Herramientas de programación en Python

1. Fundamentos de programación

1.4 Estructuras (Composiciones) Algorítmicas Iterativas

1. while
2. for

Pedro Gomis

pedro.gomis@upc.edu

Composiciones iterativas

- Las estructuras o composiciones iterativas permiten repetir una instrucción o un bloque de instrucciones varias veces. Se llaman también bucles o ***loops***.

Secuencia de datos

- Una **secuencia de datos** se considera como una estructura de **secuencia** si en un algoritmo se puede:
  - (i) acceder al primer elemento de la secuencia,
  - (ii) reconocer el último elemento o su condición y
  - (iii) acceder al elemento n+1 a partir del elemento n.
- Un ejemplo de secuencia de datos serían las notas de los alumnos de una asignatura con 50 alumnos.
- Otro ejemplo de secuencia sería aquella dada por los términos de la expansión en serie de Taylor de la función del seno de un ángulo (en radianes), *seno(x)*:

$$seno(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Composiciones iterativas

Esquemas iterativos

- Los **esquemas iterativos** aplicados a realizar cálculos o utilizar expresiones de todo tipo sobre una **secuencia de datos** suelen clasificarse como esquemas **iterativos** de **búsqueda** o de **recorrido**.

Definite and indefinite loops

- En un **esquema de recorrido** hay que tratar **todos** los elementos de la secuencia para realizar los cálculos necesarios o resolver el problema. Las iteraciones de recorrido son llamadas también **iteraciones definidas**

Por ejemplo, si queremos hallar el valor medio de la nota de los 50 alumnos de la asignatura, debemos hacer un recorrido sobre toda la secuencia de notas.

- En un **esquema de búsqueda** no necesariamente hay que recorrer toda la secuencia de datos para resolver el problema. Este esquema se llama también **iteraciones indefinidas**.

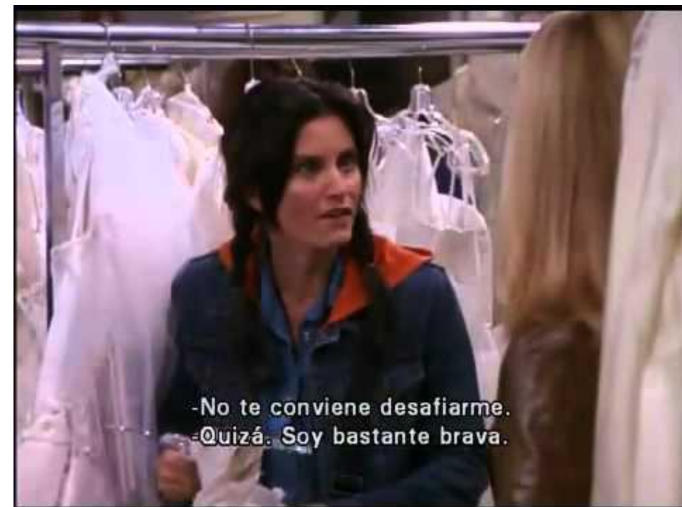
Por ejemplo, si necesitamos saber si hay alguna nota 10, o alguna nota mayor que 9, no hace falta recorrer la secuencia en su totalidad. Si se encuentra antes del final, se para.

Composiciones iterativas

Esquemas iterativos

- **Esquema de recorrido** hay que tratar **todos** los elementos de la secuencia (**iteraciones definidas**)

- **Esquema de búsqueda** no necesariamente hay que recorrer toda la secuencia, al encontrar lo buscado se para (**iteraciones indefinidas**)



Estructura iterativa while

- La sentencia **while** es la composición algorítmica **iterativa** por excelencia que sirve para **cualquier esquema iterativo**.
- Su implementación es similar en todos los lenguajes de programación:

Estructura iterativa `while`

Pseudocódigo	Python	Pascal	C/C++
mientras *condición* **hacer**	**while** *condición*:	**while** *condición* **do**	**while** (*condición*)
Instrucciones	Instrucciones	**begin**	{
		Instrucciones;	Instrucciones
fin_mientras		**end**;	}

- En la estructura **while** el bloque de la secuencia de instrucciones se **repite** siempre que la condición dada por el valor de una variable o expresión booleana ***sea cierta (True)***.

