Consejos, trucos y curiosidades de Python

Publicado el El Informatico - 25 de agosto de 2022 -

Logotipo de Python

En éste artículo se enumeran algunos consejos, trucos y curiosidades de Python que podrías haber pasado. Todo lo que se menciona en éste artículo está relacionado con las últimas versiones de Python (3.9 en adelante). Si programas en Python, no te pierdas éste artículo.

1. List Slicing

Algo básico que quizás a alguien se le ha podido pasar por alto (aunque está presente en todos los tutoriales básicos) es que las listas (o lo que llamamos «array» en otros lenguajes) se pueden cortar indicando el principio y el final del corte. Por ejemplo:

```
a = ["Uno", "Dos", "Tres", "Cuatro"]
print(a[1:3])
```

Este código genera el siguiente resultado:

```
['Dos', 'Tres']
```

El primer índice es el inicio del corte y el segundo representa el final, con índice 1 en lugar de 0. De modo que 1:3 significa «Corta del elemento 1 al tercer elemento de la lista».

Podemos también cortar desde el inicio hasta un elemento concreto:

```
a = ["Uno", "Dos", "Tres", "Cuatro"]
print(a[1:3])
```

```
['Uno', 'Dos', 'Tres']
```

O incluso cortar contando desde el final usando índices negativos:

```
a = ["Uno", "Dos", "Tres", "Cuatro"]
print(a[-2:]) # Desde el segundo elemento contando desde el final hasta
el final
print(a[:-2]) # desde el principio hasta el segundo elemento contando
hasta el final
```

```
['Tres', 'Cuatro']
['Uno', 'Dos']
```

Algo que es importante comprender es que el array que se devuelve es siempre **una copia del original, y no una referencia**:

```
a = ["Uno", "Dos", "Tres", "Cuatro"]
b = a[1:3]
b.append("Cinco")
print(a)
print(b)
```

```
['Uno', 'Dos', 'Tres', 'Cuatro']
['Dos', 'Tres', 'Cinco']
```

Este método para cortar listas se puede usar tanto con *listas* con elementos de todo tipo (como, por ejemplo, objetos de tipo *bytes*) como con ***tuples***.

2. Crear listas con iteradores

Podemos crear listas dinámicas usando iteradores. Por ejemplo:

```
numeros = [i * 2 for i in range(0,10)]
print(numeros)
```

En éste código, generamos una lísta con valores del 0 al 9 multiplicados por dos, generando el siguiente resultado:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

3. F-Strings y formatos de texto

Hasta ahora (y si quieres compatibilidad con algunas versiones más antíguas de python como la 3.5), sólo había tres formas de dar formato a un texto. La primera es símplemente concatenando valores:

```
a = 1
texto = "Hola mundo " + str(a) + "!"
print(texto)
```

```
Hola mundo 1!
```

La segunda es usando la función *format()* de la clase string para generar una cadena de texto concatenando valores:

```
a = 2
texto = "Hola mundo {valor:d}!"
print(texto.format(valor = a))
```

```
Hola mundo 2!
```

La tercera es usando el operador '%' en el string, lo que nos permite darle un formato muy similar a la función *printf* de C:

```
a = 3
texto = "Hola mundo %i!" % (a)
print(texto)
```

```
Hola mundo 3!
```

A partir de Python 3.6, es posible usar lo que se denominan *f-strings*. Los f-string funcionan de manera muy similar a la función *format*, pero en lugar de pasar valores le pasamos directamente las variables al texto y el interprete ya se encarga de construir la cadena de texto:

```
a = 4
texto = f"Hola mundo {a:d}!"
print(texto)
```

```
Hola mundo 4!
```

Podeis ver una hoja de referencia con los diferentes códigos de formato en éste enlace.

4. Operador ternario

Python no tiene un operador ternario, pero igualmente se puede usar la clausula 'if-else' en una única línea para indicar un valor condicional. Por ejemplo:

```
a = 5
print(True if a == 5 else False)
a = 4
print(True if a == 5 else False)
```

```
True
False
```

5. Clausula Switch (match)

Los lenguajes basados en C disponen una clausula condicional '*switch*':

```
switch(valor){
   case 1:
    //Hacer algo
   break;
   case 2:
    //Otra cosa
   break;
   default:
    //Acción por defecto
}
```

El equivalente en python es la clausula ***match***, que funciona de manera similar. Por ejemplo:

```
def respuesta(valor):
    match valor:
        case "UNO":
            return "1"
        case "DOS":
            return "2"
        case other: # Acción por defecto
            return "Desconocido"

print(f"{respuesta('UNO')}")
print(f"{respuesta('DOS')}")
print(f"{respuesta('CINCO')}")
```

```
1

2

Desconocido
```

6. Especificar los tipos de las variables y los parámetros de las funciones, y sus valores por defecto

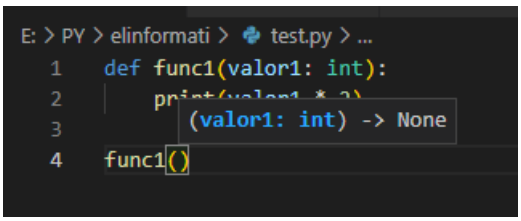
Python es un lenguaje muy flexible y dinámico que permite usar variables y parámetros sin necesidad de declarar especificamente el tipo de valor. De hecho, una variable puede almacenar cualquier tipo de valor en cualquier momento. De éste modo, podemos especificar parametros sin necesidad de especificar su tipo:

```
def func1(valor1):
    print(valor1 * 2)

func1(5)
```

Aunque no es necesario, en ocasiones es buena práctica determinar los tipos de valores con los que vamos a trabajar dentro de la función. Esto tiene dos funciones: por un lado documentamos el tipo de valores que acepta nuestra función, y por otro existen editores como VS Code que pueden mostrarnos el tipo de valor requerido en la sugerencia al escribir el código. Por ejemplo:

```
def func1(valor1: int):
    print(valor1 * 2)

func1(5)
```

VS Code reconoce éste valor como un valor íntegro:



De igual modo, podemos determinar el valor de retorno de la función:

```
def func1(valor1: int) -> int:
    return valor1 * 5

print(func1(5))
```

Esto también funciona para las variables que declaremos:

```
a: int or str = 5
print(a)
```

En este caso, indicamos que la variable a puede contener un número integro o una cadena de texto. Nada de ésto cambia la forma en la que se ejecuta el código (seguimos podiendo establecer cualquier valor sin que nada nos detenga en principio), pero puede ayudarte o ayudar a otras personas a entender mejor tu código en el futuro.

Es también posible dar un valor predeterminado a los parametros de una función, pero **éstos parametros deben estar siempre en el lado derecho de la lista de parametros**. Es decir, no podemos mezclar parametros con y sin valor por defecto:

```
def func1(valor1: int, valor2: str = "Hola ") -> int:
    return f"{valor1 * 5:d}: {valor2}"

print(func1(5, "Mundo "))
print(func1(3))
```

```
25: Mundo
15: Hola
```

Sin embargo, si los invertimos:

```
def func1(valor2: str = "Hola ", valor1: int) -> int:
    return f"{valor1 * 5:d}: {valor2}"

print(func1("Mundo ", 5))
print(func1(None, 3))
```

Al ejecutar el código nos lanza una excepción:

```
  File "E:\PY\elinformati\test.py", line 1
    def func1(valor2: str = "Hola ", valor1: int) -> int:
                                     ^^^^^^^^^^^
SyntaxError: non-default argument follows default argument
```

7. Funciones y clases anidadas

Esto es algo que no tiene gran utilidad y que dudo que sea buena práctica en la mayoría de situaciones, pero es algo que podemos hacer en python y que resulta curioso.

Es posible anidar funciones y clases dentro de otras funciones y clases. Por ejemplo:

```
def func1(a):
    def func2(b):
        return b * b
    return func2(a+1)

print(func1(1))
```

```
4
```

Naturalmente, las funciones y clases que declaremos dentro de una función o clase sólo estarán disponibles en el entorno de dicha función o clase. Por ejemplo:

```
def func1(a):
    def func2(b):
        return b * b
    return func2(a+1)

print(func2(1)) # NameError: name 'func2' is not defined. Did you mean:
'func1'?
```

Un ejemplo con una clase dentro de una función:

```
def func1(a):
    class clase1:
        def __init__(self, a):
            self.valor = a * a
    b = clase1(a + 1)
    return b.valor

print(func1(1))
```

```
4
```

Algo que si que podemos hacer es generar una clase dentro de una función u otra clase, y retornar una instancia de esa clase a una función fuera del entorno de la clase. Por ejemplo:

```
def func1(a):
    class clase1:
        def __init__(self, a):
            self.valor = a * a
    b = clase1(a + 1)
    return b

resultado = func1(1)
print(resultado.valor)
```

```
4
```

Sin embargo, no es normal usar éste tipo de clases y por lo general suelen recurrirse a objetos de tipo tuple, named tuple, diccionarios o similares.

8. No exísten los valores constantes en Python

En lenguajes como C, exíste la posibilidad de usar la palabra reservada 'const' delante de la declaración de una variable para indicar que la variable es *inmutable*. Esto indica que su valor es constante y no puede ser cambiado.

```
const int valor = 1;
valor = 5; // error: assignment of read-only variable 'valor'
```

En Python no exísten éste tipo de variables. Como algo formal, en Python solemos indicar que el valor de una variable no debe ser modificada especificando el nombre en mayúsculas. Pero esto es una formalidad y la variable no deja de ser mutable:

```
VALOR = 1
print(VALOR)
VALOR = 5
print(VALOR)
```

```
1
5
```

No obstante, sí que hay objetos que son inmutables en Python como, por ejemplo, tuples o los objetos de clase BytesIO.

9. Enumeraciones (Enum)

Es posible usar enumeraciones en Python usando el modulo enum que viene incorporado por defecto en Python:

```
from enum import Enum

class Color(Enum):
    ROJO = 1
    VERDE = 2
    AZUL = 3

a = Color.AZUL

match(a):
    case Color.ROJO:
        print("Rojo")
    case Color.VERDE:
        print("Verde")
    case Color.AZUL:
        print("Azul")
    case other:
        print("Indeterminado")
```

```
Azul
```

También podemos obtener el valor de una enumeración instanciando la clase y usando el valor como parámetro:

```
from enum import Enum

class Color(Enum):
    ROJO = 1
    VERDE = 2
    AZUL = 3

a = Color(2)

match(a):
    case Color.ROJO:
        print("Rojo")
    case Color.VERDE:
        print("Verde")
    case Color.AZUL:
        print("Azul")
    case other:
        print("Indeterminado")
```

```
Verde
```

Por último, podemos obtener el valor de un miembro de la enumeración con el parametro *value*.

```
from enum import Enum

class Color(Enum):
    ROJO = 1
    VERDE = 2
    AZUL = 3

a = Color.ROJO

print(a.value)
```

```
1
```

10. Clases dinámicas

Una clase genérica se define de la siguiente manera:

```
class Clase1():
    def __init__(self, a: int):
        self.valor = a + 1
    def suma(self, b: int):
        self.valor += b
        return self.valor

b = Clase1(5)

print(b.suma(3))
```

```
9
```

Podemos ver todos los miembros de cualquier clase instanciada mediante la función *dir*(). La función dir() devuelve una lista con los nombres de todos los miembros de la función, incluidos los métodos:

```
class Clase1():
    def __init__(self, a: int):
        self.valor = a + 1
    def suma(self, b: int):
        self.valor += b
        return self.valor

b = Clase1(5)

print(dir(b))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'suma', 'valor']
```

De modo que si queremos comprobar si existe un miembro en una función, podemos hacer lo siguiente:

```
class Clase1():
    def __init__(self, a: int):
        self.valor = a + 1
    def suma(self, b: int):
        self.valor += b
        return self.valor

b = Clase1(5)

if "valor" in dir(b):
    print(f"Valor: {b.valor:d}")
else:
    print(f"Valor indeterminado")
```

```
Valor: 6
```

Como todo en Python, las clases son *dinámicas*. Mientras que en otros lenguajes las clases se rigen por su declaración, en Python podemos modificar los miembros de cualquier instancia de clase en tiempo real.

Por ejemplo, podemos añadir un miembro a una instancia de clase con la función setattr:

```
class Clase1():
    def __init__(self, a: int):
        self.valor = a + 1
    def suma(self, b: int):
        self.valor += b
        return self.valor

b = Clase1(5)
c = Clase1(10)

setattr(c, "otrovalor", "¡Hola!")

if "otrovalor" in dir(b):
    print(f"Valor: {b.otrovalor:s}")
else:
    print(f"Valor indeterminado")

if "otrovalor" in dir(c):
    print(f"Valor: {c.otrovalor:s}")
else:
    print(f"Valor indeterminado")
```

```
Valor indeterminado
Valor: ¡Hola!
```

Igualmente podemos usar las funciones getattr() y delattr() para obtener el valor de un atributo y eliminar el atributo respectivamente:

```
class Clase1():
    def __init__(self, a: int):
        self.valor = a + 1
    def suma(self, b: int):
        self.valor += b
        return self.valor

b = Clase1(5)
c = Clase1(10)

setattr(c, "otrovalor", "¡Hola!")

if "otrovalor" in dir(b):
    print(f"B: {getattr(b, 'otrovalor'):s}")

if "otrovalor" in dir(c):
    print(f"C: {getattr(c, 'otrovalor'):s}")
    delattr(c, "otrovalor")
    if "otrovalor" in dir(c):
        print(f"C: {getattr(c, 'otrovalor'):s}")
    else:
        print("'otrovalor' se ha eliminado de C")
```

```
C: ¡Hola!
'otrovalor' se ha eliminado de C
```

Es posible añadir metodos a los objetos de clase, aunque con limitaciones. Por lo general, solo podemos añadir metodos *estáticos*. Es decir, no podremos hacer referencia al objeto de clase. Por ejemplo:

```
class Clase1():
    def __init__(self, a: int):
        self.valor = a + 1
    def suma(self, b: int):
        self.valor += b
        return self.valor

def resta(c: int):
    return c - 1

b = Clase1(5)
c = Clase1(10)

setattr(c, "resta", resta)

print(c.resta(5))
```

```
4
```

Si intentaramos insertar un método de clase, nos daría error indicando que nos falta el argumento 'self' al llamar la función:

```
def resta(self, c: int):
    self.valor -= c
    return self.valor

b = Clase1(5)
c = Clase1(10)

setattr(c, "resta", resta)

print(c.resta(5))
```

```
    print(c.resta(5))
TypeError: resta() missing 1 required positional argument: 'c'
```

De modo que al final la única manera es pasar una referencia del objeto en el parametro self:

```
print(c.resta(c, 5))
```

```
6
```

Ya que los metodos definidos como classmethod no pueden ser llamados de ésta forma:

```
@classmethod
def resta(self, c: int):
    self.valor -= c
    return self.valor

b = Clase1(5)
c = Clase1(10)

setattr(c, "resta", resta)

print(c.resta(5))
```

```
    print(c.resta(5))
TypeError: 'classmethod' object is not callable
```

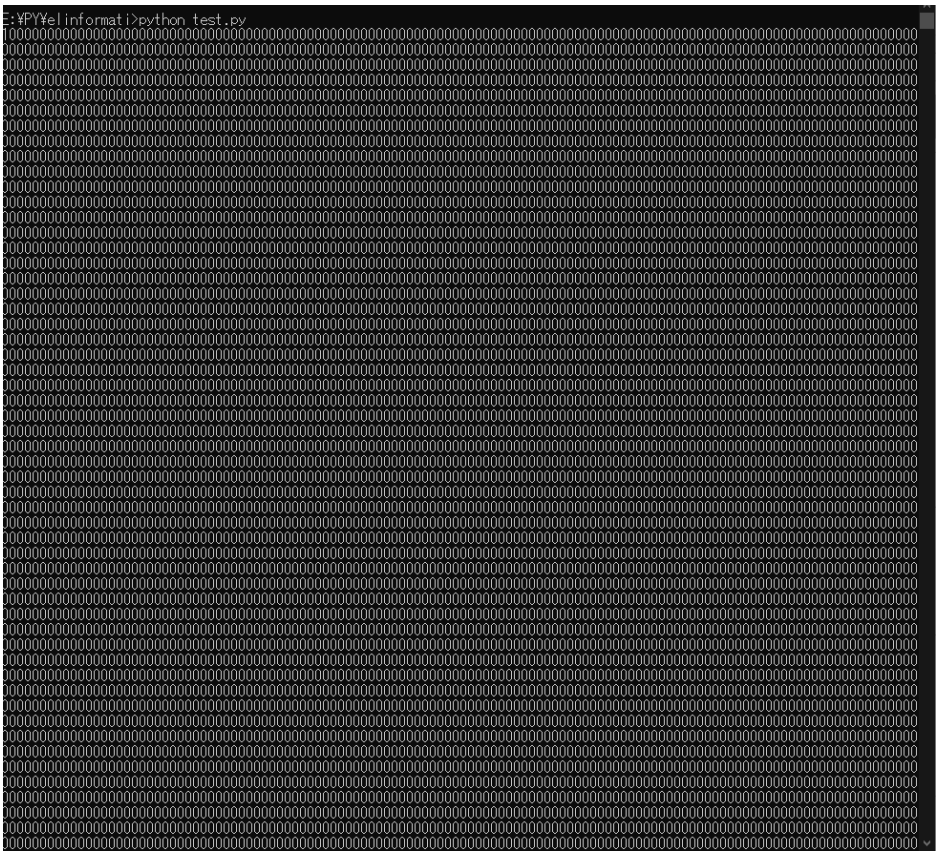
Por lo general todo éste tipo de prácticas son malas prácticas y es preferible recurrir a objetos de diccionario o similares, pero al igual que con las funciones y clases anidadas es una curiosidad el saber que podemos hacerlo.

11. Tipos de C en Python

Python no tiene limites a la hora de almacenar valores en las variables. Por lo general, podemos almacenar cualquier tipo de valor. Y podemos usar tanta memoria como nos permita el sistema. Un ejemplo de ésto es que los valores numéricos en Python no tienen limite. Por ejemplo:

```
a = 100 ** 100 ** 2
print(a)
```

Esto nos devuelve el siguiente valor:



Un número enorme, ¿verdad? Por lo general, ésto suele ser una ventaja ya que no estamos limitados a los límites de los tipos de C ni tenemos que preocuparnos del *endianess* ni del tamaño en la memoria de dichos valores. Esto es porque como en todo lenguaje POO los valores son *seguros* (memory safe).

Pero cuando estamos intentando manipular datos binarios como, por ejemplo, formatos de archivo o datos en la memoria, esto puede dar lugar a problemas. Uno de estos problemas me vino cuando estuve escribiendo un módulo para administrar imágenes de disquette en FAT12 usando Python. A la hora de actualizar el valor del siguiente cluster, se calcula de la siguiente manera:

```
        if cluster & 0x001 == 1:
            siguiente = ((nuevo_valor << 4) | (valor_previo & 0x000F))
        else:
            siguiente = ((valor_previo << 12) | nuevo_valor)
        siguiente_bytes = bytearray(siguiente.to_bytes(2,
byteorder='little'))
```

¿El problema? Que éste valor debe representarse en 2 bytes. En C normalmente usariamos un valor unsigned short o uint16_t (definido en stdint.h). Pero como en python los valores numéricos pueden tomar cualquier valor, si el resultado excede los dos bytes al hacer el shifting, el nuevo valor aumentará de tamaño a 3 o más bytes. Esto puede dar lugar a problemas. Un ejemplo más simple para ilustrarlo:

```
siguiente = 65535 # El valor máximo representable en un uint16 (2 bytes,
o 16 bits) es 65535
siguiente_bytes = bytearray(siguiente.to_bytes(2, byteorder='little')) #
Convierte el valor en un bytearray de 2 bytes.
print(len(siguiente_bytes)) # Debería mostrar '2' como resultado

siguiente = siguiente << 1 # shl 1 byte.
siguiente_bytes = bytearray(siguiente.to_bytes(2, byteorder='little')) #
Error! Ahora el valor debería tomar 3 bytes en vez de 2.
print(len(siguiente_bytes))
```

```
2
Traceback (most recent call last):
  File "E:\PY\elinformati\test.py", line 10, in <module>
    siguiente_bytes = bytearray(siguiente.to_bytes(2, byteorder='little'))
OverflowError: int too big to convert
```

Exísten dos soluciones. Una es enmascarar el resultado para forzar un valor representable en dos bytes (omitiendo los bytes que esten fuera de los dos primeros bytes):

```
valor16 = 65540 # El valor máximo representable en un uint16 (2 bytes, o
16 bits) es 65535
bytes16 = bytearray((valor16 & 0xFFFF).to_bytes(2, byteorder='little')) #
valor16 & 0xFFFF mantiene sólo los dos primeros bytes de valor16
print(len(bytes16)) # Debería mostrar 2
```

```
2
```

La segunda solución es usar el módulo ctypes integrado en Python para almacenar un valor de un tipo determinado.

```
from ctypes import c_uint16

nuevovalor = c_uint16(65540) # El valor máximo representable en un uint16
(2 bytes, o 16 bits) es 65535
print(nuevovalor.value) # Debería mostrar 4 (Rota de 65535 a 0 (overflow)
y sigue contando hasta 4)
bytes16 = bytearray(nuevovalor.value.to_bytes(2, byteorder='little')) #
Convierte el valor en un bytearray de 2 bytes.
print(len(bytes16)) # Debería mostrar 2
```

```
4
2
```

El módulo de ctypes es muy potente y nos permite usar muchos más tipos de valores, incluidos punteros, e incluso nos permite realizar llamadas a functiones de librerias en C, y usar estructuras (Struct) y uniones (Union) entre otras cosas. Puedes ver la documentación aquí.

12 Ayuda en tiempo de ejecución

Seguro que cuando necesitas ayuda con alguna de las funciones o modulos de python, acudes a la documentación online de Python, o bien buscas en Google, o en Stack Overflow. Pero quizás no sepas que Python incorpora la función help() para ofrecer ayuda de cualquier clase o función. Esto puede ser de mucha ayuda si llamas a la función desde la consola interactiva de python:

```
>>> a = 5
>>> help(a)
```

Puedes llamar a la función help sobre cualquier método o objeto de clase que contenga un *docstring*.

13. La función 'enumerate'

Si alguna vez trabajas con listas y quieres iterar sobre los elementos obteniendo a la vez el indice del elemento, puede que hayas recurrido a éste código:

```
lista = ["Uno", "Dos", "Tres", "Cuatro", "Cinco"]

for n in range(0,len(lista)):
    print(f"{n:d}: {lista[n]}")
```

```
0: Uno
1: Dos
2: Tres
3: Cuatro
4: Cinco
```

Existe una manera algo más elegante usando la función enumerate, que devuelve el indice del elemento y el elemento:

```
lista = ["Uno", "Dos", "Tres", "Cuatro", "Cinco"]

for (n,objeto) in enumerate(lista):
    print(f"{n:d}: {objeto}")
```

```
0: Uno
1: Dos
2: Tres
3: Cuatro
4: Cinco
```

14. Iterar a traves de claves y valores de diccionarios

Los diccionarios son arrays «asociativas». Es decir, son listas de valores asociadas a una clave (o nombre). Si alguna vez necesitas iterar sobre las claves de una lista y sus correspondientes valores, lo puedes hacer de la siguiente manera:

```
objetos = {
    "uno": 1,
    "dos": 2,
    "tres": 4
}

for (clave, objeto) in objetos.items():
    print(f"{clave}: {objeto:d}")
```

```
uno: 1
dos: 2
tres: 4
```

15. For...Else

Por lo general podemos iterar a traves de los elementos de una lista en orden mediante la clausula *for*. Por ejemplo:

```
lista = ["Uno", "Cinco", "Diez", "Veinte"]

for elemento in lista:
    if elemento == "Diez":
        print(f"Elemento encontrado: {elemento}")
```

```
Elemento encontrado: Diez
```

Pero, ¿Qué pasa si estamos, por ejemplo, buscando un elemento, pero éste no se encuentra en la lista?

```
lista = ["Uno", "Cinco", "Diez", "Veinte"]

for elemento in lista:
    if elemento == "Once":
        print(f"Elemento encontrado: {elemento}")
        break


# El programa no genera ninguna salida
```

En caso de que queramos ejecutar un bloque de código al finalizar la iteración completa, podemos añadir una clausula 'else' al for:

```
lista = ["Uno", "Cinco", "Diez", "Veinte"]
for elemento in lista:
    if elemento == "Once":
        print(f"Elemento encontrado: {elemento}")
        break
else:
    print("No se ha encontrado el elemento.")
```

```
No se ha encontrado el elemento.
```

Si buscamos un elemento que si se encuentra en la lista:

```
lista = ["Uno", "Cinco", "Diez", "Veinte"]

for elemento in lista:
    if elemento == "Cinco":
        print(f"Elemento encontrado: {elemento}")
        break
else:
    print("No se ha encontrado el elemento.")
```

```
Elemento encontrado: Cinco
```

El bloque '*else*' se ejecutará siempre que finalice la iteración si no se interrumpe antes mediante un *break*.

16. Repetir elementos en una lista

Además de crear una lista mediante un iterador, podemos hacer que se repitan los elementos de una lista tantas veces como queramos. Por ejemplo:

```
a = [10] * 10
print(a)
```

```
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

O también:

```
a = [1,2,3,4,5] * 3
print(a)
```

```
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Esto puede ser útil en ocasiones para formar cadenas de texto, entre otras cosas. Por ejemplo:

```
a = ''.join(["Palo"] * 5)
print(a)
```

```
PaloPaloPaloPaloPalo
```

ANTERIOR

Crear documentos PDF con Python y PyMuPDF

SIGUIENTE

Consejos de seguridad a la hora de usar el correo, mensajería instantanea, y SMS

Buscar ...

Entradas Recientes

- Estoy hasta las narices de la web moderna
- Ingenieria inversa básica con Ghidra
- Acerca de la nueva ley transgénero (Y sobre la disfória de género)
- Depresiones causadas por las redes sociales
- ¿Necesito saber matemáticas para aprender informática?
- ¿Es el fín de los discos duros tradicionales?

Categorías

Actualidad | Android | Básicos | Ciberseguridad | Clima | Criptografía | Electronica | Emulación / Virtualización | FOSS | Hacking | Hardware | Informática | Internet | Juegos | Opinion | Otros | Personal | Privacidad | Programación | Tecnología | Time Machine | Tutoriales

RSS

Subscribirse al feed RSS

Catálogo
PDFs
Manuales
Política de privacidad
Política de Cookies
Acerca de mi
Acerca de ElInformati.co