6.2. Definición de funciones

Vamos a estudiar el modo en que podemos definir (y usar) nuestras propias funciones Python. Estudiaremos en primer lugar cómo definir y llamar a funciones que devuelven un valor y pasaremos después a presentar los denominados procedimientos: funciones que no devuelven ningún valor. Además de los conceptos y técnicas que te iremos presentando, es interesante que te fijes en cómo desarrollamos los diferentes programas de ejemplo.

6.2.1. Definición y uso de funciones con un solo parámetro

Empezaremos definiendo una función muy sencilla, una que recibe un número y devuelve el cuadrado de dicho número. El nombre que daremos a la función es *cuadrado*. Observa este fragmento de programa:

cuadrado.py

```
1 def cuadrado(x):
2     return x ** 2
```

Ya está. Acabamos de definir la función *cuadrado* que se aplica sobre un valor al que llamamos *x* y devuelve un número: el resultado de elevar *x* al cuadrado. En el programa aparecen dos nuevas palabras reservadas: **def** y **return**. La palabra *def* es abreviatura de «define» y **return** significa «devuelve» en inglés. Podríamos leer el programa anterior como «define cuadrado de *x* como el valor que resulta de elevar *x* al cuadrado».

En las líneas que siguen a su definición, la función *cuadrado* puede utilizarse del mismo modo que las funciones predefinidas:

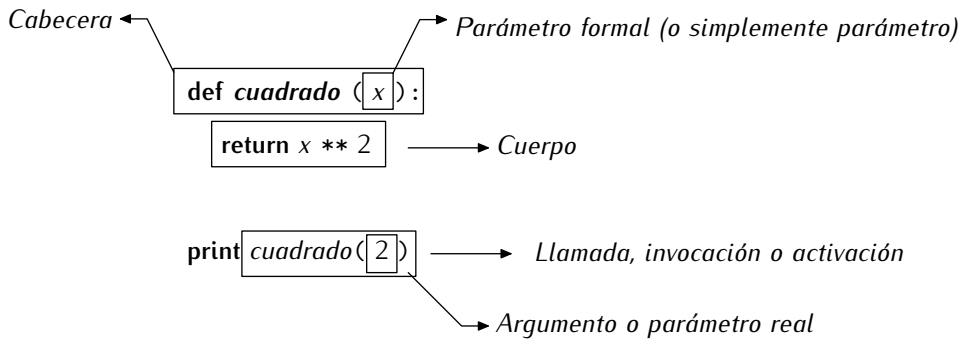
cuadrado.py

```
1 def cuadrado(x):
2     return x ** 2
3
4 print cuadrado(2)
5 a = 1 + cuadrado(3)
6 print cuadrado(a * 3)
```

En cada caso, el resultado de la expresión que sigue entre paréntesis al nombre de la función es utilizado como valor de *x* durante la ejecución de *cuadrado*. En la primera llamada (línea 4) el valor es 2, en la siguiente llamada es 3 y en la última, 30. Fácil, ¿no?

Detengámonos un momento para aprender algunos términos nuevos. La línea que empieza con **def** es la *cabecera* de la función y el fragmento de programa que contiene los cálculos que debe efectuar la función se denomina *cuerpo* de la función. Cuando

estamos definiendo una función, su parámetro se denomina *parámetro formal* (aunque, por abreviar, normalmente usaremos el término *parámetro*, sin más). El valor que *pasamos* a una función cuando la invocamos se denomina *parámetro real* o *argumento*. Las porciones de un programa que no son cuerpo de funciones forman parte del *programa principal*: son las sentencias que se ejecutarán cuando el programa entre en acción. El cuerpo de las funciones sólo se ejecutará si se producen las correspondientes llamadas.



