6 Cadenas

Una cadena es una secuencia

Una cadena es una **secuencia** de caracteres. Puede acceder a los caracteres uno a la vez con el operador de corchete:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

La segunda declaración extrae el caracter en la posición de índice 1 de la variable `fruit` y lo asigna a la variable `letter`.

La expresión entre corchetes se llama un índice. El índice indica qué caracter de la secuencia desea.

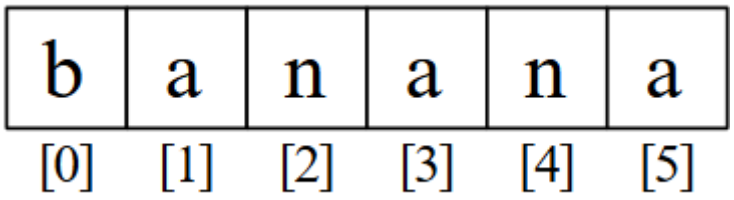
Pero puede que no consigas lo que esperas:

```
>>> print(letter)
a
```

Para la mayoría de las personas, la primera letra de "banana" es `b`, no `a`. Pero en Python, el índice es un desplazamiento desde el principio de la cadena, y el índice de la primera letra es cero.

```
>>> letter = fruit[0]
>>> print(letter)
b
```

Entonces `b` es la 0ª letra ("zero-eth") de "banana ", `a` es la 1ª letra ("one-eth"), y `n` es la 2ª letra ("two-eth").



Copyright 2026

Imagen - String Indexes

Puedes usar cualquier expresión, incluidas las variables y los operadores, como un índice, pero el valor del índice debe ser un entero. De lo contrario obtendrás:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

Obteniendo la longitud de una cadena usando `len`

`len` es una función incorporada que devuelve el número de caracteres en una cadena:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

Para obtener la última letra de una cadena, puedes tener la tentación de probar algo como esto:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

La razón para el `IndexError` es que no hay una letra en `banana` con el índice 6. Ya que comenzamos a contar desde cero, las seis letras están numeradas del 0 al 5. Para obtener el último caracter, debes restar 1 de `lenght`:

```
>>> last = fruit[length-1]
>>> print(last)
a
```

Alternativamente, puede usar índices negativos, que cuentan hacia atrás desde el final de la cadena. La expresión `fruit[-1]` extrae la última letra, `fruit[-2]` extrae la penúltima, y así sucesivamente.

Copyright 2026

Atravesar una cadena con un bucle

Muchos algoritmos implican procesar una cadena, caracter a caracter. A menudo comienzan al principio, seleccionan un caracter por iteración, le hacen algo y continúan hasta el final. Este patrón de procesamiento se denomina **recorrido**. Una forma de escribir un recorrido es con un bucle `while`:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

Este bucle atraviesa la cadena y muestra cada letra en una línea separada. La condición del bucle es `index < len(fruit)`, por lo que cuando `index` es igual a la longitud de la cadena, la condición es falsa, y el cuerpo del bucle no se ejecuta. El último caracter accedido es el que tiene el índice `len(fruit)-1`, que es el último caracter de la cadena.

Ejercicio 1 escribe un bucle `while` que comience en el último caracter de la cadena y avance hacia el primer caracter de la cadena, imprimiendo cada letra en una línea separada.

Otra forma de escribir un recorrido es con un bucle `for`:

```
for char in fruit:
    print(char)
```

Cada vez que pasa por el bucle, el siguiente caracter de la cadena se asigna a la variable `char`. El bucle continúa hasta que no quedan caracteres.

String slices (segmentando cadenas)

Un trozo de una cadena se llama **segmento**. Seleccionar segmento es similar a seleccionar un caracter:

Copyright 2026

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

El operador devuelve el segmento desde el caracter "n-eth" al caracter "m-eth", incluido el primero pero excluyendo el último.

Si omites el primer índice (antes de los dos puntos), el segmento comienza al principio de la cadena. Si omites el segundo índice, la segmento llega hasta el final de la cadena:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

Si el primer índice es mayor o igual que el segundo, el resultado es una **cadena vacía**, representada por dos comillas:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

Una cadena vacía no contiene caracteres y tiene una longitud de 0, pero aparte de eso, es la misma que cualquier otra cadena.

Ejercicio 2: Dado que `fruit` es una cadena, ¿qué significa `fruit[:]`?

Las cadenas son inmutables

Es tentador utilizar el operador en el lado izquierdo de una tarea, con la intención de cambiar un caracter en una cadena. Por ejemplo:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
```

Copyright 2026

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

El "objeto" en este caso es la cadena y el "elemento" es el caracter que intentó asignar. Por ahora, un **objeto** es lo mismo que un valor, pero refinaremos esa definición más adelante. Un **elemento** es uno de los valores en una secuencia.

El motivo del error es que las cadenas son **inmutables**, lo que significa que no puede cambiar una cadena existente. Lo mejor que puedes hacer es crear una nueva cadena que sea una variación del original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello, world!
```

Este ejemplo concatena una nueva primera letra en una porción de "greeting". No tiene efecto en la cadena original.

