después de la cadena**.

No esperes que este método demuestre habilidades sofisticadas. Es bastante simple.

El ejemplo en el editor usa corchetes para mostrar claramente donde comienza y termina realmente la cadena centrada.

```
# Demostración del método center():
```

```
print('[' + 'alpha'.center(10) + ']')
```

Su salida se ve de la siguiente manera:

```
[  alpha   ]
```

Si la longitud del campo de destino es demasiado pequeña para ajustarse a la cadena, se devuelve la cadena original.

Puedes ver el método **center()** en más ejemplos aquí:

```
print('[' + 'Beta'.center(2) + ']')
print('[' + 'Beta'.center(4) + ']')
print('[' + 'Beta'.center(6) + ']')
```

Ejecuta el código anterior y verifica que salidas produce.

La variante de dos parámetros de *center()* hace uso del carácter del segundo argumento, en lugar de un espacio. Analiza el siguiente ejemplo:

```
print('[' + 'gamma'.center(20, '*') + ']')
```

Es por eso que la salida ahora se ve así:

```
[*******gamma********]
```

El método endswith()

El método **endswith() comprueba si la cadena dada termina con el argumento especificado y devuelve True(verdadero) o False(falso)**, dependiendo del resultado.

Nota: la subcadena debe adherirse al último carácter de la cadena; no se puede ubicar en algún lugar cerca del final de la cadena.

```
# Demostración del método endswith():
if "epsilon".endswith("on"):
print("si")
else:
print("no")
```

Su salida es:

```
si
```

Ahora deberías poder predecir la salida del fragmento de código a continuación:

```
t = "zeta"
print(t.endswith("a"))
print(t.endswith("A"))
print(t.endswith("et"))
print(t.endswith("eta"))
```

El método find()

El método **find()** es similar al método index(), el cual ya conoces - busca una subcadena y devuelve el índice de la primera aparición de esta subcadena, pero:

- Es más seguro, no genera un error para un argumento que contiene una subcadena inexistente (devuelve -1 en dicho caso).
- Funciona solo con cadenas - no intentes aplicarlo a ninguna otra secuencia.

```
# Demostración del método find():
print("Eta".find("ta"))
print("Eta".find("mma"))
```

El ejemplo imprime:

```
1
-1
```

Nota: no se debe de emplear **find()** si deseas verificar si un solo carácter aparece dentro de una cadena - el operador **in** será significativamente más rápido.

Aquí hay otro ejemplo:

```
t = 'theta'
print(t.find('eta'))
print(t.find('et'))
print(t.find('the'))
print(t.find('ha'))
```

Si deseas realizar la búsqueda, no desde el principio de la cadena, sino **desde cualquier posición**, puedes usar una **variante de dos parámetros** del método find(). Mira el ejemplo:

```
print('kappa'.find('a', 2))
```

El segundo argumento **especifica el índice en el que se iniciará la búsqueda** (no tiene que estar dentro de la cadena).

De las dos letras a, solo se encontrará la segunda. Ejecuta el código y verifica.

Se puede emplear el método find() para buscar todas las ocurrencias de la subcadena, como aquí:

```
the_text = """A variation of the ordinary lorem ipsum
text has been used in typesetting since the 1960s
or earlier, when it was popularized by advertisements
for Letraset transfer sheets. It was introduced to
the Information Age in the mid-1980s by the Aldus Corporation,
which employed it in graphics and word-processing templates
for its desktop publishing program PageMaker (from Wikipedia)"""

fnd = the_text.find('the')
while fnd != -1:
    print(fnd)
    fnd = the_text.find('the', fnd + 1)
```

El código imprime los índices de todas las ocurrencias del artículo the, y su salida se ve así:

```
15
80
198
221
238
```

Existe también una **mutación de tres parámetros del método *find()*** - el tercer argumento **apunta al primer índice que no se tendrá en cuenta durante la búsqueda** (en realidad es el límite superior de la búsqueda).