Definir no es invocar

Si intentamos ejecutar este programa:

cuadrado_4.py — cuadrado.py

```
1 def cuadrado(x):
2     return x ** 2
```

no ocurrirá nada en absoluto; bueno, al menos nada que aparezca por pantalla. La definición de una función sólo hace que Python «aprenda» *silenciosamente* un método de cálculo asociado al identificador *cuadrado*. Nada más. Hagamos la prueba ejecutando el programa:

```
$ python cuadrado.py ↲
```

¿Lo ves? No se ha impreso nada en pantalla. No se trata de que no haya ningún **print**, sino de que definir una función es un proceso que no tiene eco en pantalla. Repetimos: definir una función sólo asocia un método de cálculo a un identificador y no supone ejecutar dicho método de cálculo.

Este otro programa sí muestra algo por pantalla:

cuadrado_5.py — cuadrado.py

```
1 def cuadrado(x):
2     return x ** 2
3
4 print cuadrado(2)
```

Al invocar la función *cuadrado* (línea 4) se ejecuta ésta. En el programa, la invocación de la última línea provoca la ejecución de la línea 2 con un valor de *x* igual a 2 (argumento de la llamada). El valor devuelto con **return** es mostrado en pantalla como efecto de la sentencia **print** de la línea 4. Hagamos la prueba:

```
$ python cuadrado.py ↲
4
```

Las reglas para dar nombre a las funciones y a sus parámetros son las mismas que seguimos para dar nombre a las variables: sólo se pueden usar letras (del alfabeto inglés),

Definición de funciones desde el entorno interactivo

Hemos aprendido a definir funciones dentro de un programa. También puedes definir funciones desde el entorno interactivo de Python. Te vamos a enseñar paso a paso qué ocurre en el entorno interactivo cuando estamos definiendo una función.

En primer lugar aparece el *prompt*. Podemos escribir entonces la primera línea:

```
>>> def cuadrado(x): ↲
... ↲
```

Python nos responde con tres puntos (...). Esos tres puntos son el llamado *prompt secundario*: indica que la acción de definir la función no se ha completado aún y nos pide más sentencias. Escribimos a continuación la segunda línea respetando la indentación que le corresponde:

```
>>> def cuadrado(x): ↲
...     return x ** 2 ↲
... ↲
```

Nuevamente Python responde con el *prompt* secundario. Es necesario que le demos una vez más al retorno de carro para que Python entienda que ya hemos acabado de definir la función:

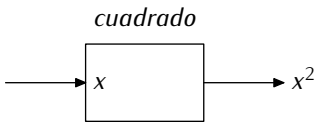
```
>>> def cuadrado(x): ↲
...     return x ** 2 ↲
... ↲
>>>
```

Ahora aparece de nuevo el *prompt* principal o primario. Python ha aprendido la función y está listo para que introduzcamos nuevas sentencias o expresiones.

```
>>> def cuadrado(x): ↲
...     return x ** 2 ↲
... ↲
>>> cuadrado(2) ↲
4
>>> 1 + cuadrado(1+3) ↲
17
>>>
```

dígitos y el carácter de subrayado; la primera letra del nombre no puede ser un número; y no se pueden usar palabras reservadas. Pero, ¡cuidado!: no debes dar el mismo nombre a una función y a una variable. En Python, cada nombre debe identificar claramente un único elemento: una variable o una función.[1]

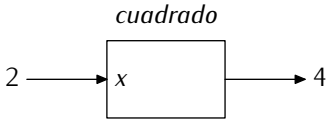
Al definir una función cuadrado es como si hubiésemos creado una «máquina de calcular cuadrados». Desde la óptica de su uso, podemos representar la función como una caja que transforma un *dato de entrada* en un *dato de salida*:



Cuando invocas a la función, le estás «conectando» un valor a la entrada, así que la «máquina de calcular cuadrados» se pone en marcha y produce la solución deseada:

[1] Más adelante, al presentar las variables locales, matizaremos esta afirmación.

```
>>> cuadrado(2) ↲
4
```



Ojo: no hay una única forma de construir la «máquina de calcular cuadrados». Fíjate en esta definición alternativa:

cuadrado.6.py — cuadrado.py

```
1 def cuadrado(x):
2     return x * x
```

Se trata de un definición tan válida como la anterior, ni mejor, ni peor. Como usuarios de la función, poco nos importa *cómo* hace el cálculo[2]; lo que importa es *qué datos recibe* y *qué valor devuelve*.

Vamos con un ejemplo más: una función que calcula el valor de x por el seno de x:

```
1 from math import sin
2
3 def xsin(x):
4     return x * sin(x)
```

Lo interesante de este ejemplo es que la función definida, *xsin*, contiene una llamada a otra función (*sin*). No hay problema: desde una función puedes invocar a cualquier otra.

Una confusión frecuente

Supongamos que definimos una función con un parámetro x como esta:

```
1 def cubo(x):
2     return x ** 3
```

Es frecuente en los aprendices confundir el parámetro x con una variable x. Así, les parece extraño que podamos invocar así a la función:

```
4 y = 1
5 print cubo(y)
```

¿Cómo es que ahora llamamos y a lo que se llamaba x? No hay problema alguno. Al definir una función, usamos un identificador cualquiera para referirnos al parámetro. Tanto da que se llame x como y. Esta otra definición de cubo es absolutamente equivalente:

```
1 def cubo(z):
2     return z ** 3
```

La definición se puede leer así: «si te pasan un valor, digamos z, devuelve ese valor elevado al cubo». Usamos el nombre z (o x) sólo para poder referirnos a él en el cuerpo de la función.

EJERCICIOS

▶ 262 Define una función llamada *raiz_cubica* que devuelva el valor de $\sqrt[3]{x}$.

(Nota: recuerda que $\sqrt[3]{x}$ es $x^{1/3}$ y ándate con ojo, no sea que utilices una división entera y eleves x a la potencia 0, que es el resultado de calcular 1/3.)

[2] ... por el momento. Hay muchas formas de hacer el cálculo, pero unas resultan más *eficientes* (más rápidas) que otras. Naturalmente, cuando podamos elegir, escogeremos la forma más eficiente.

▶ 263 Define una función llamada *area_circulo* que, a partir del radio de un círculo, devuelva el valor de su área. Utiliza el valor 3.1416 como aproximación de π o importa el valor de π que encontrarás en el módulo *math*.

(Recuerda que el área de un círculo es πr^2.)

▶ 264 Define una función que convierta grados Farenheit en grados centígrados.

(Para calcular los grados centígrados has de restar 32 a los grados Farenheit y multiplicar el resultado por cinco novenos.)

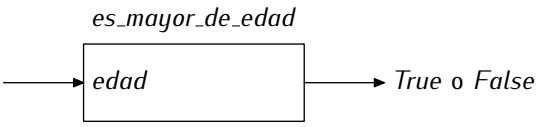
▶ 265 Define una función que convierta grados centígrados en grados Farenheit.

▶ 266 Define una función que convierta radianes en grados.

(Recuerda que 360 grados son 2π radianes.)

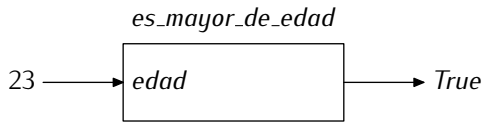
▶ 267 Define una función que convierta grados en radianes.

En el cuerpo de una función no sólo pueden aparecer sentencias **return**; también podemos usar estructuras de control: sentencias condicionales, bucles, etc. Lo podemos comprobar diseñando una función que recibe un número y devuelve un booleano. El valor de entrada es la edad de una persona y la función devuelve *True* si la persona es mayor de edad y *False* en caso contrario:

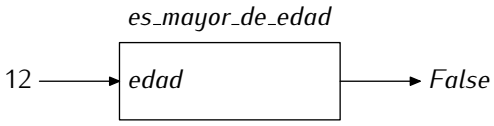


Cuando llamas a la función, ésta se activa para producir un resultado concreto (en nuestro caso, o bien devuelve *True* o bien devuelve *False*):

```
a = es_mayor_de_edad(23)
```




```
b = es_mayor_de_edad(12)
```



Una forma usual de devolver valores de función es a través de un sólo **return** ubicado al final del cuerpo de la función:

mayoria_edad_4.py — mayoria_edad.py

```
def es_mayor_de_edad(edad):
    if edad < 18:
        resultado = False
    else:
        resultado = True
    return resultado
```

Pero no es el único modo en que puedes devolver diferentes valores. Mira esta otra definición de la misma función:

mayoria_edad.py

```
def es_mayor_de_edad(edad):
    if edad < 18:
        return False
    else:
        return True
```

Aparecen dos sentencias **return**: cuando la ejecución llega a cualquiera de ellas, finaliza *inmediatamente* la llamada a la función y se devuelve el valor que sigue al **return**. Podemos asimilar el comportamiento de **return** al de **break**: una sentencia **break** fuerza a terminar la ejecución de un bucle y una sentencia **return** fuerza a terminar la ejecución de una llamada a función.

EJERCICIOS

▶ 268 ¿Es este programa equivalente al que acabamos de ver?

mayoria_edad_5.py

```
def mayoria_de_edad(edad):
    if edad < 18:
        return False
    return True
```

▶ 269 ¿Es este programa equivalente al que acabamos de ver?

mayoria_edad_6.py

```
def mayoria_de_edad(edad):
    return edad >= 18
```

▶ 270 La última letra del DNI puede calcularse a partir del número. Para ello sólo tienes que dividir el número por 23 y quedarte con el resto, que es un número entre 0 y 22. La letra que corresponde a cada número la tienes en esta tabla:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

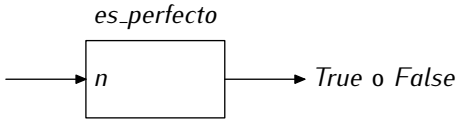
Define una función que, dado un número de DNI, devuelva la letra que le corresponde.

▶ 271 Diseña una función que reciba una cadena y devuelva cierto si empieza por minúscula y falso en caso contrario.

▶ 272 Diseña una función llamada *es_repeticion* que reciba una cadena y nos diga si la cadena está formada mediante la concatenación de una cadena consigo misma. Por ejemplo, *es_repeticion*`('abab')` devolverá *True*, pues la cadena `'abab'` está formada con la cadena `'ab'` repetida; por contra *es_repeticion*`('ababab')` devolverá *False*.

Y ahora, un problema más complicado. Vamos a diseñar una función que nos diga si un número dado es o no es *perfecto*. Se dice que un número es perfecto si es igual a la suma de todos sus divisores excluído él mismo. Por ejemplo, 28 es un número perfecto, pues sus divisores (excepto él mismo) son 1, 2, 4, 7 y 14, que suman 28.

Empecemos. La función, a la que llamaremos *es_perfecto* recibirá un sólo dato (el número sobre el que hacemos la pregunta) y devolverá un valor booleano:



La cabecera de la función está clara:

perfecto.py

```
1 def es_perfecto(n):
2     ...
```

¿Y por dónde seguimos? Vamos por partes. En primer lugar estamos interesados en conocer todos los divisores del número. Una vez tengamos claro cómo saber cuáles son, los sumaremos. Si la suma coincide con el número original, éste es perfecto; si no, no. Podemos usar un bucle y preguntar a todos los números entre 1 y $n-1$ si son divisores de n:

perfecto.py

```
1 def es_perfecto(n):
2     for i in range(1, n):
3         if i es divisor de n:
4             ...
```

Observa cómo seguimos siempre la reglas de indentación de código que impone Python. ¿Y cómo preguntamos ahora si un número es divisor de otro? El operador módulo **%** devuelve el resto de la división y resuelve fácilmente la cuestión:

perfecto.py

```
1 def es_perfecto(n):
2     for i in range(1, n):
3         if n % i == 0:
4             ...
```

La línea 4 sólo se ejecutará para valores de i que son divisores de n. ¿Qué hemos de hacer a continuación? Deseamos sumar todos los divisores y ya conocemos la «plantilla» para calcular sumatorios:

perfecto.py

```
1 def es_perfecto(n):
2     sumatorio = 0
3     for i in range(1, n):
4         if n % i == 0:
5             sumatorio += i
6     ...
```

¿Qué queda por hacer? Comprobar si el número es perfecto y devolver *True* o *False*, según proceda:

perfecto_3.py

perfecto.py

```
1 def es_perfecto(n):
2     sumatorio = 0
3     for i in range(1, n):
4         if n % i == 0:
5             sumatorio += i
6     if sumatorio == n:
7         return True
8     else:
9         return False
```

Y ya está. Bueno, podemos simplificar un poco las cuatro últimas líneas y convertirlas en una sola. Observa esta nueva versión:

perfecto.py

perfecto.py

```
1 def es_perfecto(n):
2     sumatorio = 0
3     for i in range(1, n):
4         if n % i == 0:
5             sumatorio += i
6     return sumatorio == n
```

¿Qué hace la última línea? Devuelve el resultado de evaluar la expresión lógica que compara *sumatorio* con *n*: si ambos números son iguales, devuelve *True*, y si no, devuelve *False*. Mejor, ¿no?

EJERCICIOS

▶ 273 ¿En qué se ha equivocado nuestro aprendiz de programador al escribir esta función?

perfecto_4.py — perfecto.py

```
def es_perfecto(n):
    for i in range(1, n):
        sumatorio = 0
        if n % i == 0:
            sumatorio += i
    return sumatorio == n
```

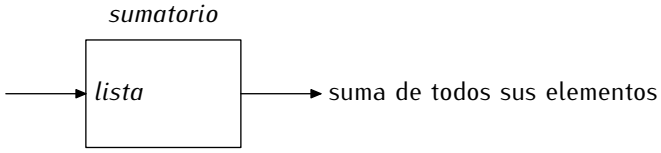
▶ 274 Mejora la función *es_perfecto* haciéndola más rápida. ¿Es realmente necesario considerar todos los números entre 1 y *n*−1?

▶ 275 Diseña una función que devuelva una lista con los números perfectos comprendidos entre 1 y *n*, siendo *n* un entero que nos proporciona el usuario.

▶ 276 Define una función que devuelva el número de días que tiene un año determinado. Ten en cuenta que un año es bisiesto si es divisible por 4 y no divisible por 100, excepto si es también divisible por 400, en cuyo caso es bisiesto.

(Ejemplos: El número de días de 2002 es 365: el número 2002 no es divisible por 4, así que no es bisiesto. El año 2004 es bisiesto y tiene 366 días: el número 2004 es divisible por 4, pero no por 100, así que es bisiesto. El año 1900 es divisible por 4, pero no es bisiesto porque es divisible por 100 y no por 400. El año 2000 sí es bisiesto: el número 2000 es divisible por 4 y, aunque es divisible por 100, también lo es por 400.)

Hasta el momento nos hemos limitado a suministrar valores escalares como argumentos de una función, pero también es posible suministrar argumentos de tipo secuencial. Veámoslo con un ejemplo: una función que recibe una lista de números y nos devuelve el sumatorio de todos sus elementos.



suma_lista_4.py — suma_lista.py

```
def sumatorio(lista):
    s = 0
    for numero in lista:
        s += numero
    return s
```

Podemos usar la función así:

suma_lista_5.py — suma_lista.py

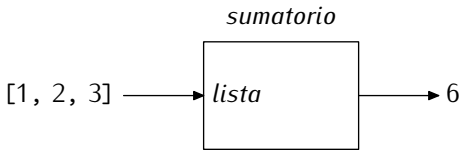
```
⋮
a = [1, 2, 3]
print sumatorio(a)
```

o así:

suma_lista_6.py — suma_lista.py

```
⋮
print sumatorio([1, 2, 3])
```

En cualquiera de los dos casos, el parámetro *lista* toma el valor `[1, 2, 3]`, que es el argumento suministrado en la llamada:



Sumatorios

Has aprendido a calcular sumatorios con bucles. Desde la versión 2.3, Python ofrece una forma mucho más cómoda de calcular sumatorios: la función predefinida *sum*, que recibe una lista de valores y devuelve el resultado de sumarlos.

```
>>> sum([1, 10, 20]) ↲
31
```

¿Cómo usarla para calcular el sumatorio de los 100 primeros números naturales? Muy fácil: pasándole una lista con esos número, algo que resulta trivial si usas *range*.

```
>>> sum(range(101)) ↲
5050
```

Mmmm. Ten cuidado: *range* construye una lista en memoria. Si calculas así el sumatorio del primer millón de números es posible que te quedes sin memoria. Hay una función alternativa, *xrange*, que no construye la lista en memoria, pero que hace creer a quien la recorre que es una lista en memoria:

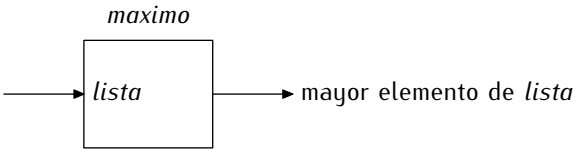
```
>>> sum(xrange(1000001)) ↲
500000500000L
```

EJERCICIOS

▶ 277 Diseña una función que calcule el sumatorio de la diferencia entre números contiguos en una lista. Por ejemplo, para la lista `[1, 3, 6, 10]` devolverá 9, que es $2+3+4$ (el 2 resulta de calcular $3-1$, el 3 de calcular $6-3$ y el 4 de calcular $10-6$).

¿Sabes efectuar el cálculo de ese sumatorio sin utilizar bucles (ni la función *sum*)?

Estudiemos otro ejemplo: una función que recibe una lista de números y devuelve el valor de su mayor elemento.



La idea básica es sencilla: recorrer la lista e ir actualizando el valor de una variable auxiliar que, en todo momento, contendrá el máximo valor visto hasta ese momento.

maximo_7.py — maximo.py

```
def maximo(lista):
```

```
2     for elemento in lista:
3         if elemento > candidato:
4             candidato = elemento
5     return candidato
```

Nos falta inicializar la variable *candidato*. ¿Con qué valor? Podríamos pensar en inicializarla con el menor valor posible. De ese modo, cualquier valor de la lista será mayor que él y es seguro que su valor se modificará tan pronto empecemos a recorrer la lista. Pero hay un problema: no sabemos cuál es el menor valor posible. Una buena alternativa es inicializar *candidato* con el valor del primer elemento de la lista. Si ya es el máximo, perfecto, y si no lo es, más tarde se modificará *candidato*.

maximo_8.py — maximo.py

```
1 def maximo(lista):
2     candidato = lista[0]
3     for elemento in lista:
4         if elemento > candidato:
5             candidato = elemento
6     return candidato
```

EJERCICIOS

▶ 278 Haz una traza de la llamada *maximo*([6, 2, 7, 1, 10, 1, 0]).

¿Ya está? Aún no. ¿Qué pasa si se proporciona una lista vacía como entrada? La línea 2 provocará un error de tipo *IndexError*, pues en ella intentamos acceder al primer elemento de la lista... y la lista vacía no tiene ningún elemento. Un objetivo es, pues, evitar ese error. Pero, en cualquier caso, algo hemos de devolver como máximo elemento de una lista, ¿y qué valor podemos devolvemos como máximo elemento de una lista vacía? Mmmm. A bote pronto, tenemos dos posibilidades:

- Devolver un valor especial, como el valor 0. Mejor no. Tiene un serio inconveniente: ¿cómo distinguiré el máximo de [-3, -5, 0, -4], que es un cero «legítimo», del máximo de []?
- O devolver un valor «muy» especial, como el valor *None*. ¿Que qué es *None*? *None* significa en inglés «ninguno» y es un valor predefinido en Python que se usa para denotar «ausencia de valor». Como el máximo de una lista vacía no existe, parece acertado devolver la «ausencia de valor» como máximo de sus miembros.