Bucles y conteos

El siguiente programa cuenta el número de veces que aparece la letra `a` en una cadena:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

Este programa demuestra otro algoritmo llamado **contador**. La variable `count` se inicializa a 0 y luego se incrementa cada vez que se encuentra un `a`. Cuando el bucle sale, `count` contiene el resultado: el número total de `a`s.

Ejercicio 3

Encapsula este código en una función llamada `count`, y generalízalo para que acepte la cadena y la letra como argumentos.

El operador `in`

La palabra `in` es un operador booleano que toma dos cadenas y devuelve `True` si la primera aparece como una subcadena en la segunda:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

Comparación de cadenas

Los operadores de comparación trabajan en cadenas. Para ver si dos cadenas son iguales:

```
if word == 'banana':
    print('All right, bananas.')
```

Otras operaciones de comparación son útiles para poner las palabras en orden alfabético:

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python no maneja las letras mayúsculas y minúsculas de la misma manera que las personas. Todas las letras mayúsculas vienen antes de todas las letras minúsculas, así que:

```
Your word, Pineapple, comes before banana.
```

Una forma común de abordar este problema es convertir las cadenas a un formato estándar, como en minúsculas, antes de realizar la comparación. Tenlo en cuenta en caso de que tengas que defenderte contra un hombre armado con una piña.

Copyright 2026

métodos 'de cadena'

Las cadenas son un ejemplo de objetos de Python. Un objeto contiene tanto datos (la propia cadena real) como **métodos**, que son efectivamente funciones que están integradas en el objeto y están disponibles para cualquier **instancia** (copia) del objeto.

Python tiene una función llamada `dir` que lista los métodos disponibles para un objeto. La función `type` muestra el tipo de un objeto y la función `dir` muestra los métodos disponibles.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
>>> dir(stuff)
['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']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
>>>
```

Mientras que la función `dir` enumera los métodos, y puedes usar `help` para obtener alguna documentación simple sobre un método, una mejor fuente de documentación para los métodos de cadena sería https://docs.python.org/3.7/library/stdtypes.html#string-methods.

Copyright 2026

Llamar a un **método** es similar a llamar a una función (toma argumentos y devuelve un valor) pero la sintaxis es diferente. Llamamos a un método agregando el nombre del método al nombre de la variable utilizando el punto como delimitador.

Por ejemplo, el método `upper` toma una cadena y devuelve una nueva cadena con todas las letras en mayúsculas:

En lugar de la sintaxis de la función `upper(word)`, utiliza la sintaxis del método `word.upper()`.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print(new_word)
BANANA
```

Esta forma de notación de puntos especifica el nombre del método, `upper`, y la cadena a la que aplicar el método. Los paréntesis vacíos indican que este método no toma ningún argumento.

Una llamada al método se llama **invocación**; en este caso, diríamos que estamos invocando `upper` en `word`.

Por ejemplo, hay un método de cadena llamado `find` que busca la posición de una cadena dentro de otra:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
```

En este ejemplo, invocamos `find` en `word` y pasamos la letra que buscamos como parámetro.

El método `find` puede encontrar subcadenas y caracteres:

```
>>> word.find('na')
2
```

Puede tomar como segundo argumento el índice donde debe comenzar:

```
>>> word.find('na', 3)
4
```

Copyright 2026

Una tarea común es eliminar los espacios en blanco (espacios, tabuladores o nuevas líneas) desde el principio y el final de una cadena usando el método `strip`:

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

Algunos métodos como **startswith** devuelven valores booleanos.

```
>>> line = 'Have a nice day'
>>> line.startswith('Have')
True
>>> line.startswith('h')
False
```

Notarás que `startswith` requiere mayúsculas y minúsculas, por lo que a veces tomamos una línea y las mapeamos en minúsculas antes de realizar cualquier comprobación utilizando el método `lower`.

```
>>> line = 'Have a nice day'
>>> line.startswith('h')
False
>>> line.lower()
'have a nice day'
>>> line.lower().startswith('h')
True
```

En el último ejemplo, se llama al método `lower` y luego usamos `startswith` para ver si la cadena en minúsculas resultante comienza con la letra "h". Mientras tengamos cuidado con el orden, podemos realizar múltiples llamadas de método en una sola expresión.

Ejercicio 4

Existe un método de cadena llamado `count` que es similar a la función en el ejercicio anterior. Lee la documentación de este método en https://docs.python.org/3.7/library/stdtypes.html#string-methods y escribe una invocación que cuente el número de veces que aparece la letra a en "banana".

Copyright 2026

Parseando cadenas

A menudo, queremos buscar en una cadena y encontrar una subcadena. Por ejemplo, si se nos presentara una serie de líneas con el siguiente formato:

`From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008`

y quisiéramos extraer solo la segunda mitad de la dirección (es decir, `uct.ac.za`) de cada línea, podemos hacerlo usando el método `find` y el corte de cadena.

Primero, encontraremos la posición del signo en la cadena. Luego encontraremos la posición del primer espacio **después de** el signo de entrada. Y luego usaremos el corte de cadena para extraer la parte de la cadena que estamos buscando.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

Usamos una versión del método `find` que nos permite especificar una posición en la cadena donde queremos que `find` comience a buscar. Cuando cortamos, extraemos los caracteres de "uno más allá del signo de entrada hasta **pero sin incluir** el caracter de espacio".

La documentación para el método `find` está disponible en
https://docs.python.org/3.7/library/stdtypes.html#string-methods.

Operador de formato

El operador de formato, `%` nos permite construir cadenas, reemplazando partes de las cadenas con los datos almacenados en variables. Cuando se aplica a enteros, `%` es el operador de módulo. Pero cuando el primer operando es una cadena, `%` es el operador de formato.

El primer operando es la **cadena de formato**, que contiene una o más **secuencias de formato** que especifican cómo se formatea el segundo operando. El resultado es una cadena.

Por ejemplo, la secuencia de formato "%d" significa que el segundo operando debe formatearse como un entero ("d" significa "dígito"):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

El resultado es la cadena "42", que no debe confundirse con el valor entero "42".

Una secuencia de formato puede aparecer en cualquier parte de la cadena, por lo que puede incrustar un valor en una oración:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

Si hay más de una secuencia de formato en la cadena, el segundo argumento debe ser una tupla [1]. Cada secuencia de formato se empareja con un elemento de la tupla, en orden.

El siguiente ejemplo utiliza "%d" para formatear un número entero, "%g" para formatear un número de punto flotante (no pregunte por qué) y "%s" para formatear una cadena:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

El número de elementos en la tupla debe coincidir con el número de secuencias de formato en la cadena. Los tipos de elementos también deben coincidir con las secuencias de formato:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

Copyright 2026

En el primer ejemplo, no hay suficientes elementos; en el segundo, el elemento es del tipo incorrecto.

El operador de formato es poderoso, pero puede ser difícil de usar. Puedes leer más sobre esto en

https://docs.python.org/3.7/library/stdtypes.html#printf-style-string-formatting.

Depuración

Una habilidad que deberías cultivar a medida que programas siempre es preguntarte: "¿Qué podría salir mal aquí?" o alternativamente, "¿Qué locura podría hacer nuestro usuario para bloquear nuestro programa (aparentemente) perfecto?"

Por ejemplo, mira el programa que usamos para demostrar el bucle `while` en el capítulo sobre iteración:

https://trinket.io/embed/python3/406b93ae34

Mira lo que sucede cuando el usuario ingresa una línea de entrada vacía:

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
```

El código funciona bien hasta que se presenta una línea vacía. Entonces no hay caracter `0`, por lo que obtenemos un error. Existen dos soluciones para hacer que la línea tres sea "segura" incluso si la línea está vacía.

Una posibilidad es simplemente usar el método `startswith` que devuelve `False` si la cadena está vacía.

```
    if line.startswith('#'):
```

Otra forma es escribir de forma segura la instrucción `if` utilizando el patrón **guardian** y asegurarse de que la segunda expresión lógica se evalúa solo cuando hay al menos un caracter en la cadena:

```
    if len(line) > 0 and line[0] == '#':
```

Ejercicios

Ejercicio 5 toma el siguiente código de Python que almacena una cadena: `

`str = 'X-DSPAM-Confidence: 0.8475'`

Usa `find` y el corte de cadena para extraer la parte de la cadena después del caracter de dos puntos y luego use la función `float` para convertir la cadena extraída en un número de punto flotante.

Ejercicio 6

Lee la documentación de los métodos de cadena en

https://docs.python.org/3.7/library/stdtypes.html#string-methods

Es posible que desees experimentar con algunos de ellos para asegurarte de que comprendes cómo funcionan. `strip` y `replace` son particularmente útiles.

La documentación utiliza una sintaxis que puede ser confusa. Por ejemplo, en `find (sub [, start [, end]])`, los corchetes indican argumentos opcionales. Entonces se requiere `sub`, pero `start` es opcional, y si incluye `start`, entonces `end` es opcional.

> “ [1]. Una tupla es una secuencia de valores separados por comas dentro de un par de paréntesis. Cubriremos las tuplas en el Capítulo 10 ↩

Revision #2
Created 2025-04-05 09:29:14 CEST by Javier Quintana
Updated 2025-04-05 09:30:07 CEST by Javier Quintana

Copyright 2026