**Listas en pyton**

Son la esencia de python, las podríamos definir como una secuencia ordenada de elementos encerrados entre corchetes y separados por comas.

*>>> Lista = ["a", "b", "alfa", "beta", "3"]  
>>> Lista  
['a', 'b', 'alfa', 'beta', '3']*

Tenemos una lista de 5 elementos, podemos referenciar un elemento en concreto de la lista.

*>>> Lista[0]  
'a'  
>>>Lista[1]  
'b'  
>>>Lista[4]  
'3'*

Referirnos a cada elemento individual de la lista mediante su posición dentro de ella, así el primer elemento de una lista (no vacía) lo tendremos siempre en la posición 0, el segundo en la 1, etc. Como es nuestro caso en una lista de cinco elementos, estos estarán referenciados del 0 al 4.

También podemos usar***índices negativos*** para acceder a sus elementos. Si el índice es negativo, este empieza a contar a partir del último elemento. En nuestro ejemplo los elementos irán de -1 a -5, donde -1 nos da el último elemento de la lista y -5 el primero.

*>>>Lista[-1]  
'3'  
>>>Lista[-4]  
'b'  
>>>Lista[-5]  
'a'*

Para saber cuantos elementos hay en una lista disponemos de la función **len**

*>>>len(Lista)  
5*

Los elementos no son inmutables, por lo que podemos ***cambiar el valor*** de los elementos.

*>>> la=[1, 2, 3, 4, 5]  
>>> la  
[1, 2, 3, 4, 5]  
>>> la[2] = la[1] + la[3] + 5  
>>> la  
[1, 2, 11, 4, 5]*

Podemos obtener un subconjunto de elementos de una lista especificando dos índices (inicio y fin)***slicing***, el resultado será una nueva lista que contendrá los elementos que se encuentren en el rango de los índices.

*>>>Lista[2:4]  
['alfa', 'beta']*

Obtenemos una nueva lista con los elementos del rango especificado. El rango va desde el elemento 2 (primer índice) hasta el elemento 3 (segundo índice -1). Como observareis el primer índice si que se incluye en el rango pero el segundo se queda fuera [2:4) por lo que solo obtenemos los elementos 2 y 3.

*>>>Lista[2:2]  
[]  
>>>Lista[2:3]  
['alfa']*

aquí también podemos usar los índices negativos.

*>>> Lista[2:-1]  
['alfa', 'beta']*

*>>> Lista[-3:-1]  
['alfa', 'beta']*

Obtendremos los dos primeros elementos de la lista.

*>>> Lista[0:2]  
['a', 'b']*

Extraemos el primer elemento de la lista.

*>>> Lista[1:5]  
['b', 'alfa', 'beta', '3']*

Veamos otra forma de realizar esto. ***(slice assignament)***

*>>> # dos primeros elementos  
... Lista[:2]  
['a', 'b']  
>>> # todos menos el primero  
... Lista[1:]  
['b', 'alfa', 'beta', '3']  
>>> # toda la lista  
... Lista[:]  
['a', 'b', 'alfa', 'beta', '3']*

Las listas son dinámicas por lo que podemos introducir nuevos elementos en ellas ***append.***

*>>> Lista.append("nou")  
>>> Lista  
['a', 'b', 'alfa', 'beta', '3', 'nou']*

***append,*** añade un único elemento al final de la lista, también podemos insertarlo en una posición en concreto ***insert***.

*>>> Lista.insert(3,"insert")  
>>> Lista  
['a', 'b', 'alfa', 'insert', 'beta', '3', 'nou']*

Tenemos esta otra forma de añadir uno o mas elementos

*>>> la=[1,2,3,4,5]  
>>> la[2:2]=['a', 'b']  
>>> la  
[1, 2, 'a', 'b', 3, 4, 5]*

Podemos añadir los elementos de otra lista concatenar (***extend***).

*>>> Lista.extend(['1','2'])  
>>> Lista  
['a', 'b', 'alfa', 'insert', 'beta', '3', 'nou', '1', '2']*

En este último ejemplo tenemos que pararnos a reflexionar un poco, fijaros que lo que hemos hecho ha sido introducir en nuestra lista los elementos que contenía una segunda lista, si queréis ver lo que quiero decir probar a ejecutar la siguiente instrucción.

*>>> len(Lista)  
9  
>>> Lista.append(['1','2'])  
>>> Lista  
['a', 'b', 'alfa', 'insert', 'beta', '3', 'nou', '1', '2', ['1', '2']]  
>>> len(Lista)  
10*

Fijaros que lo que hemos hecho ha sido añadir un único elemento (de tipo lista) a la lista. Ahora hemos abierto un nuevo frente, todos los elementos de nuestra lista eran del mismo tipo, con la última instrucción hemos añadido un elemento de tipo lista, esto es posible ya que las listas en python pueden contener elementos de diferentes tipos.

