

Ciclos y métodos

Lo que debes saber antes de comenzar esta unidad

Un algoritmo es una serie de pasos para resolver un problema:

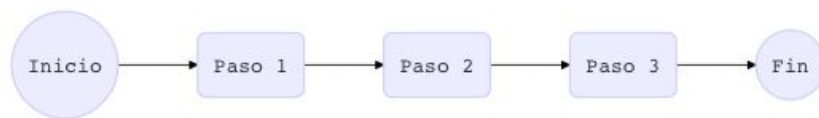


Imagen 1. Diagrama de algoritmo.

Estos pasos pueden ser:

- Entradas de datos:
 - `input.`
 - `sys.argv.`
- Salidas de datos:
 - `print.`
- Asignaciones:
 - `a = 2.`
 - `a + = 1.`
- Comparaciones:
 - `a == 3.`
- Operaciones matemáticas y precedencia:
 - `a + 1 * 2 - 1.`

- Operaciones lógicas:
 - `a and not b.`
- Decisiones:
 - `if a == 2.`

Tipos de datos

- String: `"hola"`.
- Integer: `1`.
- Float: `6.3`.
- Boolean: `True`.

Transformando datos

- `int()` para pasar un dato a Integer.
- `float()` para pasar un dato a Float.
- `str()` para pasar un dato a String.

Utilizando estas instrucciones se pueden resolver distintos tipos de problemas:

- Resolver una ecuación matemática:
 - Permitir al usuario ingresar valores
- Crear una calculadora

Capítulo 1: Ciclos (Loops)

Competencias

- Conocer los ciclos y sus posibles aplicaciones en la programación.
- Conocer sintaxis de un "bloque".
- Leer y transcribir diagramas de flujo con loops a un ciclo while.
- Validar una entrada de datos.
- Realizar programas donde el usuario ingrese múltiples datos hasta que decida detenerse.
- Crear un menú de opciones.

Introducción a ciclos

Los ciclos son sentencias que nos permiten repetir la ejecución de una o más instrucciones.

Mientras se cumple una condición:

Instrucción 1
Instrucción 2
Instrucción 3

Repetir instrucciones es la clave para crear programas avanzados.

Usos de ciclos

Los posibles usos de ciclos en algoritmos son infinitos. Entre otros, nos permiten recorrer colecciones de datos o espacios de búsqueda.

Anteriormente se revisó como ejemplo el algoritmo para preparar panqueques. En este ejemplo, una de las instrucciones era "Repetir pasos 6 a 10 hasta que se termine la mezcla". Este es un ejemplo de cómo con una sola sentencia se evita reescribir pasos que son repetitivos varias veces.

RECETA DE PANQUEQUES	ALGORITMO
<ul style="list-style-type: none"> - Agregar 1 taza de harina en un bowl. - Agregar 1 taza de leche a la harina. - Agregar 1 huevo a los ingredientes previos. - Resolver y mezclar los 3 ingredientes. - Precalentar el sartén. - Agregar parte de la mezcla hasta cubrir el sartén y esparcir una capa delgada. - Esperar 1 minuto. - Dar vuelta la masa. - Esperar otro minuto. - Retirar el panqueque. - Si queda masa, agregar parte de la mezcla hasta cubrir el sartén y esparcir una capa delgada. - Esperar 1 minuto... 	<ol style="list-style-type: none"> 1. Agregar 1 taza de la harina en un bowl. 2. Agregar 1 taza de leche a la harina. 3. Agregar 1 huevo a los integrantes previos. 4. Resolver y mezclar los 3 ingredientes. 5. Precalentar el sartén. 6. Agregar parte de la mezcla hasta cubrir el sartén y esparcir una capa delgada. 7. Esperar 1 minuto. 8. Dar vuelta la masa. 9. Esperar otro minuto. 10. Repetir pasos del 6 al 10 hasta terminar la mezcla.

Tabla 1. Receta de Panqueques.

Parte importante de aprender a programar, es entender cómo utilizarlos correctamente, y desarrollar las habilidades lógicas para entender cuándo utilizarlos.

En esta unidad veremos distintos problemas que se resuelven con ciclos, y que si bien no siempre son ocupadas en la industria, nos ayudarán a desarrollar las habilidades lógicas que necesitamos para ser buenos programadores.

Introducción a bloques

Muchos métodos de Python pueden recibir el nombre de **bloque**. Este concepto es **muy** importante, y se profundizará en él posteriormente. Sin embargo, en este punto, se busca lograr entender su sintaxis.

De igual manera que como se realizó con los condicionales **if**, **elif** y **else**, todo bloque se declara posterior a un colón **:**. La siguiente línea, donde se inicia el bloque, **debe estar indentada con 4 espacios en blanco**.

```
if 10 > 2:  
    # el bloque es todo lo que se contiene después de :  
    # y que esté precedido por 4 espacios en blanco.  
    print("Mayor a 2")
```

Ciclo **while**

La instrucción **while** nos permite ejecutar una o más operaciones **mientras** se cumpla una condición. Esta condición es booleana, y debe evaluarse como True para continuar el ciclo. Su sintaxis es la siguiente:

```
while condicion:  
    # codigo a implementar
```

while paso a paso

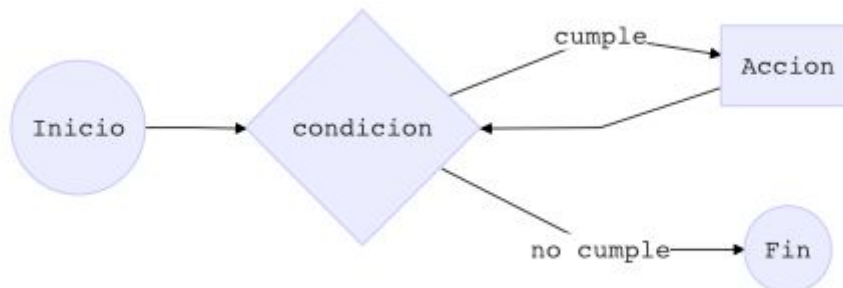


Imagen 2. Ciclo while.

1. Se evalúa la condición; si es **True**, ingresa al ciclo.
2. Se ejecutan secuencialmente las instrucciones definidas dentro del ciclo.
3. Una vez ejecutadas todas las instrucciones se vuelve a evaluar la condición:
 - Si se evalúa como **True**: vuelve a repetir.
 - Si se evalúa como **False**: sale del ciclo.

Salida del ciclo

Un algoritmo es una secuencia de pasos **FINITA** para resolver un problema. En algún momento, algunas de las instrucciones dentro del bloque debe lograr que la condición no se cumpla. Dicho de otra forma, debe existir una **condición de salida o término** del ciclo.

Ejemplo

En el siguiente ejemplo, la condición de salida es **que el número sea mayor a 10**, ya al alcanzar este valor, no se podrá volver a ingresar al ciclo.

```
numero = 0

while numero <= 10:
    print(numero)
    numero += 1
```

Validación de entrada de datos utilizando un ciclo **while**

Un ejemplo común para comenzar a estudiar los ciclos es validar la entrada de un dato, es decir, que este dato **cumpla un criterio**. Podemos, por ejemplo, validar que el usuario ingrese un número entre 1 y 10:

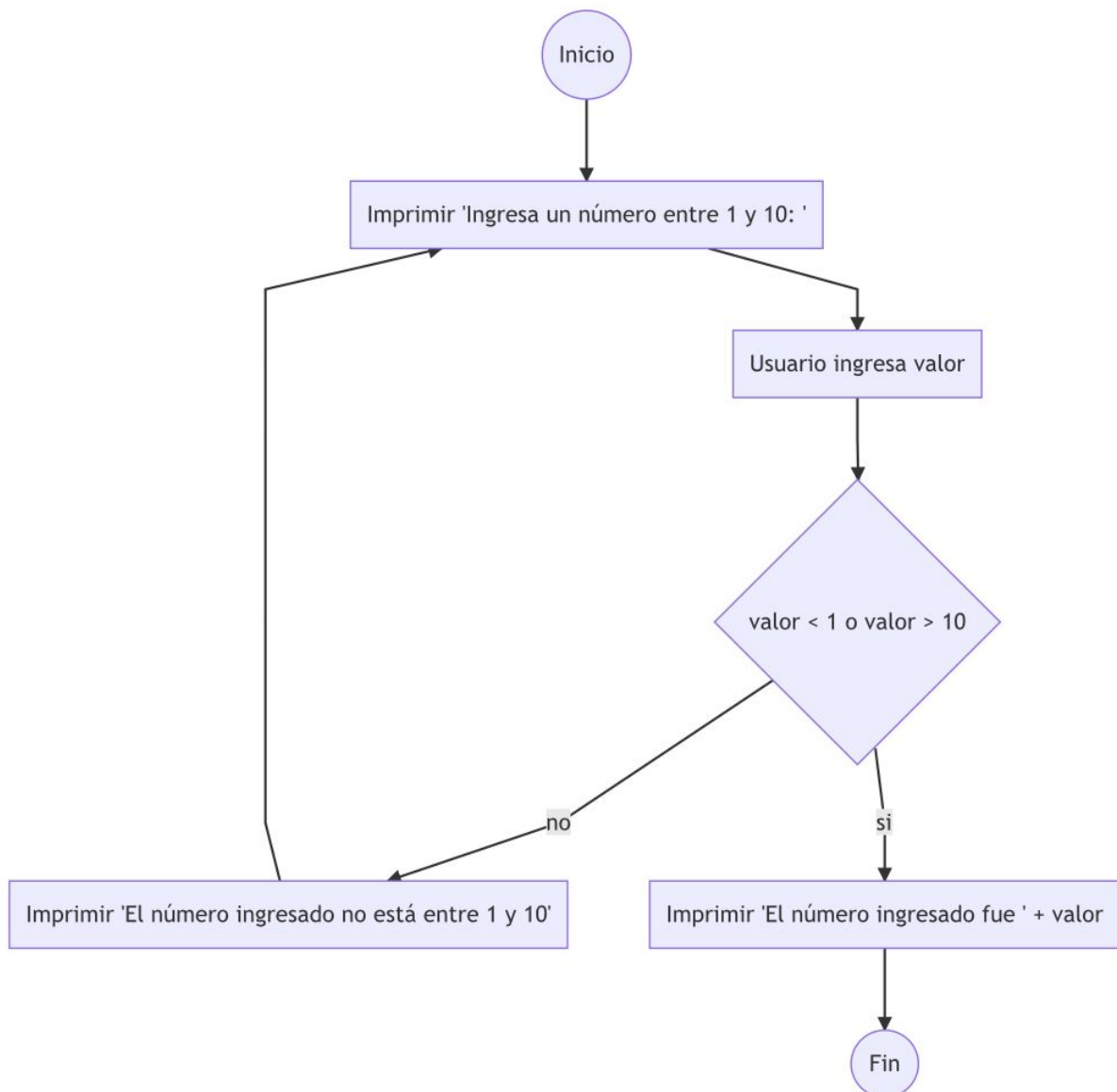


Imagen 3. Validación de entrada de datos utilizando un ciclo **while**.

```
number = int(input("Ingresa un número entre 1 y 10"))

while number < 1 or number > 10:
    print("El número ingresado no está entre 1 y 10")
    number = int(input("Ingresa un número entre 1 y 10"))

print("El número ingresado fue ", number)
```

¿Por qué es necesario declarar dos veces el ingreso de datos por parte del usuario?

Si no se pregunta por el valor del número antes de comenzar el ciclo, ocurrirá que cuando el flujo llegue a la evaluación dentro del ciclo `while`, la variable `número` no estará definida. Por tanto, no se puede ingresar al ciclo y se obtendrá un error.

```
print("Ingresa un número entre 1 y 10")

while numero < 1 or numero > 10:
    print("El número ingresado no está entre 1 y 10")
    numero = int(input("Ingresa un número entre 1 y 10"))

print("El número ingresado fue ", numero)
```

Ingresa un número entre 1 y 10

```
-----
---

NameError                                Traceback (most recent call
last)

<ipython-input-15-f0fe3769982c> in <module>()
      1 print("Ingresa un número entre 1 y 10")
      2
----> 3 while numero < 1 or numero > 10:
      4     print("El número ingresado no está entre 1 y 10")
      5     numero = int(input("Ingresa un número entre 1 y 10"))
```

NameError: name 'numero' is not defined

Ejercicio de integración

Podríamos utilizar la misma idea de validación para impedir al usuario entrar en una aplicación hasta que ingrese el password "password".

Solución

```
password = input("Ingrese el password")

while password != "password":
    print("La contraseña es incorrecta")
    password = input("Ingrese el password")

print("La contraseña es correcta")
```

Aviso sobre `input`

Durante el desarrollo de programas no utilizaremos `input`, pues esto bloquea el programa hasta que el usuario ingresa un valor y hace complicado el uso y evaluación de scripts. A la hora de crear programas, utilizaremos argumentos o valores al azar.

Desafío: Hacer un menú de opciones

Podemos implementar de forma sencilla es un menú de opciones para el usuario.

La lógica es similar a la de la validación de entrada.

- Se muestra un texto con opciones.
- El usuario tiene que seleccionar un número válido (validación de entrada).
- Si el usuario ingresa la opción 1 mostramos un texto.
- Si el usuario ingresa la opción 2 mostramos otro texto.
- Si el usuario ingresa la opción "salir" terminamos el programa.

Resuelve el ejercicio con lo aprendido y en el próximo capítulo revisaremos la solución. Antes de escribir el código desarrolla el diagrama de flujo.

Solución

Diagrama de flujo del Menú

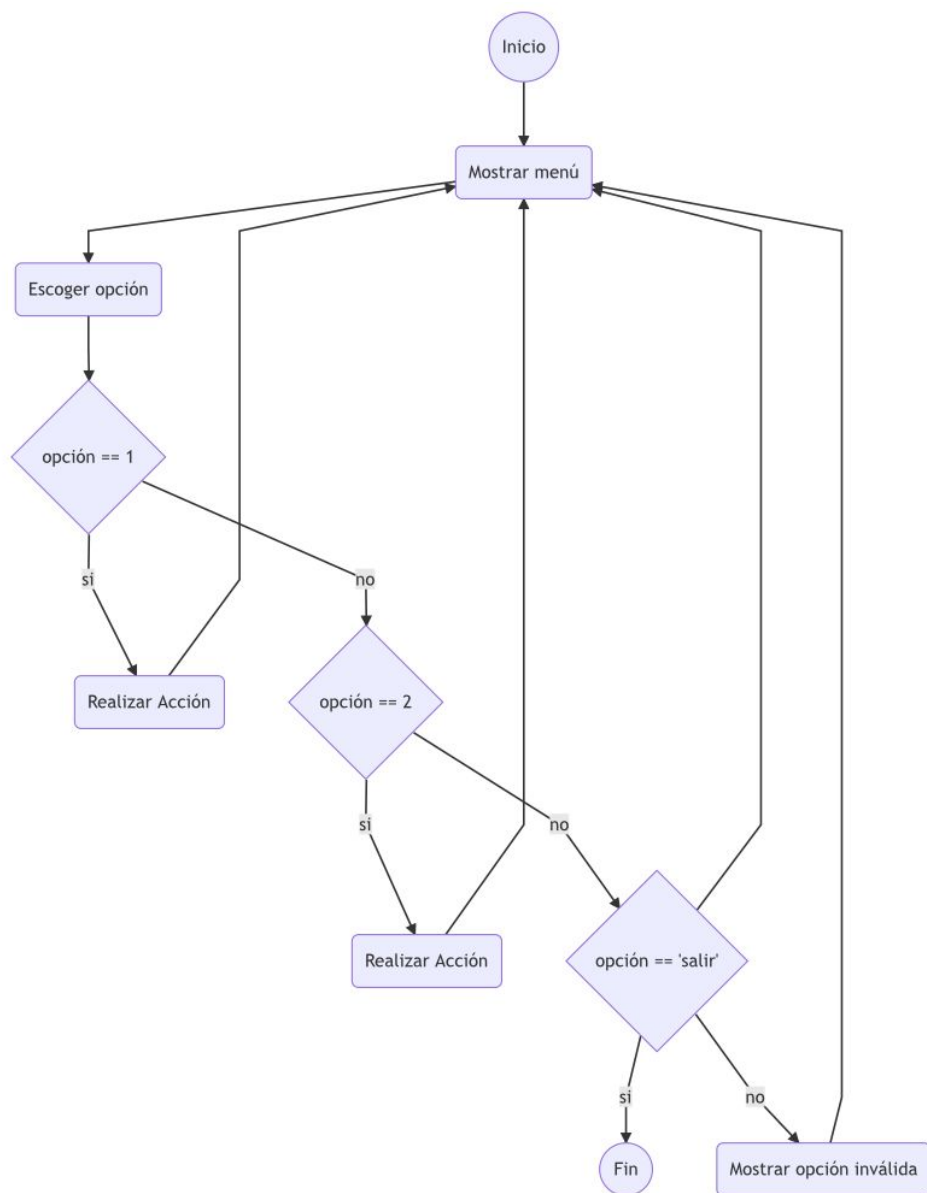


Imagen 4. Diagrama de flujo del Menú.

El código para un menú

```
opt_menu = 'cualquier valor'

while opt_menu != 'salir' and opt_menu != 'Salir':
    print('Escoge una opción')
    print('-' * 20)
    print('1 - Acción 1')
    print('2 - Acción 2')
    print('Escribe "Salir" para terminar el programa')

    opt_menu = input("Ingresa una opción")

    if opt_menu == '1':
        print('Realizando acción 1')
    elif opt_menu == '2':
        print('Realizando acción 2')
    elif opt_menu == 'salir' or opt_menu == 'Salir':
        print('Saliendo')
    else:
        print('Opción inválida')
```

Capítulo 2: Ciclos y contadores

Competencias

- Conocer el concepto de iteración.
- Contar la cantidad de veces que un programa está dentro de un ciclo.

En el capítulo anterior se resolvieron ejercicios de ciclos donde el punto de salida se producía cuando el usuario introducía un valor determinado.

En este capítulo se resolverán problemas de programación utilizando una cantidad determinada de ciclos. Por ejemplo, repetir un ciclo 10 veces, una cuenta regresiva, o problemas de sumatorias.

Iterar

Iterar es dar una vuelta al ciclo. No siempre se sabe cuántas iteraciones tendrá un ciclo. Por ejemplo, en el ejercicio de la contraseña, o en el ejercicio del menú.

Existen también muchos problemas que se pueden resolver de forma mucho más eficiente utilizando iteraciones finitas. Por ejemplo, mostrar un texto 10 veces, utilizando un contador.

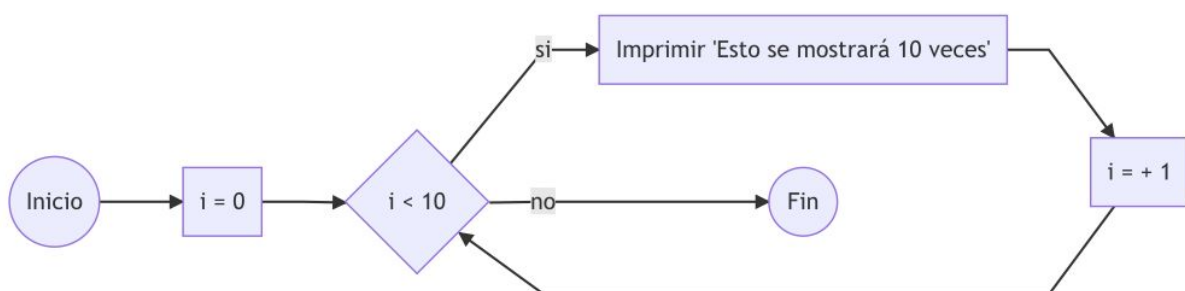


Imagen 5. Diagrama para mostrar un texto 10 veces.

Contando con while

```
i = 0
while i < 10:
    print("Esto se mostrará 10 veces") # está es la expresión a repetir
    i += 1 # IMPORTANTE
```

La instrucción `print("Esto se mostrará 10 veces")` se repetirá hasta que la variable `i` alcance el valor `10`.

Para entonces, la comparación de la instrucción `while` se evaluará como `False` y saldremos del ciclo.

En programación, es una convención utilizar una variable llamada `i` como variable de iteración para operar en un ciclo.

IMPORTANTE: Si no se aumenta el valor de la variable `i`, entonces ésta nunca llegará a ser igual o mayor a 10. Por ende, la comparación nunca se evaluará como `False`, y se producirá un loop infinito.

¿Qué significa `i+= 1`?

`+=` es un operador de asignación, muy similar a decir `a = 2`, pero la diferencia es que con `+=` se está asignando el valor anterior de la variable más 1.

```
a = 2

# Aquí el valor de a aumentó en dos y sobrescrito.
a += 2

print(a)
```

Operadores de asignación

La siguiente tabla muestra el comportamiento de los operadores de asignación:

Operador	Nombre	Ejemplo	Resultado
=	Asignación.	a = 2	a toma el valor 2.
+=	Incremento asignación.	a += 2	a es incrementado en dos y asignado el valor resultante.
-=	Decremento asignación.	a -= 2	a es reducido en dos y asignado el valor resultante.
*=	Multiplicación asignación.	a *= 3	a es multiplicado por tres y asignado el valor resultante.
/=	División asignación.	a /= 3	a es dividido por tres y asignado el valor resultante.

Tabla 2. Operadores de asignación.

Desafío: La bomba de tiempo

Realizaremos un ejercicio que realice una cuenta regresiva de 5 segundos.

Contar de forma regresiva es muy similar, solo que partiremos del valor e iremos bajando de uno a uno.

```
i = 5
while i > 0:
    print(i)
    i -=1
```

¿Y cómo hacemos para que sean segundos?. Existe una instrucción llamada `sleep` en el módulo `time`, que nos permite esperar un segundo antes de continuar.

```
import time

i = 5
while i > 0:
    i -=1 # En cada iteración descontamos 1.
    print(i)
    time.sleep(1)
    if i == 0:
        print("BOOM!")
```


Condiciones de borde

Cuando se realiza un ciclo con un contador, se debe tener cuidado con el valor con el cual se inicializa el ciclo, en cuánto y en qué momento se aumenta su valor, y qué operador se está utilizando para evaluar la condición de salida.

Por convención, los contadores se inician en 0, y de manera general, aumentan su valor en 1 al final de la iteración. Y la condición evalúa que el contador sea mayor a un valor, o menor a un valor.

Esto es de forma general, pero va a depender del caso a caso. En el ejercicio de la bomba, no se utiliza un contador, sino una variable inicializada en cierto valor, que debe llegar a 0, y al llegar a este valor, no volver a entrar al ciclo.

Por lo tanto, no es lo mismo la siguiente expresión lógica:

```
i = 5

while i > 0:
    i -= 1
    print(i)
    time.sleep(1)
```

Que:

```
i = 5

while i >= 0:
    i -= 1
    print(i)
    time.sleep(1)
```

Dado que en esta última la evaluación lógica `i >= 0` se sigue cumpliendo cuando cuando `i=0`.

Capítulo 3: Sumatorias

Competencias

- Crear código para fórmulas matemáticas del tipo sumatoria
- Aplicar un contador en un script.

Motivación

En este capítulo se aprenderá a crear programas que tengan ciclos donde además se opere sobre otra variable. Esto permite resolver diversos tipos de problemas, la mayoría de los cuales son matemáticos, pero también fomentará las habilidades necesarias para aplicar esta lógica a otros contextos.

Introducción a sumatorias

Para alguien que no le gusten las matemáticas, "sumatoria" puede ser un término que asuste. Pero lo único que se necesita para resolver una sumatoria, es saber sumar.

La sumatoria consiste en sumar todos los números de una secuencia. Por ejemplo, sumar todos los números entre 1 y 100. Esto sirve para resolver ecuaciones matemáticas, pero también sirve para generar las abstracciones necesarias para resolver diversos algoritmos.

Sumando de 1 a 100

$$1 + 2 + 3 + \dots + 100 = ?$$

Resolver esto es muy similar a contar cien veces. Pero además de contar, se debe ir guardando la suma acumulada en cada iteración.

Comenzaremos desde la base del código anterior.

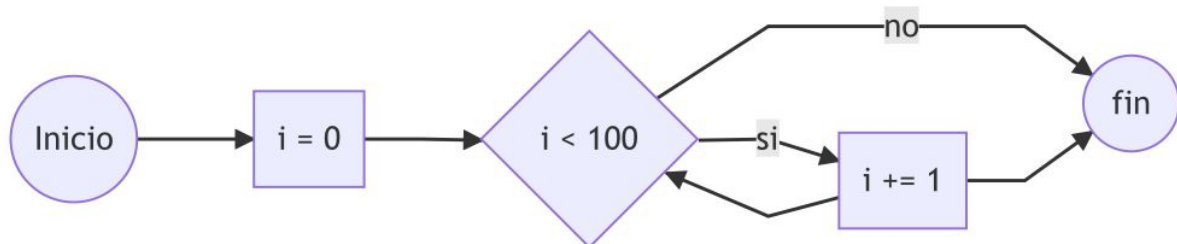


Imagen 6. Suma de 1 a 100.

```
i = 0

while i < 100:
    i += 1

print(i)
```

Luego, si quisiéramos ir guardando la suma en cada iteración, necesitamos una variable para ir guardando los datos. Esta variable la llamaremos `suma`.

```
i = 0

while i < 100:
    i += 1
    suma += i
print(i)
```

Todavía falta un detalle para que el código funcione. No se puede sumar algo a una variable que no tiene ningún valor. Por lo tanto, las variables que se utilicen para contar o para sumar, deben ser inicializadas asignándoles el valor de partida (0 en este caso), antes de iniciar el bucle.

```
i = 0
suma = 0

while i < 100:
    i += 1
    suma += i

print(i, suma)
```

Diagrama de flujo para reforzar lo aprendido:

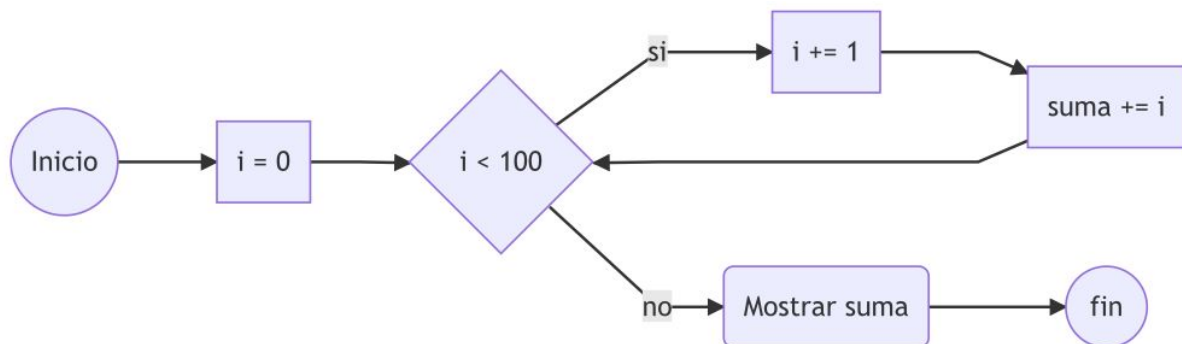


Imagen 7. Diagrama de la suma.

La instrucción `suma += i` es la encargada de aumentar el valor de la variable suma en cada iteración.

Este comportamiento, sumar y almacenar secuencialmente valores variables, se conoce como la implementación de un **acumulador**. En este ejemplo, la variable suma está acumulando la suma de los valores en cada iteración.

Desafío: Sumando de 1 a n números

suma_nb.py: El usuario debe ingresar un número, y se debe mostrar la sumatoria de todos los números desde 1 hasta el número ingresado.

Uso:

```
suma_n.py 100
```

```
5050
```

Solución al desafío de la suma de los n números

```
# codigo

import sys

limit = int(sys.argv[1])
i = 0
suma = 0

while i < limit:
    i += 1
    suma += i

print(suma)
```

Resumen del capítulo

En este capítulo trabajamos con contadores y acumuladores.

Tanto los contadores como acumuladores son ampliamente utilizados.

- **Contador:** Aumenta de 1 en 1.
 - `cont = cont + 1.`
 - `cont += 1.`
- **Acumulador:** Aumenta en función a `cont += 1.`
 - `acu = acu + valor.`
 - `acu += valor.`

Capítulo 4: Concatenación en ciclos

Competencias

- Aplicar un acumulador para concatenar textos.
- Generar una lista HTML utilizando while.

Acumulando strings

También es posible utilizar el operador de asignación `+=` para ir acumulando un string. Esto porque, como ya hemos visto, el operador `+` funciona como un concatenador al ser aplicado en tipos de dato string.

```
saludo = "hola"  
saludo += " mundo"  
print(saludo) # hola mundo
```

Se debe tener en cuenta eso sí, que como el operador de asignación `+=` trabaja sobre un valor ya existente, la variable debe estar previamente definida.

```
adios += "chao"
```

```
-----  
---  
  
NameError                                Traceback (most recent call  
last)  
  
<ipython-input-1-29165b1cc8c6> in <module>()  
----> 1 adios += "chao"
```

```
NameError: name 'adios' is not defined
```

Generar una lista en HTML

Se pide crear un programa donde el usuario ingrese un número como argumento y se genere una lista de HTML con esa cantidad de ítems.

Uso:

```
python lista_html.py 5
```

```
<ul>
  <li> 1 </li>
  <li> 2 </li>
  <li> 3 </li>
  <li> 4 </li>
  <li> 5 </li>
</ul>
```

Tip:

- Recordar que el operador `+` sirve para concatenar strings.
- Puedes tabular con `"\t"`.
- Puedes hacer un salto de línea con `"\n"`.

Solución al generador de listas

Lo primero es identificar las partes que se repiten y las que no dentro del código HTML. En este caso, las etiquetas `` son las que se repiten. En cambio, `` y `` solo aparecen al principio y al final.

Primero se codificará la lógica para el contenido que se repite. Se debe definir también la variable acumuladora (fuera del ciclo) donde se concatenará el código que se repite. Para esto crearemos el programa `lista_html.py`. Dentro se debe escribir:

```
import sys

html = ""
items = int(sys.argv[1])
i = 0

while i < items:
    i += 1
    html += "<li> item {} </li>\n".format(i)

print(html)
```

Al ejecutar el programa desde la consola con un argumento de 3 `python lista_html.py 3`, se obtiene:

```
<li> item 1 </li>
<li> item 2 </li>
<li> item 3 </li>
```

El siguiente paso es agregar el principio y el final de la lista:

```
import sys

html = "<ul>\n"
items = int(sys.argv[0])
i = 0

while i < items:
    i += 1
    html += "<li> item {} </li>\n".format(i)

html += "</ul>"
print(html)
```

Finalmente solo falta agregar las tabulaciones.

```
import sys

html = "<ul>\n"
items = int(sys.argv[0])
i = 0

while i < items:
    i += 1
    html += "\t<li> item {} </li>\n".format(i)

html += "</ul>"
print(html)
```

En este caso, el acumulador se utilizó para ir concatenando texto, en lugar de aplicarlo para una sumatoria. Esto es posible, ya que el operador `+` aplicado a textos los concatena. Lo mismo aplica para `+=`, lo cual va concatenando texto adicional a una variable existente.

Desafíos sugeridos (capítulos 3 y 4)

- `promedio.py`: El usuario debe ingresar número. El programa debe sumar todos los números desde 1 hasta el número ingresado. Luego, debe mostrar el resultado de la suma dividido por la cantidad de iteraciones.
- `promedio_usuario.py`: El usuario debe ingresar un número, el cual indica al programa la cantidad de datos que se van a ingresar. Luego, debe ingresar la cantidad de datos indicada. El programa debe mostrar el promedio de los datos ingresados a continuación del primer argumento.
- `suma_pares.py`: El programa debe sumar solo los números pares entre 1 y 100, y mostrar ese resultado. tip: Utilice `%` (módulo).
- `calculadora.py`: Crear un programa que permita ingresar de 4 opciones, identificadas con un número (1: sumar, 2: restar, 3: multiplicar, 4: dividir). Luego el usuario debe ingresar 2 números, sobre los cuales se realice la operación escogida. El programa debe mostrar el resultado.

