Son estructuras diseñadas para que cierto bloque de ordenes se repita una cantidad determinada de veces o que se ejecuten mientras se cumplen ciertas condiciones. Los Bloques son definidos por “indentation” o sea la sangría. No hay una orden específica para indicar el final, este se asume con la desaparición de la sangría usada en esa estructura de control. Aunque puede continuar otra de una estructura superior a esta. Es decir, cuando mas interna es la estructura mas larga es la sangría que determina la pertencia de las órdenes.

En las estructuras de control tenemos las siguientes órdenes

- **for** para realizar bucles o loops.
- **if( )/elif/else** para hacer preguntas y tomar decisiones.
- **while( )** para realizar una tarea mientras el argumento sea verdadero.
- **“Comprehension list”** que son una versión compacta de los comandos anteriores y sirven para realizar todos los comandos anteriores en un sólo renglón. Sirven para volver más compacto el código.

1 FOR

Empecemos con la sentencia **for** para realizar “loops” o bucles

Lo uso haciendo:

for *variable* in *contenedor*:

Bloque determinado con una sangría similar que contiene las operaciones a realizar

La *variable* es similar a como usamos la variable de en un *DO* de Fortran. El contenedor puede ser cualquiera de los que vimos la clase anterior. Es de decir una Lista, Tupla, Set, etc.

Los “**:**” al final de la línea son obligatorios porque avisan que ahora viene el bloque de órdenes del bucle.

La idea es que la variable va tomando a todos los elementos del contenedor. En muchos textos a los elementos del contenedor se los considera como *iterables*. Dado que itero sobre todos los elementos de ese conjunto.

Por lo cual las operaciónes que se especifiquen en el bloque del **for** serán aplicadas a todos los elementos del contenedor.

También hay recordar que un contenedor puede tener como un elemento a otro contenedor, así que donde hablamos de *variable* podría ser algo que englobe más de una variable de las básicas. Por ejemplo, podría tener una lista de tuplas, por lo cual el **for** actuaría de a una tupla por vez.

[1]:
```
for i in [1,2,3]:
      print(i+1)        # Prestar atención a la sangría
                        # ("indentation" en inglés)
```

```
2
3
4
```

Recordando que un contenedor puedo tener distintos tipos de variables básicas, podríamos hacer los siguiente:

[2]:
```
for cosa in [1,'ff',2]:
    print(cosa)
    print('end')
print('final end')    # El final de la sangría indica el final
                      # del bloque "for"
```

```
1
end
ff
end
2
end
final end
```

Por ejemplo, puedo recorrer un diccionario por sus *keys*.

[3]:
```
# Si defino un diccionario
ATOMIC_MASS = {}  # <-- Diccionario vacio

ATOMIC_MASS['H'] = 1
ATOMIC_MASS['He'] = 4
ATOMIC_MASS['C'] = 12
ATOMIC_MASS['N'] = 14
ATOMIC_MASS['O'] = 16
ATOMIC_MASS['Ne'] = 20
ATOMIC_MASS['Ar'] = 40
ATOMIC_MASS['S'] = 32
ATOMIC_MASS['Si'] = 28
ATOMIC_MASS['Fe'] = 55.8

# Puedo imprimir a partir de la keys, todos los valores del diccionario.
#  Ojo. la salida no está ordenada
# dict.keys() es el método que lista los "keys" de diccionario

for key in ATOMIC_MASS.keys():
    print(key, ATOMIC_MASS[key])
```

```
H 1
He 4
C 12
N 14
O 16
Ne 20
Ar 40
S 32
Si 28
Fe 55.8
```

2 IF/ELIF/ELSE

Puedo hacer una preguna al estilo:

```
if(Condición):
    pasa esto si "_condición_" es verdadero
elif(otra Condición):
    pasa esto otro si "_otra condición_" es verdera y "algo" fue falso
else:
    Como las anteriores fueron falsas, hago esto que sigue acá
```

Note la sangría que ordena los inicio y finales en cada bloque de actividades.

La *Condición* puede ser una variable lógica que contiene un verdadero o un falso, o una comparación entre magnitudes realizada por un operador. A estos casos los veremos a continuación.

Se pueden utilizar las sentencias de Control con un estilo bastante parecido al Fortran salvo por la orden que en Fortran se llama “elseif()” y en Python “elif()”

Note que cada orden de la estructura del bloque lleva los “**:**” al final

3 Operadores condicionales

En el caso de que requiera comparar dos valores o más, deberé escribir la condición en el IF(), y este se activará cuando esa condición sea verdadera. Las condiciones para activar o no un IF() tienen en python un muy rico lenguaje para detallar eventos que pueden ser verdaderos o falsos.

Operadores de Comparación: Se comparan números entre ellos, recordar que se puede poner una expresión que será evaluada antes de realizar la comparación.

Operador	Descripcion	sintaxis
>	Mayor	X > Y
<	Menor	X < Y
==	igual	X == Y
>=	Mayor o igual	X >= Y
<=	Menor o igual	X <= Y
!=	No es igual	X != Y

Operadores Lógicos:

Operador	Descripcion	sintaxis
and	Verdadero si ambos son verdaderos	X and Y
or	Verdadero si uno es verdadero	X or Y
not	Verdadero si es falso, y falso si es verdero	not X

Operadores de indentidad: Verifican que ocupen la misma memoria ram, con otras palabras, se pregunta si son el mismo objeto o parte de él.

Operador	Descripcion	sintaxis
is	x es igual a y	X is Y
is not	x no es igual a y	X is not Y

En el caso de los diccionarios se usa el operador '**in**' para preguntar si una 'key' en particular está definida. El modo de hacer estas operaciones sería el siguiente:

[4]:
```
num = 223.4

if num > 0:
    print("Es un número positivo")
elif num == 0:
    print("Cero")
else:
    print("Es un número negativo")
```

```
Es un número positivo
```

Y puede tener estructuras de control una dentro de otra pero las diferentes sangrías me indican los niveles en que cada estructura es válida.

[5]:
```
for i in range(10):
    if i > 5:
        print(i)

# Prestar atención a la doble sangría del if
```

```
6
7
8
9
```

[6]:
```
for i in range(10):
    if i > 5:
        print(i)
    else:
        print(i ,'es menor que cinco')
print('END')
```

```
0 es menor que cinco
1 es menor que cinco
2 es menor que cinco
3 es menor que cinco
4 es menor que cinco
5 es menor que cinco
6
7
```

```
8
9
END
```

También puedo usar estructuras de control en un SET aunque sus elementos no estén numerados

[7]:
```
este_set = {"frutilla", "banana", "cereza"}

for x in este_set:
  print(x)
```

```
banana
frutilla
cereza
```

En el caso de los diccionarios se usa el operador '**in**' para preguntar si una llave o 'key' en particular está definida. Veamos su modo de uso utilizando el diccionario que definimos anteriormente y preguntemos si 'mapa' nos dispara un resultado (spoiler: no está)

[8]:
```
a_diccionario = {'uno' : 1.0,
                 'dos' : 2.0,
                 'una_lista' : ['esta', 'es', 'una', 'lista']
                 }

if 'mapa' in a_diccionario:
     print(a_diccionario['mapa'])
else:
     print('Esa llave no existe')
```

```
Esa llave no existe
```

También para el caso de un diccionario se puede pedir que la orden **for**, recorra simultáneamente llaves y valores. Esto se puede hacer porque la función **items( )** de los diccionarios crea una tupla (llave, valor)

Esta orden puede ser muy confusa y se tarda en entenderla... <– **tomarlo con calma**

[9]:
```
print( a_diccionario.items())
```

```
dict_items([('uno', 1.0), ('dos', 2.0), ('una_lista', ['esta', 'es', 'una',
'lista'])])
```

Note que se imprimen una lista de tuplas, donde cada tupla es (llave, valor) y si es una lista puedo usar un for, pero descomponiendo la tupla en sus dos componentes por separado (recordar que esto es una propiedad de las tuplas).

[10]:
```
for llave, valor in a_diccionario.items():
    print(llave,'=', valor)
```

```
uno = 1.0
dos = 2.0
```

```
una_lista = ['esta', 'es', 'una', 'lista']
```

4 Try/Except

Estos comandos tienen una estructura perecida a un if()/else, pero diseñada para manejar errores.

Lo que escribo en el bloque del **try:** es probado y si es cierto ejecutado, pero si no lo es se activa el bloque del **except tipo_de_error**. Es decir, genera una excepción, avisa de ella y programa sigue corriendo, no muere como debería haber pasado. Hay un cantidad muy importante de **tipos de errores** y en cada versión de python se han ido agregando más de ellas. Para un listado se debe buscarlas en los manuales de la versión de Python que se esté utilizando.

Probémoslo con el diccionario, y en este caso la excepción es 'KeyError'

[11]:
```
try:
  print(a_diccionario['dos'])
except KeyError:
  print('No estaba la llave')
```

```
2.0
```

[12]:
```
# y si pido una llave inexistente, se activa
# el except

try:
  print(a_diccionario['mapa'])
except KeyError:
  print('No estaba la llave')
```

```
No estaba la llave
```

5 WHILE( ), Break y Continue

El comando while(algo) funciona creando un loop mientras "algo" sea verdadero. En "algo" podemos construir un condicional con los operadores que hemos visto.

Por ejemplo:

[13]:
```
i = 1
while i < 4:
  print(i)
  i += 1
```

```
1
2
3
```

Podemos incluso poner otras estructuras de control y generar un **break** que rompe al while dada una condición, veamos como:

[14]:
```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

Entonces cuando i sea igual a 3 el loop terminará.

O con el comando **continue** evitar que cierto valor sea procesado, ciclando el while() al próximo valor.

[15]:
```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
6
```

Notar que el valor “3” no fue considerado en la secuencia

6 Listas de Comprensión (List Comprehension)

Estas son formas de estructuras de control extremamadamente compactas y bastante (o muy!!!) confusas cuando se las ve por primera vez. Se caracterizan por ahorrar muchas líneas de código.

Se basan en la idea de “iterar” sobre una estructura de control para generar sus valores, o bien, por ejemplo, convertir una lista en otra (por un cálculo, o un filtrado de sus elementos). Una variantes de estas son las funciones generadoras que veremos en un capítulo siguiente.

Veamos un ejemplo de como funcionan:

[16]:
```
# creamos unas listas
animales = ['perro','gato','loro', 'tortuga','paloma']
numeros = [1,2,3,4,5]

# y las procesamos
# Por ejemplo, creo una nueva lista que sólo estén los animales
# que tengan una "a" en el nombre

nuevos_animales = [x for x in animales if "a" in x]
```

```
print("Nueva lista:",nuevos_animales)


# o calculo los cubos de los números de la lista "números"
nuevos_numeros= [x**3 for x in numeros]
print("Nuevos Números:", nuevos_numeros)
```

```
Nueva lista: ['gato', 'tortuga', 'paloma']
Nuevos Números: [1, 8, 27, 64, 125]
```

La sintaxis más general de una list comprehension es:

newlist = [expresión for item in colección if condición == True]

que en realidad se puede reducir a:

newlist = [expresión for item in colección]

Pero existe mucha libertad de como usar esta orden, por ejemplo podría haber más de if() o más de una collection de items

newlist = [expresión for item in colección if test1 and test2]

o

newlist = [expresión for item in colección1 and item2 in colección2]

Por colección nos referimos a un iterable, y la definición exacta de un iterable la veremos más adelante, pero por ejemplo las listas, tuplas, etc son iterables. Es decir puedo acceder uno por uno en un conjunto de elementos.

Note que en la definición de un List Comprehensions va rodeada de [] (corchetes) como la hace la definición de una lista en Python

Por ejemplo, puedo recorrer una lista haciendo:

[17]:
```
lista_nueva = [x for x in range(10)]
print(lista_nueva)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

O recorrerla poniendo una condición:

[18]:
```
lista_nueva = [x for x in range(10) if x < 5]
print(lista_nueva)
```

```
[0, 1, 2, 3, 4]
```

O incluso aprovechar los métodos o funciones de los objetos:

[19]:
```
# la función upper pasa la letra a mayúscula
newlist = [x.upper() for x in animales]
print(newlist)
```

```
['PERRO', 'GATO', 'LORO', 'TORTUGA', 'PALOMA']
```

Al principio, estas listas de compresión son confusas, pero esta estructura puede entenderse mejor si se la piensa dividida en renglones donde cada uno de ellos desarrolla una actividad. Por ejemplo, si tengo

[x**2 for x in range(10) if x<5]

Expresión que puedo dividir en 3 líneas:

- la expresión matemática,
- la iteración y
- la condición para que se ejecute.

En este ejemplo:

x^2 es la expresión matemática

for x in range(10) es la iteración

if x < 5 es la condición

Separando en las 3 partes es más fácil de entender

[20]:
```
[x**2 # expresión matemática
  for x in range(10) # iteración
  if x < 5] # condición
```

[20]:
```
[0, 1, 4, 9, 16]
```