Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

9 Tipos secuencia

Índice de contenidos

Introducción

Este tema es la continuación del Tema 8, *Tipos colección*.

Aquí te voy a explicar las características principales de los tipos colección conocidos como *tipos secuencia*.

Los principales tipos secuencia que implementa Python son las *cadenas de caracteres*, las *listas*, las *tuplas*, el tipo *range* y los tipos *bytes* y *bytearray* (estos dos últimos los veremos en el Tema 12).

Tipos secuencia

Formalmente, un tipo secuencia es un iterable que admite el acceso eficiente a sus elementos mediante índices enteros a través del método especial `__getitem__()` y que define un método `__len__()` que devuelve la longitud de la secuencia.

Los principales tipos secuencia que incorpora el lenguaje son `list`, `tuple`, `range`, `bytes`, `bytearray` y `str` (sí, te dije que la cadena de caracteres era un tipo básico, pero en realidad es un tipo secuencia).

Aunque la clase o tipo *dict* también implementa los métodos `__getitem__()` y `__len__()`, es considerada un mapa por el modo en que se accede a los elementos (la veremos en detalle en el tema siguiente).

A continuación, te voy a mostrar las principales operaciones que se pueden llevar a cabo sobre los diferentes tipos *secuencia*. Después, describiré las particularidades de cada uno de ellos.

Operaciones comunes de los tipos secuencia

Estas operaciones están soportadas por todos los tipos secuencia, ya sean *mutables* o *inmutables*:

Pertenencia

Ya vimos que los operadores de pertenencia eran `in` y `not in`. Estos operadores se pueden utilizar sobre secuencias para comprobar si un elemento está contenido, o no, en la secuencia:

```
>>> l = [1, 2, 3, 4, 5]
>>> 6 in l
False
```

Normalmente `in` y `not in` comprueban la existencia de un único elemento. Sin embargo, en algunos tipos como `str` o `bytes` se puede comprobar la existencia de una subsecuencia:

```
>>> saludo = 'Hola'
>>> 'ol' in saludo
True
```

Concatenación

Para concatenar dos o más secuencias se utiliza el operador de concatenación, `+`.

El resultado de esta operación es siempre un objeto nuevo.

```
>>> l1 = [1, 2]
>>> l2 = [3, 4]
>>> l = l1 + l2
>>> l
[1, 2, 3, 4]
```

Repetición

Esta operación consiste en repetir el contenido de la secuencia `n` veces. Se utiliza el operador `*`:

```
>>> saludo = 'hola'
>>> saludo * 3
'holaholahola'
```

Acceso a un elemento

Tal y como te he dicho anteriormente, una de las principales operaciones que se puede realizar sobre un tipo secuencia es acceder a sus elementos a través de un índice numérico por medio del método `__getitem__(índice)`. En realidad, este método se

traduce al operador `[índice]`. Básicamente, un índice es un número entero entre corchetes `[]` que indica la posición de un elemento en la secuencia. El primer elemento de la secuencia siempre tiene como índice 0.

```
>>> s = 'Pythonista'
>>> s[1]
'y'
>>> s[5]
'n'
```

Si se intenta acceder a un índice que está fuera del rango de la secuencia, el intérprete lanzará la excepción `IndexError`. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción `TypeError`.

Los índices también pueden ser negativos. En este caso, el índice -1 hace referencia al último elemento, -2 al penúltimo, y así, sucesivamente.

```
>>> s = 'Pythonista'
>>> s[-1]
'a'
>>> s[-3]
's'
```

Acceso a un rango de elementos

Además de acceder a un único elemento mediante un *índice*, también es posible acceder a un rango de elementos indicando el índice inferior y superior separados con dos puntos y entre corchetes (el elemento del índice superior nunca se incluye):

```
>>> s = 'Pythonista'
>>> s[2:5]
'tho'
```

De igual modo, los índices pueden ser negativos:

```
>>> s = 'Pythonista'
>>> s[-3: -1]
'st'
```

IMPORTANTE La subsecuencia devuelta es siempre un nuevo objeto.

Si cualquiera de los índices es mayor que la longitud de la secuencia entonces tomará este valor. Si no se especifica el índice inferior o es `None`, entonces tomará por defecto el valor 0. Si no se especifica el índice superior o es `None`, entonces tomará como valor la longitud de la secuencia. Si el índice inferior es mayor que el índice superior, el resultado será una secuencia vacía.

Cuando se accede de esta forma a los elementos, también es posible indicar el paso con el que se

incrementa (o decrementa) el valor de los índices que, si no se especifica, por defecto es 1.

```
>>> s = ‘Pythonista’
>>>s[2:8:2]
'toi'
```

Con paso negativo, se invierte el orden de la secuencia:

```
>>> s = 'Pythonista'
>>> s[8:2:-2]
'tio'
```

Obtener la longitud de una secuencia

Para obtener la longitud de una secuencia se debe utilizar la función de Python `len()`. A esta función se le pasa como argumento la secuencia en cuestión y devuelve el número de elementos que contiene:

```
>>> saludo = 'Hola'
>>> len(saludo)
4

>>> l = [1, 2]
>>> len(l)
2
```

Contar las ocurrencias de un elemento en la secuencia

Para contar el número de veces que aparece un elemento en la secuencia se utiliza el método `count()`:

```
>>> lista = [1, 2]
>>> lista.count(1)
1
>>> lista.count(3)
0
```

Encontrar el índice de la primera ocurrencia de un elemento

El método `index(elemento)` devuelve el índice de la primera ocurrencia del `elemento` en la secuencia si existe. En caso de no aparecer, se lanza la excepción `ValueError`.

```
>>> l = [1, 2]
>>> l.index(2)
1
>>> l.index(3)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: 3 is not in list
```

También se puede indicar, de forma opcional, los índices entre los que buscar:

```
>>> s = 'Pythonista'
>>> s.index('o', 2)
4
>>> s.index('y', 4, 8)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: substring not found
```

Operaciones de los tipos secuencia mutables

En la siguiente tabla aparece un resumen de las principales operaciones de los tipos secuencia mutables (`list` y `bytearray`, entre otros). En la tabla, `s` es un objeto de tipo secuencia mutable, `t` es un objeto iterable y `e` es un elemento. `i`, `j` y `k` hacen referencia a índices.

Operación	Descripción
`s[i] = e`	Reemplaza el elemento de la posición `i` por `e`.
`s[i:j] = t`	La subsecuencia de `i` a `j` es reemplazada por el contenido de `t`.
`del s[i]`	Elimina el elemento de la posición `i`.
`del s[i:j]`	Elimina los elementos de la subsecuencia comprendida entre `i` y `j`.
`s[i:j:k] = t`	La subsecuencia `s[i:j:k]` es reemplazada por el contenido de `t`.
`del s[i:j:k]`	Elimina los elementos de la subsecuencia `s[i:j:k]`.
`s.append(e)`	Añade el elemento `e` al final de la secuencia.
`s.extend(t)` o `s += t`	Añade los elementos de `t` al final de la secuencia.
`s *= n`	Actualiza la secuencia con su contenido repetido `n` veces.
`s.insert(i, e)`	Inserta el elemento `e` en la posición `i` de la secuencia. Los elementos cuyo índice sea mayor o igual a `i` se desplazan una posición a la derecha.

`s.remove(e)`	Elimina la primera ocurrencia del elemento `e` en la secuencia. Si el elemento no se encuentra, se lanza la excepción `ValueError`.
`s.pop(i)`	Obtiene y elimina el elemento de la secuencia en la posición `i`. Si no se especifica `i`, obtiene y elimina el último elemento.
`s.clear()`	Borra todos los elementos de la secuencia. Equivale a `del s[:]`.
`s.reverse()`	Modifica la secuencia estableciendo los elementos en orden inverso.
`s.copy()`	Devuelve una copia poco profunda de la secuencia (veremos qué significa esto en el tema de programación orientada a objetos).

IMPORTANTE:La mayoría de operaciones sobre tipos secuencia son del orden O(1).

Comparación de objetos de tipo secuencia

Las secuencias del mismo tipo también se pueden comparar. En particular, las tuplas y las listas se comparan lexicográficamente comparando sus elementos uno a uno. Esto significa que, para comparar igual, cada elemento debe comparar igual y las dos

secuencias deben ser del mismo tipo y tener la misma longitud.

Los objetos tipo secuencia, a excepción del tipo `range`, también se pueden comparar en orden, siguiendo las mismas reglas que te indicaba en el párrafo anterior. En el caso de tener distinto tamaño, la secuencia más pequeña es menor. Esto quiere decir que:

```
>>> l1 = [1, 2]
>>> l2 = [1, 1]
>>> l3 = [1, 2, 3]
>>> l2 < l1
True
>>> l1 < l3
True
>>> l2 < l3
True
```

Y hasta aquí llega la explicación de las características comunes de los tipos secuencia. En las siguientes secciones te explicaré en detalle cada uno de ellos.

Tipo str

El tipo o clase `str` en Python se utiliza para representar texto, más conocido en el mundo de la programación como *string* o *cadena de caracteres*. Ya vimos este tipo en el Tema 4, *Tipos de datos básicos*.

Poniéndome un poco más técnico, el tipo `str` es una **secuencia inmutable de caracteres Unicode**. Por tanto, un objeto de este tipo no se puede modificar después de haber sido creado.

IMPORTANTE: Cuidado cuando trabajes con texto procedente de ficheros u otras fuentes de datos. Fíjate en qué codificación está y haz las transformaciones necesarias si no quieres tener problemas. Por defecto, la codificación de un string en Python es Unicode, concretamente UTF-8.

Cómo crear una cadena en Python

Crear una cadena de texto en Python es muy sencillo. Simplemente encierra una secuencia de caracteres entre comillas simples `' '` o dobles `" "`.

```
>>> s = 'Hola Pythonista'
>>> s
'Hola Pythonista'
>>> type(s)
<class 'str'>
```

Si quieres o necesitas que un string ocupe más de una línea, entonces debes encerrar el texto entre tres comillas simples `'''...'''` o dobles `"""..."""`.

```
>>> s = '''
... Este string
... ocupa más
... de
... una línea'''
>>> s
'\nEste string\n ocupa más\n de\n una
línea'

>>> print(s)
Este string
ocupa más
de
una línea
```

Como puedes observar, el uso de las tres comillas (simples o dobles) guarda el carácter de fin de línea. Esto se puede evitar añadiendo el carácter `\` al final de cada línea.

```
>>> s = '''Este string \
... ocupa más \
... de \
... una línea'''
>>> s
'Este string ocupa más de una línea'
>>> print(s)
Este string ocupa más de una línea
```

No confundas un string multilínea con un *docstring*. Un *docstring* es un string multilínea que se utiliza para documentar un módulo, función, clase o método, entre otros.

Al ser un tipo secuencia, dos o más cadenas se pueden concatenar utilizando los operadores `+` y `*`. Además, dos strings literales también se pueden concatenar si aparecen juntos uno tras otro (esto es útil si se quiere crear un string de más de una línea):

```
>>> s = 'Hola' ' Pythonista'
>>> s
'Hola Pythonista'
>>> s = ('Hola'
... ' Pythonista'
... ' ¿Te gusta Python?')
>>> s
'Hola Pythonista ¿Te gusta Python?'
```

Bucle for y str

Al ser un iterable, es posible recorrer todos los caracteres de un objeto de tipo `str` usando la sentencia `for`. Para ello, lo más fácil es seguir la siguiente plantilla:

```
>>> saludo = 'Hola'
>>> for c in saludo:
...     print(c)
...
H
o
l
a
```

Caracteres especiales

Como un string está limitado por los caracteres `''` o `""`, ¿qué ocurre si necesito usar el carácter `'` o `"` dentro de una cadena?

Lo más fácil si tienes que usar el carácter `'` en tu cadena, es encerarla entre comillas dobles. Por el contrario, si necesitas usar `"` dentro del string, enciérralo entre comillas simples.

```
>>> s = 'Dijo: "Hola Pythonista"'
>>> print(s)
Dijo: "Hola Pythonista"
>>> s = "Dijo: 'Hola Pythonista'"
>>> print(s)
Dijo: 'Hola Pythonista'
```

También puedes usar la combinación `\'` para mostrar una comilla simple o `\"` para mostrar una comilla doble, independientemente de si la cadena está encerrada entre comillas simples o dobles.

```
>>> s = 'Dijo: \'Hola Pythonista\''
>>> print(s)
Dijo: 'Hola Pythonista'
>>> s = "Dijo: \"Hola Pythonista\""
>>> print(s)
Dijo: "Hola Pythonista"
```

Además del carácter `'` y `"`, hay otros caracteres especiales que para ser usados dentro de una cadena necesitan ser «escapados» con el carácter `\`. Son, entre otros, los siguientes: tabulador `\t`, barra invertida `\\`, retroceso `\b`, nueva línea `\n` o retorno de carro `\r`.

```
# Ejemplo para declarar una
# ruta en Windows
>>> s = 'C:\\Users\\Documents\\'
>>> print(s)
C:\Users\Documents\

# Nueva línea más tabulador
>>> s = 'Hola\n\tPythonista'
>>> print(s)
Hola
  Pythonista
```

Raw strings

En relación con la sección anterior, puede haber ocasiones en que se quiera usar el carácter `\` pero sin ser utilizado como carácter de escape. Para ello, se puede hacer uso de las *raw strings*. Una cadena de este tipo comienza anteponiendo el carácter `r` a las comillas (simples o dobles).

```
# Aquí, \n es interpretado
# como nueva línea
>>> s = 'C:\python\noticias'
>>> print(s)
C:\python
oticias

# En una raw string no se
# interpreta el carácter \
>>> s = r'C:\python\noticias'
>>> print(s)
C:\python\noticias
```

Representación de un objeto como cadena

Una singularidad de la clase `str` es que a su constructor se le puede pasar cualquier objeto. Al hacer esto, la función `str()` devuelve la representación en forma de

cadena de caracteres del propio objeto (si se pasa un string devuelve el string en sí).

Normalmente, al llamar a la función `str(objeto)` lo que se hace internamente es llamar al método `__str__()` del objeto. Si este método no existe, entonces devuelve el resultado de invocar a `repr(objeto)`. Veremos estos métodos en el tema de programación orientada a objetos.

Operaciones y métodos de la clase str

Al tratarse de un tipo secuencia, se pueden aplicar todas las operaciones comunes que hemos visto para este tipo, como acceso a los elementos a través de un índice u obtener la longitud de la cadena.

No obstante, la clase `str` define una serie de métodos propios que resultan muy útiles a la hora de trabajar con objetos de este tipo.

En la tabla siguiente te muestro un resumen de algunos de los métodos más utilizados (Hay muchos más. Revisa la documentación para consultar todos):

Método	Descripción
`str.capitalize()`	Devuelve una copia de la cadena con su primer carácter en mayúscula y el resto en minúscula.
`str.center (width [, fillchar])`	Devuelve una cadena centrada de ancho `width`. El relleno se realiza utilizando el `fillchar` especificado (el valor por defecto es un espacio ASCII). La cadena original se devuelve si el ancho es menor o igual que `len(str)`.
`str.encode(encoding="utf-8", errors="strict")`	Devuelve una versión codificada de la cadena como un objeto de tipo `bytes`.
`str.endswith(suffix[, start[, end]])`	Devuelve `True` si la cadena termina con el sufijo especificado; de lo contrario, devuelve `False`. El sufijo también puede ser una tupla de sufijos para buscar. Si se especifica el argumento opcional `start`, la comparación comienza desde esa posición. Si además se especifica el argumento opcional `end`, la comparación finaliza en esa posición.
`str.isalnum()`	Devuelve `True` si todos los caracteres de la cadena son alfanuméricos y hay al menos un carácter, `False` en caso contrario.
`str.isdigit()`	Devuelve `True` si todos los caracteres en la cadena son dígitos

	y hay al menos un carácter, `False` en caso contrario.
`str.islower()`	Devuelve `True` si todos los caracteres en la cadena están en minúscula y hay al menos un carácter que permite mayúsculas; de lo contrario, `False`.
`str.lower()`	Devuelva una copia de la cadena con todos los caracteres convertidos a minúsculas.
`str.replace(old, new[, count])`	Devuelve una copia de la cadena con todas las apariciones de la subcadena `old` reemplazada por `new`. Si se proporciona el argumento opcional `count`, solo se reemplazan las primeras `count` ocurrencias.
`str.split(sep=None, maxsplit=-1)`	Devuelve una lista de las palabras en la cadena, usando `sep` como la cadena delimitadora. Si se indica `maxsplit`, como máximo se realizan `maxsplit` divisiones (por lo tanto, la lista tendrá, como mucho, `maxsplit` + 1 elementos)
`str.startswith(prefix[, start[, end]])`	Devuelve `True` si la cadena comienza con el prefijo `prefix`; de lo contrario, devuelve `False`. El prefijo también puede ser una tupla de prefijos para buscar. Si se especifica el argumento opcional `start`, la comparación comienza desde esa posición. Si además se

	especifica el argumento opcional `end`, la comparación finaliza en esa posición.
`str.strip([chars])`	Devuelve una copia de la cadena con los caracteres iniciales y finales eliminados. El argumento `chars` es una cadena que especifica el conjunto de caracteres que se eliminarán. Si se omite o es `None`, por defecto eliminan los caracteres en blanco (se incluyen aquí el espacio, \t, \n, \r, ...). El argumento `chars` no es un prefijo o sufijo; más bien, se eliminan todas las combinaciones de sus valores:
`str.upper()`	Devuelva una copia de la cadena con todos los caracteres convertidos a mayúsculas.

Tipo list

Las listas son un tipo contenedor que se usan para almacenar conjuntos de elementos relacionados, normalmente del mismo tipo, aunque también pueden albergar elementos de tipos distintos.

Formalmente, `list` **es un tipo secuencia mutable**, por lo que es posible añadir, modificar y/o eliminar elementos de una lista.

Las listas son uno de los tipos más importantes de Python y más versátiles, dado que tienen infinidad de usos:

- Cualquier listado de objetos/valores/elementos que se necesite alterar, se puede implementar con una lista.
- Una pila se puede implementar con una lista.

Son parecidas a los arrays de *C* o a las listas de *Java*, pero mucho más potentes y dinámicas.

Crear una lista

Ya hemos visto que, para crear una lista en Python, simplemente hay que encerrar una secuencia de elementos separados por comas entre corchetes `[]`.

Por ejemplo, para crear una lista con los números del 1 al 10 se haría del siguiente modo:

```
>>> numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Como te decía, las listas pueden almacenar elementos de distinto tipo. La siguiente lista también es válida:

```
>>> elementos = [3, 'a', 8, 7.2, 'hola']
```

Incluso pueden contener otros elementos compuestos, como objetos u otras listas:

```
>>> lista = [1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola']
```

Las listas también se pueden crear usando el constructor de la clase, `list(iterable)`. En este caso,

el constructor crea una lista cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto iterable puede ser o una secuencia, un contenedor que soporte la iteración o un objeto de tipo `iterator`.

Por ejemplo, el tipo `str` es un tipo secuencia. Si pasamos un string como argumento del constructor `list()`, el resultado será una lista formada por cada uno de los caracteres de la cadena:

```
>>> vocales = list('aeiou')
>>> vocales
['a', 'e', 'i', 'o', 'u']
```

Termino esta sección mostrando dos alternativas para crear una lista vacía:

```
>>> lista_1 = []  # Opción 1
>>> lista_2 = list()  # Opción 2
```

Operaciones y métodos de la clase list

Al tratarse de un tipo secuencia, se pueden aplicar todas las operaciones comunes que hemos visto para este tipo, como acceso a los elementos a través de un índice u obtener la longitud de la lista.

Además, por ser un tipo de secuencia *mutable,* también son válidas las operaciones definidas para estos tipos.

Por ejemplo:

```
# Añadir un elemento a una lista
>>> l = [1, 2]
>>> l.append(3)
>>> l
[1, 2, 3]

# Obtener y eliminar el último elemento
>>> l.pop()
3
>>> l
[1, 2]
```

Bucle for y list

Ya hemos visto que se puede usar el bucle `for` en Python para recorrer los elementos de una secuencia. En este caso, para recorrer una lista, utilizaríamos la siguiente estructura:

```
>>> colores = ['azul', 'blanco', 'negro']
>>> for color in colores:
        print(color)
azul
blanco
negro
```

IMPORTANTE: Cuidado con modificar o alterar una lista mientras se está iterando sobre la misma (por ejemplo, eliminando elementos). El resultado puede ser impreciso. Una recomendación es iterar sobre una copia de la lista o crear una nueva.

Lo mismo se aplica para el resto de tipos colección mutables.

Ordenar una lista

Las listas son secuencias ordenadas. Esto quiere decir que sus elementos siempre se devuelven en el mismo orden en que fueron añadidos.

No obstante, es posible ordenar los elementos de una lista con el método `sort()`. El método `sort()` ordena los elementos de la lista utilizando únicamente el operador `<` y modificando la lista actual (no se obtiene una nueva lista):

```
# Lista desordenada de números enteros
>>> numeros = [3, 2, 6, 1, 7, 4]
# Identidad del objeto numeros
>>> id(numeros)
4475439216

# Se llama al método sort() para ordenar
# los elementos de la lista
>>> numeros.sort()
>>> numeros
[1, 2, 3, 4, 6, 7]

# Se comprueba que la identidad del
# objeto numeros es la misma
>>> id(numeros)
4475439216
```

Listas anidadas

Como te indiqué en la introducción, las listas pueden contener todo tipo de objetos, incluido otras listas. En este caso, se puede acceder a los elementos de estos tipos usando índices compuestos o anidados:

```
>>> lista = ['a', ['d', 'b'], 'z']
>>> lista[1][1]
'b'
```

Crear una copia de una lista

Para crear una copia de una lista puedes usar el método `copy()` o devolver el rango completo de elementos. En ambos casos se obtiene una *copia poco profunda* de la lista:

```
>>> l = [1, 2, 3]
>>> l2 = l.copy()
>>> l2
[1, 2, 3]
>>> l3 = l[:]
>>> l3
[1, 2, 3]
```

Tipo tuple

El tipo `tuple` es un tipo contenedor que, en un principio, está pensado para almacenar grupos de elementos heterogéneos, aunque también puede contener elementos homogéneos.

Formalmente, es **un tipo secuencia *inmutable***. Esto último quiere decir que su contenido no se puede modificar después de haber sido creada.

Crear una tupla

En general, para crear una tupla simplemente hay que definir una secuencia de elementos separados por comas.

Por ejemplo, para crear una tupla con los números del 1 al 5 se haría del siguiente modo:

```
>>> numeros = 1, 2, 3, 4, 5
```

Como te indicaba, la clase `tuple` también puede almacenar elementos de distinto tipo:

```
>>> elementos = 3, 'a', 8, 7.2, 'hola'
```

Incluso puede contener otros elementos compuestos y objetos, como listas, otras tuplas, etc.:

```
>>> tup = 1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola'
```

A continuación, te indico las diferentes formas que existen de crear una tupla en Python:

- Para crear una tupla vacía, usa paréntesis `()` o el constructor de la clase `tuple()` sin argumentos.
- Para crear una tupla con un único elemento: `elem,` o `(elem, )`. Observa que siempre se añade una coma al final cuando la tupla está formada por un solo elemento.
- Para crear una tupla de varios elementos, sepáralos con comas: `a, b, c` o `(a, b, c)`.
- Las tuplas también se pueden crear usando el constructor de la clase, `tuple(iterable)`. En este caso, el constructor crea una tupla cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto iterable

puede ser una secuencia, un contenedor que soporte la iteración o un objeto `iterator`.

IMPORTANTE: El hecho que determina que una secuencia de elementos sea una tupla es la coma `,` no los paréntesis. Los paréntesis son opcionales y solo se necesitan para crear una tupla vacía o para evitar ambigüedades.

```
# Aquí, a, b y c no son una tupla,
# sino tres argumentos con los que
# se llama a la función "una_funcion"
>>> una_funcion(a, b, c)

# Aquí, a, b y c son tres elementos
# de una tupla. Esta tupla, es el
# único argumento con el que se invoca
# a la función "una_funcion"
>>> una_funcion((a, b, c))
```

Operaciones y métodos de la clase tuple

Al tratarse de un tipo secuencia, se pueden aplicar todas las operaciones comunes que hemos visto para este tipo, como acceso a los elementos a través de un índice u obtener la longitud de la tupla.

Tuple packing y unpacking

El modo de crear una tupla sin paréntesis se conoce como *tuple packing* (algo así como empaquetado de tuplas).

Relacionado con este último, vamos a ver el concepto llamado *tuple unpacking* (desempaquetado de una tupla). Realmente, el *unpacking* se puede aplicar sobre cualquier objeto de tipo secuencia, aunque se usa mayoritariamente con las tuplas, y consiste en lo siguiente:

```
>>> bebidas = 'agua', 'café', 'batido'
>>> a, b, c = bebidas
>>> a
'agua'
>>> b
'café'
>>> c
'batido'
```

Como puedes apreciar, es un tipo de asignación múltiple. Requiere que haya tantas variables a la izquierda del operador de asignación `=` como elementos haya en la secuencia.

Bucle for y tuple

Al igual que para los tipos `str` y `list`, para recorrer una tupla en Python utiliza la siguiente estructura:

```
>>> colores = 'azul', 'blanco', 'negro'
>>> for color in colores:
        print(color)
azul
blanco
negro
```

Modificar una tupla en Python

El contenido de una tupla, al ser esta inmutable, no se puede alterar. No obstante, las tuplas pueden contener objetos u otros elementos de tipo secuencia, por ejemplo una lista, que sí se pueden modificar:

```
>>> tupla = (1, ['a', 'b'], 'hola', 8.2)
# tupla[1] hace referencia a la lista
>>> tupla[1].append('c')
>>> tupla
(1, ['a', 'b', 'c'], 'hola', 8.2)
```

Tipo range

`range` es un tipo de dato que representa una **secuencia de números inmutable** y se suele utilizar para iterar un número determinado de veces en un bucle `for`.

Para crear un objeto de tipo `range`, se pueden usar dos constructores:

- `range(fin)`: Crea una secuencia numérica que va desde 0 hasta `fin` - 1.
- `range(inicio, fin [, paso])`: Crea una secuencia numérica que va desde inicio hasta `fin` - 1. Si además se indica el argumento `paso`, la secuencia genera los números de `paso` en `paso`.

Veámoslo con un ejemplo:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(5, -5, -2))
[5, 3, 1, -1, -3]
```

¿Por qué estoy utilizando en los ejemplos el objeto `range` como argumento de `list`? Básicamente, para que se muestre por pantalla la secuencia completa de números que se genera con `range`. Si no lo hiciera, se mostraría algo como `range(10)`.

Si se llama al constructor `range()` con parámetros incorrectos, se obtendrá un objeto vacío.

```
# Nunca se puede ir de 0 a 10 de -1 en -1
>>> list(range(0, 10, -1))
[]
```

Bucle for y range

Como te he dicho, uno de los principales usos de `range()` es utilizarlo en bucles `for`. A continuación, te indico cómo usarlo:

```
>>> for i in range(1, 6):
...     print(i)
...
1
2
3
4
5
```

Ventajas de usar range

La principal ventaja de usar `range` sobre `list` o `tuple` es que es un *iterable* que genera los elementos solo en el momento en que realmente los necesita. Esto implica que usa una cantidad de memoria mínima, por muy grande que sea el rango de números que represente.
Veamos una comparación de una lista que almacena los números del 0 al 100.000 y un rango del 0 al 100.000:

```
>>> import sys
>>> lista = list(range(0, 100000))
>>> rango = range(0, 100000)
>>> sys.getsizeof(lista)
900120
>>> sys.getsizeof(rango)
48
```

Como puedes apreciar, la lista ocupa casi 1 MB en memoria frente a los 48 bytes que ocupa el rango.

Operaciones y métodos de la clase range

Al tratarse de un tipo secuencia, se pueden aplicar todas las operaciones comunes que hemos visto para este tipo **a excepción de la concatenación y la repetición**.

También se puede acceder con índices negativos a los elementos del rango, pero estos se interpretan como los índices desde el final de la secuencia determinada por los índices positivos.

```
>>> r = range(1, 11)
>>> r[-1]
10
```

Comparación de objetos de tipo range

Dos objetos de tipo `range` se pueden comparar si son iguales o distintos. Se consideran iguales si ambos representan la misma secuencia de números.

Por ejemplo:

```
>>> range(1, 5, 3) == range(1, 6, 3)
True
```

Otras estructuras de datos: Pilas y Colas

En esta sección vamos a ver dos estructuras de datos muy usadas en programación y cómo estas se implementan en Python. Son las conocidas *Pilas* y *Colas*.

Pilas

Las pilas son estructuras de datos que tienen dos operaciones básicas: *push* (añade un elemento a la pila) y *pop* (extrae un elemento de la pila).

Su característica fundamental es que los elementos se añaden siempre al final y al extraer se obtiene siempre el último elemento. Por esta razón también se conocen como estructuras de datos *LIFO* (del inglés, *Last In First Out*).

En Python es muy fácil implementar una pila utilizando para ello la clase `list` y únicamente dos de sus métodos: `append(e)` para añadir elementos al final de la lista y `pop()` para obtener y eliminar el último elemento de la lista.

```
>>> l = []
>>> l.append(1)
>>> l.append(5)
>>> l.append(9)
>>> l
[1, 5, 9]

>>> l.pop()
9
>>> l
[1, 5]

>>> l.pop()
5
>>> l
[1]
```

Colas

Por su parte, las colas también tienen dos operaciones básicas: añadir un elemento a la cola y sacar un elemento de la cola.

La principal diferencia con respecto a las pilas es que el primer elemento en entrar es también el primero en salir. Por este motivo se les conoce como estructuras de datos *FIFO* (del inglés, *First In First Out*).

Aunque es posible implementar una cola en Python usando listas, insertar al comienzo o hacer un `pop()` del primer elemento son operaciones algo costosas si la cola contiene muchos elementos (todos los elementos se deben mover una posición). Por tanto, el lenguaje nos ofrece una nueva clase llamada `deque` para este propósito.

La clase `deque` está implementada en el módulo `collections` e implementa operaciones muy eficientes (del orden *O(1)*) para añadir y sacar elementos por ambos lados del objeto (esto hace que también sea válida para implementar pilas).

El constructor de la clase puede recibir como argumento un *iterable*, cuyos elementos se utilizarán para inicializar el objeto. Además, se puede indicar una longitud máxima. En este caso, cuando el objeto está completo, al añadir un nuevo elemento por un lado del objeto se sacará el primer elemento del lado contrario.

Volviendo a las colas, veamos un ejemplo de cómo implementarlas en Python con la clase `deque`:

```
>>> from collections import deque
>>> cola = deque([1, 2, 3, 4])
>>> cola
deque([1, 2, 3, 4])

# Añade elementos al final
>>> cola.append(5)
>>> cola.append(6)
>>> cola
deque([1, 2, 3, 4, 5, 6])

# Saca elementos del principio
>>> cola.popleft()
1
>>> cola.popleft()
2

>>> cola
deque([3, 4, 5, 6])
```