Intr. al lenguaje de programación Python

Sentencias. Sintaxis. Estructuras de Control

Alejandro Roca Alhama

IES Cierva

Febrero de 2015

¿Qué vamos a ver? I

Introducción al leguaje de programación Python

¿Por dónde vamos?

Sentencias I

- Básicamente, una sentencia es una orden que se escribe para decirle a Python qué es lo que tiene que hacer.
- Python es un lenguaje procedimental, un lenguaje basado en sentencias y combinando estas sentencias se consigue que Python haga lo que queremos que haga.

Estructura de los programas I

La estructura de un programa en Python se resume en:

- Los **programas** están compuestos de módulos.
- Los **módulos** contienen sentencias.
- Las **sentencias** contienen expresiones.
- Las **expresiones** crean y procesan objetos.

Básicamente, la sintaxis de Python se compone de **sentencias y expresiones**.

- Una sentencia es un conjunto de expresiones. Una sentencia codifica una operación del programa.
- La expresiones procesan objetos, calculan valores. Podemos crear expresiones a partir de más expresiones.

Las sentencias en Python I

Podríamos resumir todas las sentencias de Python en la siguiente tabla:

Sentencia	Rol	Ejemplo
Asignación	Crea referencias	a = 8
Llamadas	Ejecutar funciones	len('Superman')
print	Imprime objetos	print a
if/elif/else	Selecciona acciones	
for	Bucles	
while	Bucles	
pass	Sentencias vacías	
break, continue	Salida de bucles	
def	Definir funciones y métodos	
return	Resultados de funciones	return a

Las sentencias en Python y II

(continuación)

Sentencia	Rol	Ejemplo
import	Acceso a módulos	import sys
from	Acceso a atributos	from os import kill
class	Construir objetos	class Persona()
try/except/finally	Manejo de excepciones	
raise	Lanzar excepciones	
assert	Comprobaciones de depuración	assert x ¿ y
del	Eliminar referencias	del profesor
Otras	yield, global, nonlocal, with/as...	

Sintaxis de Python I

Comparando con la sintaxis de C/C++/Java/PHP/Perl

En C, como en otros lenguajes escribiríamos este código:

```
if ( x > y ) {
    x = 1;
    y = 2;
}
```

En Python sin embargo escribiríamos este otro:

```
if x > y:
    x = 1
    y = 2
```

Python vs "resto de lenguajes" I

Al estilo Python

- En Python se escribe menos código, y éste se parece mucho al pseudocódigo.
- (+) Python añade los dos puntos después de la condición para abrir un nuevo bloque de sentencias.
  - Los : son obligatorios.
  - Cuesta acostumbrarse.
- (-) Los paréntesis alrededor de la condición son optativos. Lo normal es que no se pongan.
- (-) El final de línea marca el final de la sentencia, NO ES NECESARIO indicarlo con un ;
  - En Python una línea una sentencia.
  - No es un error poner el ';', pero se ruega NO UTILIZARLO.
- (-) El final de la identación es el final del bloque.
  - No es necesario utilizar $\{...\}$ ó begin ... end como en otros lenguajes.
  - Terminar la identación =>terminar bloque de sentencias.

¿Por qué la identación? I

En Python la identación es obligatoria. Esto hace que:

- Los programadores produzcan un código uniforme, regular y legible.
- La codificación sigue la estructura lógica del programa.
- Se genera un código más mantenible, reutilizable y legible para otros.
- Solo hay una forma de hacer las cosas.
- Menos propenso a errores.

Introducción al leguaje de programación Python

¿Por dónde vamos?

Sentencias de asignación I

- Un asignación asigna objetos a nombres.
- Básicamente tiene este formato:

```
var = objeto
```

  - Indicamos un destino a la izquierda del signo igual.
  - Indicamos un objeto a la derecha del signo igual.

Asignaciones I

A recordar

Las asignaciones son muy sencillas, pero hay que recordar.

- **Las asignaciones crean referencias a objetos**.
  - Python no es como el resto de lenguajes.
  - En Python, una asignación almacena referencias a objetos en nombres o estructuras de datos.
  - No tienen mayores implicaciones, pero hay que recordarlo.
- **Los nombres se crean durante la primera asignación**.
  - No hay que declarar las variables.
  - Todas las estructuras de datos necesarias se crean durante la asignación.
  - Una vez asignada la variable, se utilizará en cualquier expresión que aparezca.
- **Se debe hacer una asignación antes de la primera referencia**.
  - Es un error utilizar un nombre antes de haber realizado la asignación.

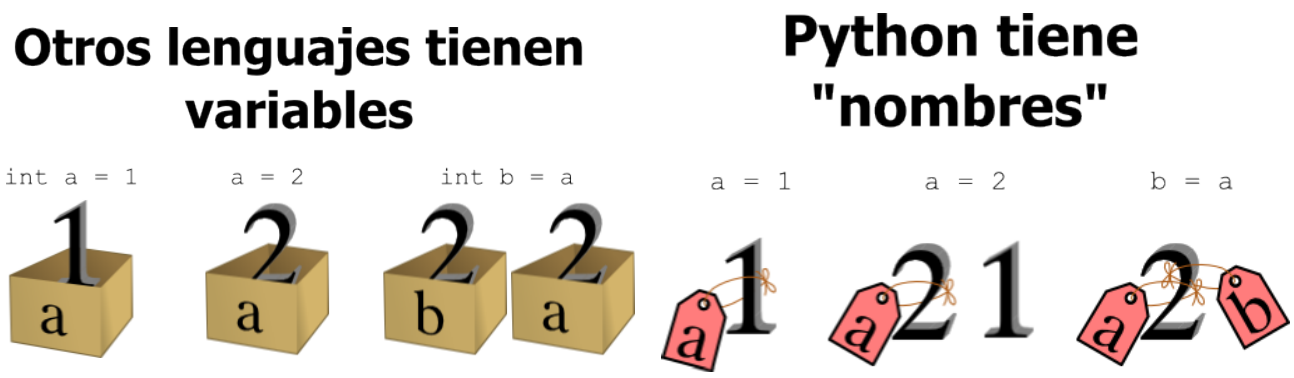
Asignaciones II

A recordar

  - Si se usa sin inicializar no se asigna ningún valor por defecto, se produce un error, se lanza una excepción.
- **Algunas operaciones realizan asignaciones implícitas**.
  - Las asignaciones no se producen únicamente con sentencias $=$.
  - También se producen asignaciones durante la importanción de módulos, definición de funciones, definición de clases, bucles for, argumentos en las llamadas a funciones, etc.

Asignaciones como referencias I

C vs Python



Asignaciones I

Otras formas de realizar asignaciones

- Forma básica:

```
heroe = "Superman"
a = 8
```

- Asignación posicional (tuplas):

```
a, b = 5, 3
```

- Asignación con listas:

```
[a, b] = [5, 3]
```

- Asignación secuencial:

```
a, b, c, d = "Alex"      # a <- 'A', b <- 'l' ...
```

Asignaciones II

Otras formas de realizar asignaciones

- Asignación múltiple:

```
x = y = z = 8.0
```

- Asignación compuesta:

```
var += 23    # Equivalente a var = var + 23
```


Nombres de variables I

Reglas para la elección de nombres...

El nombre elegido para las variables debe seguir unas reglas bien definidas:

- **Sintaxis: (_ ó letra) + (conjunto de letras, dígitos y _)**.
  - Ejemplos: _valor, valor, valor8, valor_8...
  - No legales: 1valor, valor$...
- **Sensible a las mayúsculas**.
  - No son la misma variable: var, VAR, Var, VaR, vAr...
- **No usar como variables palabras reservadas del lenguaje**.
  - Como True, for, while, import, ...

Nombres de variables I

Convenciones

Al elegir un nombre para una variable seguiremos las siguientes convenciones (no son obligatorias, pero altamente recomendadas):

- Los nombres que comienzan con _ (como _X) no se importan con un *from module import **.
- Nombres de la forma __X__ son nombres definidos por el sistema y tienen un significado especial para el intérprete.
- No se deben utilizar nombres de la forma __X ya que se usan para atributos "privados" de las clases.
- _ se utiliza para almacenar el resultado de la última expresión cuando se trabaja con el intérprete de forma interactiva.
- Hay que elegir nombres que reflejen el uso de la variable:
  - iva, importe_total, cantidad, suma, apellidos, direccion...
  - ¿Para qué se está utilizando la variable h32?

Nombres de variables II

Convenciones

- Para contadores en bucles se pueden usar i, j, k...
- Si el identificador está compuesto por varias palabras, se utilizan dos convenciones:
  - Todos los identificadores en minúsculas, las palabras se separan con el carácter _
    - Ejemplos: base_imponible, suma, importe_total ...
  - CamelCase: todas las palabras juntas, cada una empezando en mayúsculas excepto la primera (estilo C++/Java).
    - Ejemplos: baseImponible, suma, importeTotal
- Elegir idioma y mantenerlo. Todos los nombres en inglés ó todos los nombres en castellano.

Nombres y objetos I

Respecto al tipo...

¡¡ Los objetos tienen tipo !!

Los nombres (variables) son solo referencias a objetos, no poseen un tipo.

```
x = 0                  # x está ligado a un entero
x = "Hola a todos"     # x está ahora ligado a una cadena
x = [0, 1, 2]          # ... y ahora a una lista
```


Expresiones como sentencias I

En Python se pueden usar expresiones como si fuesen sentencias.

- Solo en una línea.
- El resultado no se almacena, no hay asignación.
- Solamente tienen sentido si la expresión hace algo útil como efecto secundario.
- Dos situaciones:
  - Para llamadas a funciones y métodos.
  - Mostrar valores con el prompt interactivo.

print() I

- In Python, *print()* imprime cosas, es simplemente una interfaz amigable para la salida estándar (stdout).
- Tećnicamente, lo que hace *print* es:
  1. Convierte uno o más objetos a su representación textual.
  2. Realiza ciertas modificaciones de formato.
  3. Envía el texto resultante a la salida estándar o a un fichero.

print vs print() I

Python 2.X vs Python 3.X

Hay una gran diferencia entre la versión 2 y la versión 3 de print:

- En Python 3, *print* es una función interna.
- En Python 2, *print* es una sentencia.

De momento no se le va a dar más importancia, pero en Python 3 las cosas cambian.

La sentencia print I

print en acción

```
>>> x = 'a'
>>> y = 'b'
>>> print(x, y)
a b
>>> print(x, y, end = ' '); print(x, y)
a b a b
>>> print(x + y)
ab
>>> a = "Superman"
>>> b = 8
>>> print(a + str(b))
Superman8
>>> print(a, str(b))
Superman 8
>>> print(a, str(b), sep = '')
Superman8
>>>
```


Introducción al leguaje de programación Python

¿Por dónde vamos?

3. Condicionales: la sentencia if
   - Expresiones lógicas
   - La sentencia if
   - La sentencia if...else
   - if anidados. if...elif

Estructuras de control I

Teorema de la programación estructurada

- En mayo de 1966, Böhm y Jacopini establecieron que toda función computable puede ser implementada en un lenguaje de programación que combine solamente tres tipos de estructuras de control:
  - Secuenciales.
    - Ejecutar una(s) sentencia(s) o instrucción(es) y luego otra(s).
  - **Selectivas**.
    - **Ejecutar una de dos sentencias, dependiendo del valor de una variable booleana**.
  - Repetitivas.
    - Ejecutar una sentencia mientras una variable booleana sea 'verdadera' (iteración, ciclo o bucle).

Expresiones lógicas I

Introducción

- Varias de las sentencias de Python necesitan chequear el valor de una expresión para comprobar si es **verdadera** o **falsa**.
  - Tanto las sentencias de selección como de repetición basan su funcionamiento en el resultado de alguna prueba condicional.
- En Python:
  - Verdadero ->Valor 1 ->**True**.
  - Falso ->Valor 0 ->**False**.

Expresiones lógicas I

Más

- Una expresión lógica es una expresión que solo puede tomar dos valores:
  - True.
  - False.
- Se denominan también **expresiones booleanas**.
  - En honor al matemático británico George Boole, que desarrolló el **Álgebra de Boole**.
- Las expresiones lógicas se forman combinando:
  - Constantes y variables lógicas.
  - Operadores relacionales.
  - Operadores lógicos.

Operadores relacionales I

- Los operadores relacionales de Python se corresponden con los operadores matemáticos $<$, $\leq$, $>$, $\geq$; con la excepción de que producen un resultado que puede ser:
  - True (Verdadero)
  - False (Falso).
- ... muy útil con sentencias selectivas y repetitivas.

Operadores relacionales I

Más operadores

- Los operadores relacionales son operadores binarios, se utilizan de la siguiente forma:

expresion1 OperadorRelacional expresion2

- Los operadores relacionales son:

Operador	Significado
$<$	Menor que
$>$	Mayor que
$<=$	Menor o igual que
$>=$	Mayor o igual que
$==$	Igual a
$!=$	Distinto de

Operadores relacionales I

En acción

- Los operadores relacionales se pueden utilizar para comparar enteros y reales indistintamente:

```
>>> 1 < 2.5
True
>>> 5.6 < 4
False
>>> 2 < 5 < 8
True
>>> a = 10
>>> a < 10
False
>>> a <= 10
True
>>> a == 10
True
>>> a != 10
False
```

Operadores relacionales II

En acción

```
>>>
```


Operadores relacionales I

Precedencia

- Tienen una precedencia menor que los operadores aritméticos:

```
i + j < k - 1      es equivalente a     (i + j) < (k - 1)
```

- Los operadores == y != tienen menor precedencia que $<$, $\leq$, $>$, $\geq$

```
i < j == j < k      es equivalente a      (i < j) == (j - k)
```

  - ... que es verdadera si ambas subexpresiones son verdaderas o ambas son falsas.

Operadores relacionales I

Ejemplos...

```
a = 4
b = 3
a > b
(a - 2) < (b - 4)
3 > 6
5 < 5
5 >= 5
4 > a
4 >= a
(5 - a) != 1
(a + b) == 7
```


Operadores relacionales I

... y sus soluciones

```
a = 4
b = 3
a > b                # True
(a - 2) < (b - 4)    # False
3 > 6                # False
5 < 5                # False
5 >= 5               # True
4 > a                # False
4 >= a               # True
(5 - a) != 1         # False
(a + b) == 7         # True
```

Operadores lógicos I

- Los operadores lógicos nos proporcionan un resultado a partir de que se cumpla o no una determinada condición.
- Permiten construir expresiones lógicas más complejas.
- Los operadores lógicos en Python son:

Operador	Significado
not	Negación, no lógico
and	Y lógico (and)
or	O logico (or)

Operadores lógicos I

not y and

not

a	not a
False	True
True	False

and

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

Operadores lógicos I

or

or

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

Operadores lógicos I

Ejemplos...

```
(1 > 0) and (3 == 3)
not True
not not True
(0 < 5) or (0 > 5)
(5 <= 7) and (2 > 4)
not (5 != 5)
(5 == 1) or (7 >= 4)
```

Operadores lógicos I

... y sus soluciones

```
(1 > 0) and (3 == 3)   # True
not True               # False
not not True           # True
(0 < 5) or (0 > 5)     # True
(5 <= 7) and (2 > 4)   # False
not (5 != 5)           # True
(5 == 1) or (7 >= 4)   # True
```


Valores Booleanos I

Algunas cosas más sobre True y False

Los operadores que actúan sobre los booleanos se comportan de forma un "poco" diferente en Python en determinadas situaciones:

- Cualquier número distinto de cero es *True*.
- Cualquier objeto no vacío es *True*.
- El cero, los objetos vacíos, y el objeto *None* se consideran *False*.
- Las comparaciones y tests de igualdad se aplican recursivamente a estructuras de datos.
- Las comparaciones y tests de igualdad devuelven *True* o *False*.
- Las operaciones booleanas *and* y *or* devuelven un objeto operando que puede ser verdadero o falso.

Valores Booleanos I

Algunas cosas más sobre True y False: ejemplos

```
>>> if 8: print("True")
...
True
>>> a = 4
>>> if a: print("True")
...
True
>>> if None: print("True")
...
>>> if 0: print("True")
...
>>> l=list()
>>> if l: print("True")
...
>>> l.append(8)
>>> if l: print("True")
...
True
>>> 8 or 0
```


Valores Booleanos II

Algunas cosas más sobre True y False: ejemplos

```
8
>>> 0 or 0
0
>>> 8 or "alex"
8
>>> 0 or "alex"
'alex'
```

¿Por qué una sentencia if? I

Selección, selección, decisiones

En los programas siempre se necesita hacer cosas dependiendo de la entrada del usuario o de la situación en concreto:

- **Si** el jugador acierta la pregunta consigue un punto.
- **Si** el jugador destruye la nave se oye una explosión.
- **Si** el fichero existe lo leemos, **si no** existe mostramos un mensaje de error.

Para tomar decisiones, los programas debe comprobar si determinadas condiciones son ciertas (True) o no lo son (False).

La sentencia if I

- Primera sentencia compuesta, es decir, sentencia que agrupa un conjunto de sentencias.
- La sentencia *if* permite a un programa elegir entre dos alternativas comprobando el valor de una expresión.
- Su forma más sencilla es:

```
if expresion: sentencia

if expresion:
    sentencia1
    sentencia2
    ...
```

Ejecución de una sentencia if I

En la ejecución de una sentencia *if*:

- Se evalúa la expresión.
- Si la expresión es verdadera se ejecuta(n) la(s) sentencia(s) asociada(s).
- Si la expresión es falsa, la ejecución del programa prosigue después del if (y de sus sentencias asociadas).

Ejecución de una sentencia if I

if sin else

Sentencia if I

Comprobación de rangos

- Es muy usual utilizar un if para comprobar si el valor de una variable cae dentro de un rango:

```
if 0 <= i and i < 10:
    print("Dentro")

if i < 0 or i >= 10:
    print("Fuera")

# Con paréntesis:
if (i < 0) or (i >= 10):
    print("Fuera")
```


Sentencia if I

Sentencias compuestas

- Una sentencia compuesta se identa. NO SE NECESITAN ni llaves ni bloques *begin...end* como en otros lenguajes.

Un if con varias sentencias sería así:

```
if a > 255:
    numero_lineas = 8
    j = 8
print("Este print siempre se ejecuta, está fuera del if")
```

La sentencia if...else I

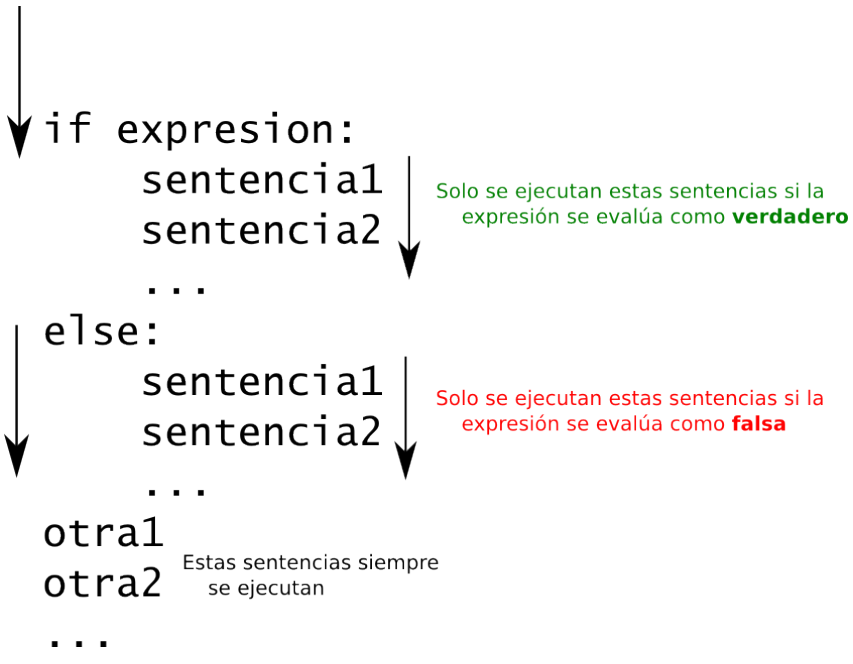
- La sentencia *if* puede ir acompañada de una cláusula *else*:

```
if expresion:
    sentencias
else:
    sentencias
```

- La(s) sentencia(s) de la cláusula else solo se ejecutarán si la expresión se evalúa como falsa (False).

Sentecia completa if + else I

Por si sí o por si no



La sentencia if...else I

Ejemplo

```
if (i > j):
    a = 0
    num_lineas = num_lineas + 1
else:
    b = 4
    num_lineas = 0
```

- A diferencia de otros lenguajes la identación es suficiente para agrupar sentencias.
- La sentencias no acaban en ;
- Aunque se recomienda la anterior identación, ésta también sería legal:

```
if (i > j): a = 0 ; num_lineas = num_lineas + 1
else: b = 4 ; num_lineas = 0
```


La sentencia if...else I

Ejemplo

Listing 1: Mayor de edad

```
#!/usr/bin/env python3

edad = int(input("¿Cuántos años tienes? "))

if edad >= 18:
    print("Con", edad, "año(s) eres mayor de edad")
else:
    print("Con", edad, "año(s) eres menor de edad")
```

La sentencia if...else I

Ejemplo

Listing 2: Mayor de dos números

```
#!/usr/bin/env python3

x = int(input("Introduzca el primer número : "))
y = int(input("Introduzca el segundo número: "))

if x > y:
    print(x, "es mayor que", y)
else:
    print(y, "es mayor (o igual) que", x)
```


La sentencia if...else I

Ejemplo

Listing 3: Mayor de dos números mejorado

```
#!/usr/bin/env python3

x = int(input("Introduzca el primer número : "))
y = int(input("Introduzca el segundo número: "))

if x == y:
    print("Los números son iguales")
elif x > y:
    print(x, "es mayor que", y)
else:
    print(y, "es mayor que", x)
```

Juego: tablas de multiplicar I

Ejemplo

- El programa nos pregunta las tablas de multiplicar, nos felicita si acertamos y nos riñe si fallamos.

Listing 4: Tablas de multiplicar

```
#!/usr/bin/env python3

import random

n1 = random.randint(0, 10)
n2 = random.randint(0, 10)

solucion = n1 * n2
pregunta = str(n1) + " x " + str(n2) + " = "

respuesta = int(input(pregunta))

if respuesta == solucion:
    print("Acertaste!!!!")
else:
    print("Has fallado,", n1, "x", n2, "=", solucion)
```


if anidados I

Nested if

- No existe ninguna restricción sobre qué tipo de sentencias pueden haber dentro de un *if*, puede haber cualquier otra... como otro *if*...
  - Dados tres números i, j, k: ¿cuál es mayor?

```
if i > j:
    if i > k:
        max = i
    else:
        max = k
else:
    if j > k:
        max = j
    else:
        max = k
```

if anidados I

Ejemplo

Listing 5: La hora siguiente

```
#!/usr/bin/env python3

print("Introduzca una hora hh:mm:ss")
horas = int(input("Introduzca las horas: "))
minutos = int(input("Introduzca los minutos: "))
segundos = int(input("Introduzca los segundos: "))

if segundos == 59:
    segundos = 0
    if minutos == 59:
        minutos = 0
        if horas == 23:
            horas = 0
        else:
            horas = horas + 1
    else:
      minutos = minutos + 1
else:
    segundos = segundos + 1

print("Hora: ", horas, ":", minutos, ":", segundos, sep = '')
```


Sentencias if en cascada I

Ejemplo (entero1.py)

- Se utilizan en aquellas ocasiones en las que tenemos que testear un conjunto de condiciones parando tan pronto una sea verdadera.

```
#!/usr/bin/env python3

n = int(input("Número: "))

if n == 0:
    print("El número es igual a cero.")
else:
    if n > 0:
        print("El número es mayor que cero.")
    else:
        print("El número es menor que cero.")
```

Sentencias if en cascada I

Ejemplo (entero2.py)

- El código anterior se puede escribir con la construcción *if..elif..elif ... else*:

```
#!/usr/bin/env python3

n = int(input("Número: "))

if n == 0:
    print("El número es igual a cero.")
elif n > 0:
    print("El número es mayor que cero.")
else:
    print("El número es menor que cero.")
```


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

Justificación

```
if expr:
    sentencia
elif expr:
    sentencia
elif expr:
    sentencia
...
elif expr:
    sentencia
else:
    sentencia
```

- Esta forma de escribir reduce la identación excesiva para tests muy largos.
- No es una sentencia nueva, sino un *if* dentro de un *else* dentro de un *if* dentro de un *else*...
- En Python no existe la sentencia *switch*.

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

Ejemplo...

- Queremos calcular la comisión de un broker sabiendo que depende del importe de la transacción de esta forma:
  - La mínima comisión son \$39.
  - El resto depende de la transacción según el tamaño:
    - Menos de \$2500 =>Comisión de \$30 + 1.7 %
    - De \$2500 a \$6250 =>Comisión de \$56 + 0.66 %
    - De \$6250 a \$20000 =>Comisión de \$76 + 0.34 %
    - De \$20000 a \$50000 =>Comisión de \$100 + 0.22 %
    - De \$50000 a \$500000 =>Comisión de \$155 + 0.11 %
    - Más de \$500000 =>Comisión de \$255 + 0.09 %

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

Otro ejemplo

Listing 6: Cálculo de la comisión de un broker

```
#!/usr/bin/env python3

cantidad = float(input("Introduzca la cantidad (en euros): "));

if cantidad < 2500.0:
    comision = 30.0 + 0.017 * cantidad
elif cantidad < 6250.0:
    comision = 56.0 + 0.0066 * cantidad
elif cantidad < 20000.0:
    comision = 76.0 + 0.0034 * cantidad
elif cantidad < 50000.0:
    comision = 100.0 + 0.0022 * cantidad
elif cantidad < 500000.0:
    comision = 155.0 + 0.0011 * cantidad
else:
    comision = 255.0 + 0.0009 * cantidad
if comision < 39.0:
    comision = 39.0

print("La comisión es:", comision , "euros")
```

Problema de los elses "colgantes" I

¡Problema de C, no de Python!

- Ejemplo de código en C:

```
if (y != 0)
    if (x != 0)
        resultado = x / y;
else
    printf("Error: y es igual a 0\n");
```

- ¿A qué sentencia *if* pertenece el *else*?
  - La identación sugiere el primero, pero realmente pertenece al segundo.

¡El compilador de C ignora la identación!
¿Pasa esto en Python?

Expresiones condicionales I

- La sentencia if de Python permite realizar unas acciones u otras dependiendo del valor de una condición.
- Python además proporciona una construcción que permite a una expresión producir un valor u otro según una condición.
  - Muy similar a la expresión de C/C++/Java:

    expr1 ? expr2 : expr3
- En Python podemos escribir:

  var = "par" if (num % 2 == 0) else "impar"

Introducción al leguaje de programación Python

¿Por dónde vamos?

Estructuras de control I

Teorema de la programación estructurada

- En mayo de 1966, Böhm y Jacopini establecieron que toda función computable puede ser implementada en un lenguaje de programación que combine solamente tres tipos de estructuras de control:
  - Secuenciales.
    - Ejecutar una(s) sentencia(s) o instrucción(es) y luego otra(s).
  - Selectivas.
    - Ejecutar una de dos sentencias, dependiendo del valor de una variable booleana.
  - **Repetitivas.**
    - **Ejecutar una sentencia mientras una variable booleana sea 'verdadera' (iteración, ciclo o bucle).**

Bucles I

- Para la realización de bucles, Python cuenta con las sentencias:
  - while
  - for
- Ambas sentencias permiten iterar sobre un conjunto de sentencias mientras que se cumpla una condición o durante un número determinado de veces.
- Para el trabajo con bucles, Python también soporta las sentencias *break* y *continue*.

El bucle while I

- El bucle while es el bucle más general de Python.
- Un bucle while ejecuta un bucle de sentencias de forma repetida mientras que se cumpla una determinada condición.

El bucle while I

Funcionamiento

En un bucle *while*:

- Se examina la condición.
- Si la condición es cierta se ejecutan las sentencias dentro del bucle.
- Si la condición no es cierta el control pasa a las sentencias más allá del bucle.

El bucle while I

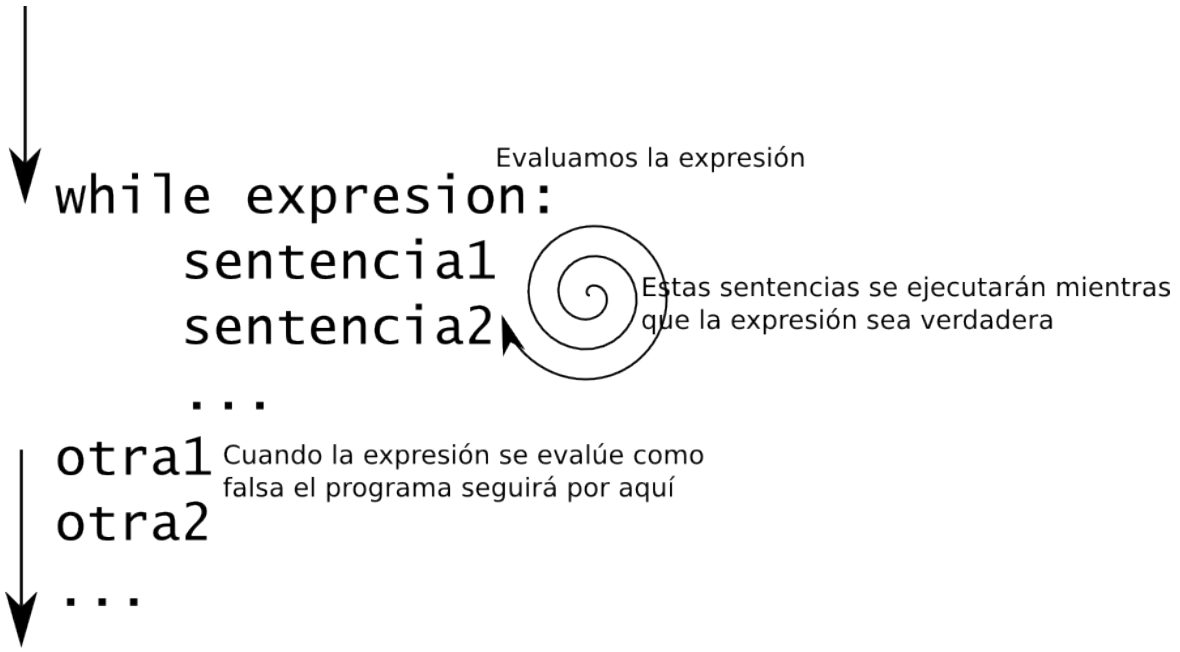
Forma general

La forma general del bucle *while* es la siguiente:

```
while <expresion>:
    sentencia1
    sentencia2
    ...
```

El bucle while I

Repetimos...



El bucle while I

Ejemplo (while1.py)

```
#!/usr/bin/env python3

while True:
    print("¡¡Hola!!",)
    print("Para parar esto hay que pulsar Ctrl + C")
```

El bucle while I

Ejemplo: contamos de 0 a 4 (while2.py)

```
#!/usr/bin/env python3

print("Vamos a contar de 0 a 4:")
i = 0
while i < 5:
    print(i)
    i = i + 1
```


El bucle while I

Funcionamiento: ¿cómo contamos de 0 a 4?

La ejecución del ejemplo anterior sería:

1. i = 0;
2. ¿ i <5 ? Sí.
3. Mostramos 0. Incrementamos i. Ahora i = 1.
4. ¿ i <5 ? Sí.
5. Mostramos 1. Incrementamos i. Ahora i = 2.
6. ¿ i <5 ? Sí.
7. Mostramos 2. Incrementamos i. Ahora i = 3.
8. ¿ i <5 ? Sí.
9. Mostramos 3. Incrementamos i. Ahora i = 4.
10. ¿ i <5 ? Sí.
11. Mostramos 4. Incrementamos i. Ahora i = 5.
12. ¿ i <5 ? ¿ 5 <5 ? **No**. Salimos del bucle.

Ejemplo: tabla de multiplicar I
(SIN bucles)

```
#!/usr/bin/env python3

numero = int(input("Introduzca un número: "))

print("La tabla de multiplicar del", numero)
print(numero, "x", 0, "=", numero * 0)
print(numero, "x", 1, "=", numero * 1)
print(numero, "x", 2, "=", numero * 2)
print(numero, "x", 3, "=", numero * 3)
print(numero, "x", 4, "=", numero * 4)
print(numero, "x", 5, "=", numero * 5)
print(numero, "x", 6, "=", numero * 6)
print(numero, "x", 7, "=", numero * 7)
print(numero, "x", 8, "=", numero * 8)
print(numero, "x", 9, "=", numero * 9)
print(numero, "x", 10, "=", numero * 10)
```


Ejemplo: tabla de multiplicar I
(con bucle while)

```
#!/usr/bin/env python3

numero = int(input("Introduzca un número: "))

print("La tabla de multiplicar del", numero)
i = 0
while i <= 10:
    print(numero, "x", i, "=", numero * i)
    i = i + 1
```

Sobre while I

Puntualizaciones

Observaciones sobre la sentencia *while*:

- La expresión que controla el bucle while toma el valor falso cuando el bucle termina.
- Es posible que el cuerpo de un bucle while no llegue a ejecutarse nunca, ya que lo primero que comprueba es la expresión.
- Un mismo bucle se puede escribir de varias formas.
- Es fácil cometer errores y provocar un bucle infinito.

Bucles infinitos I

La historia interminable...

- Un bucle while no terminará nunca si la expresión que lo controla siempre es verdadera.
- Un programador puede crear un bucle infinito deliberadamente así:

```
while True:
    ...

while 1:
    ...
```

- Estos dos bucles se ejecutarán infinitamente salvo que el cuerpo contenga una sentencia de transferencia de control fuera del bucle (break, goto, return) o llame a una función que haga que el programa termine.

Otro ejemplo: suma de los 100 primeros números I

Sumamos todos los números de 0 a 100 (suma100.py)

```
#!/usr/bin/env python3

suma = 0
i = 0
while i <= 100:
    suma = suma + i
    i = i + 1

print("La suma de los", 100, "primeros números es:", suma)
```


Otro ejemplo: tabla de potencias I

Potencias de 2 a 5 de los números del 0 al 100 (potencias.py)

```
#!/usr/bin/env python3

numero = 0
while numero <= 100:
    print(numero, ":", sep = '', end = ' ')
    i = 2
    while i <= 5:
        print(numero ** i, end = ' ')
        i = i + 1
    numero = numero + 1
    print()
```

Otro ejemplo: calcular el número de dígitos de un número entero I

2345 tiene 4 dígitos (digitos.py)

```
#!/usr/bin/env python3

numero = int(input("Introduzca un número: "))

digitos = 1
n = numero // 10
while (n > 0):
    n = n // 10
    digitos = digitos + 1

if digitos > 1:
    print("El número", numero, "tiene", digitos, "dígitos.")
else:
    print("El número", numero, "tiene", digitos, "dígito.")
```


Otro ejemplo: menú de opciones I

Si necesitamos un menú de opciones

```
#!/usr/bin/env python3

correcta = False
while not correcta:
    print("1.- Opción 1")
    print("2.- Opción 2")
    print("3.- Opción 3")
    print("4.- Opción 4")
    opcion = input("Introduzca un número: ")
    if (opcion == '1' or opcion == '2' or
        opcion == '3' or opcion == '4'):
           correcta = True

print("Ha elegido la opción:", opcion)
```

El bucle for I

- En Python, el bucle *for* es algo distinto al de otros lenguajes como C/C++ ó Java.
- El bucle *for* es un iterador de secuencias general. Permite recorrer todos los elementos de cualquier secuencia ordenada.
- *for* permite recorrer cadenas, listas, tuplas, etc.

El bucle for I

Forma general

La forma general del bucle *for* es la siguiente:

```
for <elemento> in <secuencia>:
    sentencia1
    sentencia2
    ...
```

El bucle for I

Funcionamiento

Cuando Python ejecuta un bucle for:

1. Coge el primer elemento de la *secuencia*.
2. Lo asigna a *elemento*.
3. Ejecuta todas las sentencias del cuerpo del bucle.
4. Coge el siguiente elemento de la secuencia.
5. Lo asigna a *elemento*.
6. Ejecuta todas las sentencias del cuerpo del bucle.
7. ...

Esto se ejecuta hasta terminar con todos los elementos de la secuencia.

El bucle for I

Ejemplo con una lista

```
#!/usr/bin/env python3

# Definimos una lista de superhéroes
superheroes = [ "Superman", "Batman", "Green Lantern", "Wonder Woman" ]
nombre = "Hal Jordan"

# Recorremos la lista
for heroe in superheroes:
    print(heroe)

print
# Recorremos todos los caracteres de una cadena
for car in nombre:
    print(car, "-", end = ' ')
```

El bucle for I

La función range

La función **range** tiene la siguiente definición:

range([start], stop[, step]) ->lista de enteros

- La función *range* devuelve una lista de enteros (progresión aritmética).
- range(i, j) ->[i, i+1, i+2, i+3, ..., j-1]
- Por defecto, *start* es cero.
- *step* especifica el incremento (o decremento).

En Python 2.X se recomienda el uso de *xrange* en vez de *range*.

range I

Ejemplos

```
>>> for i in range(10): print(i, end = ' ')
...
0 1 2 3 4 5 6 7 8 9
>>> for i in range(1, 11): print(i, end = ' ')
...
1 2 3 4 5 6 7 8 9 10
>>> for i in range(10, 0, -1): print(i, end = ' ')
...
10 9 8 7 6 5 4 3 2 1
>>> for i in range(0, 21, 2): print(i, end = ' ')
...
0 2 4 6 8 10 12 14 16 18 20
>>>
```

El bucle for I

Ejemplos: ejercicio

Hacer un pequeño programa que cuente:

- De 0 a 9, usando range(x).
- De 0 a 9, usando range(x, y).
- De 0 a 10.
- De 10 a 0.
- De 0 a 20 de 2 en 2.

El bucle for I

Y la solución...

```
#!/usr/bin/env python3

print("Contamos de 0 a 9:")
for i in range(10):
    print(i, end = ' ')

print("\n")


print("Contamos de 0 a 9:")
for i in range(0, 10):
    print(i, end = ' ')

print("\n")

print("Contamos de 0 a 10:")
for i in range(0, 11):
    print(i, end = ' ')

print("\n")

print("Contamos de 10 a 0:")
for i in range(10, -1, -1):
    print(i, end = ' ')

print("\n")
```

El bucle for II

Y la solución...

```
print("Contamos de 0 a 20 de 2 en 2:")
for i in range(0, 21, 2):
    print(i, end = ' ')

print("\n")
```


El bucle for I

Más ejemplos

```
# Contar desde 0 hasta n - 1:
    for i in range(0, n):
        ...
    for i in range(n):
        ...
# Contar desde 1 hasta n:
    for i in range(1, n + 1):
        ...
# Contar desde n - 1 hasta 0:
    for i in range (n - 1, -1, -1):
        ...
# Contar desde n hasta 1:
    for i in range (n, 0, -1):
        ...
```

Ejemplo: tabla de multiplicar I
(SIN bucles)

```
#!/usr/bin/env python3

numero = int(input("Introduzca un número: "))

print("La tabla de multiplicar del", numero)
print(numero, "x", 0, "=", numero * 0)
print(numero, "x", 1, "=", numero * 1)
print(numero, "x", 2, "=", numero * 2)
print(numero, "x", 3, "=", numero * 3)
print(numero, "x", 4, "=", numero * 4)
print(numero, "x", 5, "=", numero * 5)
print(numero, "x", 6, "=", numero * 6)
print(numero, "x", 7, "=", numero * 7)
print(numero, "x", 8, "=", numero * 8)
print(numero, "x", 9, "=", numero * 9)
print(numero, "x", 10, "=", numero * 10)
```


Ejemplo: tabla de multiplicar I
(con bucle for)

```
#!/usr/bin/env python3

numero = int(input("Introduzca un número: "))

print("La tabla de multiplicar del", numero)
for i in range(11):
    print(numero, "x", i, "=", numero * i)
```

Ejemplo: todas las tablas de multiplicar I

(con bucle for)

```
#!/usr/bin/env python3

for numero in range(11):
    print("La tabla de multiplicar del", numero)
    for i in range(11):
        print(numero, "x", i, "=", numero * i)

    print('-' * 30)
```


break, continue y pass I

Importantes para el control de bucles son las sentencias break y continue. Sus funciones son:

- **break** Salta fuera del bucle.
- **continue** Salta a la siguiente iteración.
- **pass** No hace nada, simplemente permite indicar una sentencia que no hace nada.

break I

Descripción

- La sentencia *break* causa una salida inmediata del cuerpo de un bucle (*for* y *while*).
- La sentencia *break* es especialmente útil para escribir bucles en los que el punto de salida está en la mitad del cuerpo, ni al principio ni al final.
- Ejemplo: bucle que termine cuando el usuario teclea una entrada determinada.
  - Escribir un programa que calcule el cubo de un número que el usuario introduce por teclado, el programa debe terminar cuando el usuario introduzca el número cero.

break I

Ejemplo

```
#!/usr/bin/env python3

while True:
    n = int(input("Introduce un número: "))

    if n == 0:
        break
    else:
        print("El cubo de", n, "es", n **3)
```

break I

Ejemplo: comprobar si un número es primo

Una primera aproximación podría ser dividir n entre todos los números comprendidos entre 2 y (n-1). El algoritmo termina si alguna de las divisiones tiene como resto cero.

```
#!/usr/bin/env python3

n = int(input("Introduzca un número entero: "))

for d in range(2, n):
    if n % d == 0:
        break;

if d < n - 1:
    print(n, "es divisible entre", d)
else:
    print(n, "es primo\n")
```


continue I

Descripción

- Similar a *break*, pero no finaliza el bucle.
- *continue* transfiere la ejecución al punto justo antes del final del bucle.
- *continue* fuerza una nueva iteración, abandonando la actual.

break y continue I

Enter the MATRIX

¿Cuántas veces se muestra en pantalla la palabra MATRIX?

```
#!/usr/bin/env python3

x = 0;
for i in range(10):
    for j in range(10):
        print("MATRIX")
        x = x + 1
        if (j == 4):
            break
print("MATRIX se ha escrito", x, "veces")
```


break y continue I

Enter the MATRIX... again

¿Cuántas veces se muestra en pantalla la palabra MATRIX?

```
#!/usr/bin/env python3

x = 0;
for i in range(10):
    for j in range(10):
        if (j == 4):
            continue
        print("MATRIX")
        x = x + 1
print("MATRIX se ha escrito", x, "veces")
```

Bibliografía I

Para saber más...


González Duque, R.
Python para todos.
http://mundogeek.net/tutorial-python/


Lutz, M.
Learning Python.
Ed. O'Reilly, 4ª ed. 2009.


Python Programming Language. Official Site.
http://python.org/