Programación de computadores con Python

(Parte I)

Fundamentos de Computación

Grado en Ingeniería en Tecnologías Industriales

14 de Septiembre 2021

Universidad de Cantabria

OBJECTS (Objetos)

Objects

- Un programa en PYTHON manipula datos denominados objetos (objects).
- Los objetos tienen una clase (class) que define que tipo de manipulaciones pueden hacer los programas con ellos:
  - Claudia es humana, así que puede caminar, hablar castellano, etc
  - Bavieca es un caballo, así que puede caminar, relinchar, etc.
- Luego los objetos son instancias o solicitudes de un clase.
- Los objetos pueden ser:
  - escalares (no pueden ser divididos)
  - no-escalares (tienen una estructura interna que podemos acceder a ella)

- INT representan números enteros ($\mathbb{Z}$), por ejemplo 12, -2027 (escalar)
- FLOAT representan números reales ($\mathbb{R}$), por ejemplo 1.4142, 2027.0 (escalar) 1.4142=14142E-4=141.42e-2.
- COMPLEX representan números complejos ($\mathbb{C}$), por ejemplo 1+2i, en PYTHON 1+2j. (no escalar)
- NONETYPE especial, y no tiene ningún valor, (escalar)
- Podemos usar TYPE() para saber que clase de objeto.

```
>>> type(4)
<class 'int'>
>>> 56.3
56.3
>>> 45e2
4500.0
```

```
>>> type(4.67)
<class 'float'>
>>> 2,1
(2, 1)
>>> type((2,1))
<class 'tuple'>
```

```
>>> type(1+2j)
<class 'complex'>
>>> (1+2j).real
1.0
>>> (1+2j).imag
2.0
```

Casting

- Algunas veces se puede convertir un objeto en otra clase de objeto (**casting**)
- `float(5), convierte el entero 5 en el float, 5.0`
- `int(1.41), trunca el número 1.41 al int, 1`
- `complex(1), convierte el entero 1 al complex, (1+0j)`

Expresiones

- Una expresión es una combinación de objetos y operadores
- La sintaxis para una expresión simple es:
  `<objeto> <operador> <objeto>,`
- `toda expresión tiene un valor(objeto).`

Operador	Operación	Ejemplo	Resultado	Prioridad
**	Exponenciación	2**3	8	1
*	Multiplicación	8.5*2.5	21.25	2
/	División	11/3	3.6666666666666665	2
//	Cociente	11//3	3	2
%	Resto o módulo	29 %8	5	2
+	Suma	11.1+3	14.1	3
-	Resta	5-3j-19	(-14-3j)	3

Cuando en una expresión aparecen varias operadores se efectúa aplicando las reglas usuales de prioridad de las operaciones, como se ilustra en la tabla. Y con la misma prioridad, se ejecutan de izquierda a derecha, excepto la exponenciación que se ejecuta de derecha a izquierda. En caso de querer que las operaciones se realicen en otro orden, se pueden utilizar paréntesis.

El entero más grande está limitado por las características del computador (memoria, etc.). Sin embargo, el número decimal más grande que acepta PYTHON *es menor que* 10.02^{308} *y el más próximo a cero es mayor que* 10^{-323}.

Los números enteros es un subconjunto de los reales y, estos últimos de los números complejos. Por lo tanto, en PYTHON *si en la expresión* `<objeto> <operador> <objeto>` *uno de los objetos es de una* `class` *superior, entonces el valor de la expresión es un objeto de esa clase.*

```
>>> 10.02**308
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    10.02**308
OverflowError: (34, 'Result too large')
>>> 10**308 + 2027
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
>>> 10**-323 + 4
4.0
>>> 1.1 + 2
3.1
```

Errores de tecleo y excepciones

Si introducimos una expresión incorectamente PYTHON nos lo indica con un mensaje de error.

```
>>> 0+1)
  File "<stdin>", line 1
    0+1)
       ^
SyntaxError: invalid syntax
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

El primero es un error de sintaxis (*SyntaxError*) y, el segundo es de naturaleza distinta, se trata de un error de división por cero.

IEEE Standard 754

```
>>> 0.1+0.2
0.30000000000000004
>>> from math import sqrt
>>> sqrt(2)**2
2.0000000000000004
>>>
```

La codificación utilizada en Python es el *IEEE standard 754* con precisión doble (53 bits).

Python

convierte números decimales a binario, realiza la operación en binario y convierte el resultado de nuevo en decimal. Muchos números decimales no se pueden representarse de forma exacta en binario, por lo que los resultados no pueden ser exactos y con el redondeo se pueden acumular los errores. Python tiene la clase **Decimal** y la clase **Fractions** para solventar esto. Lo explicaremos en el tema Representación de los números.

Clases(tipos) de objetos(datos)

Como hemos visto, cada objeto(dato) que maneja PYTHON tiene un **class** (no es lo mismo 45, cuyo tipo es «número entero», que 45.0). Hay otros tipos de datos además de los números. Tenemos, por ejemplo, el objeto cadena de la class (**str**), que es una sucesión de caracteres encerrada entre comillas (simples o dobles).

```
>>> type('Hola, Mundo')
<class 'str'>
>>> type("7")
<class 'str'>
>>> int("7")
7
>>> type(type)
<class 'type'>
>>>
```

Variables

Resulta útil utilizar «nombres» para algunos datos que pretendemos utilizar varias veces. Por ejemplo, si queremos calcular el largo de un pedazo de papel que guarde las mismas proporciones que los de la serie A de la norma ISO/DIN:

```
>>>  from math  import sqrt
>>>  razon = sqrt(2)
>>> 21 * razon
29.698484809834998
>>> 29.7 * razon
42.002142802480925
```

Las variables en PYTHON se pueden entender como *cajas* de memoria del computador en las que se guardan los objetos.

Variables

- Las variables se crean cuando se definen por primera vez, es decir, cuando se les asigna un valor por primera vez.
- Para asignar un valor a una variable se utiliza el operador asignación =. A la izquierda del operador se escribe el nombre/identificador de la variable y a la derecha el valor que se quiere dar a la variable.

```
>>> 2*x
NameError: name 'x' is not defined
>>> x = 5
>>> 2*x
10
```

Como indica el término que estamos utilizando, el valor de una variable no tiene por qué permanecer fijo. Normalmente necesitaremos que cambie.

```
>>> x = 3.
>>> 2*x
6.0
```

Variables

- Una variable puede almacenar cualquier tipo de objetos, en particular, el objeto cadena (**str**).
- El valor de un variable puede variar. A lo largo de un mismo programa, una variable puede incluso asociarse a objetos de tipos distintos.

```
>>> x = "Alto Campoo"
>>> x
'Alto Campoo'
>>>  x = 2**(1/2)
>>> x
1.4142135623730951
>>> horas = 5
>>> minutos = 60 * horas
>>> segundos = 60 * minutos
>>> segundos
18000
```

Variables

No podemos utilizar cualquier cadena de caracteres como **identificador** para denominar una variable. Por ejemplo, parece razonable que Pyhton no permita llamar a una variable `g=h`, ya que si quisiéramos asignarle un valor: `g=h=2`, el interpréte no podría decidir entre crear dos variables (`g` y `h`) con el valor 2 o una sola.

En general, podemos utilizar letras, números y el signo `_`, con las dos restricciones siguientes:

- El primer carácter del identificador no puede ser un número.
- No pueden utilizarse algunas palabras que reserva el intérprete (por ejemplo: **for, break, and**...)

```
>>> 3d = 34
  File "<stdin>", line 1
    3d = 34
     ^
SyntaxError: invalid syntax
>>> d3 = 2019
```

Asignación

Es muy importante distinguir entre una asignación y una ecuación.

- La primera es una instrucción, una acción: cierta variable «toma» un valor.
- Una ecuación matemática, en cambio, no «cambia» nada: solo indica que dos cantidades son iguales.
- Las ecuaciones son simétricas, y las asignaciones es importante distinguir dónde se coloca cada término.

Por tanto, la expresión de Python:

```
x = 2 * x
```

no es una ecuación con la solución única `x=0`. Se trata de una instrucción que asigna a la variable `x` el doble del valor que almacenaba anteriormente.

```
>>> x = 5
>>> x = 2 * x
>>> x
10
```

Sintaxis

PYTHON como cualquier otro lenguaje de programación, especifica una **sintaxis** para su código. Como en la mayoría de los lenguajes,

Por lo general, cada instrucción se escribe en una línea:

```
>>> texto = "santander"
>>> texto.capitalize()
'Santander'
```

También pueden incluirse varias instrucciones en una misma línea, utilizando el separador (;):

```
>>> altura = 25; cota = 320; nieva = altura - cota; nieva
-295
```

Otras sentencias de asignación

Los lenguajes de programación suelen disponer de símbolos para representar operaciones artiméticas que no son comunes al escribir matemáticas.

En Pyhton, la expresión `a (operador) = b` es, en general, equivalente a `a = a (operador) b`.

```
>>> a = 3; a += 4; a
7
>>> a %= 5;a
2
>>> b = "Cadena "; c = "expansible"; b += c; b
'Cadena expansible'
```

print

print(expresión,.....,expresión)

- Esta instrucción de PYTHON evalúa las expresiones que involucra según las reglas del intérprete y las imprime por la salida estándar.
- función PRINT() admite varios argumentos seguidos. En el programa, los argumentos deben separarse por comas. Los argumentos se muestran en el mismo orden y en la misma línea, separados por espacios:

```
>>> print("Hola, mundo")
Hola, mundo
>>> print("Hola", "adios", 2015)
Hola adios 2015
>>> type(print("Hola, mundo"))
Hola, mundo
<class 'NoneType'>
```

print

print(expresión,.....,expresión)

Permite incluir variables o expresiones como argumento.

```
>>> nombre = "Alan Turing"
>>> edad = 19
>>> print("Me llamo", nombre, "y tengo", edad, "años.")
Me llamo Alan Turing y tengo 19 años.
```

input

input([mensaje])

La instrucción `input` puede considerarse contraria a `print`: permite la comunicación entre programa y usuario en el sentido inverso.De forma predeterminada, la función convierte la entrada en una cadena (str).

saludo.py

```
print("¿ Como se llama ?")
nombre = input()
print("Un placer saludarle", nombre)
```

Si queremos que PYTHON interprete la entrada como un número entero o float se debe utilizar la función INT() o la función FLOAT(), para convertirlo (casting).

pesetas.py

```
cantidad = int(input("Dígame una cantidad en pesetas: "))
print(cantidad, "pesetas son", cantidad / 166.386, "euros")
```

Tipo booleano

- **Tipo booleanos.** Dos valores: TRUE (verdadero) y FALSE (falso). Unas operaciones (entre otras) que dan como resultado booleanos son las comparaciones ($<$ menor que; $==$ igual que ; $!=$ distinto de)

```
>>> 4 == 5
False
>>> 4 <  5
True
>>> type("hola" != "adiós")
<class 'bool'>
```

- PYTHON utiliza los enteros 0 y 1 para representar estos valores, dando lugar a situaciones como esta:

```
>>> True + True
2
>>> type(True * False)
<class 'int'>
```

Operadores lógicos: not, and, or

A	B	A and B	A or B
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

A	not A
V	F
F	V

```
>>> True and True
True
>>> True or False
True
>>> False or False
False
>>> not False and False
False
>>>  (True or False) and not(False and True)
True
```

Leyes de De Morgan

Si la condición para poder salir a jugar es `lentejas and deberes`; la condición contraria, que determina cuándo no se puede salir, es

`not (lentejas and deberes) <=====> not lentejas or not deberes`

Operadores lógicos: and, or, not

Operación	Resultado	Nota
A or B	Si A es falso, entonces B. En otro caso A	(1)
A and B	Si A es falso, entonces A. En otro caso B	(2)
not A	Si A es falso, entonces True. En otro caso False	(3)

- (1) Solamente se evalúa el segundo argumento si el primero es falso.
- (2) Solamente se evalúa el segundo argumento si el primero es verdadero
- (3) **not**, tiene una prioridad más baja que los operadores no booleanos. Por ejemplo, **not A == B** es evaluado como **not (A == B)** y **A == not B** produce un error.

Operadores lógicos: and, or, not

Cualquier objeto puede ser evaluado para valor booleano. Los siguientes valores son falsos:

None, False, 0, 0.0, 0j, '',(),[],{}

Todos los demás valores se consideran verdaderos, por lo que los objetos de muchos tipos siempre son verdaderos.

```
>>> not 0j
True
>>> 1 and 3
3
>>> 3 and 1
1
>>> "silencio" != "tranquilidad" and "suspenso"
'suspenso'
>>> not 56.7
False
```

Operadores de comparación

Python permite utilizar al menos 8 operadores de comparación, a casi todas clases de objetos, e incluso entre objetos de clases distintas (devolviendo un resultado poco significativo, en general.

Operador	Significado
<	menor
<=	menor o igual
>	mayor
>=	mayor o igual

Operador	Significado
==	iguales
!=	distintos
is	idénticos
is not	no idénticos

```
>>> 5 < 6
True
>>> 5 <= 5
True
>>> ''pedro' < 'marta'
False
```

```
>>> 5 == 6
False
>>> 'pedro' != 'marta'
True
>>> 3 is 4
False
```

Operadores de comparación

- Todos tienen la misma prioridad (que es más alta que la de las operadores booleanos).
- Las comparaciones pueden ser encadenadas arbitrariamente; Por ejemplo, ($\mathbf{x} < \mathbf{y} <= \mathbf{z}$) es equivalente a ($\mathbf{x} < \mathbf{y}$ *and* $\mathbf{y} <= \mathbf{z}$)

```
>>> 1 < 2 < 5 >= 4 != 3
True
>>> 1 < 2 > 5 >= 4 != 3
False
>>> 1 < 2 > 5 or 4 != 3
True
 >>> "clase" != "murmullo" or "suspenso"
True
```

Asignación, comparación e identidad

Es importante distinguir entre los operadores = , == y **is** que hemos visto:

- = sirve para asignar un valor a una variable. No es simétrico: a la izquierda se pone el nombre de la variable y a derecha, el valor que toma.
- == compara dos objetos y devuelve un valor «bool» (True o False).
- **is** compara si dos objetos son idénticos y devuelve un valor «bool» (True o False).

```
>>> v = 2
>>> v == 3
False
>>> v = 3
>>> 3.0 == v
True
>>> v is 3.0
False
```

Estructuras de control

La programación resultaría demasiado rígida si tuviéramos que restringirnos a enumerar una sucesión de instrucciones que se ejecutaran una detrás de otra. A veces, queremos «seguir caminos distintos» en función de alguna circunstancia:

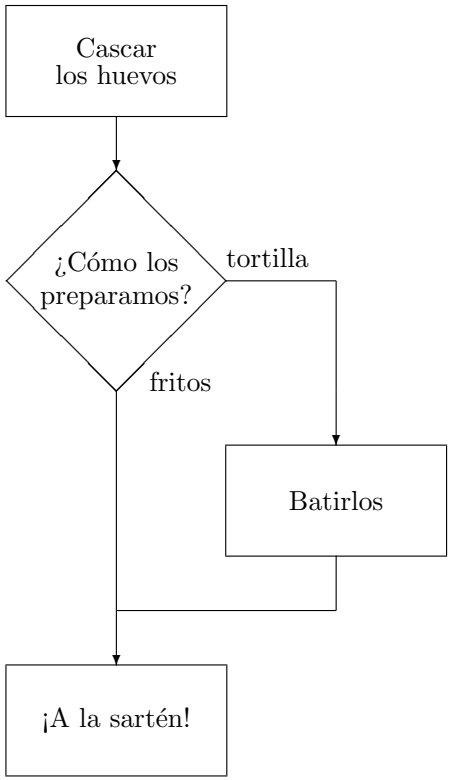
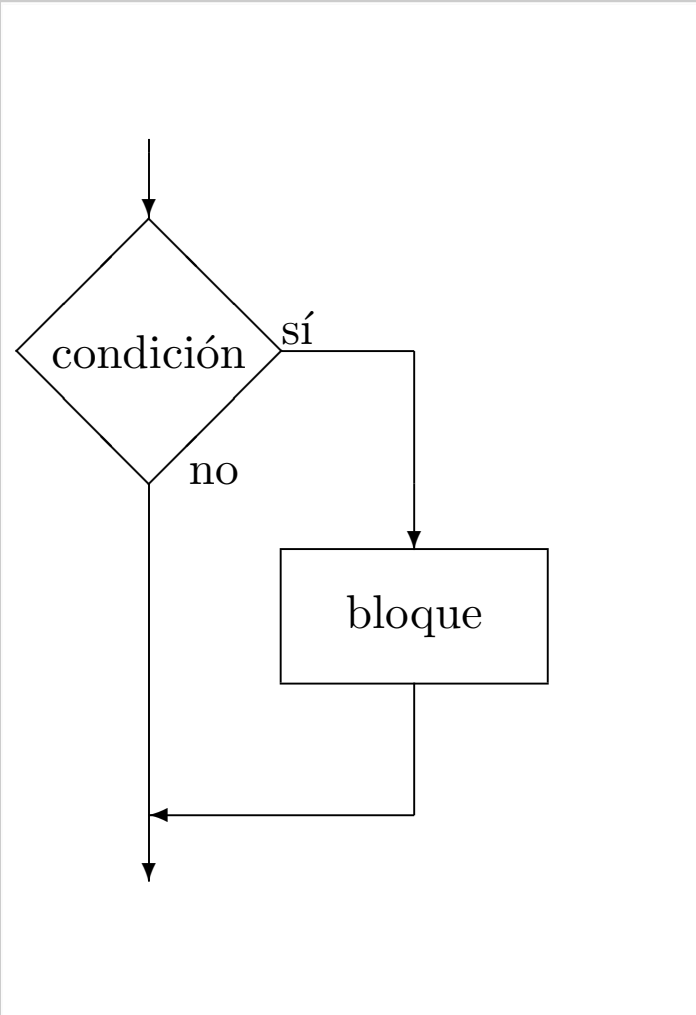


Diagrama de flujo

if ...

Para modificar la ejecución lineal de las instrucciones, los lenguajes cuentan con instrucciones denominadas estructuras de control. La instrucción **if ...** permite que un programa ejecute unas instrucciones cuando se cumplan una condición:




```
if (condición):
    (bloque)
```

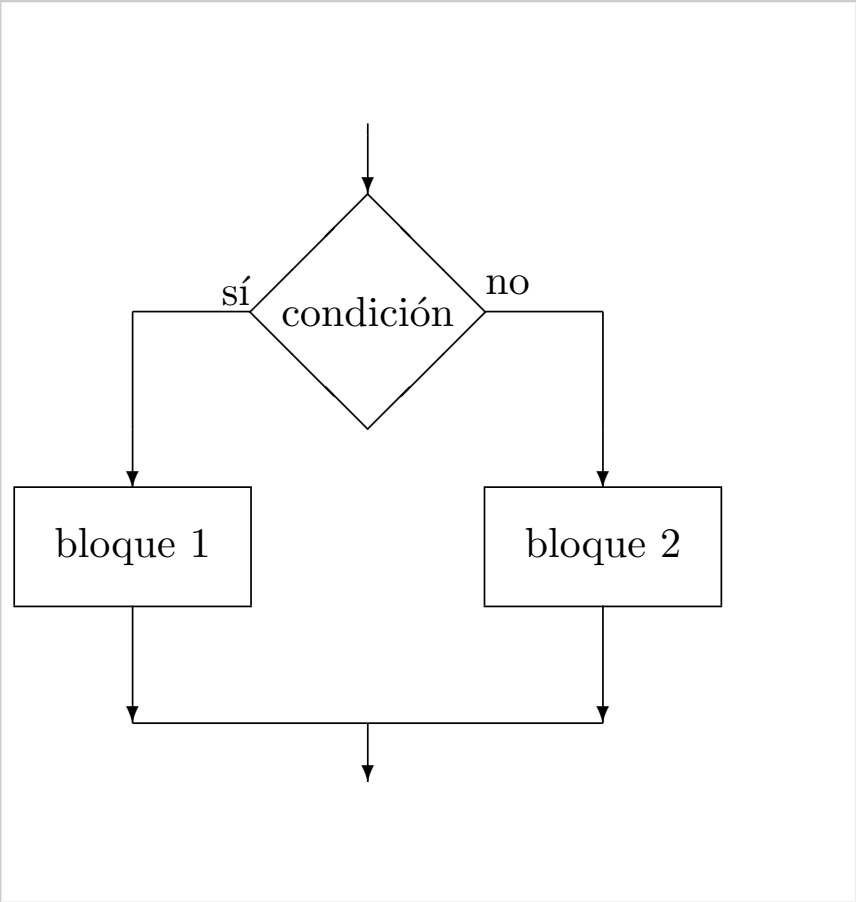
if ...

```
numero = int(input("Escriba un número positivo: "))
if numero < 0:
    print("Parece que no me has entendido")
    print("¡Le he dicho que escriba un número positivo!")
print("Ha escrito el número", numero)
print("Adios")
```

- La primera línea contiene la condición a evaluar y es una expresión lógica y, esta debe terminar (:).
- El bloque de órdenes que se ejecutan cuando la condición se cumple (es decir, cuando la condición es el boleano TRUE).
- Este bloque debe ir sangrado, utiliza el sangrado para reconocer las líneas que forman un bloque de instrucciones.

if ... else

Con esta estructura podemos que el programa ejecute unas instrucciones cuando se verifica una condición y otras instrucciones cuando no.




```
if (condición):
    | (bloque 1)
else:
    | (bloque 2)
```

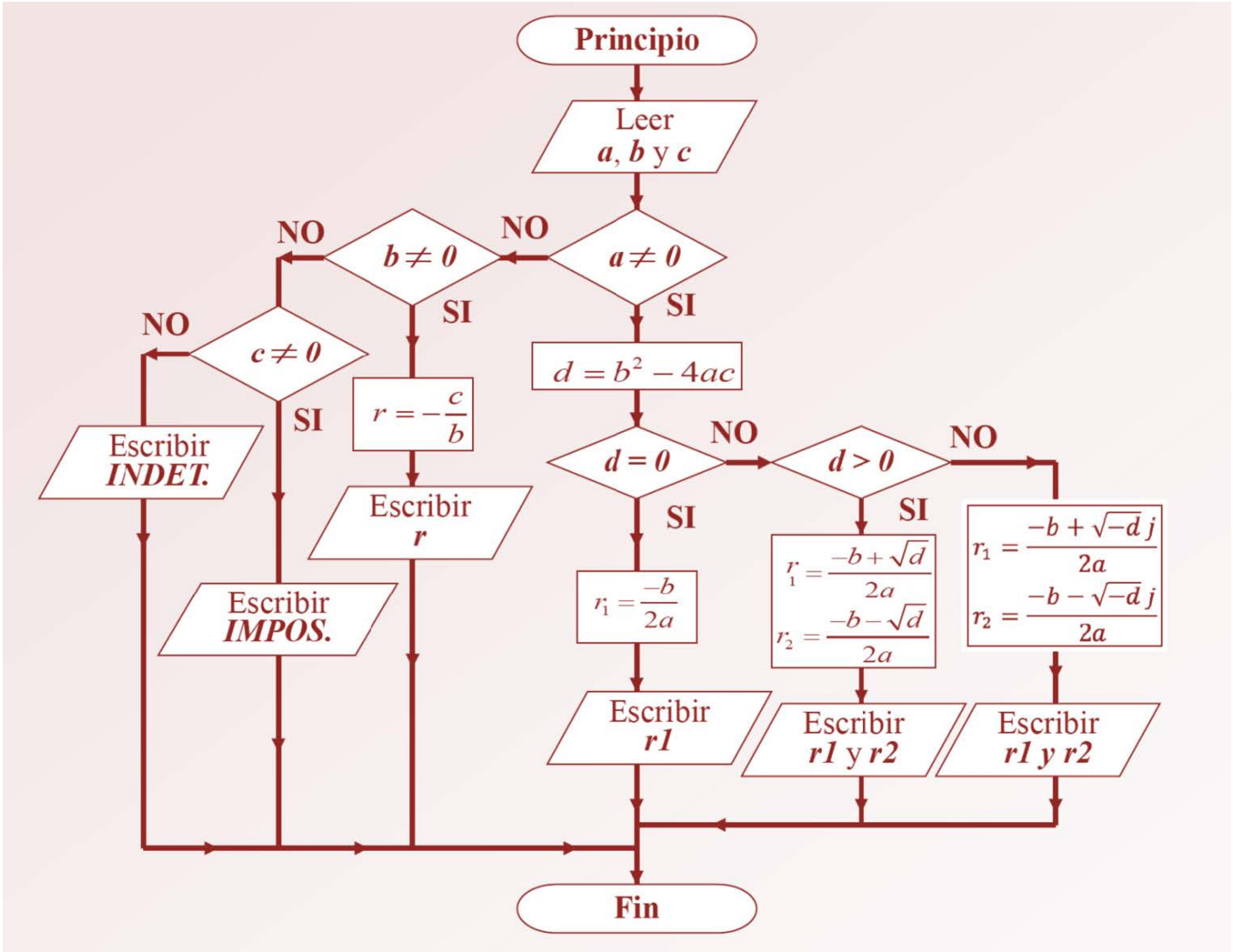
if ... else

votar.py

```
edad = int(input("¿Cuántos años tiene? "))
if edad < 18:
    print("Es usted menor de edad. No puede votar.")
else:
    print("Es usted mayor de edad. Puede votar.")
print("¡Hasta la próxima!")
```

```
$ python votar.py
Cuántos años tiene?  11
Es usted menor de edad. No puede votar.
¡Hasta la próxima!
$ python votar.py
Cuántos años tiene?  91
Es usted mayor de edad. Puede votar.
¡Hasta la próxima!
```

La ecuación $ax^2 + bx + c = 0$



Queremos comprar un regalo de una lista de bodas y elegir el más caro que permita nuestro presupuesto. Así, nuestro programa preguntará al usuario cuánto dinero está dispuesto a gastar y le informará del regalo más costoso que puede adquirir, dentro de una lista como esta:

vehículo (30 000), vajilla (600), TV (400), abrebotellas (1)

regalo-bodas(1).py

```
n = float(input("Dime tu presupuesto:  "))
if n >= 30000:
    print("vehiculo")
if  600 <= n < 30000 :
    print("vajilla")
if  400 <= n < 600 :
    print("TV")
if  1 <= n < 400:
    print("abrebotellas")
if  n < 1:
    print("no voy a la boda")
```

regalo-bodas(2).py

```
n = float(input(" Dime tu presupuesto:  "))
if n >= 30000:
    print("vehiculo")
else:
    if n >= 600:
        print("vajilla")
    else:
        if n >= 400:
            print("TV")
        else:
            if n >= 1:
                print("abrebotellas")
            else:
                print("no voy a la boda")
```

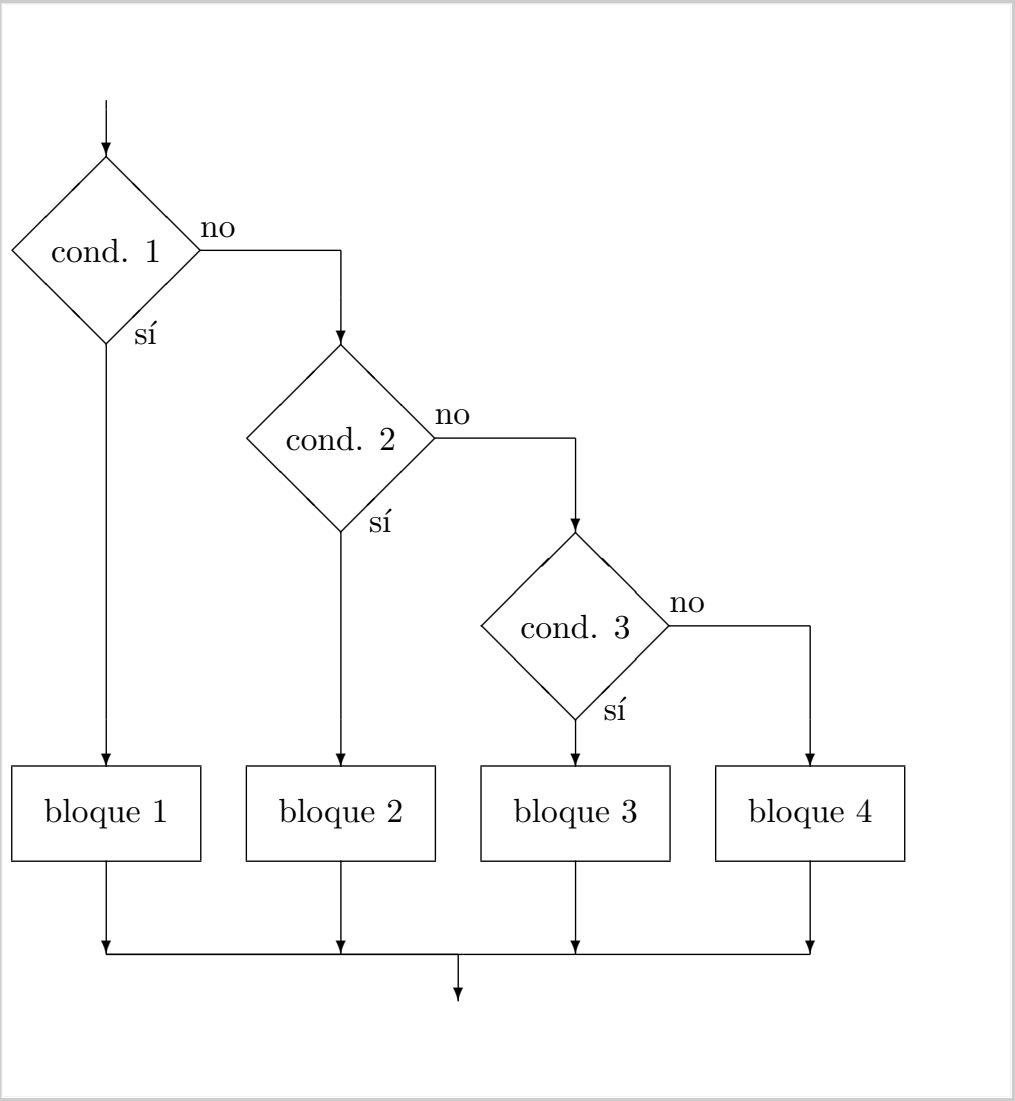
if ... elif ... elif ... else

Podemos plantear este programa siguiendo esto:

- si se cumple una condición (tener un presupuesto suficiente para el objeto más caro), llevamos a cabo una acción(comprarlo);
- en otro caso, pasamos a comprobar otra condición; y así sucesivamente.
- Pueden incluirse tantas sentencias `elif` como se necesite.
- Se ejecuta únicamente el bloque de la primera condición que se verifica o el correspondiente a `else` (en caso de estar presente).

```
if (cond. a):
    | (bloque a)
elif (cond. b):
    | (bloque b)
elif (cond. c):
    | (bloque c)
... (cond. x):
    | (bloque x)
... (cond. y):
    | (bloque y)
else:
    | (bloque z)
```

if ... elif ... elif ... else



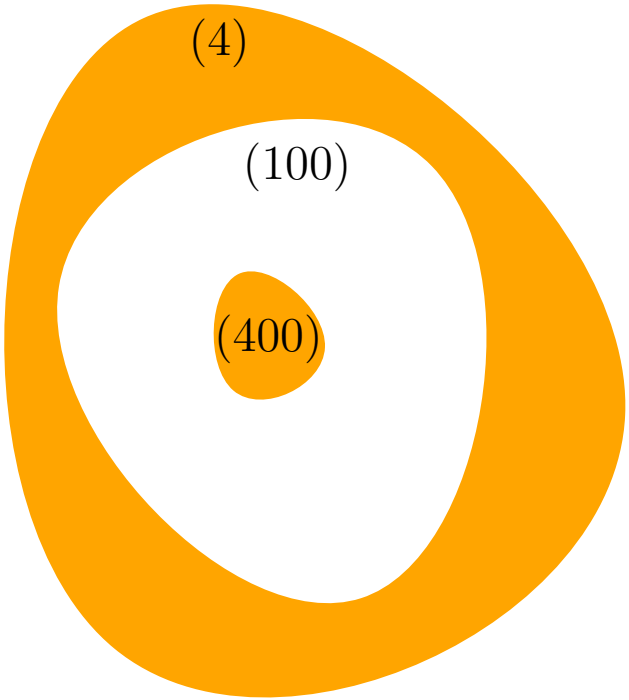

```
if (cond. 1):
    | (bloque 1)
elif (cond. 2):
    | (bloque 2)
elif (cond. 3):
    | (bloque 3)
else:
    | (bloque 4)
```

if ... elif ... elif ... else

regalo-bodas(3).py

```
print ('Programa Comprar regalo boda')
print (' Introduzca el presupuesto.')
a = float(input('a =  '))
if a >= 30000:
    print("Regalo un vehículo")
elif a >= 600:
    print("Regalo una vajilla")
elif a >= 400:
    print("Regalo un TV")
elif a >= 1:
    print("Regalo un abrebotellas")
else:
    print("Lo siento, no puedo ir a tu boda")
```

En el calendario juliano, los años tenían 365 días, salvo uno de cada cuatro, que tenía 366. Debido al desfase con respecto al año solar, en la reforma gregoriana del calendario se decidió que, de cada cuatro años finiseculares, solo uno fuera bisiesto. Así, el año 2 000 fue bisiesto, pero no el 1 900, ni lo será el 2 100.



Para ello, habrá que considerar cuatro «tipos» de años, según la región del dibujo a la que pertenecen: los múltiplos de 400; los que, no siéndolo, sí son múltiplos de 100; los que son múltiplos de 4 pero no son finiseculares; y, por último, los que no son múltiplos de 4. Las regiones coloreadas se corresponden con los años bisiestos.

De las siguientes dos propuestas `bis_1.py` y `bis_2.py`, solamente la última determina correctamente si un año es bisiesto o no.

`bis_1.py`

```
print('Escriba un año:')
a = int(input())

if a % 400 == 0:
    print('bisiesto')
if a %100 == 0:
    print('normal')
if a % 4 == 0:
    print('bisiesto')
else:
    print('normal')
```

`bis_2.py`

```
1  print('Escriba un año:')
2  a = int(input())
3
4  if a % 400 == 0:
5      print('bisiesto')
6  elif a % 100 == 0:
7      print('normal')
8  elif a % 4 == 0:
9      print ('bisiesto')
10 else:
11     print('normal')
```

El programa `bis_3.py` discrimina los años bisiestos de los ordinarios utilizando una única bifurcación condicional:

bis_3.py

```
print('Escriba un año:')
a = int(input())
if a %400 == 0 or not a % 100 == 0 and a % 4 == 0:
     print('bisiesto')
else:
    print('normal')
```

La condición contraria de `bis_3.py` y sin utilizar el operador `not`:

```
print('Escriba un año: ')
a = int(input())
if  a % 400 != 0 and  a % 100 == 0 or a % 4 != 0:
    print('normal')
else:
    print('bisiesto')
```

Cadenas/listas/tuplas

PYTHON tiene varias clases de objetos (no escalares):

- Una **str**(cadena) es una sucesión de caracteres encerrada entre comillas (simples o dobles).
- Una **list**(lista) es una secuencia de objetos (números, cadenas, listas, ...). Los elementos de una lista deben estar encerrados entre corchetes y separados por comas.
- Una **tuple**(tupla) es una secuencia de objetos (números, cadenas, listas, tuplas ...). Los elementos de una lista deben estar encerrados entre paréntesis y separados por comas.

```
>>> a = [1, 3.1, 'VIP', ["Pedro", 942100013], ["Juan", 942805678]]
>>> type(a)
<class 'list'>
>>> b = (True, [2, 3, 5, 7], 'NON SMOKING', (2**(1 /2), 2))
>>> type(b)
<class 'tuple'>
```

len(), in

- La función **len()** devuelve la longitud de la cadena/lista/tupla, es decir el número de caracteres/elementos.
- El operador de pertenencia **in** aplicado a un objeto de una cadena/lista/tupla devuelve el booleano False o True, si el objeto pertenece o no pertenece.

```
>>> a = [1, 3.1, 'VIP', ["Pedro", 942100013], ["Juan", 942805678]]
>>> len(a)
5
>>> b = (True, [2, 3, 5, 7], 'NON SMOKING', (2**(1 /2), 2))
>>> len(b)
4
>>> len('abcdef')
6
>>> 'VIP' in a
True
>>>  False in b
False
>>> 0 in []:
False
```

Indexar y Extraer

Los corchetes permiten la indexación en una cadena/lista/tupla para obtener el valor en un determinado índice/posición. `L[i]`, donde i es entero positivo, aplicado a la cadena/lista/tupla `L` devuelve el carácter/elemento de la posición $i + 1$ de `L`.

```
>>>  L = ['ax', 'by', 'cz']
>>> L[0]
'ax'
>>> L[1]
'by'
>>> 'L[2]'
'cz'
>>> len(L[0])
2
>>> L[0][1]
'x'
```

```
>>> L[-1]
'cz'
>>> L[-2]
'by'
>>> L[-3]
'ax'
>>> L[3]
Traceback (most recent cal
  File "<stdin>", line 1,
IndexError: list index out
```

Cortar/Trocear cadenas/listas/tuplas.

- Podemos cortar la cadena/tupla/lista:L, usando `L[inicio:detener:paso]`.
- Si se dan dos números, `L[inicio: detener]`, paso = 1, por defecto
- También se puede omitir números y dejar solo dos puntos

```
>>> L = (2, 4, 8, 16, 32, 64, 128, 256)
>>> L[3:6]
(16, 32, 64)
>>> L[3:6:2]
(16, 64)
>>> L[:]                                  # Evalua L[0:len(L)]
(2, 4, 8, 16, 32, 64, 128, 256)
>>> L[:2]                                 # Evalua L[0:2]
(2, 4)
>>> L[2:]                                 # Evalua L[2:len(L)]
(8, 16, 32, 64, 128, 256)
>>> L[:-1]                                # Evalua L[0:-1+Len(L)]
(2, 4, 8, 16, 32, 64, 128)
```

+, *

- Operador + (concatenación de cadena/lista/tupla): acepta dos cadenas/listas/tuplas como operandos y devuelve la cadena/lista/tupla que resulta de unir la segunda a la primera.
- Operador * (repetición de cadena/lista/tupla): acepta una cadena/lista/tupla y un entero positivo, y devuelve la concatenación de la cadena/lista/tupla consigo misma tantas veces como indica el entero.

```
>>> "Hola," +" "+ 'Santander'
'Hola, Santander'
>>> [['a',1],['b',2]] + [['c',3]]
[['a', 1], ['b', 2], ['c', 3]]
>>> [['a',1],['b',2]] + ['c',3]
[['a', 1], ['b', 2], 'c', 3]
>>> ['a','b'] + []
['a', 'b']
```

```
>>> (0,1) + (10,11)
(0, 1, 10, 11)
>>> 'holaX' * 3
'holaXholaXholaX'
>>> ()*2
()
>>> [1,2]*3
[1, 2, 1, 2, 1, 2]
```

Objetos mutables e inmutables

PYTHON usa la minima memoria necesaria. Ciertos objetos son **inmutables**, es decir, no pueden modificar su valor. Los objetos númericos, `int`, `float` son inmutables. Si dos variables almacenan el mismo objeto, sus identificadores apuntan a la misma zona de la memoria.

Las cadenas y las tuplas (STR, TUPLE) también son **immutables**. Por eso, asignar a una posición indexada de la cadena o tupla resulta en un error; sin embargo, las listas (LIST) pueden ser modificadas, son objetos **mutables**:

```
>>> a = 'hola'
>>> a[0] = 'H'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> Telf = [["Pedro", 942100013], ["Juan", 942805678]]
>>> Telf[1][0] = "Alan"
>>> Telf
[['Pedro', 942100013], ['Alan', 942805678]]
>>> Telf[0][1] = 1000000
```

Lista: objeto mutable

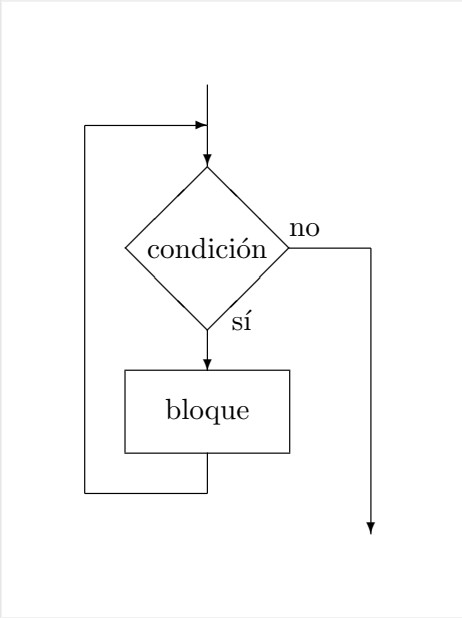
Una lista almacena en memoria referencias a los objetos, y no los objetos.

```
>>> L1 = [2, 3, 5]
>>> a = 2
>>> L2 = [2, 3, 5]
>>> L3 = L2
```

PYTHON reserva nueva memoria para la lista L2 aunque exista otra lista de idéntico valor. Como el contenido de cada celda ha resultado ser un valor inmutable (un entero), se han compartido las referencias a los mismos.

```
>>> L1 is L2
False
>>> L1 == L2
True
>>> L1[0] is a
True
>>> L3 is L2
True
>>> L2[0] = 89;  (L1,L2,L3)
([2, 3, 5], [89, 3, 5], [89, 3, 5])
```

while



| while (condición):
| (bloque)

- (condición) evalúa a un booleano
- Si (condición) es Cierta, se ejecutan todas las etapas del bloque.
- Comprueba (condición) de nuevo.
- Repite hasta que (condición) es Falsa.

```
1 i = 1
2 while i <= 50:
3   print(i)
4   i = 3*i + 1
5 print("Final")
```

```
1 i = 1
2 while i <= 10:
3   print(i, "", end="")
```

```
1 a,b = 0, 1
2 while b < 1000:
3     print(b, end = ',')
4     a,b = b, a+b
```

El segundo programa es un **bucle infinito**, para interrumpirlo hay que pulsar la combinación de teclas CRTL+C.

for <variable> in <iterable>. La función range()

```
lista=[1,2,'a',True]
for x in lista:
    print(x)
print('------')
for i in range(4):
    print(lista[i])
```

```
tupla=(1,2,'a',True)
for x in tupla:
    print(x)
print('------')
for i in range(4):
    print(tupla[i])
```

```
cadena='12aT'
for x in cadena:
    print(x)
print('------')
for i in range(4):
    print(cadena[i])
```

ITERABLES

- Listas, tuplas, cadenas
- **range(comienzo,parada,paso)**

```
misuma = 0
for x in range(7,10):
    misuma += 1
print(misuma)
```

```
misuma = 0
for x in range(5,11,2):
    misuma += x
print(misuma)
```

| for (variable) in (iterable):
| (bloque)

- cada vez que entra en el bucle la (variable) toma un valor
- la primera vez, (variable) comienza con el primer elemento(valor) del iterable
- la siguiente, (variable) toma el siguiente elemento(valor) del iterable
- Repite hasta que el iterable no tenga más elementos(valores)

continue

En un juego de dardos, se acumulan las puntuaciones obtenidas en diez tiradas, `dardos_1.py`. El programa del fichero `dardos_2.py` no tiene en cuenta las tiradas en las que se logran menos de 50 puntos.

`dardos_1.py`

```
total = 0
for i in range(10):
    p = int(input('Puntos: '))
    total += p
print ('TOTAL:',total)
```

`dardos_2.py`

```
1 total = 0
2 for i in range(10):
3     p = int(input('Puntos: '))
4     if p >= 50:
5         total += p
6 print ('TOTAL:',total)
```

Esto mismo puede conseguirse con la sentencia **continue**:

continue

dardos_3.py

```
total = 0
for i in range(10):
    p = int(input('Puntos: '))
    if p < 50:
        continue
    total += p
print('TOTAL:',total)
```

La sentencia **continue** interrumpe la iteración en curso de un bucle (el resto del bloque deja de ejecutarse) para pasar a la siguiente iteración.

break

Los siguientes dos programas descalifican aquellos jugadores que obtienen en alguna tirada menos de 1 punto, otorgándoles una puntuación total de -1 puntos.

dardos_4.py

```
total = 0; total1 = 0
for i in range(10):
    p = int(input('Puntos: '))
    if p < 1:
        total1 = -1
    total += p
if total1 < 0:
    print('TOTAL:', total1)
else:
    print('TOTAL:', total)
```

dardos_5.py

```
1 total = 0; i = 0
2 while i < 10:
3     p = int(input('Puntos: '))
4     if p < 1:
5         total = -1
6         i = 11
7     total += p
8     i += 1
9 print('TOTAL:', total)
```

Esto mismo puede conseguirse con la sentencia **break**:

break

dardos_6.py

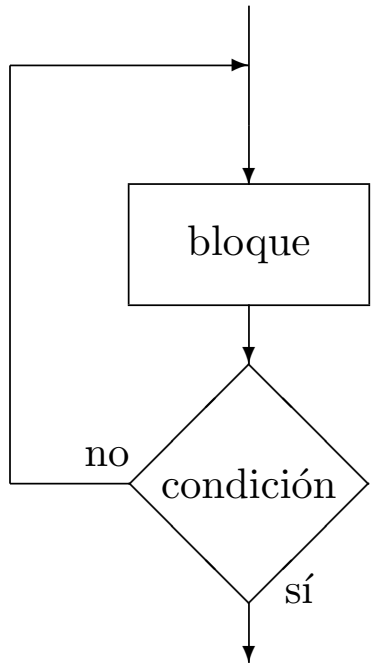
```
total = 0
for i in range(10):
    p = int(input('Puntos: '))
    if p < 1:
        total = -1
        break
    total += p
print('TOTAL:', total)
```

dardos_7.py

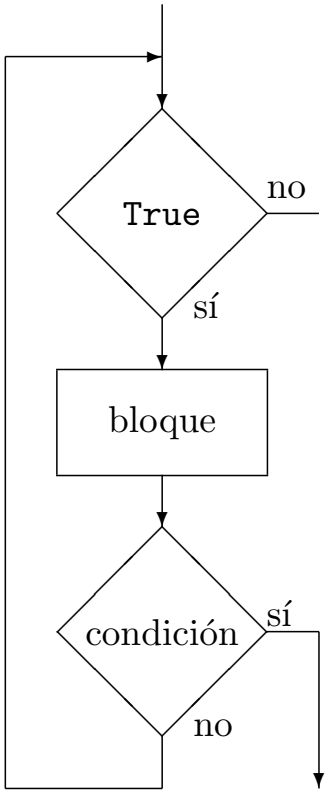
```
1 total = 0; i = 0
2 while i < 10:
3     p = int(input('Puntos: '))
4     if p < 1:
5         total = -1
6         break
7     total += p
8     i += 1
9 print('TOTAL:', total)
```

La sentencia **break** interrumpe el bucle (y no solo la iteración en curso como **continue**).

Python no cuenta con una estructura de control que reproduzca el siguiente diagrama:



Sin embargo, podemos lograr el mismo resultado combinando la sentencia **break** con un bucle **while**, en el que siempre se satisfaga la condición:




```
while True:
    (bloque)
    if (condición):
        break
```