

FUNDAMENTOS DE PROGRAMACIÓN

Guía completa de introducción a la programación con Python 3



Capítulo 1: Tipos de datos, variables, entrada y salida

Función print()

La función `print()` permite crear la salida que se mostrará cuando corramos nuestro programa. Dentro de los paréntesis de la función `print` van los argumentos que se desean imprimir por pantalla. Para escribir texto se puede usar comillas dobles o simples.

Ejemplos de como usarla:

Código

```
1 print('bienvenidos')
2 print("buena suerte")
3 print('exito')
4 print("hola mundo!")
```

Salida

```
bienvenidos
buena suerte
exito
hola mundo!
```

Tipo de datos

Por ahora conoceremos los más básicos, `int` para números enteros, `float` para números decimales, `string` para texto y lógico (`bool`) que consisten en `True` (verdadero) y `False` (falso). Los `strings` se crean con comillas dobles o simples y se trata de texto que puede contener cadenas de caracteres como letras, símbolos y números. En Python al momento de crear una variable se debe asignar un nombre y un valor, el cual puede ser el tipo de datos que queramos.

La función `print()` también sirve para imprimir el contenido de variables.

Código

```
1 a = 74
2 b = True
3 c = "Hola"
4 d = 31.13
5 print(a,b,c,d)
```

Salida

```
74 True Hola 31.13
```

El nombre de una variable se puede escoger libremente siempre y cuando no comience con un número y no contengan lo siguiente: !, ?, #, \$,% , ; . Sin embargo pueden llevar guión.

A la variable "a" se le asignó un valor entero (int), a "b" un valor lógico también llamado booleano, en Python los valores booleanos son (True y False), a "c" se le asignó un string y a d se le asignó un float (número con decimales).

Notar que cuando se decidió imprimir estos valores, se separa con comas dentro de la función print, esto es para que salgan separados, además si no se separa con comas y se corre el programa, saldrá un error, pues la sintaxis es incorrecta, cuando queramos imprimir varias variables de una sola vez, se deben separar con comas.

Función input()

Si queremos que el usuario indique el valor de una variable necesitaríamos usar la función `input()`, precedida del tipo de dato a asignarle, (si no se le precede ningún tipo de dato, se recibirá como string por defecto). Esta función espera a que el usuario ingrese por teclado lo que desee y luego de oprimir la tecla enter el programa sigue su ejecución.

Dentro de la función `input()` podemos escribir texto entre comillas simples o dobles para indicarle al usuario el tipo de dato que queremos que ingrese, (esto es opcional) . Ejemplo:

Código

```
1 num1 = int(input('Ingrese un número entero: '))
2 tex = input('Ingrese una palabra, letra, frase o número: ')
3 num2 = float(input('Ingrese un número flotante (decimal): '))
```

Salida

```
Ingrese un número entero: 91
Ingrese una palabra, letra, frase o número: avión
Ingrese un número flotante (decimal): 34.174
```

Para usar la función `input()` que permite que el usuario ingrese por teclado alguna entrada debemos crear la variable que contendrá dicha información e igualamos dicha variable a la función `input()`.

La función `input()` puede estar dentro de la función `int()`, de esta forma el usuario puede ingresar números enteros, si está dentro de la función `float()`, sirve para recibir números flotantes (con decimales). En este ejemplo, como valor entero se ingresó 91, por tanto la variable `num1` almacena ese valor. De forma parecida sucede con las demás variables creadas, `tex` guardará un string (texto), y `num2` un número flotante (decimal). Finalmente se imprime el valor de cada variable separada por espacios.

Comentarios

Usando el “#” podemos comentar nuestro código en Python. Se usa los comentarios para que los lectores de nuestro código se le facilite entender lo que hicimos., la presencia de comentarios no afectará en nada a la ejecución nuestro código ya que sólo se ejecutarán las instrucciones.

```
1 # este es un comentario
2 a = 4 # asignamos 4 a la variable a
3 print(a) # imprimimos la variable a
```

Si colocamos un “#” antes de una instrucción se convertirá en un comentario y la hora de ejecutarse el código la instrucción será ignorada.

```
1  a = 5
2  # print(a)
```

También podemos comentar múltiples líneas fácilmente si encerramos lo que queremos comentar con tres comillas simples.

```
1  '''
2  a = 4; b = 7; t=input()
3  print(a)
4  print(b) '''
```

Capítulo 2: Operadores matemáticos en Python 3

Operadores matemáticos

En Python trabajaremos con los siguientes operadores matemáticos: “+” (suma), “-” (resta), “*” (multiplicación), “/” (división), “//” (división entera), “%” (módulo) “**” (potenciación).

Aquí algunos ejemplos:

Nota: En python usamos el ‘;’ para poder escribir varias instrucciones en una misma línea.

Operador suma y resta:

```
1  a = 5; b = 3
2  c = a+b # c vale 8
3  c = a - b # ahora c vale 2
```

Operador multiplicación:

```
1  a = 5 * 3 # a vale 15
2  b = 1.5 ; c = 2
3  d = b*c # d vale 3
```

Operador de división (normal) y división entera:

```
1  x = 4 ; d = 2
2  z = x / d # z = 2.0
3  t = 5 ; d = 7
4  z = d // t # t vale 1
```

Operador módulo: Retorna el residuo de la división entera de sus operandos

```

1  x = 16 % 2
2  # x vale 0
3  a = 3; x = 8 % a
4  # ahora x vale 2

```

Operador para la potenciación:

```

1  a = 2 ** 4 ; print(a)
2  # a = 16
3  b = 4 ** (1/2); print(b)
4  # raíz cuadrada de 4
5  c = 125 ** (1/3) ; print(c)
6  # raíz cúbica de 125

```

Cuando tengamos una expresión matemática en donde tengamos varios operadores debemos tener cuidado de agrupar bien las operaciones que vayamos a escribir.

Existe un orden de prioridades a la hora de efectuar varias operaciones y ese es el siguiente: Operaciones encerradas en paréntesis en primer lugar, luego vendría potenciación, le sigue la multiplicación y la división que ambas están en el mismo nivel de prioridad, después suma y resta, y por último tenemos la operación módulo y las operaciones de comparación que veremos enseguida. Es posible usar el operador "+" con los strings, con él se puede unir dos o más strings, a este proceso se le llama "concatenación". También podemos usar el operador "*" con ellos, haciendo que un string se repita tantas veces queramos. Podemos abreviar operaciones matemáticas cuando queramos acumular cantidades con los operadores, +=, -=, *=, /=, //=, %=, **=.

