1. Sentencias repetitivas

En el encuentro anterior se expusieron las sentencias de control *selectivas* en las cuales podíamos bifurcar el flujo o secuencialidad de nuestro algoritmo para ejecutar instrucciones y omitir otras. A diferencia de estas, las sentencias de control **repetitivas** nos permiten *volver a ejecutar* instrucciones dada una condición o expresión lógica.

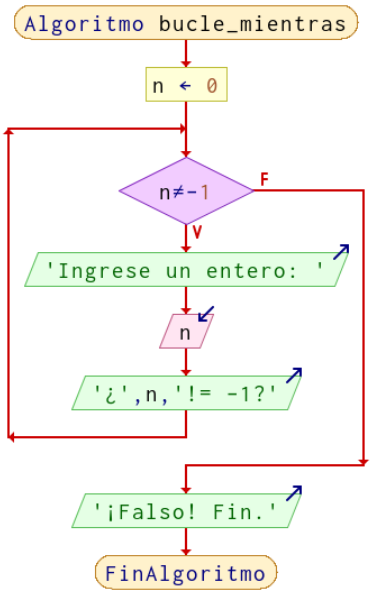
En el ámbito de la programación es común que se quiera evitar, en la medida de lo posible, la repetición de código. Es decir, escribir lo mismo (o casi lo mismo) varias veces en el algoritmo. Es por ello que existen estas estructuras donde uno puede ejecutar las mismas instrucciones cuantas veces lo requiera. A estas estructuras suelen nombrárselas como *bucles* o *loops*.

1.1. Mientras

Así como el condicional simple es la sentencia selectiva más simple, el *"Mientras"* es el primer bucle que suele introducirse. En este caso, las instrucciones dentro de la estructura se repetirán hasta que la *expresión lógica* sea *falsa* o, dicho de otra forma, **mientras** sea *verdadera*. Se utiliza la palabra reservada `while` para este fin.

```python
n = 0
while n != -1:
    n = int(input("Ingrese un entero: "))
    print(f"¿{n} != -1?")

print("¡Falso! Fin.")
```



Aquí se evaluará la expresión `n` $\neq$ `-1` y, de ser verdadera, se pedirá ingresar un entero. *"Mientras"* el entero sea distinto de -1, entonces se seguirá pidiendo ingresar valores. Ha de notarse que `n` existe antes de evaluar la condición con un valor que asegura su estado verdadero, ya que sino no se ingresaría al mismo. Una posible salida de este programa sería:

```
Ingrese un entero: 3
¿3 != -1?
Ingrese un entero: -1
¿-1 != -1?
¡Falso! Fin.
```

¡Importante!

Las expresiones lógicas dentro de una sentencia de repetición **deben *hacerse* falsas en algún momento**, ya que de lo contrario, estaríamos en un ***bucle infinito*** que causaría que nuestro programa "se cuelgue". Si bien en algunos contextos es deseable tener un *bucle infinito*, este debe ser programado a conciencia, nunca por accidente.

1.2. Hacer/Repetir

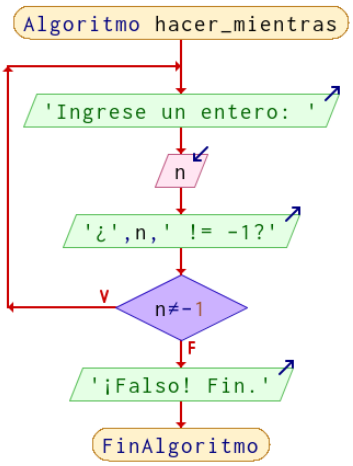
Dependiendo del lenguaje de programación podemos encontrar dos variantes que son similares o hasta podríamos decir *"iguales pero opuestas"*. Una es ***"Hacer ... mientras"*** y la otra suele ser conocida como ***"Repetir ... hasta que"***. Dependiendo del lenguaje de programación podemos encontrar una de las dos, ambas o ninguna. En `Python`, que es el lenguaje que utilizamos no existe ninguna de las dos variantes, pero se analizaran posibles alternativas cuando haya diseños de algoritmos que impliquen su utilización.

1.2.1. Hacer ... Mientras Que

Como se adelantó, en `Python` no existe ninguna de estas sentencias de control. Por lo tanto debemos emularlas. En este caso se utilizará un bucle *"infinito"* y una condición al final que obligará el corte del bucle con la instrucción `break`.

Programa 1: Hacer...mientras

```
1 while True:           # Hacer:
2     n = int(input("Ingrese un entero: "))
3     print(f"¿{n} != -1?")
4     if not n != -1:   # Mientras n != -1
5         break         # fin del bucle
6
7 print("¡Falso! Fin.")
```



A diferencia del `while` expuesto en el apartado anterior, aquí **al menos 1 vez** se ejecutará el bloque, luego se evaluará la condición y se repetirá *mientras* sea verdadera. Nótese que es necesario **negar la condición** con `not`.

