Introducción a la Programación
Ejemplos de código en Python

Programa repetidor

Problema

Escribir un programa pida un texto (entrada) y lo imprimo tres veces en pantalla. Por ejemplo, para el input

```
Beetlejuice
```

el programa debe imprimir

```
Beetlejuice
Beetlejuice
Beetlejuice
```

Solución

El siguiente programa pide un texto (entrada) y lo imprime tres veces en pantalla:

```
x = input()
print(x)
print(x)
print(x)
```

De lo anterior, la primera línea solicita un **input** al usuario y guarda ese dato, que es un texto, en la variable **x**. Luego, las tres líneas siguientes invocan a **print**, que es una función que imprime en pantalla, pasándole el contenido de la variable **x**.

Teoría

Para abordar este problema, tenemos que saber:

- Usar **variables**. Las variables son lugares en memoria donde dejaremos datos. Les colocaremos nombres.
- Para crear una variable, simplemente realizamos una **asignación**. Ej: **var = valor**.
- La instrucción **input()** pide un texto al usuario (al ambiente de ejecución en verdad) y el valor suplido es devuelto por **input()** a Python.
- La instrucción **print()** imprime en pantalla (o en el ambiente de ejecución) lo que se le entregue entre los paréntesis. Ejemplos:
  - **print(10)** imprime el número entero 10 en pantalla,
  - **print(-5, "hola", True, 0.1)** imprime el entero -5 (entero o **int**), la cadena *hola* (las cadenas o strings, **str**, representan textos), el booleano True (verdadero) y el decimal 0.1 (decimal o **float**),
  - **print()** imprime una línea vacía.

La fórmula aproximada

Problema

Escriba un programa que reciba un número decimal (**float**) y que imprima su evaluación en la siguiente fórmula:

$$f(x)=\left(\frac{1+x/2+x^2/8}{1-x/2+x^2/8}\right).$$

Por ejemplo, para el input

```
.5
```

se genera el output

```
1.64
```

Otro ejemplo, para el input

```
-.39
```

se genera el output

```
0.6787512484426641
```

Solución

El siguiente código cumple con lo solicitado:

```
x = float( input() )
y = (1 + x/2 + x**2 / 8) / (1 - x/2 + x**2 / 8)
print(y)
```

La primera línea solicita el número decimal desde el input y lo guarda en la variable **x**. La segunda línea evalúa la fórmula, usando el valor guardado en **x**, y el resultado lo guarda en la variable **y**. Finalmente, la tercera línea imprime el valor guardado en **y**.

Revisemos cómo se evalúa la fórmula. Como vemos, hay paréntesis; Python irá a determinar cuánto resulta cada paréntesis antes de evaluar la división entre ellos.

Veamos cómo se evalúa cada expresión polinomial. Centrémonos en

```
1 + x/2 + x**2 / 8
```

Aquí vemos que hay una potencia (**x**2**), que es lo que primero se ejecuta, por prioridad. Luego siguen las divisiones, en **x/2** y **x**2/8**. Finalmente, siguen las adiciones.

Si pusiéramos paréntesis, la fórmula anterior se podría leer así:

```
1 + (x/2) + ((x**2)/8)
```

Aunque siendo más estricto, hay evaluación de izquierda a derecha entre operaciones de igual prioridad, así que:

```
((1 + (x/2)) + ((x**2)/8))
```

Teoría

Para recibir números decimales, debemos usar código como

```
x = float( input() )
```

que piden un **input** de texto, lo convierten a decimal (*punto flotante*) con **float** y luego guardan ese valor en una variable, que sería **x** en este caso.