Código

```

1  a = "Hola "; b = "¿cómo estás?"; print(a+b) #concatenación de strings
2  c = "abc"; print(c*2) #se imprime ese string junto tantas veces se desee
3  d= 200; d+=10; d-=20; print(d) # es como escribir d = d+10 / d=d-20
4  f = 4 ; f *= 4; print(f) # es como escribir f = f*4
5  f//=4 ; print(f) # es como escribir f = f//4
6  f**=2; print(f) # es como escribir f = f**2
7  f%=2; print(f) # es como escribir f = f%2

```

Salida

```
Hola ¿cómo estás?  
abcabc  
190  
16  
4  
16  
0
```

Capítulo 3: Operadores lógicos y expresiones lógicas

Operadores lógicos

En Python tenemos, los siguientes operadores lógicos:

Operadores lógicos comparación: > , < , >= , <= , ==

Operadores lógicos de agrupación: and or

Operador lógico de negación: not

Estos operadores, se usan para trabajar expresiones lógicas. El resultado de utilizarlos en alguna operación lógica es siempre True o False (verdadero o falso). Ejemplo:

Código

```
1 a = 3 > 800  
2 b = 14 >= 7  
3 c = 5 == 5  
4 d = not c  
5 print(a, b, c, d)
```

Salida

```
False True True False
```

Aquí las variables se igualan a una operación lógica, por tanto estas variables contendrán el resultado de dicha operación.

Veamos caso por caso:

Es falso que 3 sea mayor que 800, entonces $3 > 800 = \text{False}$, por tanto "a", la primera variable, es igual a False.

14 es mayor igual a 7, se evalúa si es mayor o igual, en este caso es mayor por tanto el resultado de la operación $14 \geq 7 = \text{True}$, entonces $b = \text{True}$.

5 es idéntico a 5, son el mismo valor, entonces $5 == 5 = \text{True}$. Esto quiere decir que c vale True.

"not", es un operador que invierte el valor lógico, el operando se coloca en seguida del "not". En este caso se le invirtió el valor a la variable c, si c vale True, entonces $\text{not } c = \text{False}$, por tanto $d = \text{False}$. Esto no quiere decir que se le cambió el valor a c.

Estudiemos el operador "and", que a igual que el "or" es un operador binario que sirve para agrupar expresiones lógicas. Veamos el siguiente ejemplo:

Código		Salida
1	<code>a = 33 < 15</code>	
2	<code>b = 5 > 1</code>	
3	<code>c = 97 >= 97</code>	
4	<code>d = b and a</code>	
5	<code>e = c and b</code>	
6	<code>print(d, e)</code>	False True

Con la previa experiencia podemos decir rápidamente que "a" es igual a False, "b" es igual a True y "c" vale True. El operador "and" es un operador binario, es decir necesita dos operandos. El resultado de una operación con "and" será verdadero si únicamente ambos operandos valen True.

Entonces $b \text{ and } a = \text{False}$, porque $a = \text{False}$.

$c \text{ and } b = \text{True}$, porque ambos operandos son verdaderos.

Ahora veamos el operador "or", que al igual que el "and" es un operador binario. El resultado de una operación con "or" será Falso únicamente cuando ambos operandos sean falsos.

Código

```
1 a = 59 <= 1
2 b = 45 > 1381
3 c = 33 == 33
4 d = a or b
5 e = b or c
6 print(d, e)
```

Salida

```
False True
```

Notar que “a” y “b” valen False y “c” vale True. Por tanto se desprende lo siguiente:

a or b = False, ya que ambos operandos son falsos. Entonces d = False
b or c = True. Entonces e = True.

Capítulo 4: Funciones de conversión en Python 3

Strings

Como tal ya hemos visto a los strings en el anterior módulo, sabemos que se crean con comillas simples o dobles y podemos manipular texto con ellos. Pero no conocemos su estructura totalmente. Ahora conoceremos ligeros detalles sobre este tipo de dato.

Un string es una secuencia de caracteres y podemos acceder a cualquier carácter si queremos de esta manera.

< nombre de la variable que contiene el string > [posición del carácter]

La posición de los caracteres en un string se cuenta desde 0 de izquierda a derecha.

Aquí un ejemplo:

Código

```
1 a = 'Ab%!31'
2 # a[0]= A a[1]=b
3 # a[2]= % a[3]=!
4 # a[4]=3 a[5]=1
5 print(a[3])
6 print(a[0])
```

Salida

```
!
A
```

Cuando se usa la operación suma con ellos lo que ocurre es que se adhieren, es decir, se crea un nuevo string a partir de los strings anteriores.

Código

```
1 b = 'un string'
2 c = 'otro string'
3 print(b+c)
```

Salida

```
un stringotro string
```

Se puede multiplicar un string con un entero y lo que pasará es que se formará un nuevo string en donde el string original estará repetido tantas veces indique el número entero.

Código

```
1 a = 'abc'*3
2 print(a)
```

Salida

```
abcabcabc
```

Conversion de tipo de datos

Comúnmente llamado casting, este proceso consiste en convertir un tipo de dato a otro. Podemos hacer que un número con decimales, pase a ser un número entero, o un string lleno de números se pueda operar matemáticamente. Este proceso puede ser llevado a cabo con las funciones `int()`, `float()`, `str()`. Estas son las funciones de conversión de tipo de dato.

```
1 int() #convierte a entero
2 str() #convierte a string
3 float() # convierte a flotante
```

Estas funciones sólo admiten un argumento que es precisamente el dato a convertir. Tanto la función `int` como `float` no pueden tener como argumento strings alfanuméricos.

Aquí unos ejemplos de cómo usarlas.

Código

```
1 a = 18.1039; a = int(a)
2 print(a); print(float(a))
3 b = "340"; b = int(b)
4 b = b+a
5 print(b)
6 c = 34; c = str(c)
7 print(c+'texto')
```

Salida

```
18
18.0
358
34texto
```

Código

```
1 a = '48.17'
2 b = float(a); print(b)
3 c = int(b); print(c)
4 d = 4810
5 print(str(d)+'38491A')
```

Salida

```
48.17
48
481038491A
```

Capítulo 5: Impresión con formato e importación de módulos

Impresión con formato

Existe varias maneras de realizar una impresión en python, esto es gracias a que la función `print` tiene varios trucos que ahora conoceremos.