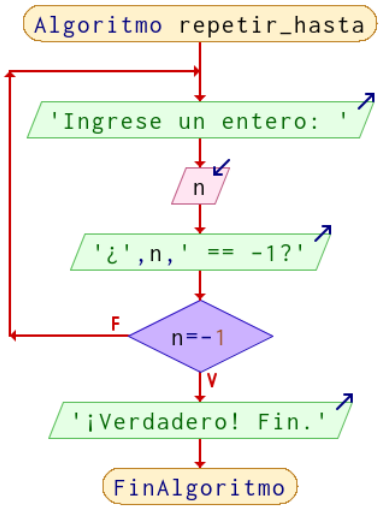
1.2.2. Repetir ... Hasta Que

Al igual que el anterior bloque, en `Python` tampoco existe este, pero es igualmente emulable, sin negar la condición

Programa 2: Repetir...hasta que

```
while True:      # Repetir:
    n = int(input("Ingrese un entero: "))
    print(f"¿{n} == -1?")
    if n == -1:  # Hasta que n == -1
        break    # fin del bucle

print("¡Verdadero! Fin.")
```



El flujo a simple vista parece igual, cambia el punto de vista lógico de la expresión a evaluar. *Repetir hasta que la condición sea verdadera*. Lo cual, como podemos observar equivale a utilizar una expresión contraria a la de un *mientras*.

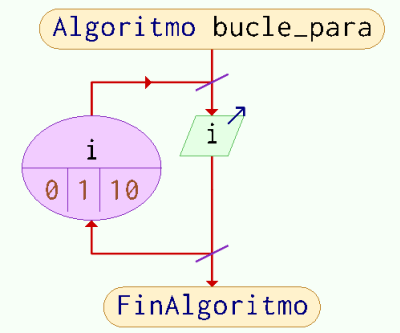
1.3. Para

La sentencia "Para" es una construcción sintética utilizada frecuentemente para ciclar una cantidad finita de veces y, habitualmente, se la utiliza con una variable que auspicia de *"contador"*. Ésta puede incrementar o decrementar según sea necesario.

En `Python` este bucle se implementa con la palabra reservada `for` declarando una variable y seguida por la palabra reservada `in` y en este ejemplo por la instrucción `range()`.

Programa 3: Bucle **para**

```
# Desde 0 hasta 10, paso de a 1
for i in range(0,10,1):
    print(f"{i}.")
```



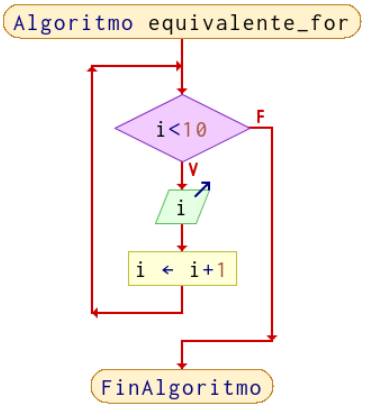
En este caso, se declara la variable "`i`" que adoptará los valores en el *rango* de 0 a 10 incrementando en pasos de 1. En otros diagramas aparece como "(0|10|1)" igual que se escribe en `Python`. La *variable i* se incrementa en cada repetición.

Este algoritmo muestra los números desde el 0 al 9, ya que el rango es "hasta" 10 sin incluirlo, equivale a decir *mientras* `i < 10`.

El `for` anterior es equivalente al siguiente código utilizando un `while`:

Programa 4: Equivalente con **while** del **for**

```
1 i = 0          # Para i desde 0
2 while i < 10: # Hasta 10
3   print(f"{i}.")
4   i += 1       # Incrementar i en 1
```



Incremento/decremento y asignaciones

- +=: asignación con incremento, ejemplo `i+=2` equivale a `i=i+2`.
- -=: asignación con decremento, ejemplo `i-=2` equivale a `i=i-2`.
- *=: asignación con multiplicación, ejemplo `i*=2` equivale a `i=i*2`.
- /=: asignación con división, ejemplo `i/=2` equivale a `i=i/2`.

Sobre el comando `range()`

El comando **`range()`** sirve para recorrer rangos, tiene algunas variantes. Por ejemplo, si el paso es `+1` podría evitarse incluir este dato. Y si el número de inicio es el `0`, entonces también podemos omitir este dato:

```
for i in range(1,11): # por default el paso es 1
    print(f"{i}")     # muestra del 1 al 10

for i in range(10):   # por default inicio en 0
    print(f"{i}")     # muestra del 0 al 9
```

Tenga en cuenta que si quiere hacer rangos decrecientes o de pasos distintos de `+1`, entonces no puede omitir ningún dato:

```
for i in range(0,-10,-1):
    print(f"{i}")   # muestra del 0 al -9

for i in range(0,11,2):
    print(f"{i}")   # 0,2,4,6,8,10
```

2. Ejercitación

Intente resolver los siguientes problemas planteando los algoritmos en diagramas y codifíquelos en `Python`:

1. Pida ingresar una nota entre 1 y 10, y si es mayor o igual a 6 muestre el texto "aprobó" de lo contrario "desaprobó".
2. Pida ingresar 3 notas, verifique que sean entre 1 y 10, y luego muestre el promedio.
3. Muestre los primeros 10 número pares.
4. Muestre los primeros 10 número impares.
5. Muestre la tabla del 2.
6. Pida ingresar un número y un texto, repita el texto la cantidad de veces indicadas por el número ingresado.
7. Pida un número entre 1 y 10, luego muestre su tabla de multiplicar.
8. Pida ingresar una cantidad de notas, las notas deben ser entre 1 y 10, si se ingresa una nota inválida, debe pedirse nuevamente. Luego, calcule el promedio entre ellas.
9. Haga un programa que muestre la tabla ASCII estándar con el formato N° = X donde 'N°' es el valor decimal y 'X' es el carácter a ASCII.
10. Genere los siguientes patrones con bucles: