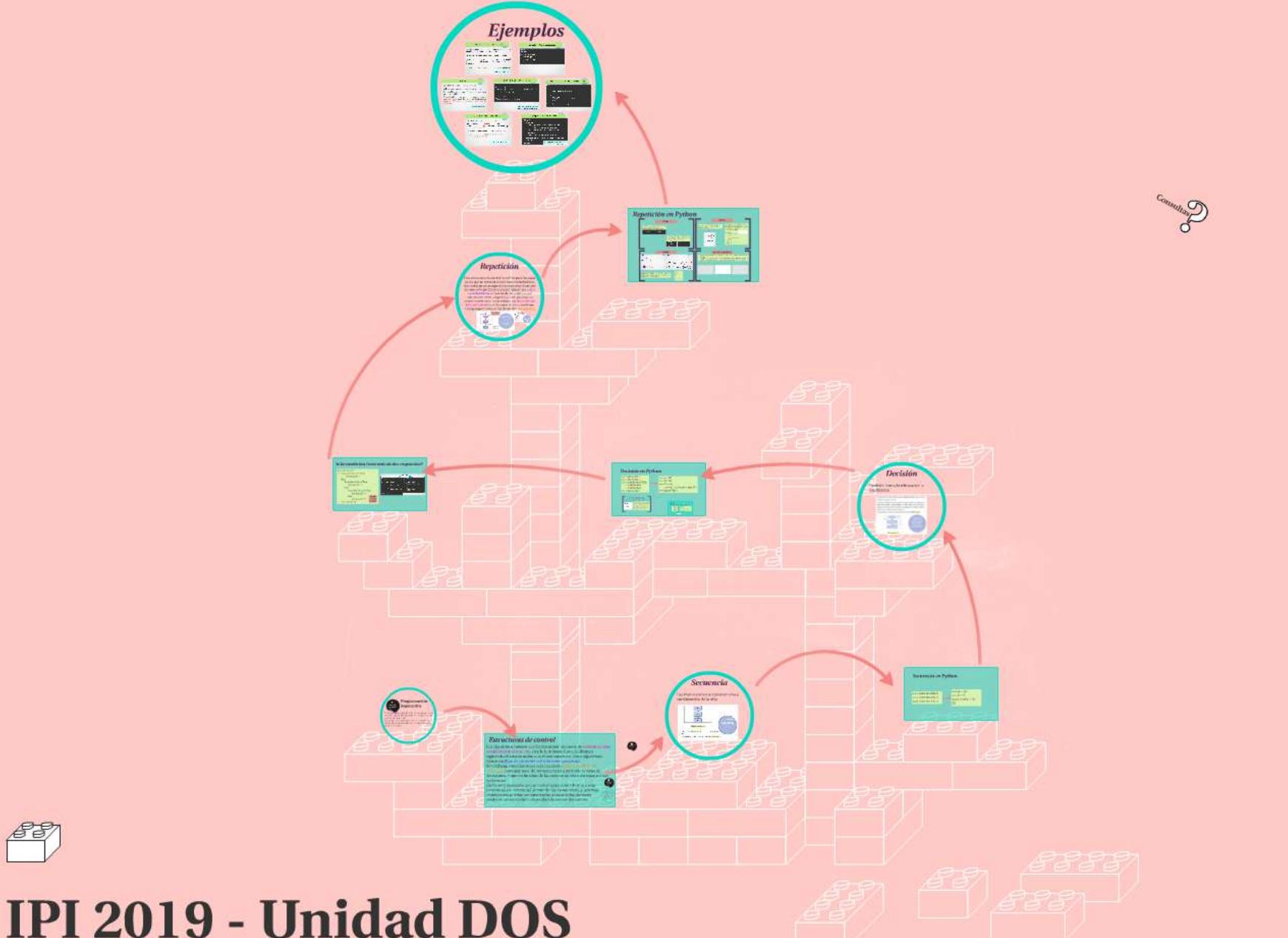


# IPI 2019 - Unidad DOS

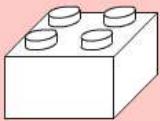
UNNOBA



# IPI 2019 - Unidad DOS

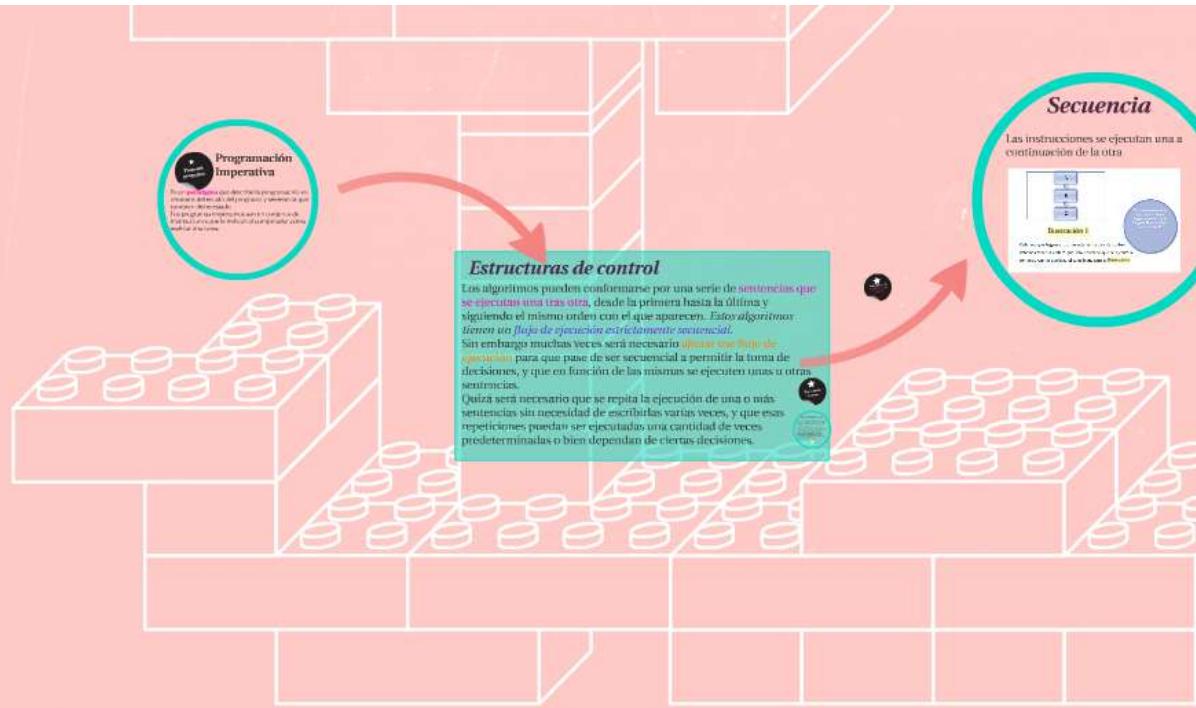
UNNOBA





# IPI 2019 - Unidad DOS

## UNNOBA





Flujo del  
programa

# Programación Imperativa

Es un **paradigma** que describe la programación en términos del estado del programa y sentencias que cambian dicho estado.

Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea.



Flujo del  
programa

P  
I

# Estructuras de control

Los algoritmos pueden conformarse por una serie de **sentencias que se ejecutan una tras otra**, desde la primera hasta la última y siguiendo el mismo orden con el que aparecen. *Estos algoritmos tienen un flujo de ejecución estrictamente secuencial.*

Sin embargo muchas veces será necesario **alterar ese flujo de ejecución** para que pase de ser secuencial a permitir la toma de decisiones, y que en función de las mismas se ejecuten unas u otras sentencias.

Quizá será necesario que se repita la ejecución de una o más sentencias sin necesidad de escribirlas varias veces, y que esas repeticiones puedan ser ejecutadas una cantidad de veces predeterminadas o bien dependan de ciertas decisiones.



Pero antes  
veamos

## Bloque de sentencias

Cuando llevas una sentencia por un código de sentencias que debes recorrer todo. Algunas condiciones ejecutivas que es útil tener en cuenta son las siguientes:  
1) Un ejemplo clásico es el comando de sentencias `while` de una estructura de control. Cada iteración dentro de los bucles de diferentes niveles tiene su condición. Siempre es útil recordar que el código dentro de la sentencia no tiene que ser



Pero antes  
veamos

# *Bloque de sentencias*

Cuando nos encontramos con un conjunto de sentencias que deben ejecutarse bajo idénticas condiciones decimos que es un "bloque de sentencias".

Por ejemplo cuando un conjunto de sentencias está "dentro" de una **estructura de control** hablamos de un bloque.

Cada lenguaje **delimita** los bloques de diferentes manera (begin-end, {}, etc.)

Para el caso de Python la **indentación** es la que estructura los bloques.



# En Python

```
1. nro1 = int(input('Ingrese el dividendo'))
2. nro2 = int(input('Ingrese divisor'))
3. if nro2 != 0:
    4.resultado = nro1/nro2
    5.print('El resultado de la división es', resultado)
6. else:
    7.print('No se puede realizar la operación porque el
    divisor es cero.')
```

En las líneas indicadas numeradas 4, 5 y 7 hay espacios desde el margen izquierdo hasta donde comienza la escritura. Esto es lo que hemos estado llamando “indentación” (o “sangría”, en español, ya que “indentación” es una adaptación del idioma inglés). La línea 4 tiene distinta indentación que la línea 7, pero esto es correcto porque pertenecen a distintos bloques: una al bloque “if” y otra al bloque “else”.

```
1. nro1 = int(input('Ingrese el dividendo'))
2. nro2 = int(input('Ingrese divisor'))
3. if nro2 != 0:
    4. resultado = nro1/nro2
    5. print('El resultado de la división es', resultado)
6. else:
    7. print('No se puede realizar la operación porque el divisor es cero.')
```

¿Qué sucede  
acá?

# *En Python*

```
1. nro1 = int(input('Ingrese el dividendo))  
2. nro2 = int(input('Ingrese divisor))  
3. if nro2 != 0:  
    4.resultado = nro1/nro2  
    5.print('El resultado de la división es', resultado)  
6. else:  
    7.print('No se puede realizar la operación porque el  
    divisor es cero.,)
```

En las líneas indicadas numeradas 4, 5 y 7 hay espacios desde el margen izquierdo hasta donde comienza la escritura. Esto es lo que hemos estado llamando “indentación” (o “sangría”, en español, ya que “indentación” es una adaptación del idioma inglés). La línea 4.tiene distinta indentación que la línea 7, pero esto es correcto porque pertenecen a distintos bloques: una al bloque “if” y otra al bloque “else”.

```
1. nro1 = int(input('Ingrese el dividendo'))  
2. nro2 = int(input('Ingrese divisor'))  
3. if nro2 != 0:  
    4.resultado = nro1/nro2
```

¿Qué sucede  
acá?

En las líneas indicadas numeradas 4, 5 y 7 hay espacios desde el margen izquierdo hasta donde comienza la escritura. Esto es lo que hemos estado llamando “indentación” (o “sangría”, en español, ya que “indentación” es una adaptación del idioma inglés). La línea 4 tiene distinta indentación que la línea 7, pero esto es correcto porque pertenecen a distintos bloques: una al bloque “if” y otra al bloque “else”.

```
1. nro1 = int(input('Ingrese el dividendo'))  
2. nro2 = int(input('Ingrese divisor'))  
3. if nro2 != 0:  
    4. resultado = nro1/nro2  
    5. print('El resultado de la división es', resultado)  
6. else:  
7.     print('No se puede realizar la operación porque el divisor es cero.')  
8.
```

¿Qué sucede  
acá?



Volvamos a las  
Estructuras de  
control

# Secuencia

Las instrucciones se ejecutan una a continuación de la otra

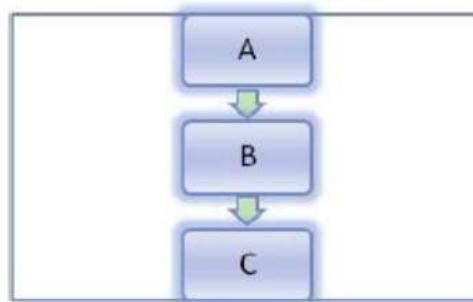
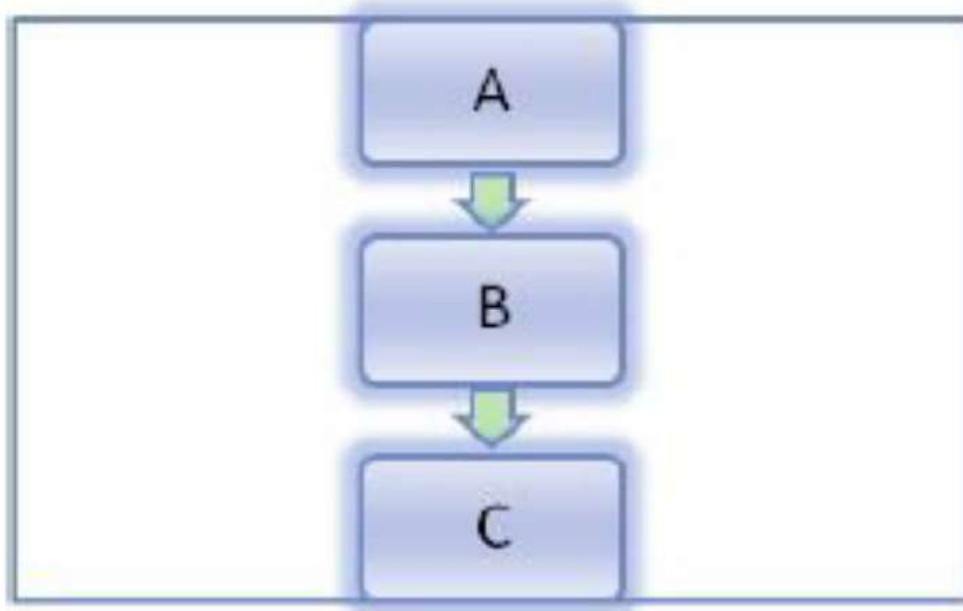


Ilustración 1

En esta secuencia se ejecuta en primer lugar la sentencia A, luego la B y por último la sentencia C.

Cada vez que hagamos uso de esta estructura de control debemos tener la certeza que será necesario que se ejecute la sentencia A antes que la B, tal como lo muestra la Ilustración 1.



## Ilustración 1

Cada vez que hagamos uso de esta estructura de control debemos tener la certeza que será necesario que se ejecute la sentencia A antes que la B, tal como lo muestra la Ilustración 1.

En esta secuencia se ejecuta en primer lugar la sentencia A, luego la B y por último la sentencia C.

# *Secuencia en Python*

```
>>> instrucción 1  
>>> instrucción 2  
>>> instrucción 3
```



```
>>> a = 10  
>>> b = 5  
>>> print(a + b)  
15
```

# Decisión

También llamada bifurcación o condicional

Esta estructura de control también llamada *condicional*, trabaja a partir de una condición y la respuesta a la misma.

En términos de programación de computadoras se llama condición a una pregunta que tendrá como resultado un valor verdadero o un valor falso. Tal es el caso de la pregunta ¿es cierto que llueve? La respuesta a esta pregunta será verdadera o falsa ya que llueve o no, no habrá más opciones que esas.

Podemos graficar una decisión o condicional como lo muestra la **Ilustración 2**

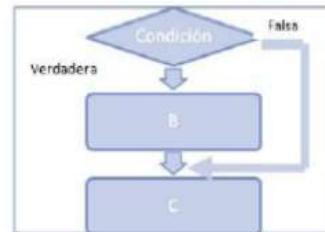


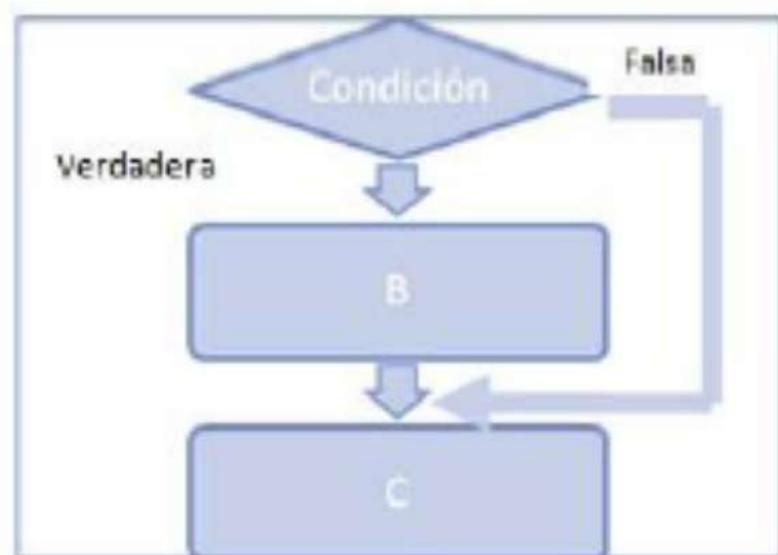
Ilustración 2

En esta estructura se trabaja con una condición. Si esta resulta verdadera se opta por ejecutar B y luego C, en caso contrario se opta por ejecutar directamente C.

Esta estructura de control también llamada *condicional*, trabaja a partir de una condición y la respuesta a la misma.

En términos de programación de computadoras se llama condición a una pregunta que tendrá como resultado un valor verdadero o un valor falso. Tal es el caso de la pregunta ¿es cierto que llueve? La respuesta a esta pregunta será verdadera o falsa ya que llueve o no, no habrá más opciones que esas.

Podemos graficar una decisión o condicional como lo muestra la **Ilustración 2**



En esta estructura se trabaja con una *condición*. Si esta resulta verdadera se opta por ejecutar B y luego C, en caso contrario se opta por ejecutar directamente C.

**Ilustración 2**

# Decisión en Python

```
>>> instrucción 1  
>>> instrucción 2  
>>> if condición es True:  
...   instrucción 3  
...   instrucción 4  
>>> instrucción 5
```



```
>>> a = 20  
>>> b = 10  
>>> if a > b:  
...     print("a es mayor que b")  
>>> print("fin")
```

## Decisión en pseudocódigo

Supongamos que necesitamos desarrollar un algoritmo para una persona que está a punto de salir de su casa y debe decidir si utiliza o no paraguas.

ALGORITMO:  
Tomar paraguas  
Abrir la puerta.  
Sali de la casa  
SI llueve  
 Abrir paraguas  
 Colocar el paraguas sobre la cabeza  
 Cerrar la puerta  
 Caminar hacia el destino

Podemos observar que sólo se abrirá el paraguas y se colocará sobre la cabeza, **si** se cumple que **llueve**. Luego cerrará la puerta y caminará hacia el destino. **En caso que no llueva**, entonces simplemente cerrará la puerta para luego caminar hacia el destino. La condición para este caso es "llueve", si redactamos a la condición como una pregunta entonces sería **llueve?** O ¿es cierto que **llueve?**, para ambas preguntas la respuesta será "**sí**" o "**no**", o bien "**verdadero**" o "**falso**".

## If...else

En ocasiones será necesario ejecutar un bloque cuando la condición resulta verdadera y otro para cuando la condición resulta falsa.

```
>>> instrucción 1  
>>> instrucción 2  
>>> if condición es True:  
...     instrucción 3  
...     instrucción 4  
else:  
...     instrucción 5  
...     instrucción 6  
>>> instrucción 7
```

```
>>> a = 20  
>>> b = 10  
>>> if a > b:  
...     print("a es mayor que b")  
else:  
...     print("a no es mayor que b")  
>>> print("fin")
```



# *Decisión en pseudocódigo*

Supongamos que necesitamos desarrollar un algoritmo para una persona que está a punto de salir de su casa y debe decidir si utiliza o no paraguas.

## ALGORITMO:

```
Tomar paraguas  
Abrir la puerta  
Salir de la casa  
Si llueve  
    Abrir paraguas  
    Colocar el paraguas sobre la cabeza  
Cerrar la puerta  
Caminar hacia el destino
```

Podemos observar que sólo se abrirá el paraguas y se colocará sobre la cabeza, **si** se cumple que **llueve**. Luego cerrará la puerta y caminará hacia el destino. **En caso que no llueva**, entonces simplemente cerrará la puerta para luego caminar hacia el destino. La condición para este caso es “llueve”, si redactamos a la condición como una pregunta entonces sería ¿llueve? O ¿es cierto que llueve?, para ambas preguntas la respuesta será “si” o “no”, o bien “verdadero” o “falso”.

## ALGORITMO:

Tomar paraguas

Abrir la puerta

Salir de la casa

Si llueve

    Abrir paraguas

    Colocar el paraguas sobre la cabeza

Cerrar la puerta

Caminar hacia el destino

uamos desarrollar un algoritmo para una persona que u casa y debe decidir si utiliza o no paraguas.

Podemos observar que sólo se abrirá el paraguas y se colocará sobre la cabeza, **si** se cumple que **llueve**. Luego cerrará la puerta y caminará hacia el destino. **En caso que no llueva**, entonces simplemente cerrará la puerta para luego caminar hacia el destino. La condición para este caso es “llueve”, si redactamos a la condición como una pregunta entonces sería **¿llueve?** O **¿es cierto que llueve?**, para ambas preguntas la respuesta será “**si**” o “**no**”, o bien “**verdadero**” o “**falso**”.

# *Decisión en pseudocódigo*

Supongamos que necesitamos desarrollar un algoritmo para una persona que está a punto de salir de su casa y debe decidir si utiliza o no paraguas.

## ALGORITMO:

Tomar paraguas

Abrir la puerta

Salir de la casa

Si llueve

    Abrir paraguas

    Colocar el paraguas sobre la cabeza

Cerrar la puerta

Caminar hacia el destino

Podemos observar que sólo se abrirá el paraguas y se colocará sobre la cabeza, **si** se cumple que **llueve**. Luego cerrará la puerta y caminará hacia el destino. **En caso que no llueva**, entonces simplemente cerrará la puerta para luego caminar hacia el destino. La condición para este caso es “llueve”, si redactamos a la condición como una pregunta entonces sería ¿llueve? O ¿es cierto que llueve?, para ambas preguntas la respuesta será “si” o “no”, o bien “verdadero” o “falso”.

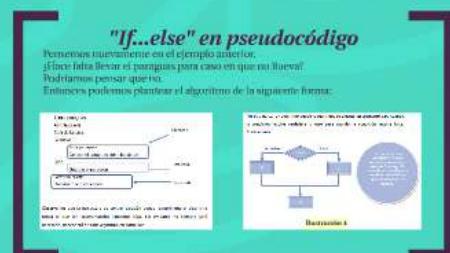
# If...else

En ocasiones será necesario ejecutar un bloque cuando la condición resulta verdadera y otro para cuando la condición resulta falsa.

```
>>> instrucción 1  
>>> instrucción 2  
>>> if condición es True:  
    ... instrucción 3  
    ... instrucción 4  
else:  
    ... instrucción 5  
    ... instrucción 6  
>>> instrucción 7
```



```
>>> a = 20  
>>> b = 10  
>>> if a > b:  
    ... print("a es mayor que b")  
else:  
    ... print("a no es mayor que b")  
>>> print("fin")
```



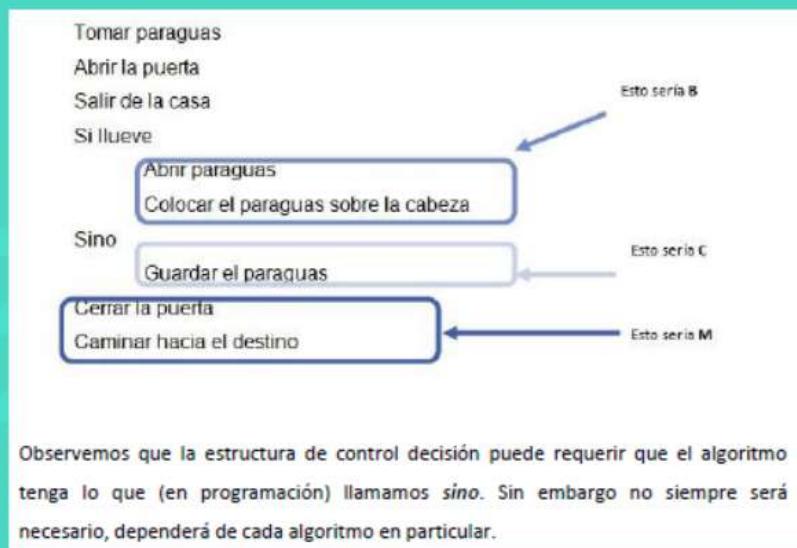
# "If...else" en pseudocódigo

Pensemos nuevamente en el ejemplo anterior.

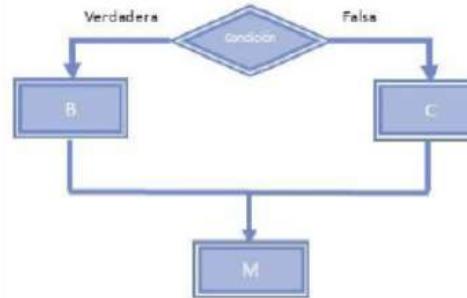
¿Hace falta llevar el paraguas para caso en que no llueva?

Podríamos pensar que no.

Entonces podemos plantear el algoritmo de la siguiente forma:



Por otra parte, un algoritmo puede requerir que se ejecute un segmento para cuando la condición resulte verdadera u otro para cuando la condición resulte falsa.  
Gráficamente:



En este caso si la condición resulta verdadera se opta por ejecutar B y luego M, para el caso en el que la condición sea falsa se opta por ejecutar C y luego M.

