

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.handle=x
                self.func=func
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

04.1 - Python

Estructuras de control

I

Indice

- Introducción
- Estructuras de selección
 - La sentencia *if*
 - La construcción *if - else*
- Estructuras iterativas
 - El bucle *while*
 - El bucle *for-in*
 - La función *range*
 - Las sentencias *break*, *continue* y *pass*

```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
    def listen(self,host,port,func):
        try:
            self.sock.bind((host,port))
            self.sock.listen(2)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error:Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x,ho=self.sock.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
            self.send(self.data)
            self.handle.send(data)
        self.close(self)
        self.sock.close()
```

Introducción

Control del Flujo de Ejecución

- Las **sentencias de control del flujo de ejecución** de nuestros programas permiten, bajo determinadas **condiciones**, alterar la ejecución **secuencial** de las instrucciones de nuestros programas.
- Python proporciona diversas instrucciones para controlar flujo de ejecución:
 - ***Sentencias condicionales o de selección:*** **if**
Permiten tomar decisiones en base a los datos y/o resultados y, en función de estos, ejecutar ciertas sentencias y otras no
 - ***Sentencias iterativas o de repetición:*** **for, while**
Permiten repetir la ejecución de una serie de instrucciones mientras se cumpla determinadas condiciones (expresión booleana)

Estructuras de selección (I)

- Supongamos que diseñamos un programa para la resolución de ecuaciones de 1^{er} grado de la forma:

$$a \cdot x + b = 0$$

- Una posible solución podría ser:

```
'''  
ecu_1G_solver.py    ver.1  
Resuelve ecuaciones de 1er grado  
'''  
  
# coeficientes  
a = float(input('Introduce el coeficiente A: '))  
b = float(input('Introduce el coeficiente B: '))  
  
# resultado  
x = -b/a  
print(f'La solución es x = {x:.2f}')
```

Estructuras de selección (II)

- Vamos a ejecutar nuestro programa:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 2  
Introduce el coeficiente B: 5  
La solución es x = -2.50
```



- Sin embargo, vamos a ejecutarlo de nuevo con estos otros coeficientes:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 0  
Introduce el coeficiente B: 4  
Traceback (most recent call last):  
File "ecu_1G_solver.py", line 10, in <module>  
    x = -b/a  
ZeroDivisionError: float division by zero
```



- Como era de esperar, se ha producido un error al tratar de realizar una división donde el divisor es 0!! Nuestro programa debe controlar esto!!

Estructuras de selección (III)

La sentencia condicional *if*

- Python nos proporciona la instrucción *if* para evaluar una **condición** y, **sólo en caso de que sea cierta**, ejecutar las acciones que consideremos.

Su formato es:

```
if condición:  
    acción  
    acción  
    ...
```

- condición** podrá ser cualquier *expresión* que devuelva un valor lógico. Si el resultado de evaluar dicha expresión es *verdadero* (**True**), se ejecutarán aquellas *acciones* contenidas dentro del **bloque** de instrucciones delimitado por los dos puntos (:) de *inicio de bloque* y **sangradas** al menos un espacio respecto al inicio de la sentencia *if*. **Todas** las sentencias del bloque deben tener la **misma indentación**.

Sentencias condicionales (IV)

- Vamos a rehacer nuestro programa:

```
'''
ecu_1G_solver.py    ver.2
Resuelve ecuaciones de 1er grado
'''

# coeficientes
a = float(input('Introduce el coeficiente A: '))
b = float(input('Introduce el coeficiente B: '))

# resultado (si a!=0 se ejecutan las instrucciones sangradas a la derecha)
if a != 0:
    x = -b/a
    print('La solución es x = {0:.2f}'.format(x))

# fin programa
print('Fin del programa')
```

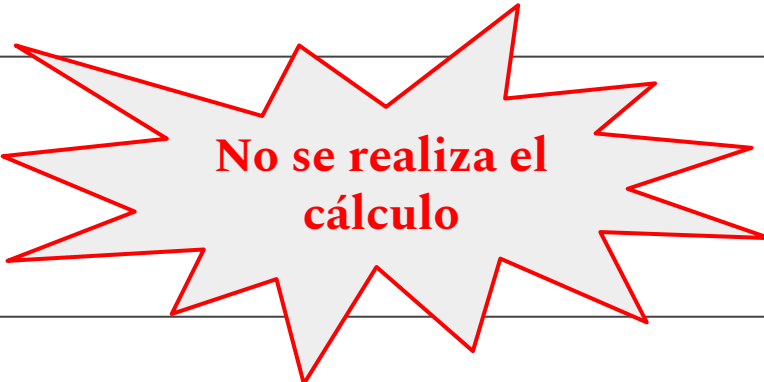
Sentencias condicionales (V)

- Vamos a ejecutar de nuevo nuestro programa:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 2  
Introduce el coeficiente B: 5  
La solución es x = -2.50  
Fin del programa
```

- Sin embargo, vamos a ejecutarlo de nuevo con estos otros coeficientes:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 0  
Introduce el coeficiente B: 4  
Fin del programa
```



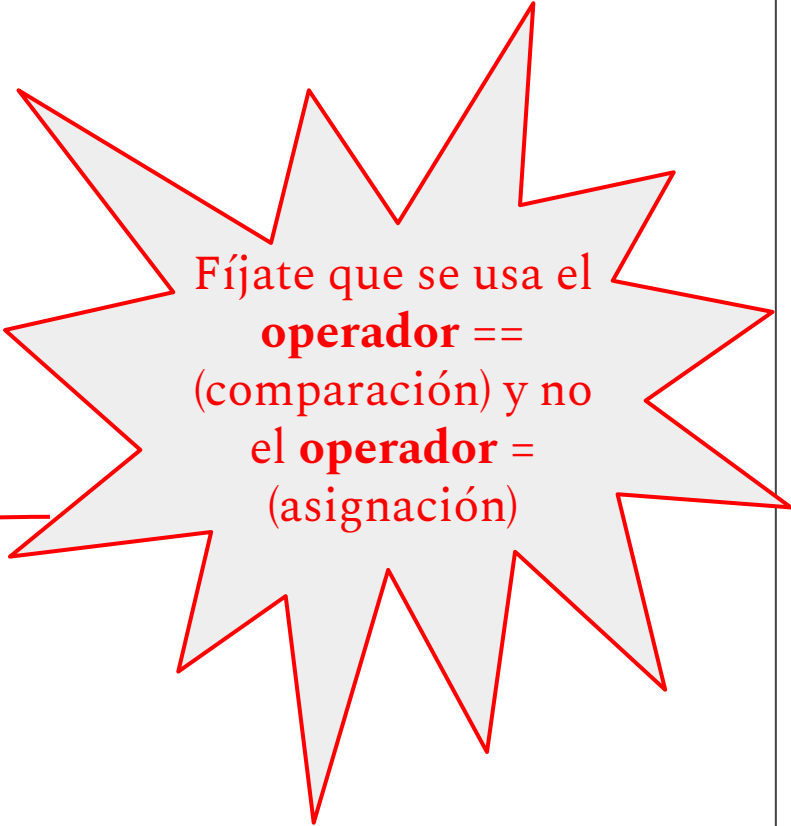
**No se realiza el
cálculo**

- En el segundo caso, al no cumplirse la condición (la expresión $a \neq 0$ devuelve un valor False), se **salta** el bloque de instrucciones incluido en el **if** y se continúa en la siguiente instrucción (`print('Fin del programa')`)

Sentencias condicionales (VI)

- Hemos corregido nuestro error pero deberíamos informar de dicha situación al usuario en lugar de finalizar abruptamente

```
'''  
ecu_1G_solver.py    ver.3  
Resuelve ecuaciones de 1er grado  
'''  
  
a = float(input('Introduce el coeficiente A: '))  
b = float(input('Introduce el coeficiente B: '))  
  
if a != 0:  
    x = -b/a  
    print('La solución es x = {0:.2f}'.format(x))  
  
# si a es 0 se informa al usuario  
if a == 0:  
    print('La ecuación no tiene solución')  
  
print('Fin del programa')
```



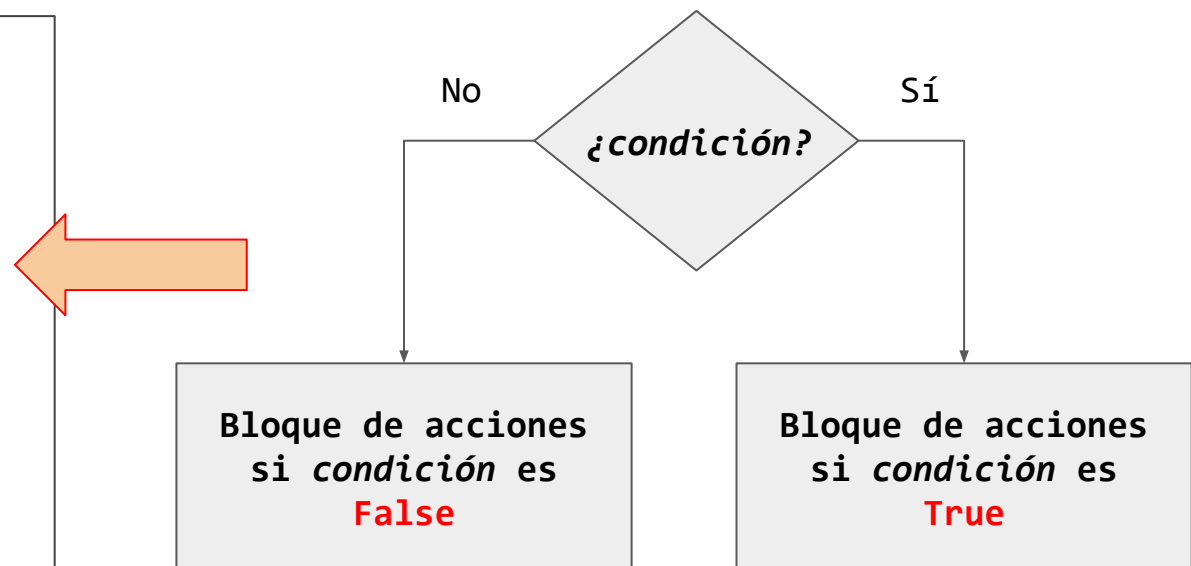
Fíjate que se usa el
operador ==
(comparación) y no
el **operador =**
(asignación)

Sentencias condicionales (VII)

La construcción *if - else*

- Para situaciones, como en el caso anterior, donde nos encontramos con que debemos realizar unas determinadas acciones en caso de que se verifique una condición y, otras acciones diferentes en el caso contrario, Python nos proporciona la construcción *if - else* que responde al siguiente formato:

```
if condición:  
    bloque de acciones si  
    condición es True  
else:  
    bloque de acciones si  
    condición es False
```



Sentencias condicionales (VII)

- Reescribamos nuestro programa de ecuaciones de 1^{er} grado...

```
'''
ecu_1G_solver.py    ver.4
Resuelve ecuaciones de 1er grado
'''

a = float(input('Introduce el coeficiente A: '))
b = float(input('Introduce el coeficiente B: '))

if a != 0:
    x = -b/a
    print('La solución es x = {0:.2f}'.format(x))
else:
    # si a es 0 se informa al usuario
    print('La ecuación no tiene solución')

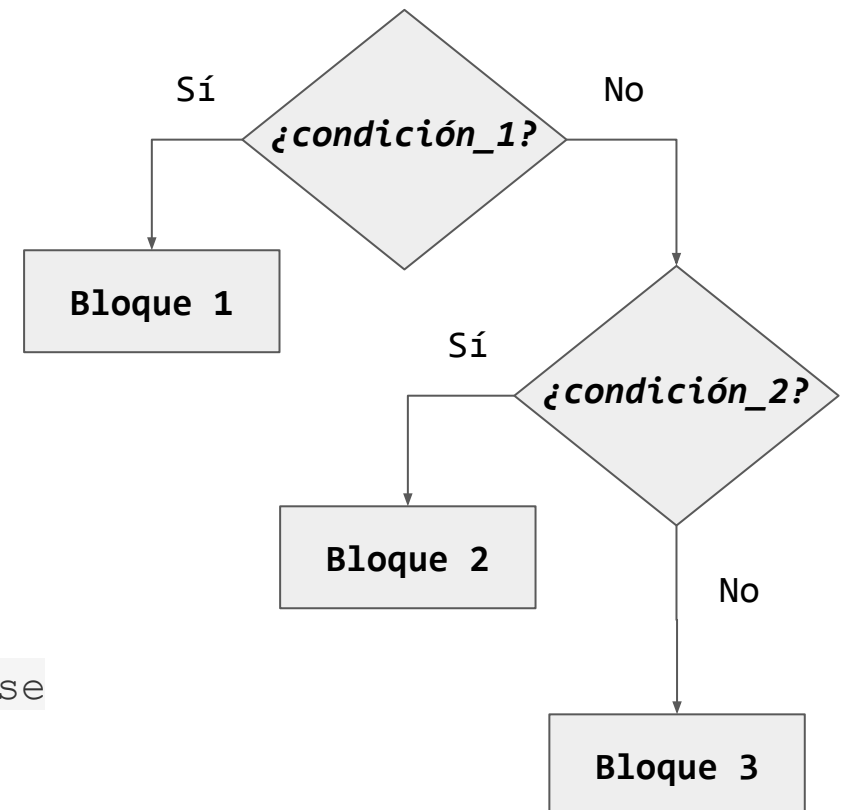
print('Fin del programa')
```

Sentencias condicionales (VII)

Condicionales anidados

- Pueden darse casos en los que necesitamos “insertar” condicionales dentro de otros (*anidamiento*), creando estructuras como la siguiente:

```
if condición_1 :  
    # Se ejecuta si condición_1 es True  
    bloque 1  
else :  
    # Se ejecuta si condición_1 es False  
    if condición_2 :  
        # Se ejecuta si condición_2 es True  
        bloque 2  
    else :  
        # Se ejecuta si ambas condiciones son False  
        bloque 3
```



Sentencias condicionales (VII)

- Volvamos al ejemplo. Queremos que, si a es 0, distinguir cuando b es 0 (infinitas soluciones) y b es distinto de 0 (no tiene solución)

```
'''
ecu_1G_solver.py    ver.5
Resuelve ecuaciones de 1er grado
'''

a = float(input('Introduce el coeficiente A: '))
b = float(input('Introduce el coeficiente B: '))

if a != 0:
    x = -b/a
    print('La solución es x = {0:.2f}'.format(x))
else:
    if b == 0:
        print('Indeterminado: la ecuación tiene infinitas soluciones')
    else:
        print('La ecuación no tiene solución')
print('Fin del programa')
```

Sentencias condicionales (VIII)

Condicionales anidados: *if - elif - else*

- Por último, para estos anidamientos de condicionales, Python nos ofrece una construcción del lenguaje más “*elegante*”, la sentencia *elif*, que surge de la **contracción** de *else* con el *if* posterior

```
if condición_1 :  
    # Se ejecuta si condición_1 es True  
    bloque 1  
elif condición_2 :  
    # Se ejecuta si las condiciones anteriores son False y condición_2 es True  
    bloque 2  
elif condición_3 :  
    # Se ejecuta si las condiciones anteriores son False y condición_3 es True  
    bloque 3  
else :  
    # Se ejecuta si todas las condiciones anteriores son False  
    bloque 4
```

Sentencias condicionales (IX)

- Así, la versión definitiva de nuestro ejemplo sería:

```
'''
ecu_1G_solver.py    ver.6
Resuelve ecuaciones de 1er grado
'''

a = float(input('Introduce el coeficiente A: '))
b = float(input('Introduce el coeficiente B: '))

if a != 0:
    x = -b/a
    print('La solución es x = {0:.2f}'.format(x))
elif b == 0:
    print('Indeterminado: la ecuación tiene infinitas soluciones')
else:
    print('La ecuación no tiene solución')

print('Fin del programa')
```

Indentación en Python

- Una nota final sobre la *indentación* o *sangrado* que usa Python para definir *bloques* de instrucciones. Si bien puede hacerse un tanto extraño al principio, especialmente a programadores que vienen de otros lenguajes como C o Java, donde se usa { } para definir dichos bloques, tiene como claro objetivo obligar y facilitar la *legibilidad* del código.
- Fíjate en el código del siguiente programa. Es nuestro ejemplo pero escrito en lenguaje C. Está un poco "ofuscado" a propósito (de hecho se podría haber escrito todo el programa en un par de líneas)

```
#include <stdio.h>
void main(){ float a,b,x;
printf("Coef A: "); scanf("%f", &a); printf("Coef B: "); scanf("%f", &b);
if(a!=0) {printf("La solución es x=%.2f", -b/a);} else {if(b==0)
{printf("Indeterminado");} else {printf("Sin solución");}}
printf("\nFin programa");}
```

- Guía de estilo de Guido van Rossum (<https://www.python.org/dev/peps/pep-0008/>)

Sentencias iterativas (I)

La sentencia *while*

- Los bucles *while* son los tipos más simples de bucle. En ellos se establece una condición y, mientras esa condición se cumpla (devuelve un valor **True**), el bloque de instrucciones asociado al bucle *while* continuará ejecutándose

- Su formato es:

```
while condición:  
    acción  
    acción  
    ...  
    acción
```

- *condición* será cualquier expresión evaluable que devuelva un valor booleano **True** o **False**

Sentencias iterativas (II)

- El siguiente ejemplo usa un bucle *while* para crear un contador. En cada **iteración** del bucle, se irán ejecutando cada una de las instrucciones del bloque *while*. Tras ejecutar la última, se volverá a **comprobar** la condición. **Si se cumple**, se realizará una nueva iteración. En caso contrario, se continuará con la instrucción siguiente al bloque *while*

```
from time import sleep
```

```
print('Se va a iniciar la cuenta atrás...')
```

```
sleep(3)
```

```
temp = 10
```

```
while temp>0:
```

```
    print(temp)
```

```
    sleep(1)
```

```
    temp -= 1
```

```
print('\aDESPEGUE!!!\n')
```

Al **principio** de cada **iteración**,
comprobamos el valor de **temp**

Si se cumplió la condición (**temp>0**),
ejecutamos las instrucciones del bucle

El proceso sigue
repitiéndose
hasta que deje
de cumplirse la
condición

Esta instrucción **sólo** se ejecutará una vez que se salga del bucle

Sentencias iterativas (III)

El bucle *for-in*

- Python dispone de otro tipo de bucle, el bucle *for-in*, que se puede leer como “*para cada elemento de la serie, hacer...*”
- Suele emplearse cuando conocemos de antemano el número exacto de iteraciones que queremos realizar o bien, cuando queremos recorrer secuencialmente una serie de elementos o valores

- Su formato es:

```
for variable in serie_de_valores:  
    acción  
    acción  
    ...  
    acción
```

- En cada *iteración* del bucle, *variable* irá tomando uno de los valores de *serie_de_valores*. Habrá tantas iteraciones como valores en la serie

Sentencias iterativas (IV)

- El siguiente ejemplo usa un bucle *for* para imprimir los nombres contenidos en una lista:

```
for nombre in ['Pedro', 'Juan', 'María']:
    print(';Hola ' + nombre + '!')
print('Bienvenidos al curso')
```

- El siguiente ejemplo usa la función *range*, que nos permite generar una secuencia de valores entre un valor inicial y otro final, para calcular el factorial de un número introducido por el usuario:

```
num = int(input('Introduce un número: '))

fact = 1
for i in range(1, num+1):
    fact *= i

print('Factorial de {}!={}'.format(num, fact))
```

range(1, num+1) genera una lista de valores desde **1** hasta **num**
Se realizará una iteración del bucle por cada valor de esa lista

Sentencias iterativas (V)

La función *range*

- La función *range* es lo que en Python se denomina un *generador*. Es decir, produce una secuencia de valores entre unos límites.
- Su formato es: `range([inicio,]final[,paso])`
 - inicio, (*por defecto*, 0) valor inicial del generador
 - final, establece el valor final del generador (**no se incluye**)
 - paso, (*por defecto*, 1) paso o salto entre los valores generados

```
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(4))
(0, 1, 2, 3)
>>> list(range(-3, 3))
[-3, -2, -1, 0, 1, 2]
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Sentencias iterativas (VI)

- En general, la sentencia *for* nos va a permitir recorrer cualquier tipo de objeto **secuenciable** (*iterable*), es decir, objetos que nos permiten acceder de forma secuencial a cada uno de sus elementos. Objetos de este tipo son las cadenas de caracteres, las listas, las tuplas, ...

```
>>> conjunto_1 = (1, 2, 3, 4, 5) # una tupla
>>> conjunto_2 = ['Tengo', '23', 'años'] # una lista
>>> acum = 0
>>> for val in conjunto_1:
    acum += val
>>> print('La suma de los valores de conjunto_1 es', acum)
La suma de los valores de conjunto_1 es 15
>>> for val in conjunto_2:
...     print(val, end=' ')
Teño 23 años
>>> for val in conjunto_2[2]:
    print('[' + val + ']', end=" ")
[a] [ñ] [o] [s]
```

Sentencias iterativas (VII)

break, continue y pass

- En ocasiones se darán casos en que precisemos alterar la ejecución normal dentro del bucle o abandonarlo. Las sentencias *break* y *continue* nos permiten modificar el flujo normal de ejecución de los bucles creados con *while* y *for*

break

- La sentencia *break* fuerza la salida del bucle en el que nos encontremos:

```
>>> while True:
...     val = input("Pulsa [S] para Salir ")
...     if val == 'S':
...         break
```

```
Pulsa [S] para Salir: a
```

```
Pulsa [S] para Salir: S
```

```
>>>
```

Sentencias iterativas (VIII)

- El siguiente ejemplo, que calcula los números primos entre 100 y 200, muestra el uso de *break* para evitar cálculos innecesarios. Para cada uno de estos números, el programa buscará la existencia de algún divisor. En caso de encontrar alguno, ya sabemos que el número no es primo y podemos pasar al siguiente.

```
print('Los números primos entre 100 y 200 son:')

for numero in range(100, 201):
    primo = True
    for divisor in range(2, numero):
        if numero%divisor == 0:
            # no es primo -> saltar al siguiente
            primo = False
            break

    # si numero es primo, lo imprimimos
    if primo:
        print(numero, end=' ')
```

FÍJATE

Podemos anidar bucles, condicionales,... *break* y *continue* sólo afectarán al bucle más interno donde se encuentren

IMPORTANTE

Vigila el *sangrado* para que Python identifique correctamente cada bloque


Sentencias iterativas (IX)

continue

- La sentencia *continue* dentro de un bucle, provoca que se salte al inicio de la siguiente iteración sin ejecutar el resto de instrucciones del bucle. Es decir, no salimos del bucle (como con *break*), si no que saltamos al inicio del mismo (y si se cumple la condición se ejecutará la siguiente iteración)

```
# Imprime número impares entre 1 y 10
print('Números impares menores que 10: ', end='')
for num in range(1, 11):
    if num%2 == 0:
        continue
    print(num, end=' ')
```

Si **num** es **par**, salta al principio del bucle (*continue*) y la sentencia *print* no se ejecuta



output:

```
Números impares menores que 10: 1 3 5 7 9
```

Sentencias iterativas (y X)

pass

- La sentencia *pass* representa una operación **nula**, es decir, no ejecuta **nada!!**
- Si bien su existencia puede parecer un tanto absurda, se usa cuando una sentencia es necesaria sintácticamente, pero no necesitamos ejecutar ningún código.
- Es habitual encontrarla en las fases iniciales de nuestro programa cuando tenemos la estructura del código pero aún está sin implementar determinada funcionalidad

```
# recorro el bucle
for val in lista_de_valores:
    if val==valor_erroneo:
        # sé que tengo que hacer algo aquí pero aún no sé qué hacer
        pass
```