Observa el ejemplo a continuación:

```
print('kappa'.find('a', 1, 4))
print('kappa'.find('a', 2, 4))
```

El segundo argumento especifica el índice en el que se iniciará la búsqueda (no tiene que estar dentro de la cadena).

Por lo tanto, las salidas de ejemplo son:

```
1
-1
```

a no se puede encontrar dentro de los límites de búsqueda dados en el segundo print().

El método isalnum()

El método sin parámetros llamado **isalnum() comprueba si la cadena contiene solo dígitos o caracteres alfabéticos (letras) y devuelve True(verdadero) o False(falso)** de acuerdo al resultado.

```
# Demostración del método the isalnum():
print('lambda30'.isalnum())
print('lambda'.isalnum())
print('30'.isalnum())
print('@'.isalnum())
print('lambda_30'.isalnum())
print(''.isalnum())
```

Nota: cualquier elemento de cadena que no sea un dígito o una letra hace que el método regrese False(falso). Una cadena vacía también lo hace.

El resultado de ejemplo es:

```
True
True
True
False
False
False
```

Existen tres ejemplos más aquí:

```
t = 'Six lambdas'
print(t.isalnum())

t = 'AβΓδ'
print(t.isalnum())

t = '20E1'
print(t.isalnum())
```

Ejecútalos y verifica su salida.

Nota: la causa del primer resultado es un espacio, no es ni un dígito ni una letra.

El método isalpha()

El método **isalpha()** es más especializado, se interesa en letras solamente.

```
# Ejemplo 1: Demostración del método isapha():
print("Moooo".isalpha())
print('Mu40'.isalpha())
```

Su salida es:

```
True
False
salida
```

El método isdigit()

Al contrario, el método isdigit() busca solo dígitos - cualquier otra cosa produce False(falso) como resultado.

```
# Ejemplo 2: Demostración del método isdigit():
print('2018'.isdigit())
print("Year2019".isdigit())
```

Su salida es:

```
True
False
```

El método islower()

El método **islower()** es una variante de isalpha() - solo acepta letras minúsculas.

```
# Ejemplo 1: Demostración del método islower():
print("Moooo".islower())
print('moooo'.islower())
```

```
False
True
```

El método isspace()

El método **isspace()** identifica espacios en blanco solamente - no tiene en cuenta ningún otro carácter (el resultado es entonces False).

```
# Ejemplo 2: Demostración del método isspace(:
print(' \n '.isspace())
print(" ".isspace())
print("mooo mooo mooo".isspace())
```

```
True
True
False
salida
```

El método isupper()

El método **isupper()** es la versión en mayúscula de islower() - se concentra solo en letras mayúsculas.

```
# Ejemplo 3: Demostración del método isupper():
print("Moooo".isupper())
print('moooo'.isupper())
print('MOOOO'.isupper())
```

```
False
False
True
```

El método join()

El método **join()** es algo complicado, así que déjanos guiarte paso a paso:

- Como su nombre lo indica, el método **realiza una unión** y espera un argumento del tipo lista; se debe asegurar que todos los elementos de la lista sean cadenas: de lo contrario, el método generará una excepción TypeError.
- Todos los elementos de la lista serán **unidos en una sola cadena** pero...
- ... la cadena desde la que se ha invocado el método será **utilizada como separador**, puesta entre las cadenas.
- La cadena recién creada se devuelve como resultado.

```
# Demonstrating the join() method:
print(",".join(["omicron", "pi", "rho"]))
```

Vamos a analizarlo:

- El método join() se invoca desde una cadena que contiene una coma (la cadena puede ser larga o puede estar vacía).
- El argumento del join es una lista que contiene tres cadenas.
- El método devuelve una nueva cadena.

Aquí está:

```
omicron,pi,rho
```

El método lower()

El método **lower() genera una copia de una cadena, reemplaza todas las letras mayúsculas con sus equivalentes en minúsculas**, y devuelve la cadena como resultado. Nuevamente, la cadena original permanece intacta.

Si la cadena no contiene caracteres en mayúscula, el método devuelve la cadena original.

Nota: El método lower() no toma ningún parámetro.

```
# Demostración del método lower():
print("SiGmA=60".lower())
```

```
sigma=60
```

El método lstrip()

El método sin parámetros **lstrip()** devuelve una cadena recién creada formada a partir de la original eliminando todos los espacios en blanco iniciales.

```
# Demostración del método the lstrip():
print("[" + " tau ".lstrip() + "]")
```

Los corchetes no son parte del resultado, solo muestran los límites del resultado.

Las salida del ejemplo es:

```
[tau ]
```

El método con **un parámetro** lstrip() hace lo mismo que su versión sin parámetros, pero **elimina todos los caracteres incluidos en el argumento** (una cadena), no solo espacios en blanco:

```
print("www.cisco.com".lstrip("w."))
```

Aquí no se necesitan corchetes, ya que el resultado es el siguiente:

```
cisco.com
```

¿Puedes adivinar la salida del fragmento a continuación? Piensa cuidadosamente. Ejecuta el código y verifica tus predicciones.

```
print("pythoninstitute.org".lstrip(".org"))
```

El método replace()

El método **replace()** con dos parámetros devuelve una copia de la cadena original en la que todas las apariciones del primer argumento han sido reemplazadas por el segundo argumento.

```
# Demostración del método replace():
print("www.netacad.com".replace("netacad.com", "pythoninstitute.org"))
print("This is it!".replace("is", "are"))
print("Apple juice".replace("juice", ""))
```

La salida del ejemplo es:

```
www.pythoninstitute.org
Thare are it!
Apple
```

Si el segundo argumento es una cadena vacía, **reemplazar significa realmente eliminar** el primer argumento de la cadena. ¿Qué tipo de magia ocurre si el primer argumento es una cadena vacía?

La variante del método replace() **con tres parámetros** emplea un tercer argumento (un número) para **limitar el número de reemplazos**.

Observa el código modificado a continuación:

```
print("This is it!".replace("is", "are", 1))
print("This is it!".replace("is", "are", 2))
```

El método rfind()

Los métodos de uno, dos y tres parámetros denominados **rfind()** hacen casi lo mismo que sus contrapartes (las que carecen del prefijo r), pero comienzan sus búsquedas desde el final de la cadena, no el principio (de ahí el prefijo r, de reversa).

```
# Demostración del método rfind():
print("tau tau tau".rfind("ta"))
print("tau tau tau".rfind("ta", 9))
print("tau tau tau".rfind("ta", 3, 9))
```

El método rstrip()

Dos variantes del método **rstrip()** hacen casi lo mismo que el método lstrip, pero a**fecta el lado opuesto de la cadena**.

```
# Demostración del método rstrip():
print("[" + " upsilon ".rstrip() + "]")
print("cisco.com".rstrip(".com"))
```

El método split()

El método **split() divide la cadena y crea una lista de todas las subcadenas detectadas**.

El método **asume que las subcadenas están delimitadas por espacios en blanco** - los espacios no participan en la operación y no se copian en la lista resultante.

Si la cadena está vacía, la lista resultante también está vacía.

```
# Demostración del método split():
print("phi       chi\npsi".split())
```

```
['phi', 'chi', 'psi']
```

Nota: la operación inversa se puede realizar por el método join().

El método startswith()

El método **startswith()** es un espejo del método endswith() - comprueba si una cadena dada comienza con la subcadena especificada.

```
# Demostración del método startswith():
print("omega".startswith("meg"))
print("omega".startswith("om"))
```

```
False
True
```

El método strip()

El método **strip()** combina los efectos causados por rstrip() y lstrip() - crea una nueva cadena que carece de todos los espacios en blanco iniciales y finales.

```
# Demostración del método strip():
print("[" + "   aleph   ".strip() + "]")
```

```
[aleph]
```

El método swapcase()

El método **swapcase()** crea una nueva cadena intercambiando todas las letras por mayúsculas o minúsculas dentro de la cadena original: los caracteres en mayúscula se convierten en minúsculas y viceversa.