Esquemas iterativos con WHILE en funciones

- **Esquema de recorrido o iteraciones definidas (*definite loops*)** con la composición ***while***, usando funciones:

```
def recorrido():
    Inicializar variables
    Obtener primer elemento
    while NO_final_secuencia:
        secuencia_instrucciones
        Obtener siguiente elemento
return  resultado
```

- **Esquema de búsqueda o iteraciones indefinidas** (***indefinite loops***) con la composición ***while*** usando funciones:

```
def busqueda():
    Inicializar variables
    Obtener primer elemento
    while NO_final_secuencia:
        secuencia_instrucciones
        if elemento actual == elemento buscado:
            return encontrado
        Obtener siguiente elemento
return  no_encontrado
```

Esquemas iterativos con WHILE en cualquier parte del código

- **Esquema de recorrido o iteraciones definidas (*definite loops*)** con la composición ***while***:

```
Inicializar variables
Obtener primer elemento
while NO_final_secuencia:
    tratar_elemento
    Obtener siguiente elemento
```

- **Esquema de búsqueda o iteraciones indefinidas (*indefinite loops*)**:

```
Inicializar variables
Obtener primer elemento
encontrado = False
while NO_final_secuencia and not encontrado:
    secuencia_instrucciones
    if elemento actual == elemento buscado:
        encontrado = True
    Obtener siguiente elemento
```

Estructura iterativa while

- Ejemplo de tabla de multiplicar del número introducido por el usuario. Esto es un **esquema** típico de **recorrido** (***definite loop***)

```
# Tabla de multiplicar (del 1 al 10) del número
introducido
n = int(input('Entra un número entero: '))
k = 1
while k <= 10:
    print(n,'x',k,'=',n*k)
    k = k + 1
print('Tabla de multiplicar del',n)
```

```
def TablaMultiplicar(n):
    """Input: entero n
       Output: void function. Print tabla multiplicar
       de n
    """
    k = 1
    print('Tabla de multiplicar del',n)
    while k <= 10:
        print(n,'x',k,'=',n*k)
        k = k + 1
    print()
```

```
Entra un número entero: 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
Tabla de multiplicar del 7
```

- El ejercicio siguiente muestra los primeros N números naturales en forma decreciente:

```
# Mostrar los N primeros números naturales en forma descendiente
N = int(input('Entra un número natural: '))
i = N
while i >= 1:
    print(i, end =' ')
    i = i - 1
print('\nHemos mostrado',N,'números naturales decreciendo')
```

```
Entra un número natural: 7
7 6 5 4 3 2 1
Hemos mostrado 7 números naturales decreciendo
```

- El ejercicio siguiente muestra los primeros N números naturales en forma decreciente (función):

```
def MostrarNdesc(n):
    """Muestra n naturales descendiendo. Función No productiva
    """
    i = n
    while i >= 1:
        print(i, end =' ')
        i = i - 1
    print()
```

```
>>> MostrarNdesc(13)
13 12 11 10 9 8 7 6 5 4 3 2 1
```

- A continuación se muestra un programa que calcula la suma de los n primeros números naturales:

```
# Calculo de la suma de los primeros n números
n = int(input('Entra un número natural: '))
i = 1
suma = 0
while i <= n:
    suma = suma + i
    i = i + 1

print('La suma de los números naturales hasta',n,'es', suma)
```

```
Entra un número natural: 100
La suma de los números naturales hasta 100 es 5050
```

- A continuación se muestra un programa que calcula la suma de los n primeros números naturales (función):

```
def SumaN(n):
    """Calcula la suma de números naturales del 1 al n dado
    >>> SumaN(20)
    210
    >>> SumaN(10)
    55
    >>> SumaN(100)
    5050
    """
    i = 1
    suma = 0
    while i <= n:
        suma = suma + i
        i = i + 1
    return suma
```

```
>>> SumaN(4)
10
```

- Esquema iterativo de búsqueda (***indefinite loop***). Queremos saber si un número natural es **primo o no**.

Un **número primo** es un **número natural mayor que 1** que tiene únicamente dos divisores positivos distintos: él mismo y el 1

```
# Programa que dice si un número es primo o no
n = int(input('Entra un número natural mayor que 1: '))
d = 2
un_divisor = False     # Si hay un divisor el número NO es primo
while d < n and not un_divisor:
  if n % d == 0:       # Se busca algún divisor
    un_divisor = True
  d = d + 1
print('Es primo? ',not un_divisor)
```

```
Entra un número natural: 29
Es primo?  True
```

- Esquema iterativo de búsqueda (***indefinite loop***). Queremos saber si un número natural es **primo o no**. **Función es_primo**?

while

```
def es_primo(n):
    """Función que devuelve True o False si n es primo o no
    >>> es_primo(57)
    False
    >>> es_primo(59)
    True
    """
    d = 2
    Primo = True      # Si hay un divisor el número NO es primo
    while d <= n//2 and Primo:   # while d*d <= n: o while d <= round(n**0.5):
        if n % d == 0:        # Se busca algún divisor
            Primo = False
        d = d + 1
    return Primo
```

```
>>> es_primo(29)
True
>>> es_primo(28)
False
```

- Queremos saber si un número natural es **primo o no**.

while

```
def es_primo(n):
    """Función que devuelve True o False si n es primo o no
    >>> es_primo(57)
    False
    >>> es_primo(59)
    True
    """
    # El valor 1 No es primo. Detecta el 1 como NO primo
    if n>1:
        d = 2
        while d <= n//2:
            if n % d == 0:       # Se busca algún divisor
                return False
            d = d +  1
        return True
    else:
        return False
```

```
>>> es_primo(29)
True
>>> es_primo(1)
False
```

Estructura iterativa while: Algoritmo clásico (1)

- **Herón de Alejandría** (destacado matemático e inventor griego, siglo I d.C.) describió un **método para aproximar la raíz cuadrada de un número** (se cree que el método ya era conocido en la antigua Babilonia) por aproximaciones sucesivas. Es decir, la raíz cuadrada de x es un número y que satisface $y^2 = x$. Isaac Newton (siglo XVII) propuso posteriormente un método general para hallar las raíces de una función no lineal, f(x) = 0.

El **algoritmo de Herón** para hallar la raíz cuadrada de ***x*** consiste en:

1. Se propone un número candidato cualquiera ***g*** (podemos, por ejemplo, comenzar con ***x***/2).
2. Si ***g*g = x***, o lo suficientemente aproximado de acuerdo a la precisión que queramos entonces paramos y ***g*** es la solución.
3. Sino, se realiza una nueva búsqueda de la raíz de ***x*** promediando el valor de ***g*** y ***x/g***. Es decir, ***(g + x/g)/2***. Este valor lo llamamos también ***g***.
4. Volvemos al paso 2 hasta conseguir que ***g*g*** sea igual o lo suficiente aproximado a ***x***.

Si el valor inicial propuesto de ***g*** lo denominamos g_0 y los siguientes valores son g_n, donde **n** = 1, 2, …, entonces las propuestas de raíz de ***x***, g_n, en la iteración ***n*** es función de las propuestas previas g_{n-1} y el número ***x***:

$$g_n = \frac{g_{n-1} + \frac{x}{g_{n-1}}}{2}$$

Estructura iterativa while: Algoritmo clásico (1)

```
# Algortimo de Herón. Cálculo de la raíz cuadrada
print('Cálculo de la raíz cuadrada de un número (algoritmo de Herón')
x = float(input('Entra un número para hallar su raíz cuadrada: '))
g = x/2             # El valor inicial g (guess), g0, puede ser x/2
epsilon = 1e-10     # Tolerancia de la aproximación a la raíz cuadrada
dif = g*g - x       # Diferencia entra el valor de raíz g0 propuesto y el real
while abs(dif) > epsilon:
    g = (g + x/g)/2
    dif = g*g – x
    print('Raiz de',x,'aprox:',g) # print usado para rastrear la búsqueda
print('La raíz cuadrada (aproximada) de',x,'es', g)
```

```
Entra un número para hallar su raíz cuadrada: 9
Raiz de 9.0 aprox: 3.25
Raiz de 9.0 aprox: 3.0096153846153846
Raiz de 9.0 aprox: 3.000015360039322
Raiz de 9.0 aprox: 3.0000000000393214
Raiz de 9.0 aprox: 3.0
La raíz cuadrada (aproximada) de 9.0 es 3.0
```

Estructura iterativa while: Algoritmo clásico (1)

```
# Algortimo de Herón. Cálculo de la raíz cuadrada
def raiz(x,epsilon=1e-10):
    """Raiz cuadrada, método de Heron
    >>> raiz(9)
    3.0
    """
    g = x/2          # El valor inicial g (guess), g0, puede ser x/2
    dif = g*g - x    # Diferencia entra el valor de raíz g0 propuesto y el real
    while abs(dif) > epsilon:
        g = (g + x/g)/2
        dif = g*g - x
    return g
a = float(input('Entra un número para hallar su raíz cuadrada: '))
r = raiz(a)
print('La raíz cuadrada (aproximada) de',a,'es', r)
from math import sqrt
print('La raíz cuadrada con fun sqrt de',a,'es', sqrt(a))
```

```
>>> raiz(4)
2.0
>>> raiz(10)
3.162277660168379
```

Estructura iterativa while: Algoritmo clásico (2)

- El **algoritmo de Euclides para hallar el máximo común divisor de 2 números naturales** es uno de los primeros métodos para resolver problemas utilizando una secuencia de instrucciones y toma de decisiones lógicas para repetir los cálculos de forma iterativa o de bucles.
- El método propuesto por Euclides (siglo III a.C.) para hallar el máximo común divisor (MCD) de dos números naturales, propone que:

Dados 2 números naturales, **a** y **b**, comprobar primero si son iguales.

I. Si **a** y **b** son iguales entonces **a** es el MCD.

II. Si no son iguales, entonces probar si **a** es mayor que **b**
   1. si lo es, restar a **a** el valor de **b**,
   2. Pero si **a** es menor que **b**, entonces restar a **b** el valor de **a**.

III. Repetir el procedimiento con los nuevos valores de **a** y **b**.

Estructura iterativa while: Algoritmo clásico (2)

```
# Algortimo de Euclides en Python
print('Cálculo del MCD de 2 números usando el algoritmo de Euclides')
a = int(input('Entra un número natural: '))
b = int(input('Entra otro: '))
aa, bb = a, b
while a != b:
    if a > b:
        a = a-b
    else:
        b = b-a
print('El MCD entre',aa,'y',bb, 'es', a)
```

```
Cálculo del MCD de 2 números usando el algoritmo de
Euclides
Entra un número natural: 28
Entra otro: 12
El MCD entre 28 y 12 es 4
```

Estructura iterativa while: Algoritmo clásico (2)

```
# Algortimo de Euclides en Python
def mcd(a,b):
    """De acuerdo al algoitmo de Euclides, halla máximo
       común divisor de a y b
    >>> mcd(8,20)
    4
    >>> mcd(24,17)
    1
    """
    while a != b:
        if a > b:
            a = a-b
        else:
            b = b-a
    return a
```

```
>>> mcd(2366, 273)
91
>>> mcd(30, 12)
6
>>> mcd(30, 10)
10
```

Acertar un número

- El programa siguiente realiza un juego típico de adivinar un número oculto, que es generado aleatoriamente por el computador.
- La solución para el jugador es dividir la búsqueda en dos conjuntos y así ir acotando el conjunto de números hasta encontrar la solución. Se suele llamar a esta estrategia "**búsqueda binaria o dicotómica**". Se ha incluido en el programa el límite de 100 para el número natural a ser adivinado. En este caso 100 es menor que 2^7 por lo que con 7 intentos será suficiente para adivinar el número.

```
# Juego de adivinar número oculto (generado aleatoriamente)
from random import randint
oculto = randint(1,100)     # Se genera aleatoriamente un número del 1 al 100
print('Se ha generado un número del 1 al 100')
print('Adivínalo en máximo 7 intentos, ')
intentos = 1
x = int(input('Adivina el número: '))
while (x != oculto) and (intentos < 7):
  if x > oculto:
    x = int(input('Prueba uno menor(quedan '+str(7-intentos)+' intentos): '))
  else:
    x = int(input('Prueba uno mayor(quedan '+str(7-intentos)+' intentos): '))
  intentos = intentos + 1
if x == oculto:
  print('Muy bien! lo has conseguido en', intentos, 'intentos')
else:
  print('Lástima, el número oculto era el', oculto)
```

Acertar un número En función

```
def juego_acertar_número():
    """
    Sin doctest pues se generan números aleatorios
    >>>juego_acertar_número()
    """
    from random import randint
    oculto = randint(1, 100)
    print('He pensado un número entre 1 y 100, aciértalo')
    acertar = int(input("Di un número entre 1 y 100: "))
    intentos = 1
    while acertar != oculto and intentos < 7:
        if acertar > oculto:
            acertar = int(input('Es menor (quedan '+str(7-intentos)+' intentos): '))
        else:
            acertar = int(input('Es mayor (quedan '+str(7-intentos)+' intentos): '))
        intentos += 1
    if acertar == oculto:
        print('Muy bien! lo has conseguido en', intentos, 'intentos')
    else:
        print('Lástima, el número oculto era el', oculto)
```

Estructura while para asegurar condiciones de entrada de datos

- Las raíces cuadradas se pueden calcular solo para números positivos.
- El programa de Herón realizado entrará en iteraciones infinitas si probamos un número negativo (si es menor que -epsilon).
- Introducimos un esquema muy usado para asegurar requisitos de entrada en programas informáticos. En este caso se debe cerciorar que el número introducido por el usuario sea positivo.

```
x = float(input('Entra un número para hallar su raíz cuadrada: '))
while x < 0:
    x = float(input('El número debe ser positivo!!: '))
```

- Este esquema puede ampliarse a múltiples casos, cambiando la condición lógica del `while`.

Ejercicios

1. Seno por serie de Taylor. Diseña una función seno_t(x, error=1e-9) que reciba un valor en grados, lo transforme en radianes y que aproxime el valor de su seno, según su desarrollo en serie de Taylor. La función puede tener el valor del error por omisión, que sirve para terminar la iteración cuando un término sea menor que el error.

$$sin\,(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

2. Diseña una función seno_t2(x, error=1e-9) que reciba un valor en grados, lo transforme en radianes y que aproxime el valor de su seno, según su desarrollo en serie de Taylor.

   Nota: No utilizar ni las funciones factorial de math ni elevar x^n, usar los términos anteriores para reducir el tiempo de cálculo.

Composiciones iterativas

Estructura iterativa *for*

- Típico para esquemas **iterativo** de recorrido (***definite loops***).

Pseudocódigo	Python	Pascal	C/C++

desde inicio **a** fin **hacer**
Instrucciones
 	
fin_desde
	for item **in** secuencia:
	Instrucciones
		for var:= ini **to** fin **do**
		begin
		Instrucciones;
		end;
			for (ini; cond; fin)
			{
			Instrucciones
			}

- Ejemplo:

```
# Saludo de cumpleaños
for nombre in ['Leo', 'Ronaldo', 'Andrés', 'Sergio']:
    print('Feliz cumpleaños', nombre)
print('Ya hemos saludado a los amigos')
```

```
Feliz cumpleaños Leo
Feliz cumpleaños Ronaldo
Feliz cumpleaños Andrés
Feliz cumpleaños Sergio
Ya hemos saludado a los amigos
```

La secuencia de datos entre corchetes forma una **lista** en Python

Composiciones iterativas

Estructura iterativa *for*

- Típico para esquemas **iterativo** de recorrido(***definite loops***).
- Ejemplo:

```
# Potencias de 2
for posicion in [1, 2, 3, 4, 5, 6, 7, 8]:
    print('Peso del bit', posicion, '=',2**(posicion-1))
```

```
Peso del bit 1 = 1
Peso del bit 2 = 2
Peso del bit 3 = 4
Peso del bit 4 = 8
Peso del bit 5 = 16
Peso del bit 6 = 32
Peso del bit 7 = 64
Peso del bit 8 = 128
```

La secuencia de datos entre corchetes forma una **lista** en Python

Composiciones iterativas

Estructura iterativa *for*

- Tipo de dato **range**(): En las versiones 2.x de *Python*, **range**() es una función que devuelve una **lista**. El concepto ha cambiado en la versión 3, donde pasa a ser un tipo de dato **range**, que viene a ser una secuencia de enteros usada típicamente para recorrer bucles **for**.
- Al ejecutar:

```
for elemento in range(8):
    print(elemento, end = ' ')
```

```
0 1 2 3 4 5 6 7
```

- De los ocho elementos que recorre la variable **`elemento`**, el primero es el 0 y el último el 7.
- Si quisiéramos emular la lista [1, 2, 3, 4, 5, 6, 7, 8] del ejemplo Potencia de 2 debemos usar **range**(1, 9). Comienza en 1 y termina en 9-1:

```
# Potencias de 2
for posicion in range(1, 9):
    print('Peso del bit', posicion,'=',2**(posicion-1))
```

Composiciones iterativas

Estructura iterativa *for*

- Sintaxis de **range**() es (lo que está entre corchetes es opcional):

```
range([inicio,] final [, pas])
```

- Ejemplos:

```
range(5)          #  → [0, 1, 2, 3, 4]  No se alcanza el 5 sino 5-1
range(2,6)        #  → [2, 3, 4, 5]     Comienza en 2 hasta 6-1
range(0,10,3)     #  → [0, 3, 6, 9]       De 0 hasta 9 de en pasos de 3
range(5,0,-1)     #  → [5, 4, 3, 2, 1] De 5 hasta 1 de en pasos de -1
range(0,-10,-3) #  → [0, -3, -6, -9]    De 0 hasta -9 de en pasos de -
3
```

- `range(1, 9)` si se convierte a lista:

```
>>> list(range(1,9))
[1, 2, 3, 4, 5, 6, 7, 8]
```

Composiciones iterativas

Estructura iterativa *for*

```
# Tabla de multiplicar (del 1 al 10) del número introducido (versión for)
n = int(input('Entra un número entero: '))
for k in range(1, 11):
    print(n,'x',k,'=',n*k)
print('Hemos mostrado la tabla de multiplicar del',n)
```

```
Entra un número entero: 7
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
Hemos mostrado la tabla de
multiplicar del 7
```

Composiciones iterativas

Estructura iterativa *for*

```
# Mostrar los N primeros números naturales en forma descendiente (for)
N = int(input('Entra un número natural: '))
for i in range(N, 0, -1):
    print(i, end =' ')
print('\nHemos mostrado',N,'números naturales decreciendo')
```

```
Entra un número natural: 12
12 11 10 9 8 7 6 5 4 3 2 1
Hemos mostrado 12 números naturales decreciendo
```

```
# Calculo de la suma de los primeros n números (versión for)
n = int(input('Entra un número natural: '))
suma = 0
for i in range(1, n+1):
    suma = suma + i
print('La suma de los números naturales hasta',n,'es', suma)
```

```
Entra un número natural: 100
La suma de los números naturales hasta 100 es 5050
```

Composiciones iterativas

- Ejemplo: Queremos saber si un número natural es **primo o no**, usando **for** en una función. El `return` nos hace salir de la función “No hace falta usar el `break`”

*Versión con **for** en función*

```
# Programa que dice si un número es primo o no
def es_primo2(n):
    """Encuentra si un número natural > 1 es primo o no
    """
    for d in range(2, n//2+1):
        if n % d == 0:        # Se busca algún divisor
            return False
    return True
x = int(input('Entra número natural mayor que 1: '))
print('Es primo?',es_primo(x))
```

Un **número primo** es un **número natural mayor que 1** que tiene únicamente dos divisores positivos distintos: él mismo y el 1

Composiciones iterativas

Estructura iterativa *for*

- La sentencia **for** recorre una secuencia de datos como las listas, ***range*** y ***strings***. Veamos un simple ejemplo con **for** recorriendo un ***string***

```
# for en secuencia tipo string
for i in 'Hola':
    print('Imprimo:',i)
```

```
Imprimo: H
Imprimo: o
Imprimo: l
Imprimo: a
```

Composiciones iterativas

Sentencias break y continue

- Existen en Python sentencias que permiten ***salir*** durante una **iteración** o definitivamente de una **estructura iterativa**.
- La sentencia **break** hace que se salga inmediatamente del bucle al ser accedida, es decir, se sale del bloque de instrucciones de la iteración **for** o **while**.

Esto implica que aquellas instrucciones de la secuencia que están a continuación del break ya no se ejecutan y se sale de la estructura iterativa.

- En cambio la sentencia **continue** hace que se salten las instrucciones dentro del bloque de la estructura iterativa, pero solo en la iteración actual.

*Es decir, se continuará con la siguiente iteración dada por el **for** o el **while**.*

- Estas sentencias deben utilizarse con ***moderación***, ya que introduce una excepción a la lógica de control normal del bucle.

Composiciones iterativas

Esquemas iterativos con FOR en funciones

- **Esquema de recorrido o iteraciones definidas (*definite loops*)** con la composición ***for***, usando funciones:

```
def recorrido():
    for elemento in secuencia:
        tratar_elemento
return resultado
```

- **Esquema de búsqueda o iteraciones indefinidas** (***indefinite loops***) con la composición ***for*** usando funciones:

```
def busqueda_en_secuencia():
    for elemento in secuencia:
        if elemento == elemento_buscado:
            return encontrado
return no_encontrado
```

Composiciones iterativas

Esquemas iterativos con FOR en cualquier parte del código

- **Esquema de recorrido o iteraciones definidas (*definite loops*)** con la composición ***for***:

```
for elemento in secuencia:
    tratar_elemento
```

- **Esquema de búsqueda o iteraciones indefinidas** (***indefinite loops***) *Usar con moderación este esquema*. Es preferible usar la composición ***while***:

```
encontrado = False
for elemento in secuencia:
    if elemento == elemento_buscado:
        encontrado = True
        break
```

Composiciones iterativas

Estructura iterativa *for. Bucles para efectuar sumatorias*

- En forma general, la serie o sumatoria de términos, f_i, que dependan de la ubicación del término i, se representan por

$$S = \sum_{i=m}^{N} f_i = f_m + f_{m+1} + f_{m+2} + \cdots + f_{N-1} + f_N$$

- La sumatoria de términos de la serie convergente que representa la paradoja de la dicotomía de Zenón de Elea, podemos usar el esquema clásico de sumatorias en lenguajes de programación

$$suma = \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} \cdots$$

```
# Paradoja de la dicotimia de Zenón (términos definidos)
n = int(input('Entra el número de términos a calcular: '))
suma = 0
for i in range(1, n+1):
    suma = suma + 1/2**i
print('El valor de la sumatoria es', suma)
```

```
Entra el número de términos a calcular: 15
El valor de la sumatoria es 0.999969482421875
```