Chapter 6

Cadenas

felipe
012345.

6.1 Una cadena es una secuencia

Una cadena es una *secuencia* de caracteres. Puedes acceder a los caracteres de uno en uno con el operador corchete:

```
>>> fruta = 'banana'
>>> letra = fruta[1]
```

La segunda sentencia extrae el carácter en la posición del indice 1 de la variable `fruta` y la asigna a la variable `letra`.

La expresión en los corchetes es llamada *índice*. El índice indica qué carácter de la secuencia quieres (de ahí el nombre).

Pero podrías no obtener lo que esperas:

```
>>> print(letra)
a
```

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

```
>>> letra = fruta[0]
>>> print(letra)
b
```

Así que "b" es la letra 0 ("cero") de "banana", "a" es la letra con índice 1, y "n" es la que tiene índice 2, etc.

Puedes usar cualquier expresión, incluyendo variables y operadores, como un índice, pero el valor del índice tiene que ser un entero. De otro modo obtendrás:

```
>>> letra = fruta[1.5]
TypeError: string indices must be integers
```

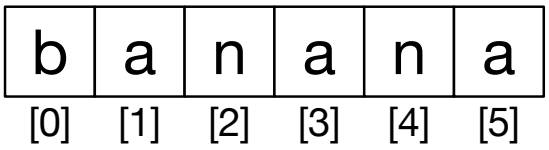

Figure 6.1: Indices de Cadenas

6.2 Obtener el tamaño de una cadena usando `len`

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

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

Para obtener la última letra de una cadena, podrías estar tentado a probar algo como esto:

```
>>> tamaño = len(fruta)
>>> ultima = fruta[tamaño]
IndexError: string index out of range
```

La razón de que haya un `IndexError` es que ahí no hay ninguna letra en "banana" con el índice 6. Puesto que empezamos a contar desde cero, las seis letras están enumeradas desde 0 hasta 5. Para obtener el último carácter, tienes que restar 1 a `length`:

```
>>> ultima = fruta[tamaño-1]
>>> print(ultima)
a
```

Alternativamente, puedes usar índices negativos, los cuales cuentan hacia atrás desde el final de la cadena. La expresión `fruta[-1]` devuelve la última letra, `fruta[-2]` la penúltima letra, y así sucesivamente.

6.3 Recorriendo una cadena mediante un bucle

Muchos de los cálculos requieren procesar una cadena carácter por carácter. Frecuentemente empiezan desde el inicio, seleccionando cada carácter presente, haciendo algo con él, y continuando hasta el final. Este patrón de procesamiento es llamado un *recorrido*. Una manera de escribir un recorrido es con un bucle `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print(letra)
    indice = indice + 1
```

Este bucle recorre la cadena e imprime cada letra en una línea cada una. La condición del bucle es `indice < len(fruta)`, así que cuando `indice` es igual al tamaño de la cadena, la condición es falsa, y el código del bucle no se ejecuta. El último carácter accedido es el que tiene el índice `len(fruta)-1`, el cual es el último carácter en la cadena.

Ejercicio 1: Escribe un bucle `while` que comience con el último carácter en la cadena y haga un recorrido hacia atrás hasta el primer carácter en la cadena, imprimiendo cada letra en una línea independiente.

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

```
for caracter in fruta:
    print(caracter)
```

Cada vez que iteramos el bucle, el siguiente carácter en la cadena es asignado a la variable `caracter`. El ciclo continúa hasta que no quedan caracteres.

6.4 Rebanado de una cadena

Un segmento de una cadena es llamado *rebanado*. Seleccionar un rebanado es similar a seleccionar un carácter:

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

El operador [n:m] retorna la parte de la cadena desde el "n-ésimo" carácter hasta el "m-ésimo" carácter, incluyendo el primero pero excluyendo el último.

Si omites el primer índice (antes de los dos puntos), el rebanado comienza desde el inicio de la cadena. Si omites el segundo índice, el rebanado va hasta el final de la cadena:

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

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

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

Una cadena vacía no contiene caracteres y tiene un tamaño de 0, pero fuera de esto es lo mismo que cualquier otra cadena.

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

6.5 Los cadenas son inmutables

Puede ser tentador utilizar el operador [] en el lado izquierdo de una asignación, con la intención de cambiar un carácter en una cadena. Por ejemplo:

```
>>> saludo = 'Hola, mundo!'
>>> saludo[0] = 'J'
TypeError: 'str' object does not support item assignment
```

El "objeto" en este caso es la cadena y el "ítem" es el carácter que tratamos de asignar. Por ahora, un *objeto* es la misma cosa que un valor, pero vamos a redefinir esa definición después. Un *ítem* es uno de los valores en una secuencia.

La razón por la cual ocurre el error es que las cadenas son *inmutables*, lo cual significa que no puedes modificar una cadena existente. Lo mejor que puedes hacer es crear una nueva cadena que sea una variación de la original:

```
>>> saludo = 'Hola, mundo!'
>>> nuevo_saludo = 'J' + saludo[1:]
>>> print(nuevo_saludo)
Jola, mundo!
```

Este ejemplo concatena una nueva letra a una parte de `saludo`. Esto no tiene efecto sobre la cadena original.

6.6 Iterando y contando

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

```
palabra = 'banana'
contador = 0
for letra in palabra:
    if letra == 'a':
        contador = contador + 1
print(contador)
```

Este programa demuestra otro patrón de computación llamado *contador*. La variable `contador` es inicializada a 0 y después se incrementa cada vez que una "a" es encontrada. Cuando el bucle termina, `contador` contiene el resultado: el número total de a's.

Ejercicio 3: Encapsula este código en una función llamada `cuenta`, y hazla genérica de tal modo que pueda aceptar una cadena y una letra como argumentos.

6.7 El operador in

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

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

6.8 Comparación de cadenas

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

```
if palabra == 'banana':
    print('Muy bien, bananas.')
```

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

```
if palabra < 'banana':
    print('Tu palabra, ' + palabra + ', está antes de banana.')
elif palabra > 'banana':
    print('Tu palabra, ' + palabra + ', está después de banana.')
else:
    print('Muy bien, bananas.')
```

Python no maneja letras mayúsculas y minúsculas de la misma forma que la gente lo hace. Todas las letras mayúsculas van antes que todas las letras minúsculas, por ejemplo:

```
Tu palabra, Piña, está antes que banana.
```

Una forma común de manejar este problema es convertir cadenas a un formato estándar, como todas a minúsculas, antes de llevar a cabo la comparación. Ten en cuenta eso en caso de que tengas que defenderte contra un hombre armado con una Piña.

6.9 Métodos de cadenas

Los cadenas son un ejemplo de *objetos* en Python. Un objeto contiene tanto datos (el valor de la cadena misma) como *métodos*, los cuales son efectivamente funciones que están implementadas dentro del objeto y que están disponibles para cualquier *instancia* del objeto.

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

```
>>> cosa = 'Hola mundo'
>>> type(cosa)
<class 'str'>
>>> dir(cosa)
['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.
>>>
```

Aunque la función `dir` lista los métodos y puedes usar la función `help` para obtener una breve documentación de un método, una mejor fuente de documentación para los métodos de cadenas se puede encontrar en https://docs.python.org/library/stdtypes.html#string-methods.

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

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

En vez de la sintaxis de función `upper(word)`, éste utiliza la sintaxis de método `word.upper()`.

```
>>> palabra = 'banana'
>>> nueva_palabra = palabra.upper()
>>> print(nueva_palabra)
BANANA
```

Esta forma de notación con punto especifica el nombre del método, `upper`, y el nombre de la cadena al que se le aplicará el método, `palabra`. Los paréntesis vacíos indican que el método no toma argumentos.

Una llamada a un método es conocida como una *invocación*; en este caso, diríamos que estamos invocando `upper` en `palabra`.

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

```
>>> palabra = 'banana'
>>> indice = palabra.find('a')
>>> print(indice)
1
```

En este ejemplo, invocamos `find` en `palabra` y pasamos la letra que estamos buscando como un parámetro.

El método `find` puede encontrar subcadenas así como caracteres:

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

También puede tomar como un segundo argumento el índice desde donde debe empezar:

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

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

```
>>> linea = '  Aquí vamos  '
>>> linea.strip()
'Aquí vamos'
```

Algunos métodos como *startswith* devuelven valores booleanos.

```
>>> linea = 'Que tengas un buen día'
>>> linea.startswith('Que')
True
>>> linea.startswith('q')
False
```

Puedes notar que `startswith` requiere que el formato (mayúsculas y minúsculas) coincida, de modo que a veces tendremos que tomar la línea y cambiarla completamente a minúsculas antes de hacer la verificación, utilizando el método `lower`.

```
>>> linea = 'Que tengas un buen día'
>>> linea.startswith('q')
False
>>> linea.lower()
'que tengas un buen día'
>>> linea.lower().startswith('q')
True
```

En el último ejemplo, el método `lower` es llamado y después usamos `startswith` para ver si la cadena resultante en minúsculas comienza con la letra "q". Siempre y cuando seamos cuidadosos con el orden, podemos hacer múltiples llamadas a métodos en una sola expresión.

Ejercicio 4: Hay un método de cadenas llamado `count` que es similar a la función del ejercicio previo. Lee la documentación de este método en:

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

Escribe una invocación que cuenta el número de veces que una letra aparece en "banana".

6.10 Analizando cadenas

Frecuentemente, queremos examinar una cadena para encontrar una subcadena. Por ejemplo, si se nos presentaran una seria de líneas con el siguiente formato:

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

y quisiéramos obtener únicamente la segunda parte de la dirección de correo (esto es, `uct.ac.za`) de cada línea, podemos hacer esto utilizando el método `find` y una parte de la cadena.

Primero tenemos que encontrar la posición de la arroba en la cadena. Después, tenemos que encontrar la posición del primer espacio *después* de la arroba. Y después partiremos la cadena para extraer la porción de la cadena que estamos buscando.

```
>>> dato = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> arrobapos = dato.find('@')
>>> print(arrobapos)
21
>>> espos = dato.find(' ',arrobapos)
>>> print(espos)
31
>>> direccion = dato[arrobapos+1:espos]
>>> print(direccion)
uct.ac.za
>>>
```

Utilizamos una versión del método `find` que nos permite especificar la posición en la cadena desde donde queremos que `find` comience a buscar. Cuando recortamos una parte de una cadena, extraemos los caracteres desde "uno después de la arroba hasta, *pero no incluyendo*, el carácter de espacio".

La documentación del método `find` está disponible en

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

6.11 El operador de formato

El *operador de formato* % nos permite construir cadenas, reemplazando partes de las cadenas con datos almacenados en variables. Cuando lo aplicamos a enteros, % es el operador módulo. Pero cuando es aplicado a una cadena, % es el operador de formato.

El primer operando es la *cadena a formatear*, la cual contiene una o más *secuencias de formato* que especifican cómo el segundo operando es formateado. El resultado es una cadena.

Por ejemplo, la secuencia de formato %d significa que el segundo operando debería ser formateado como un entero ("d" significa "decimal"):

```
>>> camellos = 42
>>> '%d' % camellos
'42'
```

El resultado es la cadena '42', el cual no debe ser confundido con el valor entero 42.

Una secuencia de formato puede aparecer en cualquier lugar en la cadena, así que puedes meter un valor en una frase:

```
>>> camellos = 42
>>> 'Yo he visto %d camellos.' % camellos
'Yo he visto 42 camellos.'
```

Si hay más de una secuencia de formato en la cadena, el segundo argumento tiene que ser una tupla[1]. Cada secuencia de formato es relacionada con un elemento de la tupla, en orden.

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

```
>>> 'En %d años yo he visto %g %s.' % (3, 0.1, 'camellos')
'En 3 años yo he visto 0.1 camellos.'
```

El número de elementos en la tupla debe coincidir con el número de secuencias de formato en la cadena. El tipo de los elementos también debe coincidir con la secuencia de formato:

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

[1] Una tupla es una secuencia de valores separados por comas dentro de un par de paréntesis. Veremos tuplas en el Capítulo 10

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

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

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

6.12 Depuración

Una habilidad que debes desarrollar cuando programas es siempre preguntarte a ti mismo, "¿Qué podría fallar aquí?" o alternativamente, "¿Qué cosa ilógica podría hacer un usuario para hacer fallar nuestro (aparentemente) perfecto programa?"

Por ejemplo, observa el programa que utilizamos para demostrar el bucle `while` en el capítulo de iteraciones:

```
while True:
    linea = input('> ')
    if linea[0] == '#' :
        continue
    if linea == 'fin':
        break
    print(linea)
print('¡Terminado!')

# Código: https://es.py4e.com/code3/copytildone2.py
```

Mira lo que pasa cuando el usuario introduce una línea vacía como entrada:

```
> hola a todos
hola a todos
> # no imprimas esto
> ¡imprime esto!
¡imprime esto!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if linea[0] == '#':
IndexError: string index out of range
```

El código funciona bien hasta que se presenta una línea vacía. En ese momento no hay un carácter cero, por lo que obtenemos una traza de error (traceback). Existen dos soluciones a esto para convertir la línea tres en "segura", incluso si la línea está vacía.

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

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

Otra forma segura es escribir una sentencia `if` utilizando el patrón *guardián* y asegurarse que la segunda expresión lógica es evaluada sólo cuando hay al menos un carácter en la cadena:

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

6.13 Glosario

bandera Una variable booleana utilizada para indicar si una condición es verdadera o falsa.

búsqueda Un patrón de recorrido que se detiene cuando encuentra lo que está buscando.

cadena a formatear Una cadena, usado con el operador de formato, que contiene secuencias de formato.

cadena vacía una cadena sin caracteres y de tamaño 0, representada por dos comillas sencillas.

contador Una variable utilizada para contar algo, usualmente inicializada a cero y luego incrementada.

índice Un valor entero utilizado para seleccionar un ítem en una secuencia, tal como un carácter en una cadena.

inmutable La propiedad de una secuencia cuyos elementos no pueden ser asignados.

invocación Una sentencia que llama un método.

ítem Uno de los valores en una secuencia.

método Una función que está asociada a un objeto y es llamada utilizando la notación de punto.

objeto Algo a lo que una variable puede referirse. Por ahora, puedes usar "objeto" y "valor" indistintamente.

operador de formato Un operador, `%`, que toma una cadena de formato y una tupla y genera una cadena que incluye los elementos de la tupla formateados como se especifica en la cadena de formato.

rebanado Una parte de una cadena especificado por un rango de índices.

recorrido Iterar a través de los ítems de una secuencia, ejecutando una operación similar en cada uno.

secuencia Un conjunto ordenado; esto es, un conjunto de valores donde cada valor es identificado por un índice entero.

secuencia de formato Una secuencia de caracteres en una cadena a formatear, como `%d`, que especifica cómo un valor debe ser formateado.

6.14 Ejercicios

Ejercicio 5: Toma el siguiente código en Python que almacena una cadena:

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

Utiliza `find` y una parte de la cadena para extraer la porción de la cadena después del carácter dos puntos y después utiliza la función `float` para convertir la cadena extraída en un número de punto flotante.

Ejercicio 6: Lee la documentación de los métodos de cadenas en https://docs.python.org/library/stdtypes.html#string-methods Quizá quieras experimentar con algunos de ellos para asegurarte de entender como funcionan. `strip` y `replace` son particularmente útiles.

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