Nos inclinamos por esta segunda opción. En adelante, usaremos *None* siempre que queramos referirnos a un valor «muy» especial: a la ausencia de valor.

maximo.py — maximo.py

```
1 def maximo(lista):
2     if len(lista) > 0:
3         candidato = lista[0]
4         for elemento in lista:
5             if elemento > candidato:
6                 candidato = elemento
7     else:
8         candidato = None
9     return candidato
```

EJERCICIOS

▶ 279 Diseña una función que, dada una lista de números enteros, devuelva el número de «series» que hay en ella. Llamamos «serie» a todo tramo de la lista con valores idénticos.

Por ejemplo, la lista [1, 1, 8, 8, 8, 8, 0, 0, 0, 2, 10, 10] tiene 5 «series» (ten en cuenta que el 2 forma parte de una «serie» de un solo elemento).

▶ 280 Diseña una función que diga en qué posición empieza la «serie» más larga de una lista. En el ejemplo del ejercicio anterior, la «serie» más larga empieza en la posición 2 (que es el índice donde aparece el primer 8). (Nota: si hay dos «series» de igual longitud y ésta es la mayor, debes devolver la posición de la primera de las «series». Por ejemplo, para `[8, 2, 2, 9, 9]` deberás devolver la posición 1.)

▶ 281 Haz una función que reciba una lista de números y devuelva la media de dichos números. Ten cuidado con la lista vacía (su media es cero).

▶ 282 Diseña una función que calcule el productorio de todos los números que componen una lista.

▶ 283 Diseña una función que devuelva el valor absoluto de la máxima diferencia entre dos elementos consecutivos de una lista. Por ejemplo, el valor devuelto para la lista `[1, 10, 2, 6, 2, 0]` es 9, pues es la diferencia entre el valor 1 y el valor 10.

▶ 284 Diseña una función que devuelva el valor absoluto de la máxima diferencia entre cualquier par de elementos de una lista. Por ejemplo, el valor devuelto para la lista `[1, 10, 2, 6, 8, 2 0]` es 9, pues es la diferencia entre el valor 10 y el valor 0. (Pista: te puede convenir conocer el valor máximo y el valor mínimo de la lista.)

▶ 285 Modifica la función del ejercicio anterior para que devuelva el valor 0 tan pronto encuentre un 0 en la lista.

▶ 286 Define una función que, dada una cadena *x*, devuelva otra cuyo contenido sea el resultado de concatenar 6 veces *x* consigo misma.

▶ 287 Diseña una función que, dada una lista de cadenas, devuelva la cadena más larga. Si dos o más cadenas miden lo mismo y son las más largas, la función devolverá una cualquiera de ellas.

(Ejemplo: dada la lista `['Pepe', 'Juan', 'María', 'Ana']`, la función devolverá la cadena `'María'`.)

▶ 288 Diseña una función que, dada una lista de cadenas, devuelva *una lista con todas* las cadenas más largas, es decir, si dos o más cadenas miden lo mismo y son las más largas, la lista las contendrá a todas.

(Ejemplo: dada la lista `['Pepe', 'Ana', 'Juan', 'Paz']`, la función devolverá la lista de dos elementos `['Pepe', 'Juan']`.)

▶ 289 Diseña una función que reciba una lista de cadenas y devuelva el prefijo común más largo. Por ejemplo, la cadena `'pol'` es el prefijo común más largo de esta lista:

```
['poliedro', 'policía', 'polífona', 'polinizar', 'polaridad', 'política']
```