Capítulo 5: for

Competencias

- Aprender la sintaxis del iterador `for`.
- Conocer la ventaja y desventaja de `for`.
- Conocer los rangos.
- Aplicar el uso de `break`.

Motivación

En programación, además de `while`, existen otras instrucciones para iterar. En este capítulo veremos el iterador `for` ("para"). Si bien con `while` es posible resolver cualquier problema de iteración, en algunos casos será más sencillo resolver el problema ocupando `for`.

El ciclo for

La instrucción `for` nos permite iterar en un rango. Podemos, por ejemplo, iterar de 1 a 10 con la siguiente sintaxis:

```
for i in range(10):  
    print("Iteración: ", i)
```

Es costumbre ocupar `i` como variable de iteración. Para el caso de `for`, `i` representa un elemento dentro de una estructura que contiene elementos; se iterará dicha estructura contenedora (para el ejemplo anterior, `range`), y en cada iteración, `i` representa el elemento de la estructura recorrido en esa iteración (para el ejemplo, en cada iteración, `i` representa un número desde el 0 hasta llegar a 9).

Dentro de la sintaxis de `for`, además del iterador `i` y de la estructura contenedora de elementos, es necesaria para su sintaxis la palabra `in`. De esta forma, la instrucción se puede leer en español como "Para - el elemento recorrido - en - la estructura contenedora".

Al igual que en otros casos, se debe terminar la sentencia con `:`, y el código a ejecutarse dentro de la iteración se escribe en la línea siguiente, indentando por 4 espacios.

Ventaja y desventajas de for

La principal ventaja de `for` es que no es necesario aumentar el contador, ya que este aumenta de forma automática en cada iteración. La desventaja principal, es que es menos flexible, porque no podemos cambiar el incrementador a nuestro antojo (siempre aumentará de 1 en 1).

Range

La instrucción `range()` devuelve una estructura contenedora, o "rango", con números en un rango específico. La documentación nos indica que `range` puede recibir 1, 2, ó 3 parámetros:

- Si solo se indica 1 parámetro, `range()` devolverá el rango de números desde el 0 hasta el número anterior indicado como parámetro. En este caso, se está utilizando el parámetro `stop`.
- Si se indican 2 parámetros, el primero indicará el número de partida del rango, y el segundo indica el número hasta cuál llegará el rango, pero también se le debe restar 1. En este caso, el primer parámetro se llama `start`, y el segundo `stop`.
- Si se indican 3 parámetros, los 2 primeros siguen la regla anterior, y el tercero indica el `step` o incremento para cada intervalo del rango (por ende, por defecto es 1).

Ocupando los parámetros de Range

```
print("Utilizando 1 parámetro (stop)")
for i in range(3):
    print(i) # 0 1 2

print()
print("Utilizando 2 parámetros (start y stop)")
for i in range(1, 3):
    print(i) # 1 2

print()
print("Utilizando 3 parámetros (start, stop y step)")
for i in range(0, 10, 2):
    print(i) # 0 2 4 6 8
```

Utilizando 1 parámetro (stop)

```
0  
1  
2
```

Utilizando 2 parámetros (start y stop)

```
1  
2
```

Utilizando 3 parámetros (start, stop y step)

```
0  
2  
4  
6  
8
```

Tipo de dato Range

El problema de esta función, es que nos devuelve es un tipo de dato bien especial, del tipo range. Si solo se ejecuta `range(5,10,2)`, el output no dice mucho:

```
type(range(5,10,2))
```

```
range
```

```
range(5,10,2)
```

```
range(5, 10, 2)
```

Para poder "ver" los elementos dentro del rango, se debe recorrer con un `for`:

```
for i in range(5,10, 2):  
    print(i)
```

```
1
```

Por lo general, las operaciones con `range` son limitadas. Si deseamos convertir esta estructura en una lista, lo podemos envolver en la función `list()`. Las listas son una estructura que, siendo también un tipo de objeto, tienen varios métodos asociados que permiten manipular sus elementos más fácilmente.

```
lista = list(range(10))  
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
type(lista)  
# list
```


¿Cómo salir de un ciclo **for**?

Previamente se vio que se puede terminar un ciclo **while** con una condición de salida. Se vio también que un ciclo **for** termina de recorrerse automáticamente al terminar de iterar sobre una estructura contenedora de elementos. Pero, ¿Es posible terminar un ciclo **for** "a propósito"?

La respuesta es **sí**. Pueden existir varios casos en que se requiera terminar el ciclo sin la necesidad de recorrer todos los elementos. Para hacerlo, se debe escribir la palabra **break**. Esta instrucción hace que el ciclo se detenga, y se continúe con la ejecución del programa de la misma forma que si el ciclo ya hubiese concluido.

Ejemplo

Un uso muy típico de esto, es cuando se está buscando un elemento específico en la estructura contenedora. Uno podría almacenar el elemento buscado, y terminar de recorrer los elementos sin uso de **break**, pero esto no tiene mucho sentido, sobre todo cuando se está trabajando con estructuras que almacenen muchos datos.

Código

```
buscar = 5

for i in range(10000000):
    if i == buscar:
        print("¡Elemento encontrado! Se saldrá del ciclo")
        break
    else:
        print("Elemento no encontrado")

print("Se ha salido del ciclo")
Elemento no encontrado
Elemento no encontrado
Elemento no encontrado
Elemento no encontrado
Elemento no encontrado
¡Elemento encontrado! Se saldrá del ciclo
Se ha salido del ciclo
```

Capítulo 6: Dibujando con ciclos

Competencias

- Reconocer patrones de repetición en un ciclo.
- Utilizar ciclos para dibujar patrones.

Cuando se trabaja con ciclos, determinar el patrón puede ser uno de los problemas más complejos de resolver para un principiante. Hay personas que pueden resolver estos problemas intuitivamente, pero la mayoría debe aprender a resolverlos.

En este capítulo se resolverá diferentes problemas, aumentando la complejidad, donde se busca identificar el patrón.

Desafío: Dibujando puntos

Crear el programa `solo_puntos.py` que dibuje `n` puntos donde `n` es un valor ingresado por el usuario al momento de ejecutar el script.

Uso:

```
python solo_puntos.py 5
```

```
resultado:  
*****
```

```
python solo_puntos.py 1
```

```
resultado:  
*
```

Solución 1

El primer intento para resolverlo podría ser con ifs.

```
import sys

n = int(sys.argv[1])

if n == 1:
    print('*')
elif n == 2:
    print('**')
elif n == 3:
    print('***')
elif n == 4:
    print('****')
elif n == 5:
    print('*****')
```

Sin embargo, es una solución muy limitada.

¿Qué sucedería si el usuario ingresa el valor 6?, o 7? o 100?

¿Será lo mejor programar todas las opciones hasta 100?

Esto es un problema de falta de escalabilidad. Dentro de las buenas prácticas de programación, se requiere que los programas sean escalables. Es decir, que se puedan adaptar a futuros requerimientos. Por ejemplo, en este caso, puede que el requerimiento original fuera que el número ingresado debe estar entre 1 y 5.

Si solo se considera esa limitante, el código anterior funciona perfectamente. Pero si en el futuro alguien solicita que el programa acepte cualquier número, significará que prácticamente se deba reescribir todo el código para cumplir con el requerimiento. Es por eso que al programar, siempre es bueno ir un poco más allá (sin excederse), porque puede que en el futuro eso sea de mucha utilidad.

Para este tipo de problemas, y para que nuestro programa sea escalable, se considera mucho mejor utilizar ciclos.

Solución 2

Para resolver el problema con ciclos, simplemente se debe detectar cuál es el patrón: Si el usuario ingresa 1, se dibuja 1 asterisco; Si el usuario ingresa 2, se dibujan 2 asteriscos.

Por tanto, lo que se debe hacer, es crear un ciclo que de tantas vueltas como el número ingresado, y, por cada vuelta, añadir un *. Para ello, primero se inicializa el contenedor de asteriscos como un texto vacío, y en cada vuelta de la iteración se le añade un asterisco con el operador +=.

Esta solución, o algoritmo, es aplicable a cualquier lenguaje de programación.

```
import sys

n = int(sys.argv[1])

contain = ""
for i in range(n):
    contain += '*'

print(contain)
```

Solución 3

En Python, lograr esto es aún más fácil, sin utilizar ciclos. Al igual que el operador + es aplicable a strings, actuando como un "pegamento", el operador * multiplica el texto del string donde se aplica. Por lo tanto, se logra el resultado simplemente multiplicando "*" con n.

```
import sys

n = int(sys.argv[1])

print('*' * n)
```

Desafío: Dibujando asteriscos y puntos

Se solicita crear el programa `asteriscos_y_puntos.py`, el cual debe dibujar asteriscos y puntos intercalados hasta `n`, donde `n` es un valor ingresado por el usuario al momento de ejecutar el script.

Uso:

```
python asteriscos_y_espacios.py 3
```

```
resultado:  
*.*
```

```
python asteriscos_y_espacios.py 4
```

```
resultado:  
*.*.
```

```
python asteriscos_y_espacios.py 5
```

```
resultado:  
*.*.*
```

Solución

El patrón es que los elementos en posición par (0, 2, 4, 6, ..., $2*N$) son asteriscos *, mientras que los elementos en posición impar (1, 3, 5, ..., $2*N + 1$) son puntos ..

Existen distintas formas de saber si un elementos es par. En Python, la más simple de saber esto es mediante la operación del "módulo" %. El módulo es el operador que nos permite obtener lo que en matemáticas conocemos como resto. Por lo tanto, podemos consultar si un número es par, preguntando si el módulo de dicho número en 2 es 0. Como esta consulta es una comparación, el resultado será booleano.

```
# En este caso va a ser falso, porque 3 tiene un resto
3 % 2 == 0
```

False

```
# En este caso va a ser verdadero porque 6 no tiene resto
6 % 2 == 0
```

True

Para resolver este ejercicio, se debe iterar consultando el valor del iterador i, puesto que se requiere que saber si se iterando sobre un número par o impar.

```
import sys

n = int(sys.argv[1])
contain = ''
for i in range(n):
    if i % 2 == 0:
        contain += '*'
    else:
        contain += '.'

print(contain)
```

```
*.*.*
```

Desafío: Patrón simétrico

Crear el programa `nickname.py` que dibuje un patrón de un "nick de msn" de 3 elementos.

```
nickname.py 3
```

```
★... ^\`... 
```

```
nickname.py 5
```

```
★... ^\`... ★... 
```

```
nickname.py 7
```

```
★... ^\`... ★... ^\`... 
```


Solución

```
import sys

# Almacenamos el argumento como int
limit = int(sys.argv[1])

# Almacenamos cada elemento del patrón en una variable
a = "★"
b = "... "
c = "\n "

# Inicializamos el acumulador
contain = ""

# Recorremos el rango hasta el límite más 1
for i in range(limit + 1):
    # Primer caso: Concatenar puntos si el elemento es impar
    if i % 2 != 0:
        contain += b

    # Segundo caso: Concatenar estrella si el elemento es divisible por
    4
    elif i % 4 == 0:
        contain += a

    # El resto serán las líneas
    else:
        contain += c

# Se imprime el acumulador
print(contain)
```

Capítulo 7: Ciclos anidados

Competencias

- Utilizar ciclos anidados para resolver problemas.
- Introducir el concepto de complejidad.

Introducción a ciclos anidados

Un ciclo anidado no es más que un ciclo dentro de otro ciclo. No existe un límite explícito sobre cuántos ciclos pueden haber anidados dentro de un código, aunque por cada uno aumentará la complejidad.

Se debe tener que la cantidad total de iteraciones será la multiplicación de cuántas iteraciones se haga en cada ciclo.

Nota: Dentro de las buenas prácticas de programación, se recomienda no usar más de 3 ciclos anidados.

Los ciclos anidados abren el abanico de los tipos de problemas que podemos resolver.

Escribiendo las tablas de multiplicar

Tabla de un número

Supongamos que queremos mostrar una tabla de multiplicar. Por ejemplo la tabla del 5. Esto se puede escribir como:

```
for i in range(10):  
    print("5 * {} = {}".format(i, (5 * i)))
```

```
5 * 0 = 0  
5 * 1 = 5  
5 * 2 = 10  
5 * 3 = 15  
5 * 4 = 20  
5 * 5 = 25  
5 * 6 = 30  
5 * 7 = 35  
5 * 8 = 40  
5 * 9 = 45
```

¿Cómo podríamos hacer para mostrar todas las tablas de multiplicar del 1 al 10?

Fácil, envolviendo el código anterior en otro ciclo que itere de 1 a 10.

Tablas de todos los números

```
for i in range(10):  
    for j in range(10):  
        print("{} * {} = {}".format(i,j, (j * i)))
```

```
0 * 0 = 0  
0 * 1 = 0  
0 * 2 = 0  
0 * 3 = 0  
0 * 4 = 0  
0 * 5 = 0  
0 * 6 = 0  
0 * 7 = 0  
0 * 8 = 0  
0 * 9 = 0  
1 * 0 = 0  
1 * 1 = 1  
...  
9 * 6 = 54  
9 * 7 = 63  
9 * 8 = 72  
9 * 9 = 81
```

En este caso, el código no es complejo, y se justifica el uso de dos for anidados, porque facilita el problema planteado. Un problema se vuelve complejo, entre otras cosas, si se van anidando ciclos de forma excesiva e innecesaria.

Capítulo 8: Dibujando con ciclos anidados

Objetivo

- Aplicar el uso de loops anidados.

Los ejercicios de dibujo son ideales para practicar ciclos y especialmente ciclos anidados.

Dibujando agua nieve

Se solicita crear un programa que cada vez que se ejecute genere un lienzo de "agua nieve" diferente.

Ejemplo:



Solución

```
import sys
import random # importamos random
# Amacemos el ancho del lienzo como int
width = int(sys.argv[1])
# Validamos que no sea menor a 10
if width < 10:
    width = 10
# Creamos el contenedor
output = ""
# Recorremos de 1 a 9 en el loop externo
for i in range(1, 10):
    # Creamos el primer número al azar, utilizando el iterador
    # y el ancho del usuario
    rand_number = random.randint(1, width)

    # Usamos el número creado para agregar espacios en blanco antes de
    la nieve
    output += " " * rand_number + "*" + "\n"

    # Recorremos el segundo loop con rango de 1 a i
    for j in range(1, i):
        # Creamos el segundo número al azar,
        # utilizando el segundo iterador y el ancho
        rand_number_2 = random.randint(j, width)
        # Concatenamos la lluvia siguiendo la misma lógica
        output += " " * rand_number_2 + "/" + "\n"
# Mostramos el lienzo creado
print(output)
```

En este caso, no hay un patrón, sino que siempre se va a generar un lienzo diferente, donde solo se puede modificar el ancho.

Para ello, se utiliza el `for` más interno para dibujar la lluvia, y el `for` más externo para dibujar la nieve.

Desafío de listas y sublistas

Se pide crear un programa donde el usuario ingrese un número como argumento, y se genere una lista de HTML con esa cantidad de ítems. Se debe además ingresar un segundo número, el cual indique la cantidad de sub ítems de cada lista.

Uso:

```
python lista_html.py 3 2
```

```
<ul>
  <li>
    <ul>
      <li> 1.1 </li>
      <li> 1.2 </li>
    </ul>
  </li>
  <li>
    <ul>
      <li> 2.1 </li>
      <li> 2.2 </li>
    </ul>
  </li>
  <li>
    <ul>
      <li> 3.1 </li>
      <li> 3.2 </li>
    </ul>
  </li>
</ul>
```

Tips:

- Puedes tabular con `"\t"`.
- Puedes hacer un salto de línea con `"\n"`.

Solución

Lo primero que se debe hacer es distinguir las partes que se repiten de las que no. Dentro de cada lista (``), hay un patrón que se repite:

```
<li> 1.1 </li>
<li> 1.2 </li>
```

Pero además, fuera de la lista, también hay otro patrón que se repite:

```
<li>
  <ul>
    <li> 1.1 </li>
    <li> 1.2 </li>
  </ul>
</li>
....
```

Por tanto, se debe utilizar 2 iteraciones; una para el ciclo interior y otra para el ciclo exterior.

```
#### Lista interna

```python
n_interno = 3

for i in range(n_interno):
 print(" {} ".format(i))
```

```
 0
 1
 2
```



## Lista externa

El siguiente paso es agregar el ciclo externo. Se debe agregar también los tags `<li></li>` de apertura y cierre general, los cuales (al no repetirse) deben concatenarse al principio y al final del código que va repitiéndose.

```
n_externo = 3
n_interno = 3
print("")
for i in range(n_externo):
 print("\t\n")
 print("\t\t")
 for i in range(n_interno):
 print("\t\t\t {1}, {0} ".format(i, j))
 print("\t\t")
 print("\t")
print("")
```

```


 2, 0
 2, 1
 2, 2

 2, 0
 2, 1
 2, 2

 2, 0
 2, 1
 2, 2


```

## Capítulo 9: Dibujando con ciclos consecutivos (no anidados)

### Competencia

- Aplicar el uso de ciclos consecutivos (no anidados).

### Desafío: Cuadrado vacío

Crear el programa `cuadrado_hueco.py`, el cual al ejecutarse debe recibir un número, y luego dibujar un cuadrado de ancho igual al número, dejando vacío el interior.

```
python cuadrado_hueco.py 3
```

```
resultado:

* *

```

```
python cuadrado_hueco.py 5
```

```
resultado:

* *
* *
* *

```

**Tip:** Identificar cuál es la parte que se repite o patrón.

## Solución

La parte superior e inferior son similares y siempre son 2. La parte del medio sin embargo contempla el resto del cuadrado.

Primero dibujaremos la parte superior e inferior.

```
n = 5

contain = ''
for i in range(n):
 contain += '*'

print(contain)
```

```

```

La parte del medio consta siempre de dos asteriscos; Uno al principio y el otro al final. El resto son espacios en blanco.

- Si `n` es 4, hay 2 asteriscos y 2 espacios.
- Si `n` es 5, hay 2 asteriscos y 3 espacios.
- Si `n` es 6, hay 2 asteriscos y 4 espacios.
- Si `n` es 7, hay 2 asteriscos y 5 espacios.

```
contain_middle = "*" # Se inicia con el primer asterisco

Se llena con espacios en blanco, que será la cantidad ingresada - 2
(se restan los
espacios destinados para los asteriscos del principio y del final)
for i in range(n - 2):
 contain_middle += " "

contain_middle += "*\n" # Se concatena con el asterisco final y se
agrega el salto de línea

print(contain_middle)
```

```
* *
```

Se debe repetir el proceso anterior "n-2" veces, porque se necesitan tantas filas como el número que se ha ingresado, pero 2 de estas filas serán llenadas solo de asteriscos (inicial y final). Por tanto, las filas con espacios en blanco serán n - 2.

```
n = 5

contain_middle = "*"
for j in range(n - 2):
 contain_middle += " "
contain_middle += "*\n"

print(contain_middle * (n - 2))
```

```
* *
* *
* *
```

Finalmente, se une todo el código agregando las filas inicial y final.

```
n = 5

contain_bounds = ''
for i in range(n):
 contain_bounds += '*'

contain_middle = "*"
for j in range(n - 2):
 contain_middle += " "
contain_middle += "*\n"

print(contain_bounds)
print(contain_middle * (n - 2))
print(contain_bounds)
```

```

* *
* *
* *

```

## Desafío: Nube y lluvia

Se hará una modificación del ejercicio de agua y nieve, para ver una aplicación de ciclos no consecutivos

```
import sys
import random # importamos random

Amacenas el ancho del lienzo como int
width = int(sys.argv[1])

Validamos que no sea menor a 10
if width < 10:
 width = 10

Creamos el con el ancho de la nube
output = "@" * width + "\n"

Recorremos de 1 a 9 en el loop externo
for i in range(1, 10):
 # Creamos el primer número al azar con el 80% del ancho
 start_random = int(0.8 * width)
 rand_number = random.randint(start_random, width)

 # Transformamos la nieve en nube
 output += "@" * rand_number + "\n"

Recorremos el segundo loop con rango de 1 a 10
for j in range(1,10):
 # Creamos el segundo número al azar,
 # utilizando el segundo iterador y el ancho
 rand_number_2 = random.randint(j, width)

 # Concatenamos la lluvia siguiendo la misma lógica
 output += " " * rand_number_2 + "/" + "\n"

Mostramos el lienzo creado
print(output)
```

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

/

/

/

/

/

/

/

/

/

## Desafío: Triángulo

```
*
**

**
*
```

En este ejercicio aplicaremos:

1. Un ciclo for simple.
2. Un ciclo for anidado.
3. Aplicar ambos ciclos anteriores en forma consecutiva.

### Mitad superior

- Si el usuario ingresa 5, se dibujan 5 filas.
- En la fila 1, hay 1 \*.
- En la fila 2, hay 2 \*.
- Por lo que podemos decir que en la fila n hay n \*.

Se debe llenar cada fila con asteriscos según el número de fila donde se encuentra la iteración. Al final se concatena un salto de línea.

Como en la primera vuelta del primer `for` el elemento `i` vale "0", no entra en el segundo `for` durante esta vuelta (porque no hay rango de 0). Por lo tanto, para lograr entrar en el segundo `for` las veces necesarias para dibujar los asteriscos requeridos, debemos sumar "1" al valor ingresado por el usuario.

```
n = 5

contain=''
for i in range(n + 1): # Se suma 1, porque en la primera vuelta no
 ingresa en el segundo for (no hay range de 0)
 contain += "*" * i + "\n"

print(contain)
```

```

*
* *
* * *
* * * *
* * * * *

```

## Mitad inferior

Se invierte el patrón anterior; Si  $n$  es igual a 5, primero se debe dibujar 5 asteriscos, luego 4, luego 3; Se va decrementando de uno en uno. Esto es lo mismo que decir  $n - i$ ;

- Iteración 0:
  - $5 - 0 = 5$ .
- Iteración 1:
  - $5 - 1 = 4$ .



```
n = 5
contain=''
for i in range(n):
 for j in range(n - i):
 contain += '*'
 contain += '\n'
print(contain)
```

```


**
*
```

Finalmente, para armar el triángulo completo, solo se necesita juntar ambas soluciones como ciclos consecutivos.

```
contain=''
for i in range(n + 1):
 contain += "*" * i + "\n"
print(contain)
contain=''
for i in range(n):
 for j in range(n - i):
 contain += '*'
 contain += '\n'
print(contain)
```

```
*
**

**
*
```

## Capítulo 10: Reutilizando código

### Competencias

- Conocer la importancia de las funciones.
- Conocer la diferencia entre definir una función y llamar una función.

### Motivación

Las funciones nos permiten abstraer cómo resolver los problemas. Esto lo logramos agrupando instrucciones bajo un nombre.

### Introducción a funciones

Las funciones nos permiten agrupar instrucciones y reutilizarlas.

Supongamos que tenemos el siguiente algoritmo:

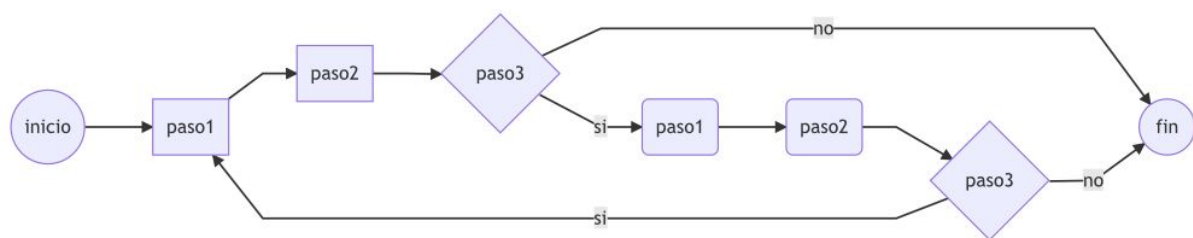


Imagen 8. Diagrama de ejemplo de algoritmo.

Si se renombran el paso1, el paso2 y el paso3 a "PasoX", ¿Sería lo mismo?.

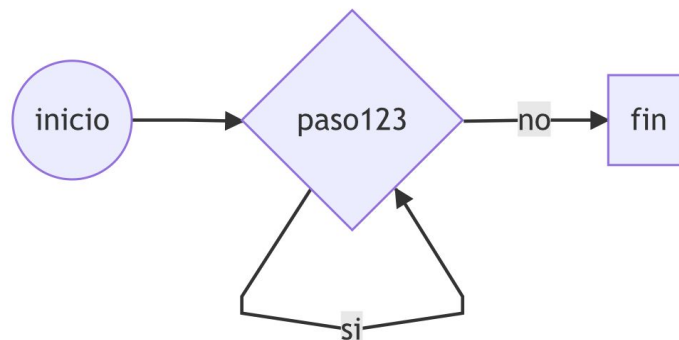


Imagen 9. Renombrando pasos..

La forma de hacer esto en distintos lenguajes recibe distintos nombres:

- procedimiento.
- subrutina.
- función.
- método.

En Python se llaman funciones. Y aunque existen diferencias entre procedimiento, subrutina, función y método, las ignoraremos por ahora puesto que en Python solo se trabaja con funciones (si se habla de método en algún momento, se entiende como sinónimo de función).

Podemos crear nuestras propias funciones. A esto le llamaremos **definir**.

También podemos ocupar métodos ya creados por nosotros, o por otras personas, a esto le denominaremos **llamar**.

## Llamar una función

En Python hay varios métodos definidos que se pueden utilizar o llamar, como por ejemplo:

- `print()`.
- `len()`.
- `input()`.

Existen distintos lugares donde las funciones se pueden llamar:

- Directamente en el espacio principal de trabajo, como `print()` y `sum()` y `len()`.
- Dentro de objetos, como `"hola".upper()`.
- Dentro de clases, como `LinearRegression().fit()`.
- Dentro de módulos, como `math.sqrt()`.

Si bien todavía no se ha explicado en profundidad la diferencia entre objetos, clases y módulos, lo importante es ver la sintaxis:

- **Opción1:** `función()`
- **Opción2:** `objeto.función()`
- **Opción3:** `Clase.función()`
- **Opción4:** `módulo.función()`

En esta unidad se trabajará definiendo funciones en el espacio principal de trabajo.

## Características de las funciones

- Se organizan en torno a clases y módulos.
- Se pueden importar desde un archivo o desde librerías externas.

En los próximos capítulos se enseñará a crear funciones propias, y a utilizar administradores de librerías para agregar funcionalidades.

## Capítulo 11: Creando funciones

En este capítulo se enseñará a crear funciones desde cero, o funciones propias. Esto permitirá simplificar código ya escrito, y reutilizar código. Las funciones son base del enfoque DRY (Don't Repeat yourself).

### Competencias

- Definir funciones.
- Llamar funciones.
- Conocer el orden en que se ejecuta un código que tiene funciones.

### Definiendo una función

En Python, una función se define utilizando la siguiente sintaxis:

```
def nombre_del_metodo(): # Definimos con def y un nombre. Se agrega ()
 al final del nombre, y luego :
 # Serie de instrucciones que ejecutará el método. (identado por 4
 espacios)
 # ...
 # ...
```

Todas las instrucciones definidas dentro de la función serán ejecutadas cuando ésta sea llamada.

## Creando nuestra primera función

```
def imprimir_menu():
 print('Menú: ')
 print('1) Opción 1')
 print('2) Opción 2')
 print('3) Opción 3')
 print('4) Salir')
```

### Llamando a la función creada

A los amigos los llamamos por su nombre, a las funciones de la misma forma. Si se quiere imprimir el menú, se llama a la función escribiendo `imprimir_menu()`. Un aspecto a considerar, es que para llamar a una función, **se debe incluir los paréntesis**. Si estos no son incluidos, y se solicita a Python `imprimir_menu`, se obtendrá una salida similar a esta:

```
imprimir_menu
```

```
<function __main__.imprimir_menu()>
```

Esta salida informa sobre características de la función creada, como su ámbito (la función existe dentro del ambiente de trabajo principal, definido como `__main__`). Si se vuelve a ejecutar la función con paréntesis, se obtendrá los `prints` que se escribieron en la definición.

```
imprimir_menu()
```

```
Menú:
1) Opción 1
2) Opción 2
3) Opción 3
4) Salir
```

El orden de ejecución

En el momento se ejecuta un código en Python, éste se lee de arriba hacia abajo. Por tanto, Python requiere primero leer las definiciones para posteriormente utilizarlas.

Llamar a una función antes de definirla genera un error

```
x()

def x():
 print("hola")
```

```


NameError Traceback (most recent call
last)

<ipython-input-5-74e1aab057eb> in <module>()
----> 1 x()
 2
 3 def x():
 4 print("hola")
```

```
NameError: name 'x' is not defined
```

Esto es equivalente a ocupar una variable antes de asignarle un valor.

## Ejemplo integrado: Definir y llamar una función

En capítulos anteriores se vio cómo crear un menú de opciones utilizando un ciclo `while`.

A continuación se muestra cómo se abstrae el código que dibuja el menú en la función `imprimir_menu()`, y luego esta función se llama dentro del `while`. El proceso de abstraer código para crear una función también es parte de la refactorización.

```
def imprimir_menu():
 print('Menú: Escoja una acción')
 print('-' * 20)
 print('1) Acción 1')
 print('2) Acción 2')
 print('Escribe "Salir" para terminar el programa')
 print()

opt_menu = 'cualquier valor'

while opt_menu != 'salir' and opt_menu != 'Salir':
 imprimir_menu() # Se llama a la función

 opt_menu = input()

 if opt_menu == '1':
 print('Realizando acción 1')
 elif opt_menu == '2':
 print('Realizando acción 2')
 elif opt_menu == 'salir' or opt_menu == 'Salir':
 print('Saliendo')
 else:
 print('Opción inválida')
 print()
```



## Capítulo 12: Parametrizando funciones

### Competencias

- Aprender a crear funciones que reciban parámetros.
- Diferenciar parámetros de argumentos.
- Crear funciones con parámetros opcionales.

### ¿Qué es un parámetro?

Los parámetros son variables locales de una función, definidas junto con ella, y que se asume serán utilizadas dentro del bloque de la función.

Se definen dentro de los paréntesis, y puede o no asignarles un valor en esta definición; Si no se les asigna un valor, el parámetro será de carácter obligatorio, y se le debe asignar un valor al llamar a la función. Si el parámetro se define con un valor asignado, entonces será de carácter opcional.

No es obligación que una función se defina con parámetros. Ejemplo de ello es la función `upper()` de los objetos string.

### Tipos de parámetros

Como se mencionó, existen también funciones cuyo parámetro es opcional, como en el caso de `input()`; Uno puede de forma opcional incluir un mensaje para mostrar al usuario en pantalla cuando se requiere que introduzca el valor, y la forma de hacerlo, es pasando el mensaje como parámetro en la función `input()`. En cambio, si no se añade ningún texto, no se mostrará mensaje al usuario, pero la función se ejecutará de todos modos.

Por último, en otros casos, los parámetros son obligatorios. Ejemplo de ello es la función `count()` de los objetos string, que se explica a continuación.

## Ejemplo de una función con parámetro obligatorio

Se desea contar la cantidad de ocurrencias de la letra p en el string 'paralelepipedo'. Para ello, se puede utilizar el método `count()`. Sin embargo, si no se indica la letra p dentro de los paréntesis, Python informará que necesita un argumento y éste no se ha entregado.

```
'paralelepipedo'.count()
```

```


TypeError Traceback (most recent call
last)

<ipython-input-6-da4c334d2771> in <module>()
----> 1 'paralelepipedo'.count()
```

```
TypeError: count() takes at least 1 argument (0 given)
```

Para poder ejecutar esta función correctamente, se necesita indicar de forma explícita qué letra se va a buscar.

```
'paralelepipedo'.count('p')
```

```
3
```

La 'p' dentro de la función `count` es el parámetro requerido. Este comportamiento permite flexibilizar de gran manera la utilidad de una función.

## Creando una función con parámetro obligatorio

Para crear una función que reciba un parámetro, se debe especificar dentro de los paréntesis el o los parámetros que **debe** recibir, sin asignarle un valor.

Al definir la función con parámetros, se está definiendo tanto la función como las variables que se utilizarán en ella, pero sin asignarles un valor inicial.

### Ejemplo

Si se desea crear una función que incremente un número en una unidad, se puede implementar de la siguiente manera:

```
def incrementar_en_uno(numero):
 total = numero + 1
 print("El resultado es: ", total)
```

### Una función puede recibir más de un parámetro

Se puede especificar todos los parámetros que se requiera para la función. Para ello, estos se escriben separándolos por una coma.

**Nota:** Dentro de las buenas prácticas de programación, se estila que una función no tenga más de 5 parámetros. Si se requiere de más de 5 parámetros, en teoría, habría que refactorizar parte del código de la función en otra (llamar una función dentro de otra es perfectamente válido). De todas formas, existen casos donde de todas formas se requiera usar más parámetros (recordar que las buenas prácticas son sugerencias, no normas).

```
def incrementar_por_cantidad(numero, cantidad):
 total = numero + cantidad
 print("El número es: ", total)

incrementar_por_cantidad(2, 3)
```

```
El número es: 5
```

## Llamando una función propia con parámetro obligatorio

De manera similar a como se vio con la función `count`, si no se ingresa el parámetro exigido, Python informará que la función `incrementar_en_uno` necesita un **argumento posicional** llamado `'numero'`.

```
incrementar_en_uno()
```

```


TypeError Traceback (most recent call
last)

<ipython-input-12-ea1a5f1be123> in <module>()
----> 1 incrementar_en_uno()
```

```
TypeError: incrementar_en_uno() missing 1 required positional argument:
'numero'
```

## Probando la función

```
incrementar_en_uno(10)
```

```
El resultado es: 11
```

```
incrementar_en_uno(123465)
```

```
El resultado es: 123466
```

## Argumentos

En el último error, Python informó `TypeError: incrementar_en_uno() missing 1 required positional argument: 'numero'`.

¿Por qué Python habla de argumentos si lo que se ingresaba en una función eran parámetros?

```
def incrementar_en_uno(numero):
 total = numero + 1
 print("El número es: ", total)

incrementar_en_uno(3)
```

El diagrama muestra el código de la función `incrementar_en_uno` con el parámetro `numero` y la llamada a la función `incrementar_en_uno(3)` con el argumento `3`. Una etiqueta **parámetro** con una flecha azul apunta al `numero` en la definición de la función. Otra etiqueta **argumento** con una flecha azul apunta al `3` en la llamada a la función.

Imagen 10. Parámetro y argumento.

- Porque las variables en la definición de la función se denominan **parámetro**.
- Al llamar a la función, el valor que se asigna al parámetro se denomina **argumento**.

Una variable también puede ser usada como argumento

```
def incrementar_en_uno(numero):
 total = numero + 1
 print("El número es: ", total)

a = 4
incrementar_en_uno(a)
```

```
El número es: 5
```

Al llamar a la función, se debe incluir todos los argumentos requeridos

```
incrementar_por_cantidad(2)
```

```


TypeError Traceback (most recent call
last)

<ipython-input-18-e0beb53d956c> in <module>()
----> 1 incrementar_por_cantidad(2)
```

```
TypeError: incrementar_por_cantidad() missing 1 required positional
argument: 'cantidad'
```

## Definir una función propia con parámetros opcionales


Para crear una función que reciba parámetros opcionales, se debe asignar un valor al parámetro en la definición de la función. De esta forma, al llamar a la función, si no se indica un argumento para el parámetro, la función utilizará el valor que se le ha asignado por defecto (el que se le asignó en su definición).

También es válido crear funciones con parámetros obligatorios y opcionales. Para que esto tenga sentido, los parámetros opcionales deben definirse al final de la lista de parámetros.

En este caso, solo uno de los valores es opcionales. También se puede crear una función donde todos los parámetros sean opcionales.

```
def incrementar_por_cantidad(numero, cantidad = 2):
 total = numero + cantidad
 print("El número es: ", total)

incrementar_por_cantidad(5) # 7
incrementar_por_cantidad(5, 1) # 6
```



Un valor opcional  
tiene un valor asignado  
por defecto

Imagen 11. Parámetros opcionales.

Llamando una función propia con parámetros opcionales

```
def incrementar_por_cantidad(numero, cantidad = 2):
 total = numero + cantidad
 print("El número es: ", total)

incrementar_por_cantidad()
```

```


TypeError Traceback (most recent call
last)

<ipython-input-20-4e3e548da779> in <module>()
 3 print("El número es: ", total)
 4
----> 5 incrementar_por_cantidad()
```

```
TypeError: incrementar_por_cantidad() missing 1 required positional
argument: 'numero'
```

Si no se asigna ningún argumento ocurre el error `TypeError: incrementar_por_cantidad() missing 1 required positional argument: 'numero'`. Python en este contexto necesita un argumento posicional (que no puede ser omitido).



## Las funciones deben ser reutilizables

Esto tiene que ver con el concepto de escalabilidad que se vio en el capítulo 6. Al programar, siempre se debe tener en cuenta que el código sea lo más flexible posible para poder seguir siendo utilizado en el futuro, sin tener que re-escribirlo. Por lo tanto, al definir una función, se debe intentar que esta sea lo más flexible posible en cuanto a su funcionamiento. En la práctica, se ve que esto se logra en gran medida por el uso de parámetros.

### Ejemplo

Se presentan dos formas de definir una función para transformar de grados Fahrenheit a grados Celcius.

# Opción 1:

```
def fahrenheit():
 fahrenheit = int(input("Ingrese la temperatura en Fahrenheit"))
 celsius = (fahrenheit + 40) / 1.8 - 40
 print("La temperatura es de {} grados Celsius".format(celsius))
```

# Opción 2:

```
def fahrenheit(f):
 celsius = (f + 40) / 1.8 - 40
 print("La temperatura es de {} grados Celsius".format(celsius))

fahrenheit(int(input("Ingrese la temperatura en Fahrenheit")))
```

- ¿Cuál de las dos es más flexible?
- ¿Qué pasaría si ya no se quiere obligar al usuario ingresar un valor?

La segunda forma es más flexible, porque permite utilizarla independiente de cómo se ingresen los valores.

Por ejemplo, se puede llamar utilizando un valor al azar

```
import random

def fahrenheit(f):
 celsius = (f + 40) / 1.8 - 40
 print("La temperatura es de {} grados Celsius".format(celsius))

fahrenheit(random.randint(1, 100))
```

La temperatura es de 31.66666666666667 grados Celsius

Y también se puede llamar utilizando un valor entregado por el usuario, el cual se puede previamente guardar en una variable:

```
far_user = int(input("Ingrese temperatura en Fahrenheit"))

fahrenheit(far_user)
```

## Ejercicio propuesto

```
def validar_edad():
 edad = int(input("Ingresar edad: "))
 if edad >= 18:
 print("es mayor")
 else:
 print("es menor")
```

- Modificar la función para que reciba la edad como parámetro obligatorio.
- Llamar 3 al método con edades generadas al azar.

## Resumen

- Los parámetros permiten hacer las funciones más flexibles.
- Algunos parámetros pueden ser opcionales. Para ello se les debe asignar un valor por defecto en la definición de la función.

```
def prueba(x = 2):
 print(x)
```

```
prueba()
prueba(3)
```

```
2
3
```

- Las variables pueden ser usadas como argumentos.

```
def prueba(x = 2):
 print(x)
```

```
a = 5
prueba(a)
```

```
5
```

## Capítulo 13: Retorno

### Competencias

- Entender la importancia del retorno de una función.
- Crear funciones con retorno.
- Saber que los retornos en Python deben ser explícitos.

### Motivación

En muchas situaciones necesitaremos utilizar el resultado de una función para seguir trabajando con ésta. Por lo mismo, solo en escasas ocasiones las funciones muestran información directo en pantalla. Normalmente, retorna el valor para poder seguir trabajando.

También será muy frecuente en nuestro trabajo que nos pidan una función que reciba cierta información y devuelva algo.

### Qué es el retorno y por qué se utiliza

Las funciones pueden recibir parámetros y pueden devolver un valor. A este valor se le conoce como **retorno**. A continuación, en la misma línea, se debe escribir lo que se desea retornar. Todo código que se escriba en las líneas siguientes a donde se declara el return, no será ejecutado.

A diferencia de otros lenguajes, en Python el retorno **debe ser explícito**. Para clarificar este punto, tomaremos la siguiente variante de la función `transformar_a_fahrenheit`.

Creando nuestra primera función con retorno

```
def transformar_a_fahrenheit(f):
 celsius = (f + 40) / 1.8 - 40
 return celsius
```

De esta forma, si se desea mostrar el resultado de la función `transformar_a_fahrenheit`, se debe envolver la función en un `print()`:

```
print(transformar_a_fahrenheit(110))
```

```
43.33333333333333
```

Al separar la instrucción `print()` de la función, esta es mucho más flexible.

```
def transformar_a_fahrenheit(f):
 celsius = (f + 40) / 1.8 - 40
 return celsius

print(transformar_a_fahrenheit(110))
```

Imagen 12. Función `transformar_a_fahrenheit`.

```
def transformar_a_fahrenheit(f):
 celsius = (f + 40) / 1.8 - 40
 celsius

transformar_a_fahrenheit(110)
```

Esta expresión se conoce como **retorno implícito**. El problema con esto en Python, es que por defecto el valor `celsius` vive sólo dentro del bloque. Si se desea extraer el resultado de la expresión al ambiente de trabajo, se debe utilizar **return**. Esto convierte a una función en una función con **retorno explícito**.

```
def transformar_de_fahrenheit(f):
 celsius = (f + 40) / 1.8 - 40
 return celsius
```

Al implementar **return** en una función, se asegura que su resultado pueda seguir siendo implementado. También se puede guardar el resultado de la función en una nueva variable.

```
grados_celsius = transformar_de_fahrenheit(110)

print("La variable grados_celsius es del tipo: ", type(grados_celsius))
print(grados_celsius)
```

```
La variable grados_celsius es del tipo: <class 'float'>
43.33333333333333
```

También es posible realizar operaciones con el retorno de esta función. Supongamos que se desea corregir el cálculo al incrementar en 10:

```
grados_celsius + 10
```

```
53.33333333333333
```

**return** sale de la función

Todo lo que viene después de la instrucción **return** es ignorado.

```
def prueba_return():
 a = "Esta línea se va a imprimir"
 b = "Esta línea no se va a imprimir"
 return a # Punto de salida
 print(b)

prueba_return()
```

```
'Esta línea se va a imprimir'
```

## Resumen

- Retorno, devolver, salida son sinónimos para el valor que devuelve una función.
- Podemos hacer que una función devuelva un valor utilizando la instrucción **return**.
- Retornar **NO ES LO MISMO** que mostrar en pantalla.

## Capítulo 14: Alcance de variables

### Competencias

- Conocer los tipos de variables.
- Conocer el concepto de alcance.
- Diferenciar variables locales de variables globales.
- Entender qué sucede con las variables dentro de un método.

### Motivación

Los tipos de variable y el alcance de éstas son conceptos muy importantes. Nos permite entender desde dónde se puede acceder a una variable.

En este capítulo se estudia las reglas de alcance de las variables locales y globales.

### Alcance y tipos de variables

El `transformar_a_fahrenheit` (Scope), es desde dónde podemos acceder a la variable.

### Tipos de variable

En Python existen 4 tipos de variables:

- globales.
- locales.
- de instancia.
- de clase.

Las últimas dos se ocupan dentro de la creación de objetos, por lo que aún no serán abordadas. Nos enfocaremos en las variables globales y locales.



## Variables locales

Son aquellas definidas dentro de un contexto específico, como una función. Para la función, serán variables locales aquellas que se definan dentro de su bloque, y también sus parámetros.

### Alcance de una variable local

Una variable definida dentro de una función, al ser local, no puede ser accedida fuera de ésta (solo existe en el scope de la función donde se definió).

```
def aprobado(promedio, nota_aprobacion = 4):
 if promedio >= nota_aprobacion:
 status = True
 else:
 status = False
 return status

print(status)
```

```


NameError Traceback (most recent call
last)

<ipython-input-19-48c6094df5ce> in <module>()
 6 return status
 7
----> 8 print(status)
```

```
NameError: name 'status' is not defined
```

Python arrojará un `NameError: name 'status' is not defined`, dado que `status` **no existe fuera de la función** `aprobado`.

Esto se debe a que cada función define su propio espacio de trabajo (o scope).

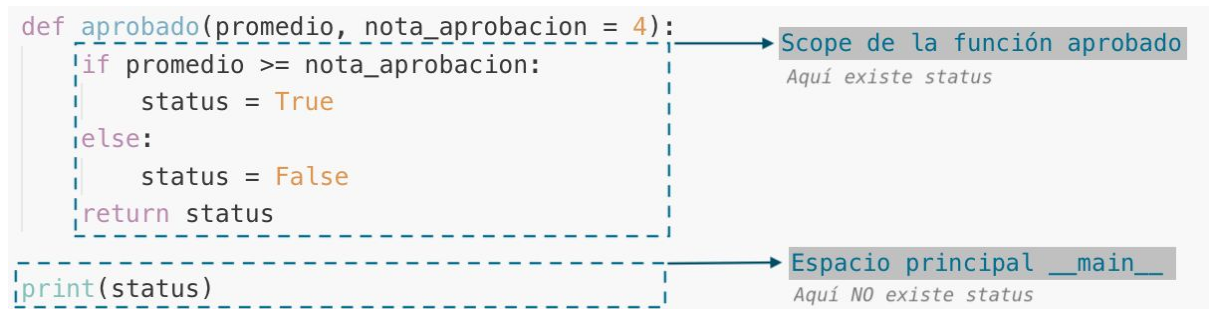


Imagen 13. Scope y espacio principal,

## Variables globales

A diferencia de otros lenguajes, en Python todas las variables que se encuentran definidas en el espacio principal de trabajo funcionan como globales. El espacio principal de trabajo recibe el nombre de **main**, por lo que al hablar de espacio `__main__`, nos referimos a este contexto.

### Alcance de una variable global

Como lo indica el nombre, se puede acceder a una variable global desde todos los scopes de trabajo.

```
continent = 'South America'

def print_continent():
 print(continent)

print_continent()
```

```
South America
```

La variable `continent`, al estar definida como una variable global, puede ser utilizada en diferentes ambientes. Esto es porque el scope de `print_continent` **es parte** del scope de `__main__`, por ende, puede acceder sin problemas a las variables creadas ahí.

## Asignación de un nuevo valor a una variable global en un contexto local

Ya se vio que es posible acceder a una variable declarada globalmente dentro del scope de una función, y que ésta tendrá aquel valor que se le asignó en el espacio `__main__`.

### Ejemplo de uso conjunto de variables locales y globales

```
estas variables se definen dentro del ambiente __main__
name = 'Alan Turing'
age = 41

def any_function():
 # Esta variable está siendo definida en un ambiente nuevo: el de
 any_function
 birthplace = 'Londres'

 # Como age es una variable global, se puede acceder desde este scope
 # Y también se puede acceder a birthplace, porque aunque sea local,
 se está en su mismo scope
 print("Edad de", age, "años. Residencia: ", birthplace)

Acá se vuelve al __main__ (se elimina la indentación de 4 espacios)
La variable birthplace no existe en este espacio.

print("Nombre: ", name)
any_function()
```

```
Nombre: Alan Turing
Edad de 41 años. Residencia: Londres
```

Pero, ¿Qué ocurre al asignar un nuevo valor a `age` dentro de `any_function()`? Si se vuelve a llamar a `age` en el espacio `__main__`, luego de llamar a `any_function()`, ¿Tendrá su valor original, o el asignado dentro de la función? La mejor forma de comprobarlo es ejecutando el código.

```
age = 41

def any_function():
 # Se asigna un nuevo valor
 age = 100
 print("En el scope de la función, age tiene el valor de:", age)

any_function()
print("Luego de ejecutar any_function(), tiene el valor de:", age)
```

```
En el scope de la función, age tiene el valor de: 100
Luego de ejecutar any_function(), tiene el valor de: 41
```

Se puede ver que, si bien dentro de la función el valor de `age` fue reescrito (primer `print`), al ser consultada nuevamente, de vuelta en el espacio `__main__` (segundo `print`), ésta conserva el valor que se le asignó originalmente.

Esto es porque en Python (a diferencia de otros lenguajes), cuando se ingresa en una función, se está trabajando en un ambiente nuevo que, aunque posea variables con el mismo nombre de una global, **no afecta las variables 'externas'**. Es decir, al utilizar el nombre de una variable global dentro de un contexto local y asignarle un nuevo valor, simplemente **se está declarando una nueva variable**, que funcionará en el scope local.

## Otro ejemplo

```
estas variables se definen dentro del ambiente __main__
name = 'Alan Turing'
age = 41
birthplace = 'Maida Vale, London, England'
gender = "male"

def say_hi():
 # Estas variables están siendo definidas en un ambiente nuevo: el de
 any_function
 # Las variables name y age son "variables nuevas",
 # que no tienen relación con las variables globales definidas
 previamente.
 name = 'Ian MacKaye'
 age = 56
 occupation = 'Musician'

 print("{} is a {} and is {} years old. He is {}".format(name,
 occupation, age, gender))

name y age tomarán los valores de su scope local
say_hi()

name y age toman los valores del scope main
print("{} was born in {} and died when he was {} years old. He was
{}".format(name, birthplace, age, gender))
```

Ian MacKaye is a Musician and is 56 years old. He is male.  
Alan Turing was born in Maida Vale, London, England and died when he was  
41 years old. He was male.

Por consiguiente, podemos afirmar que:

- `birthplace` y `gender` son variables globales, cuyo alcance es `__main__` (`say_hi` está dentro de `__main__`).
- `occupation` es una variable local, cuyo alcance es el método `say_hi`.
- `birthplace` no existe dentro del método `say_hi`, pero se podría acceder a ella desde `say_hi`, porque es global.
- `occupation` no existe fuera del método `say_hi`, y **no** se puede acceder a ella desde `main`, porque es local de `say_hi`.
- `name` y `age` **existen dos veces**; En un contexto global, cuyo alcance es `__main__`. Y en un contexto local, cuyo alcance es `say_hi`.
- `gender` es una variable global.

Otra forma de entender esto, es entender a los scopes (solo con fines pedagógicos) como "cajas dentro de cajas":

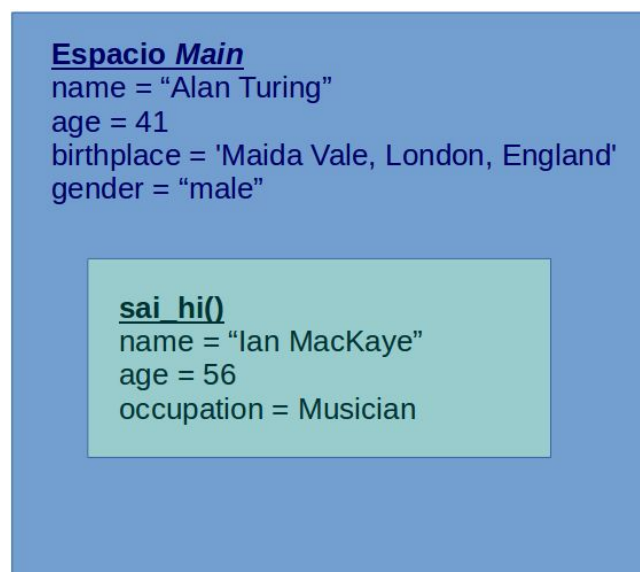


Imagen 14. Cajas dentro de cajas.

## Resumiendo el flujo

Al llamarse y ejecutarse `say_hi()`, dentro de análisis léxico, Python hará lo siguiente:

1. Creará la variable local `name`, cuyo valor será 'Ian MacKaye'.
2. Creará la variable local `age`, cuyo valor será 56.
3. Creará la variable local `occupation`, cuyo valor será "Musician".
4. Para efectuar el print, leerá e imprimirá:
  1. **El valor de `name`:** Busca en el scope local, y encuentra el valor "Ian MacKaye" (deja de buscar).
  2. **El valor de `occupation`:** Busca en el scope local, y encuentra el valor "Musician" (deja de buscar).
  3. **El valor de `age`:** Busca en el scope local, y encuentra el valor 56 (deja de buscar).
  4. **El valor de `gender`:** Busca en el scope local, y no encuentra el objeto `gender`. "Sube" al scope `__main__`, y encuentra el valor "male" (deja de buscar).

## Precauciones con variables globales

Como se vio previamente, trabajar con variables globales puede resultar un poco confuso. Las variables globales son consideradas una mala práctica, ya que hacen muy fácil romper un programa por error.

### ¿Cómo se rompe un programa con variables globales?

Un programa puede tener miles de líneas de código e incorporar varias librerías (programas de terceros). En este aspecto, es sencillo que alguien llame por error a una variable global de su código de la misma forma que otra persona la llamó en su librería, y cualquier cambio en ella podría alterar el funcionamiento del código.

Existen muchas otras razones por las que estas no se recomiendan, pero que las dejaremos fuera de discusión por ahora.



## Cierre

En esta unidad aprendimos sobre:

- Ciclos.
- Funciones .

### Ciclos

Los ciclos nos ayudan a resolver problemas en base a iteraciones y son la base de la resolución de problemas.

- El código que se ejecuta dentro del ciclo corresponde a un "bloque".
- Las variables (locales) definidas dentro de los bloques no pueden ser accedidas desde fuera del bloque.
- Al utilizar while se debe tener cuidado de cambiar el índice para que haya un punto de salida del ciclo.

### Funciones

Las funciones permiten reutilizar código. Sobre éstas aprendimos:

- Pueden recibir parámetros.
- El retorno **debe ser explícito**.