*>>> Lista=[1, 'a', ['juan', 'pepe'], 'a']  
>>> Lista  
[1, 'a', ['juan', 'pepe'], 'a']  
>>> len(Lista)  
4*

Es posible el anidamiento.

*>>> l1 = [1, 3, 5]  
>>> l2 = ['a', 'b',****l1****, 'c']  
>>> len(l2)  
4  
>>> l2  
['a', 'b', [1, 3, 5], 'c']  
>>> l2[2][1]  
3  
>>> l2[2].append(7)  
>>> l2  
['a', 'b', [1, 3, 5, 7], 'c']  
>>> l1  
[1, 3, 5, 7]*

Fijaros que en la segunda linea, el tercer elemento es una lista, y fijaros que l2[2] se refiere a l1, tal vez sea un poco complicado si no estáis muy duchos, pero creo que vale la pena que perdáis un poco el tiempo en observarlo y comprenderlo.

Podemos borrar un elemento de la Lista (**remove**).

*>>> Lista  
[1, 'a', ['juan', 'pepe'], 'a']  
>>> Lista.remove('a')  
>>> Lista  
[1, ['juan', 'pepe'], 'a']*

Igual que en la búsqueda, elimina la primera aparición del elemento y si no lo encuentra nos lanzara una excepción.

También podemos borrar un elemento por su índice (***del***).

*>>> del Lista[1]  
>>> Lista  
[1, 'a']*

otra forma para borrar uno o varios elementos es la siguiente. (eliminamos los dos primeros elementos)

*>>> la = [1, 2, 11, 4, 5]  
>>> la   
[1, 2, 11, 4, 5]  
>>> la[0:2]=[]  
>>> la  
[11, 4, 5]*

Y por último tenemos ***pop***, que nos permite borrar el último elemento de la Lista como si de una cola se tratara. Esta función nos devuelve el último elemento de la lista y lo borra de ella.

*>>> Lista  
[1, 'a']  
>>> Lista.pop()  
'a'  
>>> Lista  
[1]*

Si a la función***pop*** se le añade un índice, nos mostrará y eliminará de la lista el elemento apuntado por el índice.

*>>> lst = range(10)  
>>> lst  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> lst.pop(3)  
3  
>>> lst  
[0, 1, 2, 4, 5, 6, 7, 8, 9]*

Hasta ahora nos hemos referido a los elementos a partir de su índice, las listas nos permiten hacer la operación inversa, la función***index*** a la que pasado un elemento, nos devuelve su índice.

*>>> Lista.index(1)  
0  
>>> Lista.index(['juan', 'pepe'])  
2  
>>> Lista.index('a')  
1  
>>> Lista.index('aa')  
Traceback (most recent call last):  
File "", line 1, in   
ValueError: list.index(x): x not in list*

Como podéis ver, nos devuelve el índice del primer elemento que coincida, el elemento 'a' lo tenemos en 1 y en 3. Si hacemos una búsqueda de un elemento que no existe, como es en el último caso, el programa nos devolverá una excepción.

Podemos evitar el error anterior si antes de hacer la búsqueda nos aseguramos que el elemento exista, esto lo hacemos mediante ***in*** que nos devolverá ***True***o ***False*** dependiendo si el elemento existe o no.

*>>> 'aa' in Lista  
False  
>>> 1 in Lista  
True*

Los valores booleanos True y False, se incorporaron a python a partir de la versión 2.2.1, hasta entonces se establecían una serie de reglas que determinaban cuando algo era cierto o verdadero.

* Cero es falso; el resto de los números son verdaderos.
* Una cadena vacía ("") es falso, cualquier otra cadena es verdadera.
* Una lista vacía ([]) es falso; el resto de las listas son verdaderas.
* Una tupla vacía (()) es falso; el resto de las tuplas son verdaderas.
* Un diccionario vacío ({}) es falso; todos los otros diccionarios son verdaderos.

Esto se sigue manteniendo la diferencia es que ahora disponemos de True (1) y False (0).

También nos puede ayudar la función ***count*** que nos dirá el número de veces que aparece un elemento en una lista.

