PYTHON

Programación estructurada

Estructuras selectivas

- Hasta ahora los programas que hemos hecho siguen todos una secuencia fija de operaciones: muestran datos por pantalla y/o piden datos al usuario, pero siempre ejecutan la misma secuencia de pasos
- Con las **estructuras selectivas** podemos hacer que un programa ejecute una secuencia u otra de pasos dependiendo de ciertos valores de entrada
- El funcionamiento es siempre el mismo:
  - La estructura selectiva evalúa una expresión
  - Según el resultado de esa expresión, ejecuta un conjunto de instrucciones u otro
  - La expresión es en general una expresión booleana, de cuyo resultado (verdadero o falso) se decidirá el camino o bloque de instrucciones a ejecutar

Estructuras selectivas: if

- La estructura selectiva **if** permite ejecutar un bloque de instrucciones (tabulado hacia la derecha) si se cumple una determinada condición

```
if condicion:
    instruccion1
    instruccion2
    ...
...
```

- Ejemplo:

```
if x >= 0:
    print "x es positivo"
print "Fin"
```

- Es importante TABULAR las instrucciones que dependen del bloque *if*

Estructuras selectivas: if...else

- La estructura selectiva **if...else** permite ejecutar un bloque de instrucciones si se cumple una determinada condición, y otro bloque de instrucciones si no se cumple

```
if condicion:
    ... instrucciones
else:
    ... otras instrucciones
```

- Ejemplo:

```
if x >= 0:
   print "x es positivo"
else:
   print "x es negativo"
```

Ejercicio

- Crea un proyecto llamado **MayorEdad.py** que le pida al usuario su edad, y luego saque por pantalla el mensaje "Eres mayor de edad", o "No eres mayor de edad", según si el usuario es mayor de edad o no.

Estructuras selectivas: if...elif...else múltiples

- Para contemplar distintas posibilidades o caminos, según distintas condiciones, podemos utilizar la estructura **if...elif...elif..... else**

```
if condicion:
    ... instrucciones
elif condicion2:
    ... otras instrucciones
elif condicion3:
    ... otras instrucciones
else:
    ... otras instrucciones
```

Ejercicio

- Modifica el proyecto **MayorEdad.py** anterior para que, si el usuario tiene 17 años, saque el mensaje "No eres mayor de edad, pero te queda poco", manteniendo además los dos mensajes originales para el resto de edades.

Comprobar varias condiciones

- Hasta ahora los ejemplos que hemos visto comprobaban una sola condición (por ejemplo, que un número sea mayor o igual que cero). ¿Qué pasa si queremos comprobar varias?
  - Por ejemplo, que el número sea mayor o igual que cero y menor o igual que 10
- Para enlazar varias comprobaciones tenemos los **operadores lógicos**
  - **and**: sirve para enlazar dos condiciones. El resultado será verdadero si ambas condiciones son verdaderas
  - **or**: enlaza dos condiciones. El resultado es verdadero si alguna de las condiciones se cumple (no necesariamente las dos)
  - **not**: sirve para negar una condición. El resultado es verdadero si la condición original era falsa

Comprobar varias condiciones (II)

- Este *if* se cumple si el número está entre 0 y 10 (inclusive)
  - Sólo cumplen la condición los números del 0 al 10

```
if numero >= 0 and numero <= 10:
    ... instrucciones
```

- Este *if* se cumple si el número es mayor que 0 o menor que 10 (una de las dos).
  - Por tanto, el número 20 lo cumpliría, porque es mayor que 0, y el número -6 lo cumpliría, porque es menor que 10

```
if numero >= 0 or numero <= 10:
    ... instrucciones
```

Comprobar varias condiciones (III)

- Podemos enlazar varios de estos operadores. Hay que tener en cuenta que, si no usamos paréntesis, primero se evaluarán todos los *not,* luego todos los *and* y luego todos los *or*
  - En la precedencia general de operadores, los lógicos se evalúan justo antes de las asignaciones, después de las comparaciones
- Ejemplo: la siguiente expresión es cierta
  - Se evalúan primero los paréntesis (el OR es cierto), y luego el AND (que también es cierto)

```
(4 < 2 or 2 == 2) and 6 > 3
```

Ejercicio

- Indica si el resultado final de estas expresiones es verdadero o falso:
  1. `2 < 5 and 6 > 6`
  2. `3 == 3 or 2 == 5`
  3. `4 > 8 or (2 == 3 or 3 > 2)`
  4. `4 > 8 or 2 == 3 or 3 > 2`
  5. `4 > 1 and (2 == 2 and 3 > 4)`
  6. `4 > 1 and (2 == 2 and (not(3 > 4)))`
- Supongamos que n1 y n2 son las notas de dos exámenes de una asignatura:
  1. Construye una expresión lógica que sea cierta si las dos notas son mayores que 5
  2. Construye una expresión lógica que sea cierta si alguna de las notas es mayor que 5 (no necesariamente las dos)
  3. Construye una expresión lógica que sea cierta si una nota es igual a 7 y la otra está entre 5 y 7 (inclusive)

Ejercicio

- Crea un proyecto llamado **ResultadoQuiniela.py** que le pida al usuario el resultado de un partido de quiniela (1, X o 2). Según el dato introducido, el programa debe sacar por pantalla "Ha ganado el de casa" (si es un 1), "Han empatado" (si es una X) o "Ha ganado el de fuera" (si es un 2).

Estructuras repetitivas

- Hemos visto que las estructuras selectivas sirven para elegir entre varios caminos posibles cuando hay una condición que comprobar
- Las estructuras repetitivas nos van a permitir repetir varias veces un bloque de código, cuando sea necesario
  - Esto nos evitará tener que copiar y pegar varias veces el mismo código
  - También sirve para casos en los que no sabemos cuántas veces vamos a tener que repetir un código
- Al bloque de código ejecutado por una estructura repetitiva también se le llama **bucle**

Estructuras repetitivas: while

- La estructura **while** comprueba una condición (como la estructura *if*), y mientras esta condición se cumpla, ejecutará el bloque de código que tenga indentado (tabulado)

```
while condición:
    ... instrucciones
```

- Ejemplo: el siguiente ejemplo escribe el valor de n hasta que llega a 5 (sacaría los números del 0 al 4):

```
n = 0
while n < 5:
   print n
   n = n+1
```

Ejercicio

- Crea un proyecto llamado **Cuenta.py** que le pida al usuario un número entero (positivo) y realice la cuenta desde 0 hasta ese número.

Estructuras repetitivas: for

- La estructura **for** permite repetir un bloque de código un número determinado de veces. Le podemos dar diversos usos
  1. Usar el **for** para que una variable vaya tomando diversos valores (especificados entre corchetes) y repita el código del *for* cada vez con un valor

```
for dato in [2, 4, 8]:
   print dato * 2
# Sacaría por pantalla 4, 8 y 16
```

  2. Usar el **for** para que una variable vaya desde un valor inicial hasta uno final (indicados entre paréntesis), aumentándose de uno en uno automáticamente

```
for dato in range (1, 5):
   print dato * 2
# Sacaría por pantalla 2, 4, 6, 8 y 10
```

  3. Para lo mismo que el caso 2, pero en lugar de ir incrementándose de uno en uno, indicar nosotros el incremento como tercer dato del paréntesis

```
for dato in rante (0, 100, 10):
   print dato
# Sacaría por pantalla 0, 10, 20, 30, 40... hasta 100
```

Bucles anidados

- Cualquiera de estas estructuras se puede colocar dentro de otra, formando lo que se conoce como **bucles anidados**.
- Ejemplo: el siguiente ejemplo escribe el valor de n hasta que llega a 5. Para cada repetición, escribe 3 veces "Hola":

```
for n in range (0, 5):
    print n
    a = 0
    while a < 3:
        print "Hola"
        a = a + 1
```

break

- **break** es una instrucción que se usa para salir de un bloque de código (sale justo fuera de la llave de cierre)
- Ejemplo: en este ejemplo, el **break** nos haría salir del *while*, pero seguiríamos en el *for*

```
for n in range (0, 5):
    print n
    a = 0
    while a < 3:
        print "Hola"
        if a == 2:
            break
        a = a + 1
```

Ejercicio

- Crea un proyecto llamado **PruebaBreak.py** y copia dentro el código de la transparencia anterior. Prueba a cambiar la condición del *break* para ver cómo se comporta.