Para escribir fórmulas. También debemos saber cómo evaluar operaciones matemáticas con Python. Al respecto. las siguientes son las principales operaciones matemáticas que conciernen al curso:

```
10 + 25    # adición
10 - 7     # sustracción
10 * 9     # multiplicación
81 / 9     # división: siempre genera un número decimal!
81 // 9    # división entera: genera entero si divisor y dividendo son enteros
37 % 5     # resto: residuo de la división entera
5 ** 20    # potencia: aquí es 5 elevado a 20
```

También debemos saber que las operaciones tienen prioridad, o sea, se evalúan en orden preferente:

- La potencia tiene la más alta prioridad, por lo que se evalúa primero;
- Luego siguen *, /, //, % (se evalúan de izquierda a derecha);
- Luego siguen + y -.

Para asegurar que las operaciones se ejecuten en un orden que nosotros queramos, usamos **paréntesis redondos**. Por ejemplo:

```
( 10 + 5 ) / ( 10 - 5 )
```

resulta en 3, porque equivale a 15/5. Sin embargo

```
10 + 5 / 10 - 5
```

resulta en 4.5, porque $10 + 5/10 - 5$ vale $10 + 0.5 - 5 = 4.5$.

Los paréntesis son muy importantes en Python. **Cada vez que se abre un paréntesis, hay que cerrarlo**.

El menor de tres números

Problema

Escriba un programa que reciba tres números enteros y que luego imprima en pantalla el valor del menor de éstos. Por ejemplo, para el input

```
100
40
75
```

el programa deberá imprimir

```
40
```

Por cierto, es posible que no haya números repetidos. Para el input

```
100
100
101
```

el programa deberá imprimir

```
100
```

Aquí no hubo sólo un mínimo, sino que hubieron dos. Su programa deberá funcionar bien en estos casos.

Solución 1

El código siguiente crea tres variables, **a**, **b** y **c**, con los números enteros recibidos y luego decide cuál es el menor usando **if-elif-elif**:

```
a = int( input() )
b = int( input() )
c = int( input() )
if a <= b and a <= c:
     print(a)
elif b <= a and b <= c:
     print(b)
elif c <= a and c <= b:
     print(c)
```

Cuando evaluamos si **x<=y and x<=z**, estamos preguntando si **x** es el mínimo. Entonces, la estructura **if-elif-elif** usada primero evalúa si **a** es mínimo, luego si **b** es mínimo y, finalmente, si **c** es mínimo.

Solución 2

Es posible mejorar el código anterior al simplificar las condiciones y usar una estructura **if-elif-else**:

```
a = int( input() )
b = int( input() )
c = int( input() )
if a <= b and a <= c:
     print(a)
elif b <= c:
     print(b)
else:
     print(c)
```

Hubo ahorro en las consultas. Los casos que se consideraron son:

- ¿Es **a** el mínimo? Si así es, imprimir **a**;
- Si **a** no es mínimo, ¿es **b** mínimo? Si así es, imprimir **b**;
- Si ni **a** ni **b** son mínimos, entonces imprimir **c**, pues es mínimo.

Solución 3

El código siguiente crea tres variables, **a**, **b** y **c**, con los números enteros recibidos y luego decide cuál es el menor usando **if-else**:

```
a = int( input() )
b = int( input() )
c = int( input() )
if a <= b:
     if a <= c:
          print(a)
     else:
          print(c)
else:
     if c <= a:
          print(c)
     else:
          print(b)
```

Esta solución es menos directa que las anteriores, pero utiliza condiciones más sencillas, sin hacer uso de los operadores booleanos como **or**, **and** y **not**.

Podemos describir la estructura de decisiones así:

- ¿Es **a** menor que **b**?
  - Si así es, entonces el mínimo será el menor entre **a** y **c**. ¿Es **a** menor que **c**?
    - Si así es, **a** es el mínimo;
    - De lo contrario, **c** es el mínimo.
  - Si a no es menor que **b**, entonces el mínimo está entre **b** y **c**. ¿Es **b** menor que **c**?
    - Si así es, **b** es el mínimo;
    - De lo contrario, **c** es el mínimo.

Teoría

Para recibir números enteros, debemos usar lineas de código como

```
x = int( input() )
```

que piden un **input** de texto, lo convierten a entero con **int** y luego guardan ese valor en una variable, que sería **x** en este caso.

Para ejecutar código condicional. Lo siguiente a saber, es que hay que usar instrucciones para ejecución condicional o alternativa para resolver este problema. Esto se logra haciendo uso de la instrucción if, cuya sintaxis es como sigue:

```
if condicion:
     bloque indentado a ejecutar si
     la condición se cumplió
```

Aquí la condición se puede escribir usando comparadores:

- == para igualdad (¿valen lo mismo?),
- != para distinto,
- < y > para *menor que* y *mayor que*, respectivamente,
- <= y >= para *menor o igual que* y para *mayor o igual que*, respectivamente.

Además de lo anterior, es posible mezclar condiciones:

- **and** exige que ambas condiciones se cumplan (ej. que el valor de la variable **num** sea un número positivo menore a 10 se puede escribir como **num > 0 and num < 10**),
- **or** exige que alguna de las condiciones se cumpla (ej. que **num** sea negativo o mayor a 100 se puede escribir como **num < 0 or num > 100**).

Por último, es posible utilizar:

- **not**, que niega la condición siguiente (**not x<y** es equivalente a exigir **x>=y**).

Todas estas formas de comparar y combinar valores booleanos sirven para escribir condiciones.

Adicionalmente, podemos tener un bloque **else** (en inglés, *de lo contrario*) para cuando la condición no se cumple:

```
if condición:
    bloque indentado a ejecutar si
    la condición se cumplió
else:
    bloque indentado a ejecutar si
    la condición no se cumplió
```

Finalmente, es posible colocar bloques **elif** (*portmanteau* de los términos **else** e **if**) para exigir condiciones intermedias luego del **if** y antes del **else**, si es que hay uno:

```
if condición1:
    bloque indentado a ejecutar si la condición se cumplió
elif condición2:
    bloque indentado a ejecutar si la condición1 no se cumplió, pero
    sí condición2
elif condición3:
    bloque indentado a ejecutar si la condición1 y la condición2 no se
    cumplieron, pero sí la condición3
else:
    bloque indentado a ejecutar si la condición no se cumplió
```

Se pueden usar tantos bloques **elif** como se estime conveniente.

Es importante saber que:

- La indentación debe ser **consistente**, nada de reemplazar espacios por tabs o cambiarlos arbitrariamente en la línea siguiente del bloque
- Cuando se deshace la indentación, Python asume que el bloque indentado ha terminado
- Es posible colocar nuevas instrucciones **if** dentro del bloque indentado, lo que inicia bloques con mayor indentación

Programa repetidor variable

Problema

Escriba un programa que pida un texto y luego un número entero (int), para luego repetir el texto tantas veces como el número solicitado. Por ejemplo, para el input

```
izquierda, derecha
5
```

el programa debe imprimir

```
izquierda, derecha
izquierda, derecha
izquierda, derecha
izquierda, derecha
izquierda, derecha
```

Solución 1, for

El siguiente programa imprime el texto tantas veces como el número solicitado:

```
txt = input()
num = int( input() )
for i in range(num):
    print( txt )
```

Explicando, la línea **txt = input()** pide un texto al usuario mediante la función **input()**, parar guardado en la variable **txt**. La línea siguiente, **num = int(input())** pide un texto al usuario mediante la función **input()**, luego convierte este texto a número entero mediante la función **int()** y luego guarda el resultado en la variable **num**.

Luego viene un bucle o repetición. La línea **for i in range(num):** se puede traducir a "por cada valor de **range(num)**, que guardamos en la variable **i**, hacer:". Aquí, **range(num)** genera los números 0, 1, 2, ..., hasta **num**. La instrucción **for** saca un valor de **range(num)**, lo guarda en la variable **i**, luego ejecuta el *bloque indentado que viene inmediatamente a continuación* y vuelve a pedir un valor a **range(num)**, repitiendo el proceso hasta que no queden más valores.

La siguiente línea es el *bloque indentado*. Simplemente hace **print(txt)**, omitiendo la variable **i**, cuyo contenido no nos interesa, pues sólo queremos imprimir **txt**.

Como **for i in range(num)** extrerá **num** valores distintos, **print(txt)** se repetirá **num** veces.

Solución 2, while

El programa anterior también se puede escribir usando **while**:

```
txt = input()
num = int( input() )
i = 0
while i < num:
    print( txt )
    i = i+1
```

Aquí definimos **i=0** para inicializar la variable **i** con el valor **0**, como ocurriría con la primera iteración del **for** de la solución anterior. Luego, se exige **i<num** y se realiza **i=i+1** dentro del bloque indentado, lo que hará que **i** asuma los valores 0, 1, 2, ..., **num**-1. Cuando **i=num**, entonces **i<num** no se cumplirá y se saldrá del bucle.

Solución 3, while

Podemos también usar una lógica algo distinta para el **while**:

```
txt = input()
num = int( input() )
while num > 0:
     print( txt )
     num = num-1
```

Ahora, sólo la variable **num** cambia. Esta variable irá decreciendo en 1 hasta llegar a 0, que es cuando **num>0** no se cumple, por lo que **while** deja de ejecutar el bloque indentado y sale. El bloque indentado se repite **num** veces.

Teoría

Este problema exige saber de bucles o repeticiones en Python. A saber, hay de dos tipos.

while. Los bucles **while** tienen por notación:

```
while condición:
     bloque indentado que se ejecutará
     mientras la condición sea cierta
```

El bucle **while** es como un **if**. Sin embargo, en vez de ejecutar el bloque indentado cero o una vez, ahora se puede ejecutar un número indeterminado de veces.

Ojo con la indentación, con la instrucción **while** y con el dos punto (**:**) al final de la línea.

for. El otro tipo de bucle es **for**, que obtiene un valor de un *generador*, lo asigna a una variable y ejecuta el bloque indentado; esto lo repite hasta que el generador no emita más valores.

La sintaxis de la instrucción **for** es:

```
for var in generador:
     bloque indentado que se ejecutará mientras el generador entre valores;
     cada valor generado quedará en la variable var
```

Aquí el generador es **range**. Este generador funciona así:

- **range(N)** genera los números 0, 1, 2, ..., N-1;
- **range(N,M)** genera los números N, N+1, N+2, ..., M-1;
- **range(N,M,k)** genera los números N, N+k, N+2k, ..., Ñ < M.

Otros generadores:

- **strings**: generan los caracteres del texto;
- **listas**: generan los elementos de la lista;
- **tuplas**: generan los elementos de la tupla;
- **set**: generan los elementos del conjunto;
- **dict**: generan las *llaves* del diccionario;
- **file**: generan las líneas del archivo.

Promediar números

Problema

Escriba un programa que promedie los números decimales recibidos desde el input, hasta recibir la palabra **fin**. Ejemplo, a partir del input

```
30.0
100
50
fin
```

el output debe ser

```
60.0
```

Solución

El siguiente código responde a este problema:

```
# 1. creamos las variables que usaremos para calcular el promedio
suma = 0
num = 0

# 2. procesamos el input y actualizamos las variables
x = input()
while x != "fin":
     suma += float(x)
     num += 1
     x = input()

# 3. generamos la respuesta
if num > 0:
     print( suma / num )
else:
     print( 0.0 )
```

Aquí es muy importante entender la ubicación de las líneas que dicen **x=input()**. Una antecede al bucle **while**, la otra está al final de su bloque indentado.

Calcular el factorial

Problema

Para un número entero n, su factorial, denotado como n!, se define como

$$n! = 1, \quad \text{para } n \leq 1$$

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n, \quad \text{para } n > 1$$

Escriba un programa que reciba un número entero de input y que imprima su factorial. Por ejemplo, para el input

```
4
```

el programa deberá imprimir

```
24
```

Otro ejemplo, para el input

```
6
```

su programa deberá imprimir

```
720
```

Otro ejemplo más, para el input

```
-2
```

su programa deberá imprimir

```
1
```

Solución 1, for

```
n = int( input() )
if n <= 1:
     print( 1 )
else:
     fact = 1
     for k in range(1,n+1):
           fact = fact * k
     print( fact )
```

Dos observaciones:

- Nunca deje ocurrir que **fact=0**;
- **range(1,n+1)** debe tener segundo término **n+1**, para que vaya desde **1** hasta **n**.

Solución 2, for

Este código aprovecha que **range(a,b)** no generará números si **b <= a**:

```
n = int( input() )
fact = 1
for k in range(1,n+1):
     fact = fact * k
print( fact )
```

Solución 3, while

Ambas soluciones anteriores se pueden escribir con **while**. Por ejemplo, la Solución 2 quedaría como:

```
n = int( input() )
fact = 1
k = 1
while k < n+1:
     fact = fact * k
     k = k + 1
print( fact )
```

Solución 4, while

Podemos también usar un esquema de números en reversa para calcular la factorial:

```
n = int( input() )
fact = 1
while n>1:
     fact = fact * n
     n = n - 1
```

Notemos que **n! = 1·2·3·...·(n-1)·n = n·(n-1)·...·3·2·1**. La multiplicación es conmutativa: *el orden de los factores no altera el producto*.

Similitud de parlamentarios

Problema

Representaremos las votaciones de dos parlamentarios, digamos, diputados, usando strings que compuestos de caracteres "+", "-" y "A", para designar voto A Favor ("+"), En Contra ("-") y Abstención o Ausencia ("A").

Los símbolos anteriores, dentro de un string, denotan su comportamiento en la cámara durante un período de tiempo. Para dos parlamentarios, sus strings podrían ser

```
+-A+
+-+-
```

Estos strings se leen así. Para el proyecto 1, ambos parlamentarios votaron A Favor. Para el proyecto 2, ambos parlamentarios votaron En Contra. Para el proyecto 3, el primer parlamentario se abstuvo o ausentó, mientras que el segundo votó A Favor. Para el proyecto 4, el primer parlamentario votó A Favor, mientras que el segundo votó En Contra.

Escribir un programa que reciba dos strings de votaciones, para dos parlamentarios en un mismo periodo de tiempo, y que calcule cuántas veces los parlamentarios hicieron lo mismo.

Ejemplo, para el input

```
+++++---A--
+---++-A++-
```

el output debe ser

```
4
```

puesto que votan A Favor (+) dos veces y En Contra (-) dos veces también.

Otro ejemplo, para el input

```
++++--+++A----
+++AAA-+AAA---
```

el output debe ser

```
8
```

puesto que votan A Favor (+) cuatro veces, En Contra (-) tres veces y se Ausentan o Abstienen (A) una vez.

Solución

Pese a la descripción del problema (que es realista, en cuanto métodos muy similares se usan en ciencia política), la solución simplemente pasa por contar cuándo los strings son iguales en el mismo lugar. Por eso, la solución es:

```
par1 = input()
par2 = input()
puntos = 0
for i in range( len( par1 ) ):
    if par1[i] == par2[i]:
        puntos = puntos + 1
print( puntos )
```

Teoría

Este problema exige entender de strings y colecciones similares, como listas y tuplas, que tienen largo y posiciones.

Sea X un string, lista o tupla. Entonces:

- len(X) entrega el largo de X;
- X[i] nos entrega el valor en la posición i-ésima,
  - si X es un string, X[i] es un caracter (ej. letra, número, espacio, etc.),
  - si X es una lista, X[i] es el elemento guardado en ese lugar,
  - si X es una tupla, X[i] es el elemento guardado en ese lugar;
- X[0] nos entrega el valor en la posición 0 (la primera);
- X[-1] nos entrega el valor en la última posición;
- X[-2] nos entrega el valor en la penúltima posición;
- X[-3] nos entrega el valor en la antepenúltima posición, etc.

Similitud de parlamentarios 2

Problema

Seguimos con el problema anterior, representando las votaciones de dos parlamentarios, usando strings que compuestos de caracteres "+", "-" y "A", para designar voto A Favor ("+"), En Contra ("-") y Abstención o Ausencia ("A").

Ahora, debemos calcular un puntaje distinto. Por proyecto,

- Si ambos votan A Favor (+) o En Contra (-), suman 1 punto
- Si votan opuesto, pierden 1 punto; y
- Si alguno se ausentó o abstuvo (A), no suman puntos.

Ejemplo, para el input

```
++++++---A--
+---++-A++-
```

el output debe ser

```
-1
```

puesto que votan igual (++, --) cuatro veces y votan opuesto (-+, +-) cinco veces, resultando en un total de 4 - 5 = -1 puntos.

Otro ejemplo, para el input

```
++++--+++A----
+++AAA-+AAA---
```

el output debe ser

```
6
```

puesto votan igual (++, --) siete veces y votan opuesto (-+, +-) una vez, lo que resulta en un neto de 7 - 1 = 6 puntos.

Solución 1

Para solucionar este problema, es necesario detectar todos los casos que interesan.:

```
par1 = input()
par2 = input()
puntos = 0
for i in range( len( par1 ) ):
     if par1[i]=='+' and par2[i]=='+':
          puntos = puntos + 1
     if par1[i]=='-' and par2[i]=='-':
          puntos = puntos + 1
     if par1[i]=='+' and par2[i]=='-':
          puntos = puntos - 1
     if par1[i]=='-' and par2[i]=='+':
          puntos = puntos - 1
print( puntos )
```

Notemos que aquí no necesitamos usar **elif**, porque los casos no tienen sobrelape.

Solución 2

Esta solución tiene una estructura de **if**s más sencilla, porque se simplifican las comparaciones:

```
par1 = input()
par2 = input()
puntos = 0
for i in range( len( par1 ) ):
     par = par1[i] + par2[i]
     if par == "++" or par == "--":
          puntos = puntos + 1
     if par == "-+" or par == "+-":
          puntos = puntos - 1
print( puntos )
```

Solución 3

Otra solución alternativa, ahora usando listas:

```
par1 = input()
par2 = input()
puntos = 0
for i in range( len( par1 ) ):
     par = par1[i] + par2[i]
     if par in [ "++", "--" ]:
          puntos = puntos + 1
     if par in [ "-+", "+-" ]:
          puntos = puntos - 1
print( puntos )
```

Teoría

Concatenación. Como novedad, las Soluciones 2 y 3 hicieron uso de la concatenación, que es la *suma* que se da entre strings. Por ejemplo:

- **"hola" + "mundo"** resulta en **"holamundo"**,
- **"" + "hola" + " " + "mun" + "do"** resulta en **"hola mundo"**,
- **"dia" + 1** resulta en **error**.

A saber, no podemos concatenar strings con números u otro tipo de datos (listas, etc). Arroja una excepción, porque no tiene sentido concatenar un string con otro tipo de dato.

Listas. Las listas son colecciones de valores que podemos definir como

```
L = [1, 2, 'hola', False]
```

y podemos acceder a sus elementos usando índices, por ejemplo

```
print( L[0] )
```

imprime **1**, mientras que

```
print( L[1] )
```

imprime **2**, y

```
print( L[-1] )
```

imprime **False**.

Pertenencia (in). La Solución 3 usó la instrucción **in** dentro de una condición lógica. El uso de **in** es como sigue. Para **x in y**,

- Si **y** es una tupla o lista, **x in y** verifica si **x** es uno de los elementos de **y**, por ejemplo:
  - **1 in [1, 2, 3, 4]** evalúa a **True**,
  - **10 in [1, 2, 3, 4]** evalúa a **False**,
  - **'Mario' in ['Mario', 'Luigi', 'Peach', 'Toad']** evalúa a **True**,
  - **'Bowser' in ['Mario', 'Luigi', 'Peach', 'Toad']** evalúa a **False**;
- Si **y** es un string, **x in y** verifica si **x** es un *substring* de **y**, por ejemplo,
  - **"cuela" in "Escuela"** retorna **True**,
  - '**Goku**' **in** '**Sengoku**' evalúa a **False** (la '**G**' evita el calce exacto aquí).

Por cierto, **in** es muy práctico con strings. Por ejemplo, podemos probar rápidamente:

- ¿Es el caracter **x** un dígito? **x in '0123456789'** (vale si **len(x)==1**);
- ¿Es el caracter **x** una vocal? **x in 'æiouÆIOU'** (vale si **len(x)==1**).

Función que obtiene el coseno de los números de una lista

Problema

Defina la siguiente función, liscos(L), que reciba de parámetro una lista L de números y que retorne una nueva lista, digamos M, cuyos elementos correspondan al coseno de cada número de la lista.

Por ejemplo, el código

```
L = [0, 1, 2, 0.5]
M = liscos(L)
print(M)
```

debe imprimir

```
[1.0, 0.5403023058681398, -0.4161468365471424, 0.8775825618903728]
```

Para evaluar el coseno de un número, use la función **cos** que viene en la biblioteca o módulo **math**.

Solución

```
import math

def liscos(L):
    M = []
    for x in L:
        cos_x = math.cos( x )
        M.append( cos_x )
    return M
```

Teoría

Este problema requiere saber algunas cosas acerca de listas y de funciones que no hemos visto en esta guía.

Creación de una lista vacía. Como vemos, hay una línea que dice

```
M = []
```

Esta línea crea una lista sin elementos, o sea, vacía.

Alternativamente, el siguiente código también define **M** como una lista vacía:

```
M = list()
```

for sobre una lista. Podemos notar la línea que dice

```
for x in L:
```

Esta línea sirve para iterar sobre cada elemento de la lista **L**, dejando esos números en la variable **x**.

Usando append con una lista. Finalmente, podemos agrandar una lista usando **append** (en castellano, *anexar*). Esta función agrega un elemento al final de una lista. Su notación es así:

```
lista.append( valor )
```

Esto pondrá **valor** al final de la lista llamada **lista**.

La función **append** se usó para colocar los cosenos de los números en la lista contenida en la variable **M**:

```
M.append( cos_x )
```

Módulos y funciones. Para resolver este problema, es necesario saber cómo:

- Importar módulos,
- Usar funciones de un módulo importado,
- Definir funciones.

Importar y usar módulos. Es posible importar módulos (a veces llamados paquetes, bibliotecas o librerías) para usar sus funciones, constantes, etc, en nuestros programas de Python. En este caso, debemos importar **math**, lo que hacemos como:

```
import math
```

Es buena costumbre importar módulos al principio de nuestros programas. En la solución al problema de arriba, **import math** es la primera línea de código siguiendo esta buena costumbre.

Una vez importado el módulo, podemos usar sus funciones al escribir

```
nombre_modulo.nombre_funcion( parametros )
```

En este caso, debemos usar la función coseno, que llamamos con

```
math.cos(x)
```

como dice la solución.

Definir funciones. También podemos definir funciones propias. Esto lo hacemos como

```
def nombre_funcion( param1, param2, etc ):
    bloque indentado de codigo que calcula lo que hace la funcion,
    usando las variables param1, param2, etc, que se definen en la
    llamada de función.
    return resultado
```

Es muy importante la instrucción **def**, que inicia la definición (firma) de una función. Esa línea debe terminar en dos puntos (**:**), porque le sigue un bloque indentado con el código que hace funcionar a la función (valga la redundancia).

La instrucción **return** sirve para devolver el resultado al trozo de código que llama a la función. No solo eso, además termina la ejecución de la función; si la función tiene código después del **return**, ese código no se ejecutará.