

# Programación Estructurada

El concepto de programa

# Introducción

- La programación estructurada proporciona una serie de técnicas que facilitan la tarea de programar, ya que reducen el tiempo para escribir programas.
- Concepto clave: algoritmo
  - Definición
  - Fases del proceso de programación
- Técnicas representación de algoritmos: diagramas de flujo de datos

# El concepto de Algoritmo (I)

- Por algoritmo se entiende la descripción precisa de una sucesión ordenada de instrucciones que permitan resolver un problema.
- El programa se compone de una serie de instrucciones (algoritmo) que operan sobre unos datos (estructuras de datos).

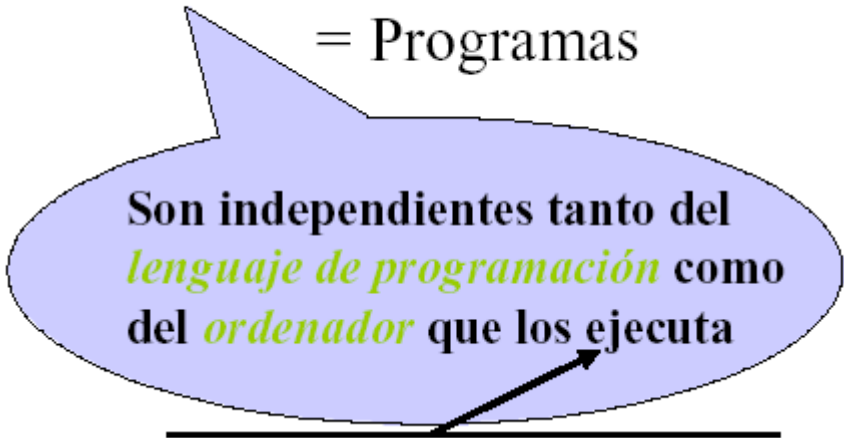
Algoritmos + Estructuras de Datos  
= Programas

**Descripción precisa de una sucesión  
ordenada de instrucciones no ambiguas  
que permiten resolver un problema en  
un número finito de pasos**

# Concepto de algoritmos (II)

- Son independientes del lenguaje de programación que se quiera emplear
- Son independientes del ordenador en donde se ejecute el programa

Algoritmos + Estructuras de Datos  
= Programas



Son independientes tanto del  
*lenguaje de programación* como  
del *ordenador* que los ejecuta

# Fases de proceso de programación



1. Comprensión del problema: establecer requerimientos
2. Plantear la lógica (formalizar)
3. Codificar el programa en un lenguaje de programación
4. Compilar el programa fuente (crear ejecutable) (o interpretar)
5. Probar el problema/programa
6. Evaluar la solución
7. Corregir posibles problemas
8. Utilizar el programa.

# 1. Comprensión del Problema

Ejemplo:

“En una empresa necesitan la lista de los clientes que no han pagado en tres meses”

Cuestiones:

- ⇒ ¿no han realizado un pago a tres meses?
- ⇒ ¿no han pagado nada en los tres últimos meses?
- ⇒ ¿y si no han pagado en cuatros meses, o en ...?
- ⇒ ...

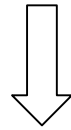
## 2. Plantear la lógica

- ⇒ Se plantean los pasos del programa (qué pasos incluir y en qué orden: algoritmo)
- ⇒ Se decide la herramienta a utilizar (pseudocódigo, diagrama de flujo de datos,...)
- ⇒ El programador **no se preocupa de la sintaxis del lenguaje de programación** que se empleará en las últimas fases.
- ⇒ Se estudiarán los datos de entrada necesarios y disponibles, así como los datos que se obtendrán como salida



### 3. Codificar el programa (teclear)

Una vez desarrollada la lógica del programa



Escribir el programa en uno de los más de 400  
lenguajes de programación que existen

**¿Qué es más complicado,  
plantear la lógica o  
codificar el problema?**

**¿Cómo se gana más  
dinero?**



## 4. Compilar/Interpretar el programa fuente

- Un programa traductor (compilador o intérprete): Cambia el lenguaje de alto nivel con el que escribimos a lenguaje máquina de bajo nivel que el ordenador entiende (ejecutables)
- Los compiladores no siempre saben qué es lo que se les quiere decir, ni saben cuál sería la corrección adecuada, pero sí saben cuando algo está mal en la sintaxis.

	<b>Ventajas</b>	<b>Desventajas</b>
<b>Compilados</b>	<ul style="list-style-type: none"> <li>➤ A excepción de Java, se ejecutan es más rápido</li> <li>➤ No necesitan tener el interprete instalado</li> <li>➤ Ocultamiento del código fuente</li> </ul>	<ul style="list-style-type: none"> <li>➤ Suelen requerir mayor atención para mantener el código</li> <li>➤ A excepción de Java, se compromete la portabilidad de binarios</li> </ul>
<b>Interpretados</b>	<ul style="list-style-type: none"> <li>➤ Código más fácil de mantener</li> <li>➤ Mayor portabilidad del código</li> </ul>	<ul style="list-style-type: none"> <li>➤ Relativa lentitud de ejecución</li> <li>➤ No ocultamiento del código fuente</li> </ul>

# 5. Probar el problema/programa

Un programa libre de errores de sintaxis no está necesariamente exento de **errores lógicos**

¿Errores sintácticos?  
¿Errores lógicos?

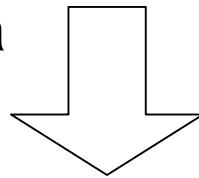


## 6. Evaluar la solución.

Una vez que se ha terminado de escribir un algoritmo es necesario comprobar que realiza las tareas para las que ha sido diseñado y produce el resultado correcto y esperado

Para comprobar el algoritmo:

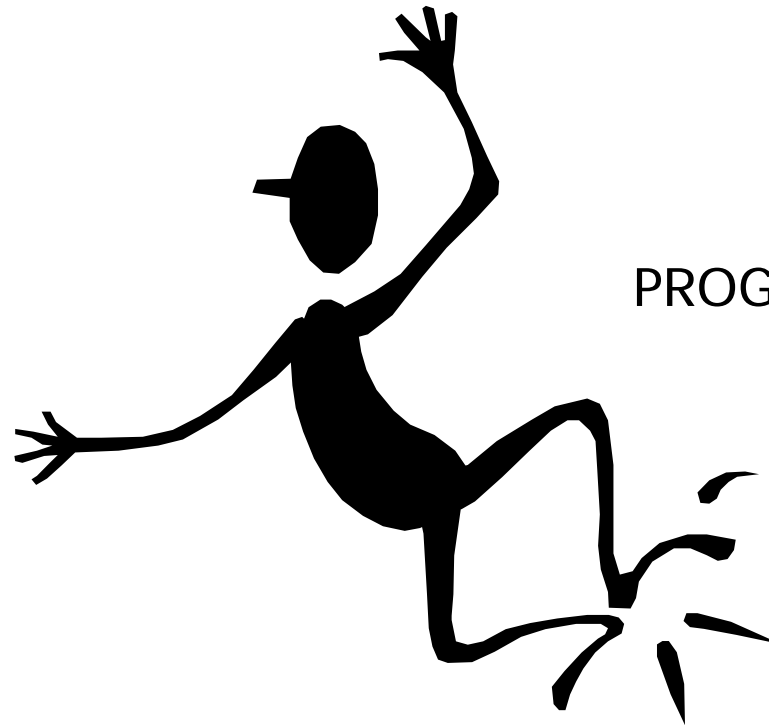
- EJECUCIÓN MANUAL mediante datos significativos
- UTILIZACIÓN DE PAPEL para anotar las modificaciones de las distintas fases del programa



## Fase 7: Corrección de Errores

## 8. Utilización del programa

- Una vez finalizadas todas las etapas de programación hay que llevar el programa al usuario final para el que se realizó.
- La conversión para utilizar un nuevo programa puede llevar meses no sólo por la instalación, acondicionamiento de las máquinas, sino por la preparación de los propios usuarios.

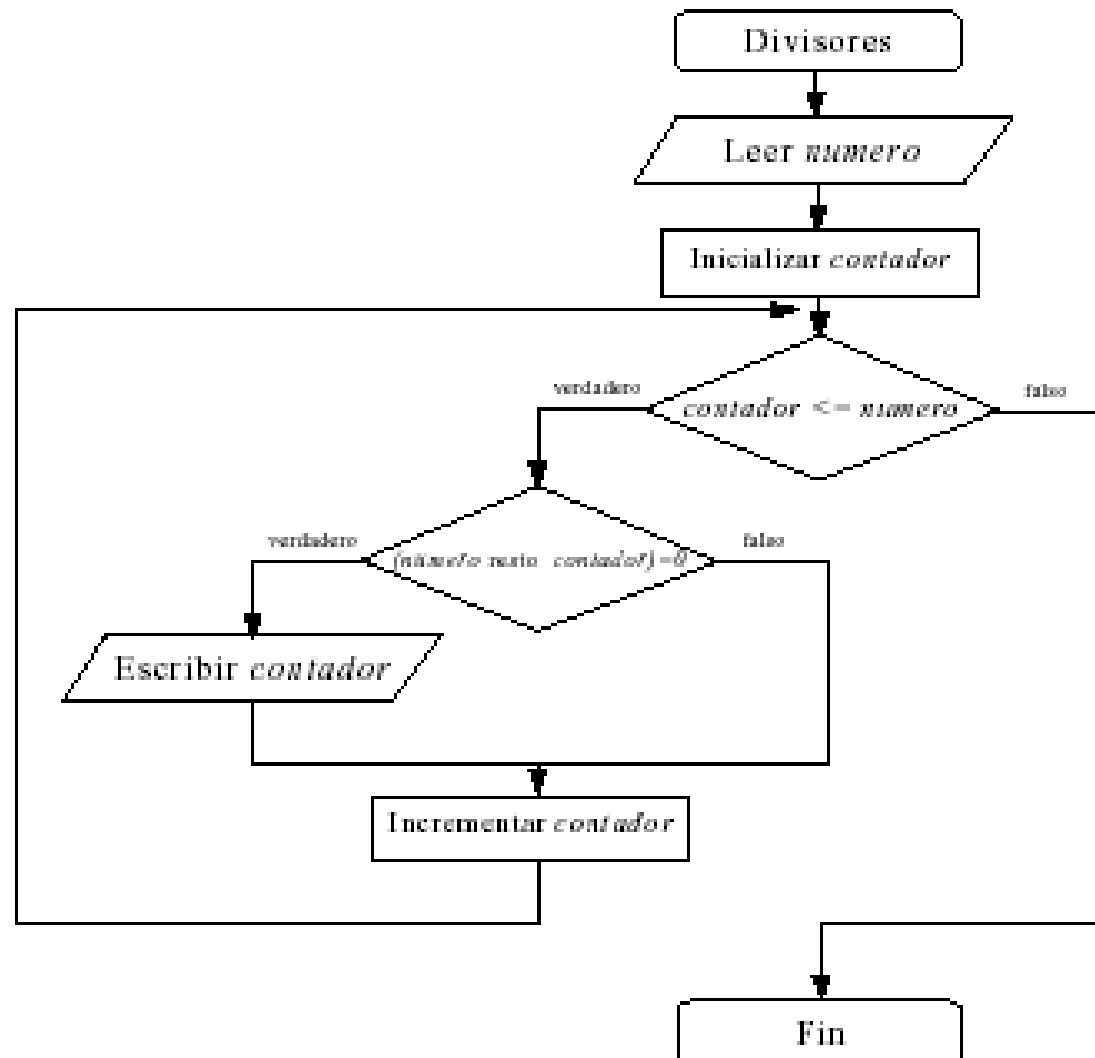


PROGRAMA FUNCIONANDO

# Técnicas de Representación de Algoritmos

- Ayudan a visualizar la lógica del programa de forma esquemática
- Una de estas técnicas son: Diagramas de Flujo de Datos
- Realizan una representación gráfica de los algoritmos resaltando el flujo de control del algoritmo.
- Ejemplo:

# Diagramas de Flujo de Datos (I)



# Diagrama de Flujo de Datos: Elementos

Inicio/Fin del algoritmo



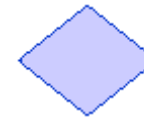
Operaciones



Entrada / Salida



Comparaciones



Conector



Línea de Flujo

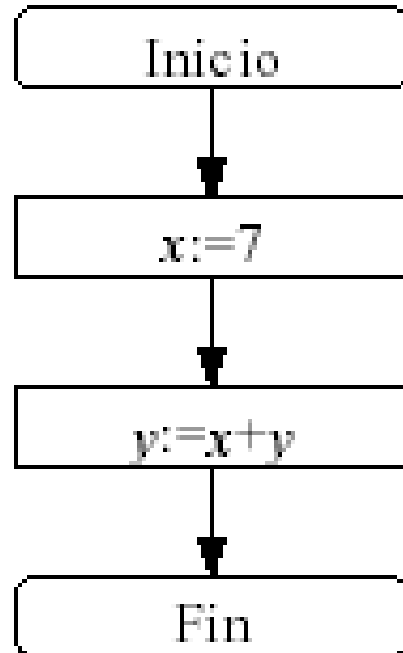




# Estructuras algorítmicas básicas (I)

## Secuencia

**SECUENCIA** : las acciones se efectúan unas a continuación de las otras, de manera consecutiva. Ejemplo:



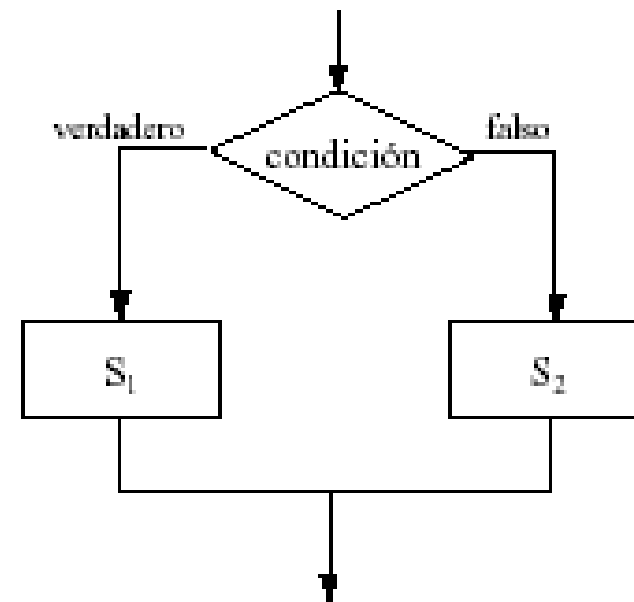
# Estructuras algorítmicas básicas (II)

## Selectiva

### Diagrama de Flujo de Datos

#### SELECCIÓN :

La estructura alternativa permite tomar decisiones entre distintas posibilidades en función de una condición.



# Estructuras algorítmicas básicas (II)

## Selectiva

## Equivalencia en Python

- Diversas estructuras:

- `if` , `if-else`

- La condición es una expresión que se construye haciéndose servir de **operadores lógicos y relacionales**.

# Expresiones lógicas y relacionales (I)

- Sirven para escribir las condiciones: básicas en sentencias condicionales e iterativas
- Operadores:

OPERADORES RELACIONES		OPERADORES LÓGICOS	
Menor que	<	Conjunción ó <b>Y</b> lógico	and
Mayor que	>	Disyunción ó <b>O</b> lógico	or
Menor o igual que	<=	Negación ó <b>NO</b> lógico	not
Mayor o igual que	>=		
Igual que	==		
Distinto que	!=		

# Expresiones lógicas y relacionales (II)

## NOTAS:

### ➤ Resultado lógico:

- falso  $\Leftrightarrow$  cero (el número cero)
- cierto  $\Leftrightarrow$  distinto de cero

➤ No confundir asignación (=) con comparación (==):

## Ejemplos:

$3 \leq 24 \hat{=}$  cierto / Ejemplos:  $23 == 22 \Leftrightarrow$  falso

$(5 \leq 0) \text{ or } !(x > x - 1) \Leftrightarrow$  falso

$4 \text{ and } 0 \Leftrightarrow$  falso

$(8 == 4 * 2) \text{ and } (5 > 2) \Leftrightarrow$  cierto

$4 > 8 \Leftrightarrow$  falso

$!(4 > 1) \Leftrightarrow$  falso

$(x < x - 8) \text{ or } 4 \Leftrightarrow$  falso

$2 \text{ and } (4 > 9) \Leftrightarrow$  falso

$4 != 5 \Leftrightarrow$  cierto

# Estructura selectiva

## Python: `if`

- Evaluación de la condición (expresión entera)
- Si la **condición** es cierta, entonces se ejecuta la sentencia o grupo de sentencias (`sentencia_1, ...`)
- Si la expresión es falsa, se ignora la sentencia o grupo de sentencias, y se ejecuta la siguiente sentencia en orden secuencial.

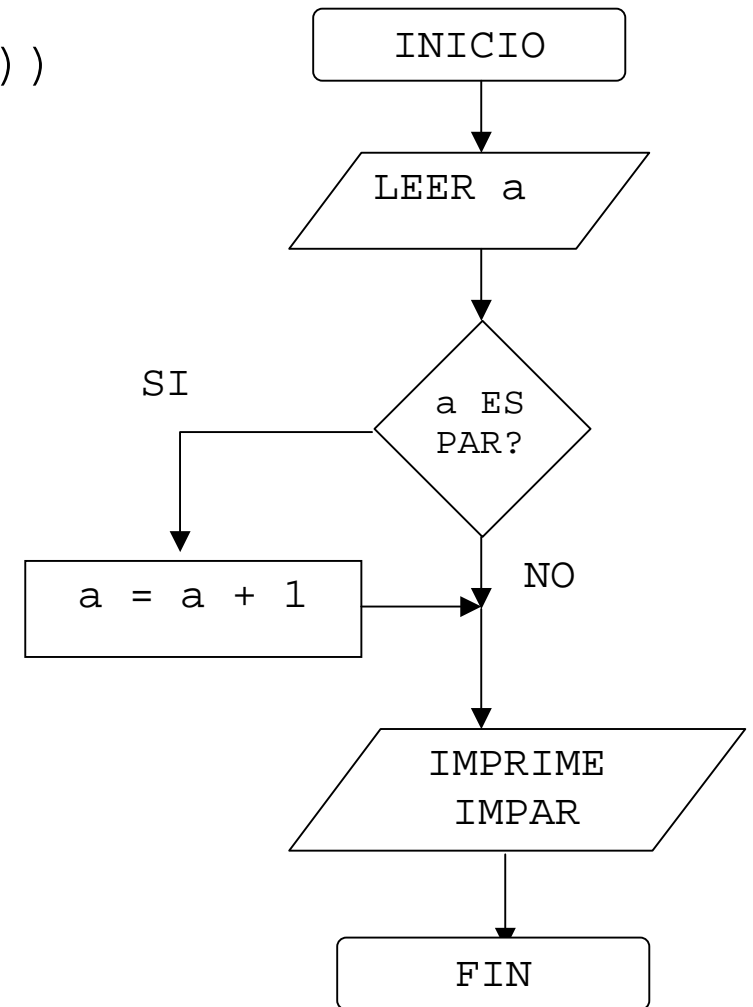
```
if condición :  
    sentencia_1  
sentencia_fuera
```

```
if condición :  
    sentencia_1  
    sentencia_2  
    ...  
    sentencia_N  
sentencia_fuera
```

# Estructura Selectiva

## Python: `if` (Ejemplo)

```
a=int(raw_input("Introduzca un entero"))
if (a % 2 )== 0 :
    a = a + 1
print "Ahora es impar ",a
```



# Estructura Selectiva

## Python: `if-else`

- Generalización de la anterior
- Evaluación de la condición (expresión entera) Si la expresión es **cierta**, entonces se ejecuta el **primer grupo de sentencias**, y si es **falsa**, entonces se ejecuta el **segundo grupo de sentencias**
- La ejecución continua en la siguiente sentencia en orden secuencial

```
if condición :  
    sentencia_1  
else :  
    sentencia_2  
sentencia_fuera
```

```
if condición) :  
    grupo_de_sentencias_1  
else :  
    grupo_de_sentencias_2
```

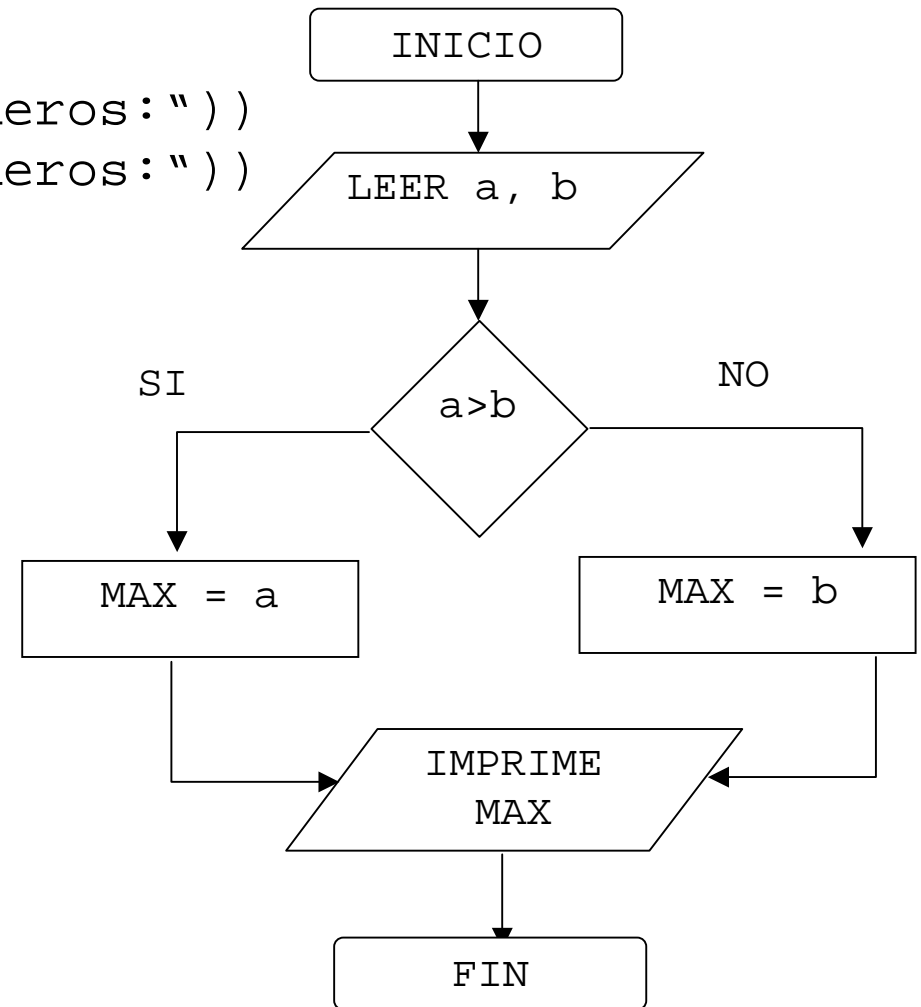


# Estructura Selectiva

## Python: `if-else` (Ejemplo)

```
a=int(raw_input("Introduce dos números:"))  
b=int(raw_input("Introduce dos números:"))
```

```
if a > b :  
    max = a  
else :  
    max = b  
print "El máximo es: ",max
```



# if-else: Otro ejemplo más complejo

```
hora=int(raw_input("Introduce la hora: "))
if (hora >= 0) and (hora < 12) :
    print "Buenos días"
else :
    if (hora >= 12) and (hora < 18) :
        print "Buenas tardes"
    else :
        if (hora >= 18) and (hora < 24) :
            print "Buenas noches"
        else
            print "Hora no válida"
print "Fin del programa"
```

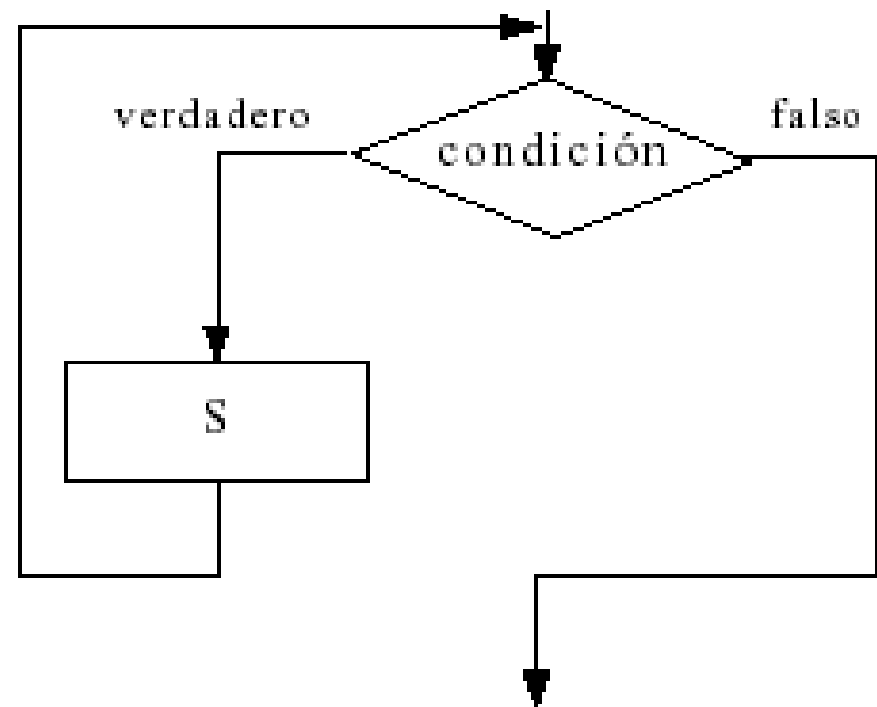
# Estructuras algorítmicas básicas (II)

## Selectiva

### Diagrama de Flujo de Datos

**ITERACIÓN :** (iteración o bucle)  
repetición de una instrucción o instrucciones en función de una condición de continuación o de finalización.

La repetición se representa mediante la flecha (el flujo) que vuelve atrás.



# Estructuras algorítmicas básicas (II)

## Iterativa

## Equivalencia en Python

- Diversas estructuras:

`while` o `for`

- Las tres estructuras se leen: *mientras se cumpla la condición* hacer una determinada acción. A dichas acciones se les llama **cuerpo del bucle**.
- La condición es una expresión que se construye haciéndose servir de **operadores lógicos y relacionales** (de la misma forma que con el if)
- En el momento en el que la condición resulte falsa, el bucle finalizará. **Es muy importante controlar la condición de terminación del bucle.**

# Estructura iterativa - Python: `while`

- La condición se evalúa al principio del bucle
- El cuerpo del bucle se puede ejecutar 0 veces o hasta N veces
- Cuidado con crear bucles infinitos ⇨ **condición de terminación**
- Las variables involucradas en la condición deben tener un valor “controlado” al llegar al inicio del bucle ⇨ **inicializar variables siempre**
- Si la expresión es falsa, se ignora la sentencia o grupo de sentencias, y se ejecuta la siguiente sentencia en orden secuencial.

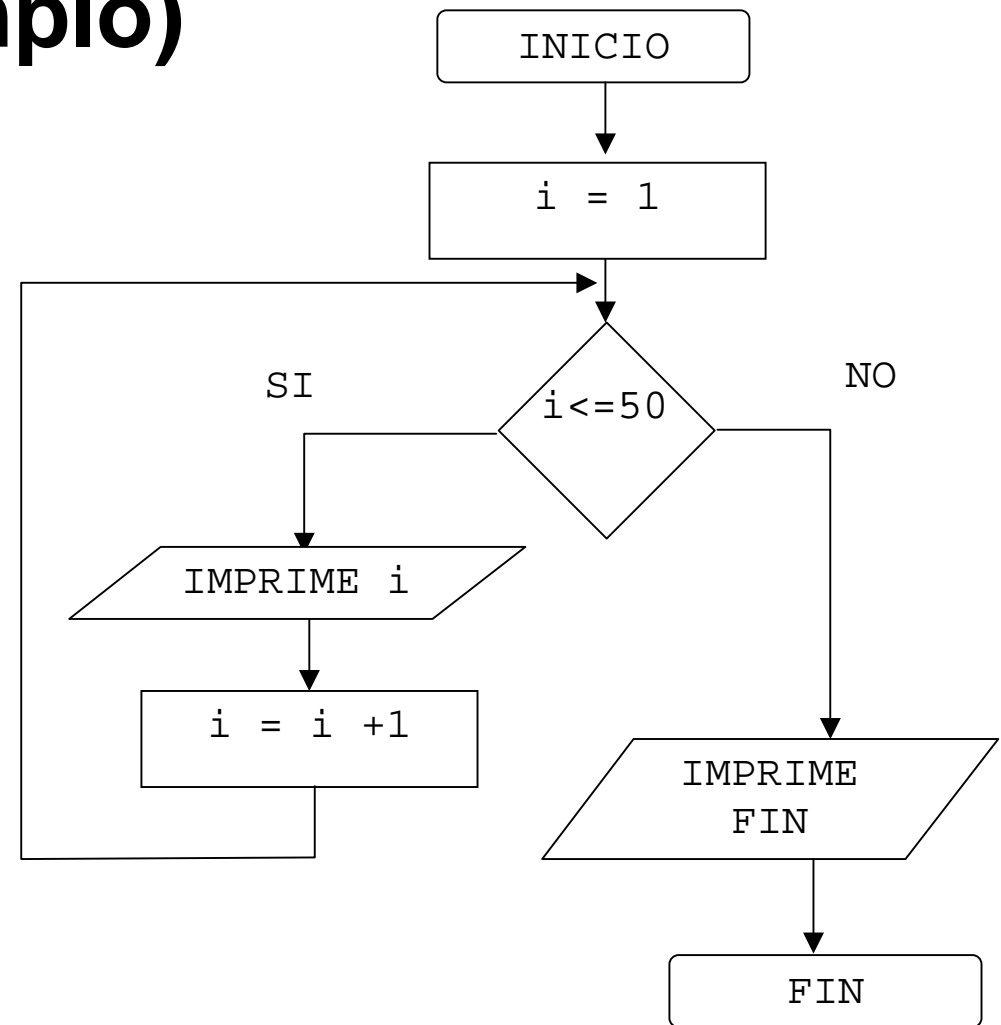
```
while condición :  
    Sentencia  
Sentencia_fuera
```

```
While condición :  
    Sentencia_1  
    Sentencia_2  
    ...  
  
Sentencia_fuera
```

# Estructura Iterativa

## Python: `while` (Ejemplo)

```
i=1
while i<= 50 :
    print i
    i = i + 1
print "Fin del programa"
```



# Características deseables de un programa

- Corrección
- Claridad
- Eficiencia
- Sencillez
- Modularidad
- Generalidad

# INTRODUCCIÓN A LENGUAJE PYTHON

Direcciones de interés:

---

<http://www.mclibre.org/consultar/python/>

<http://usuarios.lycos.es/arturosa/>

<http://es.wikipedia.org/wiki/Python>



# Introducción a Python

- Python fue creado por Guido van Rossum  
(<http://www.python.org/~guido/>)
- Da este nombre al lenguaje inspirado por el popular grupo cómico británico Monty Python
- Guido creó Python durante unas vacaciones de navidad en las que (al parecer) se estaba aburriendo

# Características de Python I

- Python es un lenguaje de programación interpretado, interactivo y orientado a objetos
- Python combina una potencia notable con una sintaxis muy clara: muy legible y elegante
  - Imposible escribir código ofuscado
- Python es multiplataforma: existen versiones para muchas variantes de UNIX, para Windows, DOS, OS/2, Mac, Amiga, etc
- Python está registrado, pero puedes utilizarlo y distribuirlo libremente, incluso para fines comerciales
- Simple y poderoso

# Características de Python II

- Minimalista: todo aquello innecesario no hay que escribirlo (;, {, }, '\n')
  - Muy denso: poco código hace mucho
  - Soporta objetos y estructuras de datos de alto nivel: strings, listas, diccionarios, etc.
  - Múltiples niveles de organizar código: funciones, clases, módulos, y paquetes
- No tienes que declarar constantes y variables antes de utilizarlas
- No requiere paso de compilación/linkage: La primera vez que se ejecuta un script de Python se compila y genera bytecode que es luego interpretado
- Alta velocidad de desarrollo y buen rendimiento
- De propósito general: puedes hacer en Python todo lo que puedes hacer con C# o Java, o más

# ¿Para qué [no] es útil?

Python no es el lenguaje perfecto, no es bueno para:

- Programación de bajo nivel (system-programming), como programación de drivers y kernels
- Aplicaciones que requieren alta capacidad de computo: no hay nada mejor para este tipo de aplicaciones que el viejo C

Python es ideal:

- Como lenguaje "pegamento" para combinar varios componentes juntos
- Para llevar a cabo prototipos de sistema
- Para desarrollo web y de sistemas distribuidos
- Para el desarrollo de tareas científicas, en los que hay que simular y prototipar rápidamente

# Sentencias y bloques

- Las sentencias acaban en nueva línea, no en ;
- Los bloques son indicados por tabulación que sigue a una sentencia acabada en ':'. E.j. (bloque.py):

```
# comentarios de línea se indican con carácter '#'
name = "Diego1" # asignación de valor a variable
if name == "Diego":
    print "Aupa Diego"
else:
    print "¿Quién eres?"
    print "¡No eres Diego!"
```

# Variables /Identificadores

## Qué es una variable?

- En Informática, una variable es "algo" en lo que puedes almacenar información para su uso posterior. En Python, una variable puede almacenar un número, una letra, un conjunto de números o de letras o incluso conjuntos de conjuntos.
- Los identificadores sirven para nombrar variables, funciones y módulos
  - Deben empezar con un carácter no numérico y contener letras, números y '\_'
  - Python es case sensitive (distingue entre mayúsculas y minúsculas)

# Tipos de datos

- Numéricos (integer, long integer, floating-point,)

```
>>> x = 4
>>> int(x)
4
>>> long(x)
4L
>>> float(x)
4.0
```

- Cadenas: strings, pueden ir delimitados por comillas simples o dobles.
- lists (listas).
- En python se declaran y asignan cuando se usan (y sin declarar el tipo).

Cuidado con usar una variable antes de asignarle un valor.

## Ejemplos:

```
1+2*3 # El resultado es 7
(1+2)*3 # El resultado es 9
variable_autor = "arturo suelves"
variable_iniciales=variable_autor[0]+variable_autor[7]
variable_nombre = variable_autor[0:6]
variable_lista=["Hola","a","todo","el","mundo"]
elemento_de_lista=variable_lista[2]
elementos_de_lista=variable_lista[0:2]
variable_lista[2] = "casi todo"
variable_lista=["Hola","a","casi todo","el","mundo"]
```



# Salida por pantalla (print)

- Aunque en IDLE puedas ver el valor de una variable escribiendo simplemente el nombre de la variable, dentro de un programa tienes que utilizar la orden ***print***.
- Puedes intercalar texto y variables en una misma orden, separándolas con comas, como muestra el siguiente ejemplo:

```
a, saludo = 5, 'Hola'
print 'a contiene el valor' , a, 'y saludo el valor', saludo
```

- Si escribes una coma al final de una orden print, el siguiente texto se escribirá en la misma línea, como muestra el siguiente ejemplo:

```
corto, medio, largo = 28, 30, 31
print 'Hay siete meses que tienen' , largo , 'días.'
print 'Hay cuatro meses que tienen' , medio , 'días ',
print 'y uno que tiene' , corto , 'o', corto+1, 'días '
```

# Entrada por teclado (raw\_input)

- La función ***raw\_input()*** permite que un programa almacene en una variable lo que escribas en el teclado. Al llegar a la función, el programa se detiene esperando que escribas algo y pulses la tecla Intro. Prueba el siguiente ejemplo.

```
print '¿Cómo te llamas?'  
nombre = raw_input()  
print 'Me alegro de conocerte,' , nombre
```

- Las dos primeras líneas se pueden comprimir en una, escribiendo la cadena como argumento de la función `raw_input()`:
- Por defecto, la función `raw_input()` convierte la entrada en una cadena. Si quieres que Python interprete la entrada como un número entero, debes utilizar la función `int()`, para número decimal la función `float()`

```
cantidad = float(raw_input("Dime cantidad en euros"))  
print cantidad, 'euros son' , cantidad * 166.386, 'pesetas'
```

# Módulos

- *import* hace que un módulo y su contenido sean disponibles para su uso.

- Algunas formas de uso son:

`import test`

Importa modulo test. Referir a x en test con "test.x".

`from test import x`

Importa x de test. Referir a x en test con "x".

`from test import *`

Importa todos los objetos de test. Referir a x en test con "x".

`import test as theTest`

Importa test; lo hace disponible como theTest. Referir a objeto x como "theTest.x".

Ejemplo: `import math`

# Casos de éxito de Python

- BitTorrent (<http://bitconjurer.org/BitTorrent/>), sistema P2P que ofrece mayor rendimiento que eMule
- Google usa Python internamente, lo mismo que Yahoo para su sitio para grupos
- Más historias de éxito de Python en: <http://pbf.strakt.com/> success