Imaginemos que necesitamos colocar el valor de una variable en medio de un string que se va a imprimir. Eso lo hemos hecho con anterioridad, pero muy poco y ahora se mostrarán ejemplos de cómo realizar la misma impresión pero de distinta manera.

Forma 1: Usando strings, comas y colocando las variables en las posiciones deseadas dentro de la función print()

```
1 nombre = input('ingrese su nombre: ')
2 edad = int(input('ingrese su edad: '))
3 print('Su nombre es: ', nombre, 'y su edad es: ', edad )
```

Forma 2: utilizando la función format(), dentro de las llaves van las posiciones de las variables que se colocaron dentro de los paréntesis de la función format, las posiciones se cuentan desde 0. La función format es propia de los strings.

```
1 nombre = input('ingrese su nombre: ')
2 edad = int(input('ingrese su edad: '))
3 print('Su nombre es: {0} y su edad es: {1}'.format(nombre, edad) )
```

Desde la versión 3.6 de python esta forma imprimir se simplificó a esto:

Se coloca una f antes del string y dentro de llaves se colocan las variables que se desea que aparezcan en esa parte del string.

```
1 nombre = input('ingrese su nombre: ')
2 edad = int(input('ingrese su edad: '))
3 print(f'Su nombre es: {nombre} y su edad es: {edad}')
```

Forma 3: utilizando la función str() y aplicando la operación suma.

```
1 nombre = input('ingrese su nombre: ')
2 edad = int(input('ingrese su edad: '))
3 print('Su nombre es: '+str(nombre)+' y su edad es: '+str(edad))
```

Forma 4: utilizando %d para variables enteras
 %s para los strings
 %f para flotantes

Estos se colocan dentro del string que se desea imprimir, luego después de éste se coloca porcentaje y luego las variables en el orden en que se colocaron los modificadores de formato respectivos a su tipo de dato.

```

1 nombre = input('ingrese su nombre: ')
2 edad = int(input('ingrese su edad: '))
3 print('Su nombre es: %s y su edad es: %d' % (nombre, edad))

```

Fácilmente se pudo notar que se mostró en todos los casos el mismo algoritmo,

se pide un nombre una edad y se muestran respectivos casos. Pero cada código fue distinto. Todos estos códigos tendrán el mismo resultado con la misma entrada.

Con el modificador de formato %f que trabaja con flotantes, podemos limitar la cantidad de decimales que se mostrará en la impresión de un número con decimales. Por defecto, con este modificador se mostrará 6 decimales y los que falte los completará con 0.

Aquí se muestra como se trabaja con el mismo número en diferentes casos con el modificador %f.

Impresión por defecto:

<u>Código</u>		<u>Salida</u>	
<pre> 1 a = 58.17274491 2 print("%f" % a) </pre>	Mostrando	<pre> 58.172745 </pre>	sólo 4

decimales:

<u>Código</u>		<u>Salida</u>
<pre> 1 a = 58.17274491 2 print("%.4f" % a) </pre>		<pre> 58.1727 </pre>

Mostrando 13 decimales: Como el número de la variable 'a' no tiene 13 decimales los que falte los completará con 0s.

<u>Código</u>		<u>Salida</u>
<pre> 1 a = 58.17274491 2 print("%.13f" % a) </pre>		<pre> 58.1727449100000 </pre>

Por defecto luego de terminar su trabajo, la función `print()` produce un salto de línea, esto hace que la próxima impresión aparezca debajo de la anterior. Podemos hacer que varias impresiones aparezcan en la misma línea colocando de último como argumento `end=' '` esto hará que la próxima impresión aparezca luego de un espacio.

Código

```
1 a = 5; print(a, end=' ')
2 b = 41.71 ; print(b, end=' ')
3 c = 7; print(c)
```

Salida

```
5 41.71 7
```

Módulos

Las librerías también llamadas módulos consisten en un repertorio de funciones diseñadas que para realizar tareas específicas y que podemos tomar prestadas. Para añadir un módulo a nuestro código usamos la palabra reservada `import` y luego el nombre del mismo.

Como ejemplo tenemos al módulo `math` que contiene una gran variedad de funciones vinculadas al mundo de la matemática. Para usar una función de un módulo importado debemos escribir el nombre del módulo, seguidamente de un punto y luego el nombre de la función.

Aquí un ejemplo con el módulo `math`.

Código

```
1 import math
2 a = int( math.log2(64) )
3 print(f'logaritmo de base 2 de 64 = {a}')
4 b = int ( math.cos(0) )
5 print(f'coseno de 0 = {b}')
6 c = int (math.fabs(-50) )
7 print(f'Valor absoluto de -50 = {c}')
```

Salida

```
logaritmo de base 2 de 64 = 6  
coseno de 0 = 1  
Valor absoluto de -50 = 50
```

Aquí un ejemplo con el módulo random, que es usado para la generar números pseudoaleatorios.

Código

```
1 import random  
2 A = random.randint(1, 100)  
3 # A contendrá un número entero entre 1 y 100  
4 B = random.uniform(1, 100)  
5 # b contendrá un número flotante entre 1 y 100  
6 print(A, B)
```

Salida

```
63 12.526670294485587
```

Al importar un módulo, se importa todas sus funciones las cuales podemos usar como en los ejemplos anteriores. Pero es posible que de un módulo tomemos prestado sólo las funciones que queramos usar. Eso hará que nuestro código sea más eficiente. Esto es posible usando la palabra clave **from** de esta manera

Podemos indicar de qué módulo se va importar una función, luego colocamos el nombre del módulo y finalmente usando import seleccionamos el nombre o los nombres de las funciones a importar. Si es más de una función se separan los nombres con coma. Aquí unos ejemplos.

Código

```
1 from math import exp
2 A = exp(2)
3 print('A = %.3f' % A, end=' ')
4 from random import randint
5 X = randint(100, 400)
6 print(f'X = {X}')
```

Salida

```
A = 7.389 X = 251
```

Código

```
1 from math import sqrt, sin, radians
2 print('Raíz cuadrada de 625 :', sqrt(625))
3 a = 'Seno de 45: '
4 print(a, sin(radians(45)) )
```

Salida

```
Raíz cuadrada de 625 : 25.0
Seno de 45: 0.7071067811865475
```


Condicionales

Los condicionales conforman un grupo de sentencias que permiten que nuestros programas tomen decisiones en función de una condición propuesta. Los condicionales están presente en todos los lenguajes de programación de propósito general.