Ilustración 4

Tomar paraguas

Abrir la puerta

Salir de la casa

Si llueve

    Abrir paraguas

    Colocar el paraguas sobre la cabeza

Sino

    Guardar el paraguas

Cerrar la puerta

Caminar hacia el destino

Esto sería B

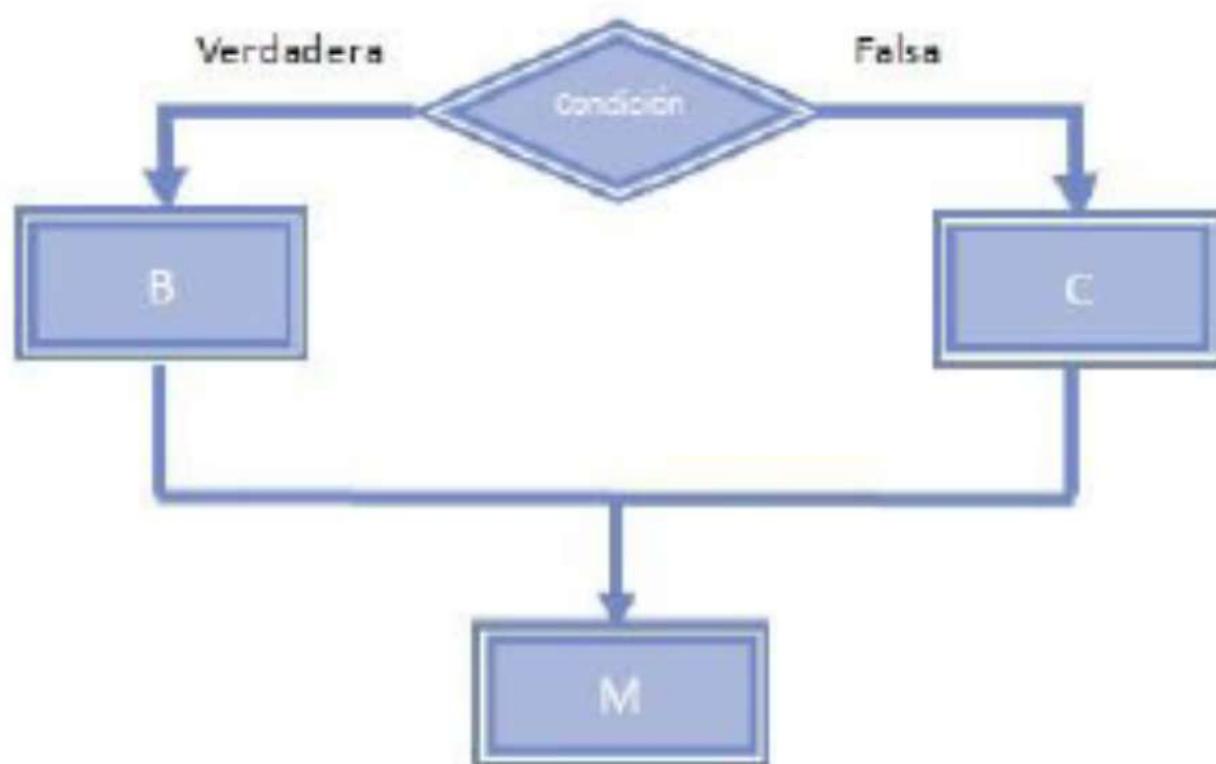
Esto sería C

Esto sería M

Observemos que la estructura de control decisión puede requerir que el algoritmo tenga lo que (en programación) llamamos *sino*. Sin embargo no siempre será necesario, dependerá de cada algoritmo en particular.

Por otra parte, un algoritmo puede requerir que se ejecute un segmento para cuando la condición resulte verdadera u otro para cuando la condición resulte falsa.

Gráficamente:



En este caso si la condición resulta verdadera se opta por ejecutar B y luego M, para el caso en el que la condición sea falsa se opta por ejecutar C y luego M.

Ilustración 4

# "If...else" en pseudocódigo

Pensemos nuevamente en el ejemplo anterior.

¿Hace falta llevar el paraguas para caso en que no llueva?

Podríamos pensar que no.

Entonces podemos plantear el algoritmo de la siguiente forma:

Tomar paraguas

Abrir la puerta

Salir de la casa

Si llueve

    Abrir paraguas

    Colocar el paraguas sobre la cabeza

    Esto sería B

Sino

    Guardar el paraguas

    Esto sería C

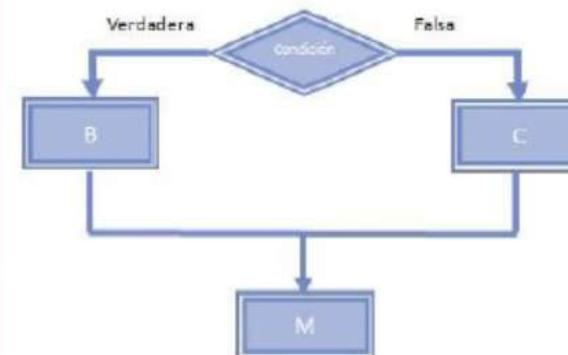
Cerrar la puerta

Caminar hacia el destino

    Esto sería M

Observemos que la estructura de control decisión puede requerir que el algoritmo tenga lo que (en programación) llamamos *sino*. Sin embargo no siempre será necesario, dependerá de cada algoritmo en particular.

Por otra parte, un algoritmo puede requerir que se ejecute un segmento para cuando la condición resulte verdadera u otro para cuando la condición resulte falsa. Gráficamente:



En este caso si la condición resulta verdadera se opta por ejecutar B y luego M, para el caso en el que la condición sea falsa se opta por ejecutar C y luego M.

Ilustración 4

# *Si la condición tiene más de dos respuestas?*

```
>>> instrucción 1  
>>> if condición 1 es True:  
    instrucción 2  
else:  
    if condición 2 es True:  
        instrucción 3  
    else:  
        if condición 3 es True:  
            instrucción 4  
        else:  
            instrucción 5  
>>> instrucción 6
```

puede  
resultar  
ilegible

```
>>> a = 10  
>>> b = 20  
>>> if a == 1:  
...     print("a igual a 1")  
... else:  
...     if a == 2:  
...         print("a igual a 2")  
...     else:  
...         if a == 3:  
...             print("a igual a 3")  
...         else:  
...             print("a mayor a 3")  
>>> print("fin")
```

```
>>> a = 10  
>>> b = 20  
>>> if a == 1:  
...     print("a igual a 1")  
... elif a == 2:  
...     print("a igual a 2")  
... elif a == 3:  
...     print("a igual a 3")  
... else:  
...     print("a mayor a 3")  
>>> print("fin")
```

```
>>> instrucción 1  
>>> if condición 1 es True:  
    instrucción 2  
else:  
    if condición 2 es True:  
        instrucción 3  
    else:  
        if condición 3 es True:  
            instrucción 4  
        else:  
            instrucción 5  
>>> instrucción 6
```

**puede  
resultar  
ilegible**

```
>>> a = 1  
>>> b = 2  
>>> if a < b:  
...     print('a is less than b')  
... else:  
...     print('a is greater than or equal to b')  
... if a < b:  
...     print('a is less than b')  
... else:  
...     print('a is greater than or equal to b')  
>>> print('done')
```

```
>>> a = 10
>>> b = 20
>>> if a == 1:
...     print("a igual a 1")
... else:
...     if a == 2:
...         print("a igual a 2")
...     else:
...         if a == 3:
...             print("a igual a 3")
...         else:
...             print("a mayor a 3")
>>> print("fin")
```

```
>>> a = 10
>>> b = 20
>>> if a == 1:
...     print("a igual a 1")
... elif a == 2:
...     print("a igual a 2")
... elif a == 3:
...     print("a igual a 3")
... else:
...     print("a mayor a 3")
>>> print("fin")
```

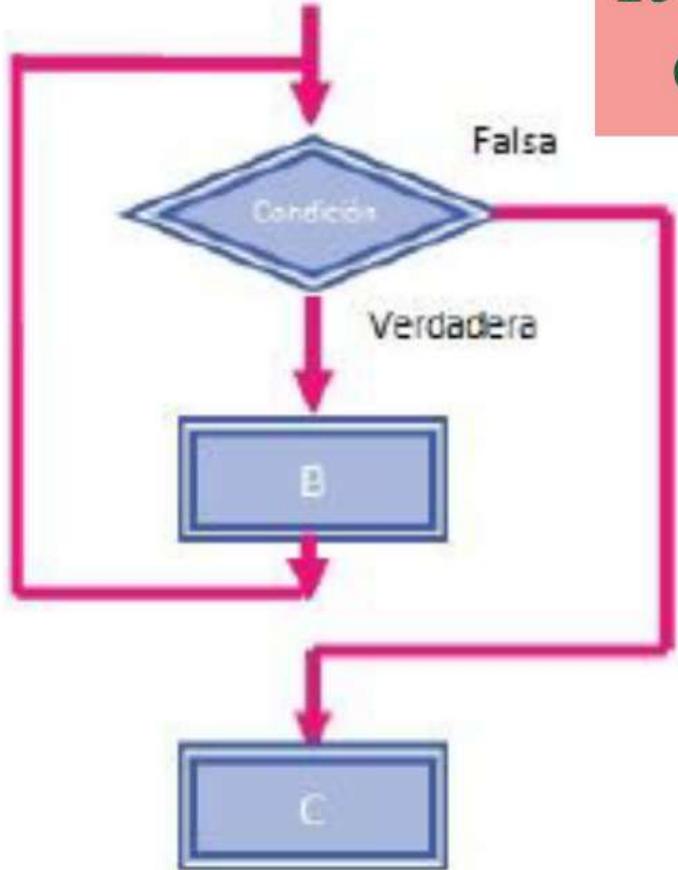
# Repetición

Esta estructura de control se utiliza para los casos en los que se necesita repetir una o más órdenes. Sin embargo, estas repeticiones pueden darse por dos motivos: podríamos querer repetir una **cierta cantidad de veces** (sabiendo de antemano el número de veces a repetir), o bien podríamos querer repetir una o más sentencias **dependiendo del resultado** de la evaluación de una condición. A estas repeticiones se las denomina **iteraciones**.



# Repeticiones se las denominan

## Iteración con condición



Acá se puede trabajar con una **condición**. Mientras que la condición resulte verdadera se opta por ejecutar B, y luego vuelve a preguntarse por la condición.

Cuando la condición resulte falsa se continuará con la ejecución de C.

Veamos un ejemplo intentando escribir un algoritmo que permita beber una sopa que, en principio está caliente, por lo que se debe enfriar hasta que pueda beberse:

INICIO  
Mientras la sopa esté caliente hacer:  
    Sacar la sopa del fuego  
    Dejarla enfriar  
Hacer el punto 2 mientras la sopa esté caliente.  
Al finalizar el bucle se procederá a beber la sopa.  
Algoritmo:  
    Mientras la sopa esté caliente hacer:  
        Sacar la sopa del fuego  
        Dejarla enfriar  
    Hacer el punto 2 mientras la sopa esté caliente.  
Al finalizar el bucle se procederá a beber la sopa.

**Veamos un ejemplo intentando escribir un algoritmo que permita beber una sopa que, en principio está caliente, por lo que se debe enfriar hasta que pueda beberse:**

**ALGORITMO:**

Mientras la sopa esté caliente

Soplar

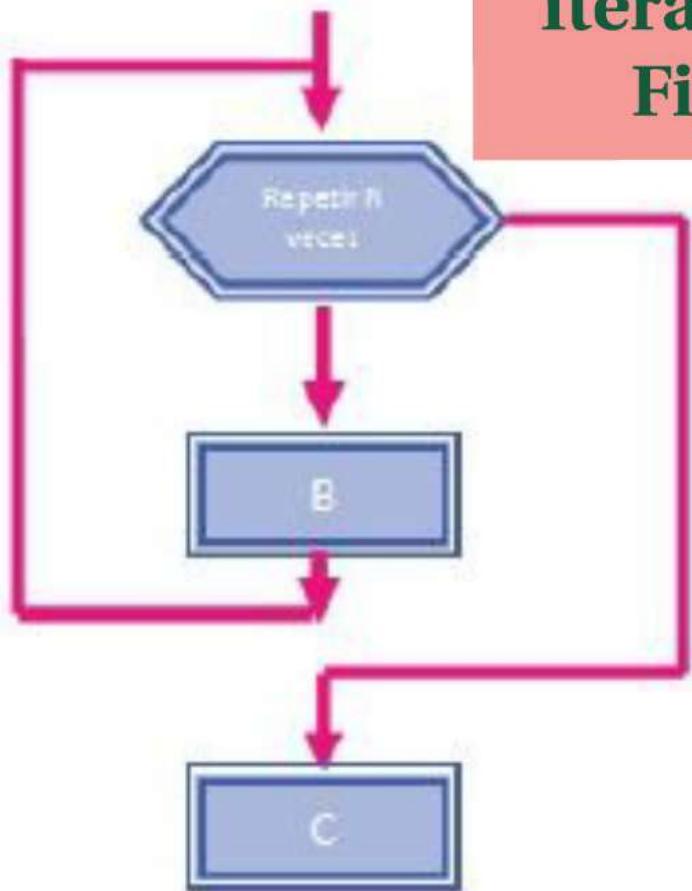
Beber la sopa

En este ejemplo como no sabemos de antemano cuántas veces vamos a *Soplar* antes de poder *Beber la sopa*, necesitamos iterar con una condición.

La condición es *la sopa está caliente*, que si la planteamos como una pregunta la respuesta será: si o no. Mientras que la respuesta sea si, el algoritmo indicará *Soplar* para luego volver a preguntar si *la sopa está caliente*. En algún momento la respuesta será no y entonces recién allí el algoritmo continuará con la ejecución de la siguiente sentencia (para este caso *Beber la sopa*).

# Iteración

## Iteración Fija



Vayamos a un cotidiano ejemplo intentando escribir un algoritmo que permita tomar una caja que se encuentra al final de una escalera de 8 escalones.

ALGORITMO:  
Acercarse a la escalera  
Repetir 8 veces  
    Subir un escalón  
    Tomar la caja  
    Girar 180 grados  
    Repetir 8 veces  
        Bajar un escalón

Se trabaja con una **cantidad fija de peticiones**. Para este ejemplo B se ejecutará N veces, y una vez que eso haya sucedido se continuará con la ejecución de C. Para estos casos será importante verificar que realmente B podrá ejecutarse N veces, a fin de evitar errores.

Vayamos a un cotidiano ejemplo intentando escribir un algoritmo que permita tomar una caja que se encuentra al final de una escalera de 8 escalones.

**ALGORITMO:**

Acercarse a la escalera

Repetir 8 veces

    Subir un escalón

    Tomar la caja

    Girar 180 grados

    Repetir 8 veces

        Bajar un escalón

# Repetición en Python

## While

Se evalúa una condición de tipo booleana y mientras ésta sea verdadera, se ejecuta el bloque asociado al «while».

```
>>> instrucción 1  
>>> while condición es True:  
...   instrucción 2  
...   instrucción 3  
...   instrucción n  
>>> instrucción n
```

Se evalúa una condición booleana y, mientras ésta sea verdadera, se ejecuta el bloque asociado al «while». Cuando la condición se hace falsa, se ejecuta el bloque asociado al «else».

```
>>> instrucción 1  
>>> while condición es True:  
...   instrucción 2  
...   instrucción 3  
...   else:  
...     instrucción 4  
>>> instrucción 5
```

## Range

`range()`:

Genera progresiones numéricas.

`range([start,] stop[, step])`

Genera números enteros desde `start` hasta (`stop`-1), con un paso `step`.

Todos los argumentos deben ser números enteros.

`start` y `step` son opcionales, haciendo que `range()` admita tres variaciones:

- `range(stop)`
- `range(start, stop)`
- `range(start, stop, step)`

El valor por defecto para `start` es 0 y para `step` es 1.

Al igual que el «while», el «for» puede contener un bloque «else», el cuál se ejecutará cuando finalice la iteración sobre la secuencia.

`range()`

Ejemplos:

```
>>> range(5)          start=0, stop=5, step=1
```

Genera enteros entre 0 y 9 (sin incluir al 9):

0, 1, 2

```
>>> range(2, 10)      start=2, stop=10, step=1
```

Genera enteros entre 2 y 10 (sin incluir al 10):

2, 3, 4, 5, 6, 7, 8, 9

```
>>> range(30, 50, 5)    start=10, stop=50, step=10
```

Genera enteros entre 10 y 50 (sin incluir al 50), saltando de a 5:

10, 15, 20, 25, 30, 35, 40, 45

```
for a in lista:  
    print(a)  
else:  
    print("se terminaron los  
    elementos de 'a'")
```

Se terminaron los  
elementos de 'a'

## for/in

Itera sobre los **elementos** de cualquier **secuencia**, en el **orden** determinado por la misma secuencia.



La estructura "for/in" se puede usar para iterar sobre los valores numéricos generados por "range", sobre los caracteres de una **cadena de caracteres** y sobre los elementos de una tupla.

Por ejemplo:

```
>>> for x in range(10)  
...   print(x)
```

```
>>> for x in 'Una cadena':  
...   print(x)
```

```
>>> for x in (1, 2, 3):  
...   print(x)
```

## Break y Continue

- "break" detiene la ejecución de la iteración y "salta" a ejecutar la primera sentencia fuera del bloque.
- "continue" detiene la ejecución, no sale de la iteración, y va a evaluar nuevamente la condición de la estructura de control en la que está (ya sea un "for" o un "while").

Resumen  
break:  
Permite romper inmediatamente el resto de ciclo de bucle asociado al mismo comando de control.

Resumen  
continue:  
Permite a la ejecución saltar las siguientes instrucciones de bucle, sin romper todo el operador de bucle ni salir de él.

Resumen  
else:  
Al igual que en los bucles condicionales, el bucle for tiene la opción de tener un bloque else que se ejecutará una vez que finalice el bucle. Es útil para ejecutar ciertas acciones al finalizar el bucle.

# Repetición en Python

## While

Se evalúa una condición de tipo booleana y mientras ésta sea verdadera, se ejecuta el bloque asociado al «while»

```
>>> instrucción 1  
>>> while condición es True:  
...     instrucción 2  
...     instrucción 3  
>>> instrucción 4
```

```
>>> a = 5  
>>> while a > 1:  
...     print('a = ', a)  
...     a -= 1  
>>> print('fin')
```

Se evalúa una condición booleana y, mientras ésta sea verdadera, se ejecuta el bloque asociado al «while». Cuando la condición se hace falsa, se ejecuta el bloque asociado al «else».

```
>>> instrucción 1  
>>> while condición es True:  
...     instrucción 2  
...     instrucción 3  
... else:  
...     instrucción 4  
>>> instrucción 5
```

```
>>> a = 5  
>>> while a > 1:  
...     print('a = ', a)  
...     a -= 1  
... else:  
...     print('a no es mayor que 1')  
>>> print('fin')
```

## Range

range():

range()

Se evalúa una condición de tipo booleana y mientras ésta sea verdadera, se ejecuta el bloque asociado al «while»

```
>>> instrucción 1  
>>> while condición es True:  
...     instrucción 2  
...     instrucción 3  
>>> instrucción 4
```

```
>>> a = 5  
>>> while a > 1:  
...     print('a = ', a)  
...     a -= 1  
>>> print('fin')
```

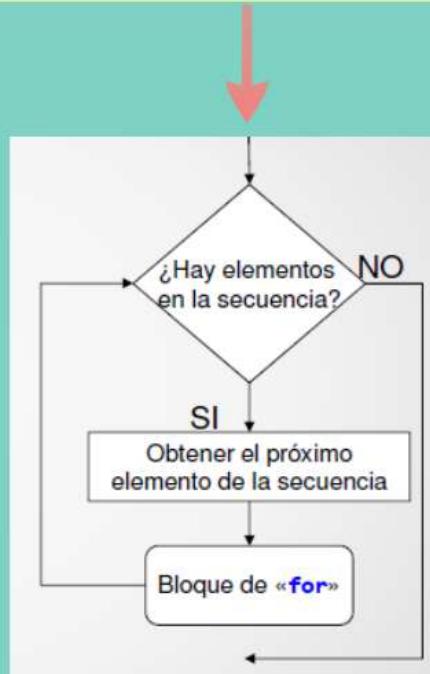
Se evalúa una condición booleana y, mientras ésta sea verdadera, se ejecuta el bloque asociado al «while». Cuando la condición se hace falsa, se ejecuta el bloque asociado al «else».

```
>>> instrucción 1  
>>> while condición es True:  
...     instrucción 2  
...     instrucción 3  
... else:  
...     instrucción 4  
>>> instrucción 5
```

```
>>> a = 5  
>>> while a > 1:  
...     print('a =', a)  
...     a -= 1  
... else:  
...     print('a no es mayor que 1')  
>>> print('fin')
```

# for/in

Itera sobre los **elementos** de cualquier **secuencia**, en el **orden** determinado por la misma secuencia.



La estructura “for/in” se puede usar para iterar sobre los valores numéricos generados por “range”, sobre los caracteres de una **cadena de caracteres** y sobre los elementos de una **tupla**.

Por ejemplo:

```
>>> for x in range(10)
... print(x)
```

```
>>> for x in 'Una cadena':
... print(x)
```

```
>>> for x in (1, 2, 3):
... print(x)
```

Cadena de caracteres

```
>>> cadena = "Un"
>>> for caracter in cadena:
...     print(caracter)
...
U
n
>>>
```

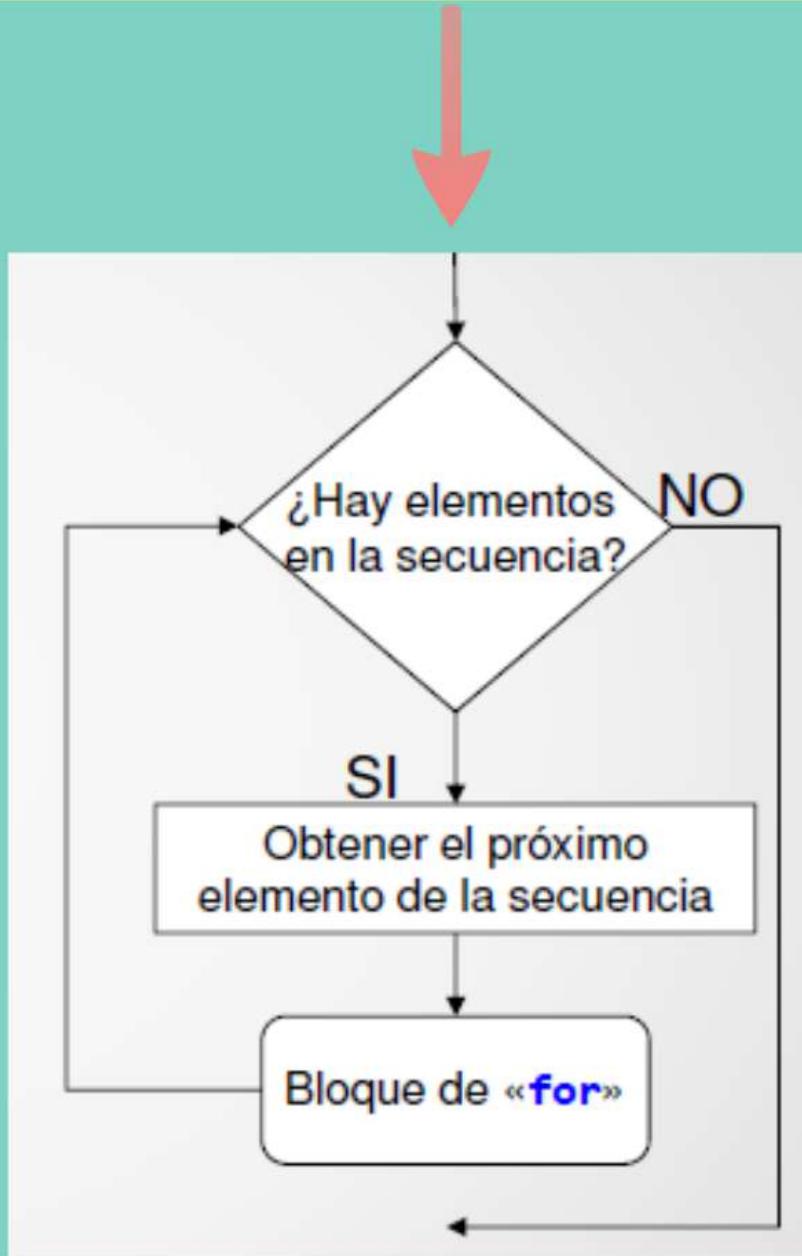
Rango numérico

```
>>> for x in range(5):
...     print(x)
...
0
1
2
3
4
>>>
```

# Break y Continue

“break” detiene la ejecución de la iteración y “salto” o saltar la primera sentencia.

Itera sobre los **elementos** de cualquier **secuencia**, en el **orden** determinado por la misma secuencia.



or la

La estructura “for/in” se puede usar para iterar sobre los valores numéricos generados por “**range**”, sobre los caracteres de una **cadena de caracteres** y sobre los elementos de una **tupla**.