Todos los demás caracteres permanecen intactos.

Observa el primer ejemplo en el editor. ¿Puedes adivinar la salida? No se verá nada bien, pero debes observarla:

```
# Demostración del método swapcase():
print("Yo sé que no sé nada.".swapcase())
```

```
yO SÉ QUE NO SÉ NADA.
```

El método title()

El método **title()** realiza una función algo similar cambia la primera letra de cada palabra a mayúsculas, convirtiendo todas las demás a minúsculas.

```
# Demostración del método title():
print("Yo sé que no sé nada. Part 1.".title())
```

```
Yo Sé Que No Sé Nada. Parte 1.
```

El método upper()

Por último, pero no menos importante, el método **upper()** hace una copia de la cadena de origen, reemplaza todas las letras minúsculas con sus equivalentes en mayúsculas, y devuelve la cadena como resultado.

```
# Demostración del método upper():
print("Yo sé que no sé nada. Part 2.".upper())
```

```
YO SÉ QUE NO SÉ NADA. PARTE 2.
```

Puntos Clave

1. Algunos de los métodos que ofrecen las cadenas son:

- **capitalize()**: cambia todas las letras de la cadena a mayúsculas.
- **center()**: centra la cadena dentro de una longitud conocida.
- **count()**: cuenta las ocurrencias de un carácter dado.
- **join()**: une todos los elementos de una tupla/lista en una cadena.
- **lower()**: convierte todas las letras de la cadena en minúsculas.
- **lstrip()**: elimina los caracteres en blanco al principio de la cadena.
- **replace()**: reemplaza una subcadena dada con otra.
- **rfind()**: encuentra una subcadena comenzando por el final de la cadena.
- **rstrip()**: elimina los caracteres en blanco al final de la cadena.
- **split()**: divide la cadena en una subcadena usando un delimitador dado.
- **strip()**: elimina los espacios en blanco iniciales y finales.
- **swapcase()**: intercambia las mayúsculas y minúsculas de las letras.
- **title()**: hace que la primera letra de cada palabra sea mayúscula.
- **upper()**: convierte todas las letras de la cadena en letras mayúsculas.

2. El contenido de las cadenas se puede determinar mediante los siguientes métodos (todos devuelven valores booleanos):

- **endswith()**: ¿La cadena termina con una subcadena determinada?
- **isalnum()**: ¿La cadena consta solo de letras y dígitos?
- **isalpha()**: ¿La cadena consta solo de letras?
- **islower()**: ¿La cadena consta solo de letras minúsculas?
- **isspace()**: ¿La cadena consta solo de espacios en blanco?
- **isupper()**: ¿La cadena consta solo de letras mayúsculas?
- **startswith()**: ¿La cadena consta solo de letras mayúsculas?

Comparando cadenas

Las cadenas en Python pueden ser comparadas usando el mismo conjunto de operadores que se emplean con los números.

Observa estos operadores: también sirven para comparar cadenas:

- ==
- !=
- >
- >=
- <
- ⇐

Existe un «pero»: los resultados de tales comparaciones a veces pueden ser un poco sorprendentes. No olvides que Python no es consciente (no puede serlo de ninguna manera) de problemas lingüísticos sutiles, simplemente **compara valores de puntos de código**, carácter por carácter.

Los resultados que se obtienen de una operación de este tipo a veces son sorprendentes. Comencemos con los casos más simples.

Dos cadenas son iguales cuando consisten de los mismos caracteres en el mismo orden. Del mismo modo, dos cadenas no son iguales cuando no consisten de los mismos caracteres en el mismo orden.

Ambas comparaciones dan True (verdadero) como resultado:

```
'alpha' == 'alpha'
'alpha' != 'Alpha'
```

La relación entre cadenas se determina al **comparar el primer carácter diferente en ambas cadenas** (ten en cuenta los puntos de código ASCII / UNICODE en todo momento).

Cuando se comparan dos cadenas de diferentes longitudes y la más corta es idéntica a la más larga, la **cadena más larga se considera mayor**.

Justo como aquí:

```
'alpha' < 'alphabet'
```

La comparación es True (verdadera).

La comparación de cadenas siempre distingue entre mayúsculas y minúsculas (**las letras mayúsculas se consideran menores en comparación con las minúsculas**).

La expresión es True (verdadera):

```
'beta' > 'Beta'
```

Aún **si una cadena contiene solo dígitos, todavía no es un número**. Se interpreta como lo que es, como cualquier otra cadena regular, y su aspecto numérico (potencial) no se toma en cuenta, en ninguna manera.

Observa los ejemplos:

```
'10' == '010'
'10' > '010'
'10' > '8'
```

```
'20' < '8'
'20' < '80'
```

Producen los siguientes resultados:

```
False
True
False
True
True
```

El comparar cadenas con los números generalmente es una mala idea.

Las únicas comparaciones que puede realizar con impunidad son aquellas simbolizadas por los operadores

** y **!=**. El primero siempre devuelve False (falso), mientras que el segundo siempre devuelve True (verdadero).

El uso de cualquiera de los operadores de comparación restantes generará una excepción TypeError.

Vamos a verlo:

```
'10' == 10
'10' != 10
'10' == 1
'10' != 1
'10' > 10
```

Los resultados en este caso son:

```
False
True
False
True
TypeError exception
```

Ordenamiento

La comparación está estrechamente relacionada con el ordenamiento (o más bien, el ordenamiento es, de hecho, un caso muy sofisticado de comparación).

Esta es una buena oportunidad para mostrar dos formas posibles de **ordenar listas que contienen cadenas**. Dicha operación es muy común en el mundo real: cada vez que ves una lista de nombres, productos, títulos o ciudades, esperas que este ordenada.

Supongamos que deseas ordenar la siguiente lista:

```
 greek = ['omega', 'alpha', 'pi', 'gamma']
```

En general, Python ofrece dos formas diferentes de ordenar las listas.

El primero se implementa con una función llamada **sorted()**.

La función toma un argumento (una lista) y **retorna una nueva lista**, con los elementos ordenados del argumento. (Nota: esta descripción está un poco simplificada en comparación con la implementación real; lo discutiremos más adelante).

La lista original permanece intacta.

```
# Demostración de la función sorted():
first_greek = ['omega', 'alpha', 'pi', 'gamma']
first_greek_2 = sorted(first_greek)

print(first_greek)
print(first_greek_2)

print()
```

El código produce el siguiente resultado:

```
['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']
```

El segundo método afecta a la lista misma - **no se crea una nueva lista**. El ordenamiento se realiza por el método denominado **sort()**.

```
# Demostración del método sort():
second_greek = ['omega', 'alpha', 'pi', 'gamma']
print(second_greek)

second_greek.sort()
print(second_greek)
```

La salida no ha cambiado:

```
['omega', 'alpha', 'pi', 'gamma']
['alpha', 'gamma', 'omega', 'pi']
```

Si necesitas un ordenamiento diferente, debes convencer a la función o método de cambiar su comportamiento predeterminado. Lo discutiremos pronto.

Cadenas frente a números

Hay dos cuestiones adicionales que deberían discutirse aquí: **¿Cómo convertir un número (un entero o un flotante) en una cadena, y viceversa?** Puede ser necesario realizar tal transformación. Además, es una forma rutinaria de procesar datos de entrada o salida.

La conversión de cadena a número es simple, ya que siempre es posible. Se realiza mediante una función llamada **str()**.

Justo como aquí:

```
itg = 13
flt = 1.3
si = str(itg)
```

```
sf = str(flt)

print(si + ' ' + sf)
```

La salida del código es:

```
13 1.3
```

La transformación inversa solo es posible cuando la cadena representa un número válido. Si no se cumple la condición, espera una excepción ValueError.

Emplea la función **int()** si deseas obtener un entero, y **float()** si necesitas un valor punto flotante.

Justo como aquí:

```
si = '13'
sf = '1.3'
itg = int(si)
flt = float(sf)

print(itg + flt)
```

Esto es lo que verás en la consola:

```
14.3
```

Puntos Claves

1. Las cadenas se pueden comparar con otras cadenas utilizando operadores de comparación generales, pero compararlas con números no da un resultado razonable, porque **ninguna cadena puede ser igual a ningún otro número**. Por ejemplo:

- cadena == número es siempre False (falso).
- cadena != número es siempre True (verdadero).
- cadena >= número siempre **genera una excepción**.

2. El ordenamiento de listas de cadenas se puede realizar mediante:

Una función llamada **sorted()**, crea una nueva, lista ordenada. Un método llamado **sort()**, el cual ordena la lista en el momento.

3. Un número se puede convertir en una cadena empleando la función **str()**.

4. Una cadena se puede convertir en un número (aunque no todas las cadenas) empleando ya sea la función **int()** o **float()**. La conversión falla si la cadena no contiene un número válido (se genera una excepción en dicho caso).

ejemplo: El Cifrado César: encriptando un mensaje

Te mostraremos cuatro programas simples para presentar algunos aspectos del procesamiento de cadenas en Python. Son intencionalmente simples, pero los problemas de laboratorio serán significativamente más complicados.

El primer problema que queremos mostrarte se llama Cifrado César - más detalles aquí: https://en.wikipedia.org/wiki/Caesar_cipher.

Este cifrado fue (probablemente) inventado y utilizado por Cayo Julio César y sus tropas durante las Guerras Galas. La idea es bastante simple: cada letra del mensaje se reemplaza por su consecuente más cercano (A se convierte en B, B se convierte en C, y así sucesivamente). La única excepción es la Z, la cual se convierte en A.

El programa en el editor es una implementación muy simple (pero funcional) del algoritmo.

```
# Cifrado César.
text = input("Ingresa tu mensaje: ")
cipher = ""
for char in text:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) + 1
    if code > ord('Z'):
        code = ord('A')
    cipher += chr(code)

print(cipher)
```

Se ha escrito utilizando los siguientes supuestos:

- Solo acepta letras latinas (nota: los Romanos no usaban espacios en blanco ni dígitos).
- Todas las letras del mensaje están en mayúsculas (nota: los Romanos solo conocían las mayúsculas).

Veamos el código:

- La línea 02: pide al usuario que ingrese un mensaje (sin cifrar) de una línea.
- La línea 03: prepara una cadena para el mensaje cifrado (esta vacía por ahora).
- La línea 04: inicia la iteración a través del mensaje.
- La línea 05: si el carácter actual no es alfabético...
- La línea 06: ...ignoralo.
- La línea 07: convierta la letra a mayúsculas (es preferible hacerlo a ciegas, en lugar de verificar si es necesario o no).
- La línea 08: obtén el código de la letra e increméntalo en uno.
- La línea 09: si el código resultante ha «dejado» el alfabeto latino (si es mayor que el código de la Z)...
- La línea 10: ... cámbialo al código de la A.
- La línea 11: agrega el carácter recibido al final del mensaje cifrado.
- La línea 13: imprime el cifrado.

El código, con este mensaje:

```
AVE CAESAR
```

Da como salida:

```
BWFDBFTBS
```

ejemplo: El Cifrado César: descifrando un mensaje

La operación inversa ahora debería ser clara para ti: solo presentamos el código tal como está, sin ninguna explicación.

Observa el código en el editor. Comprueba cuidadosamente si funciona. Usa el criptograma del programa anterior.

```
# Cifrado César - descifrar un mensaje.
cipher = input('Ingresa tu criptograma: ')
text = ""
for char in cipher:
    if not char.isalpha():
        continue
    char = char.upper()
    code = ord(char) - 1
    if code < ord('A'):
        code = ord('Z')
    text += chr(code)

print(text)
```

ejemplo: El Procesador de Números

El tercer programa muestra un método simple que permite ingresar una línea llena de números y sumarlos fácilmente. Nota: la función input(), combinada junto con las funciones int() o float(), no es lo adecuado para este propósito.

El procesamiento será extremadamente fácil: queremos que se sumen los números.

Observa el código en el editor. Analicémoslo.

Emplear listas puede hacer que el código sea más pequeño. Puedes hacerlo si quieres.

Presentemos nuestra versión:

```
#Procesador de Números.

line = input("Ingresa una línea de números, sepáralos con espacios: ")
strings = line.split()
total = 0
try:
    for substr in strings:
        total += float(substr)
    print("El total es:", total)
except:
    print(substr, "no es un numero.")
```

- La línea 03: pide al usuario que ingrese una línea llena de cualquier cantidad de números (los números pueden ser flotantes).
- La línea 04: divide la línea en una lista con subcadenas.
- La línea 05: se inicializa la suma total a cero.
- La línea 06: como la conversión de cadena a flotante puede generar una excepción, es mejor continuar con la protección del bloque try-except.
- La línea 07: itera a través de la lista…
- La línea 08: … e intenta convertir todos sus elementos en números flotantes; si funciona, aumenta la suma.
- La línea 09: todo está bien hasta ahora, así que imprime la suma.
- La línea 10: el programa termina aquí en caso de error.
- La línea 11: imprime un mensaje de diagnóstico que muestra al usuario el motivo de la falla.

El código tiene una debilidad importante: muestra un resultado falso cuando el usuario ingresa una línea vacía. ¿Puedes arreglarlo?

ejemplo: El Validador IBAN

El cuarto programa implementa (en una forma ligeramente simplificada) un algoritmo utilizado por los bancos Europeos para especificar los números de cuenta. El estándar llamado IBAN (Número de cuenta bancaria internacional) proporciona un método simple y bastante confiable para validar los números de cuenta contra errores tipográficos simples que pueden ocurrir durante la reescritura del número, por ejemplo, de documentos en papel, como facturas o facturas en las computadoras.

Puedes encontrar más detalles aquí: https://en.wikipedia.org/wiki/International_Bank_Account_Number.

Un número de cuenta compatible con IBAN consta de:

- Un código de país de dos letras tomado del estándar ISO 3166-1 (por ejemplo, FR para Francia, GB para Gran Bretaña DE para Alemania, y así sucesivamente).
- Dos dígitos de verificación utilizados para realizar las verificaciones de validez: pruebas rápidas y simples, pero no totalmente confiables, que muestran si un número es inválido (distorsionado por un error tipográfico) o válido.
- El número de cuenta real (hasta 30 caracteres alfanuméricos; la longitud de esa parte depende del país).

El estándar dice que la validación requiere los siguientes pasos (según Wikipedia):

- (Paso 1) Verificar que la longitud total del IBAN sea correcta según el país (este programa no lo hará, pero puedes modificar el código para cumplir con este requisito si lo deseas; nota: pero debes enseñar al código a conocer todas las longitudes utilizadas en Europa).
- (Paso 2) Mueve los cuatro caracteres iniciales al final de la cadena (es decir, el código del país y los dígitos de verificación).
- (Paso 3) Reemplaza cada letra en la cadena con dos dígitos, expandiendo así la cadena, donde A = 10, B = 11 … Z = 35.
- (Paso 4) Interpreta la cadena como un entero decimal y calcula el residuo de ese número dividiéndolo entre 97. Si el residuo es 1, pasa la prueba de verificación de dígitos y el IBAN puede ser válido.

```
# Validador IBAN.

iban = input("Ingresa un IBAN, por favor: ")
iban = iban.replace(' ','')

if not iban.isalnum():
```

```
    print("Has introducido caracteres no válidos.")
elif len(iban) < 15:
    print("El IBAN ingresado es demasiado corto.")
elif len(iban) > 31:
    print("El IBAN ingresado es demasiado largo.")
else:
    iban = (iban[4:] + iban[0:4]).upper()
    iban2 = ''
    for ch in iban:
        if ch.isdigit():
            iban2 += ch
        else:
            iban2 += str(10 + ord(ch) - ord('A'))
    iban = int(iban2)
    if iban % 97 == 1:
        print("El IBAN ingresado es válido.")
    else:
        print("El IBAN ingresado no es válido.")
```

- Línea 03: pide al usuario que ingrese el IBAN (el número puede contener espacios, ya que mejoran significativamente la legibilidad del número...
- Línea 04: ... pero remueve los espacios de inmediato).
- Línea 05: el IBAN ingresado debe constar solo de dígitos y letras, de lo contrario...
- Línea 06: ... muestra un mensaje.
- Línea 07: el IBAN no debe tener menos de 15 caracteres (esta es la variante más corta, utilizada en Noruega).
- Línea 08: si es más corto, se informa al usuario.
- Línea 09: además, el IBAN no puede tener más de 31 caracteres (esta es la variante más larga, utilizada en Malta).
- Línea 10: si es más largo, se le informa al usuario.
- Línea 11: se comienza con el procesamiento.
- Línea 12: se mueven los cuatro caracteres iniciales al final del número y se convierten todas las letras a mayúsculas (paso 02 del algoritmo).
- Línea 13: esta es la variable utilizada para completar el número, creada al reemplazar las letras con dígitos (de acuerdo con el paso 03 del algoritmo).
- Línea 14: iterar a través del IBAN.
- Línea 15: si el carácter es un dígito...
- Línea 16: ... se copia.
- Línea 17: de lo contrario...
- Línea 18: ... conviértelo en dos dígitos (observa cómo se hace aquí).
- Línea 19: la forma convertida del IBAN está lista: ahora se convierte en un número entero.
- Línea 20: ¿el residuo de la división de iban2 entre 97 es igual a 1?
- Línea 21: si es así, entonces el número es correcto.
- Línea 22: de lo contrario...
- Línea 23: ... el número no es válido.

Agreguemos algunos datos de prueba (todos estos números son válidos; puedes invalidarlos cambiando cualquier carácter).

```
Inglés: GB72 HBZU 7006 7212 1253 00
Francés: FR76 30003 03620 00020216907 50
Alemán: DE02100100100152517108
```

Puntos Claves

1. Las cadenas son herramientas clave en el procesamiento de datos modernos, ya que la mayoría de los datos útiles son en realidad cadenas. Por ejemplo, el uso de un motor de búsqueda web (que parece bastante trivial en estos días) utiliza un procesamiento de cadenas extremadamente complejo, que involucra cantidades inimaginables de datos.

2. El comparar cadenas de forma estricta (como lo hace Python) puede ser muy insatisfactorio cuando se trata de búsquedas avanzadas (por ejemplo, durante consultas extensas a bases de datos). En respuesta a esta demanda, se han creado e implementado una serie de algoritmos de comparación de cadenas difusos. Estos algoritmos pueden encontrar cadenas que no son iguales en el sentido de Python, pero que son **similares**.

Uno de esos conceptos es la **Distancia Hamming**, que se utiliza para determinar la similitud de dos cadenas. Si este tema te interesa, puedes encontrar más información al respecto aquí: https://en.wikipedia.org/wiki/Hamming_distance. Otra solución del mismo tipo, pero basada en un supuesto diferente, es la **Distancia Levenshtein** descrita aquí: https://en.wikipedia.org/wiki/Levenshtein_distance.

3. Otra forma de comparar cadenas es encontrar su similitud acústica, lo que significa un proceso que lleva a determinar si dos cadenas suenan similares (como «echo» y «hecho»). Esa similitud debe establecerse para cada idioma (o incluso dialecto) por separado.

Un algoritmo utilizado para realizar una comparación de este tipo para el idioma Inglés se llama **Soundex** y se inventó, no lo creerás, en 1918. Puedes encontrar más información al respecto aquí: https://en.wikipedia.org/wiki/Soundex.

4. Debido a la precisión limitada de los datos enteros y flotantes nativos, a veces es razonable almacenar y procesar valores numéricos enormes como cadenas. Esta es la técnica que usa Python cuando se le fuerza a operar con un número entero que consta de una gran cantidad de dígitos.