

# U2.1. Fundamentos de programación en Python



- Programar es:
  - Automatizar procesos o tareas (p. ej. el sistema de semáforos de una ciudad)
  - Resolver problemas (p. ej. encontrar el camino más corto entre dos lugares)
  - Facilitar acciones (p. ej. firmar documentos electrónicamente)
  - Crear (p. ej. una aplicación que pone nuestra cara en una escena de ficción)
- Programar es una **actividad multidisciplinar** en la que se combina el arte, la creatividad, la ingeniería y las matemáticas, entre otras disciplinas, que se unen mediante la **informática**.

- Programar es crear **programas**, es decir, **secuencias de órdenes o instrucciones** que le indican a un dispositivo electrónico o informático, mediante un **lenguaje de programación**, qué debe hacer y en qué orden, sin ambigüedades y con una finalidad concreta.
- Al programar, implementamos **algoritmos**, es decir, métodos para la resolución de problemas. Los algoritmos han de ser:
  - Precisos: Sin posibles ambigüedades o casos sin contemplar
  - Ordenados: Instrucciones dispuestas en un orden de realización
  - Finitos: Tienen un final, es decir, un número finito de pasos



- **Lenguaje:** Conjunto de símbolos (alfabeto) que se combinan entre sí formando palabras (léxico) para expresar un significado (semántica) bajo unas reglas u orden correcto (sintaxis).
- **Lenguaje de programación:** Lenguaje para crear programas informáticos, es decir, para implementar algoritmos sobre un dispositivo electrónico o informático. Un programa se escribirá como una secuencia de frases (instrucciones) del lenguaje de programación.



- Lenguaje máquina
  - Directamente 0 / 1
- Lenguajes de bajo nivel: se representan los 0 y 1 con un lenguaje mnemónico. Ej: lenguaje ensamblador
- Lenguaje de alto nivel: son lenguajes más próximos al humano. Ej: lenguaje Java, PHP, Python, C++

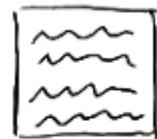
```
10100001 10111100 10010011 00000100
00001000 00000011 00000101 11000000
10010011 00000100 00001000 10100011
11000000 10010100 00000100 00001000
```

Push	a	; a → pila
Push	b	; b → pila
Load	(c),R1	; c → R1
Mult	(S),R1	; b*c → R1
Store	R1,R2	; R1 → R2
Add	(S),R1	; a+b*c → R1
Store	R1,(x)	; R1 → x
Add	#3,R2	; 3+b*c → R2
Store	R2,(y)	; R2 → y

```
# Este programa saluda y pregunta por mi nombre
print('Hola, ¿Cómo te llamas')
miNombre = input()
print('Es un placer conocerte, ' + miNombre)
```

- Se necesita de un proceso de **traducción de un programa en alto o bajo nivel a lenguaje máquina**. Esta traducción se puede hacer:
  - Antes de ejecutar el programa (**compilación**). Un programa **compilador** traduce el código fuente del programa, escrito por el programador, a un fichero ejecutable compuesto de 0s y 1s (p. ej. un archivo .exe en sistemas operativos Windows).
    - **C** y **C++** son, fundamentalmente, **lenguajes compilados**
  - Mientras se ejecuta el programa (**interpretación**). Un programa **intérprete** traduce el código fuente del programa, escrito por el programador, en tiempo real e instrucción a instrucción, leyendo una instrucción, convirtiéndola a 0s y 1s y ejecutándola, leyendo una instrucción...
    - **PHP** y **Python** son, fundamentalmente, **lenguajes interpretados**

Source code:  
hello.c



→ COMPILER →

Machine code:

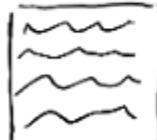
1	1	0	1	0
1	0	1	1	1
1	0	0	1	1

Program (also  
called binary,  
executable ...)

→ result  
run the  
program



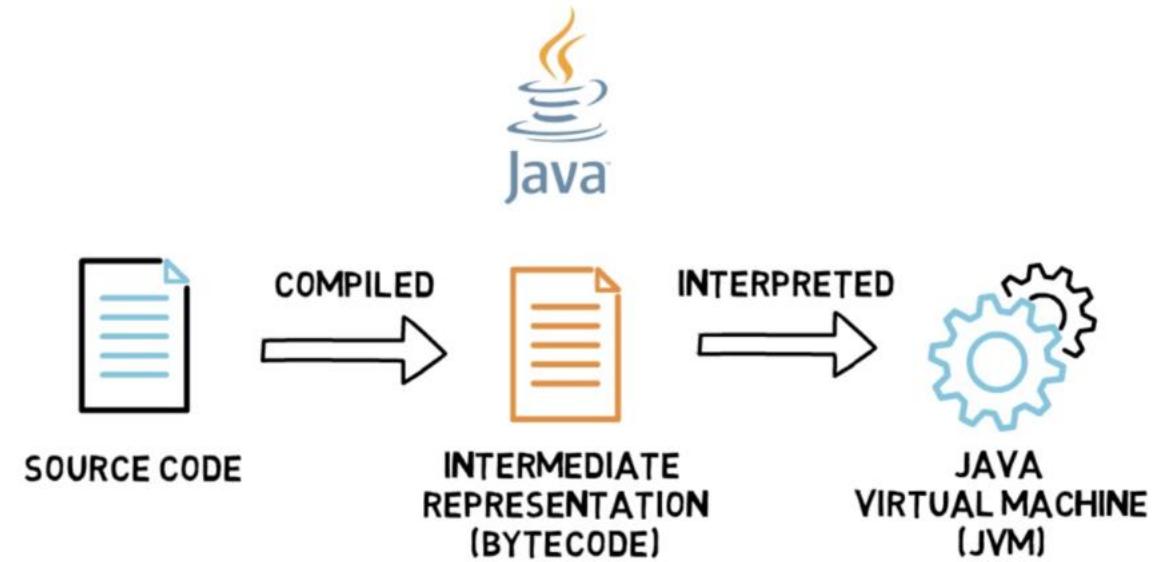
Source code:  
hello.py



→ INTERPRETER → result



- También existen fórmulas intermedias entre la compilación y la interpretación (p. ej. Java):



- De hecho, muchos de los lenguajes interpretados actuales (p. ej. Python) tienen también fases de compilación para acelerar su ejecución.

- A la hora de programar, existen diferentes maneras o enfoques para crear un programa. Son diferentes paradigmas de programación.
- Un **paradigma de programación** es un modelo básico para el diseño y la implementación de programas.
- Cada modelo tiene sus ventajas e inconvenientes y podrá ser más o menos apropiado para un caso o problema a resolver concreto.

- **Estructurado:**

Nació para solventar los problemas y limitaciones de la programación convencional (GOTO). Permite utilizar **estructuras** que facilitan la **modificación, reutilización y lectura del código**.

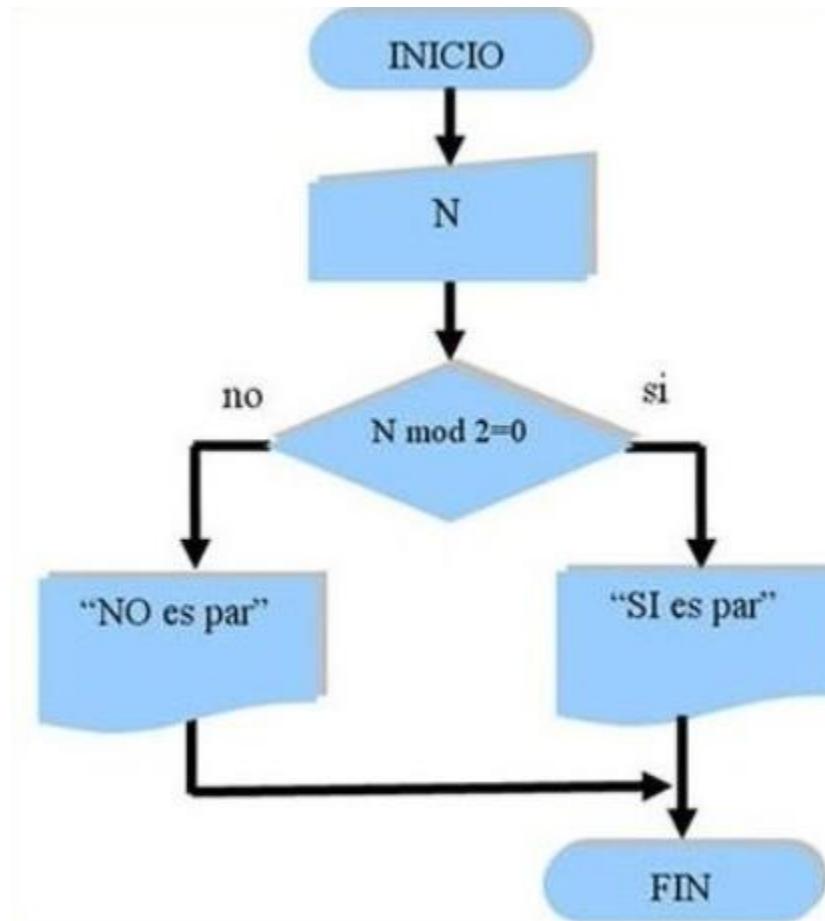
Los lenguajes de programación estructurada se conocen como lenguajes de arriba hacia abajo. La mayoría de los primeros lenguajes de programación pertenecen a este paradigma.

- **Orientado a objetos:**

En este paradigma, todas las entidades del mundo real están representadas por clases. Los objetos son instancias de clases, por lo que cada objeto encapsula un estado y un comportamiento. Los objetos interactúan entre sí a través de mensajes.



- Al analizar y representar cómo resolver un problema mediante un algoritmo en un programa informático, disponemos de varias herramientas o técnicas, complementarias entre sí:
  - **Diagramas de flujo:** Esta técnica utiliza símbolos gráficos para la representación del algoritmo
  - **Pseudocódigo:** Esta técnica se basa en el uso de palabras clave y acciones en lenguaje natural
  - **Tablas de decisión:** Esta técnica representa en una tabla las posibles condiciones del problema con sus respectivas acciones



- Ejemplo: Diseñar un programa en pseudocódigo que solicite como entrada al usuario la base y la altura de un rectángulo y devuelva como salida su área.

## PROGRAMA calcularAreaRectangulo

### DATOS

Entrada: base, altura; Salida: area

### ALGORITMO

ESCRIBIR ("Introduce la base y la altura del rectángulo: ")

LEER (base, altura)

area = base \* altura

ESCRIBIR ("El área del rectángulo es: " . area)

### FIN

- Ejemplo: Diseñar un programa que aplique descuentos a compras del 0%, 5% y 10% según si el monto de la compra es pequeño, mediano o grande y de si el cliente es frecuente o no.

CONDICIONES	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
<b>Monto de la Compra Pequeña</b>	S	S	S	S	S	S	S	S	N	N	N	N	N	N	N	N
<b>Monto de la Compra Mediana</b>	S	S	S	S	N	N	N	N	S	S	S	S	N	N	N	N
<b>Monto de la Compra Grande</b>	S	S	N	N	S	S	N	N	S	S	N	N	S	S	N	N
<b>Cliente frecuente</b>	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N
ACCIONES																
<b>Descuento = 0</b>																
<b>Descuento = 5%</b>																
<b>Descuento = 10%</b>																

- Ejemplo: Diseñar un programa que aplique descuentos a compras del 0%, 5% y 10% según si el monto de la compra es pequeño, mediano o grande y de si el cliente es frecuente o no.

CONDICIONES	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Monto de la Compra Pequeña	S	S	S	S	S	S	S	S	N	N	N	N	N	N	N	N
Monto de la Compra Mediana	S	S	S	S	N	N	N	N	S	S	S	S	N	N	N	N
Monto de la Compra Grande	S	S	N	N	S	S	N	N	S	S	N	N	S	S	N	N
Cliente frecuente	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N
ACCIONES																
Descuento = 0																
Descuento = 5%																
Descuento = 10%																



- Ejemplo: Diseñar un programa que aplique descuentos a compras del 0%, 5% y 10% según si el monto de la compra es pequeño, mediano o grande y de si el cliente es frecuente o no.

CONDICIONES	7	8	11	12	13	14
Monto de la Compra Pequeña	S	S	N	N	N	N
Monto de la Compra Mediana	N	N	S	S	N	N
Monto de la Compra Grande	N	N	N	N	S	S
Cliente frecuente	S	N	S	N	S	N
ACCIONES						
Descuento = 0						
Descuento = 5%						
Descuento = 10%						

- Ejemplo: Diseñar un programa que aplique descuentos a compras del 0%, 5% y 10% según si el monto de la compra es pequeño, mediano o grande y de si el cliente es frecuente o no.

CONDICIONES	7	8	11	12	13	14
Monto de la Compra Pequeña	S	S	N	N	N	N
Monto de la Compra Mediana	N	N	S	S	N	N
Monto de la Compra Grande	N	N	N	N	S	S
Cliente frecuente	S	N	S	N	S	N
ACCIONES						
Descuento = 0			X			
Descuento = 5%	X			X		
Descuento = 10%			X		X	X

- **Ejemplo:** Determinar la nómina de los empleados de acuerdo con estos criterios:
  - Si el empleado es altamente productivo, tendrá en nómina un plus de productividad.
  - Si el empleado es encargado de su grupo, tendrá en nómina un plus de encargado.
  - Si el empleado ha cometido una infracción grave durante ese mes, le será eliminado cualquier plus

	1	2	3	4	5	6	7	8
Empleado altamente productivo	Sí	Sí	Sí	No	No	No	Sí	No
Empleado encargado	Sí	Sí	No	Sí	No	Sí	No	No
Infracción grave	Sí	No	Sí	Sí	Sí	No	No	No
Inviable								
Plus productividad			X					X
Plus encargado			X				X	
Sin pluses	X		X	X	X			
Calcular nómina	X	X	X	X	X	X	X	X



- Un **editor de código** es un tipo de software sencillo que incluye básicamente un **editor de texto** y algunas utilidades y herramientas básicas para el proceso de edición de código en **múltiples lenguajes de programación**.

**Sublime Text, Visual Studio Code, Atom, Notepad++,  
Brackets, Gedit, Geany, nano, vim, Emacs...**

- Un **IDE (Integrated Development Environment)** o **entorno de desarrollo integrado** es un tipo de software de mayor complejidad que incluye el **editor de texto y herramientas avanzadas para programar** de la forma más eficiente y completa posible dentro del propio software (p. ej. herramientas de compilación, depuración de programas, conexión a bases de datos, etc.), orientándose normalmente a uno o a pocos lenguajes de programación.

**Visual Studio, Eclipse, PyCharm, Komodo IDE, NetBeans,**  
**Spyder, Thonny, AWS Cloud9, Codeanywhere...**

	<b>Editor de código</b>	<b>IDE</b>
<b>FUNCIÓN</b>	Escribir código	Escribir, compilar y ejecutar código
<b>CARACTERÍSTICAS</b>	Funcionalidades de asistencia al desarrollador en la escritura de código	Funcionalidades de escritura y depuración de código mediante puntos de ruptura ( <i>breakpoints</i> ), etc.
<b>LENGUAJES DE PROGRAMACIÓN</b>	Soporta múltiples lenguajes de programación	Generalmente se centra en ofrecer soporte para uno o pocos lenguajes de programación
<b>COMPILEADOR Y DEPURADOR</b>	No lo hay	Sí lo hay
<b>AUTOCOMPLETADO DE CÓDIGO</b>	Sí lo hay	Sí lo hay
<b>RESALTADO Y COLOREADO DE CÓDIGO</b>	Sí lo hay	Sí lo hay
<b>GUIADO Y ASISTENCIA</b>	Sí lo hay	Sí lo hay

- La elección de un editor de código o de un IDE está cargada de la **componente del gusto personal**
- El programador/a debe estar **en sintonía** con la herramienta con la que trabaja, a la par que la utiliza **de forma ágil y eficiente**
- Así, **no hay un mejor editor de código o un mejor IDE**. El mejor será el que más nos guste y sea más productivo para nosotros

# U2.2. Fundamentos de programación en Python



**Un programa inicial en Python...**

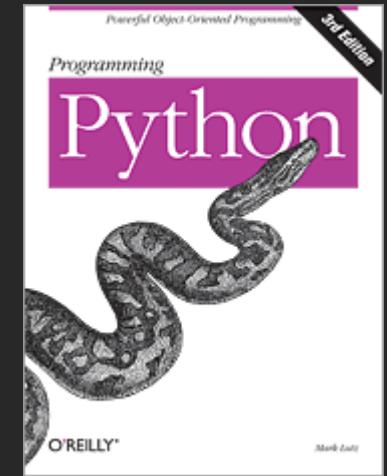
```
name = input('Enter file:')  
handle = open(name)  
  
counts = dict()  
for line in handle:  
    words = line.split()  
    for word in words:  
        counts[word] = counts.get(word, 0) + 1  
  
bigcount = None  
bigword = None  
for word, count in counts.items():  
    if bigcount is None or count > bigcount:  
        bigword = word  
        bigcount = count  
  
print(bigword, bigcount)
```

```
python words.py  
Enter file: words.txt  
to 16
```

```
python words.py  
Enter file: clown.txt  
el 7
```

# El lenguaje Python

- Python da nombre al lenguaje de programación y al intérprete en línea de comandos
- Inicialmente desarrollado por Guido van Rossum
- Ideado como lenguaje divertido y fácil de usar



```
csev$ python3
```

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)  
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on Darwin Type  
"help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

Prompt de  
Python



```
csev$ python3
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on Darwin Type
"help", "copyright", "credits" or "license" for more information.

>>> x = 1
>>> print (x)
1
>>> x = x + 1
>>> print (x)
2
>>> exit()
```

Esta es una buena prueba para asegurarse de que se ha instalado Python correctamente.

La función `quit()` también serviría para terminar una sesión interactiva.

# Palabras Reservadas

- No pueden utilizarse las siguientes palabras reservadas como nombres o identificadores de variables

False	class	return	is	finally
None	if	for	lambda	continue
True	def	from	while	nonlocal
and	del	global	not	with
as	elif	try	or	yield
assert	else	import	pass	
break	except	in	raise	

# Sentencias o instrucciones

**x = 2** ← Asignación  
**x = x + 2** ← Asignación con expresión  
**print(x)** ← Función print (imprimir)

Variable

Operador

Constante

Función

# Scripts de Python

- El intérprete de Python en línea de comandos (Python interactivo) es adecuado para pruebas y programas muy cortos.
- La mayoría de programas estarán en archivos, indicándole a Python que interprete (ejecute) las instrucciones de esos archivos.
- Esos archivos son scripts (guiones) de Python
- Como convención, los scripts de Python llevan “.py” como extensión de archivo.

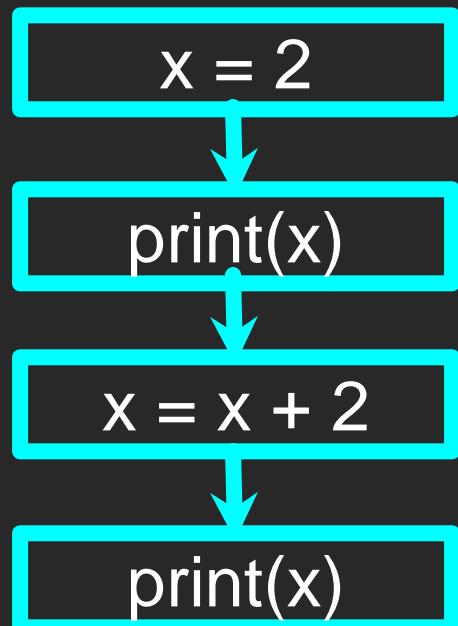
# Interactivo vs. Script

- Interactivo
  - Escribimos directamente en el intérprete de Python en línea de comandos, instrucción a instrucción
- Script
  - Escribimos todas las instrucciones que componen el script en un archivo .py y le indicamos al intérprete Python que lo ejecute

# Flujo de un programa

- Al igual que en una receta o en un manual de usuario, un programa es una **secuencia** de instrucciones que se deben dar en orden.
- Algunas instrucciones son **condicionales**, es decir, pueden saltarse.
- A veces, una instrucción o grupo de ellas debe **repetirse**.
- Otras veces, almacenamos un conjunto de instrucciones para utilizarlas una y otra vez en distintos lugares del programa (**funciones**).

# Pasos secuenciales



Programa:

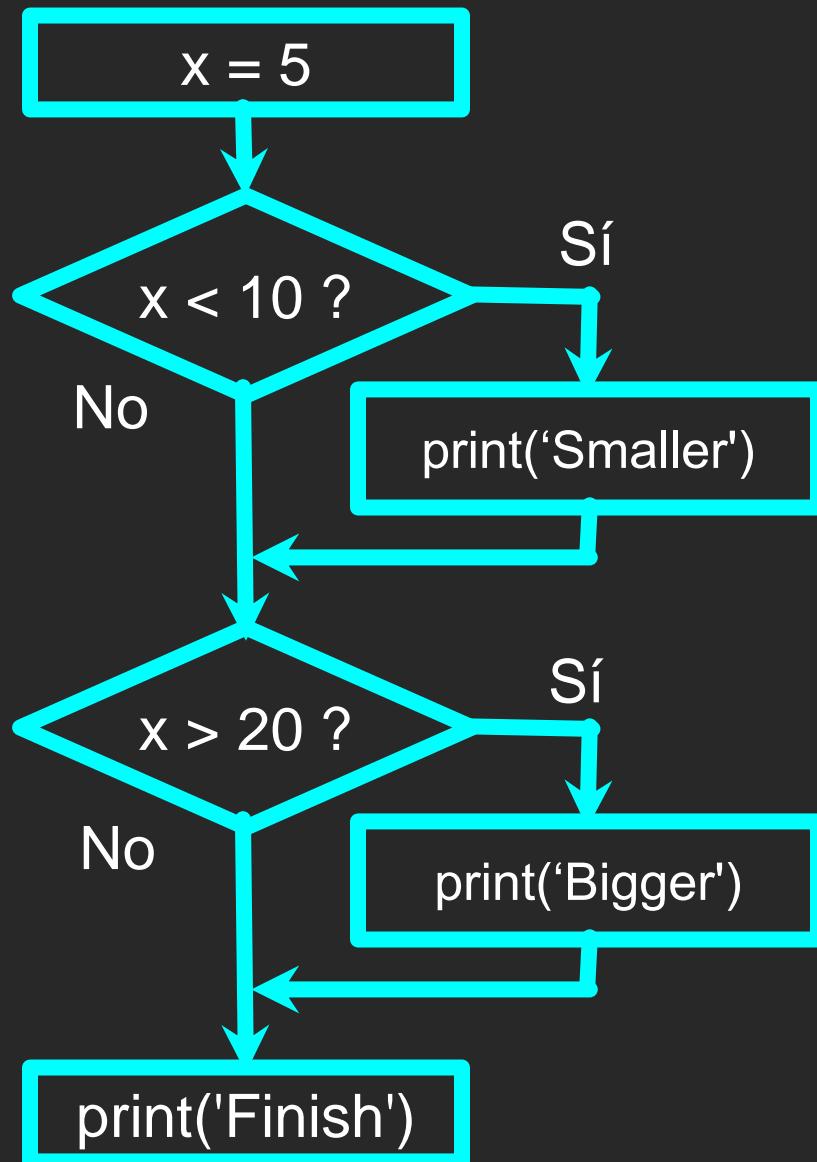
```
x = 2  
print(x)  
x = x + 2  
print(x)
```

Resultado:

```
2  
4
```

Arrows point from the second and fourth lines of code to the corresponding output values, 2 and 4, respectively.