Por ejemplo:

```
>>> for x in range(10)
... print(x)
```

```
>>> for x in 'Una cadena':
... print(x)
```

```
>>> for x in (1, 2, 3):
... print(x)
```

Cadena de caracteres	Rango numérico
<pre>&gt;&gt;&gt; cadena = 'holo' &gt;&gt;&gt; for carácter in cadena: ...     print(carácter) h o l o &gt;&gt;&gt;</pre>	<pre>&gt;&gt;&gt; for x in range(5): ...     print(x) 0 1 2 3 4 &gt;&gt;&gt;</pre>

# TIPOS Y SÓLO LOS

## Cadena de caracteres

```
>>> cadena = 'hola'  
>>> for carácter in cadena:  
...     print(carácter)  
h  
o  
l  
a  
>>>
```

## Rango numérico

```
>>> for x in range(5):  
...     print(x)  
0  
1  
2  
3  
4  
>>>
```

```
... else:  
...     instrucción 4  
>>> instrucción 5
```

```
... else:  
...     print('a no es mayor que 1')  
>>> print('fin')
```

# Range

## range():

Genera progresiones numéricas.

```
range([start,] stop[, step])
```

Genera números enteros desde *start* hasta (*stop*-1), con un paso *step*.

Todos los argumentos deben ser **números enteros**.

*start* y *step* son opcionales, haciendo que `range()` admita tres variaciones:

- `range(stop)`
- `range(start, stop)`
- `range(start, stop, step)`

El valor por defecto para *start* es **0** y para *step* es **1**.

## range()

Ejemplos:

```
>>> range(3)
```

```
start=0, stop=3, step=1
```

Genera enteros entre 0 y 3 (sin incluirlo al 3):

```
0, 1, 2
```

```
>>> range(2, 10)
```

```
start=2, stop=10, step=1
```

Genera enteros entre 2 y 10 (sin incluirlo al 10):

```
2, 3, 4, 5, 6, 7, 8, 9
```

```
>>> range(10, 50, 5)
```

```
start=10, stop=50, step=10
```

Genera enteros entre 10 y 50 (sin incluir al 50), saltando de a 5:

```
10, 15, 20, 25, 30, 35, 40, 45
```

Al igual que el «while», el «for» puede contener un bloque «else», el cuál se ejecutará cuando finalice la iteración sobre la secuencia.

```
>>> a = 'holá'  
>>> for x in a:  
...     print(x)  
... else:  
...     print("Se terminaron los  
...         elementos de 'a'")  
...  
h  
o  
l  
á  
Se terminaron los elementos de  
'a'  
>>>
```

## range():

Genera progresiones numéricas.

`range([start,] stop[, step])`

Genera números enteros desde `start` hasta `(stop-1)`, con un paso `step`.

Todos los **argumentos** deben ser **números enteros**.

`start` y `step` son opcionales, haciendo que `range()` admita tres variaciones

- `range(stop)`
- `range(start, stop)`
- `range(start, stop, step)`

El valor por defecto para `start` es `0` y para `step` es `1`.

## range()

Ejemplos:

`>>> range(3)` start=0, stop=3, step=1

Genera enteros entre 0 y 3 (sin incluirlo al 3):

0, 1, 2

`>>> range(2, 10)` start=2, stop=10, step=1

Genera enteros entre 2 y 10 (sin incluirlo al 10):

2, 3, 4, 5, 6, 7, 8, 9

`>>> range(10, 50, 5)` start=10, stop=50, step=10

Genera enteros entre 10 y 50 (sin incluir al 50), saltando de a 5:

10, 15, 20, 25, 30, 35, 40, 45

```
>>> a = 'hola'  
>>> for x in a:
```

Todos los argumentos deben ser números enteros.

```
>>> range(2, 10)
```

*start* y *step* son opcionales, haciendo que `range()` admita tres variaciones:

- `range(stop)`
- `range(start, stop)`
- `range(start, stop, step)`

```
>>> range(10, 50)
```

Genera enteros entre 10 y 50.

El valor por defecto para *start* es 0 y para *step* es 1.

Al igual que el «while», el «for» puede contener un bloque «else», el cuál se ejecutará cuando finalice la iteración sobre la secuencia.

```
>>> a = 'hola'  
>>> for x in a:  
... print(x)  
... else:  
... print("Se terminaron los  
elementos de 'a'")
```

```
...
```

```
h
```

```
o
```

```
l
```

```
a
```

Se terminaron los elementos de  
'a'

```
>>>
```

# Break y Continue

- “break” detiene la ejecución de la iteración y “salta” a ejecutar la primera sentencia fuera del bloque.
- “continue” detiene la ejecución, no sale de la iteración, y va a evaluar nuevamente la condición de la estructura de control en la que está (ya sea un “for” o un “while”).

## Iteración

**break:**

Permite romper incondicionalmente el lazo del while (el bloque asociado al «else» nunca se ejecuta).

```
>>> instrucción 1
>>> while condición 1 es True:
...     instrucción 2
...     if condición 2 es True:
...         break
...     else:
...         instrucción 3
>>> instrucción 4
```

## Iteración

**continue:**

Permite saltar a evaluar nuevamente la condición, sin terminar con la ejecución del bloque asociado al «while».

```
>>> a = 8
>>> while a > 1:
...     a -= 1
...     if a != 5:
...         continue
...     print(a)
...     else:
...         print('a es menor que 1')
>>> print('fin')
```

Cual es el resultado?

De los valores calculados, imprime el número 5 solamente.

## Iteración

**for ... in ... / else:**

Al igual que el «while», el bloque asociado al «for» puede contener:

- **break**, que finaliza la ejecución del **for** aunque no se hayan acabado los elementos de la secuencia.
- **continue**, que omite las instrucciones hasta finalizar el bloque y vuelve al inicio del mismo, continuando con el siguiente elemento de la secuencia.

Cual es el resultado?

Mostrar los impares entre 1 y 10, pero parar si aparece un múltiplo de 5.

```
>>> for x in range(1, 10):
...     if x % 5 == 0:
...         break
...     if x % 2 == 0:
...         continue
...     print(x)
1
3
```

## Iteración

### break:

Permite romper incondicionalmente el lazo del while (el bloque asociado al «else» nunca se ejecuta).

```
>>> instrucción 1
>>> while condición 1 es True:
...     instrucción 2
...     if condición 2 es True:
...         break
... else:
...     instrucción 3
>>> instrucción 4
```

```
>>> a = int(input())
>>> while a > 1:
...     print('a =', a)
...     a -= 2
...     if (a == 5):
...         break
... else:
...     print('a es menor o igual
que 1')
>>> print('fin')
```

## Iteración

### continue:

Permite saltar a evaluar nuevamente la condición, sin terminar con la ejecución del bloque asociado al «**while**».

```
>>> a = 8
>>> while a > 1:
...     a -= 1
...     if a != 5:
...         continue
...     print(a)
... else:
...     print('a es menor que 1')
>>> print('fin')
```

Cual es el resultado?

De los valores calculados, imprime el número 5 solamente.

## Iteración

### for ... in ... / else:

Al igual que el «`while`», el bloque asociado al «`for`» puede contener:

- `break`, que finaliza la ejecución del `for` aunque no se hayan acabado los elementos de la secuencia.
- `continue`, que omite las instrucciones hasta finalizar el bloque y vuelve al inicio del mismo, continuando con el siguiente elemento de la secuencia.

Cual es el resultado?

Mostrar los impares entre 1 y 10, pero parar si aparece un múltiplo de 5.

```
>>> for x in range(1, 10):
...     if x % 5 == 0:
...         break
...     if x % 2 == 0:
...         continue
...     print(x)
1
3
>>>
```

# Ejemplos

## Ejemplo 1: Sucesión de Fibonacci

La siguiente sucesión infinita de números naturales se conoce como la sucesión de Fibonacci:

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987...

La sucesión comienza con los números 0 y 1, y a partir de estos, cada elemento es la suma de los dos anteriores.

Calcular todos los números de la secuencia menores a 1000.

¿Cómo puede resolverse?

## Ejemplo 1: Solución propuesta

```
actual = 0
próximo = 1

while actual < 1000:
    print(actual)
    suma = actual + próximo
    actual = próximo
    próximo = suma
```

## Ejemplo 2

- Se solicita el ingreso de **números enteros**.
- La lectura de información finaliza al **leer un cero**.
- Se pide al final del proceso informar la suma de todos los números leídos.
- Si durante el ingreso de los números se detecta que la **suma acumulada** luego del ingreso del último número **superá a 1000**, se termina el proceso informado del problema.

¿Soluciones sugeridas?

## Ejemplo 2: Solución Propuesta

```
suma = 0

while suma < 1000:
    num = int(input("Ingrese un número (0 para terminar): "))
    if num == 0:
        print("Total:", suma)
        break
    else:
        suma += num
        print("Límite de suma excedido!")
```

Analicemos ventajas y desventajas de la solución propuesta. ¿Qué opinan?