La sentencia principal de todo condicional es `if` . La sentencia condicional `if` ejecutará las instrucciones que estén dentro de ella, sólo si se cumple una condición planteada por nosotros.

Aquí un esquema:

```
if ( CONDICIÓN ) :  
    <instrucciones>
```

En Python, se coloca `" :` para indicar que se inicia un nuevo **bloque de código** . El bloque de código es el conjunto de instrucciones que pertenecen a una determinada estructura como un condicional, un ciclo o una función. La primera sangría dentro de un bloque de código debe ser respetada en todas las líneas del mismo. La sangría debe ser de las instrucciones debe ser de por lo menos un espacio. No respetar estas reglas traerá como consecuencia errores.

```
<bloque de código> :  
---- <instrucción > ; <instrucción>  
---- <instrucción >  
---- <instrucción > ; <instrucción>
```

Colocar la condición de un condicional entre paréntesis es opcional pero recomendado. Escribir una instrucción dentro de un condicional sin respetar el espacio tabulado terminará en un error.

Aquí unos ejemplos:

Código

```
1  if(2+2 == 4):  
2      print('hola')  
3  
4  if(2+2 == 59):  
5      print('hola otra vez')
```

Salida

```
hola
```

Código

```
1  a = int(input())  
2  if(a > 5):  
3      b = 9; print(b+1)  
4  print(2)
```

Salida

caso 1

```
8  
10  
2
```

caso 2

```
1  
2
```

En este último ejemplo, sólo si el usuario ingresa un número mayor a 5, se imprimirá 10 . La instrucción de la línea 4 al estar fuera del condicional, se ejecutará indistintamente de que se cumpla o no la condición.

Ahora veamos la sentencia **else** . Se ejecutarán las instrucciones escritas dentro del ' else ' sólo si la condición propuesta resulta ser falsa. Al igual que el if se colocan ' : ' luego de escribir ' else ' y después de dar enter se generará un espacio tabulado donde podemos escribir sus instrucciones. El ' else ' se coloca luego de terminar de definir las instrucciones compuestas en el condicional anterior.

Código

```
1  if(2+2 == 5):  
2      print(1)  
3  else:  
4      print(2)
```

Salida

2

Por último abordaremos la sentencia **elif**. Esta sentencia se puede colocar debajo de un **if** y en caso de que la condición del **if** sea falsa, se evaluará la condición del **elif**. Es como colocar otro **if** pero que ni siquiera se evaluará si resulta ser que el primer **if** se cumple. Un **elif** puede ir debajo de otro **elif** y debajo de un **elif** podemos colocar un **else** que en caso de que los condicionales anteriores resultan ser falso se ejecutarán las instrucciones del **else** por defecto.

Aquí un ejemplo:

```
1  a = int(input())  
2  if(a>=42):  
3      print(1)  
4  elif(a>=29):  
5      print(2)  
6  elif(a>=0):  
7      print(3)  
8  else:  
9      print(4)
```

En este ejemplo si el usuario ingresa un número entero mayor igual a 42, se imprime 1, si ese no es el caso pero es mayor igual a 29 se imprime 2, de no ser así, si es mayor igual a 0 se imprime 3 y si este último condicional no se cumple se imprime 4.

Este ejemplo fue una cadena de condicionales, si las 3 primeras condiciones no se cumplen se imprime 4. Al cumplirse alguna de esas 3 primeras condiciones se ejecutan su respectiva instrucción y además no se evalúan las demás condiciones. Puede haber tantos **elif** como queramos.

Podemos colocar un condicional dentro de otro. Y es posible que coloquemos una o más instrucciones justo al lado los dos puntos.

Analizemos este ejemplo:

```

1  a = int(input())
2  if(a>=42):
3      b = int(input())
4      if(b == 5): print(1) ; print(2)
5      elif(b==2): print(3)
6      else: print(6)
7  else:
8      print(5)

```

Primero se le pide al usuario un número entero, éste será almacenado en la variable 'a'. De ser ese número mayor o igual a 42, se crea una variable 'b' la cual también el usuario determinará su valor que debe ser un número entero.

De ser el número ingresado por el usuario y alojado en b igual a 5, se imprime 1 y también 2. Ambas instrucciones print(1) y print(2) están vinculadas a ese condicional. De ser 2 el número, se imprime 3. Si no es 5 ni 2, entonces se imprime 6. Si al principio, la variable 'a' contiene un valor menor a 42 entonces se imprimiría 5 finalizando el programa.

Capítulo 6: Estructuras iterativas

Ciclos

Los ciclos, también llamados bucles conforman al igual que los condicionales estructuras que están presentes en todos los lenguajes de propósito general. Los ciclos son estructuras iterativas que van a ejecutar una o varias instrucciones tantas veces nosotros queramos mientras se cumpla una condición.

En Python tenemos los ciclos **while** y **for**.

Esquema del ciclo while:

```

while ( condición ) :
    <instrucciones>

```

El bucle while va a evaluar una condición, y de cumplirse ejecutará las instrucciones dentro del while, luego de ejecutar todas las instrucciones, volverá a evaluar la condición y de cumplirse seguirá ejecutando las instrucciones que tiene dentro. Luego de la condición se coloca los ':' a igual que con los condicionales abajo podemos colocar las instrucciones respetando el espacio tabulado. Debemos ser inteligentes a la hora de escoger la condición del while para evitar que se convierta en un ciclo infinito. Aquí un ejemplo:

Código

```
1 i = 1
2 while( i<=10):
3     print(i, end=' ')
4     i+=1
```

Salida

```
1 2 3 4 5 6 7 8 9 10
```

En el ejemplo se imprime los números del 1 al 10 usando el ciclo while. Las instrucciones se ejecutan hasta que la variable 'i' sea mayor a 10, en cada iteración la variable 'i' se le suma 1, de esta forma se logra el objetivo de imprimir los números desde 1 a 10.

Esquema del ciclo for:

```
for < variable > in <rango o conjunto > :
    < instrucción >
```

El ciclo for es una estructura iterativa bastante versátil y en cursos más avanzados veremos varias formas de implementar este ciclo. Luego de la palabra 'for' se crea una variable que servirá de iterador, luego se escribe 'in', después podemos colocar la función range() o colocar un conjunto como por ejemplo un string o una lista. Las listas la veremos más adelante.

El bucle antes de iterar, evalúa si el valor de la variable actual está dentro del rango o conjunto seleccionado, en caso de ser así se inicia la iteración de no ser así termina.

Veamos cómo imprimimos los primeros 10 números enteros con el bucle for:

Código

```
1 for i in range(1,11):  
2     print(i, end=' ')
```

Salida

```
1 2 3 4 5 6 7 8 9 10
```

Justo como el while se obtiene la misma salida. Pero el comportamiento es distinto. Aquí se usa la función range(). Esta función puede admitir uno, dos o tres argumentos, todos deben ser números enteros.

La función range se encarga de crear un rango de números enteros.

Con un argumento crea un rango de números enteros desde 0 hasta el número colocado menos 1.

Ejemplo : `range(5)` → 0 , 1, 2, 3 , 4

Con dos argumentos se crea un rango que va desde el primer argumento hasta el 2do menos 1

Ejemplo: `range(1, 5)` → 1, 2 , 3, 4

Con tres argumentos se crea un rango que como si tuviera dos argumentos pero es distinto. Dependiendo del tercer número, la secuencia de números irá ascendiendo de forma distinta.

Ejemplo: `range(1, 11, 2)` (irá ascendiendo en 2 en 2) → 1, 3, 5, 7, 9

Ejemplo: `range(1,21,5)` (irá ascendiendo 5 en 5) → 1, 6, 11, 16

En cada caso la variable que sirve de iterador comienza valiendo lo mismo que el primer número del rango creado por la función range() y cada iteración va recorriendo dicho rango.

Esto quiere decir que en la primera iteración 'i' vale 1, entonces se imprime 1, en las siguiente iteración 'i' vale 2, se imprime 2, así sucesivamente, hasta llegar a 10, porque 11 no está dentro del rango formado por la función range(1,11).

Luego del 'in' podemos colocar una variable que contenga un string, entonces la variable iteradora al principio contendrá el valor del carácter inicial, luego irá avanzando en carácter en carácter hasta que no haya más. Aquí un ejemplo:

Código

```
1 a = 'a3tmf'
2 for i in a:
3     print(i, end=' ')
```

Salida

```
a 3 t m f
```

Ahora veamos dos instrucciones relacionadas con los ciclos. Estas instrucciones también aparecen en otros lenguajes de programación ya que están relacionadas con las estructuras iterativas.

Instrucción **break** : Sirve para detener inmediatamente el ciclo.

Instrucción **continue** : Sirve para que inicie de forma inmediata la siguiente iteración.

Veamos el mismo ejemplo anterior, pero usando estas instrucciones.

Imaginemos que del string almacenado en la variable queramos imprimir todas sus caracteres a excepción de los caracteres 'm' y '3'.

Usando la instrucción continue y un condicional acorde con nuestro deseo logramos ese objetivo.

Código

```
1 a = 'a3tmf'
2 for i in a:
3     if(i == '3' or i == 'm'): continue
4     print(i, end=' ')
```

Salida

```
a t f
```

Ahora imaginemos que queremos imprimir sólo los 3 primeros caracteres del string. Podemos usar la instrucción break para lograr eso. Cuando la variable i valga 'm' hacemos que se detenga el ciclo.

Código

```
1 a = 'a3tmf'
2 for i in a:
3     if(i == 'm'): break
4     print(i, end=' ')
```

Salida

```
a 3 t
```

Capítulo 7: Funciones, programación modular

Funciones

Hasta ahora hemos usado funciones propias del lenguaje. Ahora aprenderemos a crear nuestras propias funciones. . Podemos definir nuestras funciones y luego decidimos cuándo usarlas. De hecho desde ahora estaremos acostumbrados a crear funciones, nos puede en muchos casos ahorrar líneas de Código.

Si tenemos por ejemplo una función que detecta si un número es primo, y necesitamos determinar si 500 números ingresados por el usuario son primos, usamos un ciclo que itere 500 veces y que en cada iteración invoque la función que hicimos para determinar si un número es primo

En Python la estructura de una función creada por nosotros es la siguiente.

```
def < nombre de la función > ( argumentos ) :
    <instrucciones>
```

Para invocar una función :

```
< nombre de la función a invocar > ( argumentos )
```


Una función empezará a realizar su trabajo en el momento que se invoca. Una vez definida nuestra función la podemos invocar cuando y donde queramos. Es posible que una función tenga o no argumentos, eso es queda a decisión nuestra.

El nombre de una función lo escogemos nosotros pero debe respetar las mismas reglas que con los nombres de las variables.

Aquí un ejemplo de funciones sin argumentos.

Código

```
1 def funcion1():
2     print('hola')
3
4 def funcion2():
5     print('Hola de nuevo')
6
7 funcion1(); funcion2()
```

Salida

```
hola
Hola de nuevo
```

En seguida un par de ejemplos de funciones con argumentos.

Código

```
1 def mitad(x):
2     print(x/2)
3
4 def duplicar(a):
5     print(a*2)
6
7 duplicar(8); mitad(412)
```

Salida

```
16
206.0
```

Cuando definimos nuestra función, dentro de los paréntesis se colocan los argumentos, estos argumentos serán variables que obtendrán su valor cuando invoque la función.

En este ejemplo se crearon dos funciones, 'mitad' que lo que hace es imprimir la mitad de su argumento y 'duplicar' que imprime el argumento multiplicado 2.

Cuando se invocó la función 'duplicar', se colocó 8 entre sus paréntesis, por lo tanto x vale 8.

Cuando se invocó la función 'mitad' se colocó 412 entre sus paréntesis, por ende, a vale 412.

Código

```
1 def una_funcion(a, b):
2     print(f"Números enteros desde {a} hasta {b}")
3     for i in range(a, b+1):
4         print(i, end=' ')
5
6 una_funcion(42, 56)
```

Salida

```
Números enteros desde 42 hasta 56
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
```

Los valores de 'a' y 'b' se asignaron en la invocación de la función. El primer valor corresponde al primer argumento, es decir 'a' y el segundo valor corresponde al segundo argumento, es decir 'b'. La función que tiene como nombre 'una _ funcion' lo que hace es imprimir una serie de números comprendidos en un rango desde el valor de 'a' hasta el valor de 'b'.

Ahora veamos la instrucción **return**. Esta instrucción devuelve el valor que se desee al lugar en donde se invocó la función.

Esquema:

return < lo que se quiera devolver >

La instrucción return también finaliza la ejecución de la función. Es decir luego de la función ejecute esta instrucción no ejecuta ninguna más dentro de ella. Además podemos elegir usar la función return sólo para este fin, tal como sucede con la instrucción break de los ciclos. Si colocamos la instrucción return sola, detendrá la ejecución de la función.