*>>> lst = [1, 2, 3, 1, 3, 5, 3, 6, 3]  
>>> lst.count(1)  
2  
>>> lst.count(3)  
4  
>>> lst.count(6)  
1  
>>> lst.count(11)  
0*

***Operadores***  
Podemos concatenar dos listas. ***+***

*>>> l1 = [1, 2, 3, 4]  
>>> l2 = ['a', 'b', 'c', 'd']  
>>> l1 + l2  
[1, 2, 3, 4, 'a', 'b', 'c', 'd']  
>>> l2 + l1  
['a', 'b', 'c', 'd', 1, 2, 3, 4]  
>>> l1 + l1  
[1, 2, 3, 4, 1, 2, 3, 4]  
>>> l1 += l2  
>>> l1  
[1, 2, 3, 4, 'a', 'b', 'c', 'd']  
>>> l2  
['a', 'b', 'c', 'd']*

El comportamiento de la suma es muy parecido al de ***extend***, la diferencia esta en que la suma devuelve una nueva lista.

Con un poco de imaginación podemos modificar este comportamiento.

*>>> la = range(1,5)  
>>> la  
[1, 2, 3, 4]  
>>> la += [5]  
>>> la  
[1, 2, 3, 4, 5]*

Podemos disponer de un multiplicador ***\****

*>>> l1=[1, 2]  
>>> l1  
[1, 2]  
>>> l1 \* 3  
[1, 2, 1, 2, 1, 2]  
>>> l1  
[1, 2]  
>>> l1 = l1 \* 2  
>>> l1  
[1, 2, 1, 2]  
>>> l1 \*= 2  
>>> l1  
[1, 2, 1, 2, 1, 2, 1, 2]*

El operador ***\**** hace de repetidor, concatenando la lista tantas veces como se indique.

La función ***sort***, nos permite ordenar los elementos de una lista.

*l1 = [3, 5, 1, 3, 4]  
>>> l1.sort()  
>>> l1  
[1, 3, 3, 4, 5]*

La función ***reverse***, invierte los elementos de la lista

*>>> l1 = [1, 2, 3, 4, 5]  
>>> l1.reverse()  
>>> l1  
[5, 4, 3, 2, 1]*

Otra función que puede resultar útil es ***list***, que permite convertir un string en una lista.

*>>> [1, 2] + list("34")  
[1, 2, '3', '4']*

Algo parecido podemos obtener con ***split.*** Que nos devuelve una lista con todas las palabras de una cadena de texto.

*>>> nom = "Pedro Ruiz Mayoral"  
>>> nom.split()  
['Pedro', 'Ruiz', 'Mayoral']*

Y como no la funcion inversa ***join***, que os convierte una lista en una cadena de texto.

*>>> l = ['Pedro', 'Ruiz', 'Mayoral']  
>>> l  
['Pedro', 'Ruiz', 'Mayoral']  
>>> ' '.join(l)  
'Pedro Ruiz Mayoral'  
>>> '<->'.join(l)  
'Pedro<->Ruiz<->Mayoral'*

Podemos ***comparar*** dos listas.

*>>> [1, 2, 3] == [1, 2, 3]  
True  
>>> [1, 2, 3] == [1, 3, 2]  
False  
>>> a = [1, 2, 3]  
>>> b = [1, 2, 3]  
>>> c = a  
>>> a == b  
True  
>>> c == b  
True*

*>>> [1, 2, 3] < [1, 2, 3]  
False  
>>> [1, 2, 3] <= [1, 2, 3]  
True  
>>> [1, 2, 3] > [1, 2, 3]  
False  
>>> [1, 2, 3] > [1, 1, 2]  
True  
>>> [1, 2, 3] < [1, 1, 2]  
False  
>>> [1, 2, 3] < [1, 3, 2]  
True*

y por ultimo una interacción por la lista con ***for***

*>>> li = range(9)  
>>> for x in li:  
... print x\*2  
...  
0  
2  
4  
6  
8  
10  
12  
14  
16*

un ejemplo más útil, podría ser este.

*>>> pagos = [['gasolina', 30], ['cena', 20], ['libros', 12]]  
>>> total = 0  
>>> for pago in pagos:  
... total += pago[1]  
...  
>>> total  
62*

Bueno, esto es todo por hoy. Pensar que solo hemos visto la punta del iceberg, la verdadera potencia de las listas la veremos más adelante, pero antes tenemos que terminar de fijar nuestra base en python.

**7. Cadenas**

**7.1. Un tipo de datos compuesto**

Hasta ahora hemos visto cinco tipos: int, float, bool, NoneType y str. Las cadenas son cualitativamente diferentes de los otros cuatro tipos porque están hechas de piezas más pequeñas — los **caracteres**.

Los tipos que comprenden piezas mas pequeñas se llaman **tipos de datos compuestos**. Dependiendo de lo que hagamos, podemos tratar un tipo de datos compuesto como una única cosa o podemos acceder a sus partes. Esta ambigüedad es útil.

El operador corchete selecciona sólo un carácter de una cadena:

**>>>** fruta = "banana"

**>>>** letra = fruta[1]

**>>> print** letra

La expresión fruta[1] selecciona el carácter número 1 de fruta. La variable letra apunta al resultado. Cuando mostramos letra, nos encontramos con una sorpresa:

a

La primera letra de "banana" no es a. A no ser que usted sea un programador. Por perversas razones, los científicos de la computación siempre empiezan a contar desde cero. La 0-ésima letra (“cero- ésima”) de "banana" es b. La 1-ésima (“uno- ésima”) es a, y la 2-ésima (“dos-ésima”) letra es n.

*Nota del traductor:* La numeración ordinal — por todos conocida — no contempla al cero como parte de su verbalización, por lo que se considera apropiada la introducción de esta forma de expresar la numeración de caracteres, dado el objetivo que se persigue.

Si quiere la cero-ésima letra de una cadena, simplemente pone 0, o cualquier expresión de valor 0, entre los corchetes:

**>>>** letra = fruta[0]

**>>> print** letra

b

A la expresión entre corchetes se le llama **índice**. Un índice identifica a un miembro de un conjunto ordenado, en este caso el conjunto de caracteres de la cadena. El índice *indica* cuál quiere usted, de ahí el nombre. El índice puede ser cualquier expresión entera.

**7.2. Longitud**

La función len devuelve el número de caracteres de una cadena:

**>>>** fruta = "banana"

**>>>** len(fruta)

6

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

longitud = len(fruta)

ultima = fruta[longitud] *# ¡ERROR!*

Eso no funcionará. Esto provoca el error en tiempo de ejecución IndexError: string index out of range. La razón es que no hay una seis-ésima letra en "banana". Como empezamos a contar desde cero, las seis letras están numeradas del 0 al 5. Para obtener el último carácter tenemos que restar 1 a la longitud:

longitud = len(fruta)

ultima = fruta[longitud-1]

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

**7.3. Recorrido y el bucle for**

Muchos cálculos implican procesar una cadena carácter por carácter. A menudo empiezan por el principio, seleccionan cada carácter por turno, hacen algo con él y siguen hasta el final. Este patrón de proceso se llama **recorrido**. Una forma de codificar un recorrido es con una sentencia while:

indice = 0

**while** indice < len(fruta):

letra = fruta[indice]

**print** letra

indice += 1

Este bucle recorre la cadena y muestra cada letra en una línea distinta. La condición del bucle es indice &lt; len(fruta), de modo que cuando indice es igual a la longitud de la cadena, la condición es falsa y no se ejecuta el cuerpo del bucle. El último carácter al que se accede es el que tiene el índice len(fruta)-1, que es el último carácter de la cadena.

Usar un índice para recorrer un conjunto de valores es tan común que Python proporciona una sintaxis alternativa más simple — el bucle for:

**for** carac **in** fruta:

**print** carac

Cada vez que recorremos el bucle, se asigna a la variable carac el siguiente carácter de la cadena. El bucle continúa hasta que no quedan más caracteres.

El ejemplo siguiente muestra cómo usar la concatenación y a un bucle for para generar una serie abecedaria. “Abecedaria” se refiere a la serie o lista en la que cada uno de los elementos aparece en orden alfabético. Por ejemplo, en el libro de Robert McCloskey *Make Way for Ducklings*, los nombres de los patitos son Jack, Kack, Lack, Mack, Nack, Ouack, Pack, y Quack. Este bucle muestra esos nombres en orden:

prefijos = "JKLMNOPQ"

sufijo = "ack"

**for** letra **in** prefijos:

**print** letra + sufijo

La salida de este programa es:

Jack

Kack

Lack

Mack

Nack

Oack

Pack

Qack

Por supuesto, esto no es del todo correcto, porque “Ouack” y “Quack” no están correctamente escritos. Usted solucionará esto en un ejercicio más adelante.

**7.4. Porciones de cadenas**

A la subcadena de una cadena se le llama **porción**. La selección de una porción es similar a la selección un carácter:

**>>>** s = "Pedro, Pablo, y Mario"

**>>> print** s[0:5]

Pedro

**>>> print** s[7:12]

Pablo

**>>> print** s[16:21]

Mario

El operador [n:m] devuelve la parte de la cadena desde el “n-ésimo” carácter hasta el “m-ésimo”, incluyendo el primero pero excluyendo el último. Este comportamiento contradice a nuestra intuición; tiene más sentido si imagina los índices señalando *entre* los caracteres, como en el siguiente diagrama:

cadena 'banana'

Si omite el primer índice (antes de los dos puntos), la porción comienza al principio de la cadena. Si omite el segundo índice, la porción llega al final de la cadena. Así:

**>>>** fruta = "banana"

**>>>** fruta[:3]

'ban'

**>>>** fruta[3:]

'ana'

¿Qué cree que significa s[:]?

**7.5. Comparación de cadenas**

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

**if** palabra == "banana":

**print** "Sí, ¡no tenemos bananas!"

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

**if** palabra < "banana":

**print** "Su palabra," + palabra + ", va antes de banana."

**elif** word > "banana":

**print** "Su palabra," + palabra + ", va después de banana."

**else**:

**print** "Sí, ¡no tenemos bananas!"

Sin embargo, debería ser consciente de que Python no maneja las letras mayúsculas y minúsculas como lo hace la gente. Todas las letras mayúsculas van antes de las letras minúsculas. Por lo tanto:

Su palabra, Cebra, va antes de banana.

Una forma común de abordar este problema es convertir las cadenas a un formato estándar — como pueden ser las minúsculas— antes de realizar la comparación. Un problema mayor es hacer que el programa se de cuenta de que las cebras no son frutas.

**7.6. Las cadenas son inmutables**

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

saludo = "¡Hola todo el mundo!"

saludo[2] = 'L' *# ¡ERROR!*

**print** saludo

*Nota del traductor:* De acuerdo con la codificación de caracteres en utf-8 ó latin-1 (véase la Nota del traductor del capítulo 2) el carácter de admiración, ‘¡’, en la variable saludo ocupa dos posiciones, de ahí que la letra ‘H’ esté localizada en el índice 2.

En lugar de presentar la salida ¡Lola todo el mundo!, este código presenta el siguiente error en tiempo de ejecución TypeError: 'str' object does not support itemassignment.

Las cadenas son **inmutables**, lo que significa que no puede cambiar una cadena existente. Lo más que puede hacer es crear una nueva cadena que sea una variación de la original:

saludo = "¡Hola todo el mundo!"

nuevoSaludo = '¡' + 'L' + saludo[3:]

**print** nuevoSaludo

Aquí la solución es concatenar la apertura del signo de admiración y una nueva primera letra a una porción de saludo. Esta operación no tiene efecto sobre la cadena original.

**7.7. El operador in**

El operador in prueba si una cadena es una subcadena de otra:

**>>>** 'z' **in** 'manzana'

True

**>>>** 'i' **in** 'manzana'

False

**>>>** 'nz' **in** 'manzana'

True

**>>>** 'az' **in** 'manzana'

False

Observe que una cadena es una subcadena de si misma:

**>>>** 'a' **in** 'a'

True

**>>>** 'manzana' **in** 'manzana'

True

Combinando el operador in con la concatenación de cadenas usando +, podemos escribir una función que elimina todas las vocales de una cadena:

**def** elimina\_vocales(s):

vocales = "aeiouAEIOU"

s\_sin\_vocales = ""

**for** letra **in** s:

**if** letra **not** **in** vocales:

s\_sin\_vocales += letra

**return** s\_sin\_vocales

Pruebe esta función para confirmar que realiza lo que queremos que haga.

**7.8. Una función encuentra**

¿Qué hace la siguiente función?

def encuentra(cadena, carac):

indice = 0

while indice &lt; len(cadena):

if cadena[indice] == carac:

return indice

indice += 1

return -1

En cierto sentido, encuentra es lo contrario del operador []. En lugar de tomar un índice y extraer el carácter correspondiente, toma un carácter y encuentra el índice donde aparece el carácter. Si el carácter no se encuentra, la función devuelve -1.

Este es el primer ejemplo que hemos visto de una sentencia return `` dentro de un bucle. Si ``cadena[indice] == carac, la función devuelve inmediatamente el índice, escapando del bucle prematuramente.

Si el carácter no aparece en la cadena, entonces el programa sale del bucle normalmente y devuelve -1.

Este patrón de computación se llama a veces un recorrido “eureka” porque tan pronto como encontramos lo que buscamos, podemos gritar “¡Eureka! ” y dejar de buscar.

**7.9. Iterando y contando**

El siguiente programa cuenta el número de veces que aparece la letra a en una cadena, y es otro ejemplo del patrón de conteo introducido en el [*Contando dígitos*](http://www.openbookproject.net/thinkcs/archive/python/spanish2e/cap06.html#contando):

fruta = "banana"

cuenta = 0

**for** carac **in** fruta:

**if** carac == 'a':

cuenta += 1

**print** cuenta

**7.10. Parámetros opcionales**

Para encontrar las posiciones de la segunda o tercera ocurrencia de un carácter en una cadena, podemos modificar la función encuentra, agregando un tercer parámetro para la posición inicial en la cadena de búsqueda:

**def** encuentra2(cadena, carac, inicio):

indice = inicio

**while** indice < len(cadena):

**if** cadena[indice] == carac:

**return** indice

indice += 1

**return** -1

La llamada a encuentra2('banana', 'a', 2) ahora devuelve 3, el índice de la segunda ‘a’ en ‘banana’.

Mejor aún, podemos combinar encuentra y encuentra2 usando un **parámetro opcional**:

**def** encuentra(cadena, carac, inicio=0):

indice = inicio

**while** indice < len(cadena):

**if** cadena[indice] == carac:

**return** indice

indice += 1

**return** -1

La llamada a encuentra('banana', 'a', 2) en esta versión de encuentra se comporta precisamente como encuentra2, mientras que, en la llamada a encuentra('banana','a'), el parámetro inicio se establecerá en el **valor por defecto**, esto es, 0.

Agregar otro parámetro opcional a encuentra hace que busque tanto hacia adelante como hacia atrás:

**def** encuentra(cadena, carac, inicio=0, paso=1):

indice = inicio

**while** 0 <= indice < len(cadena):

**if** cadena[indice] == carac:

**return** indice

indice += paso

**return** -1

Asignar un valor de -1 al paso hará que busque en dirección al inicio de la cadena en lugar de hacerlo hacia el final. Note que, para realizar este cambio, necesitamos verificar en el bucle while el límite inferior del índice así como el límite superior.

**7.11. El módulo string**

El módulo string contiene funciones útiles que manipulan cadenas. Como de costumbre, tenemos que importar el módulo antes de que podamos usarlo:

**>>> import** **string**

Para ver lo que hay dentro de él, use la función dir con el nombre del módulo como un argumento.

**>>>** dir(string)

lo que devolverá la lista de objetos dentro del módulo string:

['Template', '\_TemplateMetaclass', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_float', '\_idmap', '\_idmapL', '\_int', '\_long', '\_multimap', '\_re','ascii\_letters', 'ascii\_lowercase', 'ascii\_uppercase', 'atof', 'atof\_error', 'atoi', 'atoi\_error', 'atol', 'atol\_error', 'capitalize', 'capwords', 'center','count', 'digits', 'expandtabs', 'find', 'hexdigits', 'index', 'index\_error', 'join', 'joinfields', 'letters', 'ljust', 'lower', 'lowercase', 'lstrip','maketrans', 'octdigits', 'printable', 'punctuation', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip', 'split', 'splitfields', 'strip', 'swapcase','translate', 'upper', 'uppercase', 'whitespace', 'zfill']

Para averiguar más sobre un objeto de la lista, podemos usar el comando type. Necesitamos especificar el nombre del módulo seguido por el objeto usando la**notación punto**.

**>>>** type(string.digits)

<type 'str'>

**>>>** type(string.find)

<type 'function'>

Puesto que string.digits es una cadena, podemos mostrarla para ver qué contiene:

**>>> print** string.digits

0123456789

No es de extrañar, contiene cada uno de los dígitos decimales.

string.find es una función que hace más o menos lo mismo que la función que escribimos antes,``encuentra``. Para saber más sobre ella, podemos mostrar la**documentación del módulo string**, \_\_doc\_\_, que contiene la documentación de la función:

**>>> print** string.find.\_\_doc\_\_

find(s, sub [,start [,end]]) -> in

Return the lowest index in s where substring sub is found,

such that sub is contained within s[start,end]. Optional

arguments start and end are interpreted as in slice notation.

Return -1 on failure.

Los parámetros en corchetes cuadrados son opcionales. Podemos usar la función string.find de la misma forma que nuestra función encuentra:

**>>>** fruta = "banana"

**>>>** indice = string.find(fruta, "a")

**>>> print** indice

1

Este ejemplo demuestra uno de los beneficios de los módulos — ayudan a evitar colisiones entre los nombres de las funciones de biblioteca y las funciones definidas por el usuario. Mediante el uso de la notación punto podemos especificar cuál versión de find/encuentra queremos.

En realidad string.find es más general que nuestra versión; string.find puede encontrar subcadenas, no sólo caracteres:

**>>>** string.find("banana", "na")

2

Así como la nuestra, toma un argumento adicional que especifica el índice en el cual debería empezar:

**>>>** string.find("banana", "na", 3)

4

A diferencia de la nuestra, su segundo parámetro opcional especifica el índice en el cual la búsqueda debería terminar.

**>>>** string.find("bob", "b", 1, 2)

-1

En este ejemplo, la búsqueda falla porque la letra *b* no aparece en en el rango de índices de 1 a 2 (no se incluye 2).

**7.12. Clasificación de caracteres**

Con frecuencia es útil examinar un carácter y comprobar si está en mayúsculas o en minúsculas, o si es un carácter o un dígito. El módulo string proporciona varias constantes que son útiles para lograr estos objetivos. Una de ellas, que ya hemos visto, es `` string.digits``.

La cadena string.lowercase contiene todas las letras que el sistema considera como minúsculas. De forma similar, `` string.uppercase`` contiene todas las letras mayúsculas. Intente lo siguiente y vea que obtiene:

**print** string.lowercase

**print** string.uppercase

**print** string.digits

Podemos usar estas constantes y find para clasificar los caracteres. Por ejemplo, si find(lowercase, carac) devuelve un valor distinto de -1, entonces carac debe ser una letra minúscula:

**def** es\_minuscula(carac):

**return** string.find(string.lowercase, carac) != -1

Alternativamente, podemos aprovechar el operador in:

**def** es\_minuscula(carac):

**return** carac **in** string.lowercase

Como una alternativa más, podemos usar el operador de comparación:

**def** es\_minuscula(carac):

**return** 'a' <= carac <= 'z'

Si carac está entre *a* y *z*, debe ser una letra minúscula.

Otra constante definida en el módulo string puede sorprenderlo cuando la muestre:

**>>> print** string.whitespace

Los caracteres de **whitespace** mueven el cursor sin mostrar nada. Crean los espacios en blanco entre los caracteres visibles (al menos sobre el papel). La constantestring.whitespace contiene todos los caracteres de espacio en blanco, incluidos espacio, tabulador (\t), y salto de línea (\n).

Hay otras funciones útiles en el módulo string, pero este libro no pretende ser un manual de referencia. Por otra parte, la *Python Library Reference - Biblioteca de Referencia de Python* sí lo es, que, junto con un cúmulo de documentación adicional, está disponible en el sitio web de Python, *http://www.python.org*, (Documentación adicional en español, limitada, se puede encontrar *aquí <http://wiki.python.org/moin/Languages/Spanish?highlight=(CategoryLanguage)>*.

**7.13. Formateo de cadenas**

La forma más concisa y poderosa de dar formato a una cadena en Python es usar el *operador de formateo de cadenas*, %, junto con las operaciones de formateo de cadenas de Python. Para observar cómo funciona, empecemos con algunos ejemplos:

**>>>** "Su nombre es *%s*." % "Arturo"

'Su nombre es Arturo.'

**>>>** nombre = "Alicia"

**>>>** edad = 10

**>>>** "Soy *%s* y tengo *%d* años de edad." % (nombre, edad)

'Soy Alicia y tengo 10 años de edad.'

**>>>** n1 = 4

**>>>** n2 = 5

**>>>** "2\*\*10 = *%d* y *%d* \* *%d* = *%f*" % (2\*\*10, n1, n2, n1 \* n2)

'2\*\*10 = 1024 y 4 \* 5 = 20.000000'

>>>

La sintaxis para las operaciones de formateo de cadenas luce así:

"<FORMATO>" % (<VALORES>)

Ésta comienza con un *formato* que contiene una secuencia de caracteres y *especificaciones de conversión*. Las especificaciones de conversión inician con el operador %. Tras el formato de cadena está un solo % y después una secuencia de valores, *uno por especificación de conversión*, separados por comas y encerrados entre paréntesis. Los paréntesis son opcionales si hay solamente un valor.

En el primero de los anteriores ejemplos, hay una única especificación de conversión, %s, que indica una cadena. El único valor, "Arturo", le corresponde a la especificación de conversión, y no está entre paréntesis.

En el segundo ejemplo, nombre tiene el valor de cadena, "Alicia", y edad tiene el valor entero, 10. Estos corresponden a las especificaciones de conversión %s `` y ``%d. La d en la segunda especificación de conversión indica que el valor es un entero decimal.

En el tercer ejemplo las variables n1 y n2 tienen los valores enteros 4 y 5 respectivamente. Hay cuatro especificaciones de conversión en el formato de la cadena: tres con%d y una con %f. La f indica que el valor debe ser representado como un número de punto flotante. Los cuatro valores que corresponden a las cuatro especificaciones de conversión son: 2\*\*10, n1, n2, y n1 \* n2.

s, d, y f son todos los tipos de conversión que necesitaremos en este libro. Para ver la lista completa, vaya a la sección *String Formatting Operations <http://docs.python.org/lib/typesseq-strings.html>*, de la Python Library Reference, ó en español a la sección *Operaciones de formato de las cadenas <http://pyspanishdoc.sourceforge.net/lib/typesseq-strings.html>* de la Biblioteca de Referencia de Python.

El siguiente ejemplo ilustra la verdadera utilidad del formateo de cadenas:

i = 1

**print** "i**\t**i\*\*2**\t**i\*\*3**\t**i\*\*5**\t**i\*\*10**\t**i\*\*20"

**while** i <= 10:

**print** i, '**\t**', i\*\*2, '**\t**', i\*\*3, '**\t**', i\*\*5, '**\t**', i\*\*10, '**\t**', i\*\*20

i += 1

Este programa muestra una tabla de varias potencias de los números del 1 al 10. En su forma actual, la tabla depende del carácter de tabulación (\t) para alinear las columnas de valores, pero esto se distorsiona cuando los valores en la tabla son más largos que el ancho de 8 del carácter de tabulación:

i i\*\*2 i\*\*3 i\*\*5 i\*\*10 i\*\*20

1 1 1 1 1 1

2 4 8 32 1024 1048576

3 9 27 243 59049 3486784401

4 16 64 1024 1048576 1099511627776

5 25 125 3125 9765625 95367431640625

6 36 216 7776 60466176 3656158440062976

7 49 343 16807 282475249 79792266297612001

8 64 512 32768 1073741824 1152921504606846976

9 81 729 59049 3486784401 12157665459056928801

10 100 1000 100000 10000000000 100000000000000000000

Una posible solución sería cambiar el ancho del carácter de tabulación, pero la primera columna ya tiene más espacio del que necesita. La mejor solución es sería establecer el ancho de cada columna independientemente. Como probablemente ya ha adivinado, el formateo de cadenas proporciona la solución:

i = 1

**print** "*%-4s%-5s%-6s%-8s%-13s%-15s*" % \

('i', 'i\*\*2', 'i\*\*3', 'i\*\*5', 'i\*\*10', 'i\*\*20')

**while** i <= 10:

**print** "*%-4d%-5d%-6d%-8d%-13d%-15d*" % (i, i\*\*2, i\*\*3, i\*\*5, i\*\*10, i\*\*20)

i += 1

La ejecución de esta versión produce la siguiente salida:

i i\*\*2 i\*\*3 i\*\*5 i\*\*10 i\*\*20

1 1 1 1 1 1

2 4 8 32 1024 1048576

3 9 27 243 59049 3486784401

4 16 64 1024 1048576 1099511627776

5 25 125 3125 9765625 95367431640625

6 36 216 7776 60466176 3656158440062976

7 49 343 16807 282475249 79792266297612001

8 64 512 32768 1073741824 1152921504606846976

9 81 729 59049 3486784401 12157665459056928801

10 100 1000 100000 10000000000 100000000000000000000

El - después de cada % en la especificación de conversión indica justificación a la izquierda. Los valores numéricos especifican la longitud mínima, así que %-13d es un número justificado a la izquierda de por lo menos 13 caracteres de ancho.

**7.14. Glosario**

**tipo de datos compuesto:**

Un tipo de datos en el que los valores están hechos de componentes o elementos que son a su vez valores.

**índice:**

Una variable o valor usado para seleccionar un miembro de un conjunto ordenado, como puede ser un carácter de una cadena.

**recorrido:**

Iterar sobre los elementos de un conjunto, realizando una operación similar en cada uno de ellos.

**porción:**

Una parte de una cadena (subcadena) especificada mediante un rango de índices. Más ampliamente, una subsecuencia de cualquier tipo de secuencia en Python puede ser creada usando el operador porción (secuencia[inicio:fin]).

**inmutable:**

Tipos de datos compuestos a cuyos elementos no se les puede asignar nuevos valores.

**parámetro opcional:**

Parámetro escrito en un encabezado de una función con una asignación de un valor por defecto, el cual será recibido por la llamada a función si no se le da el correspondiente argumento.

**valor por defecto:**

Valor dado a un parámetro opcional si no se le da un argumento en la llamada a función.

**notación punto:**

Uso el **operador punto**, ., para acceder funciones dentro de un módulo.

**documentación del módulo string:**

Una cadena constante en la primera línea de una definición de función o módulo (y como veremos más adelante, también en las definiciones de clases y métodos). La documentación del módulo string proporciona una forma conveniente de asociar la documentación con el código. La documentación del módulo string es también usada por el módulo doctest para pruebas unitarias automatizadas.

**whitespace:**

Cualquiera de los caracteres que mueven el cursor sin mostrar caracteres visibles. La constante string.whitespace contiene todos los caracteres de espacio en blanco.