## Ejemplo 2: Solución Propuesta en la Clase

```
total = 0
n = 1

while n != 0 and total < 1000:
    s = input("Ingrese un número entero: ")
    n = int(s)

    total += n
    if total > 1000:
        print("La suma superó a los 1000")
        break

if n == 0:
    print("La suma acumulada es", total)
```

## Ejemplo 3: Sumar números pares

- Se solicita el ingreso de **números pares**.
- Si se ingresa un **número impar**, se informa del problema y no se lo procesa, pero no se interrumpe el ingreso de números.
- La lectura de información finaliza al leer un cero.
- Se pide al final del proceso informar la **suma de todos los números pares** leídos.

¿Soluciones sugeridas?

## Ejemplo 3: Solución propuesta

```
total = 0
while True:
    num = int(input("Ingrese un número par: "))
    if num % 2 == 0:
        break # Ingresó un número par...
    print("ERROR: No ingresó un número par.")

    if num == 0:
        break # Si se ingresa un cero, para.

    # Mientras tanto, acumula los números ingresados
    total += num
print(total)
```

Analicemos ventajas y desventajas de la solución propuesta. ¿Qué opinan?



## Ejemplo 1: Sucesión de Fibonacci

La siguiente sucesión infinita de números naturales se conoce como la sucesión de Fibonacci:

**0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987...**

La sucesión comienza con los números 0 y 1, y a partir de estos, cada elemento es la suma de los dos anteriores.

Calcular todos los números de la secuencia menores a 1000.

¿Cómo puede resolverse?

## Ejemplo 1: Solución propuesta

```
actual = 0
próximo = 1

while actual < 1000:
    print(actual)
    suma = actual + próximo
    actual = próximo
    próximo = suma
```

## Ejemplo 2

- Se solicita el ingreso de **números enteros**.
- La lectura de información finaliza **al leer un cero**.
- Se pide al final del proceso **informar la suma de todos los números leídos**.
- Si durante el ingreso de los números se detecta que **la suma acumulada** luego del ingreso del último número supera a 1000, se **termina el proceso informado del problema**.

¿Soluciones sugeridas?

## Ejemplo 2: Solución Propuesta

```
suma = 0

while suma < 1000:
    num = int(input("Ingrese un número (0 para terminar): "))
    if num == 0:
        print("Total:", suma)
        break

    suma += num
else:
    print("¡Límite de suma excedido!")
```

Analicemos ventajas y desventajas de la solución propuesta. ¿Qué opinan?

## Ejemplo 2: Solución Propuesta en la Clase

```
total = 0
n = 1

while n != 0 and total <= 1000:
    s = input("Ingrese un número entero: ")
    n = int(s)

    total += n
    if total > 1000:
        print ("La suma superó a los 1000")
        break

if n == 0:
    print("la suma acumulada es", total)
```

## Ejemplo 3: Sumar números pares

- Se solicita el ingreso de **números pares**.
- Si se ingresa un **número impar**, se informa del problema y no se lo procesa, pero no se interrumpe el ingreso de números.
- La lectura de información finaliza **al leer un cero**.
- Se pide al final del proceso informar **la suma de todos los números pares leídos**.

¿Soluciones sugeridas?

## Ejemplo 3: Solución propuesta

```
total = 0
while True:
    while True:
        num = int(input("Ingrese un número par: "))
        if num % 2 == 0:
            break    # Ingresó un número par...
        print("ERROR: No ingresó un número par.")

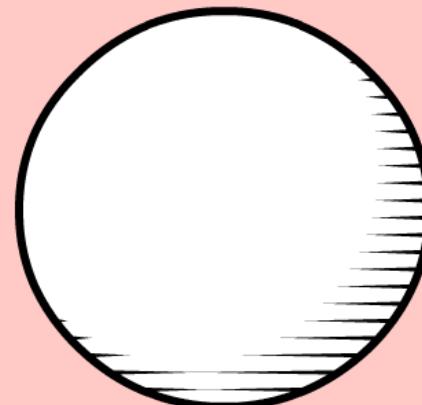
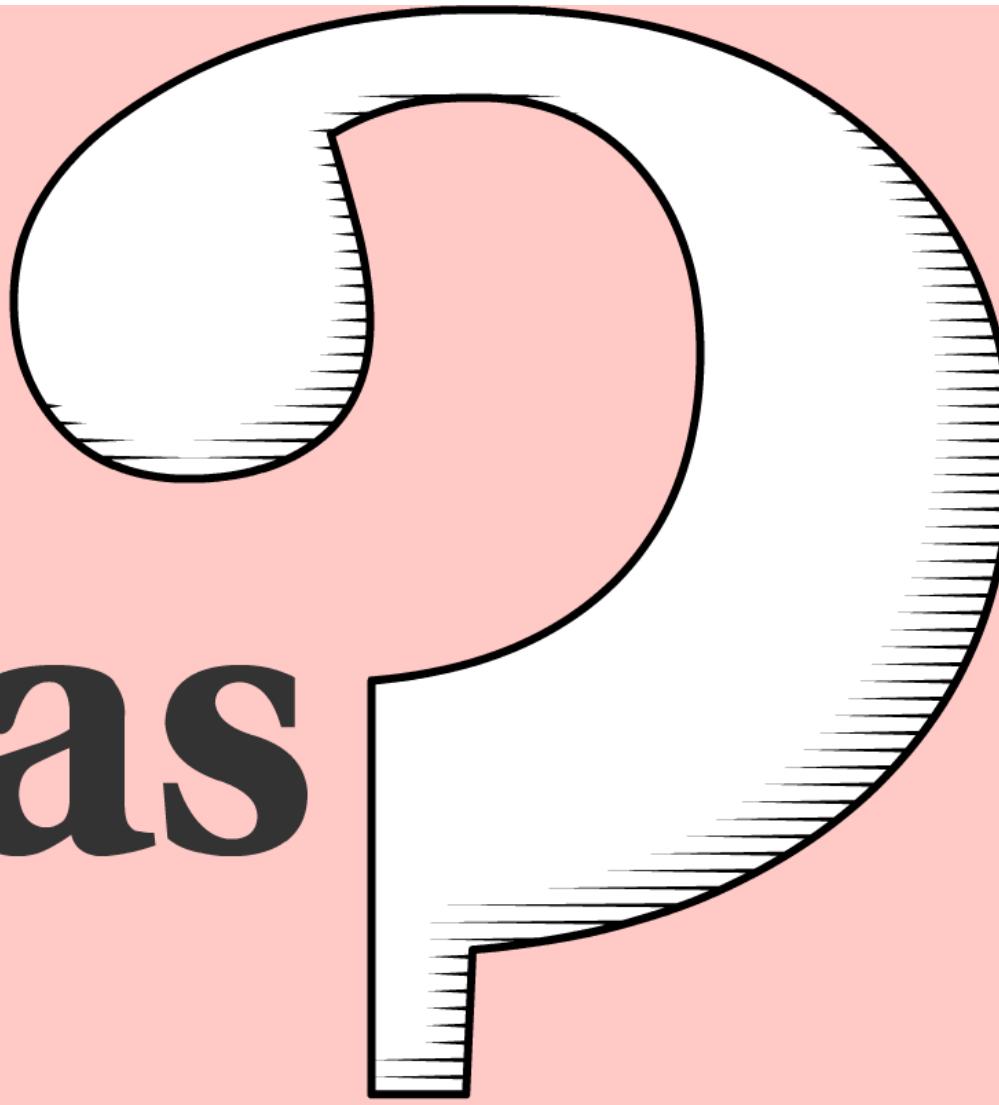
    if num == 0:
        break    # Si se ingresa un cero, para.

    # Mientras tanto, acumula los números ingresados
    total += num

print(total)
```

Analicemos ventajas y desventajas de la solución propuesta. ¿Qué opinan?

ultas



Esta presentación fue diseñada por el siguiente equipo docente:

Mg. Claudia Russo  
Lic. Paula Lencina  
AC María Lanzillotta  
Lic. Leticia Galante  
Prog. Trinidad Picco  
AC. Patricia Miguel  
AC M. del Carmen Muller  
Lic. Mariana Adó  
Lic. Cecilia Rastelli



**Atribución – No Comercial – Compartir Igual (by-nc-sa):**  
No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original. Esta licencia no es una licencia libre.



**Atribución (Attribution):** En cualquier explotación de la obra autorizada por la licencia será necesario reconocer la autoría (obligatoria en todos los casos).



**No Comercial (Non commercial):** La explotación de la obra queda limitada a usos no comerciales.



**Compartir Igual (Share alike):** La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Por lo que las autoras agradecen el reconocimiento de la autoría correspondiente.

Para mayor información se recomienda acceder a :  
<https://creativecommons.org/licenses/by-nc-sa/4.0/>