Aquí unos ejemplos:

Código

```
1 def suma(x,y):
2     return x+y
3
4 d = suma(2, 3) ; print(d)
5 # tambien serviría: print(suma(2,3))
```

Salida

```
5
```

Código

```
1 def funcion(y):
2     x=2
3     print(f'Números pares desde 2 hasta {y}')
4     while(True):
5         if(x==y+2): return
6         print(x, end=' ')
7         x+=2
8
9 a = int(input('Ingrese número par: '))
10 funcion(a)
```

Salida

```
Ingrese número par: 16
Números pares desde 2 hasta 16
2 4 6 8 10 12 14 16
```

En el último ejemplo, el programa se encarga de imprimir números pares y el usuario tiene la oportunidad de decidir hasta qué número par se va a imprimir. Notar que la condición del ciclo es True, esto quiere decir que ese ciclo while estaría iterando eternamente si no fuese por el condicional colocado dentro de su bloque de instrucciones. Cuando la variable x valga lo mismo que el primer número par mayor al ingresado por el usuario, la función finaliza.

Capítulo 8: Listas

En Python es una estructura de almacenamiento que permite guardar datos de distinto tipo de forma secuencial.

Para crear una lista en Python, creamos una variable y la igualamos a unos corchetes "[]" y dentro de ellos podemos colocar los elementos que queremos almacenar separados por comas.

```
1  f = ['True', 4, 99.3]
2  # f es una lista de
3  # 3 elementos, un booleano, un entero y un flotante
```

Para acceder a un elemento específico de una lista debemos tener en cuenta que las posiciones de los elementos almacenados en la lista se cuentan desde 0 de izquierda a derecha. Para acceder a un elemento contenido en una lista, usamos la siguiente sintaxis.

<nombre de la lista >[posición del elemento]

```
1  a = [1, 3, 'a', 3.4]
2  var1 = a[0] ; var2 = a[1]
3  # var1 = 1   var2 = 3
4  var3 = a[2] ; var4 = a[3]
5  # var3 = 'a'   var4 = 3.4
```

Las listas tienen "incluidas" una función con la cual se puede añadir un nuevo elemento, es la función `append()` dentro de los paréntesis se coloca el elemento que se quiere ingresar, ya sea que esté contenido en una variable o no. Para usarla se coloca luego del nombre de la lista un punto y seguido la función.

```
1  a = [] # lista vacia
2  b = 5
3  a.append(b) # ingresando b
4  a.append(True) # ingresando True
5  a.append(4) # ingresando 4
6
7  # Lista [5, True, 4]
```

Se puede colocar dentro de los paréntesis de la función `append()`, la función

`input()`, `int(input())` o incluso `float(input())` para que sea el usuario quien ingrese los elementos que quiera en la lista.

La función `pop()`, elimina el último elemento de la lista.

la función `len()`, retorna un entero que indica la cantidad de elementos que hay en su argumento. Su argumento puede ser una lista o un string (en ese caso retorna un entero que indica la cantidad de elementos de la lista).

```
1  a = [6, 4.1, 'abc']
2  a.pop()
3  # ahora a = [6, 4.1]
4  v = len(a) # v = 2
```

En python hay más estructuras de almacenamiento, como por ejemplo los diccionarios, tuplas etc. Pero escapa de los propósitos del curso conocerlas ya que son estructuras propias del lenguaje. Las listas de python son lo más parecida a los arreglos, que en general son estructuras de almacenamiento presentes en varios lenguajes de programación. En próximos cursos profundizaremos mucho más acerca de las estructuras de almacenamiento en general.

Con el ciclo for podemos recorrer una lista colocando el nombre de la lista luego de la palabra "in" del for. De esta manera la variable iteradora del ciclo va comenzar valiendo lo mismo que el primer elemento ingresado en dicha lista, cambiará en cada iteración su valor correspondiendo a los elementos ingresados hasta llegar al último .

Ejemplo

```
l=[2,4,6]
suma=0
for i in l:
    suma+=i
print(suma)
```

La salida de este código es 12. Notar que la variable i comienza valiendo 2, luego 4 y por último 6. En este código, se almacenó en la variable "suma" la suma de los números enteros almacenados en la lista para luego imprimir el contenido de tal variable.

Capítulo 9: Lectura de archivos de texto

Con Python somos capaces de hacer programas que trabajen con información contenida en un archivo, para eso primero debemos hacer que nuestro programa lea tal información.

La función `open()` se encarga de la manipulación de archivos en python. Normalmente usa dos argumentos, el primero será el nombre del archivo con el que se quiere trabajar y su extensión (como un string) . De esta forma el archivo deberá estar en el mismo directorio que el del código fuente de nuestro programa. Si el archivo con el que se quiere trabajar no está en el mismo directorio que nuestro código fuente, podemos colocarlo como primer argumento la ruta absoluta del archivo (como un string) .

Como segundo argumento se debe colocar el modo de acceso del archivo, si quiere leer, colocamos entre comillas simples o dobles en minúsculas la letra `r` (como un string). Si se quiere escribir en él debemos colocar como segundo argumento una `w` (como un string).

Para manejar el archivo, se crea una variable y se iguala a la función `open` con sus respectivos argumentos. Con esa variable durante todo el código nos referiremos al archivo que abrimos con la función `open` . Si queremos añadir información en un archivo debemos abrirlo en modo de escritura si tan solo queremos trabajar con la información que se encuentre en el archivo lo abrimos en el modo lectura.

Imaginemos que queremos abrir un archivo `foo.txt` en modo lectura y que está en el mismo directorio que nuestro código fuente.

Se hace lo siguiente:

```
1 a = open('foo.txt', 'r')
```

Y si lo quisiéramos abrir en modo de escritura se hace lo siguiente:

```
1 a = open('foo.txt', 'w')
```

La variable `a`, servirá entonces para referirnos al archivo.

Ahora si por ejemplo, quisiéramos abrir un archivo que esté fuera del directorio de nuestro código fuente hacemos lo siguiente:

```
a = open(r'C:\Users\userPC\Desktop\codes\foo.txt', 'r')
```

Colocamos un string referente a la ruta absoluta del archivo como primer argumento. Para evitar ciertos problemas es recomendado colocar justo antes de ese string la letra r.

En este último ejemplo se abrió en modo lectura, pero también se pudo haber abierto en modo escritura.

Si se coloca como segundo argumento `'w+'`, se abre el archivo seleccionado en modo de escritura y en caso de no existir se crea el archivo y tendrá el nombre y la extensión justo como se haya colocado en el primer argumento de la función open.

Luego de abrir el archivo mediante la función open() la variable usada para abrirlo "adquiere" funciones diseñadas para el manejo de archivos.

Veamos la función `read()`.

Si en el archivo foo.txt está escrito lo siguiente.

```
foo.txt x
1 vaca puerta
2 47
3 avion
```

La función read() se encarga de retornar todo el contenido de un archivo. por tanto si se coloca dentro de la función print() se estaría imprimiendo todo el contenido del archivo.

Código

```
1 a = open('foo.txt', 'r')
2 print(a.read())
```

Salida

```
vaca puerta
47
avion
```

Dentro de los paréntesis de la función read podemos colocar un número entero que indica la cantidad de caracteres del archivo a retornar.

Código

```
1 a = open('foo.txt', 'r')
2 print(a.read(4))
```

Salida

```
vaca
```

Se mandó a retornar solo 4 caracteres, por tanto se termina imprimiendo solo los 4 primeros caracteres del archivo que corresponde a la palabra vaca.

Veamos la función `readlines()`, que retorna una lista de strings del archivo.

Hay que entender que cuando por ejemplo escribimos en un archivo de texto, cada línea es un string, cuando se presiona enter para escribir en otra línea, estamos es generado un salto de línea (`\n`) y luego el cursor se posiciona en la siguiente línea. Entonces la primera línea del archivo `foo.txt` es una cadena de caracteres que tiene como último carácter un `\n` implícito.

Código

```
1 a = open('foo.txt', 'r')
2 print(a.readlines())
```

Salida

```
['vaca puerta\n', '47\n', 'avion\n', '\n']
```

Mediante un ciclo `for` podemos recorrer línea por línea un archivo. La variable iteradora será igual a la línea actual, es decir un string, que recordemos tiene un salto de línea.

Código

```
1 a = open('foo.txt', 'r')
2 for i in a:
3     print(i)
```

Salida

```
vaca puerta
47
avion
```

Recordemos que la función `print()` por defecto genera un salto de línea y si a esto le adicionamos el hecho de que cada string tiene un salto de línea al final, el resultado del código anterior es coherente, se está haciendo un salto de línea de más en la salida.

Como `i` en cada iteración será un string, y los strings tienen varias funciones, podemos usar una que nos permita quitar el salto de línea del contenido de la variable `i`.

Código

```
1 a = open('foo.txt', 'r')
2 for i in a:
3     print(i.strip('\n'))
```

Salida

```
vaca puerta
47
avion
```

`strip('\n')`, nos permite eliminar del string de `i` el carácter de salto de línea antes de generar la impresión.

Ahora veamos la función `readline()` (no confundir con `readlines()` que termina en `s`) esta función retorna el string actual, es decir la línea actual del archivo, luego de usarla, el programa se sitúa en la siguiente línea, de modo que al volver usarla, la función retornara la siguiente línea del archivo.

Código

```
1 a = open('foo.txt', 'r')
2 print(a.readline().strip('\n'))
3 print(a.readline().strip('\n'))
```

Salida

```
vaca puerta
47
```

Se utilizó tal función dos veces, y en cada ocasión se mandó a imprimir lo que retornaba eliminando el salto de línea de cada string (línea), por tanto en la salida se imprimieron las dos primeras líneas del archivo.

Capítulo 10: Escritura de archivos de texto

Si queremos abrir en modo de escritura un archivo llamado (por ejemplo) `foo.txt` que este en el mismo directorio que nuestro código fuente se hace lo siguiente:

```
1 a = open('foo.txt', 'w')
```

Al colocar `'w'` como segundo argumento se está abriendo el archivo en modo escritura. Lo que hace el programa a la hora de abrir este archivo de esta manera es que accede al archivo (si existe), borra todo el contenido en él y luego prepara al archivo para que se le pueda añadir contenido.

Si no queremos que se borre el contenido del archivo y sólo queramos añadir más contenido, podemos hacer lo siguiente:

```
a = open('foo.txt', 'a')
```

El segundo argumento 'a' nos permite abrir el archivo sin borrar su contenido y prepara el archivo para añadir contenido desde el último lugar donde se escribió el archivo.

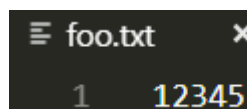
En ambos casos, con agregar un '+' al segundo argumento, se creará el archivo en caso de que no exista, tendrá el mismo nombre y extensión que se haya colocado en el primer argumento.

Primero veamos la función `write()`. Con esta función el programa puede escribir en el archivo, su argumento debe ser un string, al cual es opcional colocarle un salto de línea si se quiere volver a escribir en el archivo en la siguiente línea.

Código

```
a = open('foo.txt', 'w')
for i in range(1,6):
    a.write(str(i))
```

Salida



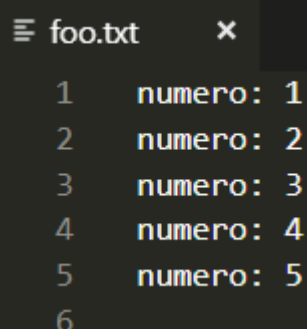
```
foo.txt
1 12345
```

Mediante el for, escribimos los numero del 1 al 5 en el archivo. Los números se escribieron juntos, ya que nunca se indico algún espacio de separación dentro de los paréntesis de la función `write()`.

Código

```
a = open('foo.txt', 'w')
for i in range(1,6):
    a.write('numero: '+str(i)+'\n')
```

Salida



```
foo.txt
numero: 1
numero: 2
numero: 3
numero: 4
numero: 5

```

Esta vez, se imprimió cada número en una línea distinta, luego del string 'numero: ' y por último se colocó '\n' para generar el salto de línea.

Capítulo 11: Programación Orientada a Objetos

En la actualidad, varios lenguajes de programación soportan el paradigma orientado a objetos, (POO por sus siglas en inglés) . Su uso popularizó en la década de los 90.

Es un paradigma de programación que trajo gran innovación al mundo programación ya que permitía llevar una mejor administración de los datos, encapsularlos en entes que llamaremos **objetos**.

Cada objeto posee un conjunto de atributos y funciones previamente definidas en una **clase**, con los cuales un objeto puede interactuar con objetos de otras clases .

Llevando este tema a tópicos de la vida real, un objeto de una clase llamada “persona” puede tener como atributos, género, edad, nombre, estatura, etc y como funciones puede ser hablar, caminar, correr, saltar etc. recordemos, todo esto estará definido en una clase.

Las funciones de las clases se le llaman **métodos**, y los atributos de un objetos pueden ser públicos o privados, eso queda a nuestra elección. Los atributos privados de un objeto solo pueden ser modificados, por sus propios métodos, o permitiendo a través de los mismos a que desde afuera de la misma se manipulen. Si los atributos son públicos, estos se pueden modificar tanto afuera como dentro de su clase.

También, puede que una clase herede de otra, de esta forma los objetos de la clase “hija” heredan todos los atributos y métodos públicos de la clase padre.

POO en python

class es la palabra reservada que nos permite crear una clase.

Toda clase debe tener un “ **método constructor** “, que es donde se colocan los atributos de la clase.

Los métodos, se crean como las funciones y deben estar dentro del bloque de código de sus clases, y siempre deben tener como argumento la palabra reservada **self**. El método constructor debe tener como nombre, **__init__**.

Los atributos, se crean como las variables, sólo que al principio se le coloca la palabra **self** y luego de un punto el nombre del atributo.

```
1 class personaje:
2     def __init__(self, v, p):
3         self.vida = v
4         self.poder = p
5
6 x = personaje(500, 15)
```

Este es un ejemplo de una clase en python. La clase personaje tiene 2 atributos, vida y poder, primero sus valores se pasan al constructor y luego a ellos. El primer argumento del constructor será el valor del atributo vida y el segundo será el valor del atributo poder.

x es un objeto de la clase personaje, los objetos se crean como las variables y se igualan al constructor de la clase que deseemos. Como el constructor admite 2 argumentos, se le colocan dentro de los paréntesis los valores que queremos que posea sus atributos anteriormente definidos en la clase. El 500 se trata del argumento "v" y 15 del argumento "p".

```
1 class personaje:
2     def __init__(self, v, p):
3         self.vida = v
4         self.poder = p
5
6     def imprimir_atributos(self):
7         print(self.vida, self.poder)
8
9
10 x = personaje(500, 15)
11 x.imprimir_atributos()
```

Ahora la clase personaje tiene un método que se encarga de imprimir los atributos de la clase. En la línea 11, el objeto de la clase personaje, x, está haciendo uso de ese método. Por tanto la salida de este código es la siguiente: **500 15**

Una clase puede tener tantos métodos como queramos.

Los atributos self.vida y self.poder están en modo público, es decir que pueden ser modificados afuera de la clase, como en este ejemplo:

```

1  class personaje:
2      def __init__(self, v, p):
3          self.vida = v
4          self.poder = p
5
6      def imprimir_atributos(self):
7          print(self.vida, self.poder)
8
9
10 x = personaje(500, 15)
11 x.vida = 700
12 x.poder = 33
13 x.imprimir_atributos()

```

Salida: **700 33**

Para colocar los atributos en modo privado se escribe dos guiones bajos seguidos antes del nombre del atributo. De esta forma los atributos no serán modificados afuera de la clase.

```

1  class personaje:
2      def __init__(self, v, p):
3          self.__vida = v
4          self.__poder = p
5
6      def imprimir_atributos(self):
7          print(self.__vida, self.__poder)
8
9
10 x = personaje(500, 15)
11 x.vida = 700
12 x.poder = 33
13 x.imprimir_atributos()

```

Salida: **500 15**

También podemos tener tantos objetos de una clase como queramos.

Herencia

Una clase puede “hija” de otra, de esta manera la clase hija puede heredar los atributos y métodos públicos de su clase padre. Además, es posible que una clase tenga dos clases padres.

Ejemplo:

```
1  class A:
2      def __init__(self):
3          self.x = 5
4          self.y = 8
5
6  class B(A):
7      def atributos_heredados(self):
8          print(self.x, self.y)
9
10
11  objeto = B()
12  objeto.atributos_heredados()
```

La sintaxis para la herencia, es colocar después del nombre de una clase entre paréntesis el nombre de la clase que queremos que sea su clase padre. Notamos en este ejemplo que la clase A es padre de la clase B. La clase B, no tiene constructor sino que más bien tiene un solo método que se encarga de imprimir los atributos heredados de su clase padre.

Si una clase hija no tiene constructor, sus objetos se usarán el constructor de la clase padre, y si lo tiene sus objetos usarán el de su propia clase. En este ejemplo, el constructor de la clase A, no admite más argumentos que el self ya que de por sí a sus atributos se les otorgó de una vez sus valores. La clase B, hereda los atributos de su clase padre con los valores que se les otorgaron. Es por eso que la salida de este código es el siguiente:

```
5 8
```

Cuando una clase deriva de dos clases, es decir cuando una clase tiene más de una clase padre se le conoce como herencia múltiple.

Ejemplo:

```

1  class A:
2      def __init__(self, n):
3          self.a = n
4      def metodo1(self):
5          print(1)
6
7      def metodo2(self):
8          print(3)
9
10 class B:
11     def __init__(self, a, b):
12         self.a = a
13         self.b = b
14     def metodo1(self):
15         print(2)
16
17 class C(B, A):
18     def atributos_heredados(self):
19         print(self.a, self.b)
20
21 objeto = C(73, 3); objeto.metodo1()
22 objeto.metodo2() ; objeto.atributos_heredados()

```

En este ejemplo, la clase C tiene a las clases B y A como clases padres. Notar que la sintaxis de herencia múltiple consiste en colocar luego del nombre de una clase entre paréntesis, los nombres de las clases padres y separados por coma.

El primer nombre de la clase que se coloque entre los paréntesis corresponde con la clase a la que se le tendrá mayor prioridad en la herencia. Es decir, si se da el caso de que la clase hija no tiene constructor, el objeto de la clase hija utilizará el constructor de la clase cuyo nombre se colocó de primero entre los paréntesis. Vemos que en este ejemplo, tanto la clase A como la clase B tienen un método que tienen el mismo nombre, entonces, el objeto de la clase hija va a heredar el método de la clase B puesto que su nombre se colocó de primero en los paréntesis de la herencia múltiple. Notemos que el objeto heredó los atributos de la clase B y el método “metodo2” de la clase A y el método “metodo1” de la clase B. Recordemos que el constructor se utiliza cuando se está creando el objeto, como el constructor heredó el constructor de la clase B, y este admite 2 argumentos, por eso se le colocó 2 argumentos, en este caso 73 y 3.

```

2
3
73 3

```

La salida de este código es la siguiente: