

TIPOS DE DATOS DE PYTHON

[PARTE 11]

Anteriormente hemos visto algunos de los tipos de datos que maneja Python. Vimos:

int , float , str , bool

También vimos como convertir datos a cada uno de estos tipos específicos. Para esto, hemos utilizado funciones propias de Python. Vimos:

int(), float(), str(), bool()

En los ejemplos realizados en el taller y en las prácticas que hemos realizado, hemos usado también otras funciones propias de Python, que sirven para hacer otras cosas.

También existen las instrucciones:

print() (imprimir datos en la pantalla),
raw_input() (ingresar datos con el teclado).

También hemos utilizado variables para almacenar nuestros datos y hemos realizado operaciones con ellos.

Para realizar operaciones, hemos utilizado “operadores”. Los operadores que hemos utilizado, dependen del tipo de operación a realizar con los datos.

Si vamos a hacer *operaciones matemáticas*, los operadores son:

+	suma
-	resta
/	división
*	multiplicación

Cuando comparamos datos, también utilizamos *operadores lógicos*. Ellos son:

and	y
or	o
not	no

Si vamos a realizar comparaciones entre datos, utilizamos los *operadores de comparación*:

==	igual que
!=	distinto que
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que

Para aclarar esto, veamos el siguiente ejemplo:

Supongamos que en un programa deseo hacer la siguiente prueba lógica:

En pseudocódigo:

Si naciste en Bogotá, eres colombiano.

En Python:

```
if naciste == "Bogotá":  
    print "Eres bogotano."
```

Según este ejemplo, sólo hacemos una comparación.

Esta comparación es ver donde naciste y si naciste en Bogotá, eres bogotano, porque si la condición **naciste == "Bogotá"** se cumple, devuelve **True** (la prueba lógica da verdadero) y se ejecuta lo que hay dentro del bloque **if**.



Ahora bien, podríamos mejorar el código ampliándolo, por ejemplo a los nacidos en Cali, de la siguiente manera:

En pseudocódigo:

Si naciste en Bogotá o en Cali, eres colombiano.

En Python:

```
if naciste=="Bogotá" or naciste=="Cali":  
    print "Eres colombiano"
```

Ahora, estamos haciendo 2 comparaciones en una misma prueba lógica. Probamos si naciste en Bogotá o en Cali y si naciste en uno de los dos lugares, eres colombiano. El operador lógico **or** sirve para enlazar las dos comparaciones en la misma prueba lógica de la sentencia **if**.

Ahora, la sentencia **if naciste == "Bogotá" or naciste == "Cali"**: (que es la prueba lógica), devolverá **True** siempre que al menos una de las dos comparaciones que la forman se cumpla. En el caso del operador lógico **and**, toda la sentencia **if** devolverá **True** en caso de que las dos comparaciones de la prueba se cumplan.

Otro operador lógico es la negación **not** el cual nos permite negar el resultado de una prueba lógica (**not True == False**).

Para aclarar mejor el operador **not**, retomamos el primer ejemplo y lo modificamos para hacer lo contrario:

En pseudocódigo:

Si no naciste en Bogotá, no eres bogotano.

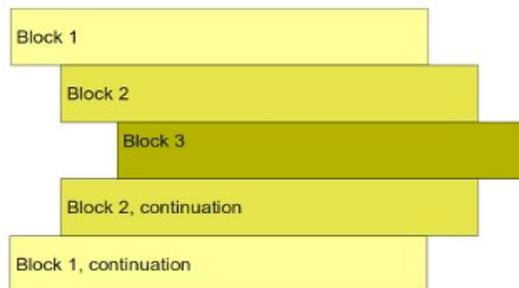
En Python:

```
if not naciste == "Bogotá":  
    print "No eres bogotano"
```

Este código imprimirá "No eres bogotano" siempre que **naciste** no sea igual a "Bogotá", porque **not** invierte el valor de la comparación para toda la prueba lógica, o sea, si **naciste** es igual a "Bogotá", la comparación es verdadera, pero al poner **not** delante, el valor de la prueba lógica se invierte.

Bloques de código:

En Python el código necesita ser dividido en bloques. Estos bloques se demarcan con una sangría de **cuatro espacios**.



Verificación de ejemplo anterior:

```
>>> naciste="Medellín"  
>>> if not naciste=="Bogotá":  
...     print ""No eres bogotano""  
...  
No eres bogotano
```

No eres bogotano

Control de Flujo

Si un programa no fuera más que una lista de órdenes a ejecutar una tras otra (de forma secuencial), no tendría mucha utilidad. Por eso, existen formas de probar condiciones y valores durante la ejecución de nuestro programa para decidir que partes de código queremos ejecutar. Estas formas nos permiten controlar como fluye nuestro programa e incluso hasta cuando se ejecuta, son las llamadas **sentencias de control de flujo**.

Ellas son: **Condicionales, bucles e iteradores**.



Sentencias condicionales

Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de estas condiciones.

Aquí es donde cobra importancia el tipo de datos booleano y los operadores lógicos y relacionales que aprendimos.

```
if condición:  
    #Código a ejecutar
```

La forma más elemental de un condicional es el `if` donde se evalúa una condición y si dicha condición se cumple, se ejecuta el código que se encuentra dentro de bloque.

Una estructura condicional más completa es **`if ... else:`**

```
if condición:  
    #Código a ejecutar si  
    #se cumple la condición  
else:  
    #Código a ejecutar si  
    #no se cumple la condición
```

La estructura anterior nos viene muy bien si queremos ejecutar un código cuando se cumple la condición y otro código distinto si no se cumple esa

condición, pero si queremos evaluar más de 2 posibles valores para la condición, debemos utilizar una estructura condicional mas completa, **`if ... elif ... elif ... else:`**

```
if numero < 0:  
    print "Negativo"  
elif numero > 0:  
    print "Positivo"  
else:  
    print "Cero"
```

En este código, primero se evalúa la condición del `if`.

Si es cierta, se ejecuta su código y se continúa ejecutando el código posterior al condicional;

si no se cumple, se evalúa la condición del `elif`.

Si se cumple la condición del `elif` se ejecuta su código y se continúa ejecutando el código posterior al condicional;

si no se cumple y hay más `elifs` se continúa con el siguiente `elif` en orden de aparición.

Si no se cumple la condición del `if` ni de ninguno de los `elif`, se ejecuta el código del `else`.

Bucles

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

El bucle `while` (*mientras*) ejecuta un fragmento de código mientras se cumpla una condición.

```
edad = 0  
while edad < 18:  
    edad = edad + 1  
    print "Felicidades, tienes " + str(edad)
```

Por ejemplo, veamos un pequeño programa que repite todo lo que el usuario escriba hasta que escriba "adios".

```
entrada = ""  
while entrada != "adios":  
    entrada = raw_input("> ")  
    print entrada
```



Para romper un bucle, también podemos utilizar la palabra clave **break**:

```
while True:
    entrada = raw_input("> ")
    if entrada == "adios":
        break
    else:
        print entrada
```

La palabra clave **break** (*romper*) sale del bucle en el que estamos.

Otra palabra clave que nos podemos encontrar dentro de un bucle es **continue**:

```
edad = 0
while edad < 18:
    edad = edad + 1
    if edad < 18:
        continue
    print "Felicidades, tienes " + str(edad)
```

Como ven, **continue** no hace otra cosa que pasar directamente a la siguiente iteración del bucle, por lo cual, en este código solo se imprimirá felicitaciones al cumplir 18 años.

Iteradores

for se utilizan en Python para recorrer secuencias de valores.

```
for x in range(1,10):
    print x
```

range es una función de python para especificar un rango de valores, **in** quiere decir *en*.

En este caso se especifica un rango de números del 1 al 9.

Este código, en pseudocódigo se leería así:

```
para x igual a cada número
en un rango de 1 a 9:
    imprimir el valor de x
```

JUEGO #1

```
print """En el campo me crie,
atada con verdes lazos,
y aquel que llora por me
me esta partiendo en pedazos."""

while True:
    respuesta = raw_input("quien soy?")

    if respuesta == "cebolla":
        print "Acertaste!"
        break
    else:
        print "Intenta otra vez"
```

Un buen sitio para aprender sobre estos temas: <http://www.mclibre.org/consultar/python/>