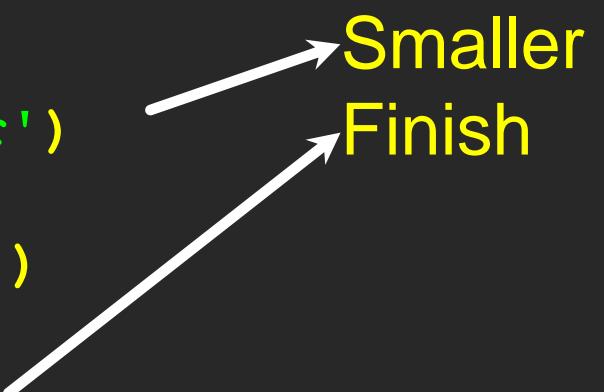
# Pasos condicionales



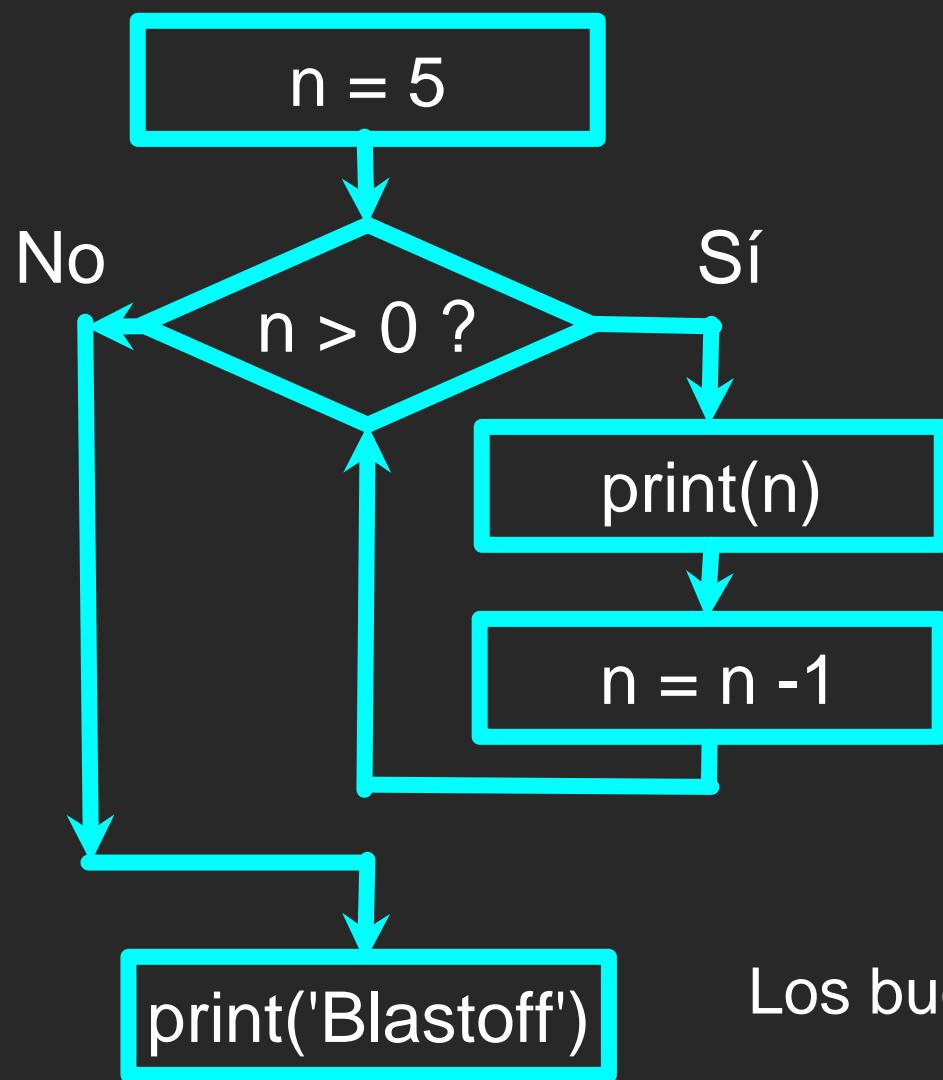
Programa:

```
x = 5
if x < 10:
    print(Smaller)
if x > 20:
    print(Bigger)
print('Finish')
```

Resultado:



# Pasos repetidos



Programa:

```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Blastoff')
```

Resultado:

5  
4  
3  
2  
1  
Blastoff

Los bucles (pasos repetidos) tienen variables de iteración que cambian cada vez a través del bucle

```
name = input('Enter file:')                                Secuencial
handle = open(name, 'r')                                

counts = dict()                                         Repetido
for line in handle:                                     Condicional
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None
for word, count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

```
name = input('Enter file:')  
handle = open(name, 'r')  
  
counts = dict()  
for line in handle:  
    words = line.split()  
    for word in words:  
        counts[word] = counts.get(word, 0) + 1  
  
bigcount = None  
bigword = None  
for word, count in counts.items():  
    if bigcount is None or count > bigcount:  
        bigword = word  
        bigcount = count  
  
print(bigword, bigcount)
```

Partes secuenciales para contar palabras en un archivo

Lectura de la entrada de datos del usuario

Instrucción para actualizar uno de los muchos contadores

Fragmento condicional para encontrar el número más grande en una lista

# Constantes, variables, expresiones e instrucciones

# Constantes

- Los **valores fijos** como números, letras y cadenas de texto reciben el nombre de **constants** porque su valor no cambia
- Las **constants numéricas** funcionan como se espera
- Las **constants** de cadena de texto funcionan con comillas simples ('') o dobles (")

```
>>> print(123)  
123  
>>> print(98.6)  
98.6  
>>> print('Hola mundo')  
Hola mundo
```

# Variables

- Una **variable** es una zona de memoria RAM donde el programa almacena y recupera datos en tiempo de ejecución
- El nombre de **variable** utilizado, elegido por el programador, permite acceder a esa ubicación de memoria.
- Es posible cambiar el contenido de una **variable**, de ahí su nombre.

**x** = 12.2

**y** = 14

**x** = 100

**x** ~~12.2~~ 100

**y** 14

# Nombres de variables en Python

- Debe comenzar con una letra o guión bajo\_
- Debe constar de letras, números y guión bajo
- Son sensibles a mayúsculas y minúsculas

**Bien:**      spam      eggs      spam23      \_speed

**Mal:**      23spam      #sign      var.12

**Diferentes:**      spam      Spam      SPAM

# Nombres de variables en Python

- Como programadores, tenemos la libertad de elegir los nombres de variables, por lo que intentaremos aplicar “buenas prácticas”
- Nombramos a las variables de modo que nos permitan recordar qué nos proponemos guardar en ellas
- Seguiremos convenciones de código (por ejemplo, camelCase) a la hora de nombrarlas

```
x1q3z9ocd = 35.0  
x1q3z9afd = 12.50  
x1q3p9afd = x1q3z9ocd * x1q3z9afd  
print(x1q3p9afd)
```

¿Qué hace este  
código?

```
x1q3z9ocd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ocd * x1q3z9afd
print(x1q3p9afd)

a = 35.0
b = 12.50
c = a * b
print(c)
```

¿Qué hacen  
estos códigos?

```
x1q3z9ocd = 35.0  
x1q3z9afd = 12.50  
x1q3p9afd = x1q3z9ocd * x1q3z9afd  
print(x1q3p9afd)  
  
a = 35.0  
b = 12.50  
c = a * b  
print(c)
```

¿Qué hacen  
estos códigos?

```
hours = 35.0  
rate = 12.50  
pay = hours * rate  
print(pay)
```

*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.3. Fundamentos de programación en Python

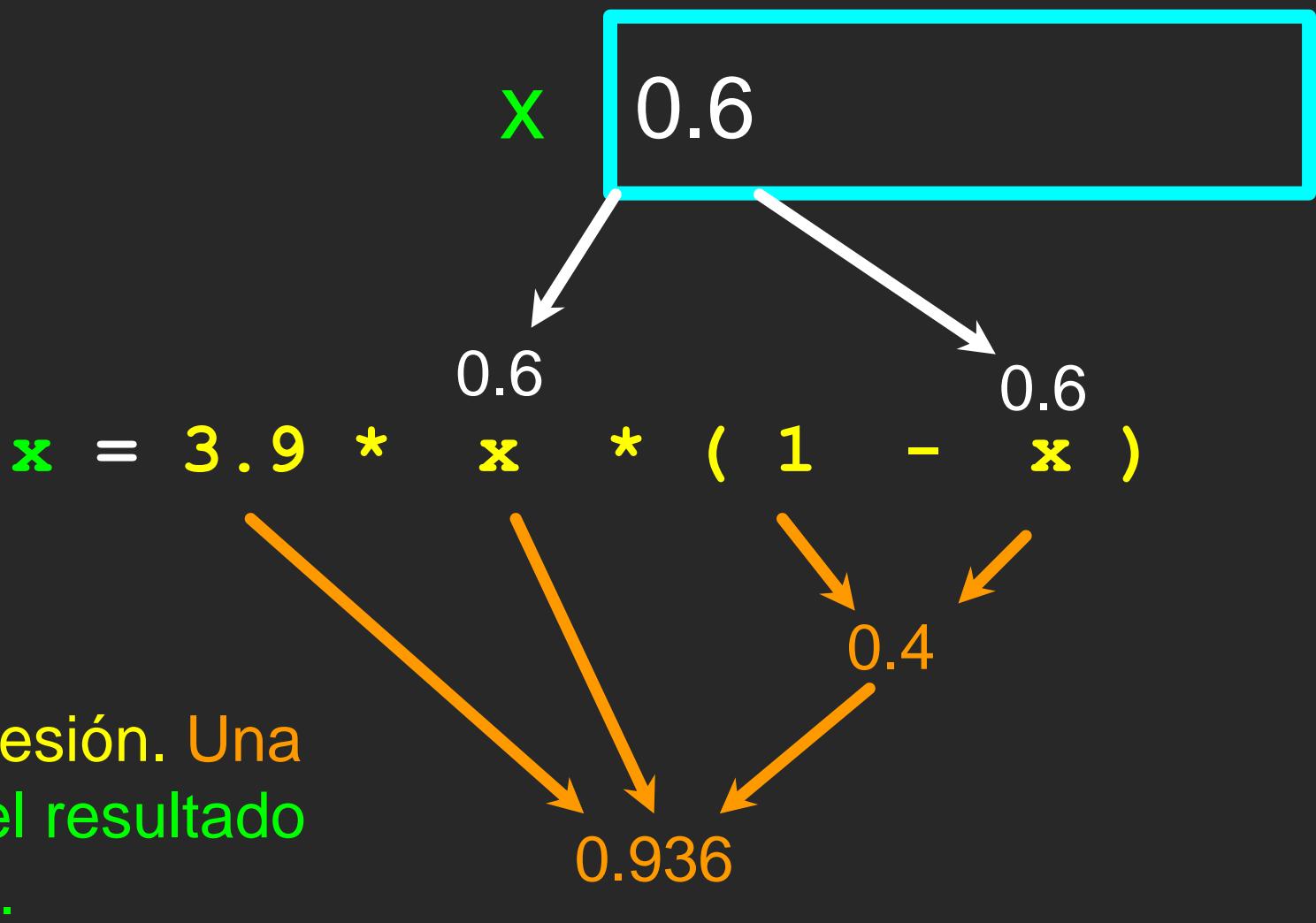


# Asignación

Asignamos un valor a una variable utilizando la asignación (=)

Una instrucción de asignación consta de una expresión en el lado derecho y una variable para almacenar el resultado

$$x = 3.9 * x * (1 - x)$$

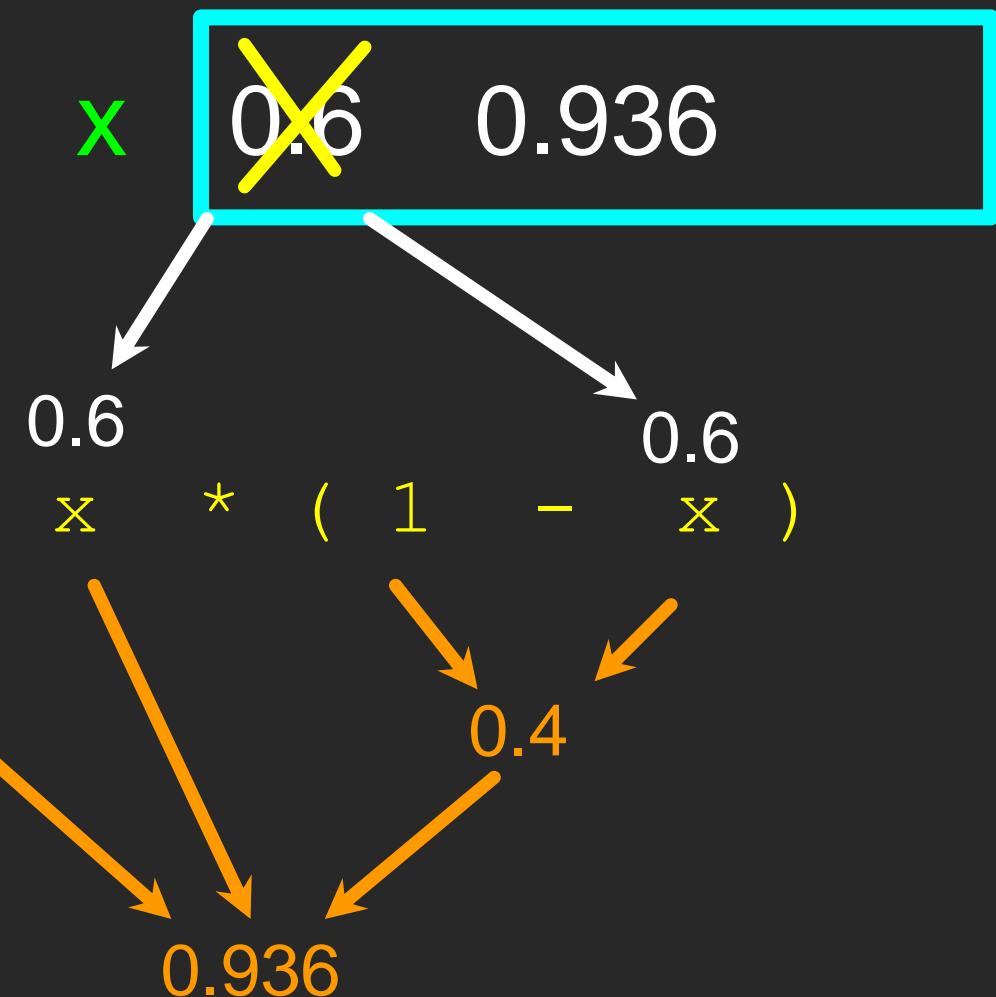


El lado derecho es una expresión. Una vez evaluada la expresión, el resultado se coloca en (se asigna a) x.

En una variable se puede reemplazar el valor anterior (0.6) con uno nuevo (0.936).

$$x = 3.9 * x$$

El lado derecho es una expresión. Una vez evaluada la expresión, el resultado se coloca en (se asigna a) la variable que está a la izquierda (es decir, x).



# Expresiones numéricas

- Expresamos las operaciones matemáticas con ciertos símbolos que no tienen por qué coincidir con los utilizados en papel
- El asterisco es la multiplicación
- La barra es la división
- La potenciación (elevar a la potencia) utiliza el doble asterisco
- El porcentaje indica módulo o resto de la división entera

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
**	Potencia
%	Resto

# Expresiones numéricas

```
>>> xx = 2  
>>> xx = xx + 2  
>>> print(xx)  
4  
>>> yy = 440 * 12  
>>> print(yy)  
5280  
>>> zz = yy / 1000  
>>> print(zz)  
5.28
```

```
>>> jj = 23  
>>> kk = jj % 5  
>>> print(kk)  
3  
>>> print(4 ** 3)  
64
```

$$\begin{array}{r} 4 \text{ R } 3 \\ \hline 5 \quad \left| \begin{array}{r} 23 \\ 20 \\ \hline 3 \end{array} \right. \end{array}$$

Operador	Operación
+	Suma
-	Resta
*	Multiplicación
/	División
**	Potencia
%	Resto

# Orden de evaluación

- Cuando introducimos una serie de operadores, Python debe saber cuál tiene que evaluar o calcular primero
- Esto recibe el nombre de “precedencia de los operadores”
- En el ejemplo, ¿qué operador “tiene precedencia” sobre los otros?

```
x = 1 + 2 * 3 - 4 / 5 ** 6
```

# Reglas de precedencia

De la regla de precedencia más alta a la regla de precedencia más baja:

- Siempre se respetan los paréntesis
- Potenciación (elevar a la potencia)
- Multiplicación, división, resto
- Suma y resta
- De izquierda a derecha

Paréntesis

Potencia

Multiplicación

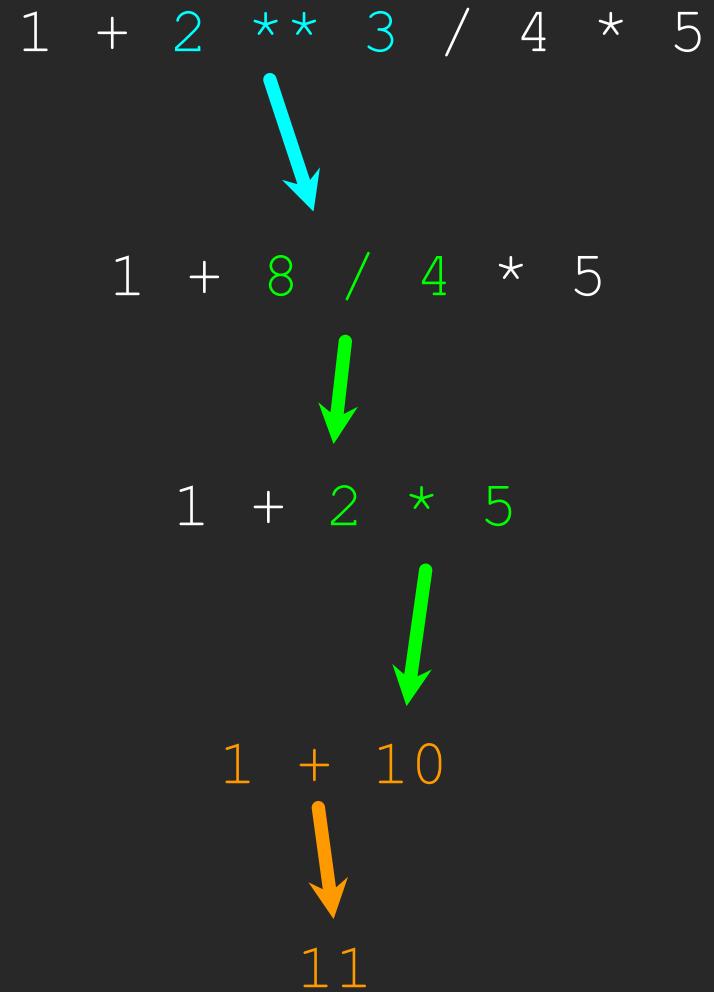
Suma

Izquierda a  
derecha



```
>>> x = 1 + 2 ** 3 / 4 * 5  
>>> print(x)  
11.0  
>>>
```

Paréntesis  
Potencia  
Multiplicación  
Suma  
Izquierda a  
derecha



# Reglas de precedencia

- Tener en cuenta las reglas de arriba hacia abajo
- Es más que recomendable utilizar paréntesis
- Usar expresiones matemáticas lo más simples posible para que sean fáciles de entender
- Dividir las series de operaciones matemáticas largas en varias operaciones más cortas para mayor claridad

Paréntesis  
Potencia  
Multiplicación  
Suma  
Izquierda a  
derecha



# El tipo de las variables

- En Python, las variables, literales y constantes tienen un **tipo**
- Python **diferencia** entre un número entero y una cadena
- Por ejemplo, “**+**” significa “sumar” si se trata de números y “concatenación” si se trata de cadenas

```
>>> ddd = 1 + 4  
>>> print(ddd)  
5  
  
>>> eee = 'hola ' + 'a  
todos'  
>>> print(eee)  
Hola a todos
```

concatenación = unión

# El tipo importa

- Python asigna un “**tipo**” a cualquier variable
- Algunas operaciones están prohibidas según el tipo de variable que sea
- No se puede sumar 1 a una cadena
- Podemos averiguar de qué tipo se trata con la función **type()**

```
>>> eee = 'hello ' + 'there'  
>>> eee = eee + 1  
Traceback (most recent call last):  
File "<stdin>", line 1, in  
<module>TypeError: Can't convert  
'int' object to str implicitly  
>>> type(eee)  
<class 'str'>  
>>> type('hello')  
<class 'str'>  
>>> type(1)  
<class 'int'>  
>>>
```

# Diferentes tipos de número

- Los números tienen dos tipos
  - Enteros (int):  
-14, -2, 0, 1, 100, 401233
  - Números de punto flotante (float), que tienen decimales:  
-2.5 , 0.0, 98.6, 14.0
- Hay otros tipos : son variantes entre los decimales y los enteros

```
>>> xx = 1
>>> type(xx)
<class 'int'>
>>> temp = 98.6
>>> type(temp)
<class 'float'>
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>>
```

# Conversiones de tipo

- Al introducir un número entero y un decimal en una expresión, el entero (int) se convierte **implícitamente** en uno decimal (float)
- Esto se puede controlar de **explícitamente** con las funciones incorporadas int() y float()

```
>>> print(float(99) + 100)
199.0
>>> i = 42
>>> type(i)
<class'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class'float'>
>>>
```

# División de números enteros

- La división de números enteros siempre devuelve un resultado decimal (con punto flotante o float)

```
>>> print(10 / 2)  
5.0  
>>> print(9 / 2)  
4.5  
>>> print(99 / 100)  
0.99  
>>> print(10.0 / 2.0)  
5.0  
>>> print(99.0 / 100.0)  
0.99
```

La división de enteros era diferente en Python 2.x

# Conversiones de cadenas

- También se puede utilizar `int()` y `float()` para realizar conversiones entre cadenas y enteros
- Obtendremos un `error` si la cadena no contiene caracteres numéricos

```
>>> sval = '123'  
>>> type(sval)  
<class 'str'>  
>>> print(sval + 1)  
Traceback (most recent call last): File  
<stdin>, line 1, in <module>  
TypeError: Can't convert 'int' object to  
str implicitly  
>>> ival = int(sval)  
>>> type(ival)  
<class 'int'>  
>>> print(ival + 1)  
124  
>>> nsv = 'hello bob'  
>>> niv = int(nsv)  
Traceback (most recent call last): File  
<stdin>, line 1, in <module>  
ValueError: invalid literal for int()  
with base 10: 'x'
```

# Input o entrada de usuario

- Podemos hacer que Python haga una pausa y lea datos introducidos por el usuario con la función `input()`
- La función `input()` devuelve una cadena

```
name = input('¿Quién eres?')  
print('Bienvenid@,' , name)
```

¿Quién eres?  
Carlos  
Bienvenid@, Carlos

# Conversión de la entrada de usuario

- Si queremos leer un número de la entrada de usuario, debemos convertirlo de cadena a número usando la función de conversión de tipo adecuada



```
euf = input('¿Planta en Europa?')  
usf = int(euf) + 1  
print('Planta en USA:', usf)
```

¿Planta en Europa? 0  
Planta en USA: 1

# Comentarios en Python

- Todo lo que vaya después de un `#` es ignorado por Python
- ¿Por qué usar comentarios?
  - Permiten describir lo que se realiza en un fragmento de código
  - Permiten documentar quién escribió el código o proporcionar información adicional
  - Permiten desactivar una línea de un código, quizás de manera temporal

```
# Get the name of the file and open it
name = input('Enter file:')
handle = open(name, 'r')

# Count word frequency
counts = dict()
for line in handle:
    words = line.split()
    for word in words:
        counts[word] = counts.get(word, 0) + 1

# Find the most common word
bigcount = None
bigword = None
for word, count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

# All done
print(bigword, bigcount)
```

# Comentarios multilínea

- También podemos hacer **comentarios multilínea** con `'''` (tres comillas simples para iniciar el comentario y para finalizarlo) o con `"""` (tres comillas dobles para iniciar el comentario y para finalizarlo)
- Sin embargo, **las guías de estilo de Python no recomiendan los comentarios multilínea** (recomiendan el uso de `#`)

```
''' Esto es  
un comentario  
multilínea
```

```
""" Esto  
también lo es """
```

## Ejercicio

Escribe un programa que solicite, como entrada al usuario, el número de horas trabajadas al día y la tarifa por hora y obtenga, como salida, el salario mensual bruto, considerando que, en promedio, hay 22 días laborables al mes.

Introducir horas/día: 8

Introducir tarifa/hora: 11.5

Salario: 2024.0 €

*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

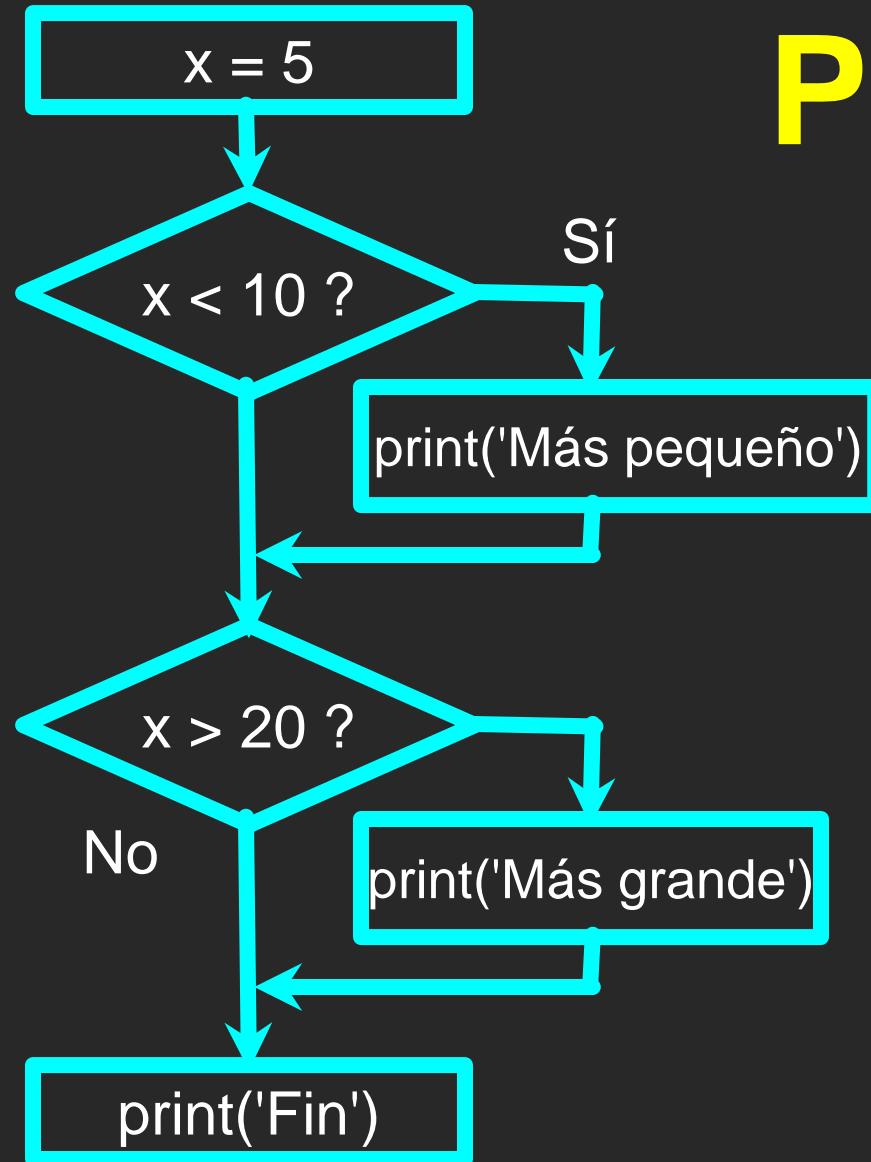
*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.4. Fundamentos de programación en Python



# Pasos Condicionales



Programa:

```
x = 5
if x < 10:
    print ('Más pequeño')
if x > 20:
    print ('Más grande')
print ('Fin')
```

Resultado:

```
Más pequeño
Fin
```

# Operadores de comparación

- Las expresiones booleanas formulan una pregunta y generan un resultado Sí (afirmativo) o No (negativo) que utilizamos para controlar el flujo del programa
- Las expresiones booleanas utilizan operadores de comparación para evaluar si es True (Verdadero) / False (Falso) o Yes (Sí) / No
- Los operadores de comparación observan las variables pero no las modifican

Python	Significado
<	Menor que
<=	Menor o igual que
==	Igual a
>=	Mayor o igual que
>	Mayor que
!=	Distinto

Recuerde: “=” se usa para asignación.

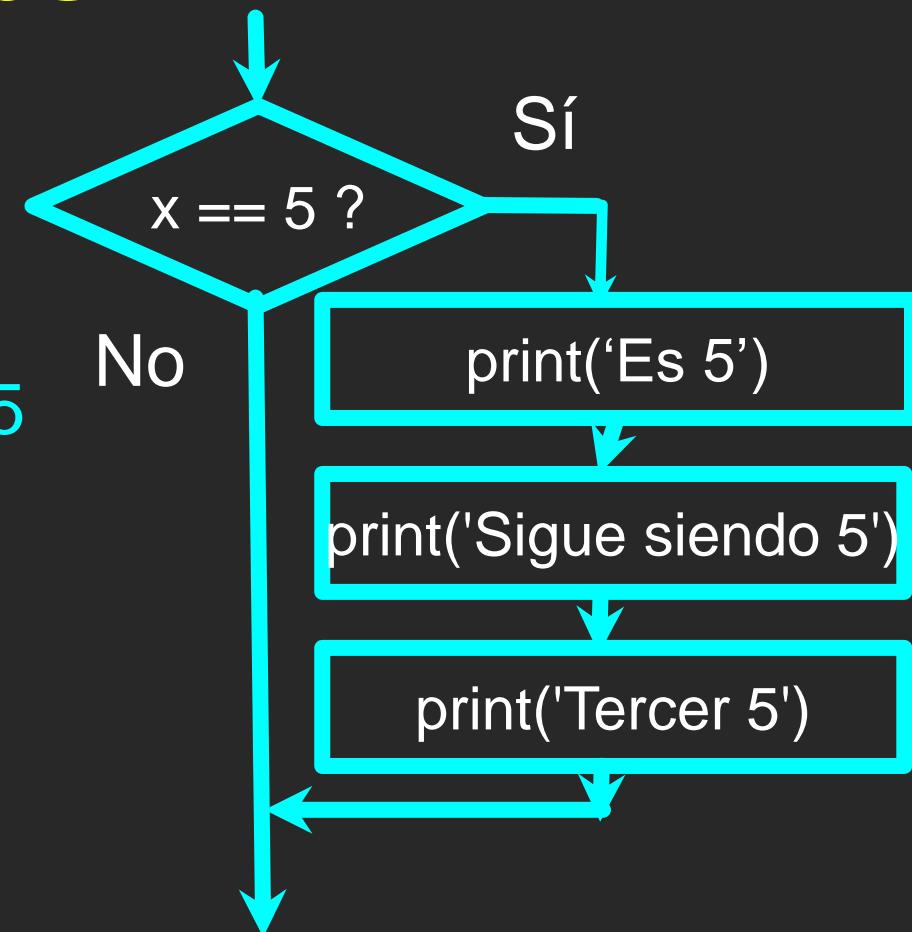
# Operadores de Comparación

```
x = 5
if x == 5 :
    print('Igual a 5')                                Igual a 5
if x > 4 :
    print('Mayor que 4')                            Mayor que 4
if x >= 5 :
    print('Mayor que o Igual a 5')                  Mayor que o Igual a 5
if x < 6 : print('Menor que 6') → Menor que 6
if x <= 5 :
    print('Menor que o Igual a 5')                  Menor que o Igual a 5
if x != 6 :
    print('Distinto de 6')                           Distinto de 6
```

# Decisiones unidireccionales

```
x = 5
print('Antes de 5')
if x == 5 :
    print('Es 5')
    print('Sigue Siendo 5')
    print('Tercer 5')
print('Después de 5')
print('Antes de 6')
if x == 6 :
    print('Es 6')
    print('Sigue siendo 6')
    print('Tercer 6')
print('Después de 6')
```

Antes de 5  
Es 5  
Sigue siendo 5  
Tercer 5  
Después de 5  
Antes de 6  
Después de 6



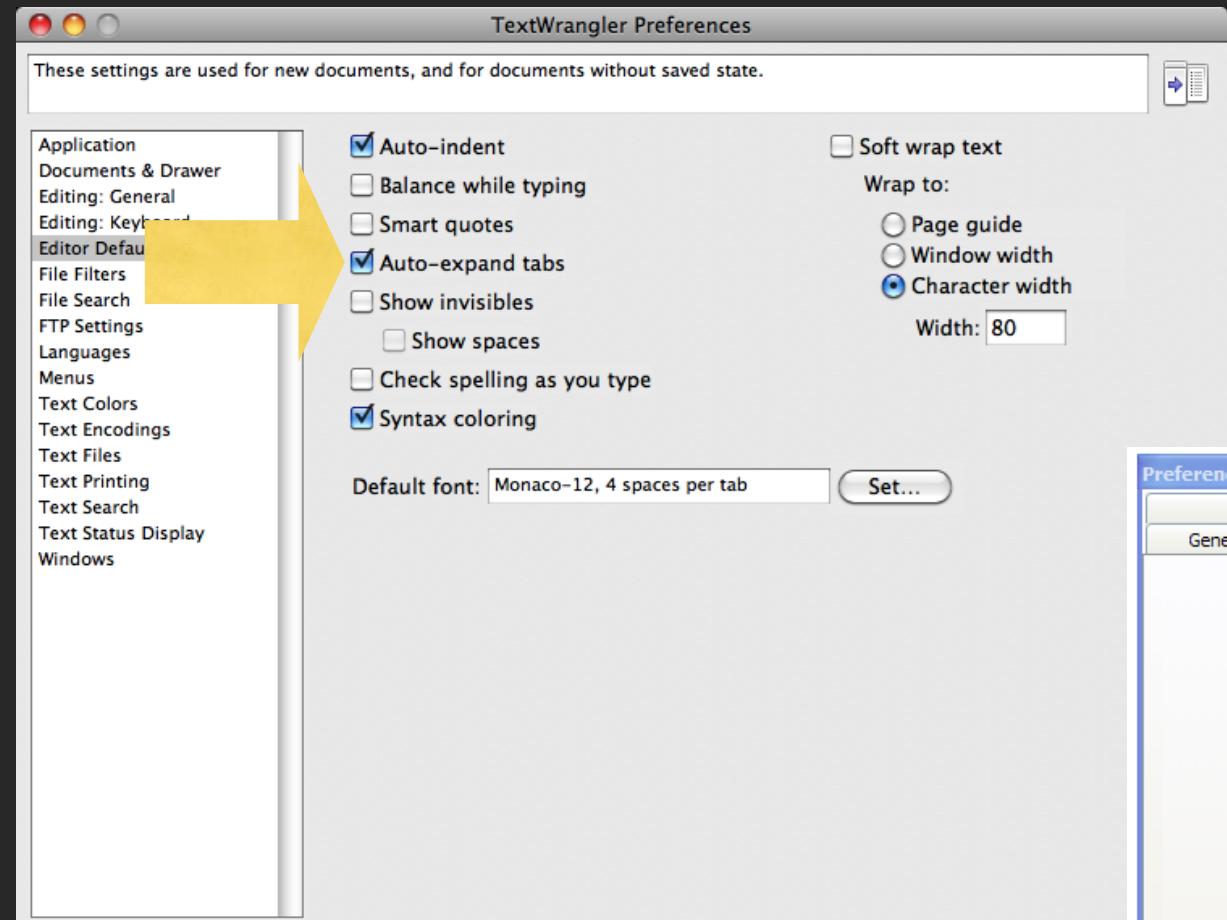
# Indentación

- Aumentar la indentación sirve para indentar luego de un enunciado **if** o **for** (después: )
- Mantener la indentación sirve para indicar el **alcance** del bloque (qué líneas son afectadas por **if/for**)
- Reducir la indentación permite regresarla al nivel del enunciado **if** o **for** para indicar el final del bloque
- Las líneas en blanco son ignoradas y no afectan la indentación
- Los comentarios en una línea en sí mismos se ignoran en lo que respecta a la indentación

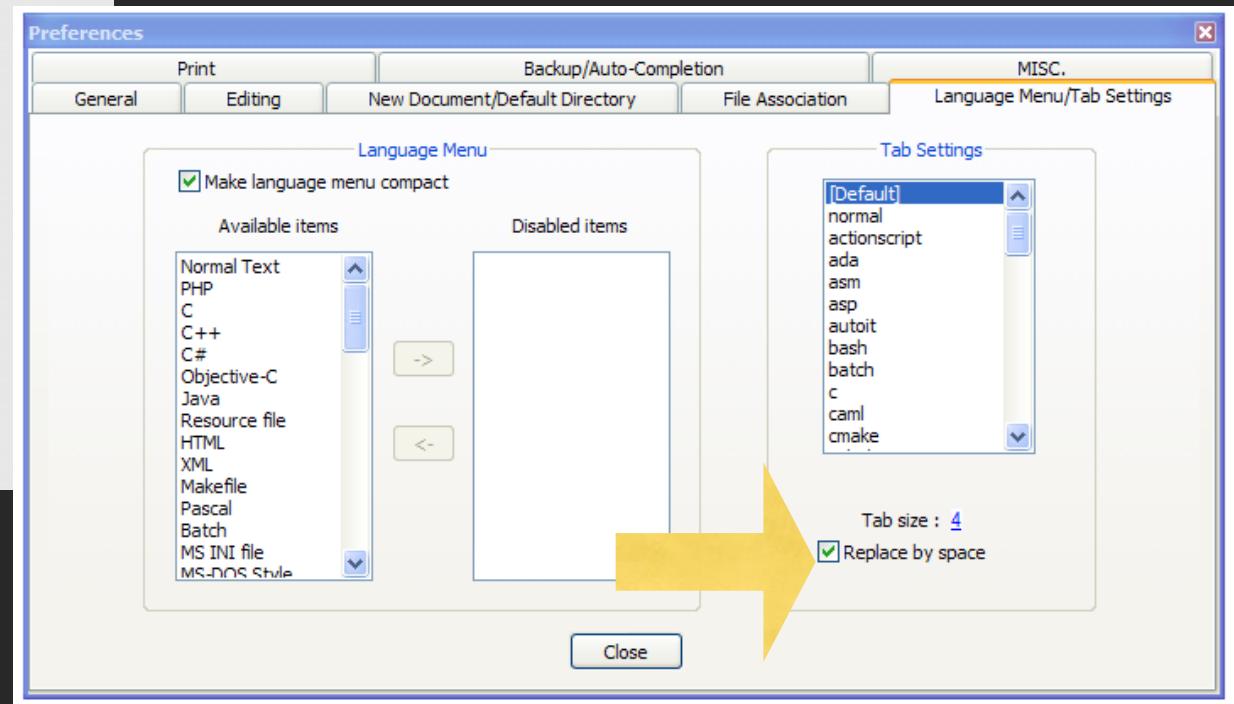
# Deshabilitar las tabulaciones

Atom u otros editores de código usan automáticamente los espacios para los archivos con la extensión ".py" (¡genial!)

- La mayoría de los editores de texto pueden convertir las **tabulaciones** en **espacios** – asegúrese de habilitar esta funcionalidad
  - NotePad++: Settings -> Preferences -> Language Menu/**Tab Settings** (Configuración -> Preferencias -> Menú de Idiomas/Configuración de **Tabulación**)
  - TextWrangler: TextWrangler -> Preferences -> Editor Defaults (TextWrangler: TextWrangler -> Preferencias -> Valores Predeterminados del Editor)
- A Python le importa *\*mucho\** cuánta indentación tiene una línea. Si usted mezcla **tabulaciones** y **espacios**, tal vez obtenga “**indentation errors**” (errores de indentación) incluso aunque todo se vea bien



Esto te ahorrará dolores  
de cabeza innecesarios.



aumentar / mantener después de if o for  
reducir para indicar el final del bloque

```
x = 5
if x > 2 :
    print('Mayor que 2')
    print('Sigue siendo mayor')
print('Terminado con 2')

for i in rango(5) :
    print(i)
    if i > 2 :
        print('Mayor que 2')
        print('Terminado con i', i)
print('Todo Terminado')
```

# Piensa en los bloques de inicio/fin

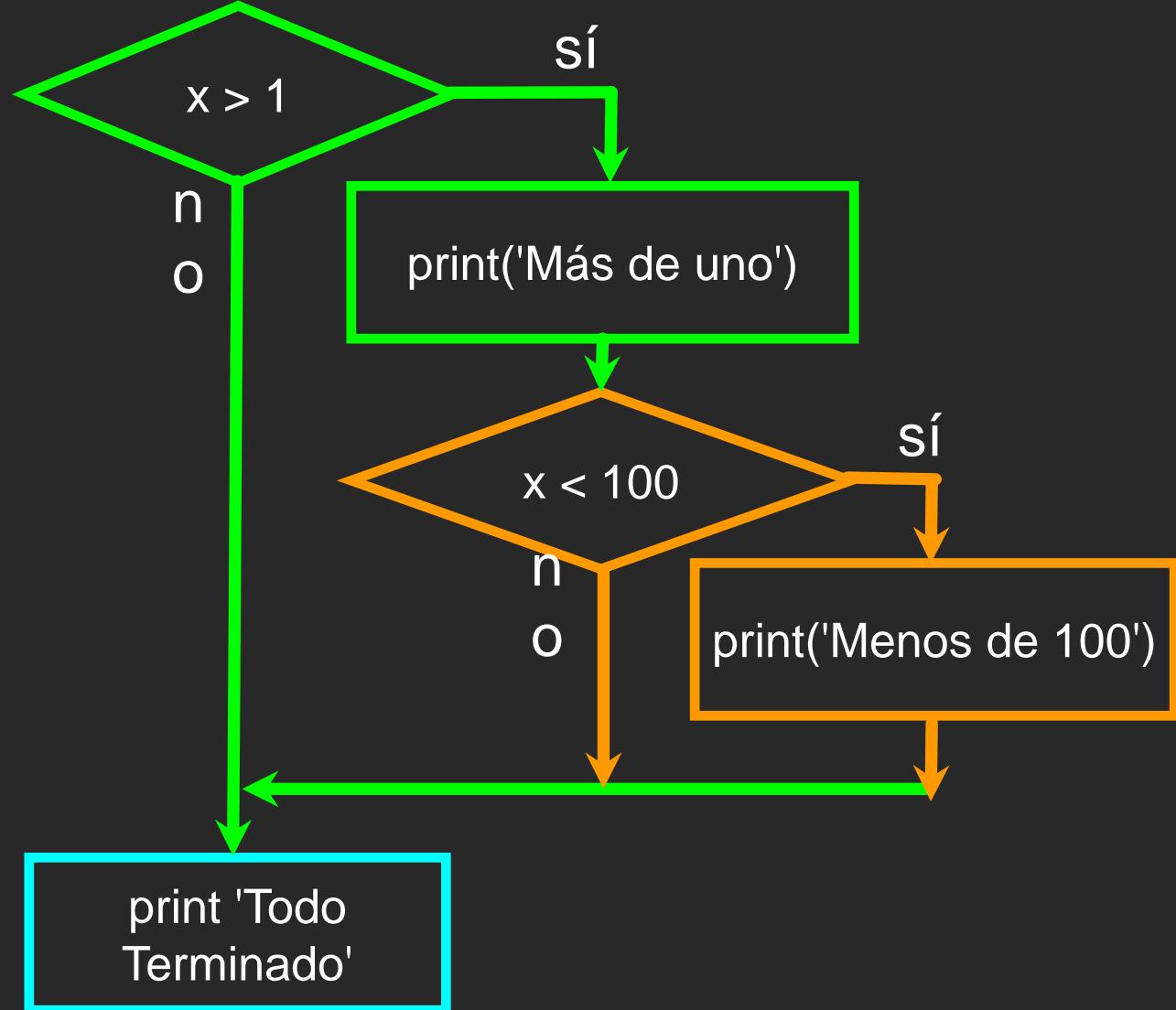
```
x = 5
if x > 2 :
    print('Mayor que 2')
    print('Sigue siendo mayor')
print('Terminado con 2')
```

```
for i in rango(5) :
    print(i)
    if i > 2 :
        print('Mayor que 2')
    print('Terminado con i', i)
```

```
print('Todo Terminado')
```

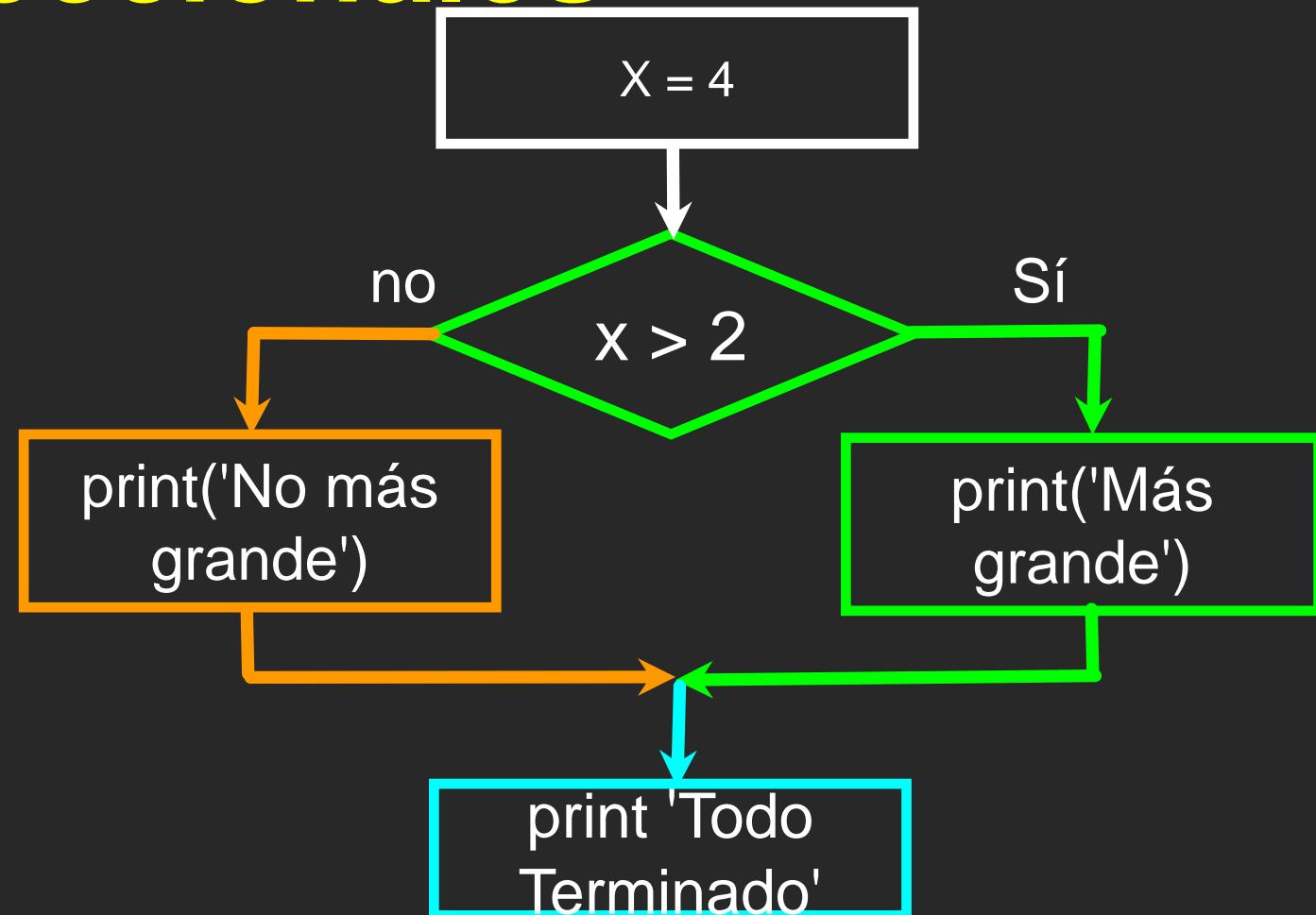
# Decisiones Anidadas

```
x = 42
if x > 1 :
    print('Más de 1')
    if x < 100 :
        print('Menos de 100')
print('Todo Terminado')
```



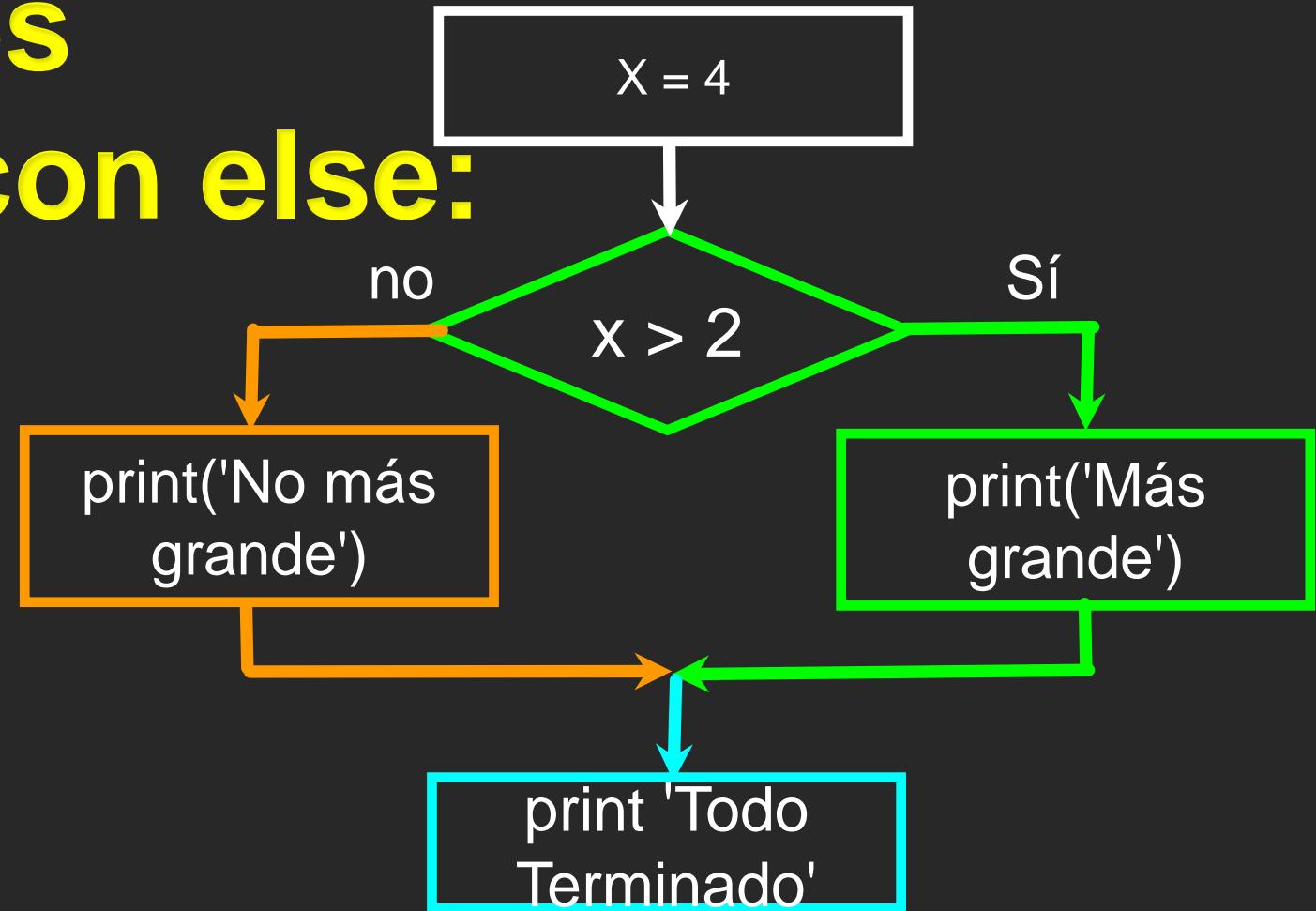
# Decisiones bidireccionales

- A veces, queremos hacer una cosa si una expresión lógica es verdadera y otra cosa si la expresión es falsa
- Es como una encrucijada – debemos elegir **un camino u otro** pero no podemos elegir ambos



# Decisiones bidireccionales con else:

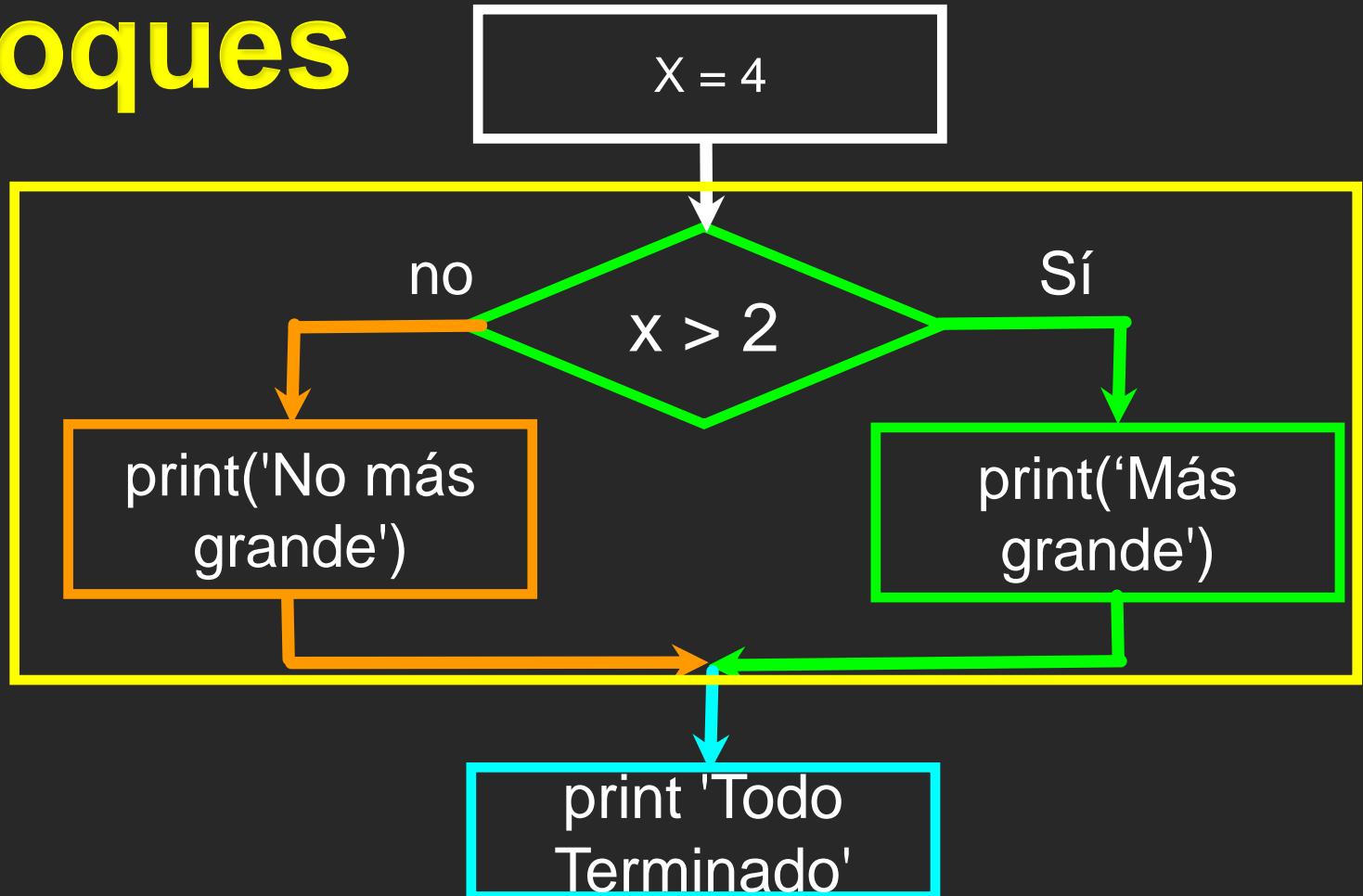
```
x = 4  
  
if x > 2 :  
    print('Más grande')  
else :  
    print('Más pequeño')  
  
print 'Todo Terminado'
```



# Más patrones de ejecución Condicional

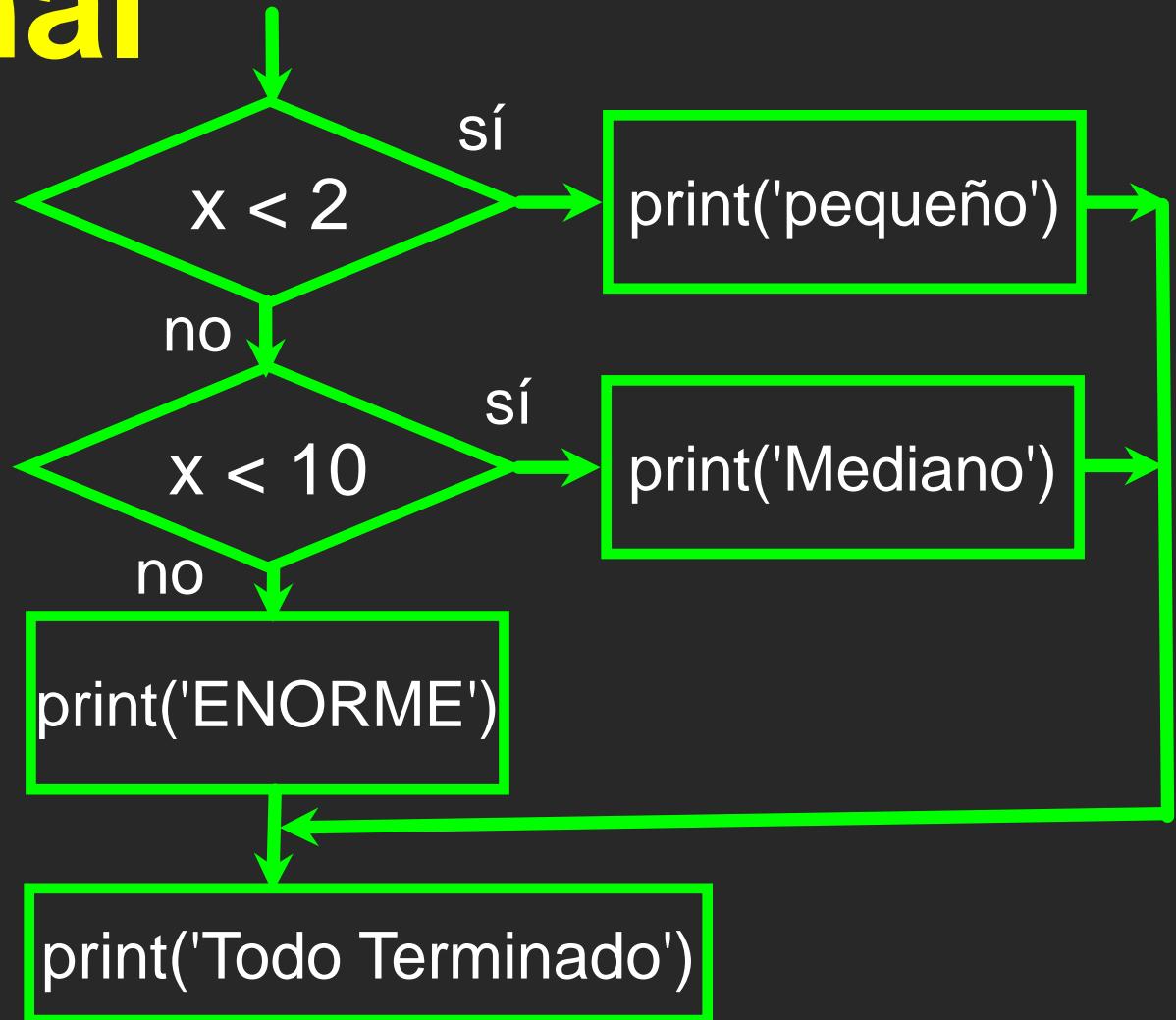
# Visualizar Bloques

```
x = 4  
if x > 2 :  
    print('Más grande')  
else :  
    print('Más pequeño')  
  
print 'Todo Terminado'
```



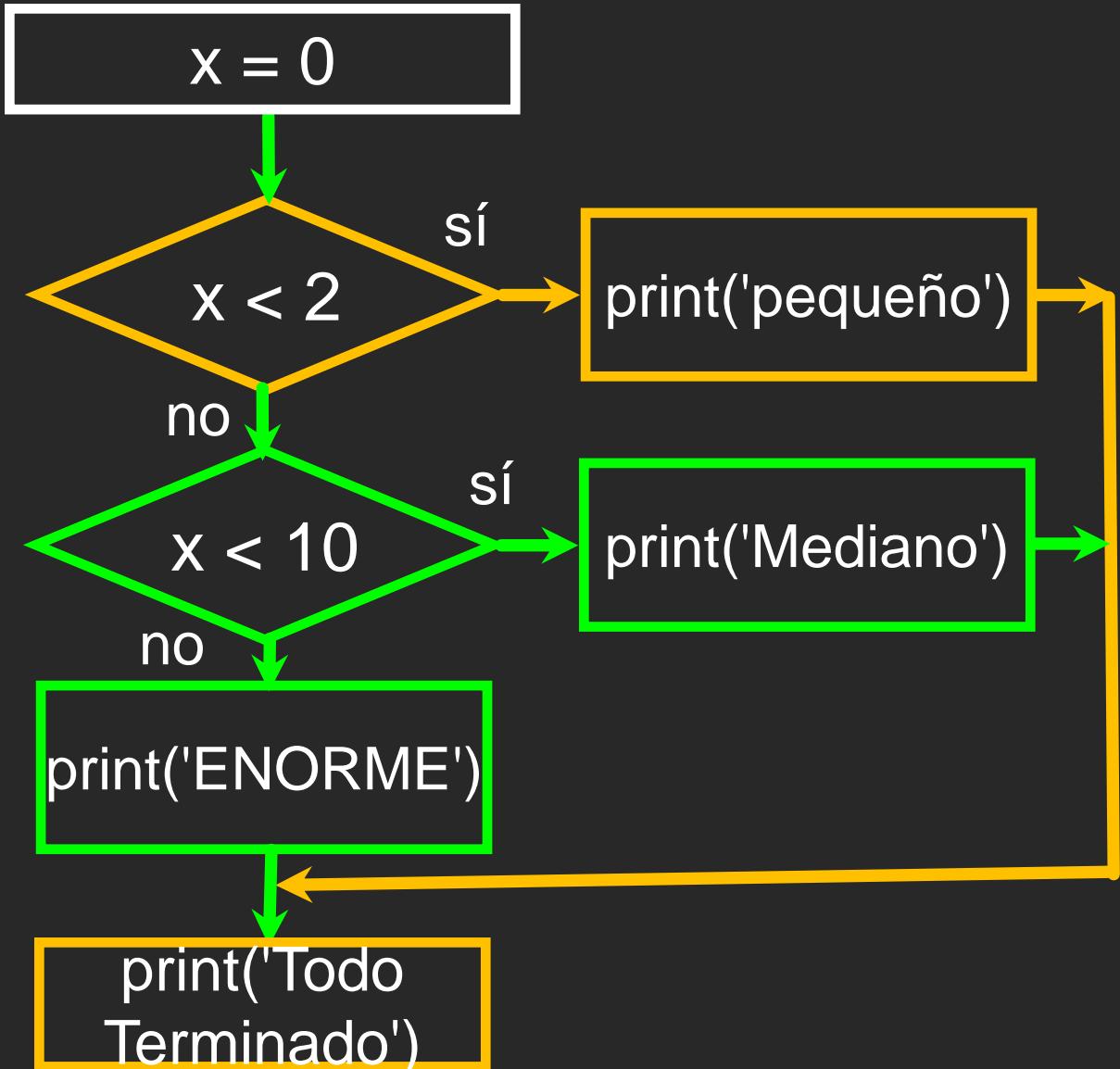
# Multidireccional

```
if x < 2 :  
    print('Pequeño')  
elif x < 10 :  
    print('Mediano')  
else :  
    print('ENORME')  
print('Todo terminado')
```



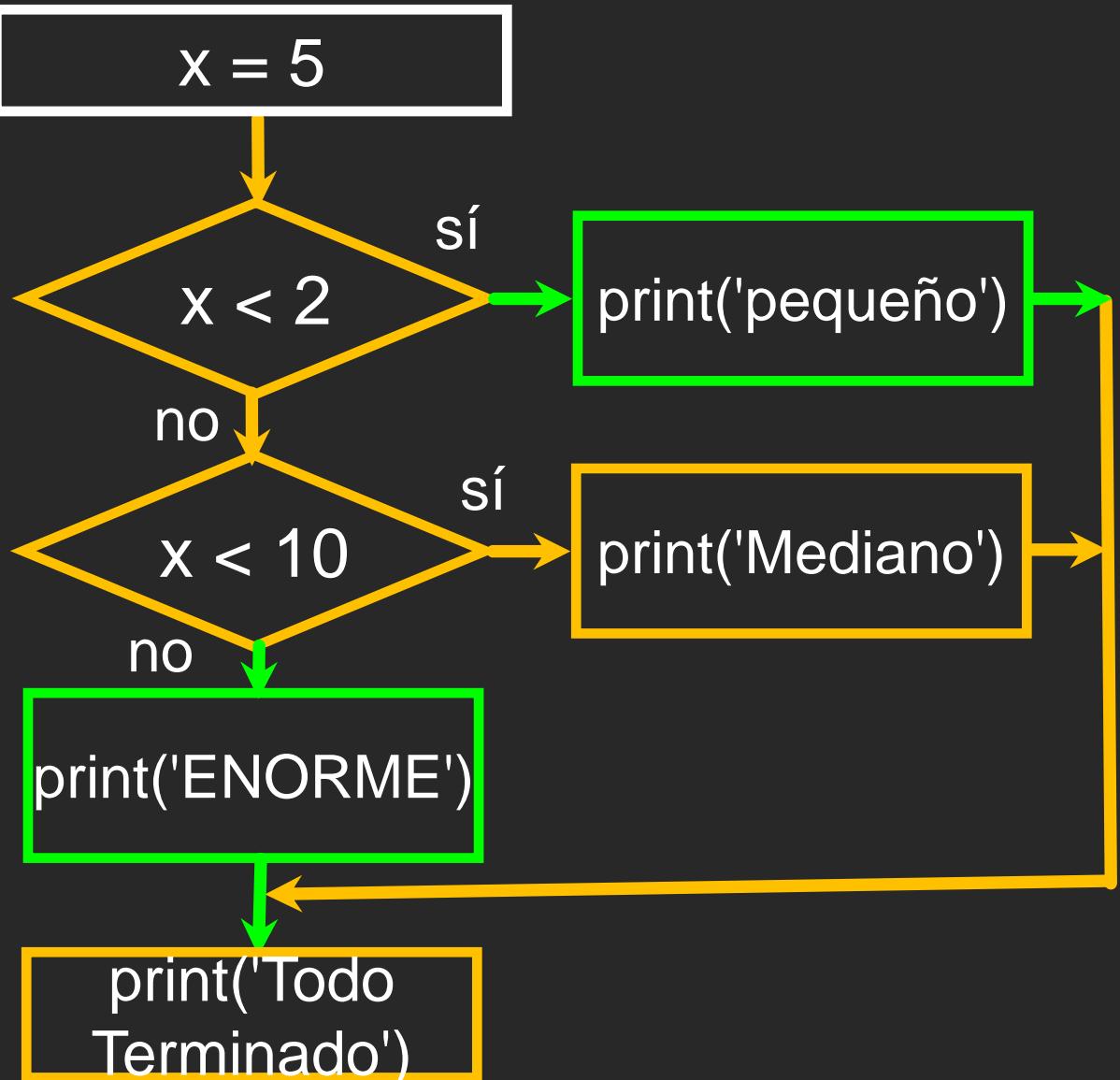
# Multidireccional

```
x = 0
if x < 2 :
    print('pequeño')
elif x < 10 :
    print('Mediano')
else :
    print('ENORME')
print('Todo
terminado')
```



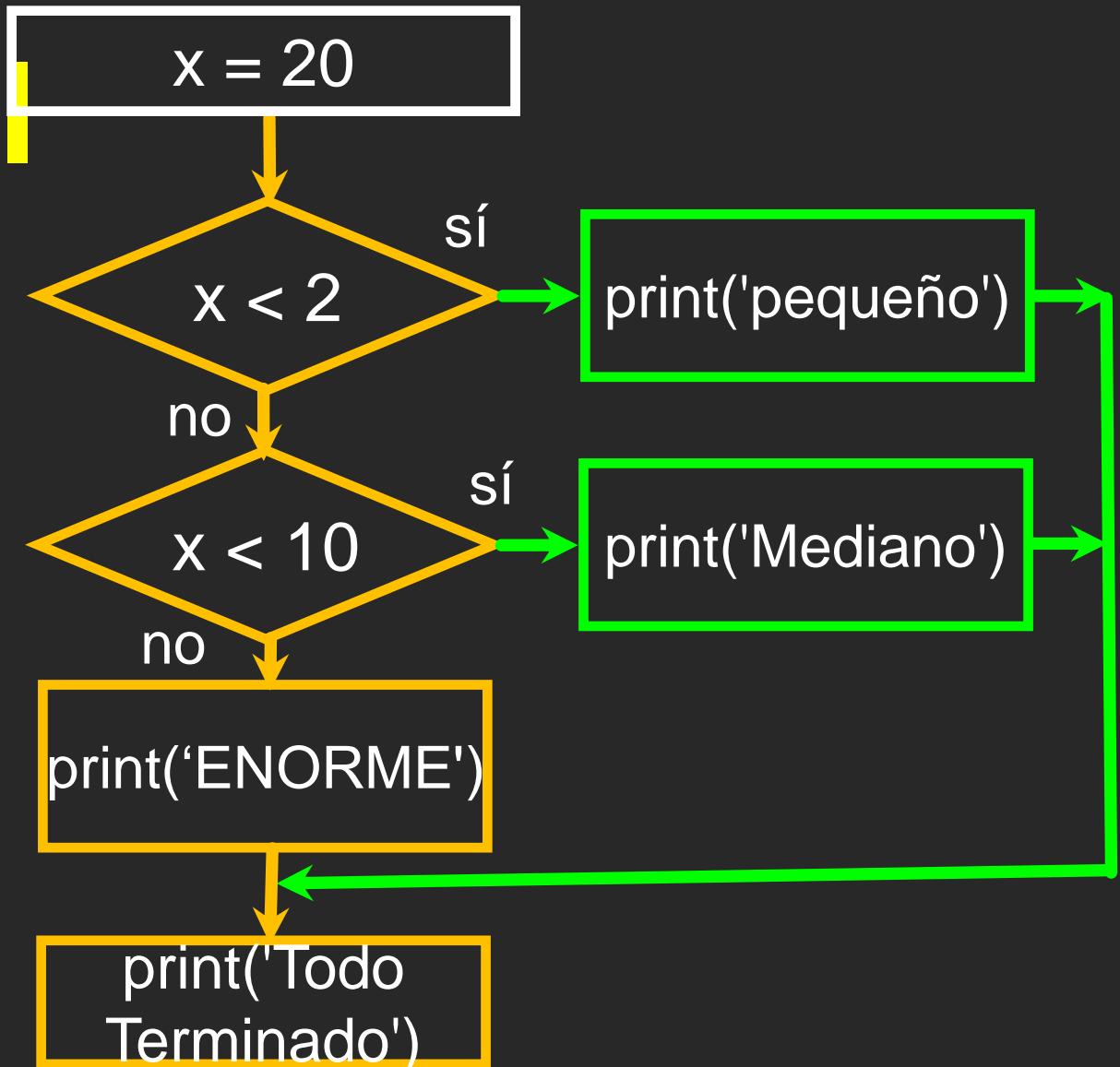
# Multidireccional

```
x = 5
if x < 2 :
    print('pequeño')
elif x < 10 :
    print('Mediano')
else :
    print('ENORME')
print('Todo
terminado')
```



# Multidireccional

```
x = 20
if x < 2 :
    print('pequeño')
elif x < 10 :
    print('Mediano')
else :
    print('ENORME')
print('Todo
terminado ')
```



# Multidireccional

```
# No Else
x = 5
if x < 2 :
    print('Pequeño')
elif x < 10 :
    print('Mediano')
print 'Todo terminado'
```

```
if x < 2 :
    print('pequeño')
elif x < 10 :
    print('Mediano')
elif x < 20 :
    print('Grande')
elif x < 40 :
    print('Enorme')
elif x < 100:
    print('Gigante')
else :
    print('Descomunal')
```

# Enigmas Multidireccionales

¿Cuál es el que nunca se imprimirá independientemente del valor de x?

```
if x < 2 :  
    print('Deabajo de 2')  
elif x >= 2 :  
    print('Dos o más')  
else :  
    print('Otro')
```

```
if x < 2 :  
    print('Deabajo de 2')  
elif x < 20 :  
    print('Deabajo de 20')  
elif x < 10 :  
    print('Deabajo de 10')  
else :  
    print('Otro')
```

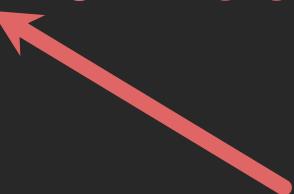
# La Estructura try / except

- Rodea una sección peligrosa de código con `try` y `except`
- Si el código en `try` funciona – `except` es omitido
- Si el código en `try` falla – pasa a la sección `except`

```
$ cat notry.py
astr = 'Hola Bob'
istr = int(astr)
print('Primero', istr)
astr = '123'
istr = int(astr)
print('Segundo', istr)
```

\$ python3 notry.py  
Trazo de rastreo (llamada más reciente a la último): Archivo "notry.py", línea 2, in <module> istr = int(astr) ValueError: invalid literal for int() with base 10: 'Hola Bob'

Todo  
Terminado



El  
programa  
se detiene  
aquí

```
$ cat notry.py
astr = 'Hola Bob'
istr = int(astr)
```



```
$ python3 notry.py
Trazas de rastreo (llamada más
reciente a lo último): Archivo
"notry.py", línea 2, in <module> istr
= int(astr)ValueError: invalid literal for
int() with base 10: 'Hola Bob'
```

Todo  
Terminado

```
astr = 'Hola Bob'  
try:  
    istr = int(astr)  
except:  
    istr = -1  
  
print('Primero', istr)  
  
astr = '123'  
try:  
    istr = int(astr)  
except:  
    istr = -1  
  
print('Segundo', istr)
```

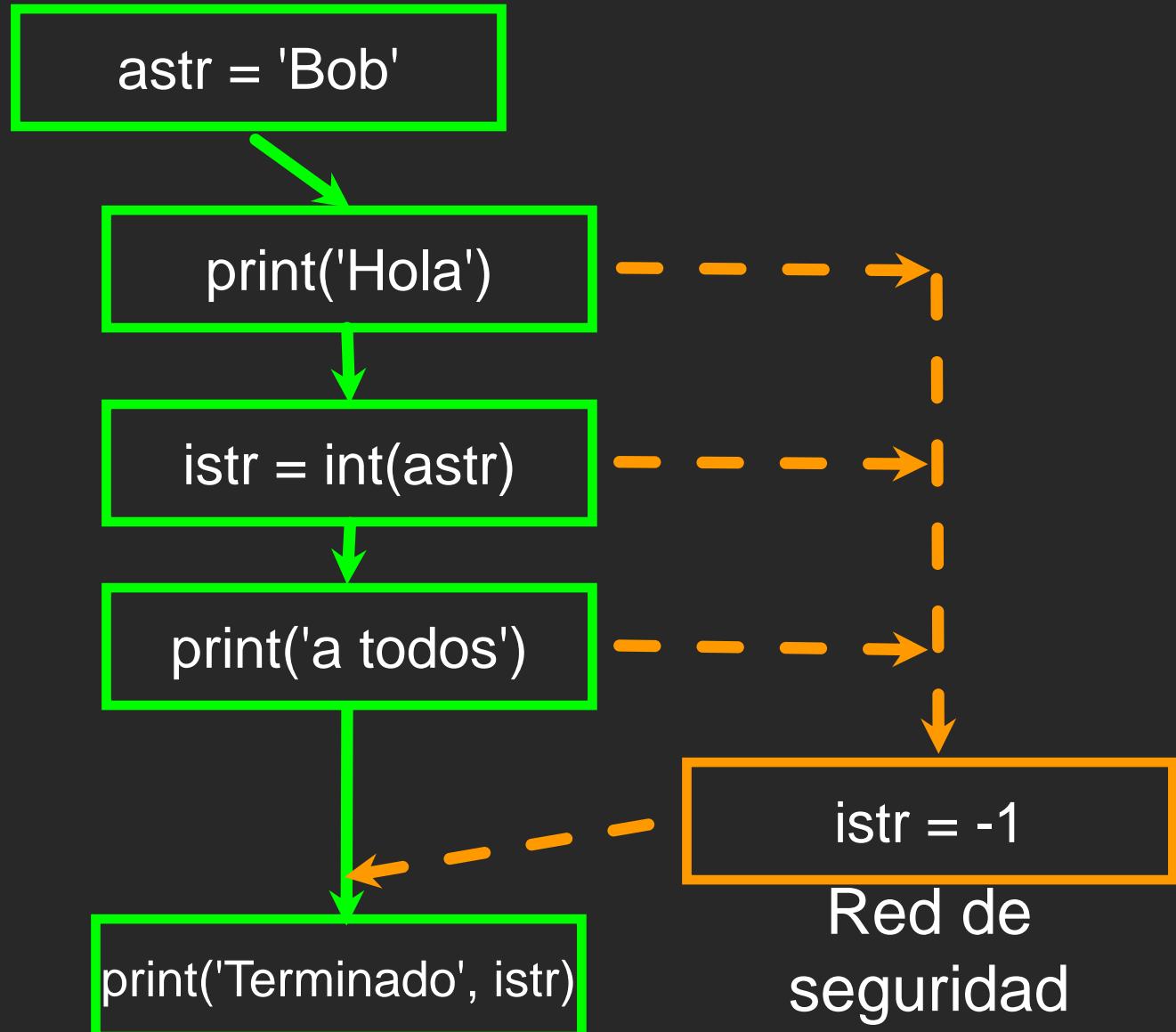
Cuando la primera conversión falla  
– simplemente cae en except  
(excepción): clausula, y el  
programa continúa.

```
$ python tryexcept.py  
Primero -1  
Segundo 123
```

Cuando la segunda conversión es  
exitosa – solo omite except  
(excepción): clausula, y el  
programa continúa.

# try / except

```
astr = 'Bob'  
try:  
    print('Hola')  
    istr = int(astr)  
    print('a todos')  
except:  
    istr = -1  
  
print('Terminado',  
      istr)
```



# Ejemplo de try / except

```
rawstr = input('Ingresar un número: ')
try:
    ival = int(rawstr)
except:
    ival = -1

if ival > 0 :
    print('Buen trabajo')
else:
    print('No es un número')
```

```
$ python3 trynum.py
Ingresar un número:42
Buen trabajo
$ python3 trynum.py
Ingresar un
número:cuarenta-y-dos
No es un número
$
```

## Ejercicio

Reescribe tu cálculo del salario para darle al empleado 1,5 veces la tarifa por hora para las horas trabajadas que excedan de las 8 horas diarias.

## Ejercicio

Reescribe tu programa de salarios usando try y except de modo que su programa maneje input (entradas) no numéricas de forma correcta.

Ingresar Horas: 20

Ingresar Tarifa: nueve

Error, por favor, ingresar un valor numérico

Ingresar Horas: cuarenta

Error, por favor, ingresar un valor numérico

*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.5. Fundamentos de programación en Python



# Funciones en Python

- En Python una **función** es un código reutilizable que toma **argumento(s)** como entrada, realiza algunos cálculos y luego devuelve uno o más resultados
- Para definir una **función** utilizamos la palabra reservada **def**
- Llamamos/Invocamos a la **función** utilizando una expresión que contenga el nombre de la función, paréntesis y los **argumentos**

# Funciones de Python

- Existen dos tipos de funciones:
  - **Funciones incorporadas (*built-in*)** que se presentan como parte del lenguaje: `print()`, `input()`, `type()`, `float()`, `int()` ...
  - **Funciones definidas por nosotros** que luego utilizamos

A diagram illustrating a Python assignment statement with annotations:

```
mayor = max('Hola mundo')
```

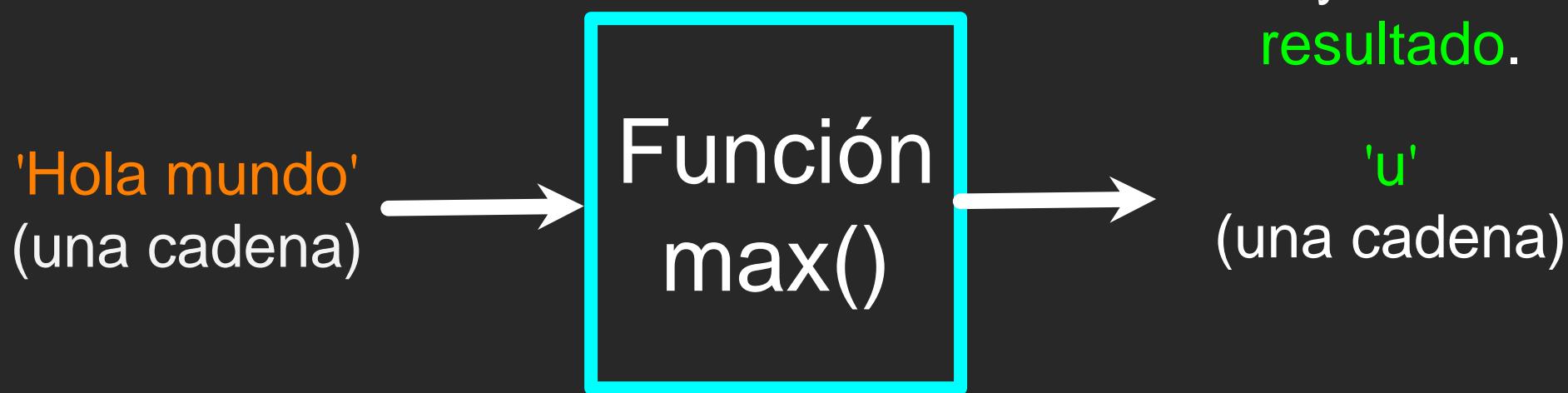
- An arrow labeled "Asignación" (Assignment) points to the assignment operator (=).
- An arrow labeled "Resultado" (Result) points to the character 'u'.
- An arrow labeled "Argumento" (Argument) points to the string 'Hola mundo'.

```
>>> mayor = max('Hola mundo')
>>> print(mayor)
'u'
>>> menor = min('Hola mundo')
>>> print(menor)
' '
>>>
```

# Función max()

```
>>> mayor = max('Hola mundo')  
>>> print(mayor)  
u
```

Una función es un código almacenado que podemos reutilizar. Una función toma una entrada y devuelve un resultado.



Función incorporada

# Funciones definidas por nosotros

- Creamos una nueva función usando la palabra clave **def** seguida de parámetros opcionales entre paréntesis
- Indentamos el cuerpo de la función
- Esto **define** la función pero **no** ejecuta el cuerpo de la función

```
def mostrar_letra():
    print("You may say I'm a dreamer")
    print('But I'm not the only one')
```

```
x = 5
```

```
print('Antes')
```

```
def mostrar_letra():
```

```
    print("You may say I'm a dreamer")
```

```
    print('But I'm not the only one')
```

```
print('Después')
```

```
x = x + 2
```

```
print(x)
```

mostrar\_letra():

```
print "You may say I'm a dreamer"  
print 'But I'm not the only one'
```

Antes  
Después

7

# Definición y uso de la función

- Una vez que hemos **definido** una función, podemos **llamarla** (o **invocarla**) todas las veces que queramos
- Se trata de un patrón de uso para **almacenar** y **reutilizar** código

```
x = 5
print('Antes')

def mostrar_letra():
    print("You may say I'm a dreamer")
    print('But I'm not the only one')

print('Después')
mostrar_letra()
x = x + 2
print(x)
```

Antes  
Después  
You may say I'm a dreamer  
But I'm not the only one

# Argumentos

- Un **argumento** es un valor que pasamos a la **función** como su **entrada** cuando llamamos a la función
- Utilizamos **argumentos** para poder instruir a la **función** que realice diferentes tareas cuando la llamamos **diferentes** veces
- Colocamos los **argumentos** entre paréntesis después del **nombre** de la función

```
mayor = max('Hola mundo')
```



Argumento

# Parámetros

Un parámetro es una variable que utilizamos en la definición de la función. Es un manejador (*handler*) que permite al código de la función acceder a los argumentos con los que se ha invocado una función en particular.

```
>>> def saludar(lang):
...     if lang == 'es':
...         print('Hola')
...     elif lang == 'fr':
...         print('Bonjour')
...     else:
...         print('Hello')
...
>>> saludar('en')
Hello
>>> saludar('es')
Hola
>>> saludar('fr')
Bonjour
>>>
```

# Valores de retorno

- Normalmente, una función tomará los argumentos, hará algunos cálculos, y **retornará** un valor que se usará como el valor de la llamada de la función en la **expresión de llamada**. La palabra clave **return** se utiliza para esto.
- El enunciado **return** termina la ejecución de la **función** y “devuelve” el **resultado** de la **función**

```
>>> def saludar(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print(saludar('en'), 'Pablo')
Hello Pablo
>>> print(saludar('es'), 'Sally')
Hola Sally
>>> print(saludar('fr'), 'Michael')
Bonjour Michael
>>>
```

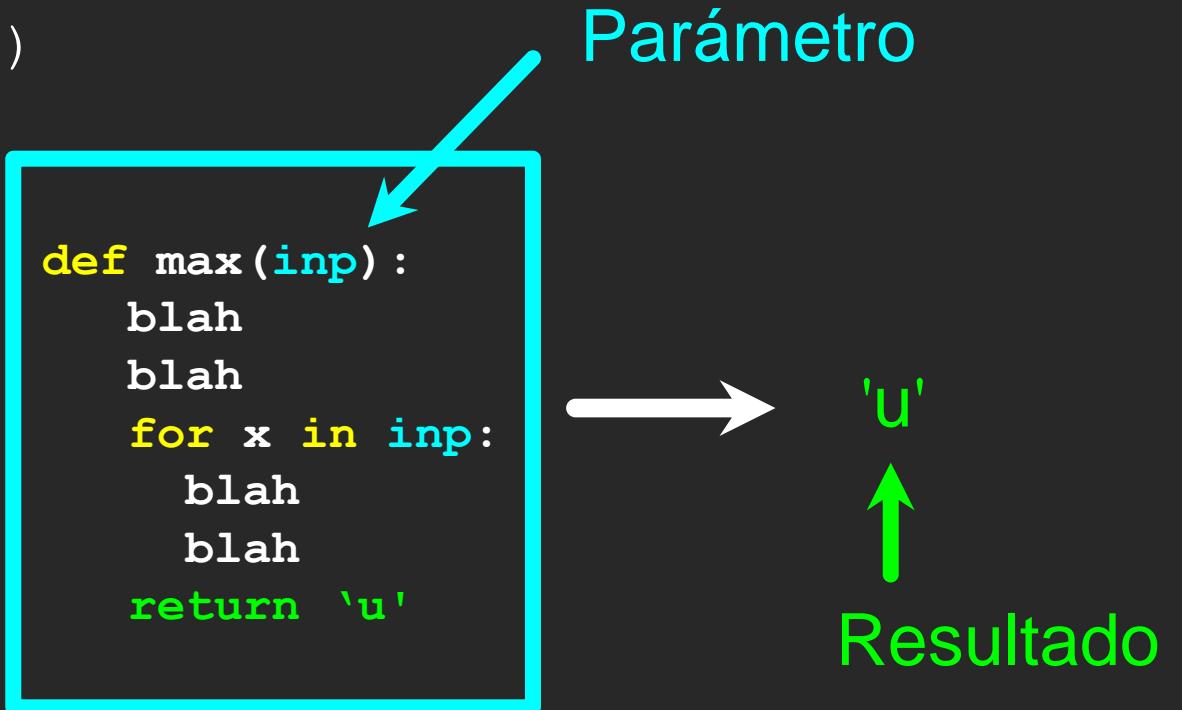
# Argumentos, parámetros y resultados

```
>>> mayor = max('Hola mundo')
>>> print(mayor)
```

u

'Hola mundo'

Argumento



# Múltiples parámetros / argumentos

- Podemos definir más de un **parámetro** en la **definición** de la función
- Simplemente agregamos más **argumentos** cuando llamamos a la función
- Haremos coincidir el número y orden de los argumentos y parámetros

```
def sumar(a, b):  
    resultado = a + b  
    return resultado
```

```
x = sumar(3, 5)  
print(x)
```

# Uso de funciones

- Organiza el código en funciones: captura una funcionalidad concreta y ponle un nombre
- No repitas código, crea una función y luego reutilízala
- Si algo se vuelve demasiado largo o complejo, desglósalo en varios bloques lógicos y coloca esos bloques en funciones
- En programas complejos, haz una biblioteca de funciones comunes que se puedan repetir y reutilizar a lo largo del tiempo

## Ejercicio

Reescribe el programa de cálculo de salario con horas extras creando una función llamada `calcular_salario()` que recibe dos parámetros (horas y tarifa).

*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

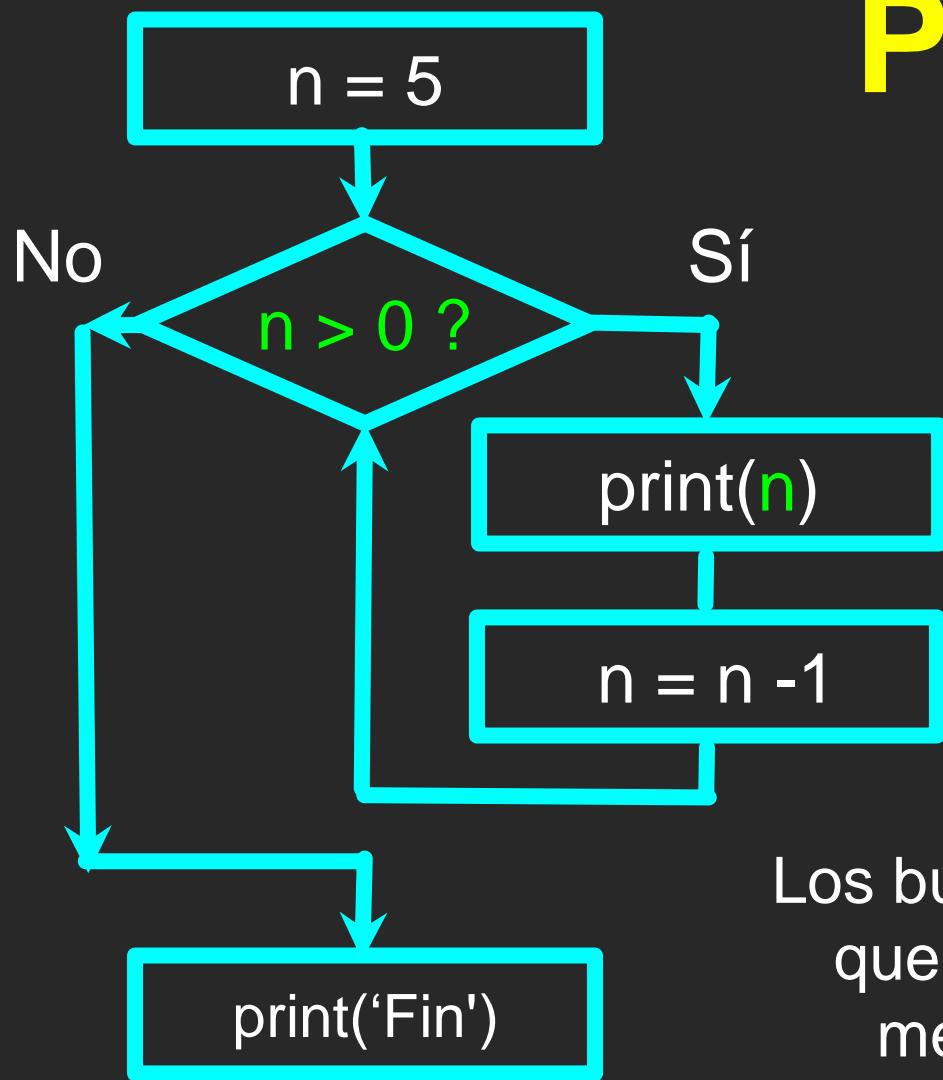
*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.6. Fundamentos de programación en Python



# Pasos repetidos



Programa:

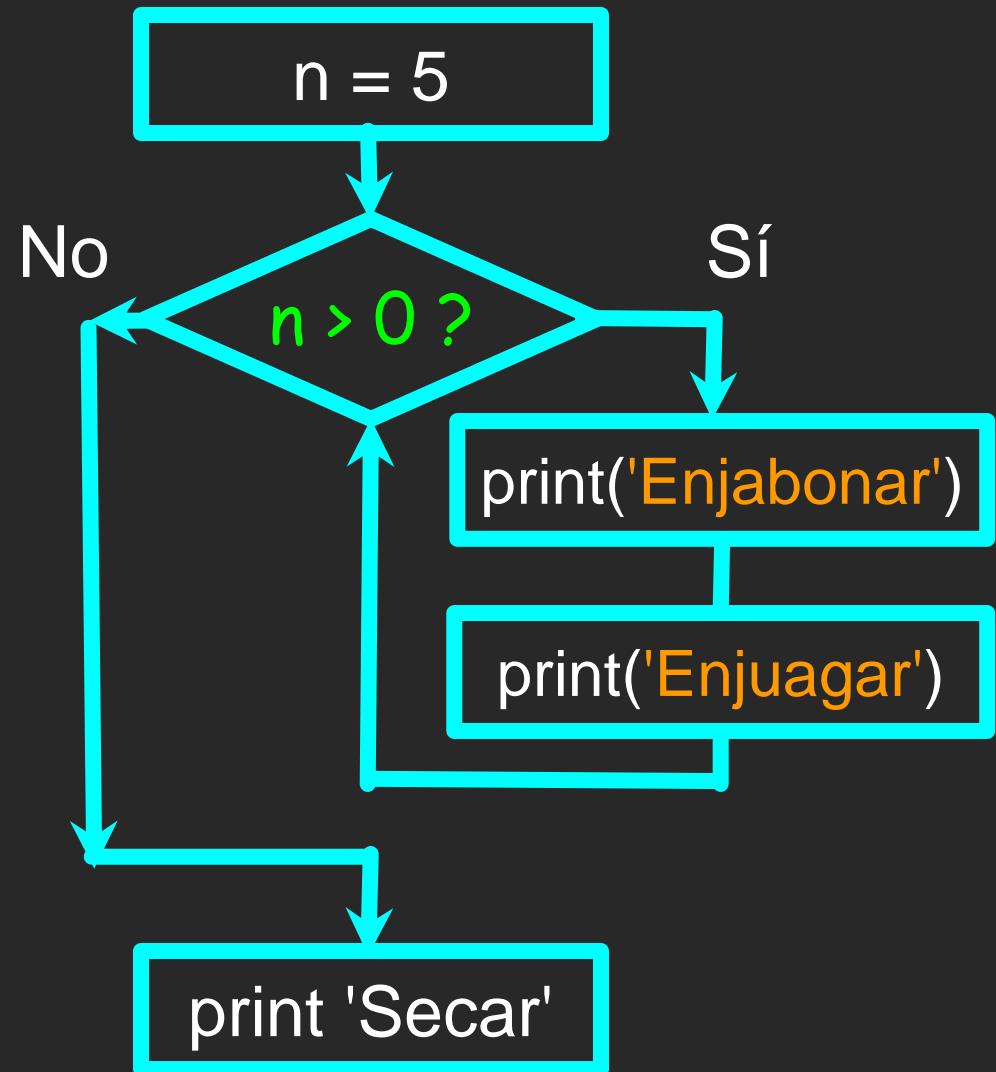
```
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Fin')
print(n)
```

Resultado:

5  
4  
3  
2  
1  
Fin  
0

Los bucles (pasos repetidos) tienen **variables de iteración** que cambian de valor en cada ejecución del bucle. A menudo, estas **variables de iteración** recorren una secuencia de números.

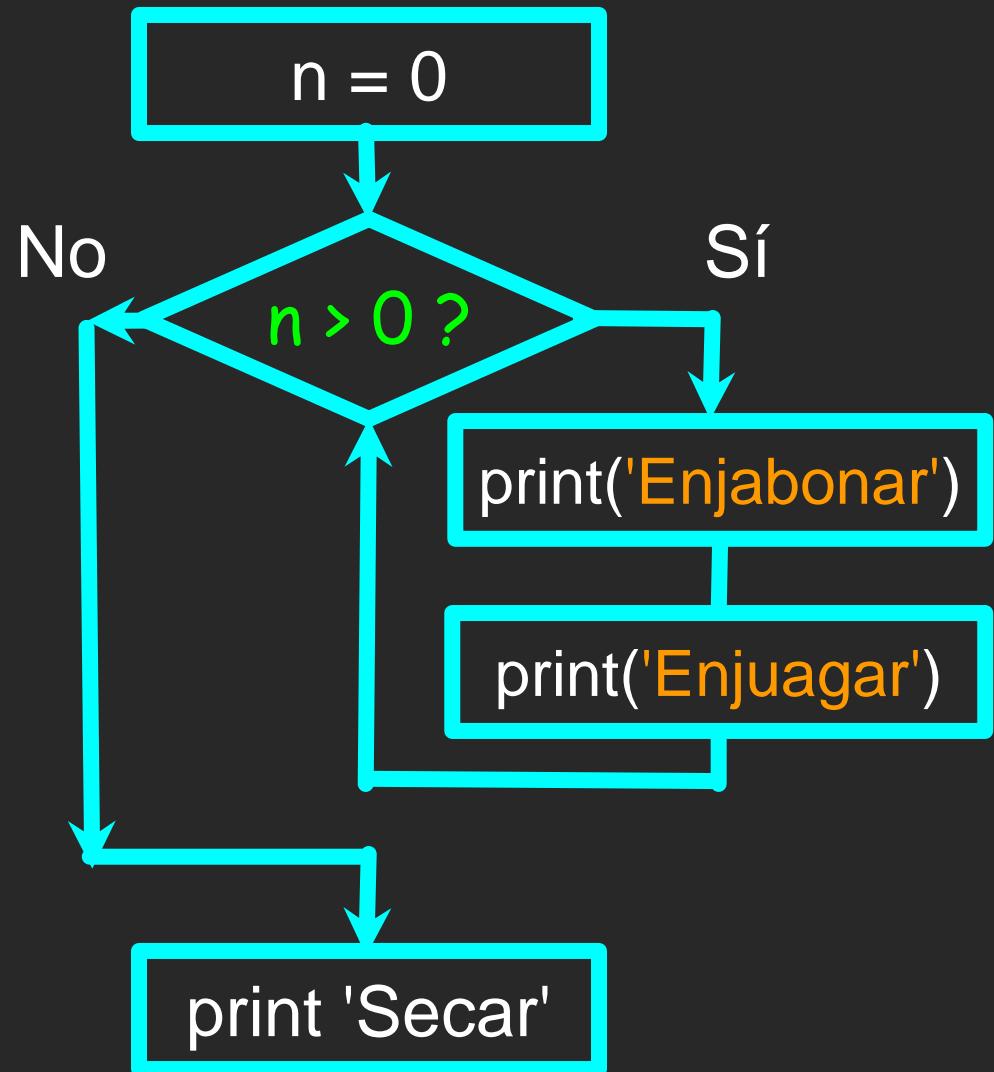
# Bucles infinitos



```
n = 5  
while n > 0 :  
    print('Enjabonar')  
    print('Enjuagar')  
    print('Secar')
```

¿Qué es lo que está mal en este bucle?

# Otro bucle



```
n = 0  
while n > 0 :  
    print('Enjabonar')  
    print('Enjuagar')  
    print('Secar')
```

¿Qué es lo que está haciendo este bucle?

# Romper un bucle

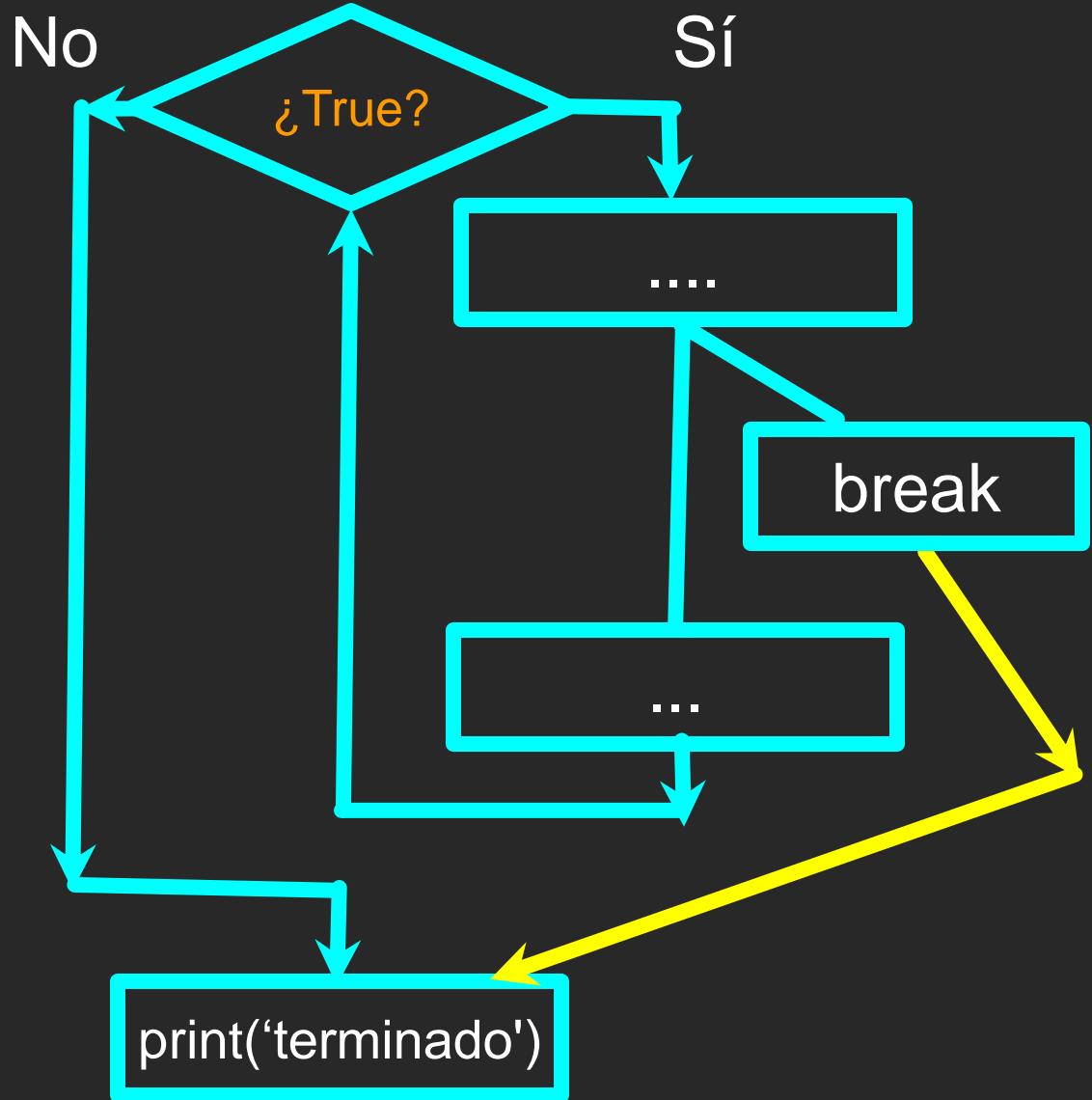
- La palabra reservada **break** (romper) termina el bucle actual y salta al bloque que sigue inmediatamente después del bucle

```
while True:  
    linea = input('> ')  
    if linea == 'terminado':  
        break  
    print(linea)  
print('terminado')
```



> hola
hola
> finalizado
finalizado
> terminado
terminado

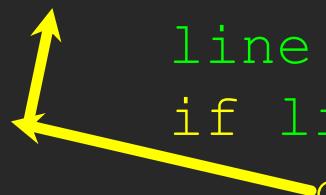
```
while True:  
    linea = input('> ')  
    if linea == 'terminado' :  
        break  
    print(linea)  
print('terminado')
```



# Finalizar una iteración con continue

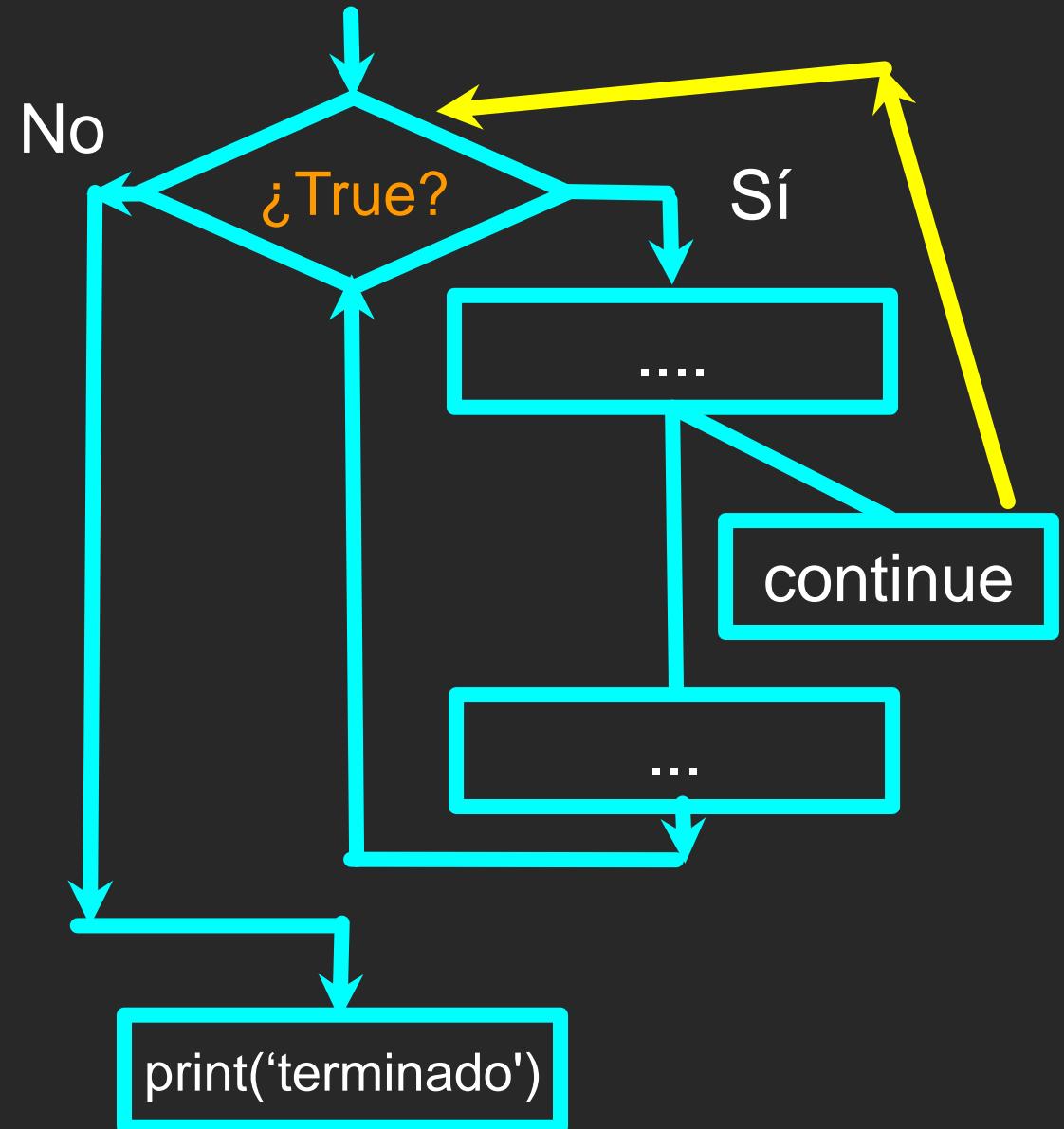
La palabra reservada `continue` (continuar) termina la iteración actual y salta a la parte superior del bucle, comenzando la siguiente iteración

```
while True:  
    line = input('> ')  
    if line[0] == '#':  
        continue  
    if line == 'terminado':  
        break  
    print(line)  
print('terminado')
```



> hola  
Hola  
> # no imprimir esto  
> Imprimir esto  
imprimir esto  
> terminado  
terminado

```
while True:  
    linea = raw_input('> ')  
    if linea[0] == '#':  
        continue  
    if linea == 'terminado':  
        break  
    print(linea)  
print('terminado')
```



# Bucles indefinidos

- Los bucles while se llaman **bucles indefinidos** porque continúan hasta que una condición lógica se vuelve **False**
- Los bucles vistos hasta ahora son bastante fáciles de examinar para determinar si terminarán o si serán bucles infinitos
- Muchas veces, es difícil saber con seguridad si un bucle terminará

# Bucles definidos

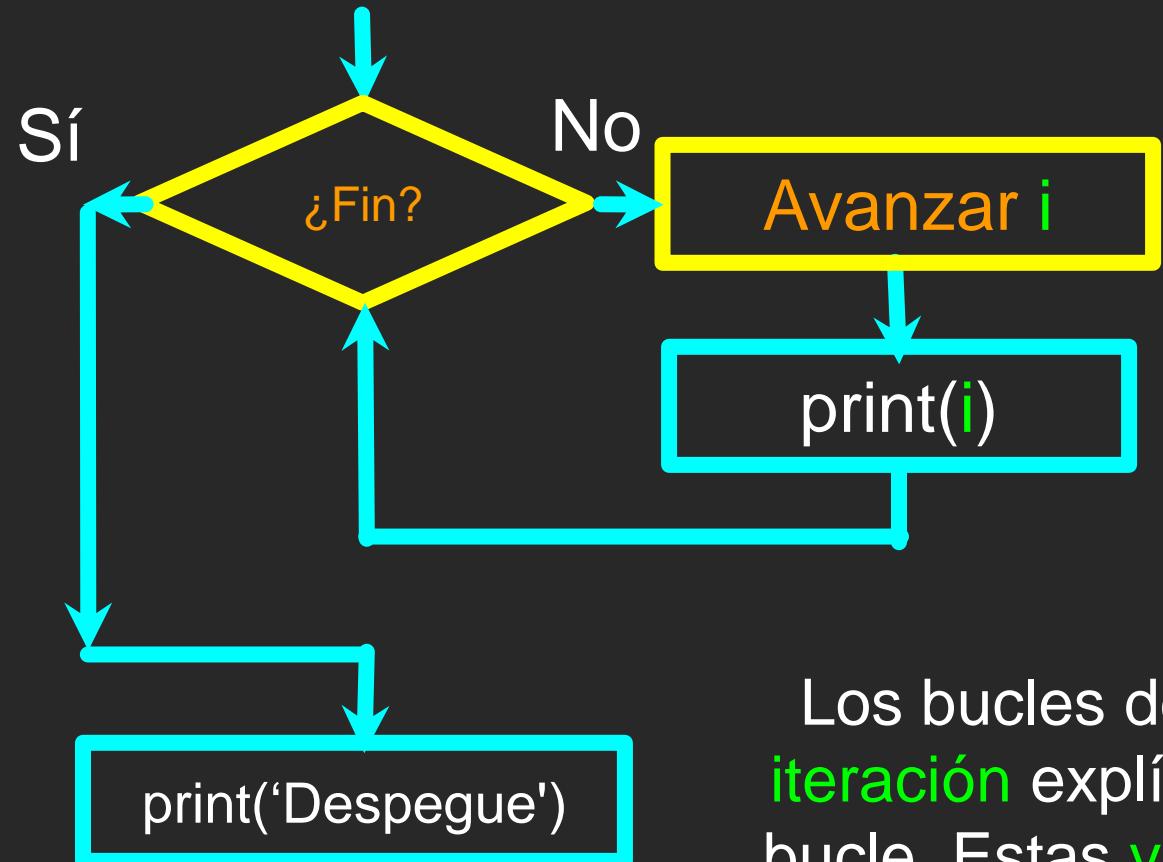
- Con bastante frecuencia tenemos una **lista** de ítems (p. ej. las **líneas en un archivo**), es decir, un **conjunto finito** de cosas
- Podemos escribir un bucle para ejecutarlo por cada uno de los ítems de un conjunto utilizando la secuencia **for** de Python
- Estos bucles se denominan **bucles definidos** porque se ejecutan un número exacto de veces
- Decimos que los **bucles definidos iteran a través de los miembros de un conjunto**

# Un bucle definido simple

```
for i in [5, 4, 3, 2, 1] :  
    print(i)  
print('Despegue')
```

5  
4  
3  
2  
1  
Despegue

# Un bucle definido simple



```
for i in [5, 4, 3, 2, 1] :  
    print(i)  
print('Despegue')
```

5  
4  
3  
2  
1  
Despegue

Los bucles definidos (bucles **for**) tienen **variables de iteración** explícitas que cambian cada vez a través del bucle. Estas **variables de iteración** se mueven a través del conjunto o secuencia.

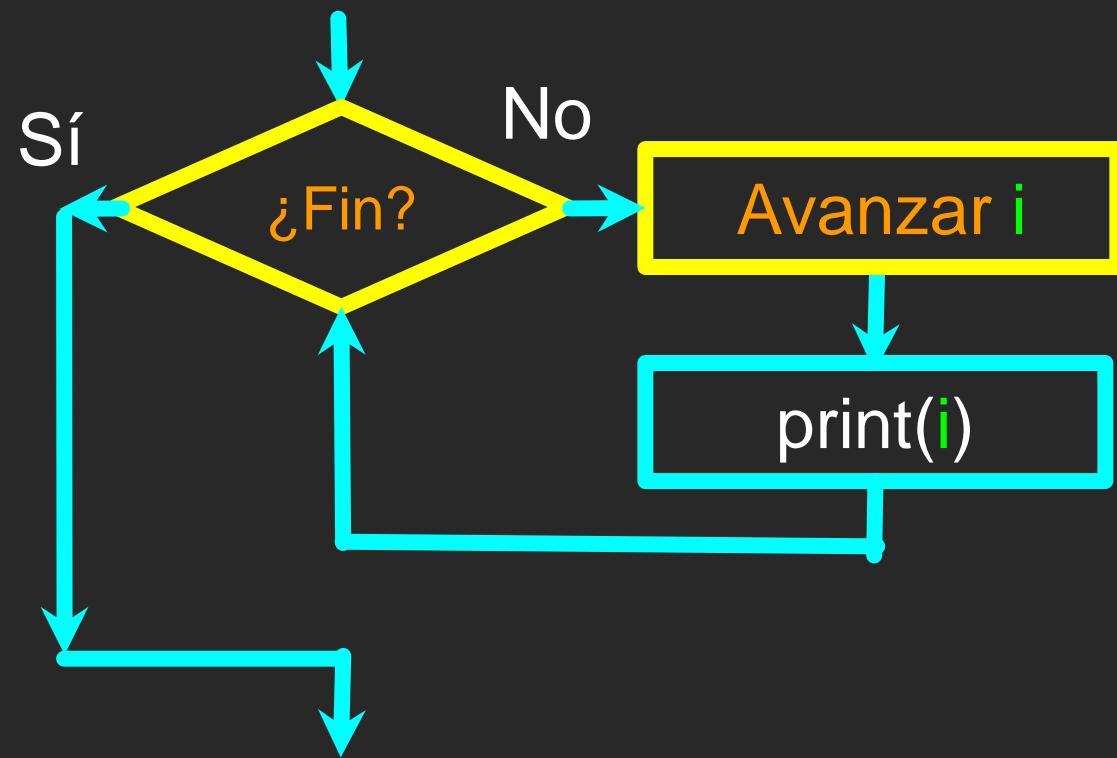
# for... in...

- La **variable de iteración** itera a través de la **secuencia** (conjunto ordenado), es decir, la recorre
- El **bloque** de código se ejecuta una vez para cada valor en (**in**) la **secuencia**
- La **variable de iteración** va tomando todos los valores en (**in**) la **secuencia**

Variable de  
iteración

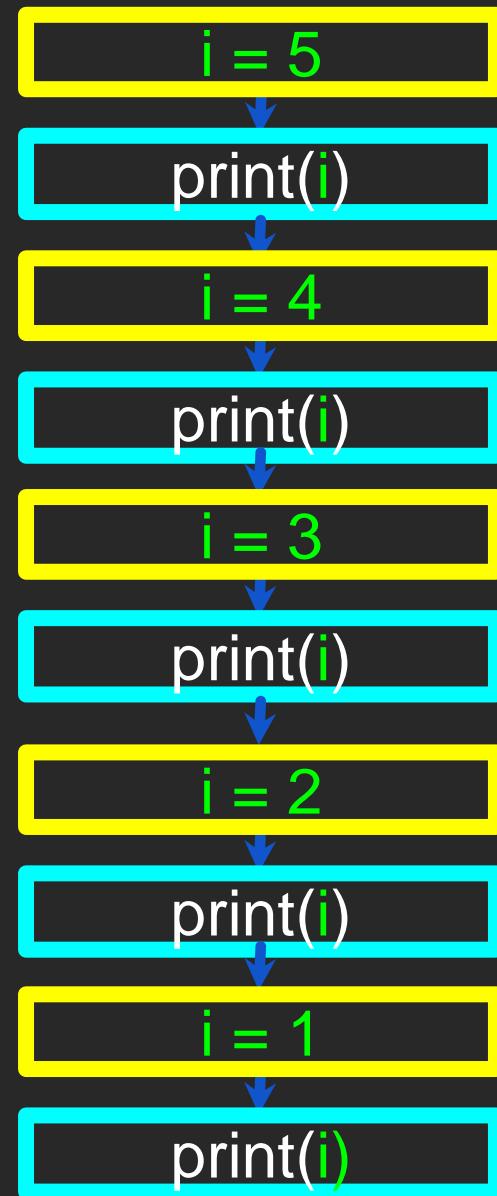
```
for i in [5, 4, 3, 2, 1] :  
    print(i)
```

Secuencia de cinco  
elementos



```

for i in [5, 4, 3, 2, 1] :
    print(i)
    
```



# Un bucle definido con cadenas

```
amigos = ['Natalia', 'Lucía', 'Javi']
for amigo in amigos :
    print('Feliz año nuevo:', amigo)
print('Terminado')
```

Feliz año nuevo: Natalia

Feliz año nuevo: Lucía

Feliz año nuevo: Javi

Terminado

# Iteración de un conjunto

```
objetos = [9, 41, 12, 3, 74, 15]
print('Antes')
for objeto in objetos:
    print(objeto)
print('Después')
```

\$ python basicloop.py

Antes

9

41

12

3

74

15

Después

# Búsqueda del mayor valor

```
mayor_valor = -1
print('Antes', mayor_valor)

for num in [9, 41, 12, 3, 74, 15] :
    if num > mayor_valor :
        mayor_valor = num
    print(mayor_valor, num)

print('Después', mayor_valor)
```

```
$ python largest.py
Antes -1
9 9
41 41
41 12
41 3
74 74
74 15
Después 74
```

Creamos una variable que contenga el mayor valor que se haya visto hasta el momento (mayor\_valor). Si el número actual que estamos comprobando es más grande, entonces será el nuevo mayor valor que se haya visto hasta el momento (mayor\_valor).

# Búsqueda del menor valor

```
menor_valor = None
print('Antes')
for valor in [9, 41, 12, 3, 74, 15] :
    if menor_valor is None:
        menor_valor = valor
    elif valor < menor_valor :
        menor_valor = valor
    print(menor_valor , valor)
print('Después', menor_valor )
```

```
$ python smallest.py
Antes
9 9
9 41
9 12
3 3
3 74
3 15
Después 3
```

Tenemos una variable que es el menor valor (`menor_valor`) hasta ahora. La primera vez en el bucle `menor_valor` es `None`, por lo que tomamos el primer `valor` como `menor valor`.

# Conteo en un bucle

```
contador = 0
print('Antes', contador)
for objeto in [9, 41, 12, 3, 74, 15] :
    contador += 1
    print(contador, objeto)
print('Después', contador)
```

```
$ python countloop.py
Antes 0
1 9
2 41
3 12
4 3
5 74
6 15
Después 6
```

Para **contar** cuántas veces ejecutamos un bucle, introducimos una **variable de conteo** que **comience en 0** y le sumamos **uno en cada iteración del bucle**.

# Suma en un bucle

```
suma = 0
print('Antes', suma)
for objeto in [9, 41, 12, 3, 74, 15] :
    suma = suma + objeto
    print(suma, objeto)
print('Después', suma)
```

```
$ python countloop.py
Antes 0
9 9
50 41
62 12
65 3
139 74
154 15
Después 154
```

Para **sumar** un **valor** que encontramos en un bucle, introducimos una **variable de suma** que **comience en 0** y le **sumamos el valor** a la suma cada vez a través del bucle.

# Obtener la media con un bucle

```
cont = 0
suma = 0
print('Antes', cont, suma)
for valor in [9, 41, 12, 3, 74, 15] :
    cont += 1
    suma = suma + valor
    print(cont, suma, valor)
print('Después', cont, suma, suma / cont)
```

```
$ python averageloop.py
Antes 0 0
1 9 9
2 50 41
3 62 12
4 65 3
5 139 74
6 154 15
Después 6 154 25
```

Una media solo combina los patrones de conteo y suma en un bucle y divide cuando el bucle ha terminado.

# Filtrar en un bucle

```
print('Antes')
for valor in [9, 41, 12, 3, 74, 15] :
    if valor > 20:
        print('Mayor que 20:', valor)
print('Después')
```

```
$ python search1.py
Antes
Mayor que 20: 41
Mayor que 20: 74
Después
```

Utilizamos el condicional “if” en el **bucle** para filtrar los valores que estamos buscando.

# Búsqueda mediante bandera o flag

```
encontrado = False
print('Buscamos un 3')
print('Antes', encontrado)
for valor in [9, 41, 12, 3, 74, 15] :
    if valor == 3 :
        encontrado = True
    print(encontrado, valor)
print('Después', encontrado)
```

```
$ python search1.py
Buscamos un 3
Antes False
False 9
False 41
False 12
True 3
True 74
True 15
Después True
```

Si solo queremos buscar y saber si un valor fue hallado o no (`encontrado`), utilizamos una **variable** que comience como **False** (Falsa) y se vuelva **True** (Verdadera) tan pronto como `encontremos` (`encontrado`) lo que estamos buscando.

# Los operadores “is” e “is not”

```
menor_valor = None
print('Antes')
for valor in [9, 41, 12, 3, 74, 15] :
    if menor_valor is None:
        menor_valor = valor
    elif valor < menor_valor :
        menor_valor = valor
    print(menor_valor , valor)
print('Después', menor_valor )
```

- Python tiene un operador lógico **is** que puede ser utilizado en expresiones lógicas
- Similar a **==** pero más fuerte
- **is not** también es un operador lógico, similar a **!=** pero más fuerte

# Los operadores “is” e “is not”

```
menor_valor = None
print('Antes')
for valor in [9, 41, 12, 3, 74, 15] :
    if menor_valor is None:
        menor_valor = valor
    elif valor < menor_valor :
        menor_valor = valor
    print(menor_valor , valor)
print('Después', menor_valor )
```

- El operador **is** devuelve verdadero cuando los dos objetos a comparar son el mismo.
- Por otro lado, el operador **==** devuelve verdadero si el contenido es el mismo, pero no necesariamente el objeto.
- Cuando hablamos de datos primitivos como enteros, reales o cadenas de texto, el comportamiento de los dos operadores es casi indistinguible

## Ejercicio

Implementa un programa que calcule la media de todos los números introducidos por el usuario hasta que introduzca “fin”. El programa ha de tener en cuenta posibles entradas erróneas del usuario (no numéricas).

*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.7. Fundamentos de programación en Python



# Repaso de strings

- Una cadena o string es una secuencia de caracteres.
- En Python se permiten comillas simples (p. ej. 'Hola') o dobles (p.ej. "Hola")
- En strings, + significa “concatenar”.
- Si una cadena contiene números, sigue siendo una cadena.
- Podemos convertir números dentro de una cadena, a enteros, utilizando int()

```
>>> str1 = "Buenos"  
>>> str2 = 'días'  
>>> cad = str1 + str2  
>>> print(cad)  
Buenosdías  
>>> str3 = '123'  
>>> str3 = str3 + 1  
Traceback (most recent call  
last): File "<stdin>", line 1,  
in <module>  
TypeError: cannot concatenate  
'str' and 'int' objects  
>>> x = int(str3) + 1  
>>> print(x)  
124  
>>>
```

# Leyendo y convirtiendo datos

- Leemos datos de entrada utilizando **strings o cadenas** y después se convierten de ser necesario
- Esto proporciona un mayor control sobre situaciones de error y/o datos de entrada del usuario erróneos
- Los números como datos de entrada deben ser **convertidos** de cadenas a enteros

```
>>> nombre = input('Escribe:')
Escribe:Juan
>>> print(nombre)
Juan
>>> numero = input('Escribe:')
Escribe:100
>>> x = numero - 10
Traceback (most recent call
last): File "<stdin>", line 1,
in <module>
TypeError: unsupported operand
type(s) for -: 'str' and 'int'
>>> x = int(numero) - 10
>>> print(x)
```

# Caracteres en cadenas

- Podemos obtener cualquier carácter en una cadena usando un índice especificado en **corchetes**
- El valor del índice es un entero a partir del cero (primera posición)
- El valor del índice puede ser una expresión

b	a	n	a	n	a
0	1	2	3	4	5

```
>>> fruta = 'banana'  
>>> letra = fruta[1]  
>>> print(letra)  
a  
>>> x = 3  
>>> w = fruta[x - 1]  
>>> print(w)  
n
```

# Caracteres en cadenas

- Obtendremos un **error de Python** si tratamos de acceder a un índice más allá del final de la cadena.

```
>>> cad = 'abc'  
>>> print(cad[5])  
Traceback (most recent call  
last): File "<stdin>", line  
1, in <module>  
IndexError: string index out  
of range  
>>>
```

# Tamaño de cadenas

La función nativa `len()` nos devuelve el tamaño de una cadena



```
>>> fruta = 'banana'  
>>> print(len(fruta))  
6
```

# Recorriendo una cadena

Utilizando una sentencia **while**, una **variable de iteración**, y la función **len()**, podemos construir un bucle para explorar de manera individual cada uno de los caracteres de una cadena

```
fruta = 'banana'  
i = 0  
while i < len(fruta):  
    letra = fruta[i]  
    print(i, letra)  
    i = i + 1
```

0	b
1	a
2	n
3	a
4	n
5	a

# Recorriendo una cadena

- Un bucle definido utilizando una sentencia `for...in...` es más elegante
- La **variable de iteración** es gestionada completamente por el bucle `for...in...`

```
fruta = 'banana'  
for letra in fruta:  
    print(letra)
```

b  
a  
n  
a  
n  
a

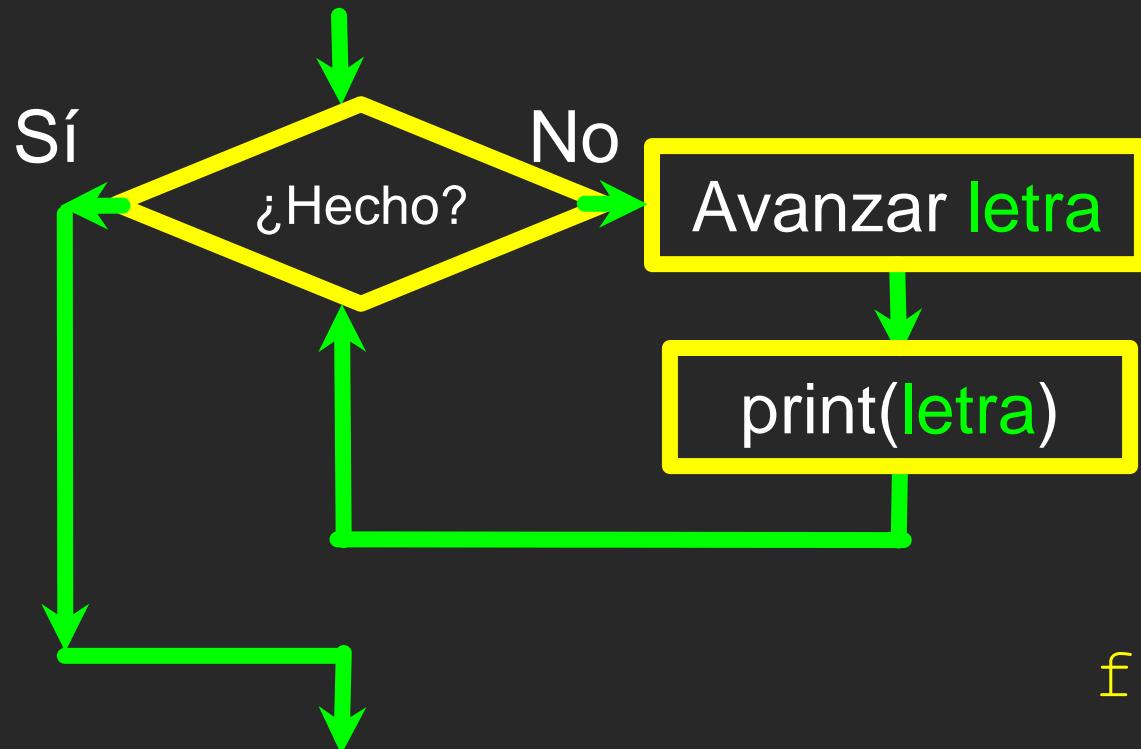
# for... in... en detalle

- La **variable de iteración** “itera” a través de una **secuencia** (un conjunto ordenado)
- El **bloque (cuerpo)** de código es ejecutado una vez para cada valor **en (in)** la **secuencia**
- La **variable de iteración** se mueve a través de todos los valores **en (in)** la **secuencia**

Variable de  
iteración

```
for letra in 'banana' :  
    print(letra)
```

Cadena de seis  
caracteres



```
for letra in 'banana' :
    print(letra)
```

La **variable de iteración** “*letra*” a través de la **cadena** y el **bloque** (**cuerpo**) de código es ejecutado para cada valor **en (in)** la secuencia

# Recorriendo y contando

Bucle prototípico que itera a través de una cadena y cuenta el número de apariciones de un carácter

```
palabra = 'banana'  
contador = 0  
for letra in palabra :  
    if letra == 'a' :  
        contador = contador + 1  
print(contador)
```

# Seccionado de cadenas



- También podemos seccionar una cadena utilizando el **operador** :
- El segundo número implica una posición más del final de la sección
- Si el segundo número está más allá del final de la cadena, no causa error

```
>>> s = 'Monty Python'  
>>> print(s[0:4])  
Mont  
>>> print(s[6:7])  
P  
>>> print(s[6:20])  
Python
```

# Seccionado de cadenas

Si dejamos en blanco el primer o el segundo número de la sección, se asume que es el inicio o el final de la cadena, respectivamente



```
>>> s = 'Monty Python'  
>>> print(s[:2])  
Mo  
>>> print(s[8:])  
thon  
>>> print(s[:])  
Monty Python
```

# Concatenación de cadenas

Cuando el operador `+` se aplica a una cadena, significa “concatenación”

```
>>> a = 'Buenos'  
>>> b = a + 'días'  
>>> print(b)  
Buenosdías  
>>> c = a + ' ' + 'días'  
>>> print(c)  
Buenos días  
>>>
```

# Uso de `in` como operador lógico

- La palabra reservada `in` puede ser utilizada para revisar si una cadena se encuentra `en (in)` otra cadena
- La expresión `in` es una expresión lógica que retorna `True` o `False` y puede ser utilizada en sentencias `if`

```
>>> fruta = 'banana'  
>>> 'n' in fruta  
True  
>>> 'm' in fruta  
False  
>>> 'ana' in fruta  
True  
>>> if 'a' in fruta :  
...     print('Encontrada!')  
...  
Encontrada!  
>>>
```

# Comparación de cadenas

```
if palabra == 'banana':  
    print('Es una banana.')  
  
if palabra < 'banana':  
    print('Tu palabra,' + palabra + ', viene antes de banana.')  
elif palabra > 'banana':  
    print('Tu palabra,' + palabra + ', viene después de banana.')  
else:  
    print('Es una banana.')
```

# Librería String

- Python tiene numerosas **funciones sobre cadenas** dentro de la librería **string**
- Las invocamos agregando la función a la variable de cadena
- Dichas **funciones** no modifican la cadena original, sino que retornan una nueva cadena modificada

```
>>> saludo = 'Hola Mundo'  
>>> minusc = saludo.lower()  
>>> print(minusc)  
hola mundo  
>>> print(saludo)  
Hola Mundo  
>>> print('Bon Dia'.lower())  
bon dia  
>>>
```

```
>>> cad = 'Hola mundo'  
>>> type(cad)  
<class 'str'>  
>>> dir(cad)  
['capitalize', 'casefold', 'center', 'count', 'encode',  
'endswith', 'expandtabs', 'find', 'format', 'format_map',  
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',  
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',  
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',  
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',  
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',  
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',  
'zfill']
```

<https://docs.python.org/3/library/stdtypes.html#string-methods>

# Librería String

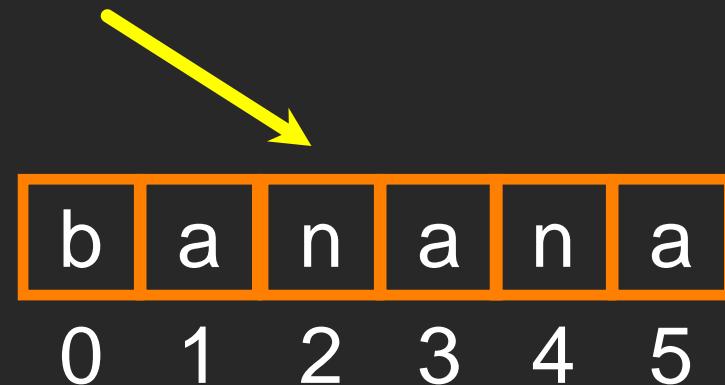
str.capitalize()	
str.center(width[, fillchar])	str.lower()
str.count(sub[, start[, end]])	str.upper()
str.startswith(prefix[, start[, end]])	str.swapcase()
str.endswith(suffix[, start[, end]])	str.lstrip([chars])
str.find(sub[, start[, end]])	str.rstrip([chars])
str.replace(old, new[, count])	str.strip([chars])

# Librería String

```
str.isalnum()  
str.isalpha()  
str.isdigit()  
str.islower()  
str.isupper()  
str.isspace()
```

# Buscando una cadena

- Utilizamos la función `find()` para buscar una subcadena dentro de otra cadena
- `find()` encuentra la primer ocurrencia de la subcadena
- Si la subcadena no se encuentra, `find()` devuelve `-1`
- Recuerda que las posiciones de una cadena comienzan en cero.



```
>>> fruta = 'banana'  
>>> pos = fruta.find('na')  
>>> print(pos)  
2  
>>> x = fruta.find('z')  
>>> print(x)  
-1
```

# Convertiendo todo a MAYÚSCULAS

- Puedes crear una copia de una cadena en **minúsculas** o **mayúsculas**
- Muchas veces al buscar una subcadena mediante **find()** primero convertimos la cadena a minúsculas, de modo que podemos buscar una subcadena sin importar si está en mayúsculas o minúsculas

```
>>> saludo = 'Hola Bob'  
>>> mayusc = saludo.upper()  
>>> print(mayusc)  
HOLA BOB  
>>> minusc = saludo.lower()  
>>> print(minusc)  
hola bob  
>>>
```

# Buscar y reemplazar

- La función `replace()` es como una operación de “buscar y reemplazar” en un editor de texto
- Esta función reemplaza todas las ocurrencias de una cadena de búsqueda con una cadena de reemplazo

```
>>> saludo = 'Hola Bob'  
>>> cad = saludo.replace('Bob', 'Jane')  
>>> print(cad)  
Hola Jane  
>>> cad = saludo.replace('o', 'X')  
>>> print(cad)  
HXla BXb  
>>>
```

# Eliminando espacios en blanco

- A veces filtramos o limpiamos una cadena eliminando espacios en blanco al inicio y/o al final
- `lstrip()` y `rstrip()` eliminan espacios en blanco a la izquierda o a la derecha
- `strip()` elimina espacios en blanco tanto al inicio como al final de la cadena

```
>>> saludo = '    Hola Bob    '
>>> saludo.lstrip()
'Hola Bob    '
>>> saludo.rstrip()
'    Hola Bob'
>>> saludo.strip()
'Hola Bob'
>>>
```

# Búsqueda de prefijos

```
>>> linea = 'Que tengas un buen día'  
>>> linea.startswith('Que')  
True  
>>> linea.startswith('q')  
False
```

# Extracción de datos

21                    31  
↓                    ↓  
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
>>> datos = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'  
>>> arrpos = datos.find('@')  
>>> print(arrpos)  
21  
>>> esppos = datos.find(' ', arrpos)  
>>> print(esppos)  
31  
>>> direccion = datos[arrpos+1 : esppos]  
>>> print(direccion)  
uct.ac.za
```

## Ejercicios

1. Implementa una función que reciba cadenas de la forma “etiqueta:valor decimal”, por ejemplo:

`'X-DSPAM-Confidence: 0.8475'`

Y devuelva la cadena que recibe con el valor redondeado a 2 decimales. Por ejemplo:

`'X-DSPAM-Confidence: 0.85'`

NOTA: No utilizar `str.split()`

2. Implementa una segunda función que elimine el prefijo “`X-`” de una cadena.

NOTA: No utilizar `str.replace()`.

3. Testea las funciones para ver que funcionan correctamente.

*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.8. Fundamentos de programación en Python



# Procesamiento de ficheros

Un fichero de texto, especialmente de almacenamiento de datos científicos o informáticos, puede procesarse como una secuencia de líneas

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
```

Details: <http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772>

# Apertura de un fichero

- Antes de poder leer el contenido de un fichero, le indicaremos a Python con qué fichero vamos a trabajar y lo que haremos con él
- Esto se hace mediante la función `open()`
- `open()` devuelve un **manejador de ficheros o archivos (file handle)** – una variable usada para ejecutar operaciones sobre el fichero

# Uso de open()

manejador = `open(nombrefichero, modo)`

retorna un manejador que se usa para manipular el fichero

`nombrefichero` es el nombre del fichero

`modo` es opcional y será, entre otros:

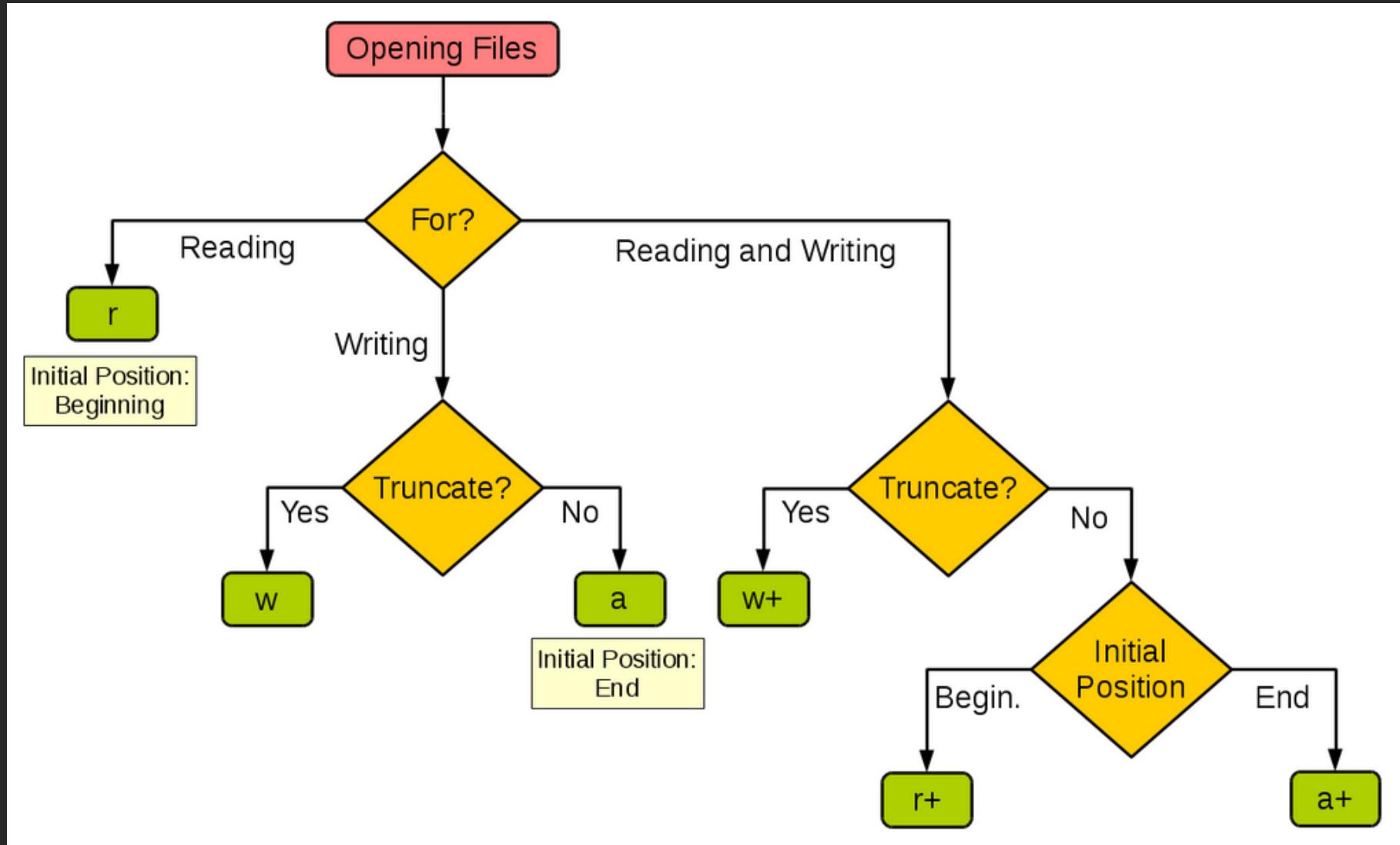
- 'r' si vamos a leer el fichero,
- 'w' si vamos a escribirlo

`manejador = open('mbox.txt', 'r')`

# Modos de abrir un fichero de texto

- 'r': abre un fichero existente para lectura. Es el modo por defecto. El puntero se sitúa al principio del fichero.
- 'w': abre un fichero para escritura. El contenido previo será sobrescrito. Si el fichero no existe, se creará uno.
- 'a': abre un fichero para adición. El contenido previo se mantiene. El puntero se sitúa al final del fichero. Si el fichero no existe, se creará uno.
- 'r+': abre un fichero existente para lectura y escritura. El contenido previo se mantiene. El puntero se sitúa al principio del fichero.
- 'w+': abre un fichero para escritura y lectura. El contenido previo será sobrescrito. Si el fichero no existe, se creará uno.
- 'a+': abre un fichero para adición y lectura. El contenido previo se mantiene. El puntero se sitúa al final del fichero. Si el fichero no existe, se creará uno.
- 'x': similar a 'w' pero genera una excepción si el fichero ya existe.
- 'x+': similar a 'w+' pero genera una excepción si el fichero ya existe.

# Modos de abrir un fichero de texto



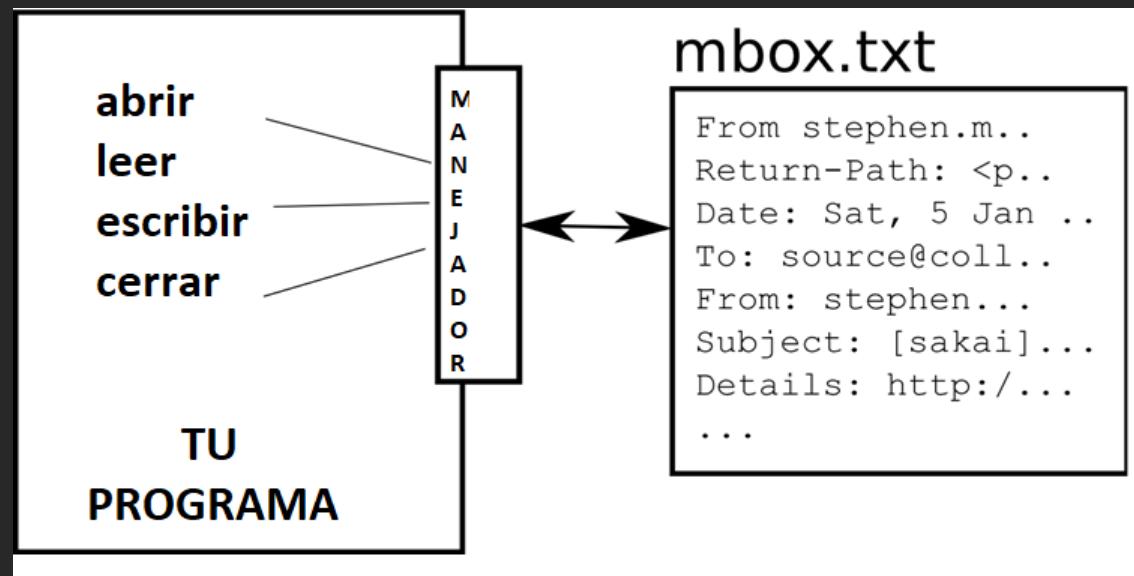
# Modos de abrir un fichero de texto

# Modos de abrir un fichero binario

- 'rb': abre un fichero binario existente para lectura. Es el modo por defecto. El puntero se sitúa al principio del fichero.
- 'wb': abre un fichero binario para escritura. El contenido previo será sobrescrito. Si el fichero no existe, se creará uno.
- 'ab': abre un fichero binario para adición. El contenido previo se mantiene. El puntero se sitúa al final del fichero. Si el fichero no existe, se creará uno.
- 'r+b': abre un fichero binario existente para lectura y escritura. El contenido previo se mantiene. El puntero se sitúa al principio del fichero.
- 'w+b': abre un fichero binario para escritura y lectura. El contenido previo será sobrescrito. Si el fichero no existe, se creará uno.
- 'a+b': abre un fichero binario para adición y lectura. El contenido previo se mantiene. El puntero se sitúa al final del fichero. Si el fichero no existe, se creará uno.
- 'xb': similar a 'wb' pero genera una excepción si el fichero ya existe.
- 'x+b': similar a 'w+b' pero genera una excepción si el fichero ya existe.

# ¿Qué es un manejador de ficheros?

```
>>> man = open('mbox.txt')
>>> print(man)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='UTF-8'>
```



# Cuando el fichero no existe

```
>>> man = open('fichero.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundException: [Errno 2] No such file or
directory: 'fichero.txt'
```

# El carácter salto de línea

- Utilizamos el carácter de escape **salto de línea (\n)** para indicar cuando termina una línea
- **Un salto de línea** es un solo carácter, no dos

```
>>> cad = '¡Hola\nMundo!'  
>>> cad  
'¡Hola\nMundo!'  
>>> print(cad)  
¡Hola  
Mundo!  
>>> cad = 'X\nY'  
>>> print(cad)  
X  
Y  
>>> len(cad)  
3
```

# Procesamiento de ficheros

Un fichero de texto tiene **saltos de línea** al final de cada línea,  
aunque sean invisibles

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008\n
Return-Path: <postmaster@collab.sakaiproject.org>\n
Date: Sat, 5 Jan 2008 09:12:18 -0500\n
To: source@collab.sakaiproject.org\n
From: stephen.marquard@uct.ac.za\n
Subject: [sakai] svn commit: r39772 - content/branches/\n
\n
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772\n
```

# Manejador de ficheros como secuencia de líneas

- Un **manejador de ficheros o archivos** abierto en modo de lectura puede ser tratado como una **secuencia** de cadenas donde cada línea en el archivo es una cadena en la secuencia
- Podemos usar la sentencia **for...in...** para iterar a través de una **secuencia**
- Recuerda: una **secuencia** es un conjunto ordenado

```
fichero = open('mbox.txt')  
for linea in fichero :  
    print(linea)
```

# Contar líneas en un fichero

- Abrir un **fichero** en modo lectura
- Utilizar un bucle **for...in...** para leer cada línea
- **Contar** las líneas e imprimir el número total de líneas

```
man = open('mbox.txt')
contador = 0
for linea in man:
    contador = contador + 1
print('Total de líneas:', contador)
```

```
$ python abrir.py
Total de líneas: 132045
```

## Ejercicio

1. Implementa una función que cuente las líneas de un fichero de forma “pythonica” (es decir, como acabamos de ver).
2. Implementa esa misma función mediante una llamada a la función `subprocess.run()` vista anteriormente, para que ejecute el comando de Shell de Linux “`wc -l`” que también nos contaba líneas. Si estás en Windows, averigua con qué comando realizarlo.
3. Compara si la primera función y la segunda retornan el mismo número de líneas para el fichero ejemplo.

# Un fichero entero como cadena

Podemos **leer** un fichero entero (saltos de línea incluidos) dentro de una sola **cadena**

```
>>> man = open('mbox-short.txt')
>>> cad = man.read()
>>> print(len(cad))
94626
>>> print(cad[:20])
From stephen.marquar
```

# Búsqueda en un fichero

Podemos poner una sentencia `if` en nuestro bucle `for...in...` para imprimir únicamente aquellas líneas que satisfacen cierta característica

```
man = open('mbox-short.txt')
for linea in man:
    if linea.startswith('From:'):
        print(linea)
```

# Problema

¿Qué hacen ahí todas  
esas líneas en blanco?

From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu

...

# Problema

¿Qué hacen ahí todas  
esas líneas en blanco?

- Cada línea del fichero tiene un **salto de línea** al final
- La función **print** agrega un **salto de línea** a cada línea

```
From: stephen.marquard@uct.ac.za\n\nFrom: louis@media.berkeley.edu\n\nFrom: zqian@umich.edu\n\nFrom: rjlowe@iupui.edu\n\n...
```

# Búsqueda en un fichero (corregido)

- Podemos eliminar los espacios en blanco del lado derecho de la cadena utilizando `rstrip()` de la librería String
- El salto de línea es considerado como un “espacio en blanco” y es eliminado

```
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    if linea.startswith('From:'):
        print(linea)
```

From: stephen.marquard@uct.ac.za  
From: louis@media.berkeley.edu  
From: zqian@umich.edu  
From: rjlowe@iupui.edu  
....

# Ignorando líneas con continue

Podemos ignorar líneas de forma conveniente utilizando la sentencia **continue**

```
man = open('mbox-short.txt')
for linea in man:
    linea = linea.rstrip()
    if not linea.startswith('From:'):
        continue
    print(linea)
```

# Usando `in` para seleccionar líneas

Podemos buscar una subcadena `en` cualquier parte de una **línea** como nuestro criterio de selección

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f...
```

```
man_a = open('mbox-short.txt')
for linea in man_a:
    linea = linea.rstrip()
    if not '@uct.ac.za' in linea:
        continue
    print(linea)
```



```
nombre = input('Introduce el nombre del fichero: ')
manejador = open(nombre)
contador = 0
for linea in manejador:
    if linea.startswith('Subject:'):
        contador = contador + 1
print('Hay', contador, 'líneas de Subject en', nombre)
```

# Solicitar nombre de fichero

Introduce el nombre del fichero: mbox.txt  
Hay 1797 líneas de Subject en mbox.txt

Introduce el nombre del fichero: mbox-short.txt  
Hay 27 líneas de Subject en mbox-short.txt

# Gestionando nombres de fichero incorrectos

```
nombre = input('Introduce el nombre del fichero: ')
try:
    man = open(nombre)
except:
    print('El fichero no se puede abrir:', nombre)
    sys.exit()

contador = 0
for linea in man:
    if linea.startswith('Subject:'):
        contador = contador + 1
print('Hay', contador, 'líneas de Subject en', nombre)
```

Introduce el nombre del fichero: mbox.txt  
Hay 1797 líneas de Subject en mbox.txt

Introduce el nombre del fichero: noexiste.txt  
El fichero no se puede abrir: noexiste.txt

## Ejercicios

1. Dado el fichero “**mbox-short.txt**”, mostrar por pantalla las líneas que comienzan por “**X-...**” en forma de **etiqueta:valor**, que tengan **valores numéricos**, aplicándoles las **funciones** desarrolladas anteriormente.
2. Dado el fichero “**mbox.txt**”, determinar qué día de la semana se han recibido más correos.

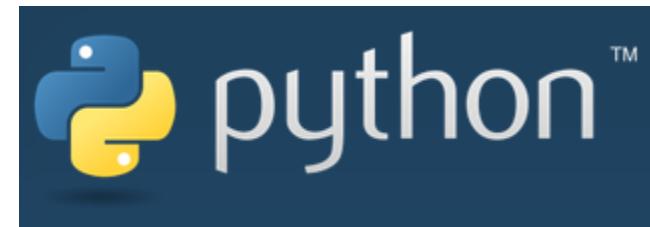
*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.9. Fundamentos de programación en Python



# Listas



- Una **colección** nos permite almacenar múltiples valores en una sola **variable**
- Con las colecciones podemos gestionar **múltiples valores** de manera adecuada
- Las **listas** son uno de los tipos de **colección** disponibles en Python

```
amigos = [ 'Joseph', 'Glenn', 'Sally' ]
```

```
equipaje = [ 'calcetines', 'camisa', 'colonia' ]
```

# Elementos en listas

- Los elementos de una **lista** están encerrados por corchetes y separados por comas
- Un elemento de una **lista** puede ser cualquier objeto de Python – incluso **otra lista**
- Una **lista** puede estar vacía

```
>>> print([1, 24, 76])
[1, 24, 76]
>>> print(['rojo', 'amarillo', 'azul'])
['rojo', 'amarillo', 'azul']
>>> print(['rojo', 24, 98.6])
['rojo', 24, 98.6]
>>> print([ 1, [5, 6], 7])
[1, [5, 6], 7]
>>> print([])
[]
```

# Ya habíamos usado listas...

```
for i in [5, 4, 3, 2, 1] :  
    print(i)  
print('¡Despegue!')
```

5  
4  
3  
2  
1  
¡Despegue!

# Listas y bucles definidos

```
amigos = ['Joseph', 'Glenn', 'Sally']
for amigo in amigos :
    print('¡Feliz año nuevo!', amigo)
print('¡Hecho!')
```

```
z = ['Joseph', 'Glenn', 'Sally']
for x in z:
    print('¡Feliz año nuevo!', x)
print('¡Hecho!')
```

# Búsqueda dentro de listas

De la misma manera que con cadenas de caracteres, podemos obtener cualquier elemento individual de una lista utilizando un índice especificado entre **corchetes**



```
>>> amigos = [ 'Joseph', 'Glenn', 'Sally' ]  
>>> print(amigos[1])  
Glenn  
>>>
```

# Mutabilidad de listas

- Las cadenas son “**inmutables**” – no podemos cambiar el contenido de una cadena – tenemos que crear una **nueva cadena** para hacer cualquier cambio
- Las listas son “**mutables**” - podemos **cambiar** un elemento de una lista utilizando el operador índice

```
>>> fruta = 'Banana'  
>>> fruta[0] = 'b'  
Traceback  
TypeError: 'str' object does not  
support item assignment  
>>> x = fruta.lower()  
>>> print(x)  
banana  
>>> loto = [2, 14, 26, 41, 63]  
>>> print(loto)  
[2, 14, 26, 41, 63]  
>>> loto[2] = 28  
>>> print(loto)  
[2, 14, 28, 41, 63]
```

# Longitud de listas

- La función `len()` recibe una **lista** como parámetro y retorna el número de **elementos** en la **lista**
- De hecho, `len()` nos retorna el número de elementos de cualquier conjunto o secuencia (tal como una cadena de caracteres...)

```
>>> x = [ 1, 2, 'joe', 99]
>>> print(len(x))
4

>>> saludo = 'Hola Bob'
>>> print(len(saludo))
9
```

# La función range

- La función `range` retorna una lista de números que van desde el cero hasta el número anterior al parámetro
- Podemos construir un bucle mediante índices usando un `for` y un iterador

```
>>> print(range(4))
[0, 1, 2, 3]
>>> amigos = ['Joseph', 'Glenn', 'Sally']
>>> print(len(amigos))
3
>>> print(range(len(amigos)))
[0, 1, 2]
>>>
```

# Bucles equivalentes

```
amigos = ['Joseph', 'Glenn', 'Sally']

for amigo in amigos :
    print('Feliz año nuevo:', amigo)

for i in range(len(amigos)) :
    amigo = amigos[i]
    print('Feliz año nuevo:', amigo)
```

```
>>> amigos = ['Joseph', 'Glenn', 'Sally']
>>> print(len(amigos))
3
>>> print(range(len(amigos)))
[0, 1, 2]
>>>
```

Feliz año nuevo: Joseph  
Feliz año nuevo: Glenn  
Feliz año nuevo: Sally

# Concatenación de listas con el +

Podemos crear una  
nueva lista uniendo o  
concatenando dos listas  
creadas previamente

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3]
```

# Seccionamiento de listas con :

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41, 12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

**Recuerda:** Tal y como ocurría con las cadenas de caracteres, el segundo número indica “**hasta pero sin incluir**”

# Métodos sobre listas

```
>>> x = list()
>>> type(x)
<type 'list'>
>>> dir(x)
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>>
```

<http://docs.python.org/tutorial/datastructures.html>

# Creación de listas desde cero

- Podemos crear una **lista** vacía y después añadir elementos usando el método **append**
- La **lista** mantiene su orden y los nuevos elementos se **añaden** al final de la **lista**

```
>>> cosas = list()
>>> cosas.append('libro')
>>> cosas.append(99)
>>> print(cosas)
['libro', 99]
>>> cosas.append('galleta')
>>> print(cosas)
['libro', 99, 'galleta']
```

# Búsqueda de elementos en listas

- Python proporciona los operadores habituales `in` y `not in` que te permiten comprobar si un elemento se encuentra en una lista o no
- Son operadores lógicos que retornan `True` o `False`

```
>>> numeros = [1, 9, 21, 10, 16]
>>> 9 in numeros
True
>>> 15 in numeros
False
>>> 20 not in numeros
True
>>>
```

# Ordenación de listas

- Una **lista** puede almacenar muchos elementos y los mantiene en el orden en que estos se han añadido a menos que lo cambiemos
- Podemos **ordenar** una **lista** de manera alfabética o numérica ascendente mediante el método **sort ()**

```
>>> amigos = [ 'Joseph', 'Glenn', 'Sally' ]
>>> amigos.sort()
>>> print(amigos)
['Glenn', 'Joseph', 'Sally']
>>> print(amigos[1])
Joseph
>>>
```

# Funciones nativas sobre listas

- Las **funciones** nativas de **Python** que se habían utilizado anteriormente también admiten **listas** como parámetros

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25.6
```

```
total = 0
contador = 0
while True :
    inp = input('Introduce un número: ')
    if inp == 'hecho' : break
    valor = float(inp)
    total = total + valor
    contador = contador + 1

promedio = total / contador
print('Promedio:', promedio)
```

Introduce un número: 3  
Introduce un número: 9  
Introduce un número: 5  
Introduce un número: hecho  
Promedio: 5.66666666667

```
numlista = list()
while True :
    inp = input('Introduce un número: ')
    if inp == 'hecho' : break
    valor = float(inp)
    numlista.append(valor)

promedio = sum(numlista) / len(numlista)
print('Promedio:', promedio)
```

# Cadenas y listas

```
>>> cad = 'Con tres palabras'  
>>> palabras = cad.split()  
>>> print(palabras)  
['Con', 'tres', 'palabras']  
>>> print(len(palabras))  
3  
>>> print(palabras[0])  
Con  
>>> print(palabras)  
['Con', 'tres', 'palabras']  
>>> for palabra in palabras :  
    print(palabra)
```

`split()` separa una cadena en partes, produciendo una lista de cadenas. Podemos verlas como palabras, y `acceder` a una palabra en particular o `iterar` a través de todas ellas.

```
>>> linea = 'Muchos espacios'  
>>> lista1 = linea.split()  
>>> print(lista1)  
['Muchos', 'Espacios']  
>>>  
>>> linea = 'primero;segundo;tercero'  
>>> lista2 = linea.split()  
>>> print(lista2)  
['primero;segundo;tercero']  
>>> print(len(lista2))  
1  
>>> lista3 = linea.split(';')  
>>> print(lista3)  
['primero', 'segundo', 'tercero']  
>>> print(len(lista3))  
3  
>>>
```

- Si no se especifica un **delimitador**, el carácter espacio, aunque sean múltiples espacios, será el delimitador por defecto
- La función **split()** también permite especificar qué carácter **delimitador** utilizar al separar o dividir la cadena

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
manejador = open('mbox-short.txt')
for linea in manejador:
    linea = linea.rstrip()
    if not linea.startswith('From ') : continue
    palabras = linea.split()
    print(palabras[2])
```

Sat

Fri

Fri

Fri

...

```
>>> linea = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> palabras = linea.split()
>>> print(palabras)
['From', 'stephen.marquard@uct.ac.za', 'Sat', 'Jan', '5', '09:14:16', '2008']
>>>
```

# Subdivisiones de listas

From **stephen.marquard@uct.ac.za** Sat Jan 5 09:14:16 2008

```
palabras = linea.split()
email = palabras[1]
email_dividido = email.split('@')
dominio = email_dividido[1]
print(dominio)
```

stephen.marquard@uct.ac.za  
['stephen.marquard', 'uct.ac.za']  
'uct.ac.za'

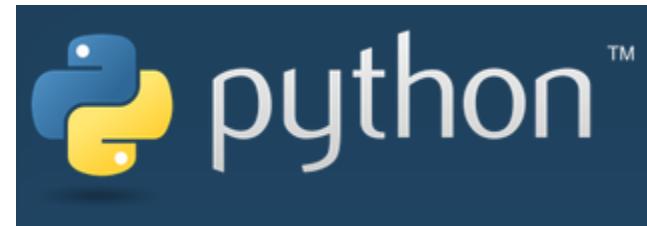
*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.10. Fundamentos de programación en Python



# Diccionarios

- Los **diccionarios** son un tipo de colección que almacena la información sin un orden particular, accediendo a los **valores** mediante su **etiqueta** o **clave**
- Un **diccionario** es, por tanto, una colección de pares **clave:valor** (o **etiqueta:valor**)
- Constituyen una estructura de datos muy potente que permite efectuar operaciones rápidas similares a las de selección, inserción y actualización en una base de datos

# Diccionarios

- En un **diccionario**, la información queda almacenada de manera muy similar a como se almacena en un archivo JSON
- Los **diccionarios** son una estructura de datos que también existen en otros lenguajes de programación, aunque bajo nombres diferentes:
  - Vectores asociativos - Perl / PHP
  - Tabla o Map o HashMap - C++ / Java
  - PropertyBag - C#

# Diccionarios

- Las listas **indexaban** sus entradas basándose en la posición de la lista
- Los **diccionarios** son como bolsas, es decir, no tienen orden, así que sus entradas se **indexan** mediante una “etiqueta” o “clave”

```
>>> bolsa = dict()
>>> bolsa['tabaco'] = 5
>>> bolsa['boli'] = 2
>>> bolsa['pañuelos'] = 1
>>> print(bolsa)
{'tabaco': 5, 'pañuelos': 1, 'boli': 2}
>>> print(bolsa['boli'])
2
>>> bolsa['boli'] = bolsa['boli'] + 2
>>> print(bolsa)
{'tabaco': 5, 'pañuelos': 1, 'boli': 4}
```

# Comparación entre listas y diccionarios

- Lista

- Una colección lineal de valores que mantienen un orden



- Diccionario

- Una colección de valores desordenada, similar a una bolsa en la que cada valor se asocia a una etiqueta o clave



# Comparación entre listas y diccionarios

Los **diccionarios** son como **listas** a excepción de que utilizan **claves** en lugar de números para buscar los **valores**

```
>>> lst = list()  
>>> lst.append(21)  
>>> lst.append(183)  
>>> print(lst)  
[21, 183]  
>>> lst[0] = 23  
>>> print(lst)  
[23, 183]
```

```
>>> dic = dict()  
>>> dic['edad'] = 21  
>>> dic['curso'] = 182  
>>> print(dic)  
{'curso': 182, 'edad': 21}  
>>> dic['edad'] = 23  
>>> print(dic)  
{'curso': 182, 'edad': 23}
```

```
>>> lst = list()  
>>> lst.append(21)  
>>> lst.append(183)  
>>> print(lst)  
[21, 183]  
>>> lst[0] = 23  
>>> print(lst)  
[23, 183]
```

Lista

Clave	Valor
[0]	21
[1]	183

lst

```
>>> dic = dict()  
>>> dic['edad'] = 21  
>>> dic['curso'] = 182  
>>> print(dic)  
{'curso': 182, 'edad': 21}  
>>> dic['edad'] = 23  
>>> print(dic)  
{'curso': 182, 'edad': 23}
```

Diccionario

Clave	Valor
['curso']	182
['edad']	21

dic

# Definición de diccionarios

- Los diccionarios se definen entre llaves como una colección de pares **clave:valor**
- Se puede inicializar un diccionario vacío con **dict()** o con **{ }**

```
>>> dic = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print(dic)
{'jan': 100, 'chuck': 1, 'fred': 42}
>>> dic_vacio = { }
>>> print(dic_vacio)
{ }
>>>
```

# Encontrar el elemento más frecuente

marcos

alejandro

natalia

alejandro

mar

marcos

alejandro

natalia

mar

alejandro

natalia

marcos

alejandro

# Encontrar el elemento más frecuente

marcos

alejandro

natalia

alejandro

mar

mar

alejandro

natalia

marcos

||

||||

|||

|||

mar

alejandro

natalia

marcos

alejandro

# Múltiples contadores en un diccionario

Un uso común de diccionarios es **contar** la frecuencia de aparición

```
>>> dic = dict()  
>>> dic['marcos'] = 1  
>>> dic['mar'] = 1  
>>> print(dic)  
{'marcos': 1, 'mar': 1}  
>>> dic['mar'] = dic['mar'] + 1  
>>> print(dic)  
{'marcos': 1, 'mar': 2}
```

Clave	Valor
mar	
alejandro	<del>    </del>
natalia	
marcos	

# Errores en diccionarios

- Se genera un **error** si se referencia una clave que no existe en el diccionario
- Mediante el operador **in** se puede comprobar si cierta clave se encuentra en el diccionario o no

```
>>> dic = dict()  
>>> print(dic['mar'])  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    KeyError: 'mar'  
>>> 'mar' in dic  
False
```

# Inserción o actualización de elementos

Al encontrar un nuevo **nombre**, necesitamos insertar una nueva entrada en el **diccionario** y, para las siguientes veces, debemos encontrar el **nombre** y sumarle uno al contador en el **diccionario** bajo ese **nombre**

```
contadores = dict()
nombres = ['marcos', 'mar', 'mar', 'alejandro', 'marcos']
for nombre in nombres :
    if nombre not in contadores:
        contadores[nombre] = 1
    else :
        contadores[nombre] = contadores[nombre] + 1
print(contadores)
```

# El método `get` en diccionarios

Es tan común verificar si una `clave` ya existe en un diccionario, o darle un valor por defecto si la `clave` no existe, que hay un **método** llamado `get()` para hacerlo directamente

```
if nombre in contadores:  
    x = contadores[nombre]  
else :  
    x = 0  
  
x = contadores.get(name, 0)
```

Valor por defecto si la clave no existe (no se producen errores).

# El método `get` en diccionarios

Es posible usar `get()` y proporcionar un **valor por defecto** cuando la **clave** todavía no existe en el diccionario

```
contadores = dict()
nombres = ['marcos', 'mar', 'mar', 'alejandro', 'marcos']
for nombre in nombres :
    contadores[nombre] = contadores.get(nombre, 0) + 1
print(contadores)
```

Valor por  
defecto



# Contador de palabras

```
contadores = dict()
print('Introduce una línea de texto:')
linea = input('')

palabras = linea.split()
print('Palabras:', palabras)

print('Contando...')
for palabra in palabras:
    contadores[palabra] = contadores.get(palabra, 0) + 1
print('Contadores', contadores)
```

El patrón general para contar las palabras en una línea de texto es **dividir** la línea en palabras, y después recorrer las palabras y usar un **diccionario** para mantener el contador de cada palabra de forma independiente.

```
python contador_palabras.py
```

```
Introduce una línea de texto:
```

```
el payaso corrio detras del carro y el carro corrio dentro  
de la tienda y la tienda cayo sobre el payaso y el carro
```

```
Palabras: ['el', 'payaso', 'corrio', 'detras', 'del',  
'carro', 'y', 'el', 'carro', 'corrio', 'dentro', 'de',  
'la', 'tienda', 'y', 'la', 'tienda', 'cayo', 'sobre',  
'el', 'payaso', 'y', 'el', 'carro']
```

```
Contando...
```

```
Contadores {'el': 4, 'payaso': 2, 'corrio': 2, 'detras':  
1, 'del': 1, 'carro': 3, 'y': 3, 'dentro': 1, 'de': 1,  
'la': 2, 'tienda': 2, 'cayo': 1, 'sobre': 1}
```

# Recorrer diccionarios

A pesar de que los **diccionarios** no se almacenan en orden, es posible utilizar un bucle **for** que recorra todas las **entradas** del **diccionario**

De hecho, este recorrerá todas las **claves** del diccionario

```
>>> contadores = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for clave in contadores:
...     print(clave, contadores[clave])
...
jan 100
chuck 1
fred 42
>>>
```

# Obtención de claves y valores

Es posible obtener una lista de las **claves**, de los **valores**, o de los **ítems (ambos)** de un diccionario

```
>>> dic = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print(dic.keys())
['jan', 'chuck', 'fred']
>>> print(dic.values())
[100, 1, 42]
>>> print(dic.items())
[('jan', 100), ('chuck', 1), ('fred', 42)]
>>>
```

# Obtención de claves y valores

- Es posible iterar a través de los pares **clave:valor** de un diccionario mediante dos variables de iteración
- En cada iteración, la primera variable es la **clave** y la segunda variable es el **valor** correspondiente a dicha clave

```
dic = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
for miclave,mivalor in dic.items() :
    print(miclave, mivalor)
```

jan 100  
chuck 1  
fred 42

miclave	mivalor
[jan]	100
[chuck]	1
[fred]	42

# La palabra más frecuente en un fichero

```
nombre = input('Introduce un nombre de archivo: ')
manejador = open(nombre)

contadores = dict()
for linea in manejador:
    palabras = linea.split()
    for palabra in palabras:
        contadores[palabra] = contadores.get(palabra, 0) + 1

maximocontador = None
palabramasfrecuente = None
for palabra, contador in contadores.items():
    if maximocontador is None or contador > maximocontador:
        palabramasfrecuente = palabra
        maximocontador = contador

print(palabramasfrecuente, maximocontador)
```

```
python palabras.py
Enter file: clown.txt
the 7
```

## Ejercicios

1. Dado el fichero “[el\\_quijote.txt](#)”, mostrar por pantalla todas las palabras distintas del mismo junto con su frecuencia de aparición.
2. Sobre el mismo fichero, determinar la palabra más frecuente del mismo.
3. Determinar la menor frecuencia de aparición de palabras y mostrar todas las palabras con esa frecuencia de aparición mínima

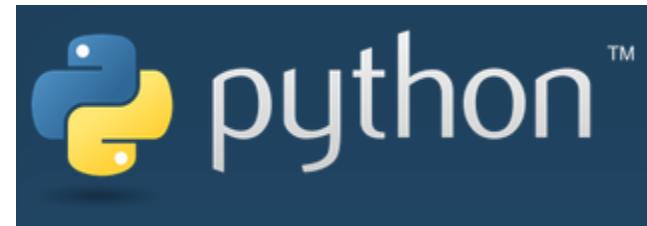
*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*



# U2.11. Fundamentos de programación en Python



# Tuplas

Las tuplas son otra estructura de datos disponible en Python que funciona de manera similar a una lista, en el sentido de que está ordenada y se indexa mediante su posición

```
>>> tuplacad = ('Glenn', 'Sally', 'Joseph')           >>> for i in tuplanum:  
>>> print(tuplacad[2])                                print(i)  
Joseph                                                 1  
>>> tuplanum = ( 1, 9, 2 )                            9  
>>> print(tuplanum)                                  2  
(1, 9, 2)                                            >>>  
>>> print(max(tuplanum))  
9
```

# Inmutabilidad de tuplas

A diferencia de una lista, una vez que se crea una **tupla** no se puede alterar su contenido, es decir, es **inmutable**

```
>>> x = [9, 8, 7]           >>> y = 'ABC'                 >>> z = (5, 4, 3)
>>> x[2] = 6                >>> y[2] = 'D'                  >>> z[2] = 0
>>> print(x)                Traceback: 'str'               Traceback: 'tuple'
>>>[9, 8, 6]                  object does                   object does
>>>                           not support item          not support item
>>>                           Assignment            Assignment
>>>
```

# Inmutabilidad de tuplas

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no attribute 'reverse'
>>>
```

# Listas vs. tuplas

```
>>> l = list()  
>>> dir(l)  
['append', 'count', 'extend', 'index', 'insert', 'pop',  
'remove', 'reverse', 'sort']  
  
>>> t = tuple()  
>>> dir(t)  
['count', 'index']
```

# Listas vs. tuplas

- Puesto que Python no tiene que crear la estructura de una tupla de modo que sea modificable, las tuplas son más simples y eficientes que una lista en términos de uso de memoria y rendimiento
- Por tanto, a la hora de programar, será preferible utilizar tuplas en lugar de listas cuando se necesiten múltiples variables temporales que no haya que modificar

# Asignación de tuplas

- Es posible colocar una **tupla** en el **lado izquierdo** de una sentencia de asignación, lo que nos permite realizar una asignación múltiple de variables
- Incluso es posible omitir los paréntesis

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred
>>> a, b, c = 99, 98, 97
>>> print(a)
99
```

# Tuplas y diccionarios

El método **items()** en un diccionario retornaba, en realidad, una lista de **tuplas (clave, valor)**

```
>>> dic = dict()
>>> dic['mar'] = 2
>>> dic['marcos'] = 4
>>> for (clave, valor) in dic.items():
    print(clave, valor)

mar 2
marcos 4
>>> listatuplas = dic.items()
>>> print(listatuplas)
dict_items([('mar', 2), ('marcos', 4)])
```

# Comparabilidad de tuplas

Los **operadores** de comparación funcionan con **tuplas** y otras secuencias. Si el primer elemento es igual, Python revisa el siguiente elemento, y así sucesivamente hasta que encuentra elementos diferentes, momento en el que se detendrá.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ('Jones', 'Sally') < ('Jones', 'Sam')
True
>>> ('Jones', 'Sally') > ('Adams', 'Sam')
True
```

# Ordenación de listas de tuplas

- Es posible ordenar una lista de **tuplas** para obtener una versión ordenada de un diccionario
- Para ello, se hace uso de la función **sorted()** aplicada al retorno del método **items()** sobre el diccionario

```
>>> dic = {'a':10, 'b':1, 'c':22}  
>>> dic.items()  
dict_items([('a', 10), ('c', 22), ('b', 1)])  
>>> sorted(dic.items())  
[('a', 10), ('b', 1), ('c', 22)]
```

# Ordenación de listas de tuplas

```
>>> dic = {'a':10, 'b':1, 'c':22}
>>> lt = sorted(dic.items())
>>> lt
[('a', 10), ('b', 1), ('c', 22)]
>>> for c, v in lt:
...     print(c, v)
...
a 10
b 1
c 22
```

# Ordenación por valores en vez de por claves

- Si se construyera una lista de **tuplas** en la forma **(valor, clave)**, sería posible **ordenar (sort)** por valor
- Esto se puede hacer mediante un bucle **for** que vaya añadiendo tuplas a una lista

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for c, v in d.items() :
...     l.append( (v, c) )
...
>>> print(l)
[ (10, 'a'), (22, 'c'), (1, 'b') ]
>>> l = sorted(l, reverse=True)
>>> print(l)
[ (22, 'c'), (10, 'a'), (1, 'b') ]
```

# Ordenación por valores en vez de por claves

```
man = open('romeo.txt')
contadores = dict()
for linea in man:
    palabras = linea.split()
    for palabra in palabras:
        contadores[palabra] = contadores.get(palabra, 0) + 1

lst = list()
for clave, valor in contadores.items():
    nuevatupla = (valor, clave)
    lst.append(nuevatupla)

lst = sorted(lst, reverse=True)
for valor, clave in lst[:10] :
    print(clave, valor)
```

Obtención de las 10  
palabras más  
frecuentes del archivo

# Comprensión de listas

La comprensión de listas supone un atajo para crear nuevas listas a partir de otras, de manera rápida y eficiente.

Su sintaxis es la siguiente:

*nuevaLista = [ expresión(elemento) for elemento in viejaLista if condición ]*

```
>>> dic = {'a':10, 'b':1, 'c':22}
```

```
>>> listaord = sorted( [ (v,c) for c,v in dic.items() ], reverse=True )
```

```
print(listaord)
```

```
[ (22, 'c'), (10, 'a'), (1, 'b') ]
```

*Estas diapositivas están protegidas por derechos de autor 2010- Charles R. Severance ([www.drchuck.com](http://www.drchuck.com)) de la Facultad de Información de la Universidad de Michigan, y se ponen a disposición bajo licencia de Creative Commons Attribution 4.0. Por favor, conserve esta última diapositiva en todas las copias del documento para cumplir con los requisitos de atribución de la licencia. Si realiza algún cambio, siéntase libre de agregar su nombre y el de su organización a la lista de colaboradores en esta página cuando republique los materiales.*

*Desarrollo inicial: Charles Severance, Facultad de Información de la Universidad de Michigan*

*Modificaciones: Roberto González, Unidad TIC de Florida Universitària*

