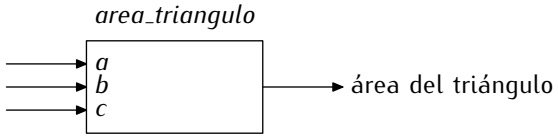
6.4. Variables locales y variables globales

Observa que en el cuerpo de las funciones es posible definir y usar variables. Vamos a estudiar con detenimiento algunas propiedades de las variables definidas en el cuerpo de una función y en qué se diferencian de las variables que definimos fuera de cualquier función, es decir, en el denominado programa principal.

Empecemos con un ejemplo. Definamos una función que, dados los tres lados de un triángulo, devuelva el valor de su área. Recuerda que si a, b y c son dichos lados, el área del triángulo es

$$\sqrt{s(s-a)(s-b)(s-c)},$$

donde $s = (a+b+c)/2$.



La función se define así:

triangulo_6.py — triangulo.py

```
1 from math import sqrt
2
3 def area_triangulo(a, b, c):
4     s = (a + b + c) / 2.0
5     return sqrt(s * (s-a) * (s-b) * (s-c))
```

La línea 4, en el cuerpo de la función, define la variable s asignándole un valor que es instrumental para el cálculo del área del triángulo, es decir, que no nos interesa por sí mismo, sino por ser de ayuda para obtener el valor que realmente deseamos calcular: el que resulta de evaluar la expresión de la línea 5.

La función *area_triangulo* se usa como cabe esperar:

triangulo_7.py — triangulo.py

```
⋮
7 print area_triangulo(1, 3, 2.5)
```

Ahora viene lo importante: la variable *s sólo existe en el cuerpo de la función*. Fuera de dicho cuerpo, s no está definida. El siguiente programa provoca un error al ejecutarse porque intenta acceder a s desde el programa principal:

triangulo_8.py — triangulo.py

```
1 from math import sqrt
2
3 def area_triangulo(a, b, c):
4     s = (a + b + c) / 2.0
5     return sqrt(s * (s-a) * (s-b) * (s-c))
6
7 print area_triangulo(1, 3, 2.5)
8 print s
```

Cuando se ejecuta, aparece esto por pantalla:

```
1.1709371247
Traceback (innermost last):
  File "triangulo.py", line 8, in ?
    print s
NameError: s
```

La primera línea mostrada en pantalla es el resultado de ejecutar la línea 7 del programa. La línea 7 incluye una llamada a *area_triangulo*, así que el flujo de ejecución ha pasado por la línea 4 y s se ha creado correctamente. De hecho, se ha accedido a su valor en la línea 5 y no se ha producido error alguno. Sin embargo, al ejecutar la línea 8

se ha producido un error por intentar mostrar el valor de una variable inexistente: *s*. La razón es que *s* se ha creado en la línea 4 y se ha destruido tan pronto ha finalizado la ejecución de *area_triangulo*.

Las variables que sólo existen en el cuerpo de una función se denominan *variables locales*. En contraposición, el resto de variables se llaman *variables globales*.

También los parámetros formales de una función se consideran variables locales, así que no puedes acceder a su valor fuera del cuerpo de la función.

Fíjate en este otro ejemplo:

triangulo_9.py — triangulo.py

```
1 from math import sqrt
2
3 def area_triangulo(a, b, c):
4     s = (a + b + c) / 2.0
5     return sqrt(s * (s-a) * (s-b) * (s-c))
6
7 print area_triangulo(1, 3, 2.5)
8 print a
```

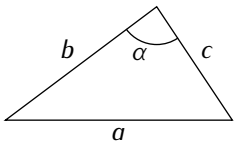
Al ejecutarlo obtenemos un nuevo error, pues *a* no existe fuera de *area_triangulo*:

```
1.1709371247
Traceback (innermost last):
  File "triangulo.py", line 8, in ?
    print a
NameError: a
```

¿Y cuándo se crean *a*, *b* y *c*? ¿Con qué valores? Cuando llamamos a la función con, por ejemplo, *area_triangulo*(1, 3, 2.5), ocurre lo siguiente: los parámetros *a*, *b* y *c* se crean como variables locales en la función y apuntan a los valores 1, 3 y 2.5, respectivamente. Se inicia entonces la ejecución del cuerpo de *area_triangulo* hasta llegar a la línea que contiene el **return**. El valor que resulta de evaluar la expresión que sigue al **return** se devuelve como resultado de la llamada a la función. Al acabar la ejecución de la función, las variables locales *a*, *b* y *c* *dejan de existir* (del mismo modo que deja de existir la variable local *s*).

Para ilustrar los conceptos de variables locales y globales con mayor detalle vamos a utilizar la función *area_triangulo* en un programa un poco más complejo.

Imagina que queremos ayudarnos con un programa en el cálculo del área de un triángulo de lados *a*, *b* y *c* y en el cálculo del ángulo α (en grados) opuesto al lado *a*.



El ángulo α se calcula con la fórmula

$$\alpha = \frac{180}{\pi} \cdot \arcsin\left(\frac{2s}{bc}\right),$$

donde *s* es el área del triángulo y arcsin es la función arco-seno. (La función matemática «arcsin» está definida en el módulo *math* con el identificador *asin*.)

Analiza este programa en el que hemos destacado las diferentes apariciones del identificador *s*:

area_y_angulo_3.py **area_y_angulo.py**

```
1  from math import sqrt, asin, pi
2
3  def area_triangulo(a, b, c):
4      s = (a + b + c) / 2.0
5      return sqrt(s * (s-a) * (s-b) * (s-c))
6
7  def angulo_alfa(a, b, c):
8      s = area_triangulo(a, b, c)
9      return 180 / pi * asin(2.0 * s / (b*c))
10
11 def menu():
12     opcion = 0
13     while opcion != 1 and opcion != 2:
14         print '1) Calcular área del triángulo'
15         print '2) Calcular ángulo opuesto al primer lado'
16         opcion = int(raw_input('Escoge opción: '))
17     return opcion
18
19 lado1 = float(raw_input('Dame lado a: '))
20 lado2 = float(raw_input('Dame lado b: '))
21 lado3 = float(raw_input('Dame lado c: '))
22
23 s = menu()
24
25 if s == 1:
26     resultado = area_triangulo(lado1, lado2, lado3)
27 else:
28     resultado = angulo_alfa(lado1, lado2, lado3)
29
30 print 'Escogiste la opción', s
31 print 'El resultado es:', resultado
```

Ejecutemos el programa:

```
Dame lado a: 5
Dame lado b: 4
Dame lado c: 3
1) Calcular área del triángulo
2) Calcular ángulo opuesto al primer lado
Escoge opción: 1
Escogiste la opción 1
El resultado es: 6.0
```

Hagamos una traza del programa para esta ejecución:

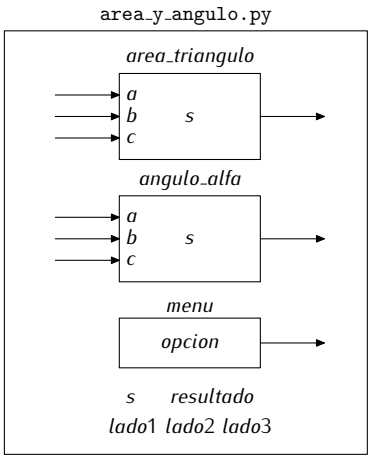
- La línea 1 importa las funciones *sqrt* (raíz cuadrada) y *asin* (arcoseno) y la variable *pi* (aproximación de π).
- Las líneas 3–5 «enseñan» a Python cómo se realiza un cálculo determinado al que denominamos *area_triangulo* y que necesita tres datos de entrada.
- Las líneas 7–9 «enseñan» a Python cómo se realiza un cálculo determinado al que denominamos *angulo_alfa* y que también necesita tres datos de entrada.
- Las líneas 11–17 definen la función *menu*. Es una función sin parámetros cuyo cometido es mostrar un menú con dos opciones, esperar a que el usuario escoja una y devolver la opción seleccionada.

- Las líneas 19–21 leen de teclado el valor (flotante) de tres variables: *lado*1, *lado*2 y *lado*3. En nuestra ejecución, las variables valdrán 5.0, 4.0 y 3.0, respectivamente.
- La línea 23 contiene una llamada a la función *menu*. En este punto, Python memoriza que se encontraba ejecutando la línea 23 cuando se produjo una llamada a función y deja su ejecución *en suspenso*. Salta entonces a la línea 12, es decir, al cuerpo de la función *menu*. Sigamos el flujo de ejecución en dicho cuerpo:
  - Se ejecuta la línea 12. La variable local *opcion* almacena el valor 0.
  - En la línea 13 hay un bucle **while**. ¿Es *opcion* distinto de 1 y de 2? Sí. Entramos, pues, en el bloque del bucle: la siguiente línea a ejecutar es la 14.
  - En la línea 14 se imprime un texto en pantalla (el de la primera opción).
  - En la línea 15 se imprime otro texto en pantalla (el de la segunda opción).
  - En la línea 16 se lee el valor de *opcion* de teclado, que en esta ejecución es 1.
  - Como el bloque del bucle no tiene más líneas, volvemos a la línea 13. Nos volvemos a preguntar ¿es *opcion* distinto de 1 y a la vez distinto de 2? No: *opcion* vale 1. El bucle finaliza y saltamos a la línea 17.
  - En la línea 17 se devuelve el valor 1, que es el valor de *opcion*, y la variable local *opcion* se destruye.
- ¿Qué línea se ejecuta ahora? La ejecución de la llamada a la función ha finalizado, así que Python regresa a la línea desde la que se produjo la llamada (la línea 23), cuya ejecución había quedado en suspenso. El valor devuelto por la función (el valor 1) se almacena ahora en una variable llamada *s*.
- La línea 25 compara el valor de *s* con el valor 1 y, como son iguales, la siguiente línea a ejecutar es la 26 (las líneas 27 y 28 no se ejecutarán).
- La línea 26 asigna a *resultado* el resultado de invocar a *area_triangulo* con los valores 5.0, 4.0 y 3.0. Al invocar la función, el flujo de ejecución del programa «salta» a su cuerpo y la ejecución de la línea 26 queda *en suspenso*.
  - Saltamos, pues, a la línea 4, con la que empieza el cuerpo de la función *area_triangulo*. ¡Ojo!, los parámetros *a*, *b* y *c* se crean como variables locales y toman los valores 5.0, 4.0 y 3.0, respectivamente (son los valores de *lado*1, *lado*2 y *lado*3). En la línea 4 se asigna a *s*, una nueva variable local, el valor que resulte de evaluar `(a + b + c)/2.0`, es decir, 6.0.
  - En la línea 5 se devuelve el resultado de evaluar *sqrt*`(s * (s-a) * (s-b) * (s-c))`, que también es, casualmente, 6.0. Tanto *s* como los tres parámetros dejan de existir.
- Volvemos a la línea 26, cuya ejecución estaba suspendida a la espera de conocer el valor de la llamada a *area_triangulo*. El valor devuelto, 6.0, se asigna a *resultado*.
- La línea 30 muestra por pantalla el valor actual de *s*... ¿y qué valor es ése? ¡Al ejecutar la línea 23 le asignamos a *s* el valor 1, pero al ejecutar la línea 4 le asignamos el valor 6.0! ¿Debe salir por pantalla, pues, un 6.0? No: la línea 23 asignó el valor 1 a la *variable global s*. El 6.0 de la línea 4 se asignó a la *variable s local a la función area_triangulo*, que ya no existe.
- Finalmente, el valor de *resultado* se muestra por pantalla en la línea 31.

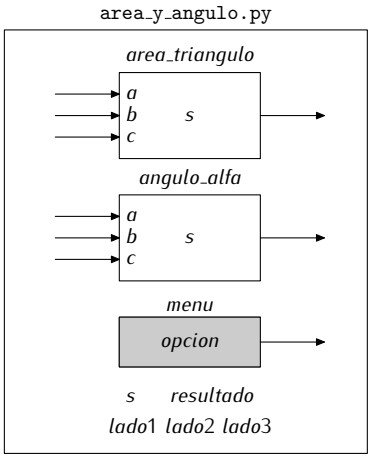
Observa que llamamos *s* a dos variables diferentes y que cada una de ellas «recuerda» su valor sin interferir con el valor de la otra. Si accedemos a *s* desde *area_triangulo*, accedemos a la *s* local a *area_triangulo*. Si accedemos a *s* desde fuera de cualquier función, accedemos a la *s* global.

Puede que te parezca absurdo que Python distinga entre variables locales y variables globales, pero lo cierto es que disponer de estos dos tipos de variable es de gran ayuda. Piensa en qué ocurriría si la variable *s* de la línea 4 fuese global: al acabar la ejecución de *area_triangulo*, *s* recordaría el valor 6.0 y habría olvidado el valor 1. El texto impreso en la línea 30 sería erróneo, pues se leería así: «`Escogiste la opción 6.0000`». Disponer de variables locales permite asegurarse de que las llamadas a función no modificarán accidentalmente nuestras variables globales, aunque se llamen igual.

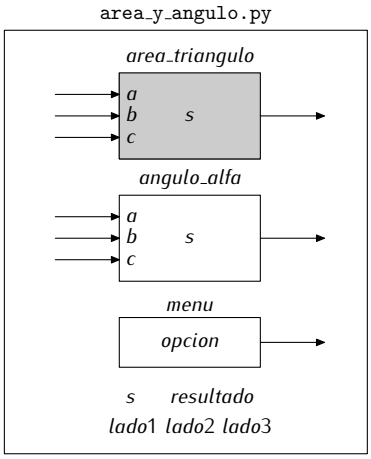
La siguiente figura ilustra la idea de que cada elemento del programa tiene un identificador que lo hace accesible o visible desde un *entorno* o *ámbito* diferente.



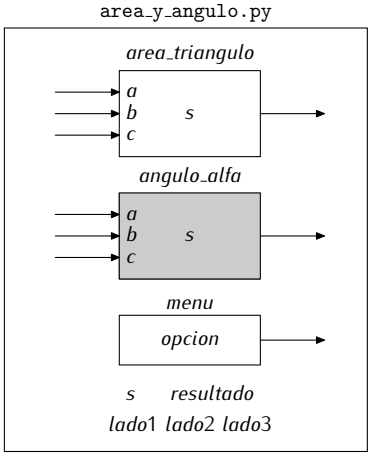
Cada función define un *ámbito local* propio: su cuerpo. Los identificadores de las variables locales sólo son visibles en su ámbito local. Por ejemplo, la variable *opcion* definida en la función *menu* sólo es visible en el cuerpo de *menu*. En este diagrama marcamos en tono gris la región en la que es visible esa variable:



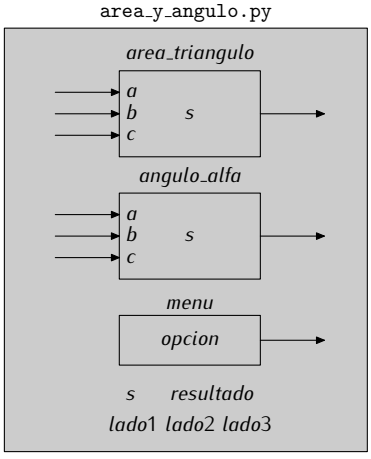
Fuera de la zona gris, tratar de acceder al valor de *opcion* se considera un error. ¿Qué pasa con las variables o parámetros de nombre idéntico definidas en *area_triangulo* y *angulo_alfa*? Considera, por ejemplo, el parámetro *a* o la variable *s* definida en *area_triangulo*: sólo es accesible desde el cuerpo de *area_triangulo*.



No hay confusión posible: cuando accedes al valor de *a* en el cuerpo de *area_triangulo*, accedes a su parámetro *a*. Lo mismo ocurre con la variable *s* o el parámetro *a* de *angulo_alfa*: si se usan en el cuerpo de la función, Python sabe que nos referimos esas variables locales:

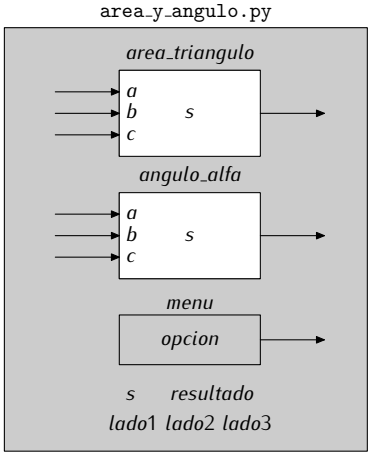


Hay un *ámbito global* que incluye a aquellas líneas del programa que no forman parte del cuerpo de una función. Los identificadores de las variables globales son visibles en el ámbito global *y desde cualquier ámbito local*. Las variables *resultado* o *lado*1, por ejemplo, son accesibles desde cualquier punto del programa (esté dentro o fuera del cuerpo de una función). Podemos representar así su «zona de visibilidad», es decir, su ámbito:



Hay una excepción a la regla de que las variables del ámbito global sean accesibles desde cualquier punto del programa: si el identificador de una variable (o función) definida

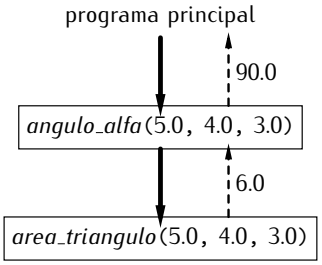
en el ámbito global se usa para nombrar una variable local en una función, la variable (o función) global queda «oculta» y no es accesible desde el cuerpo de la función. Por ejemplo, la variable local *s* definida en la línea 4 hace que la variable global *s* definida en la línea 23 no sea visible en el cuerpo de la función *area_triangulo*. Su ámbito se reduce a esta región sombreada:



En el programa, la función *angulo_alfa* presenta otro aspecto de interés: desde ella se llama a la función *area_triangulo*. El cuerpo de una función puede incluir llamadas a otras funciones. ¿Qué ocurre cuando efectuamos una llamada a *angulo_alfa*? Supongamos que al ejecutar el programa introducimos los valores 5, 4 y 3 para *lado*1, *lado*2 y *lado*3 y que escogemos la opción 2 del menú. Al ejecutarse la línea 28 ocurre lo siguiente:

- Al evaluar la parte derecha de la asignación de la línea 28 se invoca la función *angulo_alfa* con los argumentos 5, 4 y 3, con lo que la ejecución salta a la línea 8 y *a*, *b* y *c* toman los valores 5, 4 y 3, respectivamente. Python recuerda que al acabar de ejecutar la llamada, debe seguir con la ejecución de la línea 28.
    - Se ejecuta la línea 8 y, al evaluar la parte derecha de su asignación, se invoca la función *area_triangulo* con los argumentos 5, 4 y 3 (que son los valores de *a*, *b* y *c*). La ejecución salta, pues, a la línea 4 y Python recuerda que, cuando acabe de ejecutar esta nueva llamada, regresará a la línea 8.
        - En la línea 4 la variable *s* local a *area_triangulo* vale 6.0. Los parámetros *a*, *b* y *c* son *nuevas* variables locales con valor 5, 4, y 3, respectivamente.
        - Se ejecuta la línea 5 y se devuelve el resultado, que es 6.0.
    - Regresamos a la línea 8, cuya ejecución había quedado suspendida a la espera de conocer el resultado de la llamada a *area_triangulo*. Como el resultado es 6.0, se asigna dicho valor a la variable *s* local a *angulo_alfa*. Se ejecuta la línea 9 y se devuelve el resultado de evaluar la expresión, que es 90.0.
- Sigue la ejecución en la línea 28, que había quedado en suspenso a la espera de conocer el valor de la llamada a *angulo_alfa*. Dicho valor se asigna a *resultado*.
- Se ejecutan las líneas 30 y 31.

Podemos representar gráficamente las distintas activaciones de función mediante el denominado *árbol de llamadas*. He aquí el árbol correspondiente al último ejemplo:



Las llamadas se producen de arriba a abajo y siempre desde la función de la que parte la flecha con trazo sólido. La primera flecha parte del «programa principal» (fuera de cualquier función). El valor devuelto por cada función aparece al lado de la correspondiente flecha de trazo discontinuo.

EJERCICIOS

▶ 321 Haz una traza de *area_y_angulo.py* al solicitar el valor del ángulo opuesto al lado de longitud 5 en un triángulo de lados con longitudes 5, 4 y 3.

▶ 322 ¿Qué aparecerá por pantalla al ejecutar el siguiente programa?

triangulo_10.py — triangulo.py

```
1  from math import sqrt
2
3  def area_triangulo(a, b, c):
4      s = (a + b + c) / 2.0
5      return sqrt(s * (s-a) * (s-b) * (s-c))
6
7  s = 4
8  print area_triangulo(s-1, s, s+1)
9  print s
10 print a
```

▶ 323 La función *area_triangulo* que hemos definido puede provocar un error en tiempo de ejecución: si el argumento de la raíz cuadrada calculada en su última línea es un número negativo, se producirá un error de dominio. Haz que la función sólo llame a *sqrt* si su argumento es mayor o igual que cero. Si el argumento es un número negativo, la función debe devolver el valor cero. Detecta también posibles problemas en *angulo_alfa* y modifica la función para evitar posibles errores al ejecutar el programa.

▶ 324 Vamos a adquirir una vivienda y para eso necesitaremos una hipoteca. La cuota mensual m que hemos de pagar para amortizar una hipoteca de h euros a lo largo de n años a un interés compuesto del i por cien anual se calcula con la fórmula:

$$m = \frac{hr}{1-(1+r)^{-12n}},$$

donde $r = i/(100 \cdot 12)$. Define una función que calcule la cuota (redondeada a dos decimales) dados h, n e i. Utiliza cuantas variables locales consideres oportuno, pero al menos r debe aparecer en la expresión cuyo valor se devuelve y antes debe calcularse y almacenarse en una variable local.

Nota: puedes comprobar la validez de tu función sabiendo que hay que pagar la cantidad de 1 166.75 € al mes para amortizar una hipoteca de 150 000 € en 15 años a un interés del 4.75% anual.

▶ 325 Diseña una función que nos devuelva la cantidad de euros que habremos pagado finalmente al banco si abrimos una hipoteca de h euros a un interés del i por cien en n años. Si te conviene, puedes utilizar la función que definiste en el ejercicio anterior.

Nota: con los datos del ejemplo anterior, habremos pagado un total de 210 015 €.

▶ 326 Diseña una función que nos diga qué cantidad *de intereses* (en euros) habremos pagado finalmente al banco si abrimos una hipoteca de h euros a un interés del i por cien en n años. Si te conviene, puedes utilizar las funciones que definiste en los ejercicios anteriores.

Nota: con los datos del ejemplo anterior, habremos pagado un total de 210 015 − 150 000 = 60 015 € en intereses.

▶ 327 Diseña una función que nos diga qué tanto por cien del capital inicial deberemos pagar en intereses al amortizar completamente la hipoteca. Si te conviene, puedes utilizar las funciones que definiste en los ejercicios anteriores.

Nota: con los datos del ejemplo anterior, habremos pagado un interés total del 40.01% (60 015 € es el 40.01% de 150 000 €).

▶ 328 Diseña un *procedimiento* que muestre por pantalla la cuota mensual que corresponde pagar por una hipoteca para un capital de h euros al i% de interés anual durante 10, 15, 20 y 25 años. (Si te conviene, rescata ahora las funciones que diseñaste como solución de los ejercicios anteriores.)

▶ 329 Diseña un *procedimiento* que muestre por pantalla el capital total pagado al banco por una hipoteca de h euros al i% de interés anual durante 10, 15, 20 y 25 años. (Si te conviene, rescata ahora las funciones que diseñaste como solución de los ejercicios anteriores.)

Las variables locales también pueden contener valores secuenciales. Estudiemos un ejemplo de función con una variable local de tipo secuencial: una función que recibe una lista y devuelve otra cuyos elementos son los de la primera, pero sin repetir ninguno; es decir, si la función recibe la lista `[1, 2, 1, 3, 2]`, devolverá la lista `[1, 2, 3]`.

Empecemos por definir el cuerpo de la función:

sin_repetidos.py

```
1 def sin_repetidos(lista):
2     ...
```

¿Cómo procederemos? Una buena idea consiste en disponer de una nueva lista auxiliar (una variable local) inicialmente vacía en la que iremos insertando los elementos de la lista resultante. Podemos recorrer la lista original elemento a elemento y preguntar a cada uno de ellos si ya se encuentra en la lista auxiliar. Si la respuesta es negativa, lo añadiremos a la lista:

sin_repetidos_3.py — sin_repetidos.py

```
1 def sin_repetidos(lista):
2     resultado = []
3     for elemento in lista:
4         if elemento not in resultado:
5             resultado.append(elemento)
6     return resultado
```

Fácil, ¿no? La variable *resultado* es local, así que su tiempo de vida se limita al de la ejecución del cuerpo de la función cuando ésta sea invocada. El contenido de *resultado* se devuelve con la sentencia **return**, así que sí será accesible desde fuera. Aquí tienes un ejemplo de uso:

sin_repetidos_4.py — sin_repetidos.py

```
⋮
8 una_lista = sin_repetidos([1, 2, 1, 3, 2])
9 print una_lista
```

▶ 330 Diseña una función que reciba dos listas y devuelva los elementos comunes a ambas, sin repetir ninguno (intersección de conjuntos).

Ejemplo: si recibe las listas `[1, 2, 1]` y `[2, 3, 2, 4]`, devolverá la lista `[2]`.

▶ 331 Diseña una función que reciba dos listas y devuelva los elementos que pertenecen a una o a otra, pero sin repetir ninguno (unión de conjuntos).

Ejemplo: si recibe las listas `[1, 2, 1]` y `[2, 3, 2, 4]`, devolverá la lista `[1, 2, 3, 4]`.

▶ 332 Diseña una función que reciba dos listas y devuelva los elementos que pertenecen a la primera pero no a la segunda, sin repetir ninguno (diferencia de conjuntos).

Ejemplo: si recibe las listas `[1, 2, 1]` y `[2, 3, 2, 4]`, devolverá la lista `[1]`.

▶ 333 Diseña una función que, dada una lista de números, devuelva otra lista que sólo incluya sus números impares.

▶ 334 Diseña una función que, dada una lista de nombres y una letra, devuelva una lista con todos los nombres que empiezan por dicha letra.

▶ 335 Diseña una función que, dada una lista de números, devuelva otra lista con sólo aquellos números de la primera que son primos.

▶ 336 Diseña una función que, dada una lista de números, devuelva una lista con todos los pares de números que podemos formar con uno de la primera lista y otro de la segunda. Por ejemplo, si se suministran las listas `[1, 3, 5]` y `[2, 5]`, la lista resultante es

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

▶ 337 Diseña una función que, dada una lista de números, devuelva una lista con todos los pares de números *amigos* que podemos formar con uno de la primera lista y otro de la segunda.